1. Az algoritmus és a program fogalma, jellemzői. Az algoritmus-tervezés helye és szerepe a szoftverfejlesztésben. Algoritmusok építő elemei. Algoritmuslépések és programutasítások kapcsolata. Programvezérlési szerkezetek egy választott programozási nyelvben.

1.1 Az algoritmus és a program fogalma, jellemzői

Algoritmus: egy feladat megoldását eredményező véges számú, egyértelműszabályokkal megfogalmazható lépések sorozata.

Jellemzői:

1) véges számú lépésekből (elemi tevékenységekből, instrukciókból, utasításokból) áll
2) minden lépésnek egyértelműen végrehajthatónak kell lennie (a végrehajtó egység. minden lépés után eldönti, mi lesz a következő lépés)
3) hivatkozhatunk összetett lépésekre is.(külön megadhatjuk)
4) a végrehajtandó instrukciónak valamilyen célja van. (végrehajtás során valamilyen változás következik be. Általában megváltoznak az adatok értékei)
5) általában vannak bemenő (input) adatai, melyeket felhasznál.
6) legalább 1 kimenő (output) adatot eredményeznie kell.
7) használhat ciklusokat, rekurziót a tömörség végett
8) általános legyen, ne csak egyetlen adatra használható
9) nem függ programnyelvtől, platformtól
10) elronthatatlannak kell lennie

Program:: Az algoritmus megfogalmazása a számítógépek, vagy a fordító programok számára érthető nyelven., ennek elemi lépéseit nevezzük utasításnak

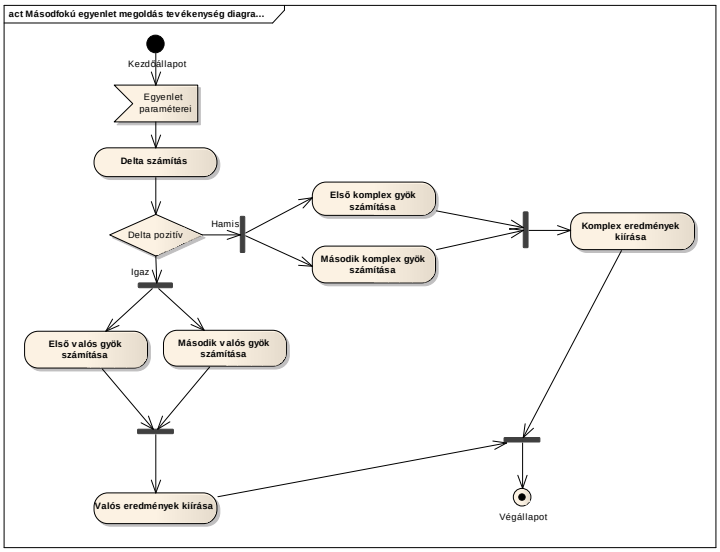
Jellemzői:

1) összetett lépésekből (algoritmus) áll.
2) az utasítás. végrehajtásának mindig van tárgya. Ezeket a tárgyakat a programozásban adatoknak nevezzük.
3) mindig van célja.
4) felhasználóbarátnak kell lennie

1.2 Algoritmus tervezés helye és szerepe.

Az algoritmus kigondolása után, grafikus megjelenítési formát ölt a tevékenységdiagramban. Itt mindenek előtt ügyelni kell az átláthatóságra. A következő fázis lehet a pszeudokód.

Tevékenység diagram: algoritmus leírására szolgáló grafikus jelölésrendszer. Segítségével a program dinamikus viselkedését tudjuk ábrázolni. A probléma megoldásának a lépéseit szemlélteti, a párhuzamosan zajló vezérlési folyamatokkal együtt.



1.3 Az algoritmusok építő elemei:

Vezérlőszerkezetek:

- **Szekvencia:** lépések, utasítások egymás utáni végrehajtása
- **Elágazás (szelekció):** az algoritmust egy feltételtől függően tovább folytatjuk
- **Ciklus (iteráció):**ugyanazt a lépést többször hajtjuk végre(határozott lépésszámú, elől és hátul tesztelő)
- **Feltétel nélküli ugrás:** vezérlés átadása az algoritmus egy megadott pontjára

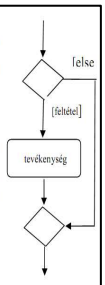
A csak szekvencia, szelekció, iterációból építkező programokat strukturált programnak nevezzük.

Vezérlési struktúrák

Szekvencia: egymás utáni tevékenységek sorozata. A szekvenciát a nyilak irányában felsorolt egymás utáni tevékenységek alkotják.


```
Tevékenység1
Tevékenység2
...
```

Szelekció: programelágazást jelent, egy adott ponton a tevékenységek végrehajtása feltételektől függ. Szelekció esetén a tevékenységből vagy az elágazási pontból kifelé vezető nyilak mindegyikén szerepel egy feltétel. Ha igaz egy adott feltétel, akkor a feltétel vonalának irányában halad tovább a vezérlés, és a mutatott tevékenység hajtódik végre. Megadható egy „egyébként" ág(else), ha a többi feltétel nem teljesül, akkor erre az ágra kerül a vezérlés.



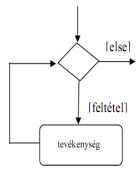

```
if feltétel
    tevékenység
end if
```

```
if feltétel1
    tevékenység1
else if feltétel2
    tevékenység2
end if
```

Iterációk: bizonyos tevékenységek ismételt végrehajtása. Az ismétlendő tevékenységet ciklusmagnak nevezzük. Iteráció esetén a vezérlés ismételten visszatér a ciklusmag elé, de az újbóli végrehajtás előtt egy elágazási pont található, mely megengedi a ciklus elhagyását. Ha a vezérlés nem tudja elhagyni a ciklust, végtelen cilkusról van szó. Két fajtája létezik:

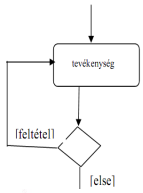
- *Elöl tesztelős*: a cilkusmag előtt tesztel. A ciklusmag akkor kerül végrehajtásra, ha a feltétel teljesül. Elképzelhető, hogy a cilkusmag sohasem hajtódik végre!!!!
- *Hátul tesztelős*: a ciklusmag után vizsgálja a feltételt, és a ciklusmag egyszer mindenképpen végrehajtódik! („do while" utasítás).

Elöl tesztelős ciklus:




```
while feltétel
  tevékenység
end while
```

Hátul tesztelős ciklus:



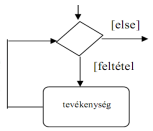

```
do
  tevékenység
end do while feltétel
```

(a „while" szó jelentése: amíg)

Léptető (növekményes) ciklus: a ciklusmag minden egyes végrehajtásakor egy ún. cilkusváltozó automatikusan lép egyet (megváltozik, növekszik, vagy csökken). „FOR" ciklusnak is szokták hívni.

```
for (cvált=kezd; cvált<=vég; cvált=cvált+lépésköz)
    tevékenység
end for
```

A tevékenység a ciklusváltozó kezdő és végértéke között minden olyan értékre végrehajtódik, amelyik a léptetési utasítás hatására előáll.



Vátozók: olyan memória területek, amelyek értékeket vehetnek fel.
Típus: Minden változónak jól meghatározott típusa van. A változó csak a típusának megfelelő értéket vehet fel.
Értékadás: Az algoritmus változóit deklarálni kell, megadni neveiket és tipusaikat.

1.3 Algoritmuslépések és a programutasítások kapcsolata.

Az algoritmusainkat szekvenciákból, elágazásokból, ill. iterációból építjük fel. Ha program utasításait 1egyszerűen egymás után kell végrehajtanunk, akkor szekvenciát írunk. Előfordulhat, hogy egy feltételtől függ az, hogy egy utasítást végre kell-e hajtanunk, vagy hogy. melyik utasítást kell végrehajtanunk. Ekkor alkalmazzuk a szelekciót (elágazás). Valamint elképzelhető, hogy egy tevékenységet (programutasítás sorozatot) többször kell ismételnünk, ekkor iterációt, vagy ciklust alkalmazunk.

1.4 Programvezérlési szerkezetek egy kiválasztott pogramozási nyelvben (MQL4)

Különbség a C nyelvtől, hogy nincs do-while ciklus és goto utasítás.

Szelekció:

- *Egyágú* – ha egy adott feltétel igaz, akkor végrehajtjuk a hozzá tartozó tevékenységet, egyébként pedig kikerüljük.

```
if(feltétel)
        utasítás;
```

- *Kétágú* – ha igaz egy adott feltétel, akkor a hozzá kapcsolódó tevékenységet hajtjuk, egyébként a másikat.

```
if(feltétel)
        utasítás;
else
        utasítás;
```

- Többágú – A feltételek közül legfeljebb egy teljesülhet. (if – else kombinációk; switch) *switch* – akkor alkalmazható, ha egy kifejezés jól meghatározott, különálló értékeire szeretnénk bizonyos utasításokat végrehajtani. (Pl. menü)

```
switch (kifejezés) {
        case érték1: utasítások; break;
        case érték1: utasítások; break;
        ...
        default: utasítások;
```

Iteráció:

- *while:* Elöltesztelő ciklus, a ciklusba való belépési feltételt fogalmazzuk meg, ha teljesül a ciklusmag végrehajtódik.

```
while(feltétel){
        utasítás 1
        utasitás 2
}
```

- Léptető ciklus for – egy vagy több ciklusváltozó szerepelhet, melyek minden egyes végrehajtáskor automatikusan lépnek egyet. (Pl. több elemeinek kiolvasása)

```
for (inicializálás; feltétel, léptetés){
        utasítás 1;
        utasítás 2;
}
```

- Ciklusból való kiugrás, illetve a ciklusmag átlépése *break, continue*

break – Az aktuális utasításblokkból való azonnali kiugrást eredményez.

```
for (int i=0; i<10; i++) {
  for (int j=0; j<20; j++) {
    if (ok)
      break;              // kiugrás a belső ciklusból
    ...
  }
  System.out.println("Vége a belső ciklusnak");
}
System.out.println("Vége a külső ciklusnak");
```

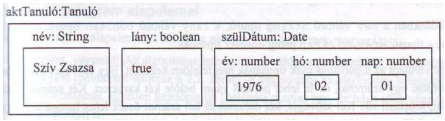
continue – A vezérlés az utasításblokk végére kerül. A ciklus folytatódik, csak az aktuális ciklusmag ettől kezdve nem kerül végrehajtásra.

```
for (int i=0; i<10; i++) {
  for (int j=0; j<20; j++) {
    if (ok)
      continue; // folytatás a ciklusváltozó következő értékével
    ...
  }
  System.out.println("Vége a belső ciklusnak");
}
System.out.println("Vége a külső ciklusnak");
```

2. A típus és a változó fogalma. Egyszerű és összetett adattípusok. Adatok láthatósága az objektumokban. Közvetlen és közvetett hivatkozású (referencia/dinamikus) változók. Az SQL adattípusai.

2.1 A típus és a változó fogalma

Típus: a változó tulajdonsága,amely meghatározzaa a lefoglalt memóriaterület nagyságát, van 1 byte-os, 2 byte-os, 4 byte-os, stb. másrészt meghatározza, hogy az adatot hogyan lehet kezelni, egész számként, valós számként, karakterkódként, stb. harmadrészt meghatározza, hogy milyen műveletek végezhetők az adattal.Vannak beépített (standard) típusok, valamint mi is létrehozhatunk újabbakat. az új típusok definiálása a -type - paranccsal történik. Minden változónak van egy jól meghatározott típusa. A változó csak a típusának megfelelı értékeket vehet föl.
• Number (szám).egyszerő (primitív) adat. Pl. 45, 99.9, 10000.
• Boolean (logikai) egyszerő (primitív) adat. értéke true (igaz) vagy false (hamis).
• String (szöveg) összetett típus. pl. Szív Zsazsa, Egri Kata.
• Date (dátum), összetett típus (év, hó, nap)pl: 1978,06,20
Változó: olyan memóriaterületek, amelyek különbözı értéket vehetnek fel. jellemzői: az azonosítója, adott méretı memóriahelye, típusa és aktuális értéke.
Deklarálás: Az algoritmus változóit deklarálni kell, meg kell adni neveiket és típusaikat.
Inicializálás: amikor egy változó definiálásával egyidejűleg értéket is adunk a változónak.



a program által használt összes változót deklarálni kell, vagyis meg kell adni a nevét és a típusát! Elıször megadunk egy típust, aztán felsorolunk egy vagy több azonosítót, vesszıvel elválasztva. `double fejadag, osszesen.` Amikor a változónak kezdeti, induló értéket adunk, akkor a változót inicializáljuk! `double fejadag=0.4, osszesen;` Konstans: Ez egy megváltoztathatatlan változó.

2.2 Egyszerű és összetett adattípusok

Egyszerő (primitív) adattípusok: Memóriaterülete oszthatatlan. az egy „élettelen”, viselkedés nélküli tulajdonságot tárol.

- *Egész típusok:*

Típus neve:	Foglalt memória:	Legkisebb érték:	Legnagyobb érték:
byte	1 bájt	- 128	127
short	2 bájt	- 32.768	32.767
int	4 bájt	- 2147483648	2147483647
long	8 bájt	- 10^{19}	~ 10^{19}

- *Valós típusok (lebegőpontos típusok):*

A pontosság itt azt jelenti, hogy a program maximum hány számjegyig terjedő tizedest tárol. Kétféle valós típus van:

Típus neve:	Foglalt memória	Legkisebb érték:	Legnagyobb érték:	Pontosság:
Float	4 bájt	1.401298464324817070e -45	3.402823466385288860e +38	6 -7 jegy
Short	8 bájt	4.940656458412465544e -324	1.797693134862315700e +308	14 -15 jegy

Megjegyzés: a java.math csomagban lévő *BigInteger* és *BigDecimal* osztályokkal tetszőleges pontosság érhető el!

- *Karakter típus:*

Típus neve:	Foglalt memória:	Legkisebb érték:	Legnagyobb érték:
char	2 bájt	'\u0000' (0)	'\u0000' (65536)

- *Logikai típusú:*

Egy logikai típusú változónak csak a *true*, vagy *false* értéket adhatjuk.

Típus neve:	Foglalt memória:	Igaz érték:	Hamis érték:
boolean	1 bájt	true	false

Összetett (referencia) adattipusok: Olyan mutató, mely egy objektum hivatkozását tartalmazza. Az objektum összetett memóriaterület, több primitív típusú változót, valamint további referenciákat is tartalmazhat. Tömb, Interfész, Osztály

2.3 Adatok láthatósága az objektumokban

Láthatóság: Az osztály deklarációi a következı láthatósággal (hozzáférési móddal) rendelkezhetnek:
- Nyilvános (public): minden kapcsolatban álló kliens eléri és használhatja, UML jelölése +
- Védett (protected): hozzáférés csak öröklésen keresztül lehetséges, UML jelölése #
- Privát (private):az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá, UML jelölése -

Az objektum a valós világ egy egyedét reprezentálja, attribútumokból és módszerekből áll, egyértelműen azonosítható, rendelkezik a bezárás, elrejtés, üzenetküldés, öröklés és polimorfizmus tulajdonságokkal. → Az azonos felépítésű objektumok egy osztályba tartoznak. Minden művelet, amit egy objektumon végre kell hajtani, egy metódust képvisel. Minden objektumnak van egy állapota, amelyet az attribútumok pillanatnyi értéke határozz meg. Az objektumok szerkezetüket tekintve zártak, azaz az attributumok és a törzsét alkotó utasítások közvetlenül nem érhetők el egyetlen másik objektumból sem, de egy objektum egy másik objektum metódusait üzenetküldéssel aktivizálhatja.
Bezárás: A bezárás az adatok és metódusok összezárását, egységbezárását jelenti. (egységbezárás elve!). ily módon az objektum zárt és sérthetetlen lesz, tehát bizonyos metódusok meghívásával nem tud elromolni az objektum egysége. Más szóval információkat rejtünk el a kliens elıl, amelyeket csak az interfészen keresztül lehet megközelíteni. Fontosabb szabályok:
• az objektumnak van egy interfésze, amely a programozó által kijelölt metódusok összessége.
• Az objektumot csak az interfészen lehet megközelíteni.
• Az adatok csak metódusokon keresztül érhetık el
• Az objektum interfész része a lehetı legkisebb

2.4 Közvetlen és közvetett hivatkozású (referencia/dinamikus) változók

Az Objektum Orientált programozásban alapvetően két típus létezik: primitív és referencia típus.
• Primitív típus: egy primitív típusú változó azonosítójával közvetlenül hivatkozhatunk a változó memóriahelyére. Ezt a helyet a rendszer a deklaráció utasítás végrehajtásakor foglalja le. A programozó nemdefiniálhat primitív típust.
• Referenciatípus: a referencia típusú változók objektumokra mutatnak. Egy referencia típusú változó azonosítójával az objektum memóriahelyére közvetve hivatkozhatunk egy referencián (hivatkozáson) keresztül.

2.5 Az SQL adattípusai.

Az 1. szabvány SQL nyelv elemei:
Alapelemek:
- Az SQL táblákat (relációkat kezel). Elnevezésük: table. A tábla azonosítója (neve) betővel kezdıdik, hossza pedig legfeljebb az operációs.
- A tábla attribútumait, oszlopoknak nevezzük, amelyek oszlopazonosítóját, típusát és hosszát a tábla definiálásakor adjuk meg.

- A táblák, névvel ellátott együttese, az adatbázis (database).

Táblafajták

- Adattábla: a valódi relációk, melyek a karbantartott adatokat tartalmazzák (A-tábla).
- Eredménytábla: adat- vagy eredménytáblából lekérdezéssel nyert tábla, elmenthetı (E-tábla).
- Nézettábla. Virtuális tábla, amelyrıl csak a definíciója tárolódik (V-tábla)
- Index: automatikus a használata, tehát csak a létrehozásáról kell gondoskodni (I-tábla).
- Szinonima: létezı adat- vagy nézettábla átnevezése (S-tábla).

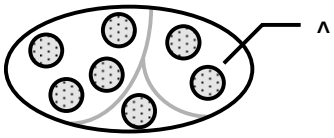
- SMALLINT: 6 számjegyű egész szám (az előjelet is beleszámítva): -12 345 – 123 456
- INTEGER: 11 számjegyű egész szám (az előjelet is beleszámítva): -1 234 567 890 – 12 345 678 901
- DECIMAL(x,y) x számjegyű, fixpontos decimális szám, y tizedesjeggyel: x= 1-19, y=0-18
- NUMERIC(x,y Előjellel és a tizedes jellel együtt x számjegyű, fixpontos decimális szám, y tizedesjeggyel: x=1-20, y=0-18
- FLOAT(x,y): Előjellel és a tizedes jellel együtt x számjegyű lebegőpontos szám, y tizedesjeggyel: x=1-20, y=0-18
- CHAR(x): karakterlánc(string): x=1-254
- DATE: dátum
- LOGICAL: logikai érték, t az igaz érték(true), n a hamis érték (false)

Az SQL befogadó tipusú (host) nyelv, azaz más nyelvbe való beépülésre készült, igy az SQL nyelvben nem lehet változókat definiálni, csak a behívó nyelv változóit (vendégváltozók- host-variable)kezeli.

3.Adatszerkezetek (tömb, verem, sor, lista, kollekció-keretrendszer, tábla, gráf, fa). Létrehozásuk, feldolgozásuk, bejárásuk, adattárolás lehetséges módszerei, indexelés).

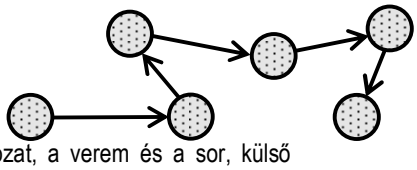
Az **adatszerkezet** egymással kapcsolatban álló adatok, objektumok összessége. Feladata az adatok hatékony, szervezett tárolása és kezelése. A kapcsolatokban részt vevő struktúraelemeket csomóponti adatoknak nevezzük. A csomópontokat körökkel, azok közötti kapcsolatokat pedig nyilakkal ábrázoljuk. Az adatmodell működését az objektumok közötti relációk határozzák meg. A kapcsolatok alapján az adatszerkezetek négy fő csoportba sorolhatók:

Asszociatív adatszerkezetek



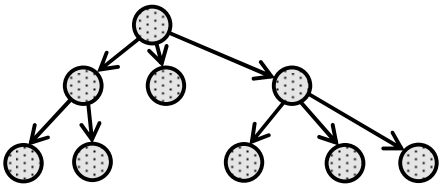
A struktúraelemek közötti kapcsolatokat az elemek azonos tulajdonságértékei létesítik. Az elemeket e tulajdonságok alapján csoportosítjuk. A kapcsolatok az asszociatív (csoportosítható) adatszerkezetekben a leglazábbak. Asszociatív adatszerkezet memóriában a tömb, a ritka a mátrix és a különböző táblák, külső tárolóeszközön pedig a direkt szervezésű állomány.

Szekvenciális adatszerkezetek



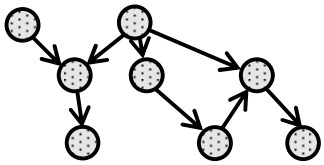
Az egyes struktúraelemek egymás után helyezkednek el. Mindig van egy kezdő elem, és minden elemet a struktúra egy jól meghatározott eleme követ. A kapcsolat egy-egy jellegű: minden elem csak egy helyről látható, és minden elem csak egy elemet lát. A memóriában megvalósítható szekvenciális adatszerkezetek a jelsorozat, a verem és a sor, külső tárolóeszközön ilyenek a szekvenciális és láncolt adatállományok.

Hierarchikus adatszerkezetek



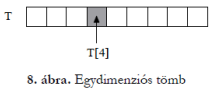
A struktúraelemek hierarchikusan egymás alá vannak rendelve. A kapcsolatok jellege egy-sok: minden csomópont csak egy helyről látható, egy csomópontból viszont sok csomópont látható. A hierarchikus adatszerkezeteket a belső tárban fának, a külső tárolókon hierarchiaállománynak nevezik.

Hálós adatszerkezetek

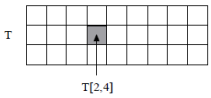


Hálós adatszerkezetek esetén bármelyik csomópont bármelyik csomóponttal kapcsolatban állhat. A kapcsolatok sok-sok típusúak. A hálós adatszerkezeteket a belső tárban irányított gráfnak, ill. hálózatnak, a külső tárolókon sémának nevezik.

Tömb:



8. ábra. Egydimenziós tömb



9. ábra. Kétdimenziós tömb

A tömb egy **asszociatív** adatszarkezet**,** lényege, hogy elemeit indexeken keresztül érjük el. A tömb használatának hátránya, hogy méretét előre meg kell adni és az indexhatárok dimenziónként kötöttek. Az *egydimenziós* tömb egy adatsor (egy index) (8. ábra), a *kétdimenziós* egy téglalapalakú adattáblázat egy sor és egy oszlopindex) (9. ábra). A tömbelemek típusa bármilyen lehet, akár összetett is. A tömb adatcsoport jellemző kezelési módja az elemenkénti feldolgozás, tehát a műveletekben a tömbelemek az operandusok.

Tömbök tárolása: *Sorfolytonosan:* ez lehetővé teszi, hogy bármely elem helyét az indexek ismeretében kiszámíthassuk, s így közvetlenül elérhessük őket. A tömb méretétől függetlenül bár-melyik elemét ugyanannyi idő alatt érhetjük el. Elemei indexeléssel közvetlenül címezhetők

Tömb megszüntetése: ***Java***-ban tömb által lefoglalt memóriaterületet csak az automatikus szemétgyűjtő algoritmus szabadít fel. Kikényszerítés: System.gc(); Destruktor egyébként minden programnyelvben van, kivéve a ***Java***-ban. Tehát a tömb megszüntetéséről nekünk kell gondoskodni.

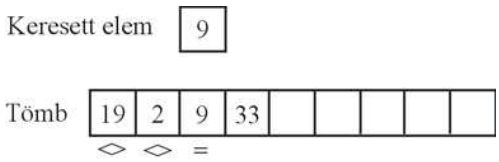
Tömb deklarálása:

- Deklaráláskor meg kell adnunk a tömb:
  - nevét
  - elemeinek típusát
  - indextartományát
  - dimenzióját.
- Egy dimenziós tömb: vektor, sorozat.
- Két dimenziós tömb: mátrix, táblázat.

Tömbök feldolgozása :

- A tömbök feldolgozása gyakran jelenti ugyanazon műveletek ismétlését az egyes elemekre, ez pedig tipikusan valamilyen ciklus alkalmazásához vezet.
- Különösen a léptető (for) ciklus kötődik szorosan a tömbökhöz, mert a ciklusváltozó jól használható a tömbelemek egymás utáni megcímzésére.

Keresés tömbben: Egyenként meg kell vizsgálnunk a tömb elemeit. A vizsgálat akkor érhet véget, ha megtaláltuk az értéket, vagy már nincs több vizsgálandó elem. A vizsgálatot a leggyakoribb és legegyszerűbb módon, növekvő indexsorrendben végezzük (11. ábra).



Ritka Mátrix:

- Olyan többdimenziós tömb, ahol a tömb nagy része kihasználatlan
- Tárolás inkább láncolt listában ajánlott.

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	3	0	0	0
2	0	21	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	99	0	0	0	0	7	0
5	0	0	0	0	0	0	0	0	0

Tábla:

A tábla egy **asszociatív** adatszerkezet, melynek elemei kulcs és adat párok, ahol a kulcsok egyediek, és bármely elem a kulcsán keresztül érhető el. A táblával kapcsolatos műveletek központi kérdése az adott kulcshoz tartozó adatok minél rövidebb idő alatt történő megkeresése, és a tábla karbantartása. A tábla tárolása vektorban és listában is megvalósítható. A táblára érvényes szabályok két csoportra bonthatók:

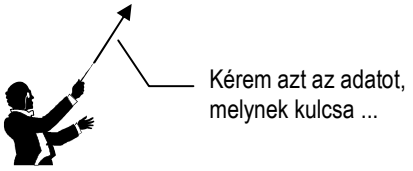
• A **logikai szabályok** minden táblára igazak fizikai megvalósítástól függetlenül (pl. ugyanaz a kulcs nem vehető fel kétszer).

• A **fizikai szabályok** az adott tábla fizikai megvalósításának korlátai (pl. nem vihető fel több adat, ha a tábla betelt).

Műveletek: keresés, beszúrás, törlés és szekvenciális(folyamatos, sorrendi) elérés

- Asszociatív adatszerkezet
- Egy elem: egyedi kulcs + adat
- Tárolás: egydimenziós tömbben vagy láncolt listában

adat	adat	adat	adat	adat	adat	adat
kulcs	kulcs	kulcs	kulcs	kulcs	kulcs	kulcs



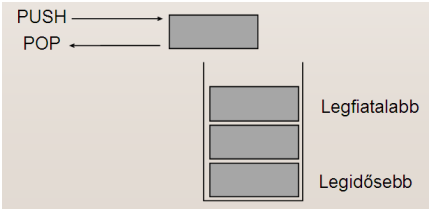
Oszlopok **különböző típusúak** tudnak lenni (pl. dátum, egész, tört, adatfolyam, ujjlenyomat képe, titkosított jelszó).

Verem:

A verem (stack) egy **szekvenciális** adatszerkezet, melynek mindig csak a legutoljára betett elemét lehet látni, illetve kivenni (LIFO). Kell egy mutató, mely a mindenkori verem tetejére illetve az első szabad helyre mutat. A veremről meg kell tudni állapítani, hogy az üres, vagy tele van, hiszen üres veremből nincs értelme kivenni, tele verembe pedig nincs értelme betenni elemet.

Műveletek:

• PUSH – elem betétele a verembe, mindig a tetejére
• POP – elem kivétele a veremből, mindig a legfelsőt
• TOP – a legfelső elem lekérdezése, a verem változatlan marad



Az elemek sorrendjét a legkönnyebben verem alkalmazásával fordíthatjuk meg. Ugyanígy nagyon jól használható a verem különböző visszatérési utak megjegyzésére is.
Tárolása általában egydimenziós tömbben vagy egy irányban láncolt listában történik.
A tömbnél a verem tetején a tömb legnagyobb indexű (utolsó) eleme van, az aktuális elemszám mint veremmutató (a verem tetejére mutató index) funkcionál

• Ráhelyezésnél a tömb aktuális elemszáma eggyel nő, az új elem az utolsó (legfelső) lesz.
• Levételnél az utolsó elemet vesszük le, az aktuális elemszám eggyel csökken

A listánál a verem tetején a lista kezdőeleme van, a kezdőmutató tölti be a veremmutató szerepét 39. ábra:

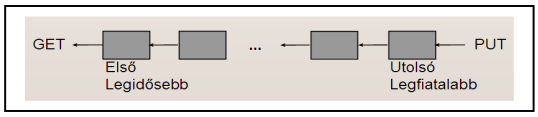
• Ráhelyezésnél, az új elem a lista elejére kerül.
• Levételnél az első elemet vesszük le.

Sor

A sor (queue) egy szekvenciális adatszerkezet, melyből mindig a legelsőnek betett elemet lehet kivenni (FIFO). Tárolása vektorban és listában egyaránt megvalósítható.

Műveletek:

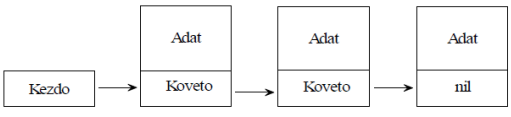
• PUT – elem betétele a sorba, mindig a sor végére
• GET – elem eltávolítása a sorból, mindig a sor elejéről
• FIRST – az első elem lekérdezése, a sor változatlan marad



Sorokkal tipikusan olyan feladatok oldhatók meg, melyekben az elemek feldolgozása érkezési sorrendben történik. Sorként működnek a pufferek, mint például a billentyűzet- vagy a nyomtatópuffer.

Lista

A láncolt listák **szekvenciális** adatszerkezetek, a sorrendi kapcsolatot (előző, követő) a lista elemeiben elhelyezett mutatók hozzák létre. A legegyszerűbb láncolt lista az egy irányban láncolt (vagy röviden egyirányú) lista (28. ábra).



28. ábra. Egy irányban láncolt lista

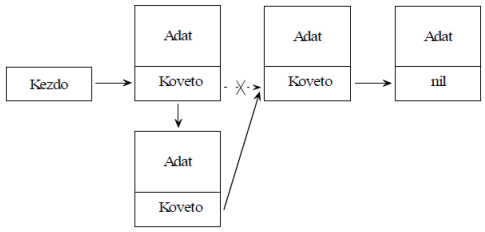
- Gazdaságos memóriafoglalás, egyszerű karbantartási műveletek (törlés, beszúrás).
- Szekvenciális adatszerkezet.
- *Listafej*: a lista első elemére mutat.
- *Végjel (null)*: speciális mutató érték, az utolsó elem mutatórészében állva jelzi a lánc végét.

További láncolt lista változatok:

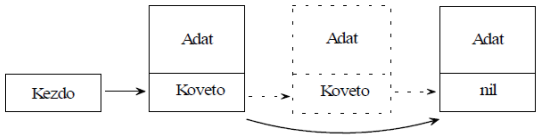
- rendezett láncolt lista: nem utólag rendezzük a listát, hanem az új elem felvitele a rendezettség megtartása mellett történik;
- ciklikusan láncolt lista: az utolsó elem mutatója az első elemre mutat;
- két irányban láncolt lista: egy listaelemben két mutató;
- többszörösen láncolt lista: több láncolat mentén is bejárható, azaz több szempont szerint is rendezett, egy elemben több mutató.

A láncolt listáknál nincs indexe az elemnek, a hozzáférés alapvetően *soros,* vagyis, ha egy elemet el akarunk érni, ahhoz végig kell járnunk a megelőző elemeket is.

A láncolt listák előnyös tulajdonságai leginkább a *módosító–karbantartó* jellegű feladatoknál mutatkoznak meg. Míg a tömböknél egy elem beszúrása vagy a törlése szükségszerűen az elem helyétől függő mennyiségű adatmozgatással jár (gondoljuk meg pl. hogy az első tömbelem törléséhez az összes többit „át kell pakolni"), addig a láncolt listáknál a beszúrás és a törlés az *elem helyétől függetlenül,* bármely elemre nézve is csak *néhány műveletet* igényel, hiszen egy-két mutató átírásával megvalósítható (29. és 30. ábra).



29. ábra. Beszúrás egy irányban láncolt listába



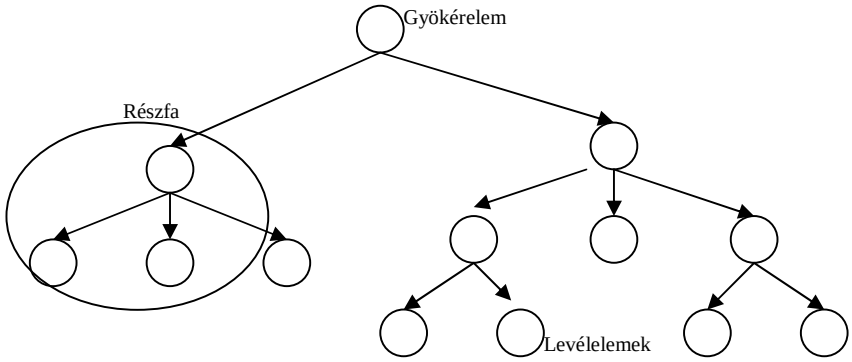
30. ábra. Elem törlése egy irányban láncolt listából

Fa

A fa egy **hierarchikus** adatszerkezet, amelyben egy elemnek akárhány rákövetkezője, de minden elemnek csak egyetlen megelőzője létezik. A fa egy olyan dinamikus, homogén adatszerkezet, amelyben minden elem megmondja a rákövetkezőjét.
Elnevezések:

- **Gyökérelem**: a fa azon eleme, amelynek nincs megelőzője. A legegyszerűbb fa egyetlen gyökérből áll. Mindig csak egy gyökérelem van, de az kötelezően, kivétel az üres fa ahol egy sem.
- **Levélelemek**: a fa azon elemei, amelyeknek nincs rákövetkezőjük.
- **Közbenső elem**: a fa nem gyökér, illetve levél elemei, hanem az összes többi. Megelőző eleme és rákövetkező elemei is vannak.
- **Út**: az út egy olyan szekvenciális (folyamatos,sorrendi) adatelem sorozat, lista, amely a gyökérelemtől kiinduló, különböző szinteken átmenő, és levélelemben véget érő egymáshoz kapcsolódó él sorozat. Az út hosszán az adott útban található élek számát értjük. Minden levélelem a gyökérelemtől kiindulva pontosan egy úton érhető el
- **szülő**: ha A csomópontból él mutat B csomópontba, akkor A szülője B-nek;
- **gyerek**: ha A csomópontból él mutat B csomópontba, akkor B gyereke A-nak;
- **testvér:** egy szülőhöz tartozó csomópontok.

A fákkal kapcsolatban beszélhetünk **szint**ekről, egy **elem szintje** megegyezik a gyökérelemtől vett távolságával. A 0. szinten a gyökérelem van, az első szinten a gyökérelem rákövetkezői, stb. A maximális szintszámot a **fa magasságának vagy mélységének** nevezzük. Minden közbenső elem egy részfa gyökereként tekinthető, így a fa részfákra bontható.



***bináris* fa**: minden csomópontnak,elemnek max. 2 gyereke van.
Szigorúan bináris fa: a levélelemek kivételé-vel minden csomópontnak pontosan két gyereke van.

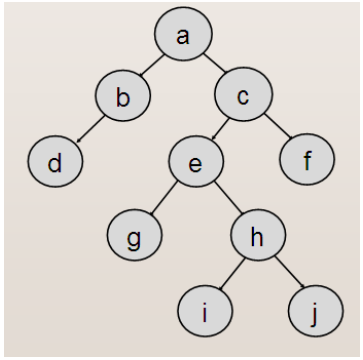
A bináris fa megvalósítása:

- Tárbeli megvalósítása a láncolt listában alkalmazott adatszerkezet bővítésével: minden elemnek két rákövetkezője lehet, ezért két mutatót alkalmazunk a csomópontokban, az egyik a baloldali, a másik a jobboldali részfa gyökerére mutat.
- Ha valamely mutató értéke a VégJel (null), akkor ebben az irányban nincs folytatása a fának.
- A levélelemek mindkét mutatója VégJel.

Műveletek: nincs indexelés

- Lekérdező műveletek:
  - Üres-e a fa struktúra
  - Gyökérelem lekérdezése
  - Meghatározott elem megkeresése, az arra vonatkozó referencia visszaadása
- Módosító műveletek:
  - Üres fa létrehozása _ konstruktor
  - Új elem beszúrása
  - Meghatározott elem kitörlése
  - Összes elem törlése
  - Egy részfa törlése
  - Részfák kicserélése egymással
  - Gyökér megváltoztatása
- Fák bejárása. Bejárásnak nevezzük azt a folyamatot, amikor a fa minden elemét pontosan egyszer érintve feldolgozzuk.
  - *preorder* (Gyökérkezdő) bejárás: a b d c e g h i j f
  - *postorder* (Gyökérvégző) bejárás: b c a , d b e f c a , d b g h e f c a , d b i j g h e f c a

Bináris fa bejárása:

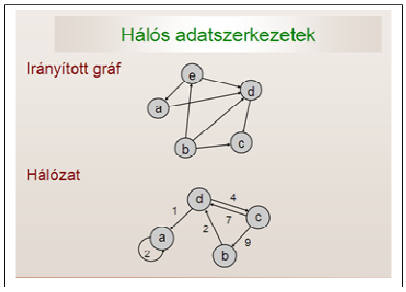


preorder (Gyökérkezdő) bejárás: a b d c e g h i j f

- gyökérelem feldolgozása;
- baloldali részfa preorder bejárása;
- jobboldali részfa preorder bejárása;

- *inorder* (Gyökérközepű) bejárás: d b a g e i h j c f
  - baloldali részfa inorder bejárása;
  - gyökérelem feldolgozása;
  - jobboldali részfa inorder bejárása;
- *postorder* (Gyökérvégző) bejárás: d b g i j h e f c a
  - baloldali részfa postorder bejárása;
  - jobboldali részfa postorder bejárása;
  - gyökérelem feldolgozása;

Gráf:



A gráf egy hálós adatszerkezet, adatelemei között a kapcsolat sok-sok jellegű: bármelyik adatelemre több helyről is eljuthatunk, és bármelyik adatelemtől elvileg több irányban is mehetünk tovább.
A gráf egy köröket is tartalmazó fa, nincs kitüntetett gyökéreleme, tárolása többszörösen láncolt listában.
bejárása:

- vak keresés
  - szélességben először keresés
  - mélységben először keresés (nem teljes, nem optimális)
  - mélységben először keresés, mélységi korláttal (nem teljes)
  - mélységben először keresés, fokozatosan növelt mélységi korláttal
- heurisztikus keresés (valamilyen szempont szerint osztályozzuk az egyes csomópontokat és éleket)

a már meglátogatott elemeket (körök, hurkok) nem járjuk be újra.

<u>Kollekció keretrendszer</u> (Collections Framework)

A kollekciók (konténerek) olyan objektumok, melyek célja egy vagy több típusba tartozó objektumok memóriában történő összefoglaló jellegű tárolása, manipulálása és lekérdezése. Olyan, mint a könyvespolc vagy a ruhásszekrény. Mindkettő képes objektumok (könyvek, ruhák) tárolására, azonban jelentős különbség van közöttük például az elemek elérésében. Egy szépen elrendezett könyvespolcról bármelyik könyvet gyorsan ki lehet emelni, ráadásul ha abc sorrendben vannak felrakva a könyvek, még a keresés is elég gyors. Egy megfelelően mély ruhásszekrény abban különbözik a könyvespolctól, hogy abból csak a ***legutoljára*** berakott ruha-oszlopot tudjuk kivenni, a mögötte levőket nem. Ha a legbelső oszlopra van szükségünk, akkor ki kell venni az összes előtte levőt, hogy elérhessük az utolsó elemet.
Az egyféle objektum tárolására kihegyezett - kollekciókat ***típusos kollekcióknak*** nevezzük. A típusosság előnye az, hogy az adatok berakásakor típusellenőrzés történik, így nem köt ki a zokni a könyvespolcon. Az adatok kiolvasásánál is biztosak lehetünk a kapott elem típusában, így a ruhásszekrénybe nyúlva nem ragadunk meg egy szakácskönyvet.
A nem típusos (általános) kollekciókból kiolvasott elemek használatához a kiolvasott típust legtöbbször vissza kell kasztolni (típus-konvertálni) a tényleges típusra.

A kollekció keretrendszer egy egységes architektúra, ami a kollekciók megvalósítására, kezelésére szolgál.
Elemei:
- Interfészek: absztrakt adattípusok, amelyek a kollekciókat reprezentálják. Lehetővé teszik a kollekciók implementáció független kezelését.
- Implementációk: a kollekció interfészek konkrét implementációi.
- algoritmusok: azok a metódusok, amelyek hasznos műveleteket valósítanak meg, mint például keresés, rendezés különböző kollekciókon.

Előnyei:
- Minimálisra csökkenti a programozás ráfordítást.
- Nem a „csináld magad” szemlélet az uralkodó.
- Növeli a kód minőségét, és a sebességét.
- Elősegíti a kód újrafelhasználást.

Egy adatszerkezet megválasztásánál, minősítésénél számítástechnikai szempontból három fő tényező veendő tekintetbe:
• A *tárkihasználás* és az operatív tárban való *tárolhatóság.*
  számításigényes feladatoknál kiemelt fontosságú, hiszen a háttértár használata nagyságrendekkel lassítja a számításokat.
• A *karbantarthatóság* vagyis a változások átvezetésének műveletigénye.
  fontossága attól függ milyen gyakoriak a változások.
• A *lekérdezhetőség* vagyis az információ kinyerés műveletigénye.

4. Az adatmodell alapelemei. Adatmodell típusok és jellemzőik. A relációs adatmodell fogalma, kulcsok kategóriái, kapcsolatok felállítása. Az adatmodellek és a szakterületi modellek kapcsolata, összefüggése.

4.1 Az adatmodell alapelemei

Adatmodell: Az adatmodell a valós világ számítógépes leképzésére kidolgozott koncepció, amely modellezési alapelemek, integritási kényszerek, megszorítások és az alapelemekkel végezhető műveletek együttese. *A modellezési alapelemek biztosítják az adatok tárolását. Az alapelemek szerkezetének kialakítását az adatok közötti összefüggésnek és kapcsolatoknak megfelelően lehet kialakítani. Az adatmodellek olyan feltételeket is előírnak, melyeket az adatbázisban tárolt adatoknak mindenkor ki kell elégíteni. Ezeket nevezzük megszorításoknak. A megszorítások közül kifejezetten fontosak azok, melyeket az adatok egyértelmű visszakereshetősége és a kapcsolatok hibátlan tükrözése érdekében írunk elő – ezek az integritási kényszerek.* Az adatmodell Egy séma, melyben megadjuk mely tulajdonságok határozzák meg az egyedeket, mely egyedek szerepelnek a sémában, és ezek közt milyen kapcsolatok vannak.

Alapelemek:

Egyed: Egyednek hívunk minden olyan tetszőleges dolgot , objektumot, ami minden más dologtól (objektumtól) megkülönböztethető és amiről adatokat tárolunk. Pl. dolgozó, autó, stb.

Tulajdonság: Az egyedeket tulajdonságokkal (attribútumokkal) írjuk le. A tulajdonság az egyed egy jellemzője, ami megadja, meghatározza az egyed egy részletét. Pl. a dolgozó egyednek tulajdonsága lehet: név, fizetés, stb.

Kapcsolat: Kapcsolatnak nevezzük az egyedek közötti viszonyt. A kapcsolat mindig valóságos objektumok közötti viszonyt fejez ki.

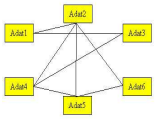
Az 1-1 típusú kapcsolat (1:1): Az egyik egyedhalmaz mindegyik eleméhez a másik egyedhalmaznak pontosan 1 eleme kapcsolódik.

Az 1-több típusú kapcsolat (1:N): Az „A" egyedhalmaz mindegyik eleméhez a B egyedhalmaznak több eleme is tartozik. Pl. VEVő : RENDELÉS. 1 vevőhöz több rendelés is tartozhat, míg fordítva nem igaz: 1 rendelés kizárólag 1 vevőtől jön.

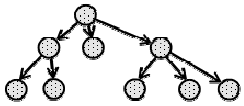
A több-több típusú kapcsolat (N:M): Az „A" egyedhalmaz minden eleméhez a B egyedhalmaz több eleme tartozhat, és fordítva. Pl. TERMÉK : ALKATRÉSZ Ha végiggondoljuk, h. 1 termék több alkotóból állhat, és 1 alkatrész is több terméknek lehet az alkotója.

4.2 Adatmodell típusok és jellemzőik

Három adatmodell létezik az alapelemek fizikai tárolásától függően.

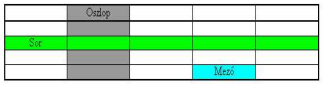


A **hálós modellben** gráffal ábrázolható az adatbázis: a gráf csúcsain az egyed-előfordulások, az éleken a köztük lévő kapcsolat helyezkedik el. Tetszőleges két csomópont között, akkor van kapcsolat, ha őket él köti össze. Egy csomópontból tetszőleges számú él indulhat ke, de egy él csak két csomópontot köthet össze. Azaz minden elem tetszőleges számú másik elemmel lehet összefüggésben, 1:n és n:m tipusú adatkapcsolatok is megvalósíthatóak.



A **Hierarchikus adatmodell** az elemek fölé és alárendeltségi viszonyban vannak, úgynevezett fa elrendezést (struktúrát) valósítanakmeg. A fa csomópontjaiban és leveleiben helyezkednek el az adatok. A csomópontok az egyedeket, míg a kapcsolatokat az élek reprezentálják A csomópontok között levő kapcsolat, szülő gyermek kapcsolatnak felelmeg. Így csak 1:n típusú kapcsolatok képezhetők le segítségével. Az 1:n kapcsolat azt jelenti, hogy az adatszerkezet egyik típusú egyede a hierarchiában alatta elhelyezkedő egy vagy több más egyeddel áll kapcsolatban.

Relációs adatmodell (asszociatív)– A relációs adatszerkezetben az egyedek közötti kapcsolatokat az egyedek azonos tulajdonság értékei valósítjákmeg. Az egyedeket a tulajdonságok alapján csoportosítjuk. A relációt (reláció dolgok viszonyát jelenti) táblázat formájában adjuk meg. A táblázat oszlopait a tulajdonságok (attribútumok) neveivel azonosítjuk, melyeknek a reláción belül egyedieknek kell lenniük. A reláció soraiban tárolódnak a logikailag összetartozó adatok az úgynevezett egyed előfordulások. Az előfordulások sorrendje közömbös, de 2 teljesen azonos sor (rekord), nem lehet a táblázatban. A sor és oszlop metszésében található elemet mezőnek nevezzük, a mezők tartalmazzák a tulajdonságok értékeit. A mezők oszloponként különböző típusú (numerikus, szöveges stb.) mennyiségek lehetnek a tulajdonságot leginkább meghatározó érték alapján.



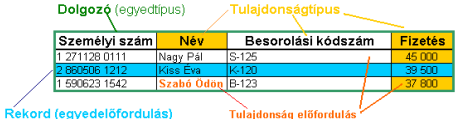
4.3 A relációs adatmodell fogalma, kulcsok kategóriái, kapcsolatok felállítása.

A korai modellekkel szemben a relációs modellt egy magas szintő lekérdező nyelv támogatja, amely nemcsak a lekérdezést, hanem az adatdefiniálást és az adatmanipulálást is ellátja. Ez a nyelv az SQL.

Relációs adatbázisok(oszlopokba szedett adatok összegsége)

A **reláció** az adatelemek megnevezett, összetartozó csoportjából kialakított olyan kétdimenziós táblázat, amelyik sorokból és oszlopokból áll, és ahol az oszlopok egy-egy tulajdonságot írtnak le, a sorok adják az egyedhalmaz/reláció/táblázat egyedeit. Ahhoz, hogy egy táblázatot relációnak lehessen tekinteni, a következı feltételeket kell kielégítenie:

- nem lehet két egyforma sora,
- minden oszlopnak egyedi neve van,
- a sorok és oszlopok sorrendje tetszőleges.



A relációs adatbázisok általában nem 1, hanem több, logikailag összekapcsolható relációból (táblázatból) állnak. A reláció oszlopainak (attribútumainak) számát a reláció fokszámának, sorainak számát a reláció kardinalitásának nevezik.

Kulcsok kategóriái:

Elsődleges kulcsnak a reláció azon kulcsát nevezzük, amely egyértelműen azonosítja a táblázat egy sorát.

DOLGOZÓ tábla – AZONOSÍTÓ a kulcsa a relációnak

AZONOSÍTÓ	NÉV	FIZETÉS
001	Nagy	100000
002	Kiss	110000

PRÉMIUM tábla – AZONOSÍTÓ oszlop külső kulcs a táblában

AZONOSÍTÓ	ÖSSZEG	DÁTUM
001	45000	99.01.12
002	50000	99.01.13

Külső kulcsnak vagy idegen kulcsnak nevezzük egy relációnak azokat az attribútumait, amelyek egy másik relációban kulcsot alkotnak.

A kulcsok csoportosítása:

Egyszerő kulcs: egyetlen attribútumból áll. A reláció kulcs a reláció egy sorát azonosítja egyértelműen. A reláció - definíció szerint- nem tartalmazhat két azonos sort, ezért minden relációban létezik kulcs. A reláció kulcsnak a következő feltételeket kell teljesítenie

- az attribútumok egy olyan csoportja, melyek csak egy sort azonosítanak (egyértelműség)
- a kulcsban szereplő attribútumok egyetlen részhalmaza sem alkot kulcsot

Diák

Személyi szám	Születési év	Név

3.7 ábra Reláció kulcs

- a kulcsban szereplő attribútumok értéke nem lehet definiálatlan (NULL)

SZEMÉLY_ADATOK={{ SZEMÉLYI_SZÁM, SZÜL_ÉV, NÉV}). A SZEMÉLYI_ADATOK relációban a SZEMÉLYI_SZÁM attribútum kulcs, mert nem lehet az adatok között 2 kül.ző személy azonos személyi számmal. A születési év vagy a név nem azonosítja egyértelműen a reláció egy sorát mivel ugyanazon a napon is született tanulók vagy azonos nevűek is lehetnek az osztályban.

Összetett kulcs: Előfordulnak olyan relációk is, melyekben a kulcs több attribútum érték összekapcsolásával állítható elő. Készítsünk nyilvántartást a diákok különböző tantárgyakból szerzett osztályzatairól az alábbi relációval: NAPLÓ={{SZEMÉLYI_SZÁM, TANTÁRGY, DÁTUM, OSZTÁLYZAT)}

Napló

Személyi szám	Tantárgy	Dátum	Osztályzat

A NAPLÓ relációban a SZEMÉLYI_SZÁM nem azonosít egy sort, mivel egy diáknak több osztályzata is lehet akár ugyanabból a tantárgyból is. Ezért még a SZEMÉLYI_SZÁM és a TANTÁRGY sem alkot kulcsot. A SZEMÉLYI_SZÁM, TANTÁRGY és a DÁTUM is csak akkor alkot kulcsot, ha kizárjuk annak lehetőségét, hogy ugyanazon a napon ugyanabból a tantárgyból egy diák 2 osztályzatot kaphat. Abban az esetben, ha ez a feltételezés nem tartható (ennek a rendszer analiziséből kell kiderülnie!), akkor nem csak az osztályzat megszerzésének dátumát, hanem annak időpontját is tárolni kell. Ilyenkor természetesen a NAPLÓ relációt ezzel az új oszloppal ki kell bővíteni.

Kapcsolatok felállítása:

Egy – egy típusú kapcsolat: Az egyik egyedhalmaz mindegyik eleméhez a másik egyedhalmaz pontosan egy eleme kapcsolódik.

Egy – több típusú kapcsolat: Az egyik egyedhalmaz mindegyik eleméhez a másik egyedhalmaz több eleme is kapcsolódik.

Több– több típusú kapcsolat: Az A egyedhalmaz minden eleméhez a B egyedhalmaz tobb eleme tartozhat és fordítva egy B egyedhalmaz beli elemhez is több A egyedhalmaz beli elem is tartozhat. Pl. házak szinek – egy ház lehet több szinű is, és egy szin lehet több házon is.

4.4 Az adatmodellek és a szakterületi modellek kapcsolata, összefüggése.

Szakterületi modellen a szakterület objektumai és kapcsolatait tüntetjük fel. Objektumok az egyedek.

A Szakterületi modell az a közös nyelv, amelynek segítségével a megrendelői és fejlesztői oldal kommunikációja egyértelmű.

A Szakterületi modell letisztult tudás a szoftver alkalmazási területéről.

A Szakterületi modell és az implementáció közvetlen kapcsolatban van.

A Szakterületi modellt a szakterületi szakértő és a szoftverfejlesztő közösen hozzák létre.

Döntően, tábla mellett!

A Szakterületi modell objektummodell, UML nyelven megfogalmazva. A Szakterületi modell a szakterületi tudást olyan formában tükrözi vissza, amely érthető és implementálható a szoftverfejlesztő számára. A Szakterületi modell a tudást, a hagyományos, szabad szöveges leírásnál formalizáltabban, zártabb, szigorúbb szabályok szerint írja le.

A modell az „üzlet” által használt fogalmakat, ezek kapcsolatait és hozzájuk kötődő szabályokat írja le. Tehát természetes módon, érthető a megrendelői oldal számára.

A modell kellően formalizált, precíz ahhoz, hogy ennek alapján szoftver működhessen.

A modell direkt módon meghatározza az implementációt, ezért a fejlesztő képes a fejlesztés során felvetődő kérdéseit a megrendelő nyelvén feltenni.

5. Rutin, metódus, eljárás és függvény fogalma, jellemzőik. Paraméterátadás. Példány és osztálymetódusok. Eseménykezelő metódusok. Függvények az SQL-ben

5.1 Rutin, metódus, eljárás és függvény fogalma, jellemzőik

Rutin: A strukturált programozás alapfogalma, egy programból bizonyos szempontok alapján elkülönített forráskód részlet, melynek nevet adunk s ezzel a névvel hivatkozunk rá a későbbiekben.
Metódus: A objektum orientált programozás alapfogalma, utasításokok (tevékenységek) összessége, amelyet meghívhatunk a metódus nevére való hivatkozással. Olyan rutin, mely objektum adatain dolgozik. Az objektum-orientált programozásban az objektumokat le lehet írni úgy, mint egy tudáshalmaz (a benne tárolt adatok), illetve az azokon végzett műveletek (az objektumhoz tartozó rutinok) összessége, ezek lesznek a metódusok. A két fogalom között részhalmaz viszony, reláció van, azaz minden metódus rutin, de csak az a rutin metódus, ami objektumhoz van kötve.
Egy rutin attól függően, hogy van e visszatérítési értéke vagy nincs , eljárásnak vagy függvénynek szokás nevezni.
Tehát sima procedurális programozásnál mondjuk van egy 'rajzol' rutinunk, objektum orientált programozásnál vannak objektumaink, amelyeknek meg lehet hívni a 'rajzol' metódusát. (Procedurális programozásról beszélünk, ha a programozási feladat megoldását egymástól többé kevésbé független alprogramokból (procedure) építjük fel.)
Eljárás: nincsen visszatérési értéke (void = semleges). Utasítások összessége, melyet egyszerően végrehajtunk az eljárás nevére való hivatkozással. Végrehajtás után a progi azzal az utasítással folytatódik, amelyik követi a metódushívó utasítást Pl: System.out.println („ezt a println eljárás írta ki"); a System osztály println eljárását használtuk, erre hivatkoztunk.
Függvény: szintén utasításokok összessége, melyet a nevére való hivatkozással hajtunk végre, de értéket ad vissza, melyet a függvény neve képvisel. A visszatérési érték típusa a függvény visszatérési típusával egyezikmeg.

5.2 Paraméterátadás

A szubrutinok a paramétereken keresztül képesek kommunikálni az őt meghívóval. Egy szubrutin nem láthatja az őt hívó szubrutin változóit, hiszen azok hatásköre csak saját utasításblokkjukra korlátozódik(vagy olyan globális változókkal, amelyek mindkettőjük számára látható). Az eljárások, függvények fejlécében felsorolt paramétereket FORMÁLIS paramétereknek nevezzük. Azokat a paramétereket pedig, amelyekkel az eljárást, vagy függvényt meghívjuk, AKTUÁLIS paramétereknek nevezzük. A formális és aktuális paraméterek darabszámának egyeznie kell, és páronként típusának kompatibilisnek kell lennie.

```
CheckTime(StartTime,EndTime)
                          <-- Actuális paraméter
                          <-- Formális paraméter
bool CheckTime(int starttime,int endtime)
{
// Ha a bármely változó értéke nagyobb mint 23, akkor mehet
if(starttime>23 || endtime>23)
   return(true);
```

A Paraméterátadás fajtái: Legtöbb programozási nyelvben (C, C++, Java, stb.) csak az úgynevezett értékszerinti paraméterátadás létezik, míg más nyelvekben (pl. pascal, visual basic) létezik címszerinti paraméterátadás is.
Értékszerinti paraméterátadás:I az aktuális paraméterek tetszőleges kifejezések lehetnek és értékeiket rendre (első az elsőnek, második a másodiknak) átadják a formális paramétereknek. A hívott függvény nem képes megváltoztatni az aktuális paraméterek értékeit, hiszen számára azok nem láthatóak. Megoldható viszont, hogy a hívott megváltoztasson mégis egy hívóbeli értéket (azaz eredményt adjon vissza) a mutatók vagy a referenciák használatával. (A mutató egy olyan változó, amely egy memóriacímet képes tárolni, valamint képes az adott címen lévő adatot manipulálni.) Ha a formális paraméter egy mutató és a hívó ennek a mutatónak, egy változójának a címét adja át, akkor a hívott képes megváltoztatni a hívó egy változójának értékét. Ha a formális paraméter egy referencia változó, akkor képes felvenni egy változó referenciáját, azaz minden műveletben, amelyben szerepel, maga helyett a hivatkozott (akinek a referenciáját felvette) változó fog szerepelni, azaz úgy fog viselkedni, mintha a hivatkozott változónak egy másik neve lenne.
Címszerinti paraméterátadás:
Egyes nyelvek (pl. pascal, visual basic) ismerik a címszerinti paraméterátadást is. Ennél a paraméterátadásnál a formális paraméter számára ugyanaz a memória terület lesz kijelölve, mint amelyet az párjaként szereplő aktuális paraméter használ. Így bármilyen műveletet végzünk a formális paraméterrel az kihatással van az aktuális paraméterre, azaz ha az előállított eredményt egy címszerint átadott paraméterbe pakoljuk, akkor azt a hívó is megkapja az aktuális paraméteren keresztül. A szubrutin lefutásakor a címszerint átadott formális paraméter területe nem szabadul fel, csak az azonosító szűnik meg.

5.3 Példány és osztálymetódusokok

Egy osztály metódusának a végrehajtását, úgy kérhetjük, hogy a metódus nevére hivatkozunk. A metódus neve után a „( )" karakterpár szerepel, benne az esetleges paraméterekkel. A metódus neve előtt megadható egy osztály, vagy egy objektum neve, attól függően, hogy mit akarunk megszólítani:
• Osztálymetódus hívása: Osztály.metódus (paraméterek)
• Példánymetódus hívása: objektum.metódus (paraméterek)
Ha a metódust a saját osztályából hívjuk, akkor nem kell minősíteni, csak a metódus nevét kell leírni: metódus (paraméterek).
Osztálymetódus: Vannak adatok, amelyek nem egy konkrét példányra, hanem az egész osztályra jellemzőek, ezek az osztályadatok, osztályváltozók. Tehát az osztályváltozó értéke az osztály összes példányára ugyanaz, vagyis felesleges példányonként eltárolni. Az osztálymetódus az osztályváltozókat manipulálja, a példányokat nem éri el, tehát objektumok nélkül is tud dolgozni Az osztály-szintű metódusok hasonlóan az osztály-szintű mezőkhöz static kulcsszóval vannak megjelölve. Az osztályszintű metódusok akkor is meghívhatóak, ha az adott, tartalmazó osztályból egyetlen példányt sem készítettek, ezért csak osztályszintű mezőkre, és konstansokra szabad hivatkozni.

```
class TFoiskola
{
    public static void FelevMegnyitasa( int tanev, int felev)
    {
      Console.WriteLine("A foiskolan a {0}. év {1}. tanévét megnyitom.",
                        tanev,felev);
      OrarendekTorlese();
      IndexekLezarasa();
      GazdasagiEsemenyekNullazasa();
    }
}
```

A metódust meghivása az osztálynév.metódusnév formában történhet:

```
public static void Main()
{
 TFoiskola.FelevMegnyitasa( 2006, 2 );
}
```

Példánymetódus: Értelemszerűen a példányváltozókon dolgozik. Minden példányban megtalálhatjuk az osztály leírásában szereplö összes adatot, mivel azok példányonként más-más értéket vehetnek fel. Azonban metódusokat nem kell minden példányban tárolni, ezt elegendő egyszer az osztályban letárolni. A példányszintű metódusok meghívásához példányra van szükség, a metódushívás szintaktikája példánynév.metódusnév.

5.4 Eseménykezelő metódusok

valamilyen esemény (Event) hatására (kattintás, görgetés, tabulátor, stb.) bekötvekező metódusok.

Alacsony szintő esemény – az operációs rendszer szintjén történő elemi esemény, amely forrása csak komponens lehet. (Komponens-, Konténer-, Fókusz-, Ablak-, Billentyőzet-, Egér-esemény). Pl. megváltozott a komponens mérete; fókuszba került a komponens; egy komponenst hozzáadtak a konténerhez; becsuktak egy ablakot; lenyomtak egy billentyő vagy egérgombot.

Magas szintő esemény – ez általában logikai esemény (a progi állítja össze), forrása nem feltétlenül komponens. Magas szintő események lehetnek**: ActionEven**t – AbstractButton leszármazottja lehet a forrása**; ItemEvent** – kiválasztás esemény (1 tétel kiválasztása +változott); **AdjustmentEvent** – igazítási esemény 1 gördítı-sávon; **ListSelectionEvent** – listakiválasztás esemény.

Pl. Java JButton esemény – lenyomtam 1 gombot.

```
public void actionPerformed(ActionEvent ev){
System.out.println(„lenyomtak egy gombot”);
}
```

Esemény – lenyomtam a gombot.

Eseményforrás – maga a gomb, az esemény itt keletkezik, elızetesen felfőztük az esemény figyelıre.

Eseményfigyelő – az eseményt figyelő és lekezelő objektum, az osztálynak implementálni kell a figyelő interfészt – ez esetben az ActionListner-t. Az osztályban létezik az eseményt kezelı metódus (actionPerformed(ActionEvent ev)), a paraméter az eseményforrás azonosítója.

5.5 Függvények az SQL-ben

A különféle SQL megvlósulások sok függvényt tartalmaznak. AZ SQL általában kezeli a behívó nyelv függvényeit. A függvény szerkezet: függvénynév(argumentumok)

Van néhány függvény, amely alapértelmezés szerint be van építve az SQL-be, ezeket **beépített függvényeknek** nevezzük.

Beépített függvények :

Összesítő (aggregáló) függvények:

A lekérdezés eredményeként előálló táblák egyes oszlopaiban lévő értékeken végrehajthatunk bizonyos összesítő műveleteket, amelyek egyetlen értéket állítanak elő.

- COUNT([DISTINCT] kifejezés):a kifejezés által meghatározot oszlopokban lévő rekordok száma.(DISTINCT esetén csak a különböző értékeket számlálja, az azonosokbó csak egyet vesz figyelembe)
- SUM([DISTINCT] aritmetikai kifejezés): az aritmetikai kifejezés által meghatározot oszlopértékek összege.
- AVG([DISTINCT] aritmetikai kifejezés): a kifejezés által meghatározot oszlopértékek átlaga.
- MAX([DISTINCT] kifejezés): a kifejezés által meghatározot oszlopokból a legnagyobb értékeket adja vissza.
- MIN([DISTINCT] kifejezés): a kifejezés által meghatározot oszlopokból a legkisebb értékeket adja vissza.

logiakai típusú vagy predikátum függvények

- BETWEEN : **`kif1 BETWEEN kif2 AND kif3`** Igaz értéket vesz fel, ha: kif2 <= kif1 <= kif3 (Tagadható is NOT BETWEEN)
- IN : **`oszlopnév IN (értéklista)`** Igaz, ha az oszlop értéke eleme a listának (valamelyikkel megegyezik).
- LIKE : **`oszlopnév LIKE érték`** Igaz, ha az oszlop értéke megegyezik a like utáni értékkel/hasonlít a like utáni értékre (ugyanis maszkolható: _ %). A % jel bármilyen karakter 0 vagy nagyobb hosszúságú sorozatát, az _ jel egyetlen bármilyen karaktert helyettesít.

6. A kifejezés fogalma. Kifejezések kiértékelése, a műveletek precedenciája. egy választott programozási nyelv aritmetikai, logikai és relációs műveletei. Kifejezések SQL-ben.

6.1 Kifejezés Kifejezések kiértékelése, a műveletek precedenciája

Kifejezés: Operandusokból (konstans, változó) és Operátorokból (műveletekből) áll. A kifejezésben szerepelhet egy vagy több Operandus, bármelyik Operandus lehet maga is egy kifejezés. (a+5 → „a" és „5" operandus, „+" operator) A kifejezés állhat egyetlen operandusból is, és bármelyik operandus lehet egy újabb kifejezés

Kifejezések lehetnek:

- Aritm.i: számtani alapműveletek
- Logikai: logikai alapműveletek
- Karaktéres: karaktereken, sztringeken végzett műveletek, nem minden proginyelv támogatja.

Operátor	Asszociativitás
() []	Balról jobbra
! - (előjelváltás) ++ -- ~	Jobbról balra
& | ^ << >>	Balról jobbra
* / %	Balról jobbra
+ -	Balról jobbra
< <= > >= == !=	Balról jobbra
||	Balról jobbra
&&	Balról jobbra
= += -= *= /= %= >>= <<= %= != ^=	Jobbról balra
, (vessző)	Balról jobbra

A **kifejezések kiértékelési sorrendjét** a zárójelek és az Operátorok határozzákmeg a következő szabályok szerint:

• Elsőként a zárójelben található kifejezések értékelődnek ki,
• Ezen belül előbb mindig a magasabb precedencia szintű művelet hajtódik végre,
• Az azonos precedencia szintű operátorok között a leírás sorrendisége dönt., akkor a művelet asszociativitásától függıen jobbról balra (🡐) vagy balról jobbra (🡒) történik a kiértékelés.

6.2 választott progiozási nyelv aritmetikai, logikai és relációs műveletei. (C , MQL4)

Az MQL4-ben a következő fajta műveletek találhatóak meg :

- Aritmetikai műveletek

Szimbólum	Művelet	Példa	Analógia
+	Értékek összeadása	x + 2	
-	Kivonás vagy előjelváltás	x - 3, y = - y	
*	Sorzás	3 * x	
/	Osztás	x / 5	
%	Az osztás maradéka	A perc = az idő % 60	
++	A változó értékének növelése 1-el	y++	y = y + 1
--	A változó értékének csökkentése 1-el	y--	y = y - 1

Értékadó műveletek

Szimbólum	Művelet	Példa	Analógia
=	Az y változó x értékét kapja	y = x	
+=	Az y változó növelése x-el	y += x	y = y + x
-=	Az y változó csökkentése x-el	y -= x	y = y - x
*=	Az y változó szorzása x-el	y *= x	y = y * x
/=	Az y változó osztása x-el	y /= x	y = y / x
%=	Az y változó x-el történő osztásának a maradéka	y %= x	y = y % x

- Relációs műveletek

Szimbólum	Művelet	Példa
==	Igaz, ha x egyenlő y	x == y
!=	Igaz, ha x nem egyenlő y	x != y
<	Igaz, ha x kevesebb mint y	x < y
>	Igaz, ha x kevesebb mint y	x > y
<=	Igaz, ha x egyenlő vagy kevesebb mint y	x <= y
>=	Igaz, ha x egyenlő vagy nagyobb mint y	x >= y

- Logikai műveletek

Szimbólum	Művelet	Példa	Magyarázat
!	NOT (NEM) (logikai tagadás)	! x	Igaz(1), ha az operandus értéke Hamis(0); Hamis(0) ha az operandus értéke nem Hamis(0)
||	OR (VAGY) (logikai választás)	x < 5 || x > 7	Igaz(1), ha bármelyik feltétel igaz
&&	AND (ÉS) (logikai összeadás)	x == 3 && y < 5	Igaz(1), ha minden feltétel igaz

- Bitenkénti műveletek

Bitenkénti műveleteket csak egész számokkal hajthatunk végre.

Szimbólum	Művelet	Példa	Magyarázat
~	Komplemens	x = ~ y	
>>	Jobbra shift	x = x >> y	Számjegyek elmozdítása jobbra,balról nulla be
<<	Balra shift	x = x << y	Számjegyek elmozdítása balra, jobbról nulla be
&	És művelet	a = x & y	Bitenkénti AND
|	Vagy művelet	a = x | y	Bitenkénti Or
^	Kizáró vagy	a = x ^ y	Bitenkénti XOR

Ha egy műveletben két nem egyforma típusú operátor találkozik, akkor implicit (automatikus) típuskonverzió történik. A művelet végrehajtása során az operandusok típusa módosul a legmagasabb prioritású operandus típusára. A típusok konverzió szerinti prioritása: string, double, int, datetime, color, bool.

```
string x = 2.0 / 5;
// Az eredmény egy tíz karakteres string:"0.40000000”.
```

6.3 Kifejezések SQL-ben

Az SQL a kifejezések szerkezete és tartalma tekintetében megegyezik más nyelvekkel.

- **Aritmetikai kifejezések**: Numerikus vagy dátum típusú oszlopnevekből, változókból, konstansokból, műveleti jelekből (+, -, *, /, **) és zárójelekből állnak. Szerepelhet bennük aritmetikai függvény is.
- **Karakter-kifejezések**: Karakter típusú oszlopnevekből, változókból, szöveg konstansokból, műveleti jelből (+, a konkatenáció jele, pl: „ab”+”lak”=”ablak”) és zárójelekből állnak. A szöveg konstansokat idézőjelek vagy aposztrófok közé tesszük.
- **Logikai kifejezések**: Logikai típusú oszlopnevekből, változókból, konstansokból, műveleti jelekből (AND/OR/NOT) és zárójelből áll. A logikai kifejezésekben szerepelhetnek a relációs operátorokok (< , > , =…).

7. Programozási tételek I. Elemi programozási tételek: sorozatszámítás, kiválasztás, eldöntés, lineáris keresés, megszámolás, maximum-kiválasztás (adatszerkezet nélkül, tömbbel, kollekciókkal, állományokkal).

7.1 Sorozatszámítás(összegzés): Az összegzés tétele összeadja vagy összeszorozza tömbben lévő számokat. Szorzásnál az S 1 kezdőértéket vesz fel.

X: A tömb.
N: A tömb elemszáma, egész.
S: Az eredmény, egész típus

```
S:=0
Ciklus I=1-től N-ig
S:=S+X(I)
Ciklus vége
Eljárás vége.
```

```
s=0;
for (i=1; i<=N; i++) {
    s=s+x[i];
}
```

7.2 A kiválasztás tétele: Ebben az esetben tudjuk, hogy létezik az adott tulajdonságú elem, a legelső ilyen elem sorszámát kapjuk eredményül.

X: A tömb.
N: A tömb elemszáma, egész.
T: keresendő tulajdonság.
S: Az eredmény, egész típus.

```
I:=1
Ciklus amíg T(X(I)) nem T tulajdonságú
I:=I+1
Ciklus vége
S:=I
Eljárás vége
```

```
i=1;
while (x[i] != T) {
      i=i+1;
}
s=i;
```

7.3 Az eldöntés tétele: Az eldöntés tétele megállapítja, hogy a tömbben van-e a feltételnek megfelelő elem.

X: A tömb.
N: A tömb elemszáma, egész
T: keresendő tulajdonság.
VAN: Az eredmény, logikai típus.

```
I:=1
Ciklus amíg I<=N és nem T(X(I))
I:=I+1
Ciklus vége
VAN:=(I<=N)
Eljárás vége....
```

```
i=1;
while ( (i<=N) and (x[i]!=T)) {
      i=i+1;
}
van=(I<=N);
```

7.4 Lineáris keresés: A lineáris keresésben megállapítjuk, hogy létezik-e a tömbben az adott tulajdonságú elem, ha igen, megadjuk az elsőt. A tétel az eldöntés és kiválasztás tétel együttese.

X: A tömb.
N: A tömb elemszáma, egész.
T: keresendő tulajdonság.
VAN: Logikai típus.
SORSZ: Az eredmény, egész típus.
Keresés(N,X,VAN,SORSZ):

```
I:=1
Ciklus amíg I<=N és nem T(X(I))
I:=I+1
Ciklus vége
VAN:=(I<=N)
Ha VAN akkor SORSZ:=I
Eljárás vége....
```

```
i=1;
while ( (i<=N) and (x[i]!=T)) {
      i=i+1;
}
van=(i<=N);
if (van==true) sorsz= i;
```

7.5 A megszámolás tételében megszámoljuk, hogy hány adott tulajdonságú elem van a tömbben.

X: A tömb.
N: A tömb elemszáma, egész.
DB: Az eredmény, egész típus.

```
DB:=0
Ciklus I=1-től N-ig
Ha T(X(I)) akkor DB:=DB+1
Ciklus vége
Eljárás vége....
```

```
DB=0;
for (i=1; i<=N; i++) {
   if (X[i]==T) DB++;
}
```

7.6 A maximumkiválasztás tételében megkeressük az adott tömbben a legnagyobb elemet. Eredményül a sorszámot és magát az értéket kaphatjuk. Hasonló feladat – csak a relációt kell megfordítani – a minimumkiválasztás.

X: A tömb.
N: A tömb elemszáma, egész.
MAXINDEX: maximális érték helye a sorozatban
MAXERT: Maximális érték.

```
MAXINDEX:=1:
MAXERT:=X(1)
Ciklus I=2-től N-ig
Ha MAXERT<X(I)
akkor MAX:=I : MAXERT:=X(I)
Ciklus vége
Eljárás vége....
```

```
MAXINDEX=1;
MAXERT=X[1];
for (i=2; i<=N) {
   if (MAXERT<X[i]) {
      MAXERT=X[i];
      MAXINDEX=i;
   }
}
```

A tömböket csak a tárgyalás egyszerűsége miatt alkalmaztuk, azok lehetnek azonos típusú rekordokból álló fájlok vagy listák is. Ennek megfelelően a programozási tételek megfogalmazhatók azonos hosszúságú rekordokból álló file-okra és listákra is. Ilyen esetekben az alábbi műveleteket kell kicserélni az algoritmusokban:

Tömb	File	Lista
Egy tömbelem vizsgálata	Egy rekord beolvasása egy változóba, majd a megfelelő mező vizsgálata	A listaelem beolvasása és a megfelelő mező vizsgálata
Iteráció egy ciklusban	Lépés a file következő rekordjára	Lépés a következő listaelemre
A tömb végének vizsgálata	End Of File vizsgálata	A listamutató 0-e vizsgálat
Egy tömb elem értékének az átírása	Ugrás az adott sorszámú rekordra, majd írás a rekordba, írás, után a rekordmutatót eggyel visszaállítjuk	A Listaelem kiírása
tömbindex	rekordsorszám	Egy számláló típusú érték, a listafej esetén 0 és minden iteráció esetén növekedik eggyel.

8. Programozási tételek II. Összetett programozási tételek: másolás, kiválogatás, szétválogatás, metszet, egyesítés, összefuttatás (tömbbel, kollekciókkal, halmazzal, állományokkal).

8.1 Másolás: Egy sorozathoz egy másik sorozatot rendelő feladattípusokhoz. Adott egy N elemű sorozat. A feladat ezen sorozat lemásolása, s közben egy (elemre vonatkozó) átalakítást lehet végezni. Az eredmény mindig ugyanannyi elemszámú. Bemenet: N egész * megadott sorozat elemeinek száma
A(): tömb (1..N) elemtípus [a sorozat elemei
Kimenet: B(): tömb (1..N) elemtípus * az új sorozat elemei

```
Ciklus i=1-től N-ig                                  for (i=1; i<=N; i++) {
    B(i):= A(i) //lehetséges művelet végzése A(i)-vel    B[i]=A[i]
Ciklus vége                                          }
Eljárás vége
```

8.2 Kiválogatás

a) Egy sorozathoz egy sorozatot rendelő feladattípusokhoz: Rendelkezésünkre áll egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Az a feladat, hogy a T tulajdonsággal rendelkező elemeket meghatározzuk egy másik sorozatban.
Bemenet: N egész * megadott sorozat elemeinek száma
A() tömb (1..N) elemtípus [a sorozat elemei]
T tulajdonság
Kimenet: DB : egész * T tulajdonságú elemek darabszáma
B() tömb (1..N) elemtípus [T tulajdonságú elemek]

```
DB:= 0
Ciklus i=1-től N-ig                                  for (i=1; i<=N; i++) {
    Ha A(i) T tulajdonságú akkor                         if (A[i]==T) {
    DB:=DB+1                                                DB++;
    B(DB):=A(i)                                             B[DB]=A[i];
Ciklus vége                                              }
Eljárás vége                                         }
```

b) Egy sorozathoz több sorozatot rendelő feladattípusokhoz: Rendelkezésünkre áll egy N elemű sorozat és egy, a sorozat elemein értelmezett T1, T2, …, Tx tulajdonságok. Az a feladat, hogy a különböző tulajdonságok szerint hozzunk létre új sorozatokat.
Bemenet: N egész * megadott sorozat elemeinek száma
A() tömb (1..N) elemtípus [a megadott sorozat elemei]
T1, T2, …, Tx (x darab tulajdonság)
Kimenet: DB1, DB2, …, DBx : egész * T… Tx tulajdonságú elemek darabszáma
B1(),B2(), …, Bx()tömb (1..N) elemtípus [T1.. Tx tulajdonságú elemek]

```
DB1:= 0                                              DB1=0;
DB2:= 0                                              DB2=0;
.
DBx:= 0;                                             DBx=0;
Ciklus i=1-től N-ig                                  for (i=1; i<=N; i++) {
    A(i) T1 tulajdonsága esetén                          if (A[i]==T1) {
        DB1:=DB1+1                                           DB1++;
        B1(DB1):=A(i)                                        B1[DB1]=A[i]; }
    A(i) T2 tulajdonsága esetén                          if (A[i]==T2) {
        DB2:=DB2+1                                           DB2++;
        B2(DB2):=A(i)                                        B2[DB2]=A[i]; }
    .
    A(i) Tx tulajdonsága esetén                          if (A[i]==Tx) {
        DBx:=DBx+1                                           DBx++;
        Bx(DBx):=A(i)                                        Bx[DBx]=A[i]; }
Ciklus vége                                          }
Eljárás vége
```

8.3 Szétválogatás: Egy sorozathoz két sorozatot rendelő feladattípusokhoz. Rendelkezésünkre áll egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Az a feladat, hogy a T tulajdonsággal rendelkező és a T tulajdonsággal nem rendelkező elemeket meghatározzuk egy-egy külön sorozatban.
Bemenet: N egész * megadott sorozat elemeinek száma
A(): tömb (1..N) elemtípus [a sorozat elemei]
T tulajdonság
Kimenet: DB1 egész * T tulajdonságú elemek darabszáma
DB2 egész * nem T tulajdonságú elemek darabszáma
B() tömb (1..N) elemtípus * T tulajdonságú elemek
C() tömb (1..N) elemtípus * nem T tulajdonságú elemek

```
DB1:= 0
DB2:= 0
Ciklus i=1-től N-ig
    Ha A(i) T tulajdonságú akkor
        DB1:=DB1+1
        B(DB1):=A(i)
    Különben
        DB2:=DB2+1
        C(DB2):=A(i)
Ciklus vége
Eljárás vége
```

```
DB1=0;
DB2=0;
for (i=1; i<=N; i++) {
    if (A[i]==T1) {
        DB1++;
        B1[DB1]=A[i]; }
    else {
        DB2++;
        B2[DB2]=A[i]; }
}
```

8.4 Metszet: Több sorozathoz egy sorozatot rendelő feladatokhoz. Rendelkezésünkre áll két sorozat. Az a feladat, hogy egy harmadik sorozatba azokat az elemeket válasszuk ki, amelyek mindkét sorozatban megtalálhatók.
Bemenet: Acount, Bcount egész * megadott sorozatok elemeinek száma
A(), B() tömb (1..N) elemtípus [a sorozatok elemei]
Kimenet: DB egész * metszet sorozat elemeinek darabszáma
C()tömb (1..N) elemtípus [metszet sorozat elemei]

```
DB:= 0
Ciklus i=1-től N-ig
    j:=1
    Ciklus amíg i<=M és A(i)<>B(j)
        j:=j+1
    Ciklus vége
    Ha j<=M akkor
    DB:=DB+1
    C(DB):=A(i)
Ciklus vége
Eljárás vége
```

```
DB=0;
for (i=1; i<=Acount; i++) {
    j=1;
    while (j<=Bcount && A[i] != B[j])) {
        j++;
        }
    if (j<=Bcount) {
    DB++;
    C[DB]=A[i]; }
}
```

8.5 Egyesítés (unió): Több sorozathoz egy sorozatot rendelő feladatokhoz. Rendelkezésünkre áll két sorozat. Az a feladat, hogy egy harmadik sorozatba azokat az elemeket válasszuk ki, amelyek legalább az egyik sorozatban megtalálhatók.
Bemenet: AN, BN egész * megadott sorozatok elemeinek száma
A(), B()tömb (1..N) elemtípus [a sorozatok elemei]
Kimenet: Csz egész * egyesített sorozat elemeinek darabszáma
C() tömb * egyesített sorozat elemei. Elemszáma az A és B tömb elemszámainak összege.

```
C():= A()
Csz:= AN+1
Ciklus Bsz=1-től BN-ig
Asz:=1
Ciklus amíg Asz<=AN és A(Asz)<>B(Bsz)
Asz:=Asz+1
Ciklus vége
Ha Asz>AN akkor
C(Csz):=B(Bsz)
Csz=Csz+1
Ciklus vége
Eljárás vége
```

```
for (i=1; i<=AN; i++) C[i]= A[i];
Csz=AN+1;
for (Bsz=1;Bsz<=BN; Bsz++) {
Asz=1;
    while (Asz<=AN && A[Asz] != B[Bsz]) {
    Asz++;
    }
if (Asz>AN) {
    C[Csz]:=B[Bsz];
    Csz++;
}
```

8.6 Összefuttatás tétele: Adott két rendezett sorozat, képezzük a sorozatok egyesítését! A feladat megoldását a közönséges únió feladatához képest most a sorozatok rendezettsége egyszerűsíti. A megoldás során sokkal kevesebb lépést kell tennünk, ha ezt a tulajdonságot kihasználjuk. A rendezettségből adódik, hogy mindig el tudjuk dönteni, hogy melyik sorozatból kell vennünk a következő elemet. Három lehetőségünk van:
a. A[Asz] < B[Bsz], akkor A[Asz]-et tesszük az egyesített sorozatba, és vesszük A-ből a következőt.
b. A[Asz] > B[Bsz], akkor B[Bsz]-et tesszük az egyesített sorozatba, és vesszük B-ból a következőt.
c. A[Asz] = B[Bsz], akkor mindegy, hogy A[Asz]-t, vagy B[Bsz]-t tesszük az egyesített sorozatba. Természetesen most mindkét sorozatból a következő(Asz++ és Bsz++) elemet kell választanunk. Ha az egyik sorozat végére értünk, akkor a megmaradó sorozat további elemeit kell bemásolnunk a C[]-be.

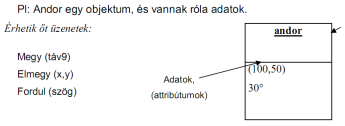
```
Asz:=1
Bsz:=1
Csz:=1
Ciklus amíg Asz<=AN és Bsz<=BN
    Ha A(Asz)<B(Bsz) akkor
            C(Csz)=A(Asz)
            Asz=Asz+1
            Csz=Csz+1
    Különben ha B(Bsz)<A(Asz) akkor
            C(Csz)=B(Bsz)
            Bsz=Bsz+1
            Csz=Csz+1
    Különben ha A(Asz)=B(Bsz) akkor
            C(Csz)=A(Asz)
            Asz=Asz+1
            Bsz=Bsz+1
            Csz=Csz+1
Ciklus vége
Ha Bsz<=BN akkor
    Ciklus amig Bsz<=BN
            C(Csz)=B(Bsz)
            Bsz=Bsz+1
            Csz=Csz+1
    Ciklus vége
Ha Asz<=AN akkor
    Ciklus amig Asz<=AN
            C(Csz)=A(Asz)
            Asz=Asz+1
            Csz=Csz+1
    Ciklus vége
Eljárás vége
```

9. Osztály és objektum fogalma. Egységbezárás. Osztály definiálása egy választott fejlesztő környezetben. Jellemzők (properties). Az osztálymodell kapcsolata az adatbázis-modellel.

9.1 Osztály és objektum fogalma

Objektum.: A hétköznapi életben az objektum egy tárgy, egy dolog, nincs ez másképp az OOP világban sem. Az objektumnak vannak adatai (tulajdonságai) és van valamilyen viselkedésmódja. Az objektum információt tárol, kérésre feladatokat hajt végre. Az objektum felel feladatainak korrekt elvégzéséért. Az objektum logikailag összetartozó adatok és rajtuk dolgozó algoritmusok (rutin, metódus, progikód) összessége.

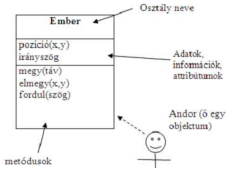
Egy Objektum Orientált program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a jól meghatározott feladatköre. Pl: adott 1 család: Laci, Erzsi, és 2 főzni tudó leány. Lacinak ebédre húslevest kell főznie." laci.elkészít (húsleves)" Üzenetküléskor a megszólított objektum és az üzenet neve közé „." teszünk. Az üzenetnek lehetnek paraméterei. Az objektumokat üzeneteken keresztül kérhetjük meg különböző feladatok elvégzésére. Az üzenet nem más, mint egy az objektumba beprogramozott rutin hívása. Az OO paradigmában a rutint metódusnak nevezzük.



Egy objektumnak vannak adatai, és metódusai:

- *Adatok: az objektum az információt adatok és attribútumok formájában tárolja.*
- *Metódusok: a metódus, olyan rutin, amely az objektum adatain dolgozik. Az objektumokat a feladatokra üzenetek által lehet megkérni. Egy üzenet hatására végrehajtásra kerül az objektumnak egy, az üzenettel azonos nevő metódusa, s ezáltal az objektum adatai megváltoznak. Az objektumnak mindig van valamilyen állapota (state) –ez megfelel az adatok pillanatnyi értékeinek. Egy feladat elvégzése után az objektum állapota megváltozhat. Az objektum mindig emlékszik az állapotára, tehát megjegyzi azt! Két azonos osztályhoz tartozó objjektumnak akkor és csak akkor ugyanaz az állapota, ha az adatok értékei rendre megegyeznek. Az objektumok egyértelműen azonosíthatók. Az objektum azonossága független a tárolt értékektől.*

Osztály: *Az osztályozás a természetes emberi gondolkodás szerves része. Az ugyanolyan adatokat tartalmazó és ugyanolyan viselkedés-leírással jellemezhető objektumokat egy osztályba soroljuk.*



Az osztály olyan objektumminta vagy típus, amelynek alapján példányokat (objektumokat) hozhatunk létre. Minden objektum egy jól meghatározott osztályhoz tartozik. Minden objektum egy osztály példánya.

Az osztályban definiáljuk:

- *az objektum adatait (hogy az objektumok milyen adatokat jegyeznek meg),*
- *az objektum által elvégzendő műveleteket (metódusokat). A metódus tulajdonképpen rutin (eljárás, függvény), mely az adott objektum adatain dolgozik. Az üzenet pedig egy rutin hívása.*

UML (Unified Modeling Language) egységesített modellező nyelv.

- *Az osztályt egy 3 részre osztott téglalappal jelöljük:*
  - *a felső részbe írjuk az osztály nevét középre, kiemelten;*
  - *a középsőbe kerülnek az osztály adatai; az adat nevét és kezdőértékét így adjuk meg: változónév:Típus=érték*
  - *a legalsóba tesszük az osztályban szereplő metódusokat; így:* metódusnév(paraméter:Típus,…):Típus
- *a példányt (objektumot)szintén téglalappal jelöljük (ez kettéosztott):*
  - *a példányba beírjuk annak azonosítóját és osztályát kettősponttal elválasztva, aláhúzva így: példány: Oszt. (az osztályt külön így jelöljük: :Oszt.)*
  - *szükség esetén megadhatjuk a második részben az objektum állapotát vagy bármit, amit fontosnak vélünk.*
  - *A példányt egy szaggatott nyíllal az oszthozhoz kötjük. A nyíl mindig az osztály felé irányul: a példány függ az osztálytól.*

Egy objektum születésekor annak osztálya egyértelműen meg van határozva. Az egyes objektumok üzenetek révén szólítják meg egymást. A feladatot kiadó objektumot kliensnek (aktor), a feladatot elvégzőt pedig szervernek (agent) hívjuk. Az objektumnak életciklusa van. (születés, élet, halál). Az objektumot létre kell hozni, majd azt követően inicializálni kell:

- *be kell állítani kezdeti adatait, // • végre kell hajtani azokat a tevékenységeket, amelyek az objektum működéséhez szükségesek.*

Az inicializálást végző metódust konstruktornak nevezzük. Pl: hozzuk létre az Ember osztályhoz tartozó Katit. Kati = new Ember(100,50)

Példányváltozó, példánymetódus: Az objektum (példány) mindig a saját adatain dolgozik, a metódusokat pedig az osztály leírásából nézi ki. A (példányokhoz tartozó) metódusokat (ezek az osztályban vannak) elegendő csak egyszer az osztályban tárolni, azok majd a megfelelő objektum adataitól (állapotától) függően fognak működni. A példányonként helyet foglaló változók a példányváltozók (példányadatok). Az osztály azon metódusait, amelyek a példányadatokon (példányváltozókon) dolgoznak, példánymetódusoknak nevezzük. (a példánymetódusok, az osztályban helyezkednek el.)

Osztályváltozó, osztálymetódus: Vannak adatok, amelyek nem egy konkrét példányra, hanem az egész osztályra jellemzıek. Az ilyen közös, osztályonként egyszer előforduló változót osztályváltozónak (osztályadatnak) nevezzük. Az osztályváltozó értéke az osztály összes példányára (objektumára) ugyanaz, teljesen felesleges lenne ezt az értéket példányonként eltárolni. Osztálymetódusnak nevezzük az olyan metódust, amely objektumok nélkül is tud dolgozni. Az osztálymetódus a példányadatokat nem éri el, csak az osztályváltozókat manipulálja. Az osztályváltozó, illetve osztálymetódus nevét az UML –ben aláhúzzuk!

9.2 Egységbezárás

Az objektum egyik legnagyobb ereje abban áll, hogy zárt és sérthetetlen, azaz az adatokat és a hozzájuk metódusokat összezárja és a kliens számára felesleges információkat elrejti. Ez úgy oldható meg, hogy csak bizonyos, a programozó által kijelölt metódusokat – amelyeket interfésznek hívunk – használhatja a kliens, és az adatokat csak ezeken keresztül érheti el. A kliens igy nem okozhat bajt, az objektumok állapotának egysége, logikája, konzisztenciája a kliens beavatkozása során is biztosított. Másképp: infókat rejtünk el a kliens elől, melyeket csak az interfészen keresztül lehet megközelíteni. Fontosabb szabályok:

- az objektumnak van egy interfésze, amely a programozó által kijelölt metódusok összessége.
- Az objektumot csak az interfészen keresztül lehet megközelíteni.
- Az adatok csak metódusokon keresztül érhetők el.
- Az objektum interfész része a lehető legkisebb

Kliens üzen a szervernek. egy objektum felkérhet más objektumokat különböző feladatok elvégzésére (üzenetküldés, felkérés). Szerepkörök: kliens – a feladatot elvégeztető objektum. (ügyfél, kérő, üzenetküldő); szerver – a feladatot elvégzı objektum (kiszolgáló, végrehajtó, választ adó, üzenetet fogadó).

Hozzáférési mód, láthatóság

- *Nyilvános (+): minden vele kapcsolatban álló kliens eléri és használhatja,*
- *Védett (#): hozzáférés csak az osztályból és annak leszármazottaiból lehetséges,*
- *Privát: (csak az osztály saját metódusai férhetnek hozzá).*

9.3 Osztály definiálása egy választott fejlesztő környezetben (java)

Az Object osztály: Javában minden osztály az Objektum osztályból származik. Mondhatjuk, bármely osztályból létrehozott példány az egy objektum, így azt minden olyan feladatra meg lehet kérni, mely már az objektum osztályban is definiálva van. Az objektum osztály olyan metódusokat tartalmaz, amelyek a Java minden objektumjára jellemzőek. Nézzünk meg néhányat:

• boolean equals (Object obj): összehasonlítja a megszólított objektumot a paraméterében megadott objektummal. A visszaadott érték true ha a 2 obj. egyenlő.

• String toString (): az obj. szöveges reprezentációját adja vissza.

• Class getClass (): visszaadja az objektum osztályát.

Egy **osztály definiálása** java-ban, egy fejből és egy blokkból áll. Az osztály fejét a class kulcsszó jelzi. Az osztály blokkja pedig az osztályra jellemző adatokat és metódusodeklarációkat foglalja magába.

Pl.

```
class OsztalyNeve{
        public String nev; // változók
        ....
        public OsztalNeve(<paraméterk>){
        } // konstruktor
        public metodusNev(<parameter>){
        } // metódus→eljárás
}
```

Objektum létrehozása, deklarálása (példányosítás): Egy osztály példányait a `new` (új) operátorral hozhatjuk létre. A new után meg kell adni a létrehozandó objektum osztályát, ezt a konstruktor aktuális paraméterlistája követi: `new <OsztályAzonosító>(<aktuális paraméterlista>)`

A new operátor feladatai:

• Létrehoz egy új OsztályAzonosító oszttályú objjektumot; lefoglalja számára a szükséges memóriát;

• Meghívja az osztálynak azt a konstruktorát, amelynek szignatúrájára ráhúzható az aktuális paraméterlista;

• Visszaadja az újdonsült objjektum referenciáját (azonosítóját). Az objektum osztálya az lesz, amit a new után megadtunk.

A konstruktor beállítja az objektum kezdeti állapotát (kezdetei értéket ad az adatoknak és esetleges kapcsolatoknak). A konstruktor neve megegyezik az osztály nevével. Egy osztálynak több konstruktora is lehetséges. Az objektum egész élete során a new után megadott osztályhoz fog tartozni, osztályát megváltoztatni nem lehet.

Minden referencia típusú változót (mint ahogyan a primitív típusúakat is) deklarálni kell! Deklaráláskor csak a referencia részére következik be tárfoglalás: `<Osztályazonosító> objektum;`

Az obj. létrehozásáról a programozónak kell gondoskodnia. A new operátor által visszaadott referencia, értékül adható a referencia típusú változónak:

```
objektum = new <OsztályAzonosító>(<aktuális paraméterlista>);
```

vagy

```
<OsztályAzonosító> objektum =  new <OsztályAzonosító> (<aktuális paraméterlista>);
```

Az objektumot a referencia típusú változón keresztül szólíthatjuk meg: objektum.metódus ().

Konstruktorok: Amikor Egy objektumotot a new operátorral létrehozunk, akkor azt inicializálni kell. A konstruktor beállítja az objktum kezdeti állapotát (kezdő értéket ad az adatainak és kapcsolatainak) beállítása, felépítése. A konstruktor neve megegyezik az osztály nevével. Egy osztálynak több konstruktora is lehet. Az objektum egész élete során a new után megadott osztályhoz fog tartozni, osztályát megváltoztatni nem lehet.

```
(<módosítók>) <OsztályAzonosító>((<formális paraméterlista>))
{
    <konstruktor blokkja>
}
```

A konstruktor általános szintakszisa:

A konstruktor hasonlít a metódushoz – a következő szabályok érvényesek rá:

• A konstruktor neve kötelezően megegyezik az osztály nevével.

• Csak a new operátorral hívható. Egy konstruktorral nem lehet újra inicializálni egy objektumot.

• A módosítók közül csak a hozzáférési (láthatósági) módosítók használhatók.

• A konstruktor túlterhelhető

• A konstruktornak nincs visszatérési értéke, és nem void.

• A konstruktor nem öröklődik.

Alapértelmezés szerinti konstruktor: Ha egy osztályban nem adunk meg explicit módon konstruktort, akkor az osztálynak lesz egy alapértelmezés szerinti (default), paraméter nélküli konstruktora. Ha tehát az osztályban nem adtak meg konstruktort, akkor a példány létrehozásakor a rendszer ezt az alapértelmezés szerinti konstruktort hívja meg. Az objektum adatai ekkor az alapértelmezések szerint lesznek beállítva. Ha az osztályban létezik egy akármilyen explicit konstruktor (akár paraméteres, akár paraméter nélküli), akkor az osztálynak nem lesz implicit, alapértelmezés szerinti konstruktora.

Konstruktor túlterhelése: *A konstruktorok ugyanúgy túlterhelhetık, mint más metódusok. Egy objektum így többféleképpen is inicializálható. Ha egy osztály több konstruktort definiál, akkor az egyik konstruktorból – annak első utsításaként – meghívható egy másik konstrukto,r a this ekkor eljárásként hajtódik végre: this (paraméterek)*

9.4 Jellemzők(Properties): Az obj. beállítható (setXY()) és lekérdezhetı (getXY()) tulajdonságát jellemzőnek nevezzünk(„property”).

9.4 Az osztálymodell kapcsolata az adatbázis-modellel.

Ha az osztálymodellünket jól terveztük meg, akkor a modellben lévő „entitás” (entity) jellegő osztályokok az adatbázismodellben egy-egy konkrét táblának felelnek majd meg. Ez az ún. objektum-relációs leképezés („O-R mapping”) amelynek egyik mai tipikus képviselıje a Javában lévő „EJB” technológia. Egy ilyen Java entitás-osztály főbb jellemzői:

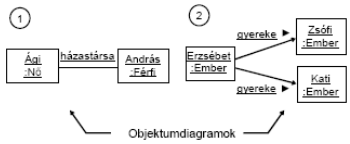
• Implementálja a „Serializable” interfészt.
• Kötelező egy elsődleges kulcs attribútum.
• A konkrét adattábla egy oszlopának (mezőjének) az entitás egy konkrét „tulajdonsága” (property) felel meg.

Az osztály és adatbázismodell kapcsolata, pedig inkabb UML-es megközelítést kíván. Ha készítesz egy alapvetően adatbázis használatára épülő rendszert, az osztályaid elnevezései és tulajdonságai nagy mértékben fognak hasonlítani a jól normalizált adatbázisod logikai adatmodellére. Persze e kettı nem lehet azonos. Adatbázisoknál nincs értelme többalakúságról vagy öröklésről beszélni, legalábbis adatbázis modell jelölésrendszerében semmiképp. Viszont az egységbezárás és újrahasznosítás értelmezhető fogalmak. Osztály-objektum kontra adattábla-rekord. Tehát a logikai adatmodell és az osztálydiagram jelentıs hasonlóságot mutat. Adattábláid mezőnevei, többnyire osztályaid tulajdonságaival egyeznek meg.

10. Objektumok és osztályok közötti kapcsolatok. A kapcsolatok implementálása. Öröklődés, polimorfizmus, virtualitás.

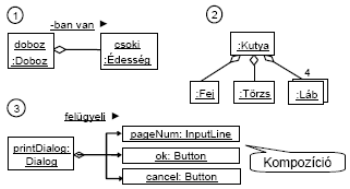
10.1 Obj.ok és osztályok közötti kapcsolatok, kapcsolatok implementálása:

Az **objektumok** csak úgy tudnak együttműködni, ha azok társítási kapcsolatban állnak egymással. Ha a kliens objektum üzenetet akar küldeni egy másik objektumnak, akkor tartalmaznia kell a megszólítani kívánt szerver objektum azonosítóját.



Kétféle társítási kapcsolat létezik:

- Ismeretségi kapcsolat van két objektum között, ha egyik léte se függ a másikétól és legalább az egyik ismeri, illetve használja a másikat. A kapcsolat nevét a vonal fölött aláhúzva írhatjuk, illetve a kapcsolat irányát a nyíl jelzi.



- Tartalmazási kapcsolat van két objektum között, ha az egyik objektum fizikailag tartalmazza vagy birtokolja a másik objektumot. A tartalmazó objektum az összetett vagy egész objektum, a benne levő pedig a részobjektum. A részobjektum léte az egész objektumtól függ. Ha az egész objektumot megszüntetjük, vele együtt pusztul a rész is.
  - Gyenge tartalmazás, ha a rész kivehető az egészből, (1)
  - Erős tartalmazás, ha a rész nem vehető ki az egészből ➔ kompozíciónak nevezzük azt a tartalmazást, amelyben az egész objektumot erős tartalmazási kapcsolat köti össze valamennyi részével. (2) teli rombusz erős, üres gyenge tartalmazás.

Függőség (dependency): *logikai kapcsolat. Az egyik (független) dolog változása maga után vonja a másik (függő) dolog változását.* ----------➤
Általánosítás *(generalization) –* ***öröklés:*** *osztályszerű elemek közötti strukturális kapcsolat.* ——▷
megvalósítás *(realization): egy dolog megvalósít (realizál, implementál) egy másikat. Logikai kapcsolat mely az általánosítás és függőség keveréke. Csak osztályszerű elemek között lehetséges.* – – – – – – – –▷

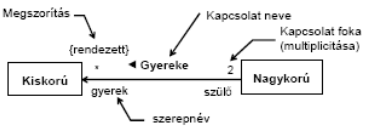
Az UML –ben úgy jelöljük a kapcsolatokat, két objektum között, hogy az objektumokat összekötjük egy vonallal. A vonal végén lévı nyíl a navigálás irányát mutatja. Információs jelleggel a vonal fölé, vagy alá írhatjuk a kapcsolat nevét és irányát. A nevet itt aláhúzzuk, mert objektumok közötti kapcsolatról van szó. Az objektumokat és azok kapcsolatait ábrázoló diagramot objektumdiagramnak nevezzük.

Osztályok között ugyanúgy értelmezünk ismeretségi, illetve tartalmazási kapcsolatot, mint objektumok között. Egy feladat kapcsán is általában nem konkrét objektumokról beszélünk, hanem osztályok közötti kapcsolatokról, hiszen a szabályokat legtöbbször általánosan kell meghatározni. Az osztályok közötti társítási kapcsolatot osztálydiagramon szokták ábrázolni és segítségével kifejezhető, hogy az egyik osztály egy-egy objektuma hány obejktummal állhat kapcsolatba egy másik osztályból és milyen lehet az a kapcsolat.

Egy-egy kapcsolat: az egyik osztály egy példánya a másik osztály legfeljebb egy (0..1) példányával állhat kapcsolatban. A másik osztályra ugyanez vonatkozik. (Pl. Férfi és Nő házastársi viszonya).

Egy-sok kapcsolat: az egyik osztály egy példánya a másik osztály sok példányával állhat kapcsolatban, a másik osztály egy példánya viszont legfeljebb egy példánnyal állhat kapcsolatban az egyik osztályból. (Anya-Gyerek, Ország-Város).

Sok-sok kapcsolat: mindkét osztály akármelyik példánya a másik osztály sok példányával állhat kapcsolatban. (Pl. Tanfolyam-Hallgató, Hallgató-Hallgató (*)).



Társítási kapcsolat megvalósítása: a kliens (kérő) objektumnak ismernie kell a szerver (kiszolgáló) objektumot, különben nem tudja azt megszólítani. A program készítésekor a kliens objektumban felveszünk egy szerverre vonatkozó referenciát.

Egy-egy kapcsolat megvalósítása: az egyik osztályban felveszünk egy referencia tulajdonságot a másik osztályra.

Egy-sok kapcsolat megvalósítása: konténer objektumokkal (pl. Vector) valósítjuk meg, de megvalósítható több referenciatulajdonság definiálásával is.

- *Tartalmazási kapcsolat esetén a kliens fizikailag tartalmazhatja a szervert, ekkor tehát nyugodtan felvehetünk a kliens osztályában egy (szerver) obj. típusú változót.*
- *Ismeretségi kapcsolat esetén a kliens nem birtokolhatja a szervert, mert akkor azt más nem használhatná. Ekkor a kliensből 1 mutatót irányítunk a szerverre.*
- *Ha 2 obj. 1:1 kapcsolatban áll egymással, akkor a kliens obj.nak tartalm.nia kell 1 mutatót a szerverre, h. +szólíthassa azt:.*
- *Az 1:N kapcsolatok +valósítására olyan konténereket használunk, amelyekbe szükség szerint akárhány obj. bedobható. Ahhoz, h. a Kliens kül.ző karbantartási és keresési műveleteket hajtson végre a kapcsolódó obj.okon, valamilyen módon fizikailag hozzá kell kötözni őket. Kliens nem mutathat külön minden 1es szerverre, hiszen a szerverek száma elvileg végtelen lehet. Legyen a kliens obj.nak 1 konténere (tárolója), amelybe elvileg akárhány szerver obj. betehető. Amelyik szervernek Kliens üzenni akar, annak a mutatóját egyszerűen elkéri a konténertől Az 1-sok kapcsolat +valósítása konténer obj.mal lehetséges.*

10.2 Öröklődés, polimorfizmus, virtualitás.

Öröklődés: Ősosztálynak nevezzük amiből örökítünk, illetve leszármazott vagy specializált osztály a létrejövő utód. Az öröklődés hierarchiájának mélysége tetszőleges (bár 10 felett már átláthatatlan) így beszélhetünk alaposztályról is, amely a hierarchia legfelső szintjén áll. Az osztály örökítésekor három lehetőségünk van:

- Új változókat adunk az ősosztályhoz,
- Új metódusokat adunk az ősosztályhoz,
- Az ősosztály metódusait átírjuk, azaz módosítjuk.

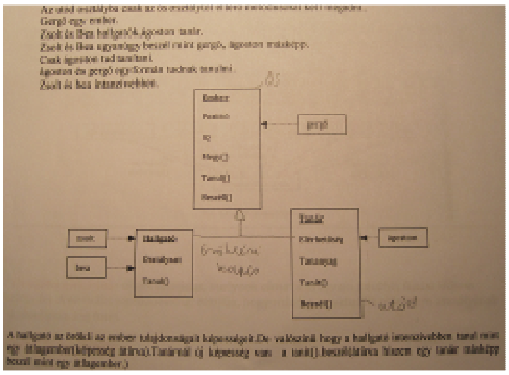
Megjegyzés: az ősosztály adatait nem lehet átírni! Új adatok létrehozásával még nem érünk el semmit, szükséges új metódusokat létrehozni.

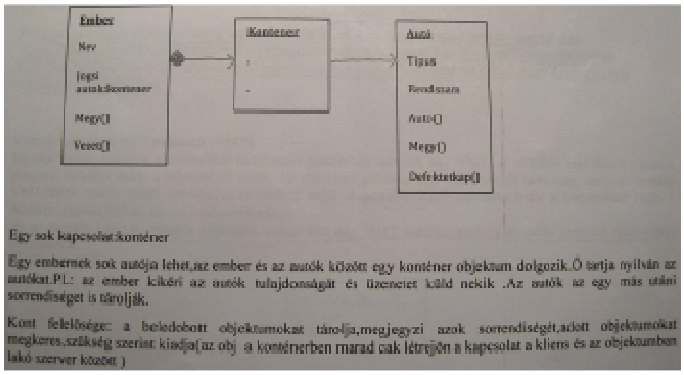
Szabályok:

- a hierarchia mélysége tetszőleges, de csak 10 alatti szám ajánlott,
- az öröklés tranzitív: ha A örökli B –t, és B örökli C –t, akkor A örökli C –t,
- példányadatok öröklése: az utód osztály példányainak adatai = ős adatok+ saját
- példánymetódusok öröklése: bármely metódust felül lehet írni.
- Az utód osztály az ős osztály kapcsolatait is örökli.

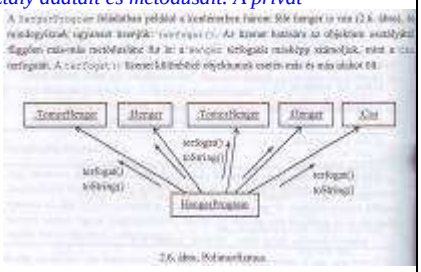
Egyszeres öröklésről akkor beszélünk, ha egy osztálynak csak egy őse lehet.

Többszörös öröklés: egy osztálynak több őse is lehet. Megjegyzés: a Java –ban a Pascalban, és a Smaltalkban csak egyszeres öröklés van.





Láthatóság, (hozzáférési mód): *Már volt szó arról, hogy az adatokat nem szabad kívülről manipulálni, és vannak olyan metódusok is, amelyeket a külső felhasználók elől el kell zárni. (lásd. Bezárás). A módszerek a következők:*
• Nyilvános (public): minden kapcsolatban álló kliens eléri és használhatja. UML jelölése: +
• Védett (protected): hozzáférés csak öröklésen keresztül lehetséges. UML jelölése: #
• Privát (private): az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá. UML jelölése: -
Egy kész osztályt kétféleképpen lehet használni:
• az osztályból példányt hozunk létre: egy objektumnak kizárólag csak a publikus deklarációit lehet elérni. A privát és védett deklarációkat az objektum megszólításával nem lehet használni.
• az osztályból örökítéssel új osztályt hozunk létre: az örökítéskor az új utód osztályban felhasználjuk a már meglévő osztály adatait és metódusait. A privát deklarációkat csak az osztály programozója érheti el, ahhoz még öröklés révén sem lehet hozzáférni.. a nyilvános és védett deklarációkat az utód osztály használhatja, hivatkozhat rájuk.
Szabályok:
− a láthatóságot nem kötelező megadni. A láthatóság alapértelmezése, az ún. csomag szintű láthatóság, ez azt jelenti, hogy a deklaráció az aktuális csomagban nyilvános.
− Egy osztálynak is van láthatósága: a publikus osztály más csomagokból is látható; alapértelmezésként egy osztály csak a saját csomagjában látható.



Polimorfizmus (többalakúság): Azt jelenti, hogy ugyanarra az üzenetre különböző objektumok különbözőképpen reagálhatnak; minden objektum a saját, az üzenetnek megfelelő metódusával. (Az üzenet küldőjének nem kell tudnia a fogadó objektum osztályát).

Virtuális metódus: Az objektumokkal történő munka során szükség lehet arra, hogy az utód osztály metódusait megváltoztassuk. Erre ad lehetőséget a virtuális metódusok használata. A virtuális metódusokkal átdefiniálható az ősosztály azonos nevű metódusa, így csak a futás közben dől el, hogy éppen melyik metódust kell használni.

Virtuális Metódusok Táblázata (VMT) Minden eggyes, virtuális metódusokat tartalmazó osztályhoz tartozik egy VMT. A virtuális metódusok címét a program futásakor ebből a táblázatból veszi. Az objektum példány egy VMT mezőt tartalmaz, mely az osztály VMT-jének relatív címét tartalmazza (mérete 2 bájt). A példány VMT hozzárendelést a konstruktor végzi a példány létrehozásakor, ill. inicializálásakor. Ha az osztály használ virtuális metódust, akkor van egy VMT mezője, mely a virtuális metódus tábla címét tartalmazza.

Másképp:

Virtualítás: Az @Override annotáció lényege, hogy egy ilyenformán jelölt metódusnak felül kell definiálnia egy azonos nevő és tulajdonságú metódust az ősosztályban. Pl.

```
public class MyClass {

    public MyClass() {
    }

    @Override
    public String toString() {
        return "Ez egy virtuális metódus";
    }

    public static void main(String[] args) {
        new MyClass().toString();
    }
}
```

A fenti kódrészletben, egy osztályban annak ősosztályában (pl. Object) már meglévő metódust (toString()) írtunk felül. Ez a metódus az Object osztályban is pontosan így néz ki (visszatérési típus, név, paraméterek). A @Override annotáció elsősorban a fordítónak jelent nagy segítséget, amikor kinyomozza, hogy mely konkrét osztály metódusát is kell majd végrehajtani. (Jelen esetben nem az ősosztály (Object) toString()-je fog végrehajtódni, hanem az adott osztály (MyClass) toString()-je).

11. Algoritmusvezérelt és eseményvezérelt programozás összehasonlítása (vezérlés elve, működési mód és felhasználóval való kommunikáció alapján)

A programvezérlési módszereket osztályozhatjuk aszerint, hogy a felhasználó hogyan avatkozik be a program menetébe. Egy program kétféle lehet aszerint, hogy lehet-e vele beszélgetni:

- **Kötegelt:** (batch) programnak az aktor megadhat indítási paramétereket, de a futó program működésébe már nem szólhat bele.
- **Interaktív:** (párbeszédes) programmal a felhasználó, a program futása közben is kommunikálhat.

Egy **interaktív** program a párbeszéd módjától függően kétféle lehet:

- **Algoritmusvezérelt:** az interakciót a program irányítja. Az aktor csak válaszol a program által feltett kérdésekre. A felhasználó pontosan akkor és azt mondhatja, amikor és amit a program kérdez tőle.
- **Eseményvezérelt:** egy tökéletesen eseményvezérelt program reagál a felhasználó által keltett kérésekre, kérdésekre. Az aktor azt mondhatja, amit ő akar, és akkor amikor akarja!

Az eseményvezéreltség együtt szokott járni a grafikus felhasználói felülettel, de ez nem szükségszerű követelmény.

Az algoritmusvezérelt program: a kommunikációt a program irányítja. Beolvassa az adatokat, elvégzi a számításokat, majd közli az eredményeket és befejezi a működést. Csak akkor van lehetőség a vezérlésre, ha a program erre külön rákérdez, vagy újra elindítjuk. Az algoritmusvezérelt program futtatása egy folyamat: a beolvasás, feldolgozás, eredményközlés folyamata.
Az eseményvezérelt program: a kommunikációt a felhasználó irányítja, bármikor beavatkozhat a program futásába és közölheti a kívánságait a programmal, tetszőlegesen módosíthatja az adatokat. A program fogadja és feldolgozza a beavatkozásokat, végrehajtja az utasításokat. A program futása is a felhasználó döntésére fejeződik be.

Az eseményvezérelt programozás lényege, hogy a program ill. egyes részei, ágai nem szekvenciális és előre meghatározható sorrendben futnak le, hanem a vezérlés lefutását bizonyos külső ill. belső események határozzák meg. A program egésze nem más, mint események bekövetkezésére válaszul végrehajtott szubrutinok (ún. eseménykezelők) laza halmaza, amelyek nagyrészt egymástól függetlenül dolgoznak és működtetik a program egészét. Az esemény-vezérelt modellben külső események alatt tipikusan a felhasználói bevitelt (billentyű lenyomása, egérművelet), valamint a hardvertől származó eseményeket (pl. az időzítőáramkor vagy perifériafia által kiváltott megszakítás, visszahívás) szokás érteni, míg a belső eseményeket a program más részeitől - vagy akár más programoktól - származó üzenetek képezik. Eseményvezérelt program szinte bármilyen programozási nyelvben készíthető, de különösen könnyű és célszerű olyan nyelvet használni, amely támogatja az objektumorientált programozást, mert a többalakúság rendkívül egyszerűvé teszi az eseménykezelő architektúrák és hívási láncok kialakítását. Az eseményvezérelt programozás olyan programozás, amely egy eseménybegyűjtő és szétosztó mechanizmuson alapszik. Az objektumok a hozzájuk rendelt eseményeket, eseménykezelő metódusok segítségével lekezelik. A metódusok ugyanolyan szabályok szerint kerülnek meghívásra, mint egy algoritmusvezérelt programban. Van azonban egy lényeges különbség: egy eseményvezérelt program fő szála mellett működik egy eseményelosztó szál, benne egy eseményelosztó ciklus, amely folyamatosan figyeli, hogy bekövetkezett-e valamilyen esemény. (event) Ha igen, akkor végignézi, hogy melyik komponenst (objektumot) illeti az esemény lekezelése – billentyűzet lenyomás esetén azt, amelyik éppen fókuszban van (aktív), egéresemény esetén azt, amelyiken rajta van az egér -. Az eseményelosztó odaadja az esemény a „jogos” tulajdonosának, hogy az lekezelhesse. Ezt az eseménykezelő metódusok végzik, a programozó dolga az, hogy megírja ezeket a lekezelő metódusokat, és közölje a rendszerrel, hogy ezen az objektumon milyen eseményt figyeljen.

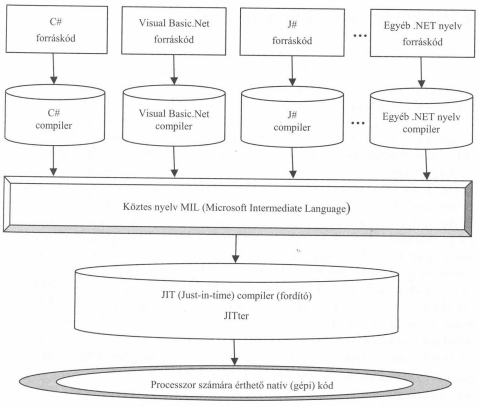
vezérlés elve:
Algoritmusvezérelt: előre meghatározott sorrend szerint, szekvenciálisan
Eseményvezérelt: az események bármikor bekövetkezhetnek, nem mindre kell reagálni
működési mód,
Algoritmusvezérelt: a processzor jól meghatározott feladatokat hajt végre, nincs szükség beavatkozásra. Csak azok az interakciók lehetségesek, amiket a programozó lekódolt.
Eseményvezérelt: a program akár percekig "áll", az operációs rendszer által küldött üzenetekre bármikor válaszol. Az esemény mindenképpen bekövetkezik, legfeljebb nincs azt lekezelő rutin.
felhasználóval való kommunikáció alapján).
Algoritmusvezérelt: a felhasználó akkor kommunikálhat, ha azt a program megengedi
Eseményvezérelt: a felhasználó (szinte) bármikor bármilyen üzenetet küldhet

pl.
a. print, polling, DMA-vezérlő, shell script
e. IRQ, interrupt-masking, exception

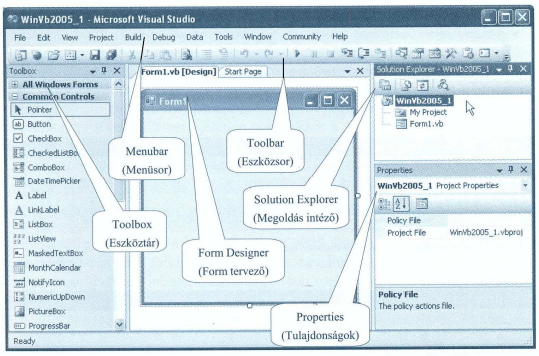
12. Egy vizuális fejlesztő eszköz bemtatása: a fejlesztőkörnyezet elemei, szolgáltatásai, osztályhierarchia, látható és nem látható komponensek, adateléréshez kötődő komponensek.

A Microsoft Visual Studio integrált fejlesztői környezet, a hozzá kapcsolódó .Net keretrendszerrel gyors alkalmazásfejlesztést tesz lehetővé. Segítségével a következő programozási nyelvekkel tudunk alkalmazásokat készíteni:
Visual C#, Visual J#, Visual Basic, Visual C++ és lehetőség van szerver oldali programozásra ASP.NET alapon.

A .NET egy alkalmazás keretrendszer, amiben egy csomó programrészlet beépítetten kész van. Így gyorsan lehet alkalmazásokat fejleszteni. Mivel sok minden gyárilag jól van megírva, a program eleve kevesebb hibalehetőséggel készül el. Ezen felül meg van a rendszer azon előnye, hogy platformtól független. Hasonlóan a Java-hoz, a bináris fájlok egy virtuális, csak specifikáció szinten létező gépre készülnek. De a program nem virtuális gépen fut, hanem indításkor fordítódik le a használt gép által értelmezhető kódra. Ennek köszönhetően egy alkalmazás futhat 32 bites rendszeren, 64 bitesen és akár Tableten is. Bizonyos körülmények teljesülése esetén Linuxon és OS-X-en is.
A .NET keretrendszer hivatott arra, hogy kapcsolatot teremtsen az alkalmazás és az operációs rendszer között. A CLR (Common Language Runtime) közös nyelvi futtatómodul felel a .NET alkalmazások futtatásáért, de ehhez egy általa emészthető nyelvre van szükség. Minden .NET nyelv rendelkezik egy saját fordítóval (compiler), amelynek bemenete az adott nyelv szintaktikájának megfelelő kód, a kimenete pedig már a közös nyelvi futtatómodul számára érthető köztes nyelv. Amikor a kiválasztott .NET nyelvű környezetben létrehozunk egy alkalmazást, annak kódját a nyelv fordítója a közös nyelvi futtatómodul számára már érthető köztes nyelvű MIL (Microsoft Intermediate Language) kódra fordítja le. A közös nyelvi futtatómodul tulajdonképpen egy kiokosított virtuális gépnek is tekinthető, amely képes feltérképezni azt a számítógépes környezetet, amelyben fut, és annak megfelelően futtatja az alkalmazást. Ez a feltérképezés kiterjed mind a hardverkörnyezetre (pl. többprocesszoros alaplap), mind pedig a szoftverkörnyezetre (pl. DOS- vagy NT-alapú operációs rendszer). Mivel minden tekintetben a CLR felügyeli a kód és az általa elérhető erőforrások biztonságos felhasználását, ezért ebben a közös nyelvi futtatókörnyezetben futó kódot felügyelt kódnak hívjuk. Ezt a köztes nyelvű kódot már könnyen az egyes platformoknak leginkább megfelelő gépi kódra lehet fordítani. Ennek megfelelően a CLR tartalmaz egy futás előtti JIT (Just-in-time) fordítót (JITter), amely a processzor számára érthető gépi kódot hoz létre.

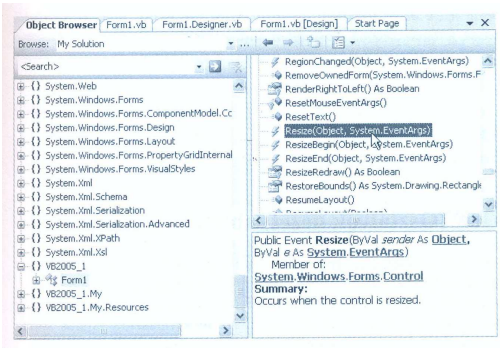


A .NET keretrendszer alatti alkalmazásfejlesztés valójában az osztályok kezelésén alapul. Habár az osztályok nagy részét készen kapjuk a keretrendszerrel, mégis előfordulhat, hogy nem találunk köztük megfelelőt. Ekkor lehetőségünk van más fejlesztők osztályait kölcsönvenni, sőt mi is létrehozhatunk saját fejlesztésű osztályokat. Amikor valamilyen .NET-kompatibilis nyelven fejlesztünk, akkor csak ki kell keresni a szükséges funkciót ellátó osztályt, így csökkentve mind a fejlesztésre fordított időt, mind a programkódunk méretét. A .NET keretrendszer osztálykönyvtárának osztályait a köztük lévő eligazodás érdekében valamilyen hierarchikus struktúrába kellett szervezni. Ezt a jól meghatározott logika szerint felépített elrendezést névtér struktúrának hívják, amire tekinthetünk akár úgy is, mint az osztálykönyvtár tartalomjegyzé-kére. Az osztálykönyvtár ilyen módon való névterekbe szervezésével egyrészt elkerülhetők az elnevezésekből fakadó ütközések, másrészt a névterek közötti eligazodást is megkönnyíti. Ez a fajta csoportszervezés megengedi, hogy egy bizonyos osztálynevet többször is felhasználjunk, ha az nem ugyanabban a névtérben található.



A Visual Studio fejlesztőrendszerben minden olyan eszköz a rendelkezésünkre áll, amely a különböző típusú alkalmazások, illetve szolgáltatások létrehozásához szükséges. A tervezési, futtatási, hibakeresési és más funkcionális részek egyetlen helyről, az integrált fejlesztői környezetből IDE érhetők el. Az IDE a legtöbb windowsos fejlesztőrendszerhez hasonlóan rendelkezik menüsorral, eszközsorokkal, eszköztárral, grafikus tervezői felülettel, projektintézővel, tulajdonságok ablakkal stb.
Az **eszköztár** alapvetően a felhasználói felületek kialakításához használható controlok (pl.: button, Label) kollekcióját tartalmazza, amely a választott projekt (windowsos, webes, konzolalkalmazás) típusának, illetve az elvégzendő feladatnak megfelelően más-más képet mutat.
A **megoldás intéző** ablakban az adott feladat megoldásához tartozó projekteket találjuk. Minden egyes projekt magában foglalhat több Forrnot, illetve több egyéb modult is, de egy Formot, vagy egy modult minimálisan tartalmaznia kell. Az egyes projektek tartalmát a könnyebb áttekinthetőség érdekében fastruktúrába rendezve láthatjuk.
A **grafikus felülettervező** segítségével lehet az alkalmazás felhasználói interfészét elkészíteni. Windowsos alkalmazás esetén ez egy Form, amely futás közben egyablakként jelenik meg. A Form az alkalmazás elsődleges grafikus felhasználói felületeként jelenik meg, amely tartalmazhat controlokat, különféle grafikákat és képeket a tervező esztétikai érzékének és a megvalósítandó funkcióknak megfelelően.
A **tulajdonságok ablak** tartalmazza a kiválasztott objektumhoz rendelt tulajdonságoknak a listáját. Ezen tulajdonságoknak a készlete objektumfüggő, hiszen minden egyes objektumnak más és más a funkciója. A *Tulajdonságok* ablakot leggyakrabban a Formok, illetve a rajta elhelyezett controlok tulajdonságainak beállítására használjuk.

Az **Objektumböngésző** (*Object Browser)* dialógusablakban egy kiválasztott objektum minden jellemzőjét (események, tulajdonságok, tagfüggvények, stb) megnézhetjük.
Az események kezelését (interaktivitást) kódszinten kell megoldani, ami a **Kódszerkesztőben** (Code Editor) történik. Nemcsak az alkalmazáshoz felhasznált objektumok eseményeihez tartozó kód, hanem minden egyéb eljárás kódjának megírása itt történik.
Az **Osztályok nézet** *(Class View)* ablakban a projektek osztályait és azok tagjait láthatjuk. Az itt kiválasztott eljárás sorára duplán kattintva, a Kódszerkesztő megfelelő helyére kerülünk.
A fejlesztőrendszernek a Professional Edition verziója a hagyományos súgó mellett egy **Dinamikus súgó** (*Dynamic Help*) ablakot is tartalmaz. Ebben az ablakban mindig a kiválasztott elemhez aktuálisan hozzárendelhető témakörök listája található.



A fejlesztőkörnyezet komponensei:
A komponens egy olyan bináris (lefordított) szoftver egység, amely jól definiáltan megvalósít egy funkció.
Azokat a komponenseket, melyeken keresztül a felhasználó kezelni tudja az alkalmazást, vezérlőknek (controls) nevezzük. Ilyenek a gomb, a menü, a szövegdoboz, kép, stb. Léteznek
olyan komponensek is, mint például az időzítő (Timer), melyek futtatás közben nem jelennek
meg a képernyőn. Ezek nem tartoznak a vezérlők közé.

- látható (visible): A közönséges vezérlők (common controls), ilyen pl: a button, label, menü …)
- nem látható (logical): Pl.: Időzítő, EventLog, DirectoryEntry, Process,...

Osztályhierarchia: A projekt osztálydiagramját megtekinthetjük, ha az adott projekten jobb egérgombbal kattintunk, és kiválasztjuk a „View Class Diagram” lehetőséget.
A programnyelvhez adott osztályokat csoportosították, ezeket online és offline(feltelepített CD) kereshetővé tették.

Adateléréshez kötődő komponensek.
.NET alkalmazás keretrendszer része az adatbázis-elérés biztosító ADO.NET. Az ADO.NET komponensei segítségével elkülöníthető egymástól az adatokkal történő manipuláció, és azok megjelenítése. A komponensek ugyanis komplex objektumok, melyekkel elérhetjük a tetszőleges típusú adathalmazokat, elvégezhetjük az adatok módosítását, törlését, valamint visszakaphatjuk a lekérdezések eredményhalmazait. A kapott adathalmazokat ezt követően adatmegjelenítő objektumokkal tesszük láthatóvá. Egy adatbázistábla sorainak eléréséhez a DataGrid vezérlőt használhatjuk, Az SqlConnection objektumokkal SQL Server adatbázisokhoz kapcsolódhatunk, míg az SqlDataAdapter objektumokkal sorokat vihetünk át az SQL Server adatbázis és egy DataSet objektum között. A Query Buildert használjuk az SQL utasítások megadására, a lekérdezést beírhatjuk, de grafikusan is felépíthetjük őket.

13. Relációs adatbázisok. Funkcionális függőség fogalma, speciális függőségek szerepe. Normálformák, a normalizálás célja. A normalizálás lépéseinek szemléltetése példán. Az adatbázis-terv dokumentációja.

13.1 Relációs adatbázisok

Relációs adatbázisok (oszlopokba szedett adatok összessége): A reláció az adatelemek megnevezett, összetartozó csoportjából kialakított olyan kétdimenziós táblázat, amelyik sorokból és oszlopokból áll, és ahol az oszlopok 1-1 tulajdonságot írtnak le, a sorok adják az egyedhalmaz/reláció/táblázat egyedeit. Ahhoz, hogy egy táblázatot relációnak lehessen tekinteni, a következő feltételeket kell kielégítenie:

- nem lehet két egyforma sora,
- minden oszlopnak egyedi neve van,
- a sorok és oszlopok sorrendje tetszőleges.

A relációs adatbázisok általában nem egy, hanem több, logikailag összekapcsolható relációból (táblázatból) állnak. A reláció oszlopainak (attribútumainak) számát a reláció fokszámának, sorainak számát a reláció kardinalitásának nevezik.

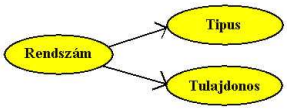
Kulcs - amennyiben egy tulajdonság vagy tulajdonságok egy csoportja egyértelműen meghatározza, hogy az egyed melyik értékéről, előfordulásáról van szó, akkor ezeket a tulajdonságokat együtt kulcsnak nevezzük.

13.2 Funkcionális függőség fogalma, speciális függőségek szerepe.

A függőség fogalmának segítségével a táblázatok belső szerkezetét tárhatjuk fel. A funkcionális és többértékű függőség a táblák oszlopai között lévő összefüggéseket írja le.

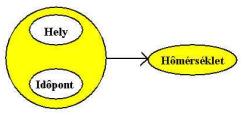
Funkcionális függőség: Adatok között akkor áll fenn funkcionális kapcsolat, ha egy vagy több adat konkrét értékéből más adatok egyértelműen következnek. Például a személyi szám és a név között funkcionális kapcsolat áll fenn, mivel minden embernek különböző személyi száma van. Ezt a **SZEMÉLYI_SZÁM -> NÉV** kifejezéssel jelöljük vagy pedig egy diagrammal.

A funkcionális függőség bal oldalát a függőség meghatározójának nevezzük. Nem áll fenn funkcionális függőség akkor, ha a meghatározó egy értékét több attribútum értékkel hozhatjuk kapcsolatba. Például a **NÉV -> SZÜLETÉSI_ÉV** állítás nem igaz, mert több személynek lehet azonos neve, akik különböző időpontokban születtek.



A funkcionális függőség jobb oldalán több attribútum is állhat. Például az **AUTÓ_RENDSZÁM -> TIPUS, TULAJDONOS** funkcionális függőség azt fejezi ki, hogy az autó rendszámából következik a típusa és a tulajdonos neve, mivel minden autónak különböző a rendszáma, minden autónak egy tulajdonosa és típusa van. Ezt diagrammal is ábrázolhatjuk.

Az is előfordulhat, hogy két attribútum kölcsönösen függ egymástól. Ez a helyzet például a házastársak esetén **FÉRJ_SZEM_SZÁMA -> FELESÉG_SZEM_SZÁMA FELESÉG_SZEM_SZÉMA <- FÉRJ_SZEM_SZÁMA**. Mindkét funkcionális kapcsolat igaz és ezt a **FÉRJ_SZEM_SZÁMA <-> FELESÉG_SZEM_SZÁMA** jelöléssel fejezzük ki.

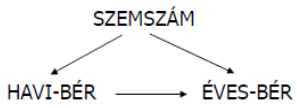


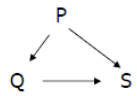
A funkcionális függőség bal oldalán több attribútum is megjelenhet, melyek együttesen határozzák meg a jobb oldalon szereplő attribútum értékét. Például hőmérsékletet mérünk különböző helyeken és időben úgy, hogy a helyszínek között azonosak is lehetnek. Ebben az esetben a következő funkcionális függőség áll fenn az attribútumok között: **HELY, IDŐPONT -> HŐMÉRSÉKLET**. A fenti összefüggést az alábbi diagrammal is jelölhetjük:

Speciális függőségek:

Teljes funkcionális függőség. Erről akkor beszélhetünk, ha a meghatározó oldalon nincsen felesleges attribútum. Például a **RENDSZÁM, TÍPUS -> SZÍN** funkcionális függőség nem teljes funkcionális függőség, mivel a rendszám már egyértelműen meghatározza a kocsi színét, ehhez nincs szükség a típusra is.

A **tranzitív függőség** gyakorlatilag azt jelenti, hogy két funkcionálisan függő attribútumhalmaz mellé található egy harmadik attribútumhalmaz a relációban, amely a két halmaz közötti függőséget átviszi.





A funkcionális függőségek tulajdonságai:

- ***(reflexivitás)*** *Ha $Q \subseteq P \subseteq A$, akkor $P \rightarrow Q$ teljesül, azaz egy attribútum halmaz a benne levő részhalmazt meghatározza. egy tulajdonságtípus bármely értéke meghatározza önmagát.*
- *additív* ***(egyesítési szabály)*** *$A \rightarrow B, A \rightarrow C \Rightarrow A \rightarrow (B+C)$ ha A tul. típus 1 B és 1 C tul. típust határomeg, úgy meghatározza a (B+C) tulajdonság típus sort is.*
- ***(tranzitivitás)*** *$A \rightarrow B$ és $B \rightarrow C \Rightarrow A \rightarrow C$, ha 1 A tulajdonság típus meghatároz egy B tulajdonság típust, ami C -t határozza meg, a C az A -tól is függ.*
- ***(pszeudo-tranzitivitási szabály)*** *$A \rightarrow B$ és $(B+x) \rightarrow C \Rightarrow (A+x) \rightarrow C$ ha egy A tulajdonság típus meghatároz 1 B -t, amely pedig bármilyen bővítményével (B+X) meghatároz 1 C -t, akkor az A megfelelő bővítménye (A+X) is meghatározza a C -t.*
- ***Bővítés:*** *Ha $P \rightarrow Q$ teljesül és $S \subseteq A$ egy tetszőleges attributúmhalmaz az A - ból, akkor $PUS \rightarrow QUS$*
- ***Dekompozició*** *Ha $P \rightarrow Q$ teljesül és $S \subseteq Q$, akkor $P \rightarrow S$ is teljesül.*

1. **Reflexivitás**
   Ha $Q \subseteq P \subseteq A$, akkor $P \rightarrow Q$
2. **Bővítés**
   Ha $P \rightarrow Q$ és $S \subseteq A$, akkor $P \cup S \rightarrow Q \cup S$
3. **Tranzitivitás**
   Ha $P \rightarrow Q$, $Q \rightarrow S$, akkor $P \rightarrow S$
4. **Egyesítési szabály**
   Ha $P \rightarrow Q$, $P \rightarrow S$, akkor $P \rightarrow Q \cup S$
5. **Pszeudotranzitivitási szabály**
   Ha $P \rightarrow Q$,$T \cup Q \rightarrow S$, akkor $P \cup T \rightarrow S$
6. **Dekompozíciós szabály**
   Ha $P \rightarrow Q$ és $S \subseteq Q$, akkor $P \rightarrow S$

13.3 Normálformák, a normalizálás célja. A normalizálás lépéseinek szemléltetése.

A *normalizálás segítségével* minimalizáljuk a tárolási helyet, megszüntetjük az adatok többszörös tárolását, redundanciát, továbbá a beírási, törlési és módosítási anomáliákat, amelyeket az okoz, hogy túl sok adatot tárolunk egy táblázatban.

Normalizálás során induláskor egyetlen táblázatot alakítunk ki, amelyben minden szükséges tulajdonságot elhelyezünk. Ezt követően feltárjuk a táblázat belső szerkezetét, azaz meghatározzuk az oszlopok függőségi viszonyait, majd megvizsgáljuk, hogy a feltárt függőségek eleget tesznek-e azoknak a követelményeknek amelyet normálformának

nevezünk. A normálformákat megsértő függőségeket úgy szüntethetjük meg, hogy a táblázatot egy meghatározott szabály szerint, több lépésben további táblázatokra bontjuk. A normálformáknak való megfelelés ellenőrzését a szétbontással keletkezett új táblázatoknál előről kezdjük. Napjainkban hat normálformát definiáltak, de a gyakorlatban elég az első 3 normálformával dolgozni.

Első normálforma: egy táblázat akkor van normálformában, ha minden sorában pontosan egy attribútum érték áll.
Ha egy reláció nincs 1NF-ben, akkor kétféleképpen alakítható át ilyen típusúvá:
1. A több attribútumértéket tartalmazó sort annyi sorra bontjuk, ahány nem elemi attribútumérték szerepelt benne.
2. Az eredeti relációt több 1NF-es relációra bontjuk úgy, hogy az egyikben a reláció kulcsának értékei mellé írjuk az egyszeres attribútumértékeket, a másik relációban pedig a kulcshoz rendelt külső kulcs mellé annyi sort írunk, ahányszoros attribútumértékek szerepelnek a többszörös attribútumokban.

Második normálforma: egy táblázat akkor és csak akkor van második normálformában, ha minden másodlagos attribútum teljesen függ a kulcstól. (A másodlagos attributomok nem függhetnek a kulcs részeitől)
• A definícióból következik két egyszerű kritérium, mely megkönnyítheti a 2NF típusú relációk felismerését:
1. Ha a kulcs egyetlen attribútumból áll, (vagyis egyszerű) akkor a reláció 2NF típusú.
2. Ha a relációban nincsenek másodlagos attribútumok (tehát minden attribútum része a kulcsnak) akkor a reláció 2NF típusú.
Ha egy reláció nincs 2NF-ben, akkor alakítható át ilyen típusúvá:
Az 1NF-jú táblázatból olyan táblázatokat kell létrehoznunk, amelyekben a kulcs minimális, azaz a kulcs tulajdonságokból és a másodlagos tulajdonságokból önálló relációkat hozunk létre.
1. Kiemeljük a kulcsból azokat az attribútumokat (vagy attribútumhalmazokat), amelyek önállóan is meghatározzák a másodlagos attribútumokat.
2. Az 1. pontban kiválasztott elsődleges és az 1. pont szerint hozzájuk tartozó másodlagos attribútumokból egy relációt állítunk elő.
3. Azokat a másodlagos attribútumokat, amelyek csak a kulcstól függnek, (a részeitől nem) tehát teljes függőségben vannak a kulccsal, a kulcsban szereplő elsődleges attribútumokkal együtt egy táblába fogjuk össze.

Harmadik normálforma: A reláció harmadik normálformában van akkor és csak akkor, ha 2NF-ben van és a másodlagos attribútumok között nincsen funkcionális függőség.
Ha egy reláció nincs 3NF-ben, akkor alakítható át ilyen típusúvá:
Egy reláció 3NF-re hozható, ha megszüntetjük a tranzitív függőségeket úgy, hogy a tranzitív függőségben részt vevő attribútumhalmazok felhasználásával új relációkat készítünk.

Példa:

Egy kórházban a betegeknek kartonjuk van. Ezen a kartonon szerepel a *nevük*, címük, más személyi adataik, továbbá *betegségeik*. A betegségükből következik hogy melyik *osztályon* fekszenek, valamint az, hogy milyen *gyógyszereket* kapnak. Képzeljük el, hogy ezek a kartonok mind a kórház adminisztrációs irodájában vannak. Tegyük fel, hogy valakinek kimutatást kell készítenie arról, hogy egy bizonyos gyógyszert hány beteg szed. Az illetőnek az összes kartont végig kell lapoznia, a rengeteg adatból ki kell keresnie a gyógyszerek nevét. Ki kell választania a kartonok közül azoknak a kartonjait, akik az adott gyógyszert szedik. Akár több napot is el lehetne ezzel a munkával tölteni. Én azt szeretném bemutatni, hogy a normalizáció eredményeként kapott relációs modell segítségével ez mennyivel egyszerűbb feladat lesz.
Tegyük fel, hogy a kartonon a következő adatok szerepelnek a betegekről:

- Beteg azonosító (B_azon)
- Beteg neve (B_név)
- Beteg címe (B_cím)
- Betegség
- Osztály azonosító (Oszt_az)
- Osztály név (Oszt_név)
- Főorvos
- Gyógyszer

Képzeljünk el néhány rekord előfordulást, ami segít az elsődleges kulcs kiválasztásában is (célszerű elképzelésünket táblázatba is foglalni, nehogy elkerülje figyelmünket valami lényeges összefüggés):

B_azon	***B_név***	***B_cím***	***Betegség***	***Oszt_az***	***Oszt_név***	***Főorvos***	***Gyógyszer***
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin Semicillin
444	Kiss Cili	Ajka	tyúkszem	02	szemészet	Dr. Joó	Semicillin
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Sumetrolim Demalgon

Ezzel a táblázattal még nem könnyítettünk sokat a munkánkon, mert az ismétlődések (tárolási redundancia) és az ebből származó anomáliák miatt nehéz karbantartani. Ráadásul első normál formában (1NF) sincs, hiszen vannak benne többértékű mezők (Betegség és Gyógyszer), amit a relációs modell nem visel el. Továbbá hiányzik még az elsődleges kulcs is! A normalizálás első lépése az 1NF kialakítása. Mindenekelőtt a többértékű mezőket kell megszüntetni. A sorok ismételt leírásával elérhetjük ezt a célt.

B_azon	***B_név***	***B_cím***	***Betegség***	***Oszt_az***	***Oszt_név***	***Főorvos***	***Gyógyszer***
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Semicillin
444	Kiss Cili	Ajka	tyúkszem	02	szemészet	Dr. Joó	Semicillin
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Sumetrolim
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Demalgon

Mivel a relációban azonos sorok nem fordulhatnak elő, a sorismétlés összetett kulcsot fog eredményezni. Látható, hogy a *B_azon* önmagában nem elég a rekordok azonosítására. Szükség van a *Betegség*, sőt a *Gyógyszer* mezőre is. A három érték együttesen már egyértelműen azonosítja a táblázat sorait. Feltételezzük, hogy egy beteg egy bizonyos betegséget csak egyszer kap meg, valamint egy időben csak egy betegséggel kezelik.

B_azon	***B_név***	***B_cím***	***Betegség***	***Oszt_az***	***Oszt_név***	***Főorvos***	***Gyógyszer***
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Semicillin
444	Kiss Cili	Ajka	tyúkszem	02	szemészet	Dr. Joó	Semicillin
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Sumetrolim
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Demalgon

Felületes szemlélődés után esetleg a *B_azon* és a *Gyógyszer* értékeket is alkalmasnak találnánk az azonosításra. A táblázat második és harmadik sorában azonban az elsődleges kulcs megegyezne (444, Semicillin), ami nem engedhető meg egy 1NF-ban lévő relációban. Ha a *Betegséget* komponensként bevonjuk a kulcsba, akkor az azonosítás már egyértelműnek látszik (ha az előfordulási adatok általánosíthatók).

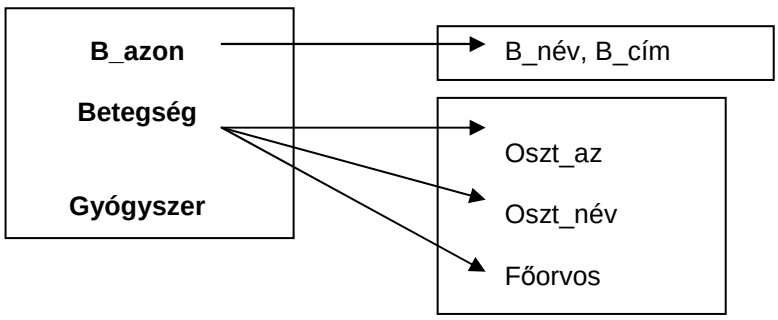
Ez a táblázat már *első normál formában* (1NF) van, mert:

- minden sora különböző
- az oszlopok száma és sorrendje minden sorban azonos
- minden oszlop csak egy attribútum értéket vesz fel
- minden sorhoz egy egyedi kulcs tartozik, amitől az összes többi attribútum funkcionálisan függ

A ***második* normalizálási** lépéshez meg kell vizsgálni, melyek azok az attribútumok, amelyeknek az egyes összetevőktől egyértelműen függenek és melyek azok, amelyeket az összetett kulcs határoz meg. Ezt követően hozzunk létre olyan táblázatokat, amelyekben az összes nem kulcs attribútum *teljesen* függ az elsődleges kulcstól. A teljesen azt jelenti, hogy az összetevők száma nem csökkenthető, vagyis a kulcs minimális. Ha a kulcs egyszerű, akkor a teljes függés ugyanazt jelenti, mint a funkcionális függés.

- a beteg *azonosítójától* teljesen függ a beteg neve és címe
- a *betegségtől* függ az osztály azonosítója és neve, valamint a főorvos
- a harmadik összetevőtől (*gyógyszer*) nem függ semmi más adat, de szükséges a sorok megkülönböztethetősége céljából.

A leírt összefüggéseket rajzzal is ábrázolhatjuk:



A rajz alapján a józan ész azt diktálja, hogy három táblázatra célszerű bontani 1NF táblázatunkat.

Beteg

B_azon	B_név	B_cím
444	Kiss Cili	Ajka
333	Nagy Pál	Veszprém

Ki_Mire_Mit_Szed

B_azon	**Betegség**	**Gyógyszer**
444	sérv	Algopyrin
444	sérv	Semicillin
444	tyúkszem	Semicillin
333	tyúkszem	Sumetrolim
333	tyúkszem	Demalgon

Osztály

Betegség	Oszt_az	Oszt_név	Főorvos
sérv	01	sebészet	Dr. Doktor
tyúkszem	02	szemészet	Dr Joó

A három táblázat (reláció) már *második normál formában* van, mert

- 1NF-ben van (előfeltétel) és
- a nem kulcs attribútumok funkcionálisan *teljesen* függnek az elsődleges kulcstól

Ebben a lépésben már megjelennek az idegen kulcsok is. A *Beteg* relációban a *B_azon* az elsődleges kulcs, mert ez határozza meg a többi tulajdonságot. A *Ki_Mire_Mit_Szed* relációban elsődleges kulcs a *B_azon*, a *Betegség* és a *Gyógyszer* attribútumok kombinációja, mert ez határozza meg az adott sort. Idegen kulcsok a *B_azon* és a *Betegség* mezők, mert a másik két táblázatra ezekkel a kulcsokkal hivatkozhatunk. Az *Osztály* relációban az elsődleges kulcs a *Betegség*.

Figyelemre méltó azonban az adatok számának csökkenése. Az 1NF táblázat összesen 40 adatot tartalmazott, míg itt a 2NF három relációja összesen 29 adatot tartalmaz csupán! Itt azonban még nem kell megállnunk. Ha az Osztály táblázatot jól megnézzük, észre lehet venni, hogy vannak mezők, amik közvetve is függenek az elsődleges kulcstól: A Betegség meghatározza az Oszt_az osztályazonosítót és az osztály azonosítója meghatározza az Osz_név osztálynevet és a főorvost. Emiatt még mindig maradtak rendellenességek.
Tegyük fel, hogy megjelenik egy új betegség. Ekkor nemcsak a betegség nevét és az osztály azonosítóját kell bevinni, hanem az osztály nevét és főorvosát is. Lássuk, mi történik, ha megjelenik a vakbél(gyulladás), mint új betegség:

Osztály

Betegség	Oszt_az	Oszt_név	Főorvos
sérv	01	sebészet	Dr. Doktor
tyúkszem	02	szemészet	Dr Joó
vakbél	01	sebészet	Dr. Doktor

Az első és a harmadik sorban sok a közös adat. Ahhoz, hogy tudjuk, milyen betegséggel melyik osztályon fekszik a beteg, szükségünk van az osztály azonosítójára, de teljesen *felesleges* újra tárolni az osztály és a főorvos nevét. Ez sok, ugyanazon osztályra tartozó betegség esetén fölösleges tárolást jelentene. Pl. ha 5 új betegség jelenne ugyanazon az osztályon, az 10 db feleslegesen felvitt adatot jelent!
Emiatt a táblázatot érdemes két további táblázatra bontani, megszüntetve ezzel a közvetett összefüggéseket:

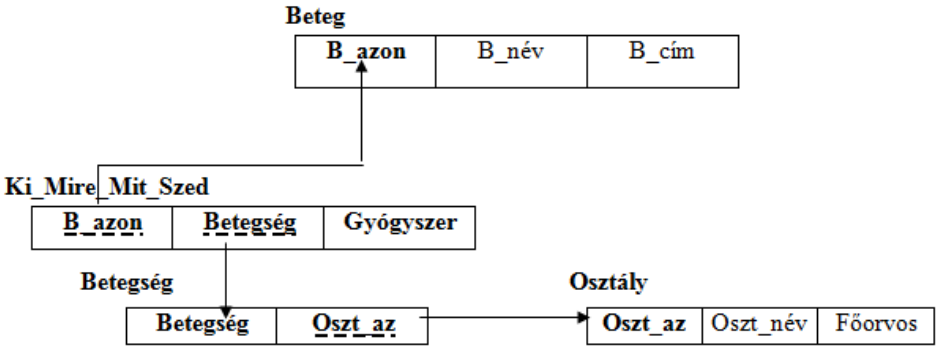
Betegség

Betegség	Oszt_az
sérv	01
tyúkszem	02
vakbél	01

Osztály

Oszt_az	Oszt_név	Főorvos
01	sebészet	Dr. Doktor
02	szemészet	Dr. Joó

A Betegség reláció elsődleges kulcsa a Betegség, idegen kulcsa az Oszt_az. Ezzel hivatkozhatunk az Osztály reláció megfelelő rekordjára. Az Oszt_az az Osztály elsődleges kulcsa.
Ábrázolva a relációk közötti összefüggéseket:



Az így kapott reláció mindegyike harmadik normál formában van, mert

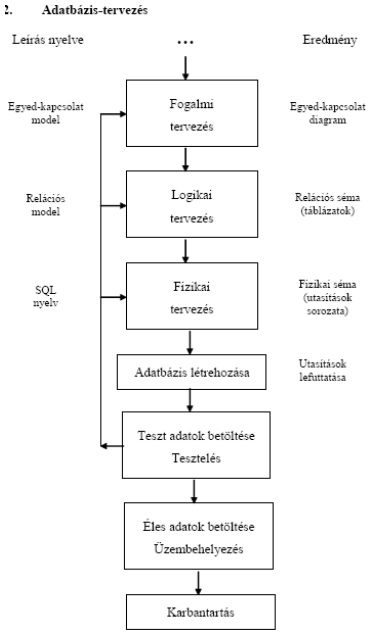
- 2NF-ben van
- funkcionális függés csak az elsődleges kulcsból indul ki; vagyis megszüntettük a közvetett (tranzitív) függéseket.

Ha most tesszük fel a korábbi kérdéseket, látható mennyivel egyszerűbben és gyorsabban kapható rájuk válasz. A gyógyszerszámláláshoz elegendő egy kisebb táblázatot végignézni, egy esetleges főorvos váltás pedig az Osztály táblázat egyetlen rekordjának módosítását jelenti.

13.4 Az adatbázis-terv dokumentációja

**** * * Az adatbázis-tervezés lépései * * ****

- *Az igények összegyűjtése, elemzése*
  Fel kell deríteni a fő alkalmazási területeket, tanulmányozni kell az adott területtel rokon, már +lévő alkalmazásokat és azok dokumentációit. A felhasználói igények, elvárások összegyűjtése érdekében célszerű a mostani és későbbi felhasználókkal elbeszélgetni. Olyan specifikációt kell készíteni, mely tartalmazza a felhasználói igényeket kielégítő tárolandó adatokat, valamint a feldolgozási műveleteket, tranzakciókat, lekérdezéseket.
- *Koncepcionális terv elkészítése*
  A terv készítésének folyamán kell a magas szintű modellt kialakítani. A modell segítségével + kell fogalmazni az előre tervezhető lekérdezéseket és tranzakciókat. Segítségével közérthető formában mutatja az adatbázis szerkezetét, az adatcsoportokat és azok kapcsolatait, +szorításait és segíti a felhasználó és a progiozó közötti párbeszédet. A koncepcionális terv elkészítéséhez leggyakrabban az egyed-kapcsolat modellt vagy a relációs modellt alkalmazzák.
- *Adatbázis-kezelő rendszer kiválasztása*
  Az adatbázis-kezelő rendszer kiválasztásában igen sok tényező játszhat szerepet, mint pl.: gazdaságossági +fontolások, a rendszer szolgáltatásai, felhasználóbarát felület, több progi is használja ugyanazt az adatbázist, az adatok sűrűn változnak, nagy az adatbázis, a különböző adattípusok között bonyolult kapcsolatrendszer áll fent, az adatbiztonságra nagy az igény.
- *Adatbázis-kezelő rendszertől függő leképezés*
  A leképezés lényegében a logikai adatmodelltől függő szabályok alkalmazása.
- *Fizikai tervezés*
  Itt kell dönteni a tárolási szerkezetről és a hozzáférési módokról, melybe az adatbázis-kezelő rendszeren kívül az operációs rendszer is beleszólhat. Relációs adatmodell használata esetén a fizikai tervesben fontos szerepet játszik az indexelés. Döntő fontosságú lehet a lekérdező és aktualizáló – vagyis a beszúró, törlő és módosító – műveletek gyakorisága és egymáshoz való viszonya. Az indexelésnél azt is figyelembe kell venni, h. az adatok elérésének gyorsítása mellett ez többlet helyfoglalással jár.



- *megvalósítás*
  Az adatleíró nyelven írt sémák alapján létrejön az adatbázis szerkezete és az üres fájlok. Az így kapott adatbázist feltölthetjük adatokkal. Készülhetnk űrlapok, lekérdezések, stb. Valódi adatok felvitele előtt célszerű a rendszert mintaadatokkal kipróbálni, h. ne túl későn derüljenek ki az esetleges hibák. A rendszer +valósítása után használat közben felmerülhetnek problémák, amiket orvosolni kell.

Az adatbázis-terv dokumentációjával kapcsolatban nem találtam semmit. Itt szokták megadni a progi kiválasztást; relációs adatmodellnél a normalizálás során létrejött táblákból a fizikai tábálk adatait, esetleg külön ábrázolva a kapcsolatokat is; képernyőtervek – lekérdezések – űrlapok.

14. SQL adatbázis, adattábla, index, nézet létrehozása és törlése. Adattábla szerkezetének módosítása. Kulcsok, külső kulcsok megadása, kapcsolatok beállítása. További megszorítások elhelyezése.

14.1 SQL adatbázis, adattábla, index, nézet létrehozása és törlése, Adattábla adatszerkezetének módosítása

SQL adatbázis: Az SQL relációs adatmodelt kezel, igy a nyelv fő objektuma a reláció, amit az SQL-ben táblának nevezünk. Táblák névvel ellátott csoportja alkot egy adatbázist. Az adatbázis olyan adatoknak a halmaza, melyeket együtt kezelünk, az adatok kapcsolataikkal együtt történő ábrázolását, tárolását jelenti.
Az adatbázis létrehozása: CREATE DATABASE adatbázisnév; Regisztrálásra kerül az új adatbázis és létrejön róla mindennemű bejegyzés a rendszer-katalógustáblákban.
Az adatbázis törlése: DROP DATABASE adatbázisnév;
START DATABASE; adatbázisnév; megnyitja a létező adatbázist (egyszerre csak egy aktív).
STOP DATABASE; Lezárja az aktív adatbázist.
SHOW DATABASE; információ a létező adatbázisokról.

Adattábla: Az adatokat tartalmzó relációt nevezzük adattáblának. A tábláknak oszlopai a tulajdonságok (attributumok), melyek típusát meg kell adni az adattábla definiálásakor,A tábla nevének az egész adatbázisban egyedinek kell lennie(max 8 karakter, betűvel kezdődik, folytatódhat betüvel, számmal, „_") Az oszlop nevének egy táblán belül egyedinek kell lennie(max10 karakter, betűvel kezdődik, folytatódhat betüvel, számmal, „_").
A tábla szerkezetének módosítása:
Adattábla definiálása:
CREATE TABLE táblanév (oszlopnév adattípus (adathossz) [,oszlopnév adattípus (adathossz)]...);

A tábla létrehozásánál a névadást követően azonnal a szerkezet megadása következik: a felsorolt oszlopoknak azonosítójuk, típusuk és hosszuk van.

SZEMELY (kod,nev,nem,anya,apa)

```
CREATE TABLE szemely
    (kod char(3),
    nev char(12),
    nem logical,
    anya char(3),
    apa char(3));
```

A tábla törlése:
DROP TABLE táblanév;

A tábla kitörlése az adatbázisból (a megfelelő katalógusbeli bejegyzések karbantartásával).

Az SQL-ben nincsen lehetőség igazi módosításra, csak új oszlopok felvételére. (egyes verziókban módosítani is lehet MODIFY)Minden más változtatást úgy tudunk elvégezni, hogy létrehozunk egy új táblát, és azt a már meglévő táblázatból töltjük fel.
Oszlop hozzáadása:
ALTER TABLE táblanév ADD | MODIFY(oszlopnév adattípus(adathossz) [,oszlopnév adattípus (adathossz)]...);

Index: Az ***indexállomány*** egy adott táblából kiemelt néhány rendezett oszlopból áll, a rendezés lehet növekvő és csökkenő) Ha az indexelt tábla értékei alapján keresünk, akkor a keresés gyorsabb lesz.
Indextáblát a következő alakú parancs hozza létre:

CREATE [UNIQUE] INDEX *indextábla -név* ON *táblanév* (oszlopnév [[ASC|DESC][,*oszlopnév*[ASC|DESC]]..);

A parancs hatása: Az ON után adott tábla felsorolt oszlopait rendezi (növekedően ASC és csökkenően DESC esetén) és belőlük egy az INDEX szó után megadott nevű táblát készít. Az UNIQUE azt jelenti, hogy az oszlop értékei egyediek, s ha ez esetben ismétlődő értékek is vannak az oszlopban, a rendszer hibát jelez. Nézet táblát nem lehet indexelni.

Indexálás törlése:

DROP INDEX Indextábla-név;

Nézet létrehozása és törlése: Nézet tábla létrehozásakor egy külön táblába definiáljuk az adatbázis azon részeit amelyre szükségünk lesz. A virtuális vagy nézet-tábla fizikailag nem jön létre, mégis táblaként kezelhető,a rendszer csak a definícióját őrzi a katalógustáblázatban. A nézet-táblat valódi táblákból származtatjuk oly módon, hogy a kijelölt táblá(k)ra lekérdezést írunk. Valahányszor a nézet-táblahoz fordulunk, a rendszer az alaptáblát kérdezi le. Nézet lehet további nézet alaptáblája is.
Nézet-tábla definiálása:
CREATE VIEW Nézetábla-név [oszlopnév-lista] AS SELECT... [WITH CHECK OPTION]

- ugyanúgy le lehet kérdezni, mintha adattábla lenne
- ha egyszerű a definíciója, akkor a sor bekerül/törlő (származtatásában nem tartalmaz DISTINCT, GROUP BY záradékot, SELECTbeli kifejezést, ill. alaptáblája nem nézet)
- adatismétlődés csökkentése miatt hasznos
- az adatbiztonság megszervezésének legegyszerőbb módja
- bizonyos felhasználók elől el kell rejteni egy tábla, különböző részeit, de bonyolultabb esetben helyesebb ideiglenes fizikai táblát létrehozni

```
CREATE VIEW nézet-táblaneve
AS
SELECT oszlop-neve FROM melyik táblákból –(*) összes
WHERE feltétel megadása;
```

A nézet-tábla törlése: DROP VIEW Nézettábla-név;

14.2 Kulcsok, külső kulcsok megadása, kapcsolatok beállítása.

Kulcsok: Egy tulajdonságot (attribútum) vagy tulajdonságok egy csoportját kulcsnak nevezzük, ha a tábla mindegyik sorát egyértelműen meghatározza, de ha bármely attribútumot elhagynánk a kulcsból, akkor az így megmaradt oszlopok kombinációja már nem tudná egyértelműen meghatározni azt. Vagyis a kulcsban az attributumok értékei különböznek, nics két olyan sor amelyben megegyeznének. Ha a kulcs egyetlen attributumból áll, akkor a kulcsot **egyszerű kulcs**nak nevezzük, ha két vagy több oszlop kombinációja alkotja, akkor **összetett kulcs**nak.

Elsődleges kulcs (primary key): az a kulcs, melyet a kulcsjelöltek közül választunk ki, és kulcsként használjuk. A ki nem választott kulcsjelölteket alternatív kulcsnak nevezzük.Az elsődleges kulcsnak nem lehet NULL az értéke.

Külső kulcs (idegen kulcs): A tábla azon attributumai, amelyek egy másik táblában kulcsot alkotnak. A külső kulcs nem azonosítja a rekordokat, nem valódi kulcs, csak a táblák sorai közti kapcsolatot fejezi ki.

Ez azt jelenti, hogy relációs modellben, amikor a redundancia megszüntetése miatt több kisebb táblára fogjuk bontani az adatbázisunkat, az egyes táblák közt fennálló kapcsolatokat úgy érjük el, hogy az egyik tábla kulcsát bevisszük a másik tábla oszlopai közé. Azt is mondhatnánk, hogy a külső kulcs értékei egyértelműen fognak rámutatni egy másik tábla valamely sorára a kulcs alapján.

Gyakran hívjuk gyerektáblának azt, ahol külső kulcsként szerepel egy másik tábla - az ő szülőtáblájának - kulcsa.

elsődleges kulcs megadása:

CREATE TABLE táblanév (attrib1 típus(n) ... PRIMARY KEY (attrib1, ...));
PRIMARY KEY kulcsszóval az elsődleges kulcsot vagy UNIQUE kulcsszóval egyedi kulcsot hozunk létre. Csak egy elsődleges, de akár több egyedi kulcsa lehet a táblának! Ezek a kulcsok természetesen nem vehetnek fel ismeretlen értéket (NOT NULL).
Amikor a kulcs egyszerű, akkor az őt alkotó attribútum mögött is megadhatjuk megszorításként,

```
CREATE TABLE Dolgozó
 ( adószám   INT(10)  PRIMARY KEY,
   név       CHAR(30));
```

de összetett kulcs esetén az attribútumok felsorolása után sorolhatjuk fel az elsődleges kulcs összetevőit zárójelek között. Az adatbázis-kezelőtől függ, hogy az egyedi kulcsok definiálásakor létrehozza-e az indexet; az elsődleges kulcshoz biztosan létrehozza.

idegen kulcs definiálása:

CREATE TABLE gyerek-tábla (attrib1 típus(n)... FOREIGN KEY (attribútumok) REFERENCES (szülő-tábla));
A hivatkozási épség fenntartása érdekében adjuk meg a külső kulcsokat. A külső kulcs szerkezetének azonosnak kell lennie azon elsődleges kulcséval, melyre hivatkozik (a kulcsot alkotó oszlopnevek egyezése nincs kikötve). Természetesen lehet több idegen kulcs is a táblában.

Kapcsolatok beállításai: Kapcsolat (relationship): Az egyedek vagy tulajdonságaik közötti viszony. A kapcsolatokat megkülönböztethetjük annak megfelelően, hogy az egyedhalmazok közötti viszonyt vizsgáljuk, vagy az egyes egyedek tulajdonsághalmazai közötti viszonnyal foglalkozunk. Az egyedhalmazok közötti kapcsolat, a táblák (relációk) közötti kapcsolatban fog megjelenni. Az egyedhalmaz tulajdonsághalmazai közötti kapcsolatokat pedig a relációs modellnél vizsgáljuk, amikor meghatározzuk a funkcionális függőséget.
REFERENCES esetén ON-feltételek megadásával szabályozhatjuk a rendszer viselkedését. A következőkben
T a hivatkozó, S a hivatkozott táblát jelenti megszorítás típusa: (RESTRICT, SET NULL, CASCADE) Műveletek: (UPDATE, DELETE)

- *Alapértelmezés (ha nincs ON-feltétel) :* A hivatkozó táblában nem megengedett olyan beszúrás és módosítás, amely a hivatkozott táblában nem létező kulcs értékre hivatkozna, továbbá a hivatkozott táblában nem megengedett olyan kulcs módosítása vagy sor törlése, amelyre a hivatkozó tábla hivatkozik.
- *ON UPDATE CASCADE :* Ha a hivatkozott táblában változik a kulcs értéke, akkor a hivatkozó tábláében is.
- *ON DELETE CASCADE :* Ha a hivatkozott táblában törlünk egy sort, akkor a hivatkozó táblában is törlődnek a kérdéses sorok.
- *ON UPDATE SET NULL :* Ha a hivatkozott táblában változik a kulcs értéke, akkor a hivatkozó táblában NULL lesz.
- *ON DELETE SET NULL :* Ha a hivatkozott táblában törlünk egy sort, akkor a hivatkozó táblában NULL lesz.

```
CREATE TABLE Dolgozó
  ( adószám    INT(10)   PRIMARY KEY,
    név        CHAR(30),
    nem        CHAR(1)   CHECK (nem IN ('F', 'N')),
    osztálykód CHAR(10)  REFERENCES Osztály(osztálykód)
                         ON UPDATE CASCADE
                         ON DELETE SET NULL );
```

14.3 További megszorítások elhelyezése.

Attribútum-értékekre vonatkozó korlátozás: A CREATE TABLE attribútumai mögött NOT NULL CHECK (feltétel) záradékkal gondoskodhatunk az oszlopérték helyességéről. DEFAULT érték definiálásával pedig az alapértelmezés szerinti érték adásáról gondoskodhatunk.

Önálló megszorítások. Az attributumra és sorra vonatkozó megszorításokat a rendszer csak akkor ellenőrzi, ha az attr. vagy reláció, melyre a feltétel vonatkozik, beszúrás vagy módosítás hatására változik meg. Ezért, hogy a több sort/ táblát érintő feltétel minden esetben igaz maradjon, önálló megszorításként adjuk meg.

CREATE ASSERTION megszorítás neve CHECK (feltétel)

Ezek az önálló feltételek elkülönülnek a táblák definíciótól, és egy vagy több tábla összefüggéseit szabályozzák. E feltételeket a rendszer mindannyiszor megvizsgálja, valahányszor beszúr/mód/törl történik az érintett táblák bármelyikében.

DROP ASSERTION megszorítás neve Kitörli az adott nevű önálló megszorítást, majd újra definiálható.

Triggerek : *Fentebb már említettük, hogy ezek különleges tárolt eljárások, mert az adatbázisban egy feltétel bekövetkezésétől függően „indítódik" a trigger. A trigger tehát alkalmas adathibák, hivatkozási referenciák megsértésének kiszűrésére. A megszorítások megsértésének megakadályozásán túl más célok is megvalósíthatók: adott feltételtől függően végrehajtódjanak a benne levő adatbázis-műveletek, vagy semmi se történjen. CREATE TRIGGER tr_név*

Egy oszlopra vonatkozó megszorítások

NULL az attribútum definíciójában arra utal, hogy az adat megadása nem kötelező, ez az alapértelmezés ezért a legritkább esetben írják ki.
NOT NULL az attribútum definíciójában arra utal, hogy az adat megadása kötelező, azaz nem vihető be olyan sor a relációban, ahol az így definiált adat nincs kitöltve.
PRIMARY KEY ez az oszlop a tábla elsődleges kulcsa.
UNIQUE ez az oszlop a tábla kulcsa.
CHECK(*feltétel*) csak feltételt kielégítő értékek kerülhetnek be az oszlopba.
[**FOREIGN KEY**] **REFERENCES** *reláció_név* [(*oszlop_név*)], ez az oszlop külső kulcs

Több oszlopra vonatkozó megszorítások

PRIMARY KEY(*oszlop1*[, *oszlop2*, ...]) ezek az oszlopok együtt alkotják az elsődleges kulcsot.
UNIQUE(*oszlop1*[, *oszlop2*, ...]) ezek az oszlopok együtt kulcsot alkotnak.
CHECK(feltétel) csak feltételt kielégítő sorok kerülhetnek be a táblába.
FOREIGN KEY (*oszlop1*[, *oszlop2*, ...]) **REFERENCES** *reláció*(*oszlop1*[, *oszlop2*, ...]), az oszlopok külső kulcsot alkotnak a megadott tábla oszlopaihoz.

Példák

```
CREATE TABLE osztaly (
  oszt_azon      INT2 PRIMARY KEY,
  nev            VARCHAR(14) NOT NULL,
  varos          VARCHAR(13) NOT NULL,
  CONSTRAINT  unev UNIQUE(nev));

CREATE TABLE alkalmazott (
  alk_azon     INT2 PRIMARY KEY,
  nev          VARCHAR(10) NOT NULL,
  beosztas     VARCHAR(9) NOT NULL,
  fonok        INT2 CONSTRAINT fonok_korl REFERENCES alkalmazott (alk_azon),
  belepes      DATE NOT NULL,
  fizetes      FLOAT4 NOT NULL,
  jutalom      FLOAT4,
  oszt_azon    INT2 NOT NULL,
  CONSTRAINT alk_kulso_kulcs FOREIGN KEY (oszt_azon) REFERENCES
               osztaly (oszt_azon));

CREATE TABLE fiz_oszt (
  f_oszt         INT2 PRIMARY KEY,
  min            FLOAT4 NOT NULL,
  max            FLOAT4 NOT NULL,
  CONSTRAINT  min_max CHECK (min < max));
```

15. SQL adattábla sorainak felvitele, módosítása, törlése. Megszorítások figyelembe vétele felvitel/módosítás/törlés esetén. Jogok kiosztása és visszavételezése.

15.1 SQL adattábla sorainak felvitele, módosítása, törlése

Sorok bevitele: INSERT INTO táblanév [(oszlopnév-lista)] VALUES (értéklista); Ha az oszlopnév lista elmarad, akkor a tábla definiálásakor megadott oszlop sorrendben kell az értéklista értékeit megadni. Az új sor fizikailag a tábla utólsó sora utáni sorban tárolódik.

A tábla rekordjainak (sorok) módosítása:

UPDATE táblanév SET oszlopnév=kfejezés[,oszlopnév=kfejezés,…] [WHERE logikai kifejezés]; A felsorolt oszlopok értékeit a megadott kifejezéssel módosítja, de csak azokra a sorokra, amelyek eleget tesznek a WHERE-ben szereplő logikai kifejezésnek.(Ha a WHERE elmarad, akkor az oszlop minden sorát módosítja)

Rekordok törlése: DELETE FROM táblanév [WHERE feltétel]; Törli a táblából azokat a sorokat amelyekre teljesül a feltétel. Ha a WHERE elmarad, akkor az oszlop minden sorát módosítja.

15.2 Megszorítások figyelembevétele felvitel/módosítás/törlés esetén.

A megszorítások olyan előírások, korlátozások, amelyekkel megadhatjuk az adatbázis tartalmára vonatkozó kívánságainkat. A mogszorításokat az adatbázisrendszer minden olyan akció során ellenőrzi, amely eredményeként az adatbázis tartalma módosul. ***A megszorítások lényege a hivatkozási épség(kulcsok, külső kulcsok) és az adatösszefüggések(pl: nemlehet NULL, értéktartományon belül) felügyelete minden adatkezelés alkalmával.*** A megszorítások megadásuktól kezdve érvényesülnek, nincs visszamenőleges hatásuk. Késleltetett ellenőrzés végrehajtása a DEFFERRED kulcsszóval. A megszorítás típusok:

- Kulcsok: megszorításként azt jelenti, hogy ellenőrizze, hogy a táblában ne legyen olyan sor, amelyben a kulcsattributomok értéke azonos lenne. Idegen kulcsok hivatkozási épségének biztosítása.
- Attribútumértékekre vonatkozó megszorítások: Az attributum lehetséges értékeinek korlátozása.(Hibás adat bevitelének, adat hibás adatra módosításának megakadályozása, értéktartomány megadása (NOT NULL, CHECK).
- önálló megszorítások: bármilyen feltétel ellenőrzése.

15.3 Jogok kiosztása és visszavételezése.

Az SQL kiköti a felhasználói nevek létezését, amiket fel lehet ruházni különféle jogokkal. Ilyen jogok az adattáblákra vagy nézettáblákra vonatkozó lekérdezés és szerkezeti módosítás, a 3 karbantartó utasítás (felvitel, módosítás, törlés), illetve valamely táblára való hivatkozás megszorító feltételekben, vagy az indexelés. Minden SQL-ben létrehozott objektum tulajdonosa megadhatja, mely felhasználók, milyen műveleteket végezhetnek az adott objektumon. Az így megadott jogok visszavonhatók. A jog tárgyát képező objektum megszüntetése maga után vonja az összes jogosultság megszűntét.

A táblákra vonatkozó jogosultság adományozása

A parancs formája:

GRANT ALL [PRIVILEGES] | *jogosultséglista* ON [TABLE] *táblalista*
TO PUBLIC | *felhasználólista* [WITH GRANT OPTION];

A parancs minden jogot (ALL PRIVILEGES) vagy a jogosultséglistában szereplô mûveletekre való jogot adja a táblalistában szereplô táblákra mindenkinek (PUBLIC esetén) vagy a felhasználólistában szereplő személyeknek. Amennyiben a WITH GRANT OPTION szerepel, akkor az e jogokat kapók át is adhatják ezeket a jogokat másoknak.
A jogosultséglista elemeit a következô táblázatban foglalhatjuk össze:

A jogosultság neve:	A jogosultság jelentése:
ALTER	Jogosultság a tábla módosítására
DELETE	Jogosultság a tábla törlésére
INDEX	Jogosultság indextábla létrehozására
INSERT	Jogosultság új sor felvételére a táblázatba
SELECT	Jogosultság lekérdezésre
UPDATE	Jogosultság a tábla módosítására

Jogosultság adományozása az adatbázison végzett mûveletekre

A parancs formája:

GRANT*adatbázisjog* TO PUBLIC/*felhasználólista;*

A parancs jogosultságot ad az adatbázisra vonatkozóan vagy mindenkinek (PUBLIC) vagy adott felhasználóknak a felhasználólista szerint. Az adatbázisjogokat a következô táblázatban foglalhatjuk össze:

A jog neve:	A jog jelentése:
CONNECT	- Hozzáférés a teljes adatbázishoz - Jog arra, hogy SELECT, INSERT, DELETE, UPDATE műveleteket végezzen más felhasználók tábláin, ha ilyen jogosultságot kapott a táblákra vonatkozó GRANT-tal - Jog nézettáblák és szinonim táblák létrehozására.
RESOURCE	- Minden CONNECT jogosultság - Jogosultság táblák és indextáblák létrehozására, jogosultságok adományozása ezekre a táblákra
DBA	-Teljes adatbázis-adminisztrátori jogkör

Táblákra vonatkozó jogosultság visszavonása
A parancs formája:

REVOKE ALL [PRIVILEGES] | *jogosultséglista* ON [TABLE] *táblalista* TO PUBLIC |*felhasználólista* ;

A parancs hatása: Az összes jogosultságot (ALL PRIVILEGES vagy csak a jogosultséglistában felsoroltakat a megadott táblákra vonatkozóan mindenkitôl (PUBLIC) vagy csak a listában szereplô felhasználóktól visszavonja.

Adatbázis-jogosultságok visszavonása
A parancs formája:

REVOKE *adatbázisjog* FROM PUBLIC | *felhasználólista* ;

A parancs adatbázisjogokat mindenkitől (PUBLIC) vagy a listában szereplőktől visszavonja.

16. Lekérdezés összeállítása és végrehajtása az SQL-ben. Belső lekérdezés beépítésének lehetőségei a lekérdező parancsba.

16.1 Lekérdezés összeállítása és végrehajtása az SQL-ben.

A lekérdezés hatására egy úgynevezett eredménytábla (E-tábla) jön létre, amelynek egy vagy több oszlopa és egy vagy több sora lehet. Az E-tábla csak ideiglenesen jön létre elmentéséről gondoskodni kell. A SELECT paranccsal kiadható lekérdezések típusát az alparancsok, valamint az alparancsok operandusai adják meg, szerkezete kötött, az alparancsok csak megfelelő sorrendben irhatóak:

SELECT…

INTO	Az E-tábla egy sorának tárolása
FROM	Descartes-szorzat
WHERE	szelekcio,sorok kiválasztása
GROUP BY	csoportositas
HAVING	csoportok közötti választás
UNION	E-táblák összefűzése(unió művelet)
ORDER BY	E-tábla rendezése
SAVE TO TEMP	E-tábla megőrzése, elmentése

```
SELECT [ALL/DISTINCT] oszlopnév [,oszlopnév…]|*
        FROM táblanév [,táblanév…]
          WHERE feltétel
            GROUP BY oszlopnév [,oszlopnév…]
              HAVING feltétel
UNION
SELECT [ALL/DISTINCT] oszlopnévlista|*
        FROM táblalista
          WHERE feltétel
            GROUP BY oszlopnév [,oszlopnév…]
              HAVING feltétel
UNION
  ORDER BY oszlopnév[ASC|DESC];
```

A **SELECT** után mindig kell **FROM**, ugyanis az utána lévő táblalistában szereplő táblákból történik az oszloplistában lévő oszlopok kiemelése (ha az oszloplista helyén * van akkor az osszes oszlopon). A DISTINCT hatása: az E-tábla a duplikált sorokat csak egyszer tartalmazza, Az ALL eseténpedig az azonosakat is annyiszor tartalmazza ahányszor szerepelnek(ez az alapértelmezett).

Ha a sorok közül kell válogatni, akkor a **WHERE** mögött adjuk meg a szelekció feltételét, ami egy logikai kifejezés, tehát azok a sorok kerülnek majd megjelenítésre, amikre a feltétel igaz.

A **GROUP BY** alparancs: oszlopok értékei alapján csoportokatkat képez (egy csoportba az azonos értékűek tartoznak).

```
SELECT osztaly, AVG(joved)
FROM tanulo
GROUP BY osztaly;
OSZTALY     AVG1
1A          24160,00
1B          18350,00
2A          33166,67
2B          27133,33
2C          34311,43
Osztályonként mekkora az egy főre eső jövedelmek átlaga.
```

A **HAVING** alparancs: A GROUP BY csoportjaiból csak azok a sorok kerülnek az E-táblába, amelyek eleget tesznek a feltételnek.

Az **UNION** alparancs(művelet): A SELECT-ekkel létrehozott táblákat összefűzi (egymás alá teszi). Azok a sorok, amelyek közösek, csak egyszer jelennek meg (unió). Az E-tábláknak kompatibilisnek kell lenniük:azonos oszlopszámok, az oszlopok azonos típusúak (az oszlopneveknek nem kell megegyezniük).

A **ORDER BY** alparancs: Megadott oszlop vagy oszlopok szerint rendezi az E-táblát, növekvően (ASC-alapértelmezett) vagy csökkenően (DESC).

16.2 Belső (beágyazott) lekérdezés lehetősége.

A WHERE és a HAVING feltételében SELECT parancs is szerepelhet, amit belső vagy beágyazott lekérdezésnek nevezünk, de nem tartalmazhat ORDER BY és UNION parancsokat és a GRUOP BY, SAVE TO TEMP alparancsok is csak egyszer fordulhat elő a teljes SELECT-ben. A külső SELECT a belső E-táblájából hozza létre az E-táblát. A belső SELECT is tartalmazhat belső SELECT-et, vagyis a SELECT-ek egymásba ágyazhatóak. A belső (vagy al-) SELECT-et mindig zárójelpárban kell megadni. A kiértékelés menet:

- a belső SELECT kiértékelődik és egy, vagy több sort vagy oszlopértéket átad a külső SELECT-nek
- a külső SELECT ezen értékek alapján összeállítja az eredményt.

<u>Egyetlen értéket tartalmazó belső E-tábla</u>: Ha a belső E-tábla egyetlen értéket tartalmaz, akkor a WHERE parancsrészben egyszerű összehasonlításokat végezhetünk.

Jelenítsük meg azoknak a dolgozóknak a nevét, fizetését akiknek a fizetése kisebb az átlagfizetésnél:

```
SELECT vnev+knev,fiz
  FROM dolgozó
    WHERE fiz<(SELECT AVG(fiz) FROM dolgozo);
```

<u>Több értéket tartalmazó belső E-tábla</u>: Amikor a belső SELECT-nek több értéke van, akkor a WHERE utasításrészben 4 féle „halmazos” logikai feltétel szerepelhet:

IN predikátum(tulajdonság): Igaz értéket ad vissza, ha az egyenlőség valamelyik belső E-táblabeli értékre igaz.

```
Az előző példa módosított változata helyesen:
SELECT nev,osztaly
    FROM tanulo
    WHERE azon IN (SELECT azon
                FROM segely
                WHERE osszeg>=(SELECT AVG(osszeg)
                            FROM segely));
NEV             OSZTALY
LOVAS LAJOS     1B
VARGA TERÉZ     2A
BALOG HMIHÁLY   2B
Azért optimális ez a megoldás, mert kiküszöbölt az előzőekhez képest egy Descartes-szorzat előállítást és lekérdezést. Valóban egy logikailag SELECT-ek közül a második összetett lekérdezés, ennek is egyszer hajtódik végre az alSELECTje, de ez már itt több értéket ad vissza, és mivel csak a TANULÓból jelenít meg, nem kellett a Descartes-szorzat.
```

ANY predikátum: Igaz, ha a megadott összehasonlítás valamelyik belső E-táblabeli értékre igaz.
ALL predikátum: Igaz értéket ad vissza, ha a megadott összehasonlításás valamennyi belső E-táblabeli értékre igaz.
EXISTS predikátum: Kiválasztja azon sorokat, amelyekhez a belső E-táblában egy vagy több sor tartozik.

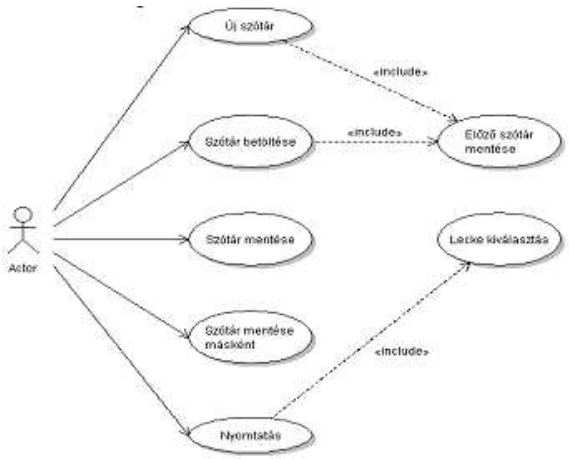
17. Modellező nyelvek és eszközök szerepe az alkalmazások tervezésében és dokumentálásában. UML diagramok: használati eset diagram, objektumdiagram, kommunikációs diagram, állapot diagram, osztálydiagram és osztályleírás, komponens diagram.

17.1 Modellező nyelvek és eszközök szerepe az alkalmazások tervezésében és dokumentálásában

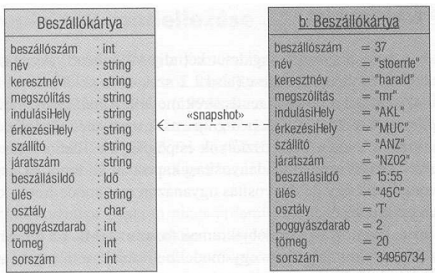
Előzmények: *A szoftverek bonyolultságának, komplexitásának növekedésével és az objektumorientált nyelvek megjelenésével vált szükségessé egy mindenki által használható tervezési módszer kialakítása, elemzési módszerek, egységes nyelvek, jelölésrendszerek kidolgozása. A '90-es évek elejéig több javaslat is született, ezek többsége azonban nem felelt meg az igényeknek. A nagy áttörést 1997-ben az UML modellezőnyelv, illetve 1998-ban a RUP hozta.* Az **UML** egy szabványos, egységesített modellezőnyelv, amelynek segítségével a tervezés, a specifikáció, a dokumentálás mind grafikus formában, beszédes ábrák, diagramok, táblázatok segítségével végezhető. Olyan módszer, amely képes kezelni ezt a sokrétűséget, viszont kellően egyszerű ahhoz, hogy széles körben elterjedjen. A tervezési folyamatban az UML-t a kiindulási fázisban használjuk, azzal a célal, hogy minél biztonságosabban, áttekinthetőbben és megbizhatóbban készítsük el a teljes szoftver tervét, az objektumorientált fejlesztési elvhez kapcsolódóan. Az UML nem más, mint egy tevezési nyelv, ami egy szoftver rendszer minél megalapozottab kidolgozásának előkészítésének folyamatát szolgálja. Az igy készült tervezési diagramok alapján válik lehetővé a forráskód megírása és a futtatható szoftver alkészítése, valamint segítségével a későbbiekben is könnyebben módosíthatjuk a programot.

17.2 UML diagramok: használati eset diagram, objektumdiagram, kommunikációs diagram, állapot diagram, osztálydiagram és osztályleírás, komponens diagram.

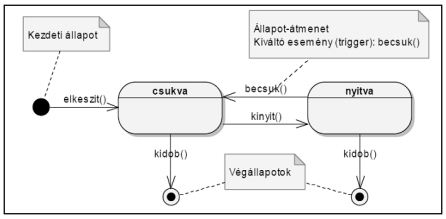
Használati eset diagram: A HE diagram a rendszer viselkedésének egy kiragadott részét írja le külső aktorok szemszögéből. A HE diagram elemei: aktorok, használati esetek, valamint az ezek közötti kapcsolatok. Egy aktor üzeneteken keresztül kommunikál a rendszerrel. Az aktor üzenetet küld a rendszernek, a rendszer pedig a használati eset végrehajtása közben üzeneteket küldhet vissza az aktor számára. Az aktor üzeneteinek és a rendszer válaszainak megadásával a rendszer határait húzzuk meg. A használati eset a szoftver használatának egy értelmes egysége, az aktor kommunikációja, párbeszéde a szoftverrel. A használati eset a rendszer viselkedését írja le a rendszeren kívülről. Felhasználói célnak nevezzük a felhasználónak azt a célját, melyet a szoftver használatával szeretne elérni. A felhasználói célok használati esetekre bontandók: a cél elérése, illetve a feladat megoldása érdekében a felhasználó konkrét használati eseteket hajt végre. A használati esetek a szoftverbe előre beépített lehetőségek, melyeket a felhasználó (aktor) indítványozhat. Egy forgatókönyv (eseményfolyam) a használati eset egy konkrét végrehajtása (példánya). Egy használati esetnek elvileg számtalan forgatókönyve lehetséges. A használatieset-modell a teljes rendszer viselkedésének a leírása. A HE modell alapján a felhasználó érti, hogyan kell használni a szoftvert. A fejlesztés, HE centrikus; a használati esetek a teljes fejlesztés során központi szerepet játszanak. A HE modell elemei: HE diagramok és leírások.



Objektumdiagramok: A modellezett rendszer egy adott időpillanatbeli állapotát mutatják az objektumdiagramok. Az objektumdiagram pillanatfelvétel a rendszer állapotáról. Osztályok példányait és kapcsolatait jeleníti meg. Az objektumdiagram konkrétabb az osztálydiagramnál, mert objektumok példányainak a kapcsolatát írja le objektumosztályok kapcsolata helyett.

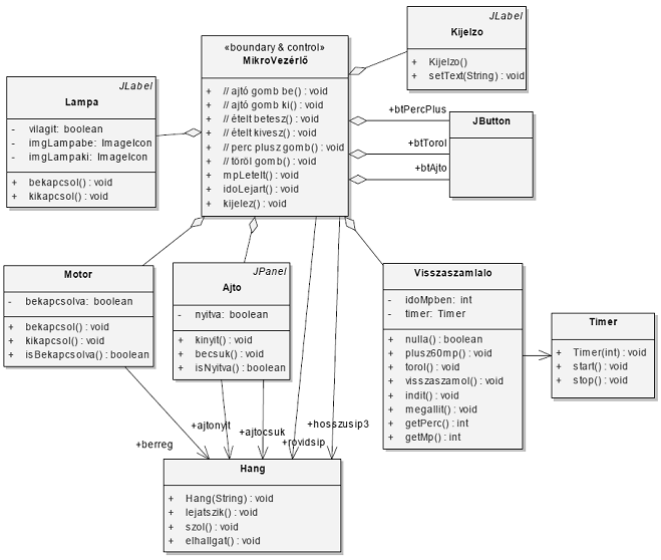


Állapot diagram: Az állapot-átmeneti diagram (state transition diagram) egyetlen osztály (annak egy előfordulásának) dinamikus viselkedését, a külvilággal való kapcsolatát ábrázolja. Az állapot-átmeneti diagram egy gráf, melynek csomópontjai állapotok, élei pedig átmenetek. Megadja, hogy az objektum mely eseményekhatására milyen állapotból milyen állapotba kerül. Egy adott állapotban levő objektum ugyanarra az eseményre mindig ugyanúgy reagál (ugyanazt az akció sorozatot hajtja végre). Az állapot az objektum életének egy szakaszát írja le. Az állapot jele az UML-ben egy lekerekített téglalap, melynek részei (bármelyik rész elhagyható): Állapot-átmenetnek nevezzük azt a folyamatot, melyben az objektum egy adott állapotából egy egyértelműen megkülönböztethetı másik állapotba kerül. Az átmenet lehetséges tulajdonságai:

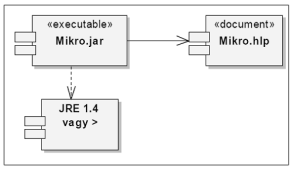


Kiváltó esemény (trigger vagy event): Az átmenet a kiváltó esemény hatására következik be. Az esemény egyaránt jöhet kívülről vagy belülről.
Őrfeltétel (guard): Az őrfeltétel egy logikai kifejezés, amely hivatkozhat az objektum adataira. Az átmenet csak akkor következik be, ha az őrfeltétel igaz.
Akció (action): Átmenetkor végrehajtódik.
Az Ajtó osztályból hozzunk létre egy példányt. Életét egy állapot-átmeneti diagrammal szemléltetjük. Születéskor –, amikor elkészítik – csukva van. Aztán egész életében nyitják-csukják. Az ajtót ki lehet dobni akár nyitott, akár csukott állapotban.

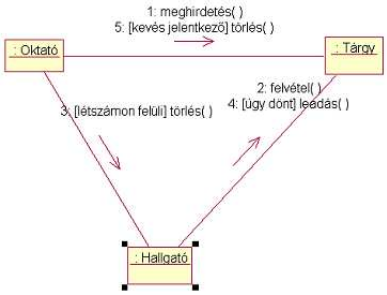
Osztálydiagram Osztálydiagram (class diagram): Olyan diagram, amely az osztályokat és a közöttük lévő társítási és öröklési kapcsolatokat ábrázolja. Az objektumdiagram az osztálydiagram elő fordulása, példánya. Az osztálydiagram rögzíti az objektumok közötti kapcsolatok szabályait. Két osztály közötti társítási kapcsolat főbb jellemzői: ismeretségi vagy tartalmazási kapcsolat (ha tartalmazási kapcsolat, akkor erős vagy gyenge); multiplicitás (egy-egy, egy-sok vagy sok-sok jellegű; kötelező vagy opcionális); szerepnév; megszorítás.



Komponens diagram: A komponensdiagram a rendszert alkotó fizikai komponenseket (szoftverelemeket) és az azok közti kapcsolatokat ábrázolja. A komponensdiagramon megadható a logikai nézet osztályainak forráskomponensekhez való hozza rendelése, valamint a forráskódok hozza rendelése futtatható komponensekhez. A komponensdiagram az implementációs nézet jellemző diagramja. A komponens (component) egy fizikailag bonthatatlan szoftver egység. A komponens lehet egy állomány, például forráskód, szerkesztendővagy futtatható szoftver elem: lefordított tárgykód,
bájtkód, futtatható program vagy dinamikus könyvtár (DLL). Mint más diagramelemek, a komponensek is csomagokba csoportosíthatók.



Együttműködési diagram – Kommunikációs diagram: Az együttműködési diagram (collaboration diagram) az üzeneteket küldő és fogadó objektumok kapcsolatát és a közöttük lezajló üzenetváltás strukturális szerkezetét ábrázolja



18. A szerveroldali programozás alapelemei az Internetes alkalmalmazások fejlesztésénél. A szerveroldali objektumok bemutatása. Adatbázis-kezelés webűrlapokkal. Egy szerveroldali programnyelvnyelv rövid bemutatása, jellemzése.

18.1 A szerveroldali programozás alapelemei az Internetes alkalmazások fejlesztésénél.

Ahhoz, hogy dinamikusan frissüljenek a weblapok - például az adatok változásának megfelelően - elengedhetetlen, hogy a web szerver ne egyszerően a tárolt lapokat küldje le a böngészőnek, hanem ezeket a lapokat dinamikusan állítsa elő. Ez gyakorlatilag annyit jelent, hogy a HTML forráskódot kell előállítani. A korai próbálkozások idején ezeket a HTML fájlokat teljes egészében szerver oldali programok segítségével írták, de a grafikus tartalmak megnövekedésével és a bonyolultság növekedésével már az a helyzet alakult ki, hogy néhány százaléknyi dinamikus tartalomért az egész oldalt újra és újra elő kellett állítani a HTTP kérés befutása esetén. Több próbálkozás történt, hogyan lehetne összeépíteni a dinamikus tartalmat és a statikust és a megoldást olyan programnyelvek jelentették, amelyek egyrészt, mint szkript nyelvek lehetővé tették a vizuális szerkesztést a statikus tartalom mellett, másrészt az ezeket a programnyelveket értelmező interpreter képes volt csak a dinamikus tartalomnak megfelelő részek értelmezésére és a leküldésre kerülő fájlban az összeillesztésére.
Az erre felkészített web szerver a lekért fájlról első lépésben eldönti, hogy csak statikus tartalmat hordoz, vagy dinamikus elemek, részletek is vannak benne. Ha a tartalom csak statikus, akkor kiszolgálja, ha dinamikus részt is tartalmaz, akkor a fájlt először átadja a megfelelő feldolgozó programnak, konténernek, ahol a dinamikus tartalmaz reprezentáló kódrészletek kerülnek feldolgozásra. Amennyiben a dinamikus tartalom HTML/XHTML kimenetet generál, ez bekerül a fájlba és ezzel kiegészítve kerül le a klienshez.

A egyik legelterjedtebb szerveroldali programozási technológia az **ASP** (**Active Server Pages**), tulajdonképpen egy HTML kódba ágyazott, speciális programozási módszerről beszélünk. *(Fontos, hogy az ASP nem egy programozási nyelv, hanem csak egy keretrendszer- futtató környezet)*. Dinamikus weboldalak készítését lehetővé tévő osztályok és komponensek együttese. Az ASP.NET-tel a szerveren (tipikusan egy webszerver vagy Internet információs szerver) tárolt adatokból állítunk elő HTML-oldalakat, amit a kliensek könnyen megjeleníthetnek -- használjanak bármilyen webböngészőt. Az ASP Az ASP oldalak írását számos beépített objektum teszi egyszerűvé. Minden objektum gyakran használt funkciók egy csoportját valósítja meg, melyek hasznosak dinamikus oldalak fejlesztésénél. ASP 2.0 -tól 6 ilyen beépített objektum van: **Application**, **ASPError**, **Request**, **Response**, **Server** és **Session**. A legtöbb ASP alkalmazást VBScriptben írták, de bármely más Active Script motor is kiválasztható.
A felhasználó tevékenységét egy eseményvezérelt modellen alapuló mechanizmus segítségével dolgozzák fel. Ha például a felhasználó egy nyomógombot megnyom, vagy egy szövegbeadási mezőt kitölt, vagy épp egy rádiógomb állapotát megváltoztatja, akkor egy ehhez tartozó eseménykezelő fog lefutni a szerveroldalon. A szerveroldalon ezeket az eseménykezelőket megírhatjuk akár C#, akár Visual BASIC.NET, akár más .NET alapú nyelveken. Az ASP lényegében nem egy különálló nyelv, hanem egy technológia melyet szinte bármilyen nyelvel együtt használhatunk melyet az ASP/ASP .NET támogat).

Az ASP oldal végrehajtásakor a webkiszolgáló végigszalad az oldal tartalmán, és ha abban ASP scriptrészletet talál, végrehajtja. (Az ASP dokumentum tartalmazhat HTML elemeket, szerveroldali scriptek, kliensoldali scriptek, egyéb komponensek (ActiveX, Java) A HTML oldal és a script által esetleg visszaadott kódrészletek együttesen képezik az eredményt, amit azután az IIS elküld a böngészőnek.
Lássunk egy példát:

```
<HTML><HEAD><TITLE></TITLE></HEAD>
<BODY>
<%
Response.Write("<center>Hello World!</center>")
%>
</BODY>
</HTML>
```

A HTML kód belsejében található <% és %> jelzi az ASP kód kezdetét és végét. A köztük található kódrészlet elvileg soha nem jut el az ügyfélhez, csakis a kód futtatása során keletkező kimenet *(ami esetünkben a Response.Write() metódusnak átadott szövegrész)*.
Az eredmény:

```
<HTML><HEAD><TITLE></TITLE></HEAD>
<BODY>
<center>Hello World!</center>
</BODY>
</HTML>
```

Az ASP kód által generált kimenet tehát összemosódott a HTML kóddal. Ez jó, hiszen ráérünk az oldalt teljes egészében elkészíteni egy külső, kényelmes HTML szerkesztővel, majd utólag beágyazhatjuk az ASP kódot.
Az ASP kódok beágyazása:
Az ASP scripte*(ke)*t az oldalba több módon is beágyazhatjuk. Lássuk mindenekelőtt a HTML szabványnak megfelelő módot:

```
<HTML><HEAD><TITLE></TITLE></HEAD>
<BODY>
<SCRIPT runat="server" language="vbscript">
```

Response.Write(" Hello World! ")
</SCRIPT>
</BODY>
</HTML>
A SCRIPT HTML elem segítségével tehát ugyanúgy ágyazhatunk be kiszolgálóoldalon futó kódot, mintha ügyféloldali scriptet írnánk – csak adjuk meg a runat="server" attribútumot. Hasonlóan az ügyféloldali megoldáshoz, természetesen kiszolgálóoldalon sem muszály az oldalon belül megírni a scriptet, megadhatunk fájlnevet is *(src attribútum)*:
<SCRIPT runat="server" src="scfile.vbs">
Látható, hogy itt elhagytuk a scriptnyelv meghatározását. Ebben az esetben a kiszolgáló az alapértelmezett scriptnyelvet használja. Ezt két helyen határozhatjuk meg: egyrészt, az adott .asp oldal tetejére írt, úgynevezett ASP direktíva segítségével :
<%@ Language=VBScript %>
Ha pedig ez hiányzik, a kiszolgáló a saját beállításait használja,

Az .asp kiterjesztésű fájl HTML elemeket, kliens- és szerveroldali scripteket, és egyéb komponenseket(ActiveX elemek, Java kisalkalmazások) tartalmazhat.
Folyamata: a böngészőből egy .asp fájlt indítunk(lehet begépelve, vagy egy másik folyamat eredményeként), A *Web szerver* szekvenciálisan végigolvassa a tartalmát, és a benne szereplő ***szerveroldali*** scripteket végrehajtja. Az ekkor létrejövő szövegeket és HTML tagokat a scripthívás helyére beilleszti. Az eredeti .asp fájlban talált normális szövegeket és HTML elemeket érintetlenül hagyja, majd a kapott eredményt leküldi a böngészőnek. Az ASP fájl Jscript, vagy VBScript nyelvű ***kliensoldali*** scriptjei forrásnyelvi formátumban kerülnek le a böngészőbe, ezeken a Web szerver nem végez módosítást. A ***kliensoldali*** scriptek a böngésző illetve a felhasználó által generált események hatására futnak le a *felhasználó gépén*.
A szerveroldali scriptek a Web szerveren, a kliensoldali scriptek a felhasználó gépén futnak le. E szétválasztás előnyei: a szerveroldali scriptek platformfüggetlenek, nem olvashatók a kliens böngészőjében, „megtakarítják" a felhasználó gépének kapacitását. A *kliensoldali* elemek és a *szerveroldali* in-line scriptek tetszőleges módon vegyíthetők.
A ***szerveroldali*** scriptek függvényeket és eljárásokat is definiálhatnak, melyeket csak a szerveroldalon lehet meghívni.
A PHP nyelv kifejezetten dinamikus oldalak előállítására készült. Széles körű funkcionalitása lehetővé teszi a különböző adatbáziskezelőkkel való kapcsolattartást. Beépített függvényei segítik a MySQL vagy valamilyen más adatbáziskezelővel való együttműködést - az adattárolás és a megjelenítés különválasztását. A PHP-ből kiadott SQL-parancsokkal lehetőség nyílik az adatok karbantartására, lekérdezésére, stb. Pozitívumként említeném, hogy elemei egyszerűen HTML kódba is illeszthetők és a kliens oldalon nem jelenik meg a php forráskód.
A szerveroldali programozás több szempontból is előnyös a Web fejlesztő számára:

1. A szerveroldali scriptek fejlesztése teljesen független a felhasználó által alkalmazott böngészőtől.
2. Így a szerveroldali scripteket olyan programnyelvben is fejleszthetjük amelyt a felhasználó böngészője nem támogat.
3. A szerveroldali scriptek forrásnyelvi változata nem olvasható a felhasználó böngészőjében. Ez fontos momentum a forrásnyelvi változat védettsége szempontjából.
4. A letöltendő HTML dokumentumok mérete csökken (mivel a böngészőben csak a szerveroldali scriptek végrehajtási eredménye látszik). Így a Web oldal beolvasása is gyorsabb.
5. Egy szerveroldali komponens alkalmazásával leolvashatjuk a felhasználó gép operációs rendszerét és böngészőprogramját, így a szerveroldalról leküldött HTML dokumentum a kliens gép környezetére alakítható.
6. Az ASP fájl az adatbázis szerverről adatbázist használhat fel a megjelenítendő Web lap feldolgozására.

18.2 A szerveroldali objektumokok bemutatása.

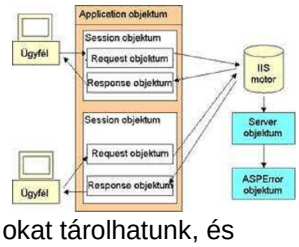
Az ASP objektummodell és elemei

Az .asp oldalak programozását, a HTTP kommunikáció, a webkiszolgáló egyes szolgáltatásainak elérését külön objektummodell segíti. Az ASP objektummodelljének minden elemét elérhetjük az .asp oldalak kódjaiból.
Ha a tranzakciós műveletekhez használt ObjectContext objektumot nem számítjuk, az objektummodell hat *(Windows NT 4.0-n öt)* objektumból áll. A Windows 2000-ben megjelent új objektum az ASPError, ami egy bekövetkezett hiba leírását tartalmazza, az .asp-be ágyazott, saját hibakezelést segíti.

Server objektum

Magát az IIS (Internet Information Services) -t képviseli, néhány kiszolgálószintű beállítással és szolgáltatással. A Server objektum egy és oszthatatlan, és főleg kényelmi szolgáltatásai miatt hasznos. Az objektum a Web szerverrel való kapcsolattartást szolgálja, akkor használjuk ha egy szerveroldali COM technológiájú komponenst akarunk felhasználni. (Pl. Adatbázis-kezelés ADO technológiát megvalósító komponensek.)

Az **Application objektum** **(.asp** kódok halmaza**)**



Egy webalkalmazást jelképez. Az Application objektum nem más, mint egy közös tárolóhely, amit az adott webalkalmazás minden scriptje elér. Az Application objektumban adatokat, objektumokat tárolhatunk el, hogy majd később kiolvassuk onnan. (Pl. a weblapot látogatok száma) HTTP eredetileg állapotmentes világ: jön egy kérés, mi kiszolgáljuk, azután jön a következő, és a kiszolgálónak végül is fogalma sincs arról, hogy melyik kérést éppen ki küldte. Lehet, hogy ugyanaz a felhasználó, lehet, hogy a világ két végéről két olvasó jelentkezett.
Mégis, milyen jó lenne, ha lenne egy globális „valami", amiben adatokat, sőt, akár kész objektumokat tárolhatunk, és bármikor szükség van rá, csak
„fel" kell nyúlnunk érte, és hopp, máris a rendelkezésünkre áll! Nos, pontosan erre való az Application objektum.

A **Session objektum** **(munkamenet)**

A session objektum célja teljesen hasonló, mint az application objektumnak, csakhogy az Application-nel ellentétben nem felhasználók közötti, hanem egy adott felhasználó műveletei fölötti globális objektum. Azaz, számos Request és Response fölött uralkodó objektum, ami megmarad egészen addig, amíg a felhasználó el nem hagyja a webhelyet, vagy be nem csukja a böngészőjét. Természetesen Session objektumból már nem csak egy van: ahány felhasználó, annyi Session. Az egyes munkamenetek azonosítása cookie-k segítségével történik. Az *(egyébként változó)* ASPSESSIONIDxxxxxxxx nevű cookie tartalmának segítségével az IIS egyértelműen azonosítja a visszatérő böngészőt, és ugyanabba a környezetbe helyezi *(azaz, mindenki az első látogatáskor részére létrehozott, különbejáratú Session objektumba kerül)*.
Elbúcsúzhatunk a munkamenetektől, ha a felhasználó böngészője visszautasítja a cookie-kat. Ha ilyenkor mégis valami hasonló funkcionalitást szeretnénk elérni, külső gyártó termékeit kell használnunk, amelyek az URL-be ágyazva valósítják meg a munkamenetek kezelését (az IIS-ben szűrő- ként működve minden, az oldalakban található URL hivatkozáshoz hozzáfűzik az azonosítót, míg a böngészőktől érkező kérésekből kiszűrik azokat). Ez a megoldás teljesen cookie-mentes, és elvileg minden böngésző boldogul vele *(csak kicsit rondák lesznek az URL-ek)*.

HTTP kérés – a Request objektum
A dinamizmus, az interaktivitás egyik mozgatórugója természetesen az, ha menet közben adatokat kapunk az ügyfél oldaláról. Ezeket az adatokat a Request objektum segítségével érhetjük el.segítségével kódból hozzáférhetünk a kérés minden eleméhez, legyen az HTTP fejléc értéke, a böngészőben tárolt cookie, vagy kérdőív tartama. *Segítségével érhetjük el azokat az információkat, amelyek a böngészőkből érkeztek a Web szerver felé. Egy HTML űrlapot jelenítünk meg, és a felhasználó által bevitt adatokat egy küldés gomb lenyomása után egy ASP fájlt hívunk meg*

***Egy HTTP válasz – a* Response objektum**
A Response objektum egy HTTP kérésre adott válasz. *Segítségével lehet vezérelni a felhasználóknak küldendő információt.* Kiemelt Response metódus Write – normál szövegeket és HTML elemeket tudunk küldeni egy szerver oldali scriptből a felhasználó böngészője felé.

Egy IIS kiszolgálón belül **Server** és **ASPError** objektumból egy-egy létezik *(utóbbi csak akkor érhető el, ha hiba történt)*. **Application** objektum minden webalkalmazás egyedi, globális objektuma, **Session** objektumból minden munkamenethez *(ügyfélhez)* egy jön létre, egyidejűleg tehát több is létezhet. **Request** és **Response** objektum pedig mindig az adott kérésre és válaszra vonatkozik, újabb kérés esetén új példány jön létre belőlük is.

18.3 Adatbázis-kezelés webűrlapokkal.

Adatbázis kezelés az ASP-ben:
Kapcsolatfelvétel az adatbázissal**:** Ahhoz, hogy az **ASP**-vel egy fizikailag létező adatbázis adataihoz hozzáférjünk, az első teendőnk, hogy létrehozzunk egy objektumot, ami a későbbiekben az adatbázist szimbolizálja. Az objektum létrehozását az ADODB.CONNECTION segítségével a következőképpen tehetjük meg: <%SET CONN=SERVER.CREATEOBJECT("ADODB.CONNECTION")%> Az adatbázis kapcsolati objektum létrehozása, az adatbázis fizikai tárolásának módjától függetlenül mindig így történik. Az adatbázis megnyitása azonban, az adott típusú adatbázistól függő lesz. A következőkben megnézzük az igen elterjedt nagyobb adatbázisok esetében használt **MS-SQL** adatbázis megnyitását. SQL adatbázis megnyitása: Nagyobb adatbázisok esetében, vállalati környezetben, illetve ahol a titkosításnak is nagy szerepe van célszerű az SQL adatbázis használata.Az SQL adatbázis megnyitása:

```
<%
SET CONN = Server.CreateObject("ADODB.Connection
CONN.OPEN "DATABASE= adatbázis ; DSN= system_
%>
```

Elem	Leírás
Adatbázis	Az SQL ENTERPRISE MANAGER-ben az adatbázisunkhoz hozzárendelt név.
System_dsn	A *vezérlőpult/ODBC Data Sources-ben az adatbázisunkhoz hozzárendelt System dsn.*

SQL parancs futtatása: Ha már sikeresen felvettük a kapcsolatot az adatbázissal, szükségünk van az abban szereplő adatok lekérdezésére, módosítására törlésére. Ezeket az adatbázis műveleteket SQL parancsok segítségével hajthatjuk végre. A következőkben arról lesz szó, hogyan futtathatjuk ASP környezetből az adatbázis-kezelő SQL parancsokat, valamint ha lekérdező utasításról van szó, hogyan jeleníthetjük meg a lekérdezés eredményét. <%CONN.EXECUTE(parancs)%>, Ahol a *Parancs* helyére kerül a végrehajtandó SQL parancs

Lista: Eredménytábla, amelybe a lekérdezés eredménye kerül, "sorokat" és "oszlopokat" tartalmaz. Az oszlopokat sorszámukkal különböztetjük meg. *Pl.: lista(0), lista(1) … lista(n).*
A sorok között a *lista.MoveNext* (következő sorra ugrik), illetve a *lista.MoveFirst* (első sorra ugrik) utasításokkal lehet mozogni. A *lista.EOF* logikai értéket ad vissza, értéke TRUE–ra változik, ha az eredménytábla végére érünk.
Parancs: Ide kerül a végrehajtandó SQL parancs

Lekérdező SQL parancs futtatása és az eredmény eltárolása:
<%SET lista = CONN.EXECUTE(parancs)%>

18.4 Egy szerveroldali programnyelv rövid bemutatása, jellemzése.

Az ASP.NET nyelvfüggetlenségének köszönhetően alkalmazásainkat bármely .NET kompatibilis nyelven fejleszthetjük, akár több nyelven is egyszerre. Ez lehet C#, Visual Basic.NET, vagy JScript.NET.
Igen egyszerűen tudunk webes alkalmazást fejleszteni ASP.NET-ben Visual Studio segítségével, azonban az eszköz designer támogatása sok részletet eltakar a fejlesztő elől. A Visual Studio.NET keretrendszer ToolBox palettájának Web Forms füle alatt találhatjuk meg azokat a vizuális kontrolokat, melyek szerver oldali alkotóelemek, vagyis amellett hogy a Webes űrlapokon megjelennek, a szerver-oldali kódból is elérhetőek.
A Webes alkalmazások esetében WebForm objektumok szolgáltatják azt a keretet, mely összefogja az egy-egy laphoz kapcsolódó funkcionalitást. Vagyis, minden weblap egy-egy WebForm, melyek fizikai állományként jelennek meg a Web-alkalmazás mappájában, és amelyeket a futtatórendszer értelmez, és dolgoz fel. Az állományok kiterjesztése .ASPX.

Visual Studio 2005 verziókban benne van: Visual Basic .NET, Visual C++ .NET, Visual C# .NET 2.0, Visual J# .NET, ASP.NET. Ingyenességük miatt megfelelő válaszások lehetnek tanulásra, otthoni munkára.

HTML oldalakba szerveroldalon is beágyazható JavaScript. A szerveroldali utasítások különböző gyártók relációs adatbázisait kapcsolhatják össze, megoszthatják egy alkalmazás adatait a felhasználók között, hozzáférést nyújthatnak a szerver fájlrendszeréhez, vagy a LiveConnect és Java használatával kommunikálhatnak más alkalmazásokkal. A szerveroldali JavaScript-tel ellátott HTML oldalak magukban foglalhatnak kliensoldali JavaScript utasításokat is.
Az egyszerű kliensoldali JavaScript oldalakkal szemben, a szerveroldali JavaScriptet használó HTML oldalak bytekódba fordított végrehajtható fájlok. Ezek a végrehajtható alkalmazások a webszerveren futnak, ahol rendelkezésre áll a JavaScript futásidejű motor. Ez okból a JavaScript alkalmazások létrehozása kétlépcsős művelet.
Az első lépcsőben, létre kell hozni a HTML oldalakat (amelyek tartalmazhatnak kliensoldali és szerveroldali JavaScript utasításokat is) és JavaScript állományokat. Ezek után az összes fájlt egy végrehajtható állományba kell fordítani.
A második lépcsőben az alkalmazás egyik lapját lekéri egy kliens böngésző. A futásidejű motor a végrehajtható alkalmazást használja, hogy kikeresse a forrásoldalt és dinamikusan létrehozza az elküldendő HTML oldalt. Az futtat minden szerveroldali JavaScript utasítást, ami az oldalon található. Az utasítások eredménye a HTML oldalhoz adhat új HTML elemeket, vagy kliensoldali JavaScript utasításokat is. A futásidejű motor ezután a hálózaton keresztül elküldi a kész oldalt a böngészőkliensnek, amely lefuttatja a kliensoldali JavaScripteket és megjeleníti a végeredményt.

19. A kliensoldali progiozás alapelemei az Internetes alkalmazások fejlesztésénél. A kapcsolódó technológiák rövid bemutatása: HTML, XHTML, XML, CSS, XSL. A kliensoldali script nyelvek használata.

19.1 A kliensoldali programozás alapelemei az Internetes alkalmazások fejlesztésénél.

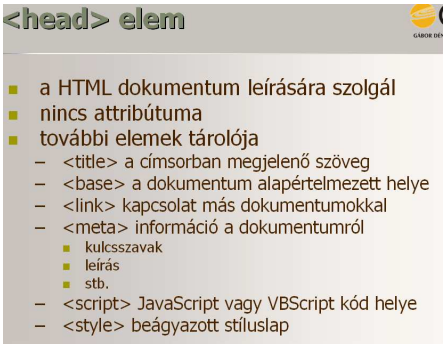
A tisztán kliens oldali megoldások esetén a kibővített térinformatikai funkcionalitást a kliens számítógépen futó alkalmazás biztosítja. Ezt a megoldást vastag kliens megoldásnak is nevezik, mivel a munka nagyobbik része a kliens számítógépen folyik, a hálózati sávszélesség mellett a kliens számítógép teljesítménye a meghatározó. A web szerver az adatokat, esetleg a letöltendő programokat szolgáltatja. Ezek a rendszerek az aktív képernyőkezelés megvalósítása érdekében igyekeznek kihasználni a korszerű WEB-böngészők dinamikus HTML lehetőségeit. A kliens oldali VBscript, illetve JavaScript technológia lehetővé teszi számos funkció elvégzését a kliens oldalon. Az Internetes alkalmazások kliens oldali programozásának elemei **:** HTML és XHTML weblapok, CSS dtílus lapok illetve kliensoldali scriptek. (VBScript, JavaScript).

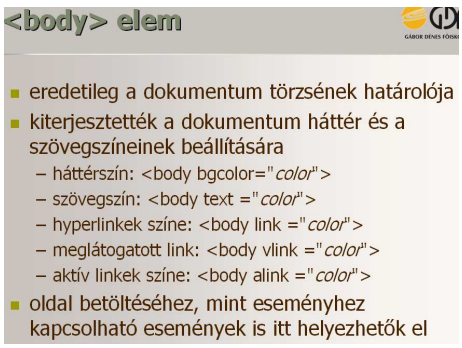
19.2 A kapcsolódó technológiák rövid bemutatása

HTML: Hyper Text Markup Language olyan szöveges dokumentum, amely magában hordozza a dokumentum megjelenítéséhez szükséges, formázást leíró paramétereket. Így a dokumentumokat olyan szöveges formában lehet tárolni, amit kis sávszélességű hálózaton is rövid idő alatt lehet továbbítani és a kliens számítógépén - bármilyen operációs rendszerrel is üzemel - a dokumentum megjeleníthető lesz.

- HyperText Markup Language – Hypertext Jelölő Nyelv
- célja: dokumentumok számára standard kód segítségével formázott megjelenítést biztosítani
- alapeleme a címke (tag) pl. <html>
- nyitó- és záró címke pl. <html> … </html>
- csak nyitó címke pl.
, <hr>
- nyitó címke belsejében attribútumok pl. <a href="http://www.gdf.hu">
- a böngészők intelligens módon értelmezik
- félreértések elkerülésére: XHTML

A HTML nyelv alapeleme a címke, vagy angol nyelven a tag. Ezek nyitó (<) és záró (>) kacsacsőr, vagy "kisebb mint" illetve "nagyobb mint" jelek közé zárt egyszavas angol nyelvű dokumentum-elemnevek, vagy azok rövidítései (például <body> címke jelzi, hogy a dokumentum törzse kezdődik. :





XHTML

Az XHTML (EXtensible HyperText Markup Language – kiterjeszthetı HTML) a HTML pontosabb és „tisztább” verziója. A HTML 4.01 leváltására hozták létre. Az XHTML az XML egy alkalmazása, vagyis XML eszközökkel is feldolgozható. Az XHTML felülről kompatibilis a HTML 4.01-el. Gyakorlatilag nincs jelentős eltérés a két nyelv között, csak a formai követelmények lettek szigorúbbak:

- Mindent kisbetűvel kell írni!
- Minden elemet le kell zárni! Az üres elemeket önmagukban egy szóközzel és egy / jellel:
.
- Az elemeket csak egymásba ágyazva lehet használni! Nem jó: Szöveg *Jó: szöveg*
- Az attribútumokat idézőjelek közé írjuk! <cimszo magyar="alma”>
- Az attribútumoknak legyen értéke!

- Extensible HyperText Markup Language
- a HTML szigorúbb és tisztább változata
- a HTML elemeket leírása XML segítségével
- minden címkének van záró eleme
- egysoros címkénél a lezárást jelezzük pl.
- minden címkenév kisbetűvel írandó
- minden eleme azonosítható a Document Object Model segítségével, az objektumok módosíthatók

XML (e**X**tensible **M**arkup **L**anguage) magyarul bővíthető jelölőnyelv. Az XML egy rendkívül hasznos technológia, de nem tekinthető egy önálló programozási nyelvnek. Az XML dokumentumban lényegében egy „nyelvtant” képezünk le az adatszerkezeteink megadására, illetve leírására. Vagyis az XML használatával egyedi leíró nyelveket alkothatunk, amelyek segítségével gyakorlatilag bármilyen információ leírható.A HTML-hez képest a változás az, hogy saját tag-eket is definiálhatunk. Nincsenek előre definiált tagok, azok minden egyes XML alkalmazás esetén egyedileg definiálhatók. Az XML-ben tárolt adatok hardver-, és szoftver független adatmegosztást tesznek lehetővé. Sokkal könnyebbé teszi olyan adatok létrehozását, amelyekkel különböző alkalmazások is dolgozni tudnak.

Jól formázott XML dokumentum:

- A címkék neveiben a kis-és nagybetők különbözőnek számítanak.
- Minden nyitó címkének meg kell legyen a záró párjai is (kivéve a rövidített formátumban leírt üres elemeknél). rövidített üres elem Pl. (</Past>)
- Egymásba ágyazás esetén a címkék nem lapolódhatnak át.
- Az attribútumok értékének mindig idézőjelek között kell megjelennie.
- Minden XML dokumentumnak kell tartalmaznia gyökérelemet (angolul root element).

XML szintaxisa:

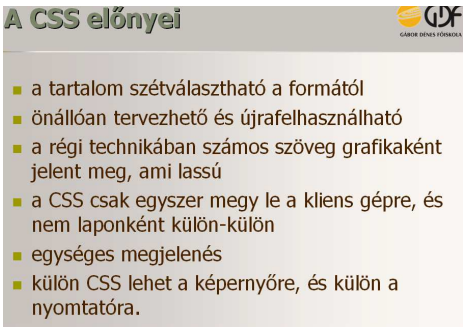
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Az első sor a dokumentum formátumát adja meg: verziószámot, és a karakterkódolást.

- A <note> és </note> tagok határolják az egész dokumentumot
- kitöltő gyökér elemet. A közbülső négy sor a bejegyzés adatait tartalmazza.

CSS

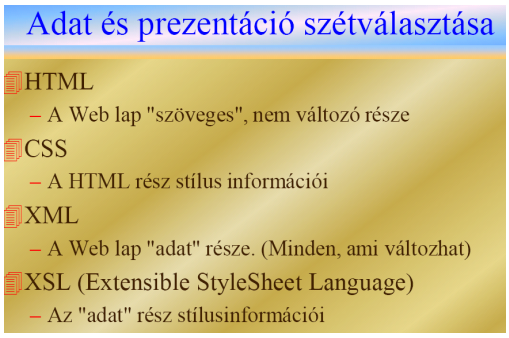
A CSS (angolul Cascading Style Sheets) a számítástechnika egyik stílusleíró nyelve, amely a HTML vagy XHTML dokumentumok megjelenését írja le. A CSS-t a html, xhtml dokumentumokban arra használják, hogy a lapok színét, hátterét, betűtípusait, elrendezéseit stb. beállítsák. . A stíluslapokat külön .css kiterjesztésű állományban szokás elhelyezni. Így könnyedén lehet ugyanazt a megjelenítést adni a honlap összes oldalához, mindössze egyetlen CSS állomány szerkesztésével. A CSS technika alapvető célja, hogy szétválassza a dokumentum tartalmi részét a megjelenítési stílus elemektől. A stíluslapok alkalmazásának nagy előnye, használatukkal hatékonyabbá, gyorsabbá és rugalmasabbá tehetjük a webszerkesztést, elfelejthetjük a korlátozott formázási lehetőségeket, segítségükkel átláthatóbbá tehetjük forráskódjainkat.



- HTML vagy XHTML dokumentumok stílusleíró nyelve
- segítségével egységesen állítható a betűméret, szín, elrendezés, stb.
- elkülönül a dokumentum struktúrája a dokumentum megjelenésétől

XSL

Az XSL(Extensible Stylesheet Language - kiterjeszthető stílusleíró nyelv) támogató technológia, ami leírja, hogy hogyan kell formázni az XML dokumentumban található adatot. Hasonló tevékenységet old meg mint a CSS-es, de XML dokumentumokon. Egy sor az XML dokumentum tetején:
<?xml-stylesheet type="text/xsl" href="transform.xsl"?>
Azt mondja meg, hogy a transform.xsl egy XSLT stíluslap, ami az XML-ről HTML formátumra való átalakításra vonatkozóan hordoz információt.



19.3 A kliensoldali script nyelvek használata

A legnépszerűbb kliensoldali script nyelv a JavaScript. Dinamikus viselkedéssel ruházhatjuk fel segítségével a HTML/XHTML-ben elkészített weblapunk elemeit. Ellentétben a HTML-lel (amely leírónyelv) és a CSS-sel (amely stíluslap nyelv), a JavaScript programozási nyelv. A HTML oldalakba ágyazott kliensoldali JavaScript utasítások válaszolhatnak a felhasználói eseményekre (egérkattintás, űrlap adatbevitel, stb...). Például, írhatunk olyan JavaScript függvényt, amely ellenőrzi, hogy a felhasználó által beírt adatok érvényesek-e (telefonszám, irányítószám, stb...). A HTML oldalba beágyazott JavaScript minden hálózati adatátvitel nélkül meg tudja vizsgálni a beírt adatot és ha a felhasználó érvénytelen adatot írt be, arra párbeszédablak megjelenítésével figyelmeztetheti.
A kliensoldali scriptek olyan programrészletek amelyek forrásnyelvi alakban a HTML dokumentum keretében töltődnek le a felhasználó számítógépre. A kliens oldali scripteket a böngészőprogram hajtja végre.
A végrehajtás bekövetkezhet:

- egérmozgatás a HTML dokumentum objektumára
- egérkattintás, vagy
- a HTML dokumentum letöltése miatt.

Javascript Szolgáltatásai:

- Egyszerű használat
- Módosíthatja a HTML oldalak tartalmát, kinézetét
- Eseményekre tud reagálni
- Bevitt adat helyességének ellenőrzése
- Megvizsgálhatjuk a böngésző típusát, így ennek függvényében más-más böngésző-specifikus tartalmat tölthetünk be
- Sütiket (cookie) hozhatunk létre a kliens gépen való információ-tárolás érdekében.

Annak érdekében, hogy azoknál a klienseknél, akiknél nincs telepítve a mefelelő Java környezet a JavaScript-et úgy kell beépíteni a weboldalakba, hogy anélkül is helyesen működjenek. Azaz diszkrét JavaScript megoldást kell alkalmazni.
A diszkrét Javascript azt mondja, hogy a HTML kódunkban ne használjunk Javascriptet, válasszuk le, s tegyük külön fájlba scriptjeinket, s építsük fel úgy az oldalt, hogy azok nélkül is teljes funkcionalitással működjön. Ismerősnek tűnhet az ötlet: a mai CSS technikák pontosan ezt mondják a stíluslapok esetén is: válasszuk szét a megjelenést és a tartalmat. A Javascript esetén a tartalom, s a használhatósági javítások szétválasztásáról van szó. A HTML azt mondja meg, mi ez a szöveg, a CSS azt, hogy hogyan nézzen ki, a Javascript pedig azt, hogy hogyan viselkedjen (az oldal).

20. Az informatikai biztonság fogalma. A biztonsági rendszer tervezése, a tervezés szakaszai. Az egyes tervezési szakaszok fő feladatai. A kockázatelemzés célja és lépései. Az informatikai rendszerek elleni támadások típusai. Kriptográfiai módszerek és eszközök, azok gyakorlati alkalmazásai.

20.1 Az informatikai biztonság fogalma. A biztonsági rendszer tervezése, a tervezés szakaszai. Az egyes tervezési szakaszok fő feladatai.

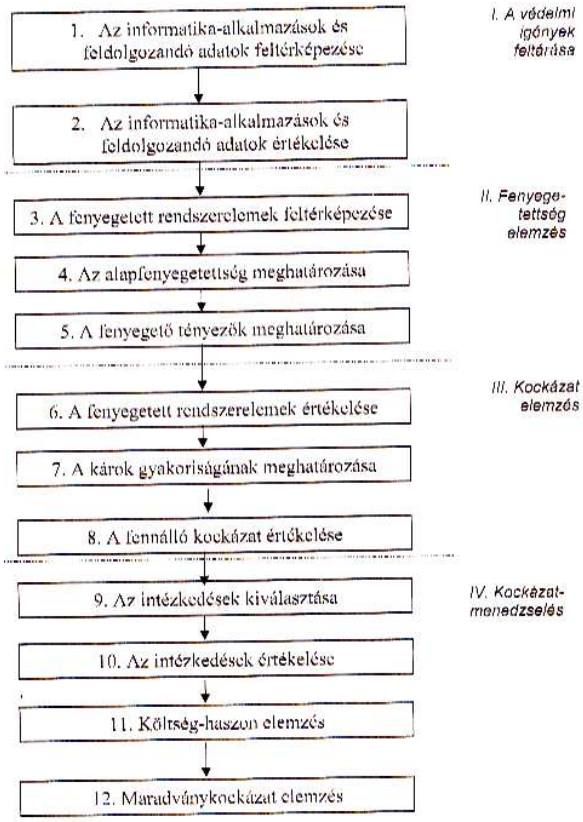
Az **informatikai biztonság** a védelmi rendszer olyan, a védő számára kielégítő mértékű **állapot**a, amely az informatikai rendszerben kezelt adatok **bizalmassága**, **sértetlensége** és **rendelkezésre állása** szempontjából **zárt, teljes körű, folytonos** és a **kockázatokkal arányos**.

Bizalmasság, *annak biztosítása, hogy az információ csak az arra felhatalmazottak számára legyen elérhető.*
Sértetlenség: Az adat tulajdonsága, amely arra vonatkozik, hogy az adat tartalma és tulajdonságai az elvárttal megegyeznek, ideértve a bizonyosságot abban, hogy az elvárt forrásból származik (hitelesség) és a származás ellenőrizhetőségét, bizonyosságát (letagadhatatlanság) is, illetve az elektronikus információs rendszer elemeinek azon tulajdonságát, amely arra vonatkozik, hogy az elektronikus információs rendszer eleme rendeltetésének megfelelően használható.
Rendelkezésre állás, *annak biztosítása, hogy a felhatalmazott felhasználók mindig hozzáférjenek az információkhoz és a kapcsolódó értékekhez, amikor szükséges*.

A biztonsági rendszer tervezése, a tervezés szakaszai:

A védelmet teljes körűen és zártan kell kialakítani. A ráfordítás mértékét az elviselhető kockázat mértéke szabja meg, amelyet a kárérték és a bekövetkezési gyakoriság osztályok alapján felállított kockázati mátrixban kijelölt elviselhetőségi határ szabja meg. Ezt a határt minden szervezet informatikai biztonsági vizsgálatánál egyedileg kell meghatározni.
Szakaszai:

- **Védelmi igények feltárása**: kiválasztjuk azokat az informatikai alkalmazásokat, amelyek a szervezet szempontjából a legfontosabbak, és később már csak ezekkel foglalkozunk → *Ennek a szakasznak az a célja, hogy reális és teljes képet kapjunk a védendő rendszer felépítéséről, tartalmáról – gondoljunk csak bele, hogy hogyan védjünk meg egy rendszert, ha pontosan nem is tudjuk mit kell védeni. Feltérképezés után valamennyi informatikai alkalmazást és feldolgozandó adatot sorba kell állítani és kiválasztani azokat, amelyek a legfontosabbak és védelmet igényelnek (akár értékskála készítése és hozzárendelése az elemekhez).*
- **Fenyegetettség elemzés:** megkeressük a informatikai rendszer gyenge pontjait és azokat a fenyegető tényezőket, amelyek az informatikai rendszerre veszélyt jelenthetnek.
  Az ITB (Informatikai Tárcaközi Bizottság) az informatikai rendszert nyolc tényezőre bontja, amelyek a teljes rendszert és annak környezetét lefedik:
  - *környezeti infrastruktúra* (épület, helyiségek, víz, világítás, telefont, védelmi berendezések, stb.) fenyegetés: földrengés, árvíz, sztrájk, merénylet, jogosulatlan személyek, közműellátás, stb.
  - *hardver* fenyegetés: műszaki hibák, környezeti hatások, szoftver által keltett probléma, személyek, stb.
  - *szoftver* fenyegetés: szoftverhiba, vírus, kezelési hiba, karbantartási hiba, stb.
  - *adathordozó* fenyegetés: ellenőrizetlen folyamatok, gyári hibás termékek, privát felhasználás, stb.
  - *dokumentumok* fenyegetés: hiányzó adminisztráció, ellenőrizetlen sokszorosítás, stb.
  - *adatok* fenyegetés: hardver-szoftver hiba, személyek, stb.
  - *kommunikáció* fenyegetés: jogosulatlan bejutás, hálózati hibák, lehallgatás, stb.



- **Kockázatelemzés:** az vizsgáljuk, hogy az informatikai rendszerre milyen káros hatása lehet a fenyegető tényezőknek. Meghatározzuk a lehetséges kár gyakoriságát és a kárértékét.
  A kockázatelemzés lépései:
  - *Fenyegetett rendszerelemek értékelése, azaz a különböző rendszerelemekhez fontosságtól, prioritástól függően értéket rendelni. Például, ha egy lemezes tároló különböző adatokat tárol, akkor a legnagyobb prioritású adat értékét adjuk neki.*
  - *Károk gyakoriságának meghatározása, azaz egy skálán megbecsüljük, hogy milyen gyakran következhetnek be fenyegetések a rendszer ellen. Akár szakértő véleményét is kikérhetjük.*
  - Fennáló kockázat értékelése*: egy mátrix segítségével értékeljük a kockázatot. Egyik tengely a gyakoriság, másik a kárérték lesz. A tábla alapján meg kell állapítani, hogy mely értékpárok jelentenek elviselhető, és melyek elviselhetetlen kockázatot. A kockázat nagysága a két értékből együtesen adódik. Az eredményt dokumentálni kell és a vezetőkkel, felelősökkel meg kell osztani.*
- **Kockázat- menedzselés**: kiválasztjuk a fenyegető tényezők elleni intézkedéseket és azok hatását értékeljük. Megnézzük, hogy az egyes intézkedések milyen költségekkel járnak és milyen hasznot hoznak. Intézkedések általánosan: védett elhelyezés, tűzvédelem, vízvédelem, sugárzásvédelem, naplózás, védelmi eszközök, javítás, karbantartás, mentések, ellenőrzések, előírások, stb.

20.2 A kockázatelemzés célja és lépései. (lásd fentebb)

20.3 Az informatikai rendszerek elleni támadások típusai.

A támadás az informatikai rendszer valamennyi elemén keresztül történhet.
Ezek a következők:

A környezeti infrastruktúra: a számítóközpont épületének területe, maga az épület, az épületben lévő helyiségek, átviteli vezetékek, áramellátás, klíma, víz, világítás, telefon, és különböző rendeltetésű berendezések (belépés-ellenőrző, tűzvédelem, betörésvédelem).
Gyenge pontok:
- Nem védett átviteli vezetékek, kábelek, informatikai berendezések
- Illetéktelen személyek felügyelet nélküli jelenléte, vagyis a belépési biztonság hanyag kezelése.
- A védelmi berendezések működési módjának vagy gyengeségeinek jogosulatlanok általi +ismerése.

Fenyegető tényezők:
- „Vis major": robbanás, repülőgép lezuhanása, sztrájk, háborús helyzet.
- Személyek által kifejtett erőszak: robbantásos merénylet, fegyveres behatolás, gyújtogatás, savazás, vandalizmus, betörés.
- Jogosulatlanok ellenőrizetlen belépése épületekbe helyiségekbe, szervezeten kívüli személyek által végzett, nem felügyelt munkálatok.
- Közműellátás: (áram, víz, telefon) és védelmi berendezések zavara vagy kiesése.

Hardver: ide tartoznak a számítástechnikai eszközök a hálózati csatoló eszközök a hálózat építő eszközök, speciális biztonsági berendezések.
Gyenge pontok:
- Eltulajdonítás: a készülékek csekély mérete, súlya miatt a lopás könnyen lehetséges.
- Külső behatások miatti meghibásodás: hőhatás, vízzel való elárasztás, érzékenység az elektromágneses sugárzásra, mechanikai behatásokra.
- Tartozékok utánpótlásának szervezetlensége: pótalkatrészek, printer festékek. Fenyegető tényezők:
- Tervezési és kivitelezési hiányosságok.
- Személyekkel összefüggő fenyegetés: készülékek károsítása vagy roncsolása, ellopása, jogosulatlan szerelés és alkatrészcsere.

Az adathordozók: ebbe a csoportba tartoznak a raktározott állapotú szoftverek, a biztonsági másolatokat, munkakópiákat, archív adatokat jegyzőkönyvi adatokat tartalmazó adathordozók, valamint az újonnan beszerzett és még használatba nem vett, ill. a felszabadított adathordozók melyek tartalmára a továbbiakban nincs szükség. Gyenge pontok:
- Fizikai instabilitás: érzékenység a behatásokra.
- Kikapcsolható írásvédelem.
- Könnyen szállíthatóak, a szállítás nehezen ellenőrizhető.

Fenyegető tényezők:
- Újrafelhasználásra vagy megsemmisítésre történő kiadás előzetes törlésük, felülírásuk nélkül.
- Ellenőrizetlen másolás ill. hozzájutás az adathordozókhoz.
- A szervezet tulajdonát képező adathordozók privát célú használata és privát adathordozók szolgálati használata.

A szoftver: ebbe a kategóriába csak a használatban lévő szoftverek tartoznak, a raktározott állapotú szoftvereket az adathordozóknál tárgyaltuk.
Gyenge pontok:
- Specifikációs hiba, a progik átvételének és ellenőrzésének hiánya.
- Bonyolult felhasználói felület, felhasználó hitelesítés hiánya.
- Az események hiányzó jegyzőkönyvezése (pl. belépés, CPU használat, fájlok +változtatása).
- A rendszer védelmi eszközeinek könnyű kiismerhetősége.
- A hozzáférési jogok helytelen odaítélése.
- Más felhasználók ismeretlen progijainak használata (vírusfertőzés).

Fenyegető tényezők:
- Jogosulatlan bejutás az informatikai rendszerbe a kezelői helyről vagy a hálózatról.
- Visszaélés a kezelési funkciókkal.
- Szoftver ellenőrizetlen bevitele, vírusveszély.
- Karbantartási hiba / visszaélés a karbantartási funkciókkal, helytelen karbantartási funkciók (távoli karbantartás a hálózaton)

Kommunikáció: ezen elemcsoport tárgya valamennyi adat továbbítási ideje alatt, amelyeket valamely szolgáltatás realizálása érdekében hálózaton továbbítanak. A hálózatok lehetnek az üzemi területen belül (pl. LAN-ok) vagy azon kívül (pl. közüzemi hálózatok), ill. e kettő kombinációja. A kommunikációhoz szükséges hardvert mindaddig, amíg az informatikai rendszer üzemeltetőjének felelősségi körén belül van, a hardver elemcsoportban, a vezetékeket pedig,amennyiben az üzem területén belül vannak, a környezeti infrastuktúra elemcsoportban szerepeltetjük.
Gyenge pontok:

- A hálózati szoftver és hardver hibái, azok manipulálhatósága.
- Üzenetek lehallgatása, +hamisítása, az adó és a fogadó hiányzó azonosítása.
- A jelszavak vagy titkos kulcsok nyílt szövegben való továbbítása. // • Függés az átvitel sorrendjétől.
- Lehetőség az üzenet elküldésének, kézhezvételének hiányzó bizonyítása.

Fenyegető tényezők:
- Jogosulatlanok bejutása a hálózatba nem ellenőrizhető csatlakozások révén.
- Hálózati hardverek/szoftverek manipulálása, átviteli hibák. // • Nem várt forgalmazási csúcsok. Célzott terhelési támadások.
- A vezetékek kompromittáló sugárzásának kihasználása. // • A kapcsolat felépítésének lehallgatása

a lehallgatásnál a támadó illetéktelenül rákapcsolódik az átviteli vonalra, az ott folyó adat- és kulcs-forgalmat figyeli. Az üzenetek gyűjtésével, különleges helyzetek észlelésével támadhat.
- A komm.s kapcsolatok kikutatása (forgalmazás elemzés), a komm.s partnerek névtelenségének veszélyeztetése.
- A kapcsolat felépítése +hamisított azonossággal, +személyesítéssel.

A megszemélyesítés esetén a támadó beépül a komm.s összeköttetésbe, az üzeneteket elnyeli, és az ellen-állomások helyett válaszol mindkét irányba. Különös veszélyforrás lehet, ha a támadó az egymással kommunikáló állomásokat sorozatos ismétlésre kényszeríti, esetleg ugyanazon üzenet kétkülönböző rejtjelezett variációját szerzi meg, vagy valamelyik állomásról ismert választ kényszerít ki, amellyel megszerzi annak rejtjelezett változatát.

Személyek: e csoportba csak olyan személyek szerepelnek, akikre közvetlenül vagy közvetve szükség van az informatikai rendszer használatához, ezáltal hozzáférhetnek a másik hét csoport elemeihez. A személyeket két nézőpontból kell figyelembe venni: egyrészt az üzemeltetéshez szükség van rájuk, ebből következően ők maguk is védelemigényes rendszerelemek. Másfelől viszont ők bírnak a belépés és a bejutás lehetőségével, ebből következően a fenyegetések jelentős része rajtuk keresztül realizálódik.

Gyenge pontok:
- A munkából való kiesés, hiányos kiképzés, a veszélyforrások ismeretének hiánya, a fenyegetettségi helyzet lebecsülése.
- Kényelmesség, eltérő reakciók. // • Hiányzó vagy hiányos ellenőrzés.

Fenyegető tényezők:
- Szándéktalan hibás viselkedés: stresszhelyzet, fáradság, hiányos ismeretek, hibás szabályozás, az előírások ismeretének hiánya miatt, az információk gyanútlan kiadása.
- Szándékos hibás viselkedés: az előírások +sértése, fenyegetés, zsarolás, +vesztegetés, haszonszerzési célból, bosszú, frusztráció miatt.

Támadás típusok:
- **Hozzáférés megszerzése.**
- **Jogosultság kiterjesztése.**
- **Szolgáltatás bénító támadások (DoS)**
- **Elosztott szolgáltatás bénítás (DdoS)**
- **Hamis megszemélyesítés, jogosultság szerzése.**
- **Sebezhetőségek kihasználása.**
- **Hálózati eszközök támadása.**

A leggyakrabban előforduló támadások ismertetése:

Szolgáltatásmegtagadás típusú támadások (Dean of Service, DoS): a szolgáltatásmegtagadás típusú támadások lényege, hogy egy rosszindulatú személy olyasmit tesz a hálózattal vagy a kiszolgálóval, ami zavarja a rendszer működését, lehetetlenné teszi a munkavégzést. (Pl. elárasztjuk a gépet pingekkel, ekkor nem marad ideje más, hasznos tevékenységekre.) Egy ilyen támadásból nem sok haszna lehet egy kalóznak, betörni nem tud a rendszerbe, csak épp működésképtelenné teheti azt.

Trójai falovak: Olyan kártékony program, amelyet alkalmazásnak, játéknak, szolgáltatásnak, vagy más egyéb tevékenység mögé rejtenek, álcáznak. Futtatásakor fejti ki károkozó hatását. Ugyanolyan jogosultságokkal rendelkezik mint az őt futtató felhasználó.

Lehallgatott átvitel: az azonosítási folyamat nyitva áll a lehallgatásra, amit megtehet bárki, pl. 1 hálózatfigyelő progi használatával.

20.4 Kriptográfiai módszerek és eszközök, azok gyakorlati alkalmazásai.

A kriptológia az adatok, üzenetek rejtjelezésével (kódolás, sifrírozás) és megoldásával (rejtjelfejtés, dekódolás, desifirozás) foglalkozó tudományága, a matematikai tudományok egyik részterülete. A kriptológia egyik fő területe a kriptográfia, magyarul a rejtjelzés, amelynek alapvető feladata matematikai módszereket alkalmazó algoritmusokkal és azok használatának pontos leírását tartalmazó – szigorúan betartandó – kriptográfiai protokollok segítségével biztosítani az üzenetek, illetve tárolt információk bizalmasságát, védettségét, hitelességét. A kriptológia másik tudományága a kriptoanalízis (kriptográfiai bevizsgálás), amely a rejtjeles üzenet birtokában, de az eljárás teljes ismerete nélküli megfejtéssel (feltörésére) irányuló eljárásokkal foglalkozik. A kriptoanalízis főként matematikai módszereket használ. A rejtjelzett kommunikáció folyamatában a küld és a fogadó üzenetváltása történik meg. A küldő a nyílt szövegből rejtjelzés segítségével rejtjelzett szöveget állít elő, majd elküldi a vevőnek, aki azt visszafejtve megkapja az eredeti nyílt szöveget. A rejtjelzési folyamat – kódolás – során a rejtjelzett szöveg előállításához az algoritmuson kívül általában szükséges egy kulcs is, amelynek ismerete elengedhetetlen a rejtjelzésnél és a visszafejtésnél is.

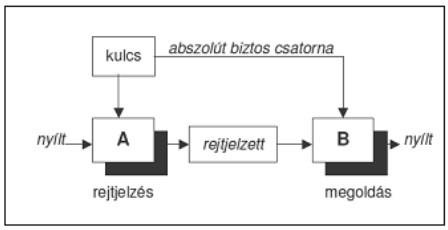
Kriptográfiai módszerek:

-Behelyettesítés:a nyílt szöveg minden karakteréhez valamilyen algoritmus szerint hozzárendelünk egy vagy több karaktert

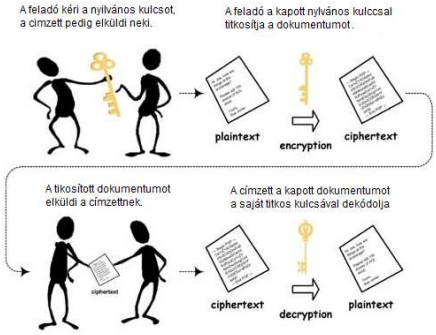
-Keverés: a nyílt szöveg karakterei változatlanok maradnak, de sorrendjük megváltozik

Akár kriptográfiai titkosításról, akár hitelesítésről (pl. aláírásról) van szó, az információt először kódolnunk kell, majd később, amikor fel szeretnénk használni dekódolnunk kell (azaz vissza kell fejtenünk). Mind a kódolás, mind annak visszafejtése (dekódolás) általában valamilyen kriptográfiai kulcs segítésével történik. E megoldások biztonsága arra épül, hogy a dekódoláshoz, illetve a hitelesítéshez szükséges kriptográfiai kulcsok kizárólag a jogosult felek birtokában vannak.
A kulcs egy szám, bitsorozat, amely meghatározott hosszúságú lehet. E hosszúságot nevezzük *kulcsméretnek*, amelye alapvetően meghatározza a kódolás biztonságát. Amikor a kulcsot létrehozzuk (ezt nevezzük *kulcsgenerálásnak*, az adott hosszúságú bitsorozatok közül – valamilyen véletlent is használó módszerrel – kiválasztunk egyet.
A támadó, aki nem ismeri a dekódoláshoz vagy hitelesítéshez használt kulcsot, megpróbálhatja kitalálni a kulcsot, ezt nevezzük a kódolás *megtörésének*. Ennek egyik módja, hogy egy számítógép segítségével az összes lehetséges (adott hosszúságú) kulcsot kipróbálja. Ha a kulcs elég hosszú, akkor ez nagyon nehéz feladat is lehet.

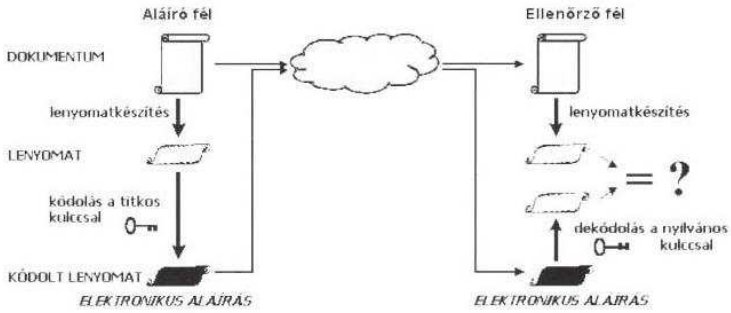
Szimetrikus kulcsú kriptográfia: A szimmetrikus kulcsú titkosítás során a dokumentumot ugyanazzal a kulccsal kódoljuk, amivel majd dekódolni is lehet; a kódolásra és dekódolásra használt kulcsot ekkor titkokban kell tartanunk, ezért *titkos kulcsnak* is nevezzük. E módszer hátránya, hogy a titkos kulcsot biztonságos módon kell eljuttatni a fogadó fél számára, hogy illetéktelen fél ne ismerhesse meg. Ez bizonyos esetekben nagyon nehéz problémát jelenthet. A szimmetrikus kulcsú titkosítás előnye, hogy az ilyen megoldások gyorsak és rövid kulcsokkal (pl. 128 bit vagy 256 bit) dolgoznak, és elég sok szimmetrikus kulcsú algoritmust ismerünk. Szimmetrikus kulcsú titkosító algoritmusok például a következők: DES (már nem használatos), 3DES, AES , stb.



Aszimetrikus kulcsú (nyilvános kulcsú) kriptográfia: Az aszimmetrikus kulcsú (más néven nyilvános kulcsú) titkosításnál a kódolás és a dekódolás nem ugyanazzal a kulccsal történik. Minden félnek van egy nyilvános kulcsa és egy magánkulcsa. Az egyik kulccsal kódolt üzenetet csak a hozzá tartozó másik kulccsal lehet dekódolni. A magánkulcs soha nem kerül ki birtokosa tulajdonából, de bárki hozzáférhet mások nyilvános kulcsához. A nyilvános kulcsot nem kell titokban tartani, azt bárki megismerheti. Ha titkosított üzenetet szeretnénk küldeni valakinek, meg kell szereznünk az ő nyilvános kulcsát, és azzal kell kódolnunk a neki szóló üzeneteket. Az így kódolt üzeneteket a címzett a saját magánkulcsával fejtheti vissza. A nyilvános kulcsú kriptográfia más módon is használható: ha a saját magánkulcsunkkal kódolunk egy dokumentumot, az így kapott adatról – a nyilvános kulcsunk alapján – bárki megállapíthatja, hogy azt mi hoztuk létre. E műveletet aláírásnak nevezzük.



Elektronikus aláírás: Digitális aláírásnak olyan elektronikus karaktersorozatot neveznek, amelyet igen nagy valószínűséggel csak az aláírótól származhat. A digitális aláírás tartalmazza az üzenet egyirányú képét (lenyomatát), s egyéb adatokat, például keltezést (dátumot, pontos időpontot), sorszámot, a küldött üzenetből képzett ellenőrző számot. Az aláírás jellemző a létrehozójára és az üzenetre egyaránt. Az elektronikus aláírást bárki ellenőrizni tudja, aki a megfelelő infrastruktúrához hozzáfér. A digitális aláírás két részből áll: a személyhez kötött aláírást generáló részből, s az ellenőrzést bárki számára lehetővé tevő részből. A digitális aláírás elkészítéséhez először kiegészítjük a dokumentumot a megfelelő azonosítókkal, majd ennek a kiegészített dokumentumnak egy alkalmas sűrítményét készítjük el. Ez lesz a digitális aláírás. Az alkalmas sűrítmények elkészítésére szolgálnak az úgy nevezett Hash eljárások. A Hash algoritmus egy olyan transzformáció, amely egy tetszőleges hosszú szöveg fix hosszúságú digitális sűrítményét készíti el, amely kizárólag az adott szövegre jellemző. Tulajdonságai: hamisíthatatlan , nem használható fel újra, letagadhatatlan, megváltoztathatatlan.



Kulcskezelés, PKI, CA: A nyilvános kulcsú rendszerben fontos tudni, hogy a nyilvános kulcs tulajdonosa valóban az a személy, akinek a levelet szánjuk. A digitális aláírást bárki létrehozhatja, ezért valakinek tanúsítani kell, hogy valóban az az aláíró, akinek vallja magát. Ennek valódiságát egyrészt az alkalmazott digitális aláírások biztosítják, másrészt különféle, úgy nevezett biztonsági modellek segítségével. A legbiztosabb megoldás a direkt biztonsági modell, amelyben mint a neve is mutatja a vevő személyesen adja át nyilvános kulcsát az adónak. Ez a valóságban, – a fizikailag nagy távolságok miatt – a legtöbbször kivihetetlen, ezért széles körben a hierarchikus biztonsági modell alapján kiépített Hitelesítés Szolgáltatón, vagy közismert nevén a Certificate Authority-n (CA) alapuló rendszer terjedt el a gyakorlatban. A résztvevők által megbízhatónak tekintett harmadik fél egy digitális közjegyző szerepét játssza. Olyan szakosodott szervezet vagy cég, amely tanúsítványokat adhat ki kliensek és szerverek számára. A CA igazolja, hogy egy adott azonosítóval rendelkező felhasználó az, akinek vallja magát. A nemzetközi feltételeket, szabványokat kielégítő infrastruktúrát magyarul is az angol Public Key Infrastructure (Nyilvános Kulcsú Infrastruktúra) kifejezésből származó PKI rövidítés jelöli.

21. Az információs rendszer fogalma és összetevői. Adat, információ, tevékenység, esemény, felhasználó, szabvány. Az információs rendszer szintjei és nézetei. (Egy példán bemutatva).

21.1 Az információs rendszer fogalma és összetevői. Adat, információ, tevékenység, esemény, felhasználó, szabvány

Az információs rendszer fogalma:
Szervezett együttese az **adatok**nak (információknak), velük kapcsolatos **események**nek, rajtuk végzett **tevékenységek**nek, ezekkel kapcsolatos **erőforrások**nak, **felhasználóik**nak, és ezeket szabályozó **szabványoknak és eljárásoknak**.

• eljárásokat biztosít információk rögzítésére, feldolgozására és elérhetővé tételére
• valamilyen szervezethez vagy annak egy részéhez kapcsolódik
• a szervezet céljainak elérését segíti

Adat: értelmezhető, de nem értelmezett ismeret.
Információ: az adatokból elemzéssel, rendszerezéssel kinyert új ismeret. az ember által értelmezett adat.
Tevékenység: adatkezelés, előállítás, illetve ezeket vezérlő műveletek.
Esemény: információs tevékenységet kiváltó és azt lezáró momentum.
Felhasználó: az információs rendszerrel kapcsolatban lévő ember(csoport). Ismeretátadásban két fél érdekelt: az ismeret küldője és fogadója. Az emberek eltérő szereppel kapcsolódnak az információs rendszerhez, egy felhasználó több szerepet is betölthet, lehet adatszolgáltató, adatfelhasználó, alkalmazás felhasználó, végső felhasználó. Az információs rendszer fejlesztői is felhasználók, és a vezető sajátos felhasználói szerepet tölt be.
Szabvány: az információs rendszerben vagy annak környezetében levő tényezőkre vonatkozó megegyezés. Hármas szerepet töltenek be: eligazítás, korlátozás és tájékoztatás.

21.2 Az információs rendszer szintjei és nézetei. (Egy példán bemutatva).

Információs rendszer szintjei:
Fogalmi szint: a valóságnak a kompromisszumoktól mentes képe.
Logikai szint: adott környezet korlátjainak megfelelően átalakított, kompromisszumokat tartalmazó fogalmi kép.
Fizikai szint: adott környezet konkrét fizikai adottságaira alkalmazott, tehát ilyen módon felhasznált és átalakított logikai kép.Ismeretek ábrázolása, tárolókon való elhelyezkedése. (környezet által konkretizált verzió, konkrét végrehajtás és implementáció)

Információs rendszer nézetei(vetületei):
Információ vagy adat vetület: az alapvető ismeretek lényege és struktúrája (adatok, adatszerkezetek), viszonylag stabil, objektív az információs rendszer többi tényezőjétől viszonylag független
Feldolgozás vetület: az eseményt és a tevékenységet foglalja magába (lekérdezések, kimutatások, jelentések), viszonylag instabil, szubjektív, a rendszer többi részétől függ.
Környezet vetület: meghatározza az információs rendszert, az eszközök objektív képességeit, az erőforrásokat, a felhasználók szubjektív igényeit, és a szabványok feltételrendszerét.

Példák:

	Szintjei			Nézetei		
	Fog	Log	Fiz	Adat	Feldolg	Körny
A vezetők a keresleti trendeket színes, grafikus képernyőn akarják látni	X				X	
A munkavállaló, a pótlék és a pótlékra jogosultság különböző egyedtípusok		X		X		
A rendelésszámot 4bájtos fixpontos bináris számként tárolom			X	X		
Meghatározom a rendszer folyamatait, azok bemenetét és kimenetét		X				X
A függvény milyen algoritmussal állítja elő a bemenetből a kimenetet		X			X	
Nyilvántartásba veszem a felhasználók különféle igényeit	X					X
A közölt eljárásokat C nyelven programozom			X		X	