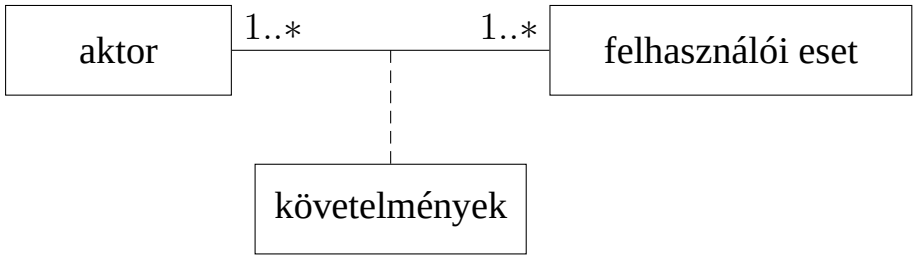
1.. Modellalkotás a programfejlesztésben

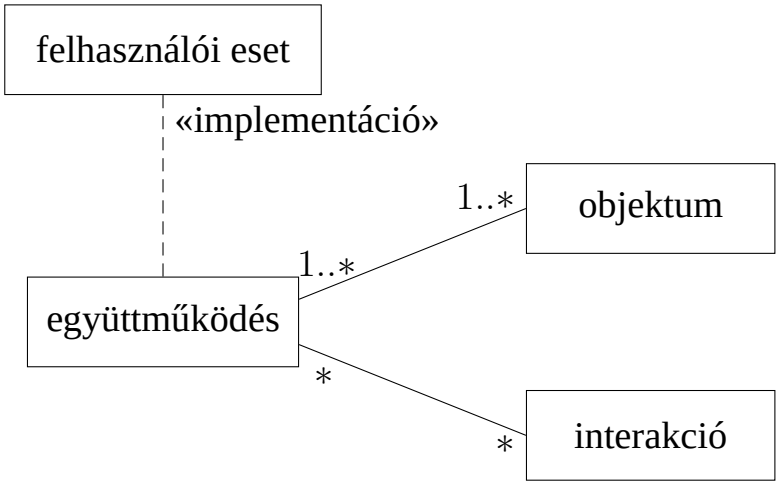
Ismétlésül foglaljuk össze milyen szerepet játszik az UML alapú modellalkotás egy programrendszer létrehozásában, melyek az egyes szakaszok. Ahol csak lehet az UML jelöléseit használjuk.

Az első szakasz a követelmények kifejtése.



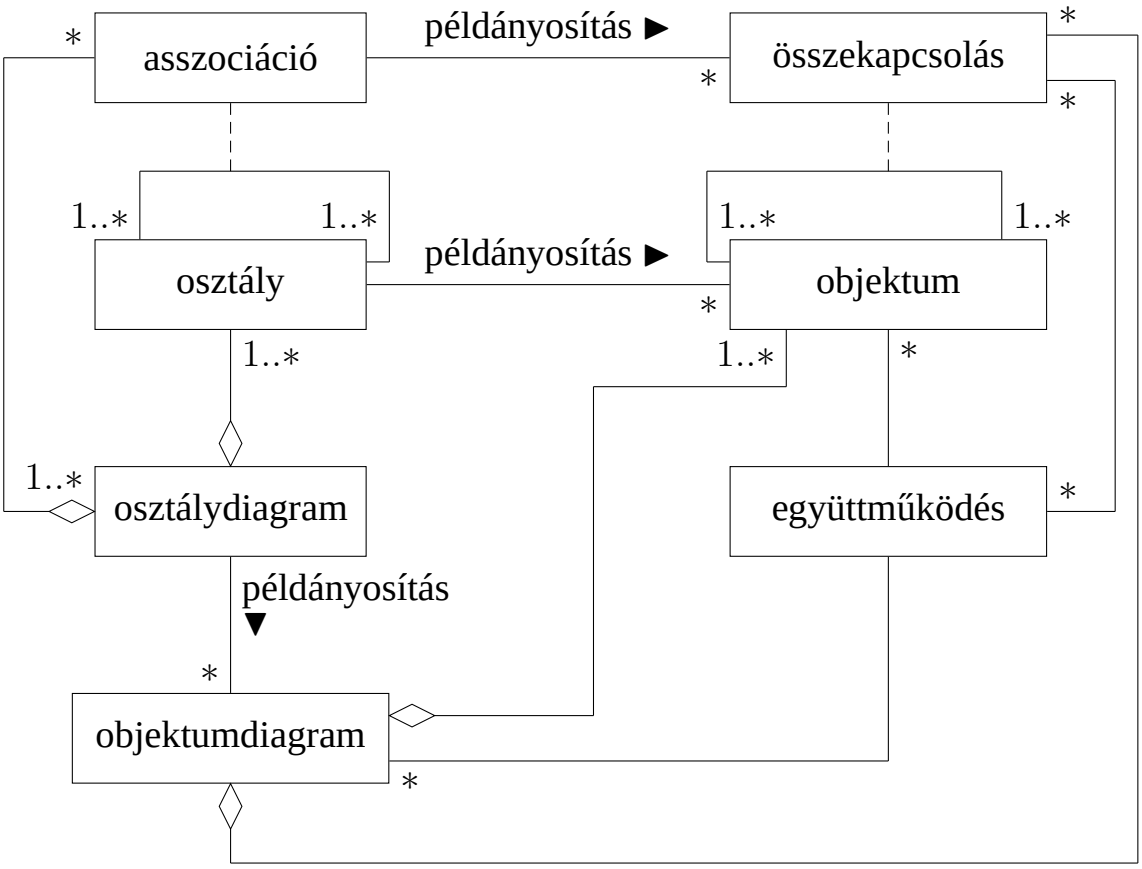
Ennek eredménye a *felhasználói esetek diagramja*, amely a rendszer funkcióit, globális szolgáltatásait és azoknak a felhasználóival (aktorokkal) együtt történő szemléltetését nyújtja. A diagramban az egyes szolgáltatásokkal (aktor – felhasználói eset összekapcsolásokkal) szemben támasztott követelményeket, elvárásokat írjuk le.

Ezután következik a felhasználói esetek objektum alapú formára alakítása. Ennek során feltárjuk a felhasználói esetekben azonosítható objektumokat, az egymásra gyakorolt hatásokat, együttműködéseket a szolgáltatás megvalósítása során.



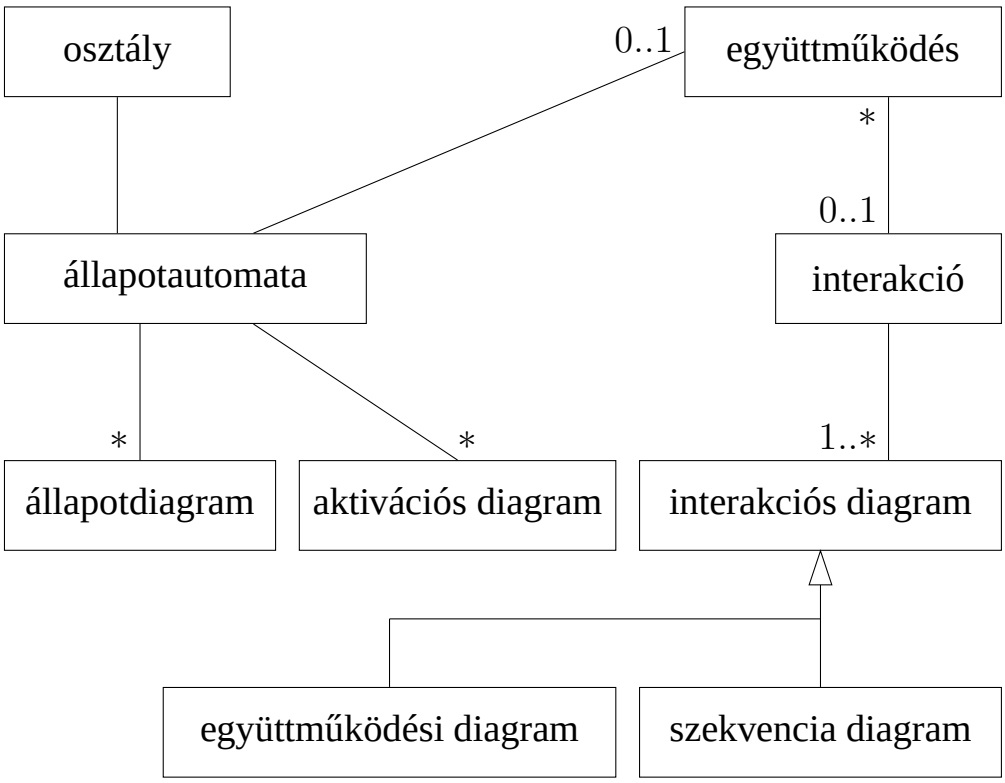
A következő lépés a megoldás szerkezetének meghatározása.

Ebben az *osztálydiagram* a probléma megoldásának statikus szerkezetét fejezi ki osztályok és az osztályok között fennálló relációk formájában. Az *objektumdiagram* pedig az osztálydiagram egy példányaként szemlélteti a probléma megoldásának statikus szerkezetét objektumok és azok összekapcsolódásainak formájában. Az objektumok közötti együttműködés az objektumdiagramban szereplő összekapcsolásokon keresztül valósulhat meg.

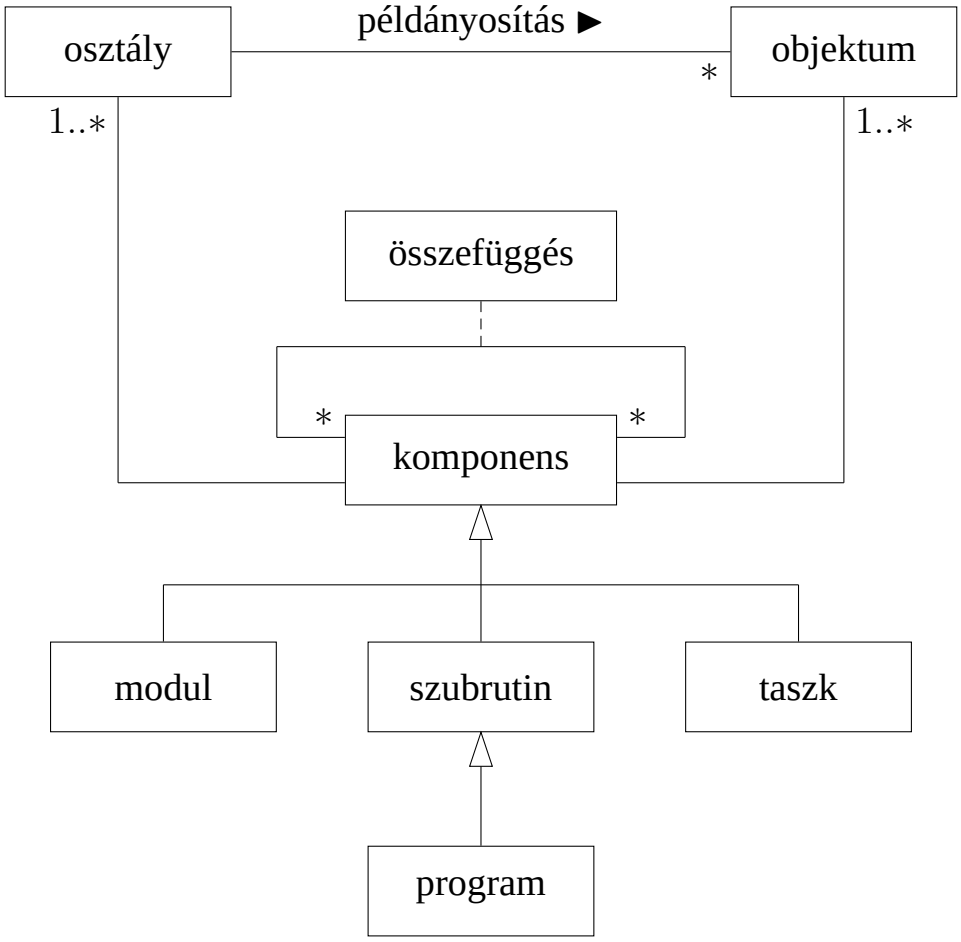

asszociáció
példányosítás ▶
összekapcsolás
*
*
*
*
1..*
1..*
1..*
1..*
osztály
példányosítás ▶
objektum
*
1..*
1..*
*
1..*
osztálydiagram
együttműködés
*
példányosítás
▼
*
objektumdiagram
*

Ezután a megoldás viselkedésének (jelentésének) meghatározása következik.

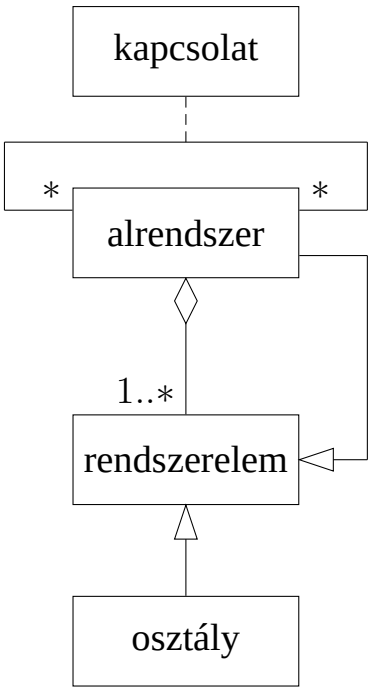
Ebben az *állapotdiagram* a probléma megoldásában részt vevő osztályok objektumainak a viselkedését mutatja állapotautomata formájában. Az *aktivációs diagram* azt szemlélteti, hogy az objektumok a probléma megoldása során időrendi sorrendben hogyan, milyen tartalmú üzenetekkel aktiválják (késztetik cselekvésre) egymást. Az objektumok közötti együttműködést az interakciós diagramok írják le. A *szekvenciadiagram* a probléma megoldása során az objektumok között időrendi sorrendben lejátszódó üzenetváltásokat szemlélteti. Az *együttműködési diagram* az objektumok közötti üzenetváltásokat szemlélteti az üzenetek tartalmára (az információ áramlásra) és nem az időbeliségre helyezve a hangsúlyt.


osztály
együttműködés
0..1
*
0..1
állapotautomata
interakció
*
*
1..*
állapotdiagram
aktivációs diagram
interakciós diagram
együttműködési diagram
szekvencia diagram

Az utolsó lépés az implementáció. A *komponensdiagram* az implementációs környezetben mutatja, az egyes komponensek helyét és egymáshoz való viszonyát.


osztály
példányosítás ▶
objektum
*
1..*
1..*
összefüggés
*
*
komponens
modul
szubrutin
taszk
program

Az *alrendszer diagram* az egységek szervezési módját szemlélteti, az alrendszerek közötti előre meghatározott kapcsolatok leírásával együtt.



A *konfigurációs diagram* azt mutatja, hogy a probléma megoldására szolgáló komponensek hol helyezkednek el a hardver környezetben.

2.. Szemléltető példa

A buszközlekedés állomások között zajlik. Minden állomást jellemez a neve. Az állomások között buszjáratok közlekednek. Minden járatot egy szám azonosít, és jellemzője a maximálisan szállítható utasok száma. Egy járat legalább két állomáson megáll, az állomások sorrendje rögzített. A buszközlekedésben utasok vesznek részt. Minden utasnak van neve, és egy kiinduló állomásról szeretne egy másik állomásra eljutni egy járattal (átszállás nélkül).

Az utasok bizonyos időközönként megérkeznek a kiinduló állomásra és ott várnak. Ha olyan buszra lehet felszállni az állomáson, amelyen még van hely és meg fog állni a célállomáson, akkor az utas felszáll. A felszállás során előbb azok az utasok szállnak fel, akik előbb érkeztek az állomásra. Ha a busz megérkezik a célállomásra, akkor az utas leszáll és befejezi a tevékenységét. Az utasok az első megfelelő buszra felszállnak, nem foglalkoznak egyéb kritériummal (menetidő, . . . stb.). A buszjáratok a menetrend szerint indulnak, azaz érkeznek meg az első állomásukra. Induláskor minden busz üres. Egy állomásra érve előbb leszállnak a megfelelő utasok a buszról, majd felszállnak az utasok a buszra. Ha a felszállás befejeződik, akkor a járat a következő állomásra megy, ahová a menetrendben megadott időpontban érkezik. Ha ez volt az utolsó állomás, akkor az utasok leszállása után a járat befejezi a tevékenységét.

- Készítsük el a buszközlekedés osztálydiagramját! Az osztályoknál adjuk meg az attribútumokat is!
- Készítsünk az osztálydiagram alapján objektumdiagramot a következő esetre! A három állomás között járnak a járatok. Az állomások nevei: a, b, c. Két buszjárat jár: az egyes számú, amelyik 2 utast szállíthat; és a kettes számú, amelyik 3 utast szállíthat. A menetrend:

idő	állomás	járat
8:00	a	1
8:15	c	1
8:20	b	2
8:25	b	1
8:30	a	2
8:35	c	2
8:40	a	1

  Az objektumdiagram az állomások és a buszok közötti kapcsolatot adja meg a menetrend alapján!
- Készítsük el a buszközlekedés állapotdiagramját!

- Készítsünk szekvenciadiagramot, amely szemlélteti az előző menetrend és a következő táblázat alapján az üzeneteket 7:55 és 8:16 között!

idő	utas	indul	cél
7:55	A	a	c
7:56	B	a	b
7:57	C	a	c
8:00	D	c	a

- Készítsünk együttműködési diagramot, amely szemlélteti azokat az üzeneteket az átadott adatokkal együtt, amelyek egy járat állomásra érkezése után kerülnek elküldésre!

- Készítsünk programot, amely alkalmas a buszközlekedés szimulálására! A program jelenítse meg a járatok, az állomások aktuális állapotait, és az utolsó időegység alatt bekövetkezett eseményeket! A forgalmi adatokat egy szövegfájlból olvassuk be, amely tartalmazza a szimuláció kezdetének időpontját, az állomások adatait, a járatok adatait, majd az utasok adatait. Az utasok a kiinduló állomásra érkezés sorrendjében szerepeljenek a fájlban!

Megoldás:

A leírás alapján három osztályt azonosíthatunk:

- állomás,
- járat,
- utas.

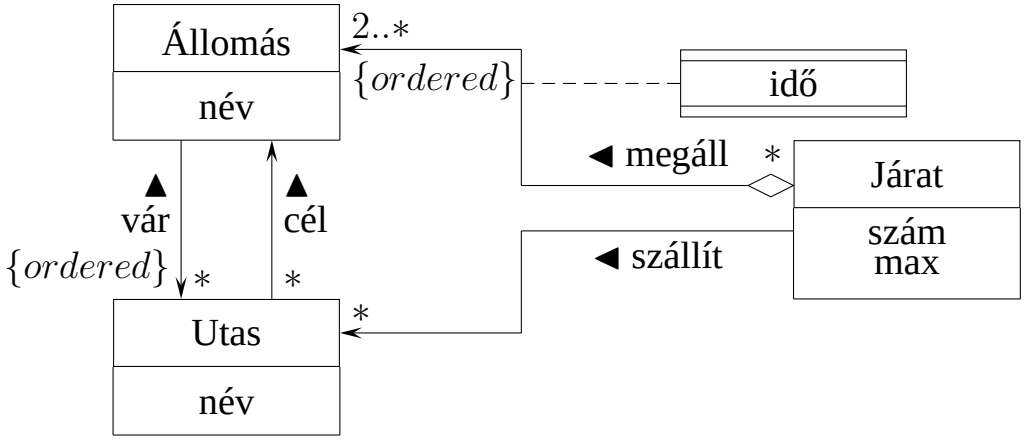
Az állomás leírásban szereplő attribútuma a név, a járaté pedig a szám és a maximálisan szállítható utasok száma.

Az osztályok közötti relációk:

- Egy járat részét képezik az állomások, mint megállók. Ez egy aggregációs kapcsolat, amelyben legalább két állomás vesz részt meghatározott sorrendben. Egy állomás ugyanakkor tetszőleges számú járathoz tartozhat. A navigálhatóság szempontjából azt mondhatjuk, hogy csak a járatnak kell ismernie a megállóit. Ugyanakkor ennek a relációnak jellemzője minden egyes esetben az állomásra érkezés időpontja. Ezt egy társult osztályban fejezhetjük ki.
- Egy járat utasokat szállít, ez asszociáció. Egy járat tetszőleges számú utast szállíthat

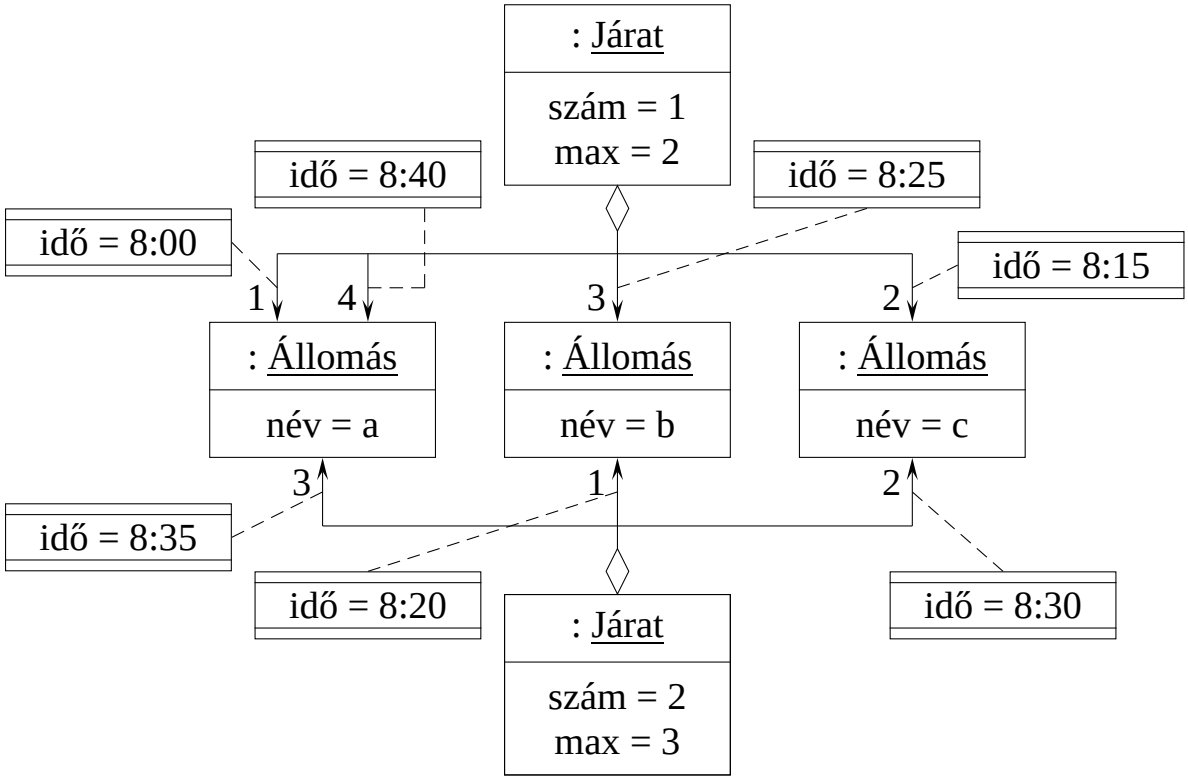
(csak egyszerre nem lehet rajta akármennyi), egy utas pontosan egy járatot használ. Itt a járat felől akarjuk elérni az utasokat.

- Az állomás és az utas osztályok között két reláció azonosítható. Egyrészt minden utas egy kiinduló állomáson várakozik, másrészt minden utasnak egy állomás a célja. Mindkét esetben egy utashoz pontosan egy állomás tartozik, és egy állomáshoz tetszőleges számú utas. A várakozás esetében az állomásról kell tudnunk elérni az ott várakozó utasokat, méghozzá az érkezés sorrendjében. A cél esetében a navigálhatóságnak az utas felől kell az állomás felé mutatnia.

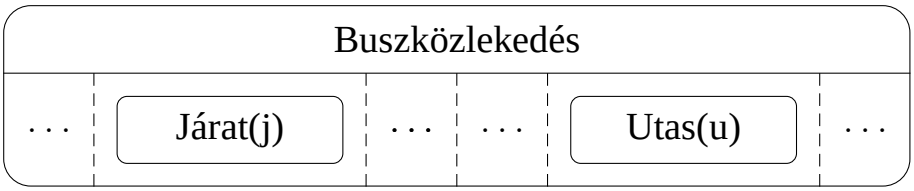


A kapcsolatok megvalósításához nyelvi szinten újabb attribútumokkal kell kiegészíteni az osztályokat. A járat esetében két, mutatókat tartalmazó sorozatra van szükség, az állomás esetében egy mutatókat tartalmazó sorozatra, az utas esetében pedig egy állomás nevére. Az attribútumok nevei egyezzenek meg a relációk neveivel. A továbbiakban így hivatkozunk ezekre.

Az objektumdiagramban 2 járat és 3 állomás objektum szerepel. A járatokat a *megáll* aggregációs kapcsolat köti össze az állomásokkal, a menetrendben szereplő időpontok adják meg a sorrendet.



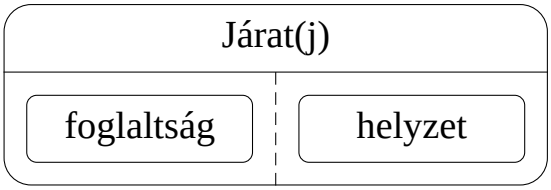
A buszközlekedés állapotait a járatok és az utasok állapotai határozzák meg.



Egy járat állapotai két részből állnak (aggregáció):

- a járat telítettsége, *foglaltság*,
- a járat helyzete, *helyzet*.

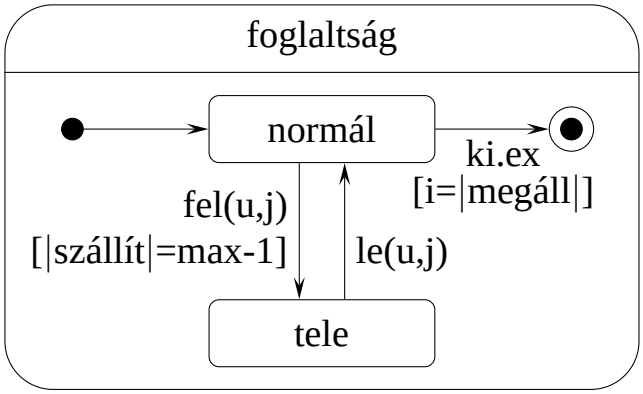
(A j paramétert a továbbiakban nem tüntetjük fel, ahol nem szükséges. Az állapotokat mindig a $Járat(j)$ állapotainak kell tekinteni.)



A foglaltságon belül két állapotot különböztethetünk meg. A *normál* állapotban van még hely a buszon, a *tele* állapotban már nincs szabad hely. Ezt a két állapothoz tartozó állapotinvariáns fejezi ki:

$$\begin{aligned} I(normál) &: |szállít| < max \\ I(tele) &: |szállít| = max \end{aligned}$$

Az állapotváltozásokat az utasok felszállása és leszállása, röviden fel és le okozza. A kezdeti állapot a normál, és ha az utolsó megállóban az utasok leszálltak (l. helyzet), akkor a járat befejezi a működését.



A járat helyzetét három állapot jellemzi:

- Az éppen aktuális állomáson kiszállnak az utasok, *ki*. Ez egy paraméteres állapot, ahol a paraméter az aktuális állomás. Vezessünk be egy új attribútumot, i, amely az aktuális állomás indexét adja meg a megállók sorozatában. Ekkor az állapot paramétere: $megáll[i]$. Az i kezdetben 1.
- Ezután az aktuális állomáson beszállnak az utasok, $be(megáll[i])$.
- A beszállás befejezése után a járat a következő állomásra *megy*. Ennek az entry fázisában i értéke eggyel nő.

A járat kezdeti állapota a $ki(megáll[i])$, hiszen kezdetben $i = 1$. A járat akkor fejezi be a működését, amikor az utolsó megállóban kiszálltak az utasok, azaz $ki(megáll[i]).ex[i = |megáll|]$.

A be és a ki paraméteres állapotokat is egy-egy invariáns jellemzi. Az állapotok egészen addig fennmaradnak, amíg az utasok le illetve fel akciói meg nem szüntetik az invariánst. Az állapotokból történő kilépésnél ($.ex$) az áttekinthetőség kedvéért nem tüntetjük fel a paramétert az állapotdiagramban.

Az invariánsok felírásához vezessünk be néhány jelölést:

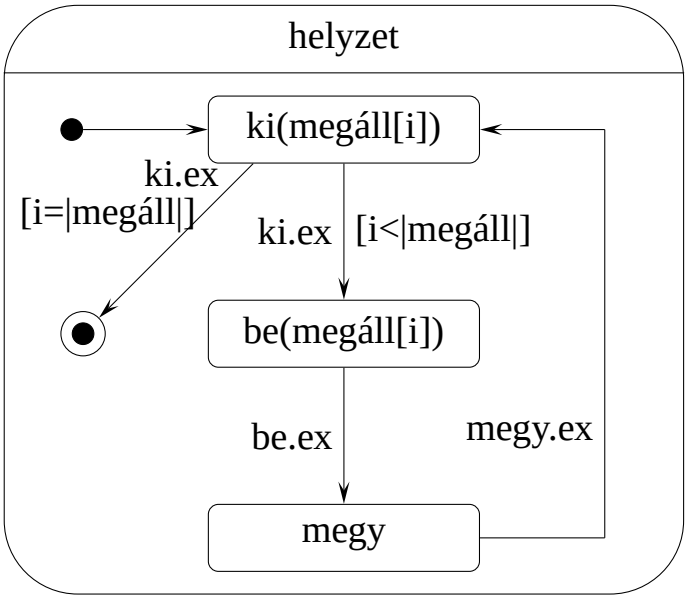
- az u utas indexe az a állomás várakozási sorában legyen $\sigma(u, a)$;
- a járat hátralévő megállóinak halmaza legyen μ;
- az a állomáson várakozó első i utas céljainak a halmaza legyen $\omega(a, i)$.

A definíciók formálisan:

$$\begin{aligned} \sigma(u,a) &: a.vár[\sigma(u,a)] = u \\ \mu &= \{megáll[k] \mid k \in [i+1..|megáll|]\} \\ \omega(a,i) &= \{a.vár[k].cél \mid k \in [1..i]\} \end{aligned}$$

Ezek felhasználásával felírhatóak az állapotinvariánsok. A ki állapot addig áll fenn, amíg van a járaton olyan utas, aki az aktuális állomásra jött. A be esetében azt kell figyelni, hogy van még hely és vár olyan utas az állomáson, aki oda akar menni ahová a járat. A $megy$ állapotból a megfelelő idő letelte után lépünk ki, azaz ebben maradunk, amíg az aktuális idő, t, kisebb az érkezési időnél.

$$
\begin{aligned}
I(ki(a)) &: \exists k \in [1..|szállít|] : szállít[k].cél = a \\
I(be(a)) &: in\ normál \wedge \omega(a, |a.vár|) \cap \mu \neq \emptyset. \\
I(megy) &: t < megáll[i].idő\ .
\end{aligned}
$$



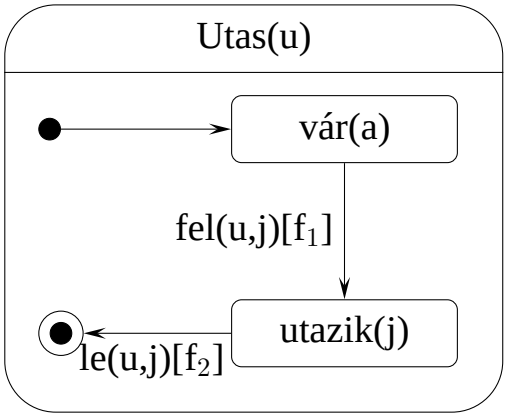
Egy utasnak két állapotát különböztethetjük meg:

- a kezdeti a állomáson várakozik, $vár(a)$,
- a j járaton utazik, $utazik(j)$.

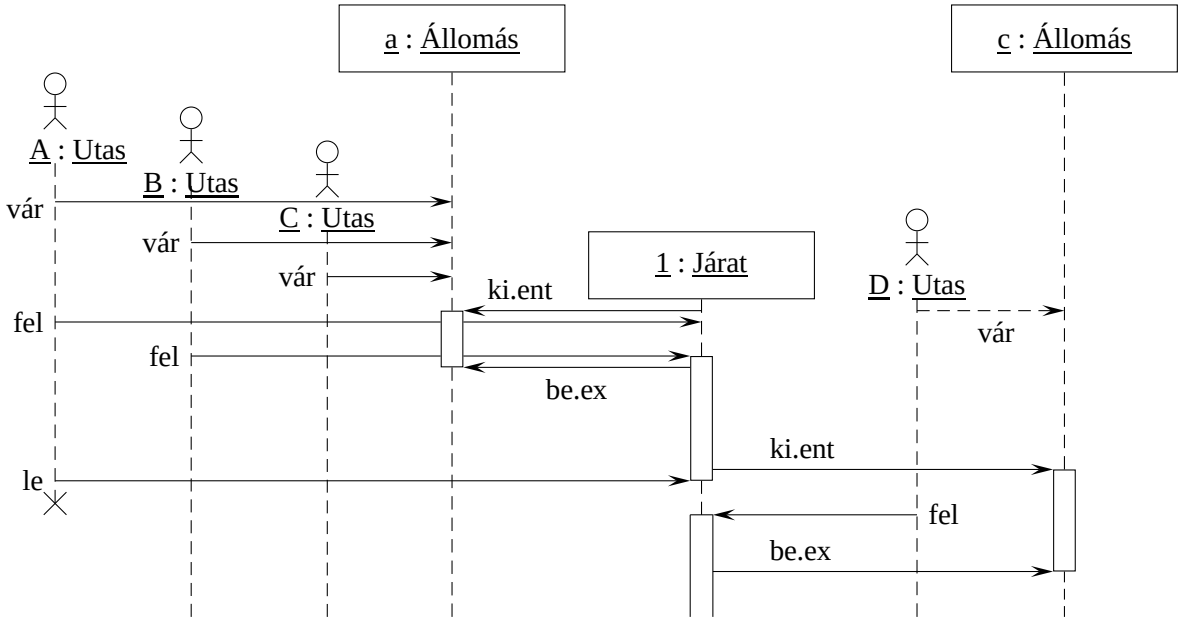
A kezdeti állapot a várakozás, amelyből a fel akció hatására megy át az utazásba. Ennek feltétele, hogy az adott járatra a megfelelő állomáson fel lehet szállni (be), a járat megáll az utas célállomásán és az utas előtt nincs olyan utas a várakozók között, aki oda menne, ahová a járat. Ezt fejezi ki az f_1 feltétel. Az utazást a le akció hatására fejezi be az utas, amelynek feltétele, hogy a járat a célállomáson legyen a leszállási állapotban (ki). Ezt fejezi ki az f_2 feltétel. Ezután az utas befejezi a tevékenykedését.

Az állapotátmenetek feltételei:

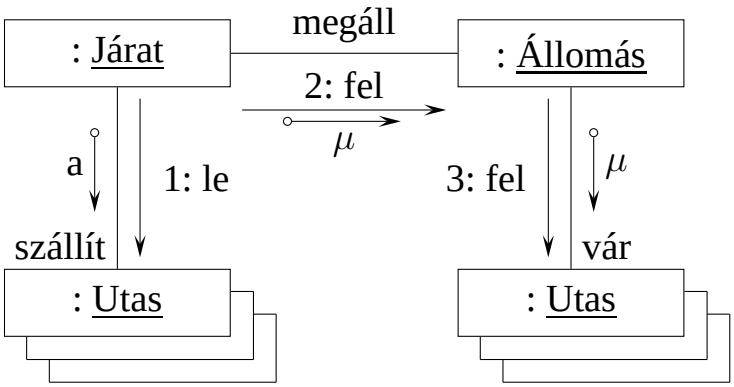
$$
\begin{aligned}
f_1 \;&:\; járat(j)\ in\ be(a) \wedge u.cél \in járat(j).\mu \wedge \\
&\quad\ \omega(a, \sigma(u,a) - 1) \cap járat(j).\mu = \emptyset \\
f_2 \;&:\; járat(j)\ in\ ki(u.cél)
\end{aligned}
$$


Utas(u)
vár(a)
fel(u,j)[f1]
utazik(j)
le(u,j)[f2]

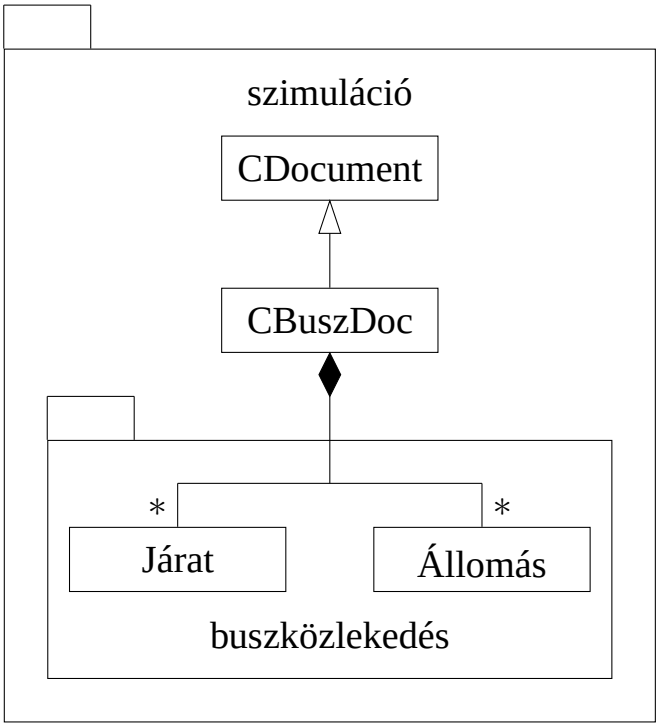
A szekvenciadiagramban a megadott táblázatban szereplő 4 utas, az egyes számú járat és az a és c állomás objektumok szerepelnek. Az utasok aktor szerepet töltenek be. Az állomás aktív szerepe legyen az, amikor ott busz tartózkodik, a járat aktív szerepe pedig az, amikor tele van.



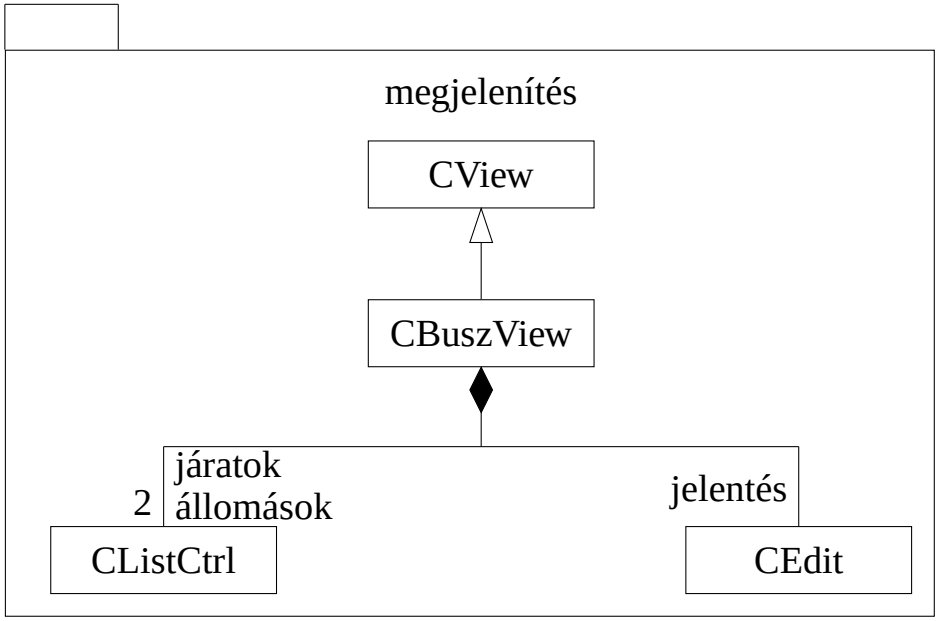
Az együttműködési diagramban fel kell tüntetni a járat objektumot, az általa szállított utasoknak megfelelő objektumokat (ezek száma tetszőleges), az aktuális állomást és az ott várakozó utasokat. A járat és az első utascsoport közötti összekapcsolás a szállít reláció, amelynek mentén küldi a járat az első üzenetet az utasoknak. Az üzenet a le, az átadott adat pedig az állomás neve. A járatot az állomással a megáll reláció kapcsolja össze, és a járat az állomásnak küldi a második üzenetet, fel, amelyhez a hátralévő állomások halmaza társul adatként. Ezt az üzenetet az állomás továbbítja az adattal együtt a várakozó utasoknak, ez lesz a harmadik üzenet. A le illetve a fel üzenetekre az utasok megfelelően, a saját akcióikkal, reagálnak.



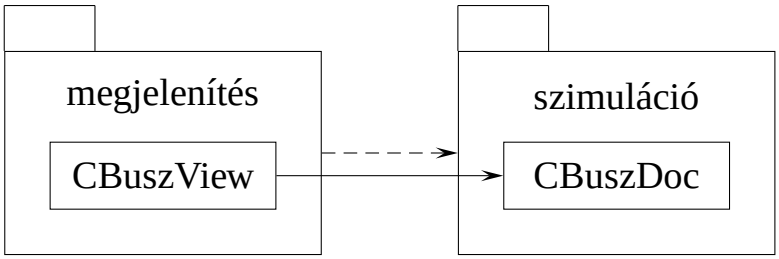
A szimulációs programot Windows operációs rendszerre, Visual C++ környezetben készítjük el. A program a MFC által biztosított alrendszerekre bomlik. Az egyik az *adatok* kezelése, azaz a tulajdonképpeni szimuláció, ami a *CDocument* osztályból származtatott *CBuszDoc* osztályban valósul meg.



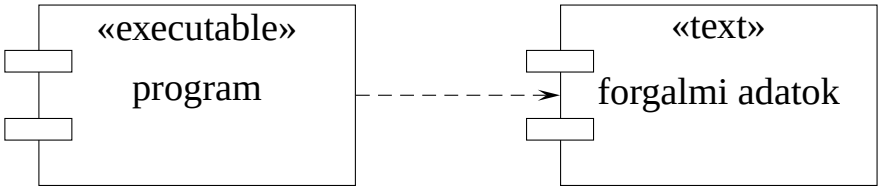
A másik alrendszer a megjelenítés, ami a *CView* osztályból származtatott *CBuszView* osztály feladata. Ebben szerepel az utolsó időegység alatt lejátszódott események leírása, *jelentés*; a járatok listája, *járatok*; és az állomások listája, *állomások*. A jelentés egy szerkesztőablakban, *CEdit*, tartalmazza a megfelelő szöveget, a két lista egy-egy report stílusú *CListCtrl*.



A két alrendszer alkotja a programot. A megjelenítő alrendszer használja a szimulációs alrendszer adatait a CBuszView és a CBuszDoc osztályok közötti kapcsolaton keresztül.



A program alkotja a rendszer egyik komponensét. A másik komponens a forgalom adatait (állomások, járatok, utasok) tartalmazó fájl.



Az eddigieket felhasználva készítsük el a programkódot! Itt most csak a szimulációval foglalkozunk, a megjelenítés a megfelelő MFC eszközök felhasználásával viszonylag egyszerűen kivitelezhető.

A programban mutatókból álló sorozatokat kell tudnunk kezelni az osztálydiagramban szereplő relációk ábrázolásához. Szükséges utasok és állomások mutatóit is kezelnünk. Ennek érdekében vezessünk be egy sablonosztályt, amelynek paramétere a mutató alaptípusa, és rendelkezik a megfelelő műveletekkel (elem hozzávétele, lekérdezés, eltávolítás, . . .). A sorozat nem szabadítja fel a mutatókat, ez a felhasználó felelőssége. Ez lehetőséget ad arra, hogy a mutatókat egyik sorozatból a másikba helyezzük át, illetve arra is, hogy egy mutató több sorozatban is szerepelhessen. Ez utóbbi tulajdonság különösen fontos lesz a járatok és az állomások közötti *megáll* reláció megvalósításánál, hiszen több járat is megáll ugyanazon az állomáson. A leírtaknak megfelelő sablonosztály legyen a $Sequence$ osztály, amelynek megvalósítása a következő.

```
#define UNIT 256

class SeqErr
{
public:
    SeqErr(const CString msg)   { message = "Seq: " + msg; }
    ~SeqErr()                   {};
    SeqErr(const SeqErr &src)   { message = src.message; }
    CString Message() const     { return(message); }
protected:
    CString    message;
};

template <class Item> class Sequence
{
public:
    Sequence();
    virtual ~Sequence();
    void Add(Item *i);
    Item *operator[](unsigned int i);
```

```
    Item *operator[](unsigned int i) const;
    void RemoveAt(unsigned  int i);
    void RemoveAll();
    void InsertAt(Item *i, unsigned int index);
    int  Size() const            { return(size); }
protected:
    Item            **content;
    unsigned int    size;
    unsigned int    act_max;
    void    Resize();

    Sequence(const Sequence &src)              {};
    Sequence &operator=(const Sequence &src)   {};
};

template <class Item> Sequence <Item>::Sequence()
{
    content = new Item*[UNIT];
    act_max = UNIT;    size = 0;
}
```

```
template <class Item> Sequence <Item>::~Sequence()
{
    delete [] content;
}

template <class Item> Item * Sequence <Item>::
        operator[](unsigned int i)
{
    if ( i >= size )    throw SeqErr("Invalid index in []");
    return(content[i]);
}

template <class Item> Item * Sequence <Item>::
        operator[](unsigned int i) const
{
    if ( i >= size )    throw SeqErr("Invalid index in []");
    return(content[i]);
}
```

```
template <class Item> void Sequence <Item>::RemoveAll()
{
    size = 0;
}

template <class Item> void Sequence <Item>::
        RemoveAt(unsigned int i)
{
    if ( i >= size ) throw SeqErr("Invalid index in RemoveAt");
    for ( unsigned int k = i; k < size - 1; k++ )
        content[k] = content[k + 1];
    size--;
}

template <class Item> void Sequence <Item>::Add(Item *i)
{
    if ( size == act_max )    Resize();
    content[size] = i;
    size++;
}
```

```
template <class Item> void Sequence <Item>::
        InsertAt(Item *i, unsigned int index)
{
    if ( index > size )
        throw SeqErr("Invalid index in InsertAt");
    if ( size == act_max )  Resize();
    size++;
    for ( unsigned int k = size; k > index; k--)
        content[k] = content[k - 1];
    content[index] = i;
}

template <class Item> void Sequence <Item>::Resize()
{
    unsigned int    i;
    Item            **tmp;
    tmp = new Item*[size];
    for ( i = 0; i < size; i++ )    tmp[i] = content[i];
    delete [] content;
```

```
    act_max += UNIT;
    content = new Item*[act_max];
    for ( i = 0; i < size; i++ )    content[i] = tmp[i];
}
```

Az utasoknak megfelelő *Utas* osztály implementációja adódik az osztálydiagramból.

```
class Utas
{
public:
    Utas(const CString &n, const CString &c);
    virtual ~Utas()          {};
    CString Nev() const      { return(nev); }
    CString Cel() const      { return(cel); }
protected:
    CString    nev;
    CString    cel;
};

Utas::Utas(const CString &n, const CString &c)
{
    nev = n;    cel = c;
}
```

Az *Állomás* osztály esetében a *vár* relációt a bevezetett *Sequence* osztály segítségével valósítjuk meg. Az ehhez kapcsolódó műveletek: a várakozó utasok számának lekérdezése, az $i.$ várakozó utas lekérdezése, az $i.$ utas felszállása egy adott buszra, új utas felvétele a várakozási sorba.

```
class Allomas
{
public:
    Allomas(const CString &n)   { nev = n; }
    virtual ~Allomas();
    CString Nev() const     { return(nev); }
    int Utasszam() const    { return(var.Size()); }
    Utas *Var(int i)        { return(var[i]); }
    Utas *Fel(int i);
    void UjUtas(Utas *u)    { var.Add(u); }
protected:
    CString           nev;
    Sequence<Utas>    var;
};
```

```
Allomas::~Allomas()
{
    for ( int i = 0; i < var.Size(); i++ )  delete var[i];
}

Utas *Allomas::Fel(int i)
{
    Utas    *u = var[i];
    var.RemoveAt(i);
    return(u);
}
```

A *megáll* reláció megvalósításához be kell vezetnünk egy új osztályt, legyen ez *Megálló*, amely megadja az állomás mutatóját és az érkezés idejét. A járatok csak ezen keresztül érhetik el az állomásokat, ezért az állomások megfelelő műveleteit itt biztosítanunk kell.

```
class Megallo
{
public:
    Megallo(Allomas *a, int mikor) { ido = mikor; all = a; }
    virtual ~Megallo()      {};
    CString Nev() const     { return(all->Nev()); }
    int Utasszam() const    { return(all->Utasszam()); }
    Utas *Var(int i)        { return(all->Var(i)); }
    Utas *Fel(int i)        { return(all->Fel(i)); }
    int Ido() const         { return(ido); }
protected:
    Allomas *all;
    int     ido;
};
```

A *Járat* osztály reprezentációjában szereplő attribútumok és a vonatkozó műveletek adódnak az osztálydiagramból. Ezt ki kell egészíteni egy olyan művelettel, amely egy idő múlását jelzi. Legyen ez a művelet az *Órajel*. Ennek paramétere az aktuális időpillanat, és egy jelentés, amelyhez a járattal kapcsolatos tevékenységeket kell hozzávennünk. A járat ki illetve be állapotának (leszállás és felszállás) feleljen meg a Le illetve Fel művelet. Miután ezek állapotnak felelnek meg, ezért ezek belső műveletek. Az állapot paramétere az aktuális állomás, ami a sorozat első eleme. (A már érintett állomásokat elhagyjuk a sorozatból). Egy másik segédművelet, $Tartalmaz$, feleljen meg $cél \in \mu$ feltételnek, azaz annak, hogy az adott állomás szerepel-e a megmaradt úticélok között. A műveletek implementációja adódik az állapotdiagramból, illetve egy egyszerű lineáris keresés a tartalmazás esetében.

```
class Jarat
{
public:
    Jarat(int s, int m)          { szam = s; max = m; }
    virtual ~Jarat();
    int Utasszam() const         { return(szallit.Size()); }
    int Megalloszam() const      { return(megall.Size()); }
    void UjMegallo(Megallo *m)   { megall.Add(m); }
```

```
    void Orajel(int ido, CString &jelent);
    int Szam() const            { return(szam); }
    Megallo *Megall(int i)      { return(megall[i]); }
    Utas *Szallit(int i)        { return(szallit[i]); }
    int Max() const             { return(max); }
protected:
    int         szam;
    int         max;
    Sequence<Megallo>   megall;
    Sequence<Utas>      szallit;

    void Le(CString &jelent);
    void Fel(CString &jelent);
    bool Tartalmaz(const CString &s) const;
};

Jarat::~Jarat()
{
    int    i;
    for ( i = 0; i < szallit.Size(); i++ )
```

```
        delete szallit[i];
    szallit.RemoveAll();
    for ( i = 0; i < megall.Size(); i++ )
        delete megall[i];
    megall.RemoveAll();
}

void Jarat::Orajel(int ido, CString &jelent)
{
    if ( ido < megall[0]->Ido() )   return;
    Le(jelent);
    if ( megall.Size() > 1 )    Fel(jelent);
    megall.RemoveAt(0);

void Jarat::Le(CString &jelent)
{
    CString info;
    CString linefeed;
    linefeed.FormatMessage("\n");
    int     i = 0;
```

```
    while ( i < szallit.Size() )
    {
        if ( szallit[i]->Cel() == megall[0]->Nev() )
        {
            jelent += szallit[i]->Nev() + " utas leszállt ";
            info.Format("%d buszról ", szam);
            jelent += info;
            jelent += szallit[i]->Cel() + " állomáson";
            jelent += linefeed;
            delete szallit[i];
            szallit.RemoveAt(i);
        }
        else    i++;
    }
}

void Jarat::Fel(CString &jelent)
{
    CString    info;
    CString linefeed;
```

```
linefeed.FormatMessage("\n");
Utas    *u;
int     i = 0;
while ( i < megall[0]->Utasszam() )
{
    if ( szallit.Size() == max )
    {
        info.Format("%d busz megtelt", szam);
        jelent += info + linefeed;
        return;
    }
    if ( Tartalmaz(megall[0]->Var(i)->Cel()) )
    {
        u = megall[0]->Fel(i);
        szallit.Add(u);
        jelent += u->Nev() + " utas felszállt ";
        info.Format("%d buszra ", szam);
        jelent += info;
        jelent += megall[0]->Nev() + " állomáson";
        jelent += linefeed;
```

```
        }
        else
            i++;
    }
}

bool Jarat::Tartalmaz(const CString &s) const
{
    for ( int i = 1; i < megall.Size(); i++ )
        if ( megall[i]->Nev() == s )    return(true);
    return(false);
}
```

A *CBuszDoc* osztály tartalmazza a szimuláció adatait és a megjelenítéshez szükséges adatokat. Ez az osztály felelős az idő kezeléséért is. Ennek megfelelően a következőkkel kell kiegészítenünk a fejlesztőkörnyezet által létrehozott osztályt. Le kell tudnunk kérdezni az aktuális időponthoz tartozó eseményeket (jelentés), a járatokat és az állomásokat. Reprezentálnunk kell az aktuális időt, az állomásokat, a járatokat, az adatokat tartalmazó fájlt és a jelentést. Az adatok esetében tudnunk kell, hogy mi a soron következő utas állomásra érkezési ideje, így akkor olvasunk be adatot, ha szükséges. Belső műveletek az idő kezelése, ezen belül a érkező utasok beolvasása, illetve az elején az állomások és a járatok beolvasása.

```
class CBuszDoc : public CDocument
{
...
public:
    CString Jelentes() const    { return(jelentes); }
    int    Jaratszam() const    { return(jaratok.Size()); }
    Jarat *Jaratok(int i)       { return(jaratok[i]); }
    int Allomasszam() const     { return(allomasok.Size()); }
    Allomas *Allomasok(int i) const { return(allomasok[i]); }
    //{{AFX_VIRTUAL(CBuszDoc)
```

```
    public:
    virtual void Serialize(CArchive& ar);
    virtual void DeleteContents();
    //}}AFX_VIRTUAL
protected:
    int                 ido;
    Sequence<Allomas>   allomasok;
    Sequence<Jarat>     jaratok;
    ifstream            adat_file;
    bool                nyitott_adat_file;
    int                 adat_ido;
    CString             jelentes;
protected:
    void Orajel();
    void UtasOlvasas();
    void AllomasOlvasas();
    void JaratOlvasas();
    int  AllomasIndex(const CString &n) const;
...
};
```

Az adatokat tartalmazó fájl megnyitásakor be kell olvasni a kezdeti időt, az állomásokat és a járatokat, majd az első utas érkezési idejét.

```
void CBuszDoc::Serialize(CArchive& ar)
{
    if (ar.IsLoading())
    {
        CString    fnev = ar.GetFile()->GetFileName();
        adat_file.open(fnev);
        nyitott_adat_file = true;
        adat_file >> ido;
        AllomasOlvasas();
        JaratOlvasas();
        adat_file >> adat_ido;
    }
}
```

Egy szimuláció befejezésekor fel kell szabadítani a lefoglalt területet, és bezárni az adat-fájlt, ha az nyitott.

```
void CBuszDoc::DeleteContents()
{
    int         i;
    if ( nyitott_adat_file )    adat_file.close();
    nyitott_adat_file = false;
    for ( i = 0; i < jaratok.Size(); i++ )
        delete jaratok[i];
    for ( i = 0; i < allomasok.Size(); i++ )
        delete allomasok[i];
    jaratok.RemoveAll();
    allomasok.RemoveAll();
    CDocument::DeleteContents();
}
```

Egy időegység leteltekor be kell olvasni az esetlegesen érkező utasokat, a járatoknak el kell küldeni a megfelelő üzenetet, és a megszűnt járatokat törölni kell.

```
void CBuszDoc::Orajel()
{
    int    i;
    ido++;
    if ( ido % 100 == 60 )    ido += 40;    // 760 -> 800
    CString    info;
    CString linefeed;
    linefeed.FormatMessage("\n");
    info.Format("Idő: %d", ido);
    jelentes = info + linefeed + linefeed;
    UtasOlvasas();
    for ( i = 0; i < jaratok.Size(); i++ )
        jaratok[i]->Orajel(ido, jelentes);
    i = 0;
    while (i < jaratok.Size() )
        if ( jaratok[i]->Megalloszam() == 0 )
        {
```

```
            delete jaratok[i];
            jaratok.RemoveAt(i);
        }
        else    i++;
}
```

Az olvasó műveletek és az adatellenőrzést biztosító segédművelet implementációja értelemszerű.

```
void CBuszDoc::UtasOlvasas()
{
    char    tmp[80];
    CString un, start, cel;
    CString linefeed;
    linefeed.FormatMessage("\n");
    Utas    *u;
    int     index;
    while ( !adat_file.eof() && adat_ido == ido )
    {
        adat_file >> tmp;
```

```
        un = tmp;
        adat_file >> tmp;
        start = tmp;    start.Replace('_', ' ');
        adat_file >> tmp;
        cel = tmp;      cel.Replace('_', ' ');
        index = AllomasIndex(start);
        if ( index == -1 )
            MessageBox(NULL, "Hibás kiinduló állomás",
                       "Hiba", MB_OK);
        else
        {
            u = new Utas(un, cel);
            allomasok[index]->UjUtas(u);
            jelentes += u->Nev() + " utas buszra vár " + start;
            jelentes += " állomáson. Célja: " +
                        u->Cel() + linefeed;
        }
        adat_file >> adat_ido;
    }
}
```

```
void CBuszDoc::AllomasOlvasas()
{
    char    tmp[80];
    CString nev;
    Allomas *a;
    int     index;
    adat_file >> index;
    while ( !adat_file.eof() && index > 0 )
    {
        adat_file >> tmp;
        nev = tmp;  nev.Replace('_', ' ');
        a = new Allomas(nev);
        allomasok.Add(a);
        index--;
    }
}

void CBuszDoc::JaratOlvasas()
{
```

```
char    tmp[80];
CString nev;
int     index, jsz, jmax, msz, jido, ai;
Jarat   *j;
Megallo *m;
adat_file >> index;
while ( !adat_file.eof() && index > 0 )
{
    adat_file >> jsz >> jmax;
    j = new Jarat(jsz, jmax);
    adat_file >> msz;
    while ( msz > 0 )
    {
        adat_file >> jido;
        adat_file >> tmp;
        nev = tmp;  nev.Replace('_', ' ');
        ai = AllomasIndex(nev);
        if ( ai == -1 )
        {
            MessageBox(NULL, "Hibás megálló",
```

```
                        "Hiba", MB_OK);
            }
            else
            {
                m = new Megallo(allomasok[ai], jido);
                j->UjMegallo(m);
            }
            msz--;
        }
        jaratok.Add(j);
        index--;
    }
}

int CBuszDoc::AllomasIndex(const CString &n) const
{
    for ( int i = 0; i < allomasok.Size(); i++ )
        if ( allomasok[i]->Nev() == n )    return(i);
    return(-1);
}
```