

# Objektumok viselkedése

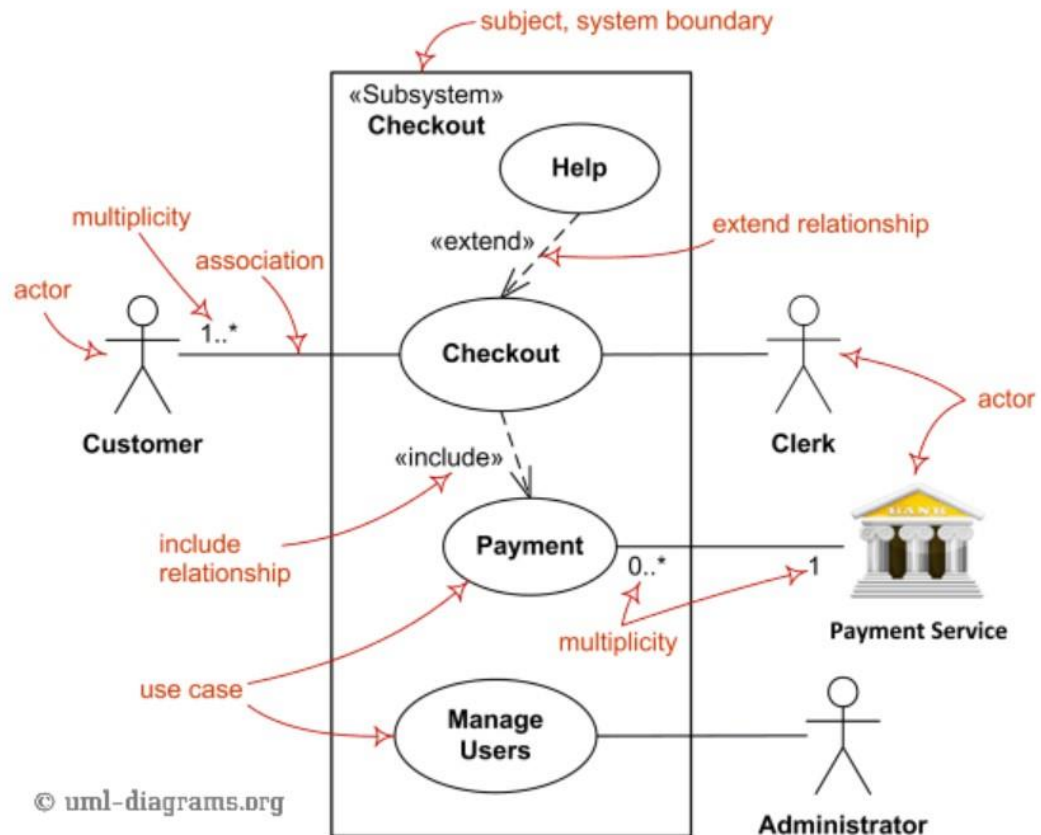
UML viselkedési nézetei

# Viselkedési nézetek

- ❑ Az UML az objektumok dinamikus viselkedésének jellemzésére is számos nézetet vezetett be. Ezek közül az alábbiakkal ismerkedünk majd meg:
  - Használati eset (*use case*) diagram
  - Kommunikációs (*communication*) diagram
  - Szekvencia (*sequence*) diagram
  - Állapotgép (*state machine*) diagram

# Használati eset diagram

- ❑ Megmutatja, hogy a tervezett rendszernek
- mi a célja,
  - milyen funkcionalitást kell biztosítania, azaz mire lesz képes,
  - milyen követelményeket támaszt a környezetével szemben.



# Használati esetek kapcsolatainak specifikátorai

## ❑ Használati esetek rákövetkezési sorrendje.

- **precede**: felhasználó által közvetlenül kezdeményezhető két tevékenység között biztosítandó sorrend
- **invoke**: egy felhasználói tevékenységet követő, de közvetlenül nem előidézhető tevékenység jelölése

## ❑ Használati eset kiegészítése.

- **include**: egy felhasználó tevékenységnek egy jól elkülöníthető, önállóan is kezdeményezhető része, amely nélkül azonban a tartalmazó tevékenység nem teljes (absztrakt).
- **extend**: egy felhasználói tevékenységet opcionálisán kiegészítő másik tevékenység, amely önmagában is teljes (soha nem absztrakt)

## ❑ Tevékenységek vagy aktorok között jelölhető leszármaztatási viszony

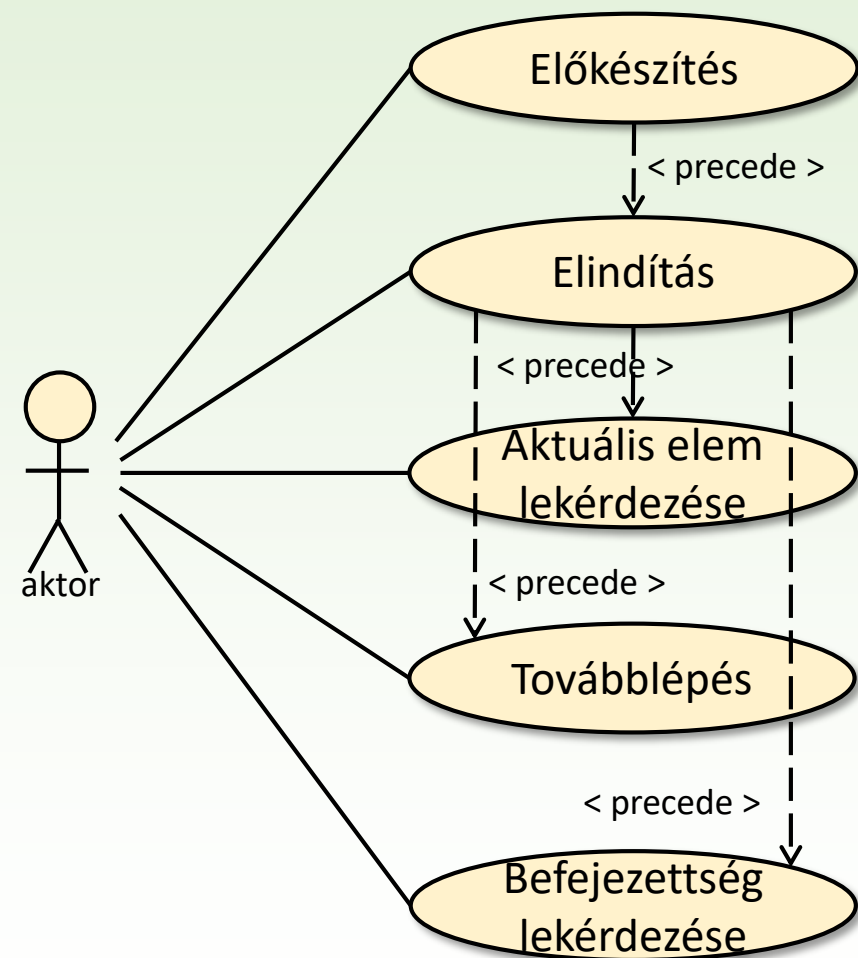
## ❑ Multiplicitás is jelölhető.

# Felhasználói esetek (user story)

- ❑ A használati eset diagram önmagában nem ad elégséges képet a megvalósítandó rendszerről.
- ❑ A felhasználói esetek **felhasználói csoportonként** („AS a ...”) történő táblázatos („user story”) leírásában minden felhasználói tevékenységet részletesen ki kell fejteni:
  - mi a tevékenység **neve**,
  - milyen **előfeltétel** meglétét feltételezi (GIVEN)
  - milyen **eseménynek** a hatására következik be (WHEN)
  - mi a **hatása** a végrehajtásának, milyen eredményt ad (THEN).

AS a ...		
eset		leírás
tevékenység	GIVEN	tevékenység kiváltásakor feltételezett alaphelyzet
	WHEN	tevékenység kiváltása
	THEN	tevékenység hatása

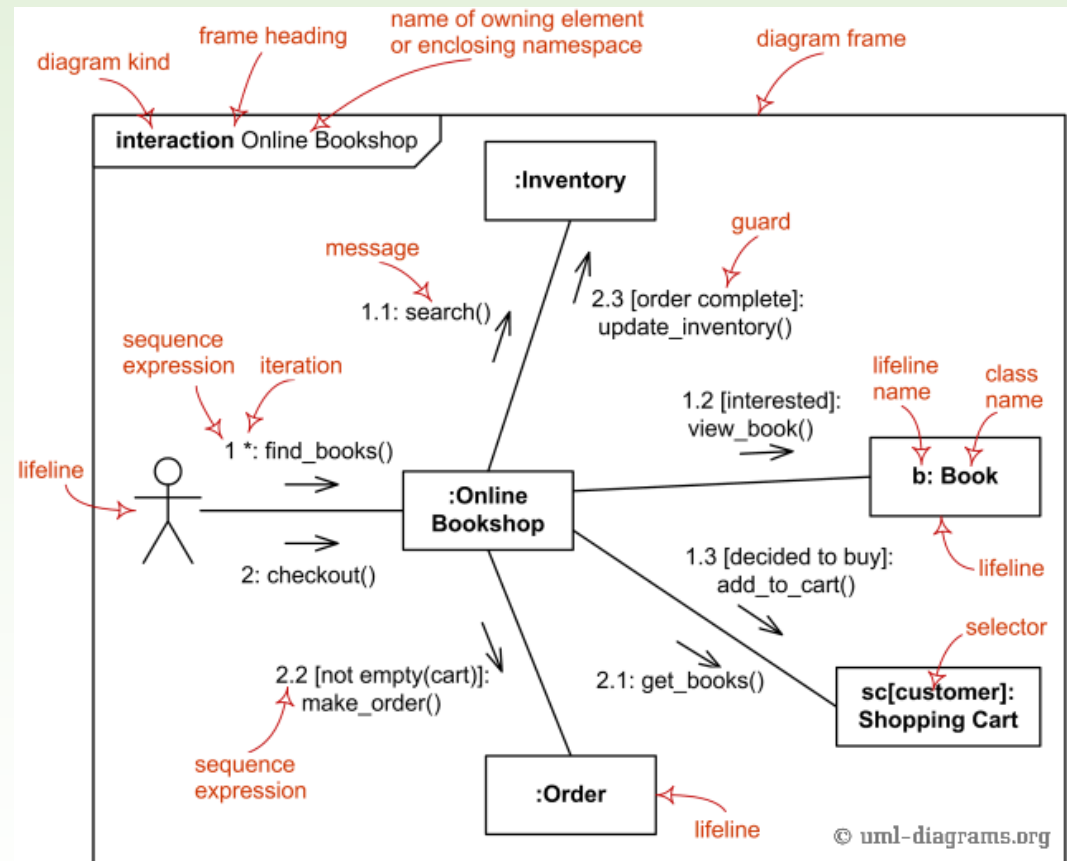
# Példa: Felsorolás



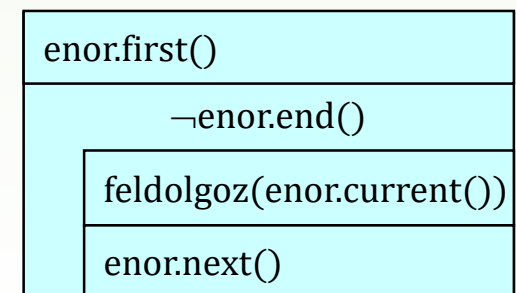
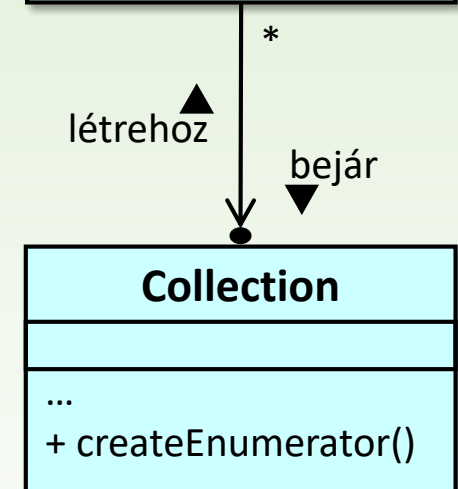
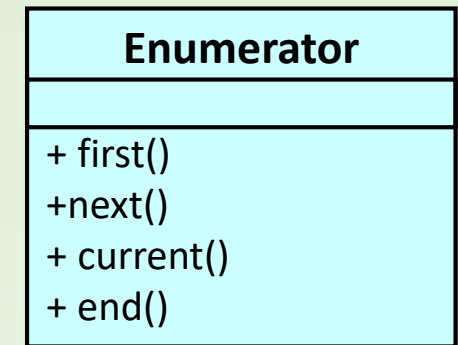
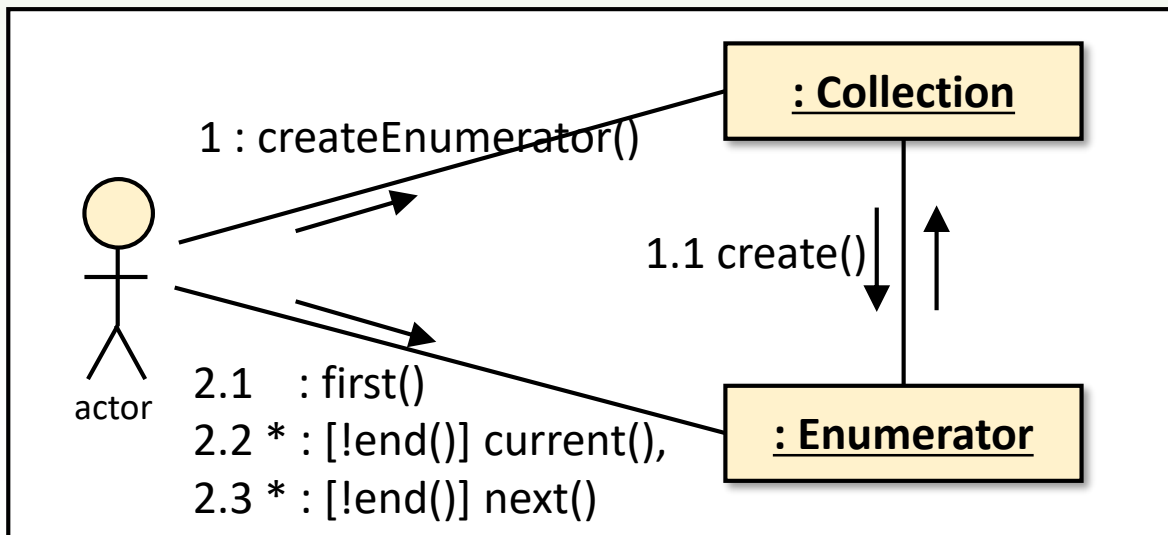
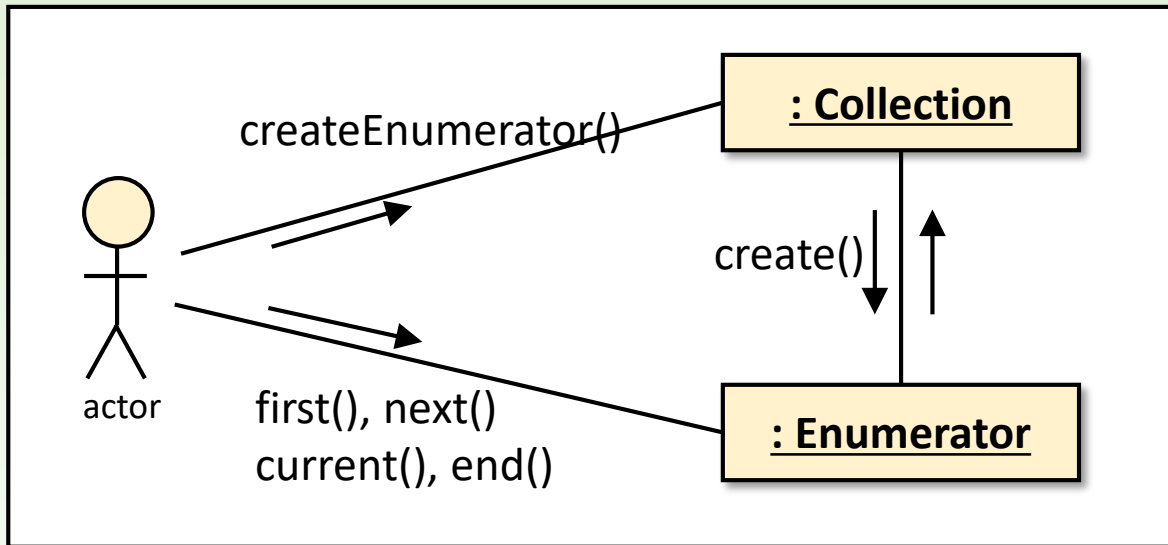
eset		leírás
Előkészítés normális esetben	GIVEN	Adott a felsorolni kívánt gyűjtemény.
	WHEN	Felsoroló példányosítása.
	THEN	Létrejön a felsoroló objektum.
Előkészítés abnormális esetben	GIVEN	Nincs felsorolni kívánt gyűjtemény.
	WHEN	Felsoroló példányosítása.
	THEN	Hiba, felsoroló objektum nem jön létre.
Elindítás normális esetben	GIVEN	Adott egy még el nem indított (pre-start) állapotú felsoroló objektum.
	WHEN	Felsorolás elindítása a <b>first()</b> művelettel.
	THEN	A felsoroló folyamatban van (in-process) állapotba kerül.
Elindítás abnormális esetben	GIVEN	Adott egy folyamatban levő (in-process) vagy befejeződött (finished) felsoroló.
	WHEN	Felsorolás elindítása a <b>first()</b> művelettel.
	THEN	Hiba, és a felsoroló megőrzi állapotát.
...		

# Kommunikációs diagram

- ❑ A kommunikációs diagram azt mutatja meg, hogy az **objektumok** milyen **üzenetekkel** (metódus-hívások, szignál-küldések) kommunikálnak egymással.
- ❑ Lehetőséget ad az üzenetek **sorrendjének** kijelölésére (sorszámozással), illetve **előfeltételek** megadására szögletes zárójelpár között.



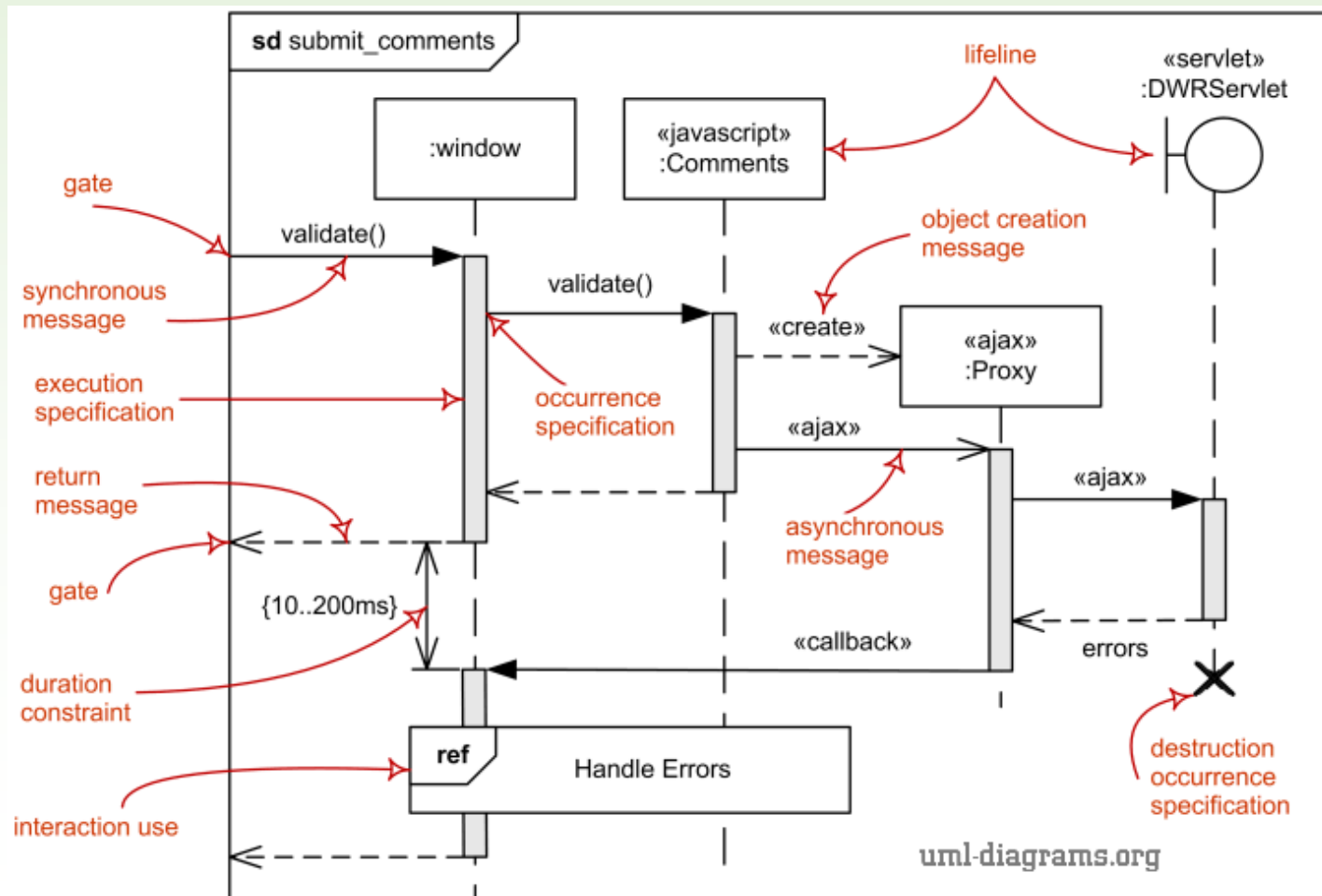
# Példa: Felsorolás





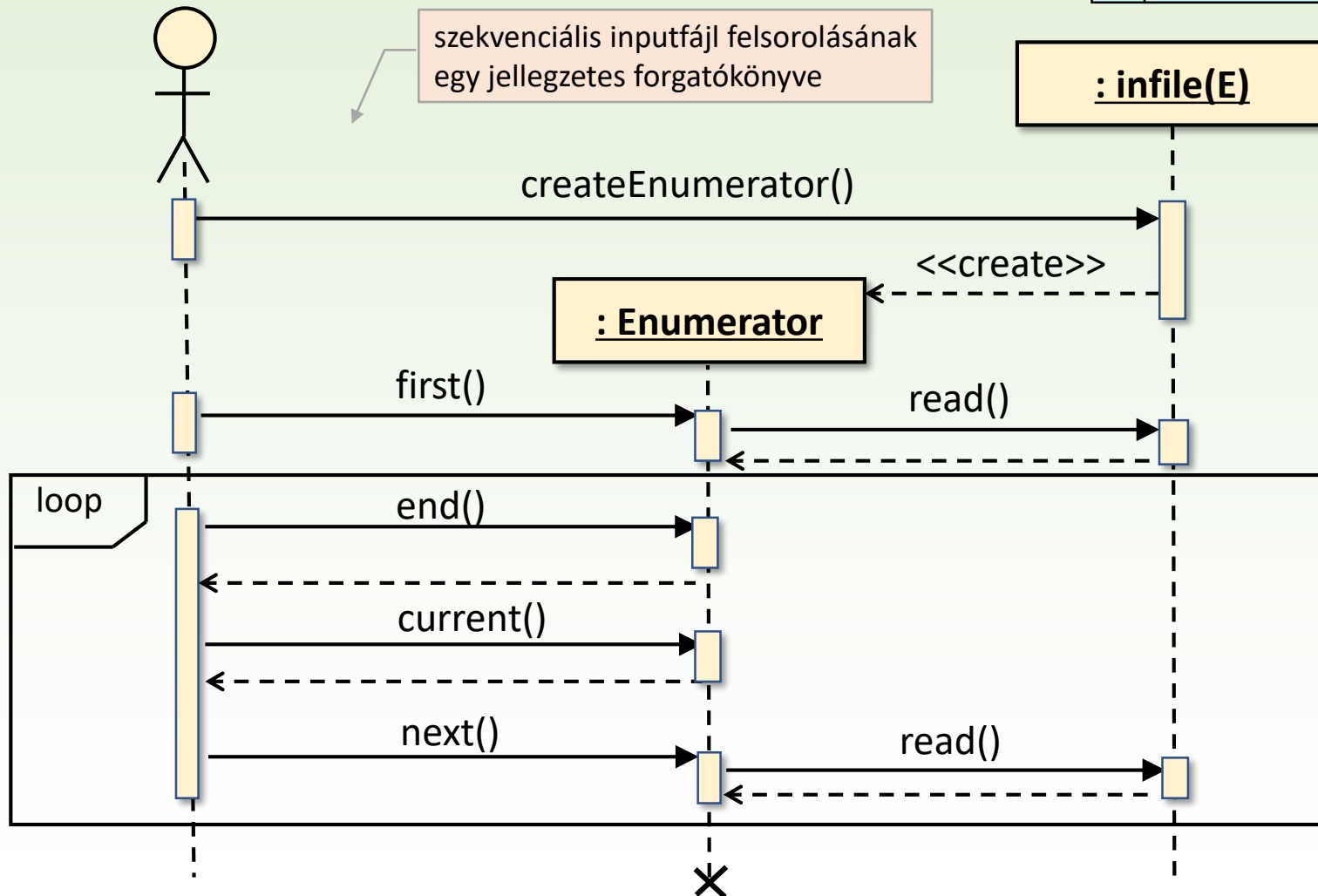
# Szekvencia diagram

- A kommunikációban az üzenetváltások időbeli sorrendjét mutatja.

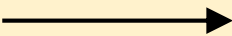
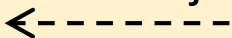



# Példa: Felsorolás

enor.first()
¬enor.end()
feldolgoz(enor.current())
enor.next()



# Üzenetek

- ❑ Az **egyszerű üzenet** az, amikor a küldő objektum átadja a vezérlést a fogadó objektumnak. Ez tulajdonképpen egy közös (szinkron) eljárás hívás. 
- ❑ A **visszatérési üzenet** egy korábbi üzenet hatásának lezárásaként, ezen korábbi üzenet fogadó objektuma küldi vissza a küldő objektumnak akkor, amikor a fogadó objektum befejezi a tevékenységét, és visszaadja a vezérlést. Ezt az üzenetet sokszor nem is tüntetjük fel az ábrán, mert kiolvasható a szerepe a környezetből. 
- ❑ Az objektumok konkurens működése esetén van jelentősége az alábbi üzenet fajtáknak: 
  - Az **aszinkron üzenet** során a küldő objektum tevékenysége nem szakad meg, nem érdekli őt, hogy mikor kapja meg a fogadó objektum az üzenetet.
  - A **szinkronizációs üzenet** a küldő objektum tevékenységét blokkolja mindaddig, amíg a fogadó objektum nem fogadta az üzenetet.
  - Az **időhöz kötött várakozó üzenet** során a küldő objektum a megjelölt ideig várakozik arra, hogy a fogadó objektum fogadja az üzenetet.
  - A **randevú üzenet** esetén a fogadó objektum várakozik arra, hogy a küldő objektum üzenetet küldjön neki.

# Objektum élelciklusa

## ❑ Az objektum **élelciklusa** során:

- létrejön: az objektum speciális műveletével, a **konstruktorral**,
- működik: más objektumokkal **kommunikál**, azaz szinkron vagy aszinkron módon hívják egymás műveleteit, vagy az egymásnak küldött jelzésekre (*signal*) reagálnak aszinkron módon, és ennek során **megváltozhatnak az adatai**,
- megsemmisül (egy másik speciális művelettel, a **destruktorral**).

## ❑ Egy objektumnak különféle **állapotai** (*state*) vannak: egy állapot (fizikai állapot) az objektum adatai által felvett értékek együttese, amely az élelciklus során változik.

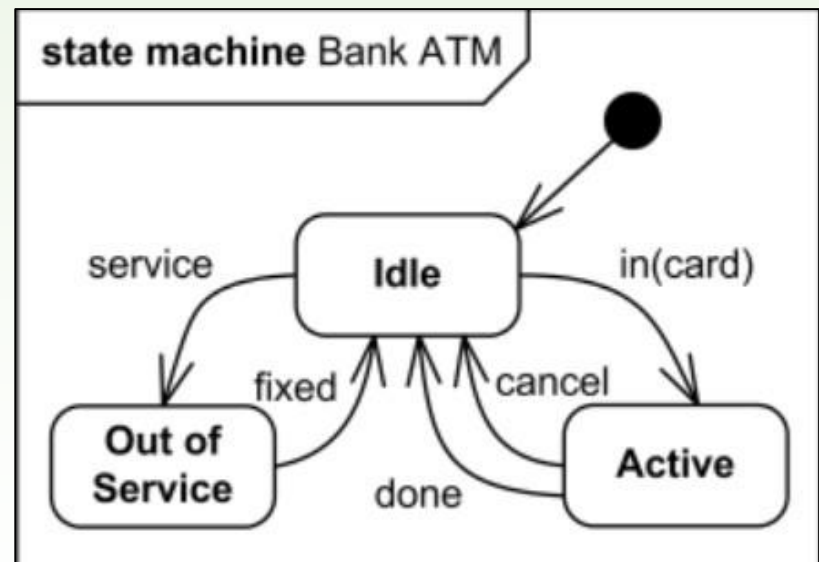
## ❑ A könnyebb áttekinthetőség kedvéért azonban gyakran egy állapotnak az objektum több különböző, de közös tulajdonságú fizikai állapotainak összességét tekintjük (logikai állapot) .

# Állapotgép

□ Az állapotgép diagram az **objektum életciklusát**, viselkedését ábrázolja. Megmutatja, hogy mi módon változik az objektum belső (logikai) állapota az objektumnak küldött üzenetek (metódus hívások vagy szignálok) hatására.

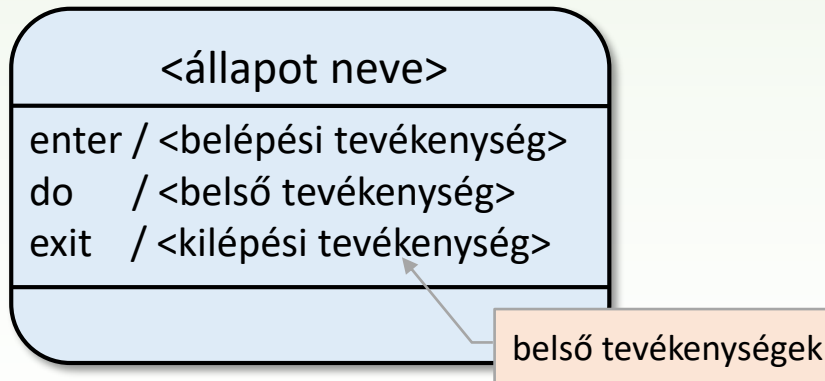
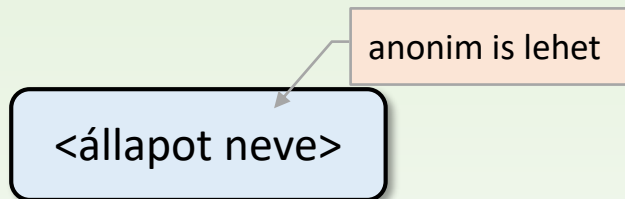
□ Az állapotgép egy irányított gráf, amelynek csomópontjai a logikai állapotokat, irányított élei pedig az állapotok közötti átmeneteket mutatja.

□ Mind az állapotokhoz, mind az átmenetekhez tartozhatnak végrehajtandó tevékenységek.



# Állapotok jelölése

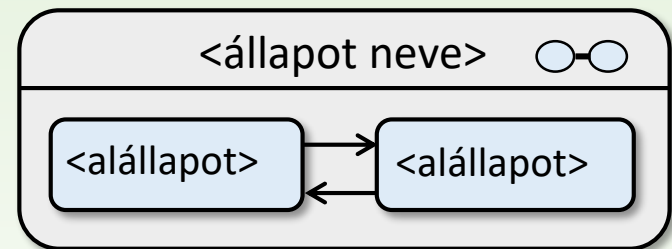
- Egyszerű állapot



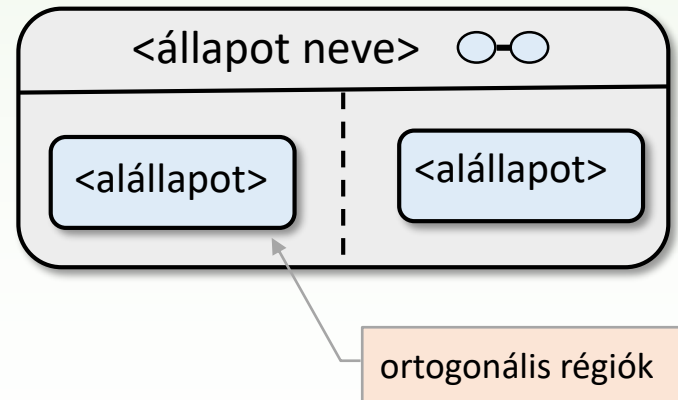
- Összetett állapot

hierarchikus

Szekvenciálisan:



Párhuzamosan:



# Pszedo állapotok

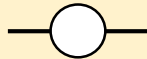
- Kezdő állapot



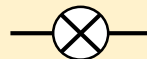
- Végállapot



- Belépési pont



- Kilépési pont



- Terminálás



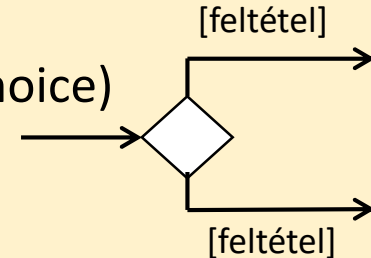
- Shallow history



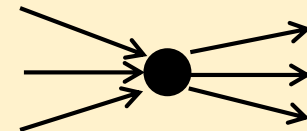
- Deep history



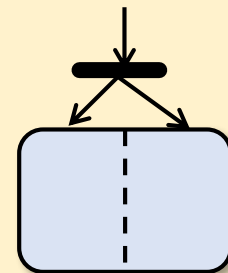
- Elágazás (choice)



- Csomópont (junction)



- Szétágazás (fork)



- Összefutás (join)



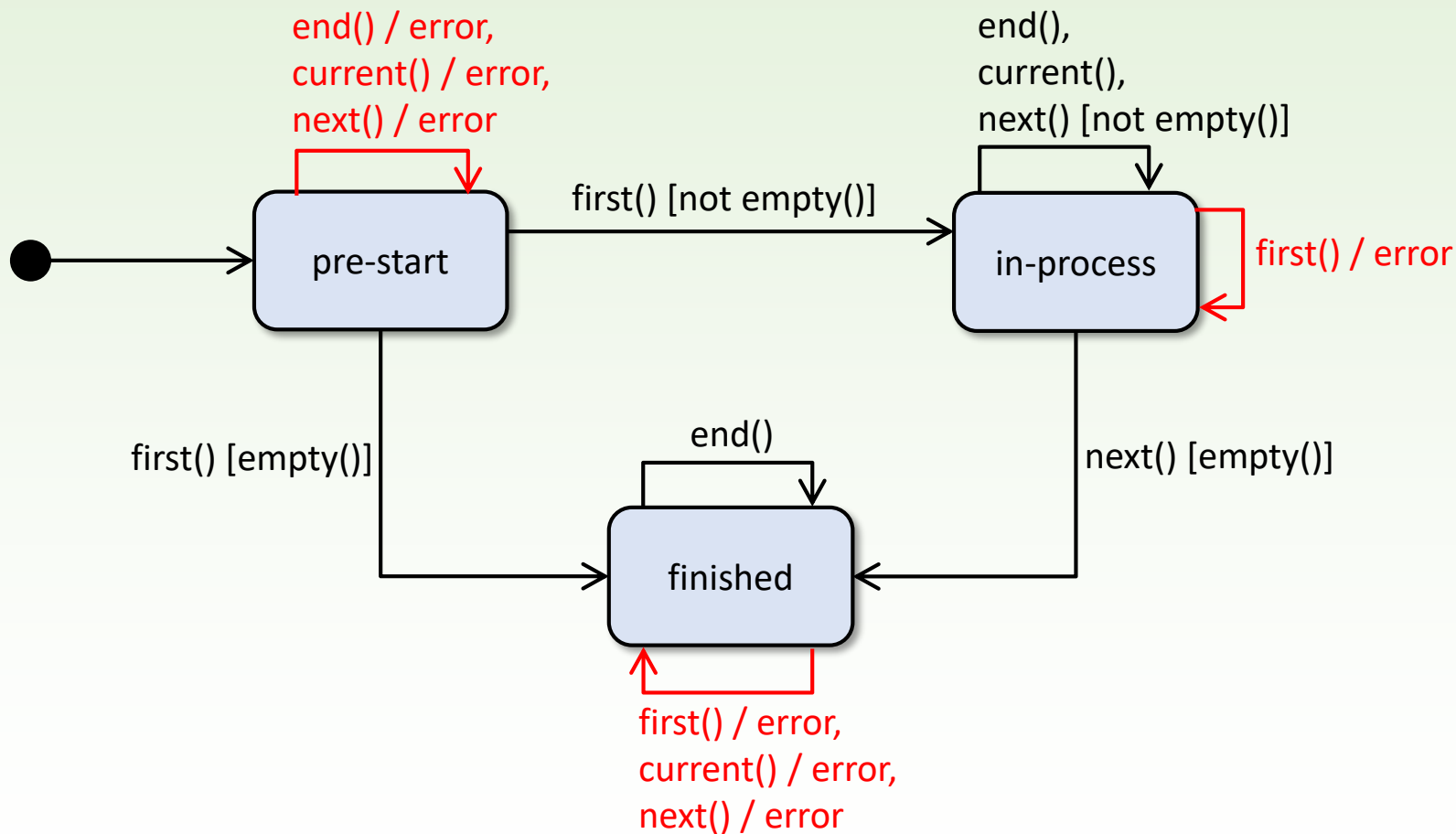
# Állapot-átmenetek jelölése

- ❑ Az átmenet leírásának elemei, amelyek közül bármelyik elhagyható:
  - az átmenetet **előidéző események** (*event, trigger*) **paramétereikkel**
    - vagy az adott objektum szinkron metódus-hívása
    - vagy egy annak küldött aszinkron módon feldolgozott szignál
  - bekövetkezését szükségszerűen megelőző **őrfeltétel** (*guard*), amely
    - vagy paramétereiktől függő logikai állítás (*when*)
    - vagy időhöz kötött várakozási feltétel (*after*)
  - az átmenethez rendelt **tevékenység** (az objektum adattagjaival és a kiváltó esemény paramétereivel operáló program)
  - rövid magyarázó **leírás** (gyakran hiányzik)
- ❑ Egy átmenet lehet reflexív (belső), amely során az állapot nem változik, és nem hajtódik végre az állapot enter és exit tevékenysége sem.





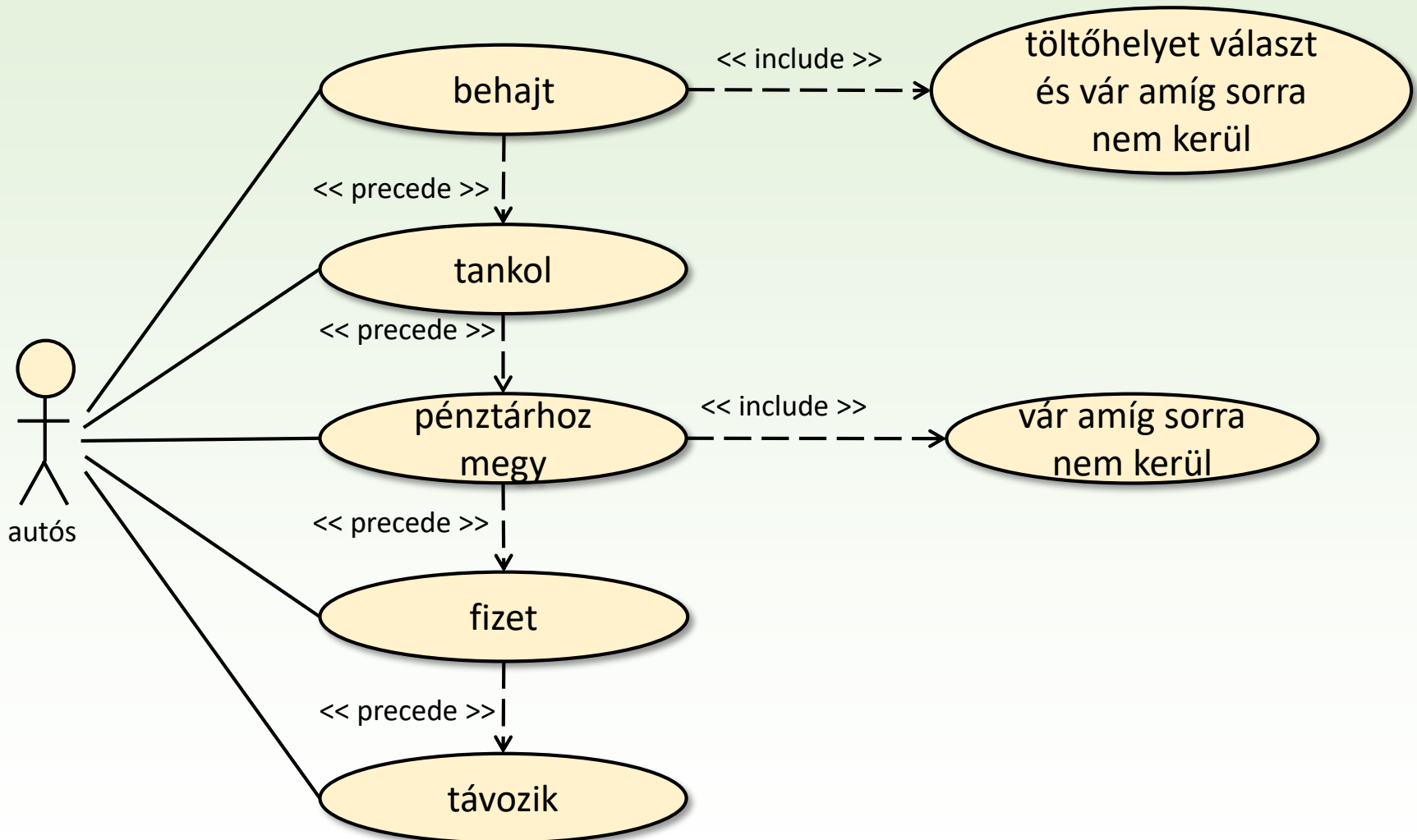
# Példa: Felsorolás



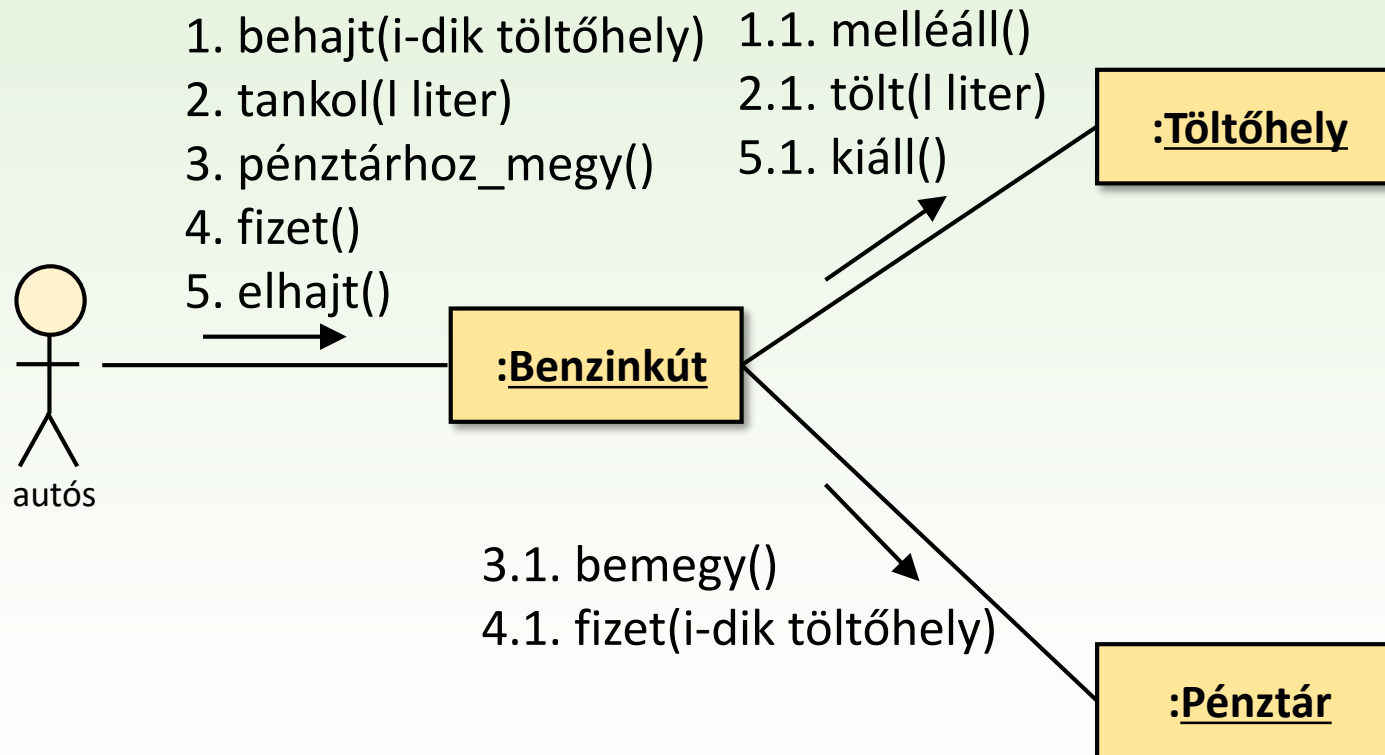
# Feladat

- ❑ Egy benzinkúton  $n$  darab töltőhely és több kasszából álló pénztár található. Az autósok behajtanak a benzinkúthoz, beállnak valamelyik töltőhelyhez tankolni. Amikor sorra kerülnek, akkor kívánt mennyiségű benzint töltenek a járművük benzintartályába. Ezután elmennek fizetni. A pénztárhoz egyetlen sor áll. Amint egy kassa szabad lesz, kiszámolja a soron következő autós által tankolt mennyiség alapján a fizetendő összeget. Fizetés után az autós kihajt a töltőhelyről.
- ❑ Modellezzük ezt a folyamatot tetszőleges számú, egymással párhozamosan tevékenykedő autós esetére.

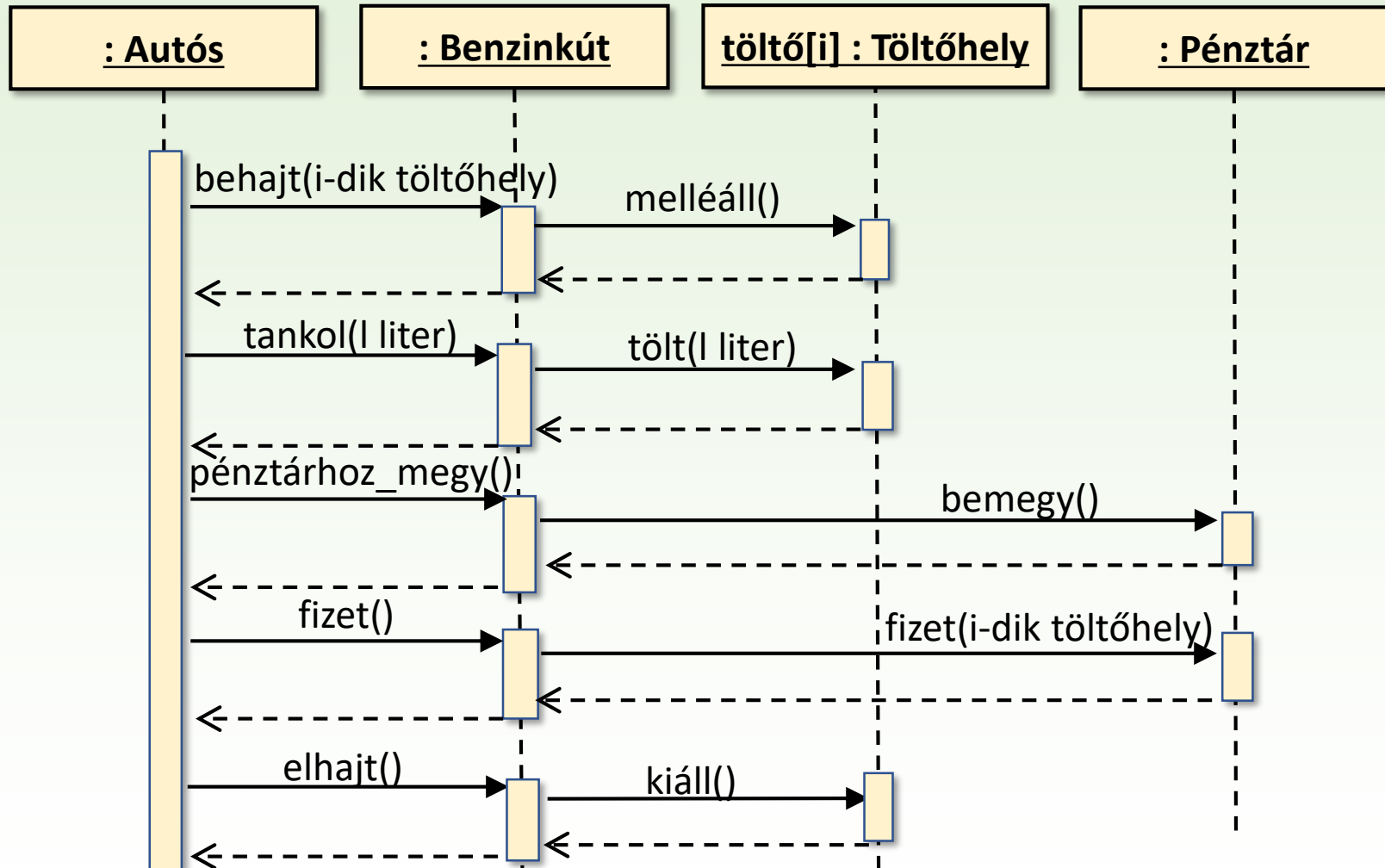
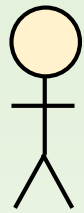
# Használati eset diagram



# Kommunikációs diagram



# Szekvencia diagramm



# Elemzés eredménye

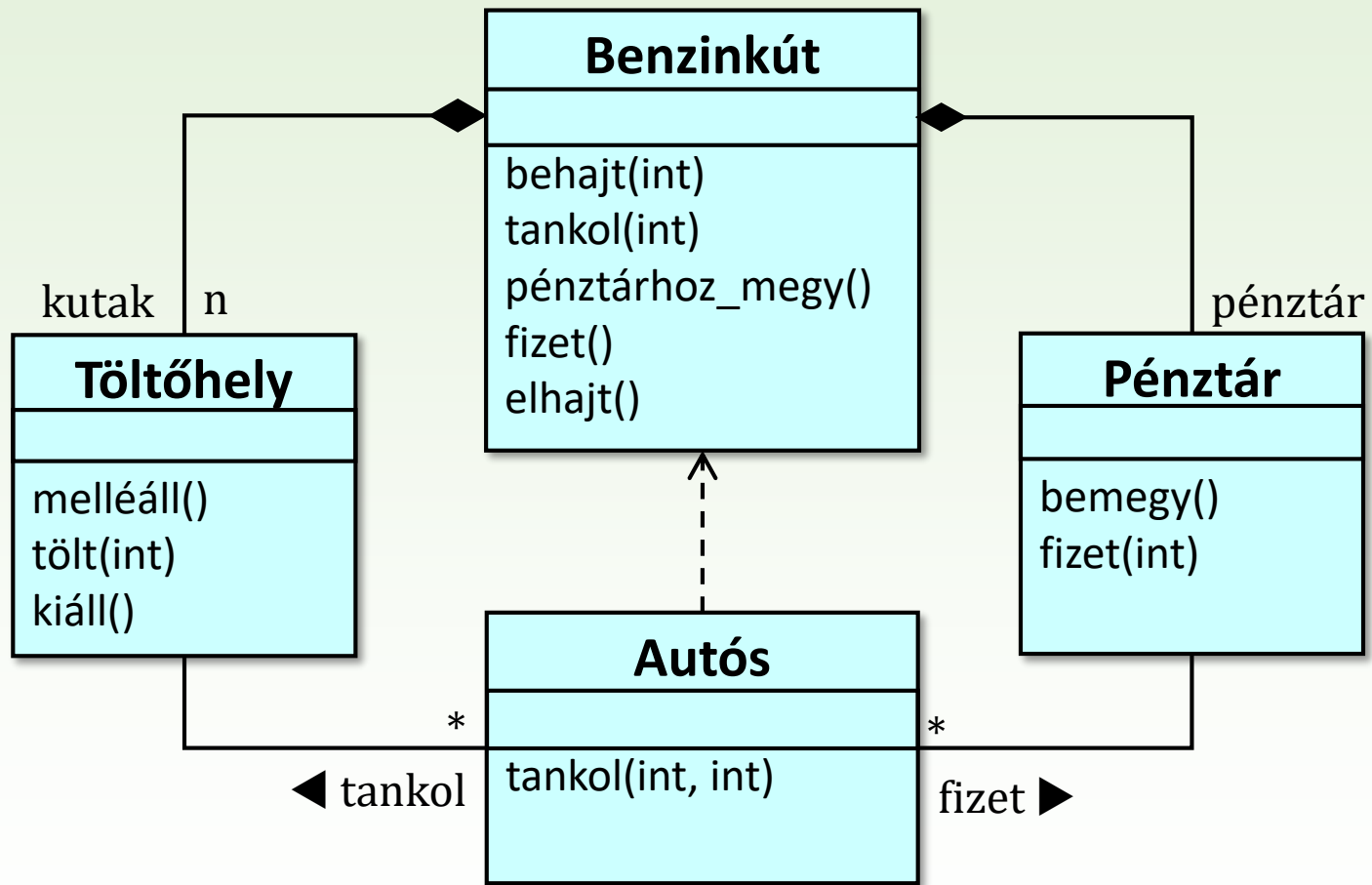
## ❑ Objektumok és tevékenységeik:

- autósok (tankolnak)
- benzinkút (ahová az autósok behajtanak, ahol tankolnak, pénztárhoz mennek, fizetnek, ahonnan elhajtanak)
- töltőhelyek (amely mellé beáll az autós , ahol benzint tölt, ahonnan fizetés után kiáll)
- pénztár több kasszával (ahová az autós bemegy, ahol fizet)

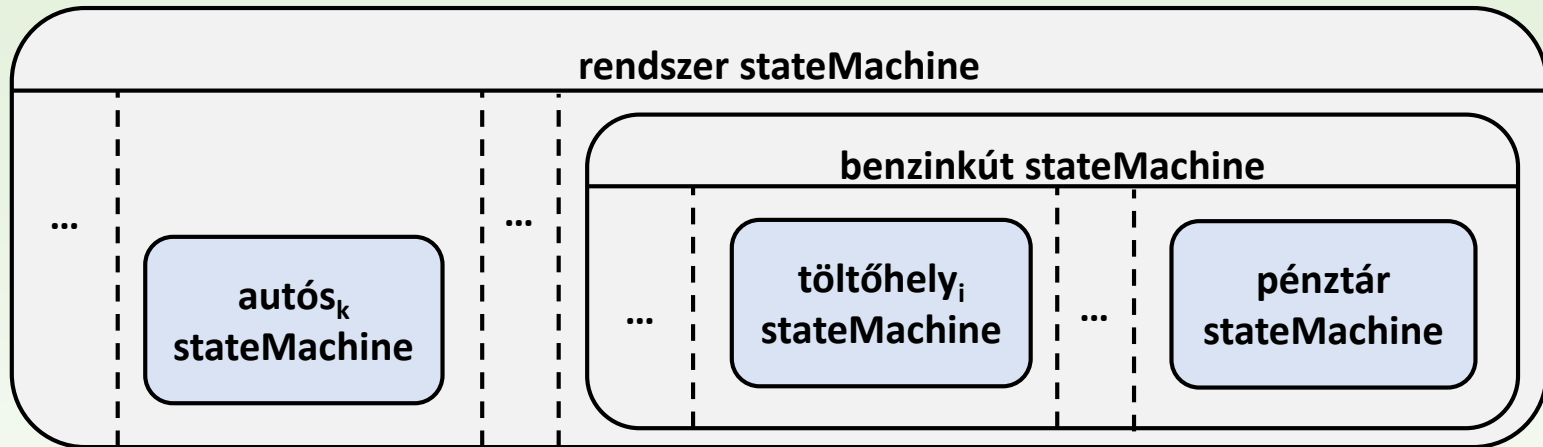
## ❑ Objektumok közötti kapcsolatok:

- a benzinkút részei a töltőhelyek és a pénztár
- egy autós ideiglenesen kapcsolatba kerül egy töltőhellyel és a pénztárral.

# Osztály diagram



# Rendszer állapotgépe

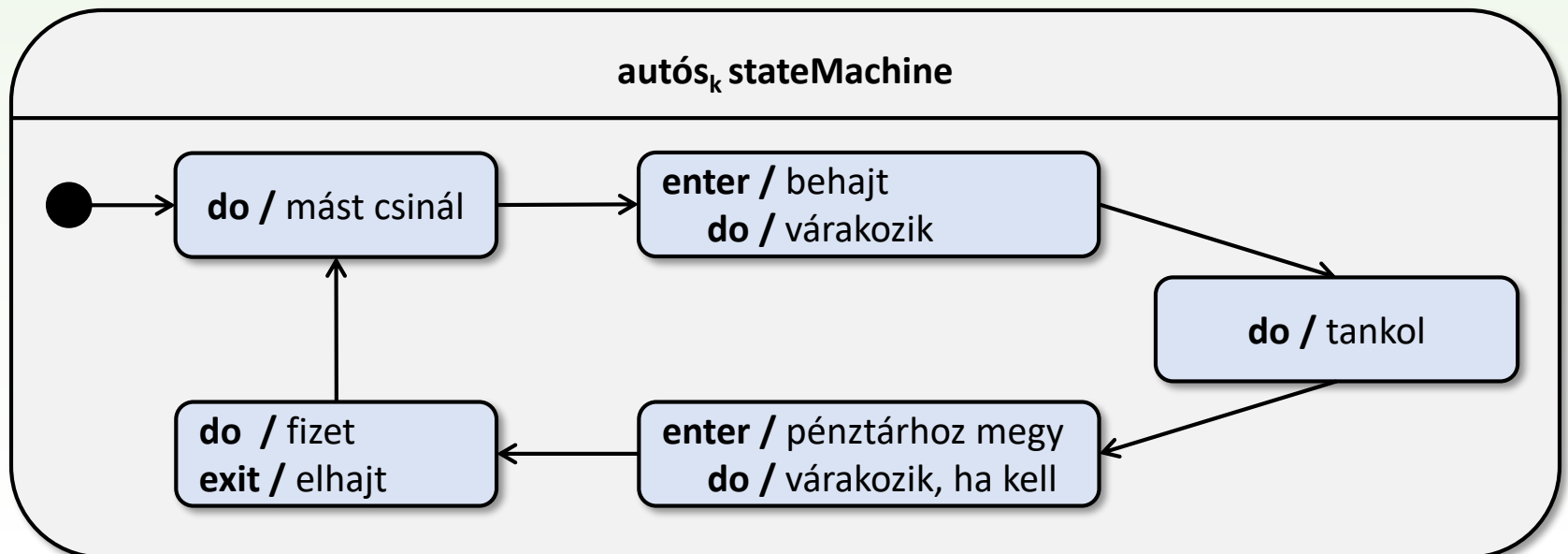


- ❑ A rendszer állapotát az autósok és a benzinkút állapota határozza meg. A benzinkút állapota a töltőhelyek és a pénztár állapotától függ.
- ❑ Az autósok ún. **aktív objektumok**: párhuzamosan végeznek tevékenységet, így állapotgépeik külön szálakon futnak majd.
- ❑ A benzinkút **passzív objektum**: állapotgépe más objektumok állapotgépével szinkron módon (metódusainak hívása által) működik. Nem igényel külön szálát.

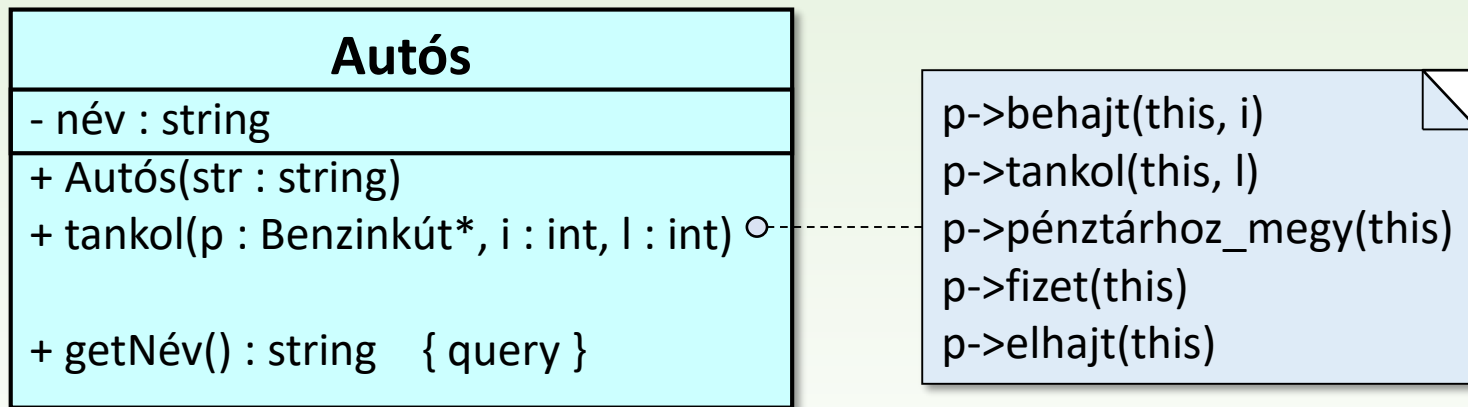


# Autósok állapotgépe

- ❑ Egy autós ötféle állapotban lehet, amelyek akár ciklikusan is változhatnak a benzinkút metódusainak hatására:
  - mást csinál, behajt a benzinkút egy kútjához és arra vár, benzint tankol, pénztárban sorba áll, fizet és elhajt
- ❑ Az átmeneteket az egyes állapotbeli tevékenységek befejeződése okozza.



# Autós osztály



# Autós osztály

```
class PetrolStation;
```

car.h

```
class Car {  
public:
```

megvárja, míg a külön  
indított szál befejeződik

```
    Car(const std::string &str) : _name(str) {}  
    ~Car() { _fuel.join(); }  
    std::string getName() const { return _name; }  
    void refuel(PetrolStation* p, unsigned int i, int l) {  
        _fuel = std::thread(process, this, p, i, l);  
    }
```

```
private:
```

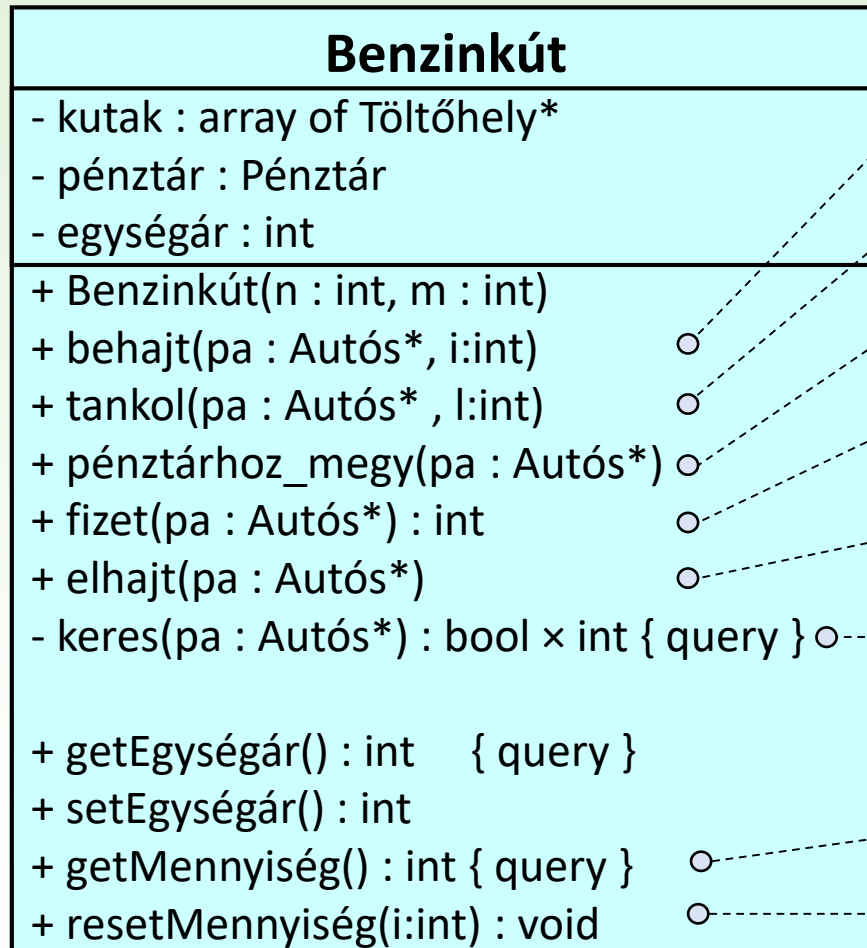
külön szálon indul el  
#include <thread>

```
    std::string _name;  
    std::thread _fuel;  
    void process(PetrolStation* p, unsigned int i, int l);  
};
```

```
void Car::process(PetrolStation* p, unsigned int i, int l)  
{  
    if( !p->driveIn(this, i) ) return;  
    if( !p->tank(this, l) ) return;  
    p->goToCash(this);  
    int price = p->pay(this);  
    p->driveOff(this);  
}
```

car.cpp

# Benzinkút osztálya



kutak[i]->melléáll(pa)

l,i := keres(pa)  
kutak[i]->tölt(pa, l)

pénztár.bemegy(pa)

l,i := keres(pa)  
return pénztár.fizet(i, pa)

l,i := keres(pa)  
kutak[i]->kiáll(pa)

return linker(pa in kutak)

kutak[i]->getMennyiség()

kutak[i]->resetMennyiség()

# Benzinkút osztálya

```
class PetrolStation {  
public:  
    PetrolStation(int n, int m) : _cash(this, m) {  
        _pumps.resize(n);  
        for(int i = 0; i<n; ++i) _pumps[i] = new Pump();  
    }  
    ~PetrolStation() { for( Pump *p : _pumps ) delete p; }  
    bool driveIn(Car* pa, unsigned int i);  
    bool tank(Car* pa, int l);  
    void goToCash(Car* pa);  
    int pay(Car* pa);  
    bool driveOff(Car* pa);  
  
    void resetQuantity(unsigned int i) { _pumps[i]->resetQuantity(); }  
    int getQuantity(unsigned int i) const { return _pumps[i]->getQuantity(); }  
    int getUnit() const { return _unit; }  
private:  
    std::vector<Pump*> _pumps;  
    Cash _cash;  
    int _unit = 400;  
  
    bool search(Car* pa, unsigned int &ind) const;  
};
```

petrol.h

# Benzinkút metódusai

```
bool PetrolStation::driveIn(Car* pa, unsigned int i){
    if( i>=_pumps.size() ) return false;
    _pumps[i]->standNextTo(pa);
    return true;
}

bool PetrolStation::tank(Car* pa, int l){
    unsigned int i; if( !search(pa, i) ) return false;
    _pumps[i]->fill(pa, l);
    return true;
}

void PetrolStation::goToCash(Car* pa){
    _cash.goIn(pa);
}
```

```
int PetrolStation::pay(Car* pa){
    unsigned int i; if( !search(pa, i) ) return 0;
    return _cash.pay(i, pa);
}

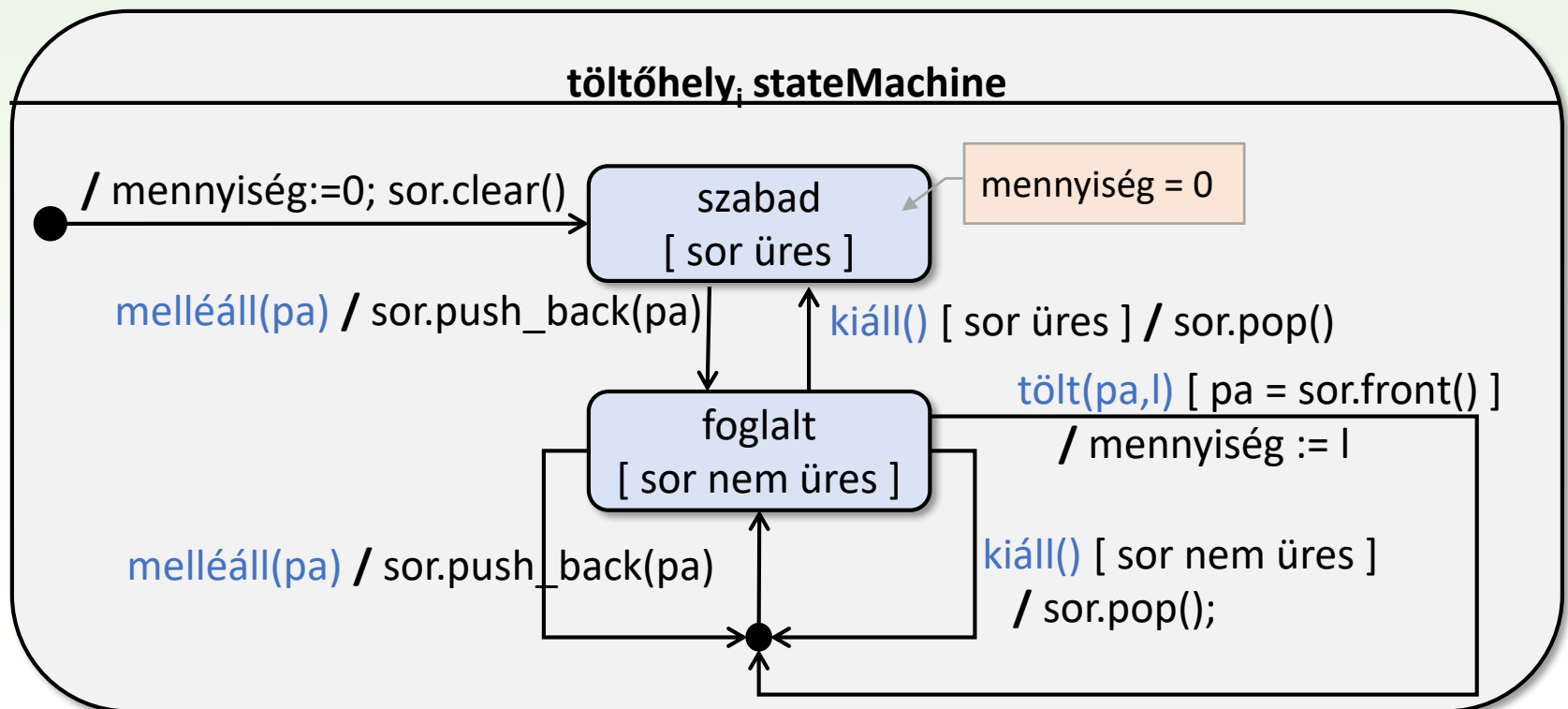
bool PetrolStation::driveOff(Car* pa){
    unsigned int i; if( !search(pa, i) ) return false;
    _pumps[i]->leave();
    return true;
}

bool PetrolStation::search(Car* pa, unsigned int &ind) const {
    bool l = false;
    for(unsigned int i = 0; !l && i<_pumps.size(); ++i) {
        l = _pumps[i]->getCurrent() == pa; ind = i;
    }
    return l;
}
```

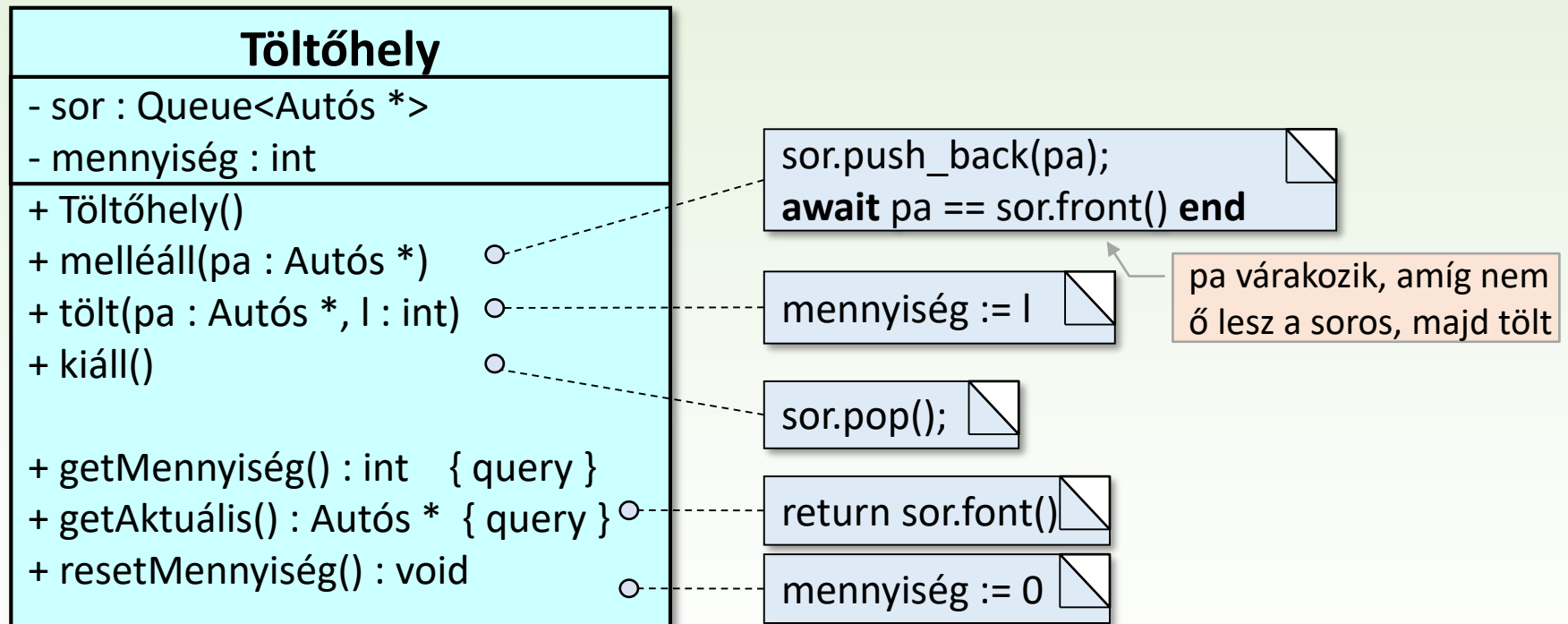
petrol.cpp

# Töltőhely állapotgépe

- ❑ Egy töltőhely lehet **szabad** vagy **foglalt**: az utóbbi esetben az autósok egy **várakozási sorban** várakoznak, és az első autós töltheti a benzint.
- ❑ Foglalt állapotban megadott **menyiségű** benzint tölt autójába az aktuális autós, amely a töltőhely kijelzőjén a fizetésig látszik.



# Töltőhely osztálya





# Töltőhely osztálya

```
class Car;

class Pump {
public:
    Pump() : _quantity(0) { }

    void standNextTo(Car *pa);
    void fill(Car *pa, int l);
    void leave();

    Car* getCurrent() const { return _queue.front(); }
    int getQuantity() const { return _quantity; }
    void resetQuantity() { _quantity = 0; }
private:
    int _quantity;
    std::queue<Car*> _queue;
    std::mutex _mu;
    std::condition_variable _cond;
};
```

#include <mutex>  
#include <condition>

pump.h

# Töltőhely metódusai

```
void Pump::standNextTo(Car* pa)
{
    std::unique_lock<std::mutex> lock(_mu);
    _queue.push(pa);
    // várakozik:
    while( pa != _queue.front() ) {
        _cond.wait(lock);
    }
}
```

egyszerre csak egy autós  
mozoghat ennél a töltőhelynél

várakozik a szál

```
void Pump::fill(Car* pa, int l)
{
    std::unique_lock<std::mutex> lock(_mu);
    _quantity = l;
}
```

egyszerre csak egy autós  
mozoghat ennél a töltőhelynél

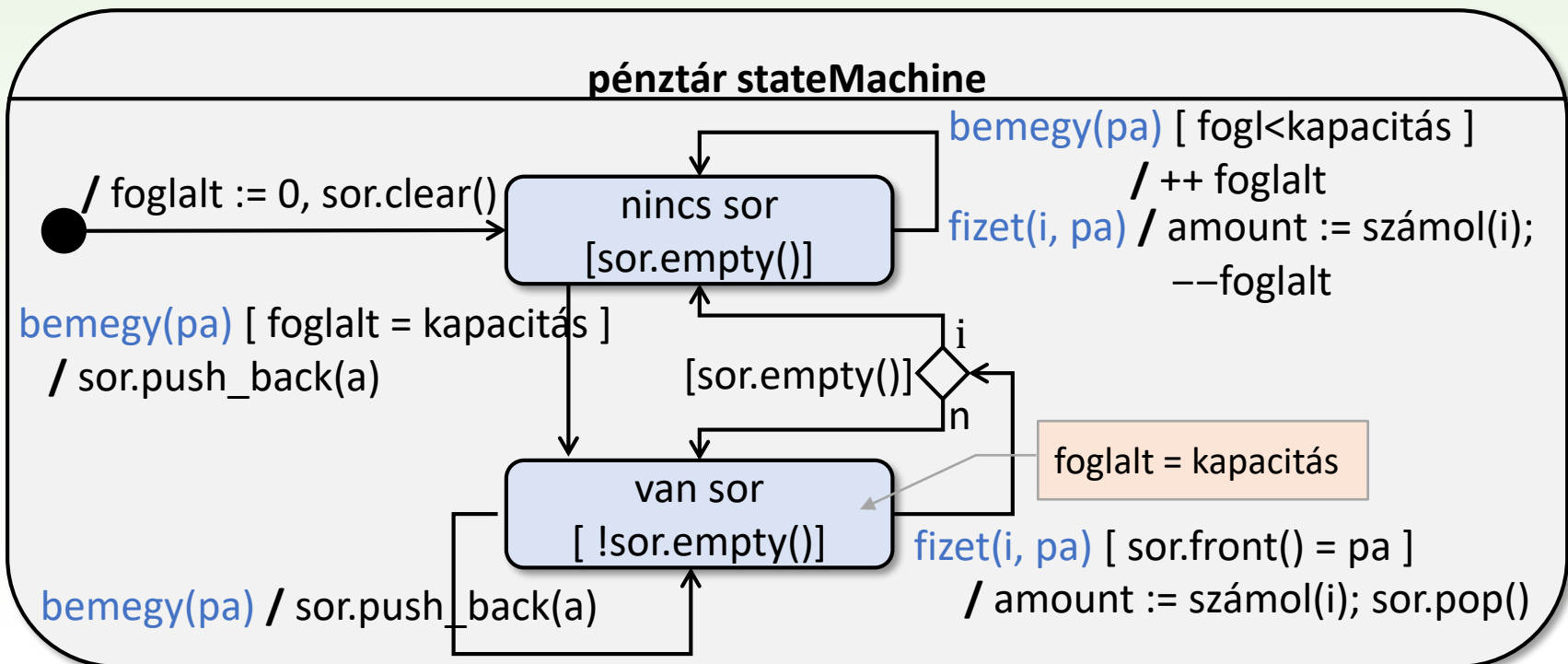
```
void Pump::leave()
{
    _queue.pop();
    _cond.notify_all();
}
```

elindítja az összes cond-nál  
várakozó szálát

pump.cpp

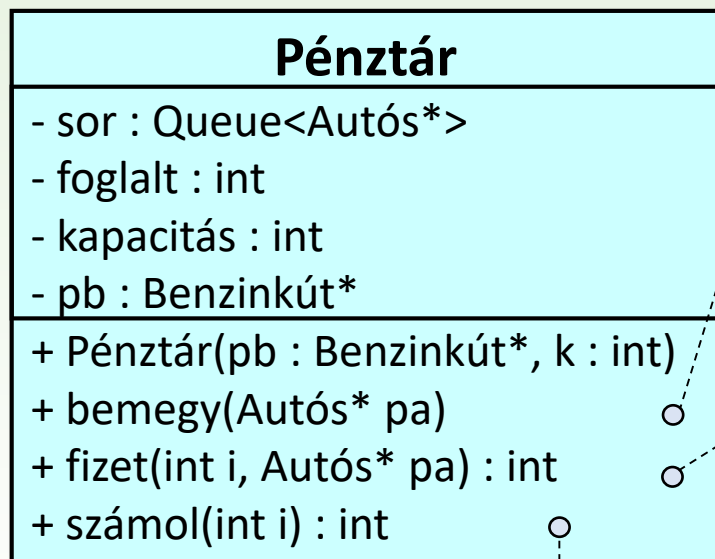
# Pénztár állapotgépe

- ❑ A pénztárnál vagy **nincs sor** (azaz a **várakozási sor** üres), ekkor a **foglalt** kasszák száma kisebb-egyenlő a kasszák számánál (**kapacitás**), vagy **van sor**, és ekkor a **foglalt** kasszák száma egyenlő a kasszák számával.
- ❑ Fizetéskor szükség van az autós által betöltött benzin mennyiségre és az egységárára. Mindkettőt a benzinkúttól kell lekérni.



# Pénztár

- ❑ A pénztárat egyszerre több autós is „használhatja”. Ezek közül legfeljebb annyi fizethet éppen, ahány kassa van, a többi várakozik.



```
if foglalt < kapacitás then ++foglalt  
else sor.push_back(pa) end
```

```
if sor.empty() then
```

```
    összeg := számol(i)  
    --foglalt
```

rendezi a számlát  
és kimegy

```
else
```

```
    await sor.front() = pa end
```

pa várakozik, amíg  
nem ő lesz a soros

```
    összeg := számol(i)  
    sor.pop()
```

rendezi a számlát  
és kimegy

```
end
```

```
return összeg
```

```
összeg := pb->getMennyiség(i) * pb->getEgységár()  
pb->resetMennyiség(i)  
return összeg
```

# Pénztár osztálya

```
class PetrolStation;
class Car;

class Cash {
public:
    Cash(PetrolStation *p, int m): _p(p), _capacity(m) {}
    void goIn(Car *pa);
    int pay(unsigned int i, Car *pa);
private:
    PetrolStation *_p;
    int _engaged;
    int _capacity;
    std::queue<Car*> _cashQueue;

    std::mutex _mu;
    std::condition_variable _cond;

    int compute(int i) const;
};
```

cash.h

# Pénztár metódusai

```
void Cash::goIn(Car* pa)
{
    std::unique_lock<std::mutex> lock(_mu);
    if( _engaged<_capacity ) ++_engaged;
    else {
        cashQueue.push(pa);
        while( _cashQueue.front() != pa ) {
            _cond.wait(lock);
        }
    }
}
```

várakozik a szál

```
int Cash::pay(unsigned int i, Car* pa)
{
    int amount = compute(i);
    if(_cashQueue.empty) --_engaged;
    else _cashQueue.pop();
    _cond.notify_all();
    return amount;
}
```

elindítja az összes cond-  
nál várakozó szálakat

```
int Cash::compute(int i) const
{
    int amount = _p->getQuantity(i) * _p->getUnit();
    _p->resetQuantity(i);
    return amount;
}
```

cash.cpp