

# Állapotgép

1.rész

## UML állapotgép diagramja

Gregorics Tibor

[gt@inf.elte.hu](mailto:gt@inf.elte.hu)

<http://people.inf.elte.hu/gt/oep>

# Objektum élelciklusa

- ❑ Egy objektum az **élelciklusa** (a működése) során
  - **létrejön**: példányosodik (konstruktor)
  - **változik**: más objektumok hívják metódusait, vagy szignált küldenek neki, és ennek következtében változhatnak az adatai
  - **megszűnik**: megsemmisül (destruktor)
- ❑ Egy objektumnak meg lehet különböztetni az állapotait (*state*).
  - **fizikai állapot**: az objektum adatai által felvett értékek együttese
  - **logikai állapot**: valamilyen szempont szerint közös tulajdonságú fizikai állapotoknak az összessége (halmaza).
- ❑ Egy objektum állapota valamilyen **esemény** hatására változhat meg.

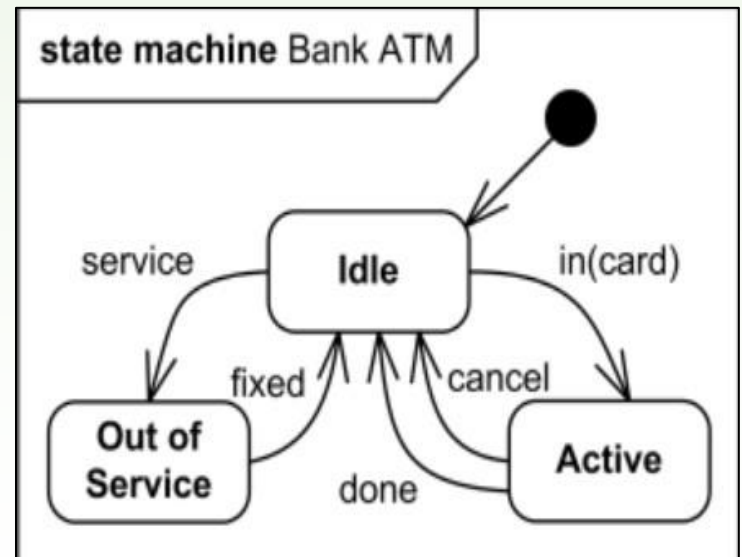
# Esemény

- ❑ Az esemény (*event*) lehet egy
  - **üzenet** (*trigger*), amely paraméterekkel is rendelkezhet. Ez
    - vagy az objektum egy metódusának hívása
    - vagy az objektumnak küldött szignál észlelése
  - **tevékenység befejeződése**
  - **őrfeltétel** (*guard*) **teljesülése**. Az őrfeltétel lehet egy
    - logikai állítás (*when*), amely többek között az objektum adattagjainak értékétől is függhet,
    - időhöz kötött várakozás (*after*)

# Állapot-átmenet gráf

- ❑ Egy **objektum életciklusát** – a logikai állapotról logikai állapotra változó működését – az ún. állapot-átmenet gráffal ábrázolhatjuk, és ezt a gráfot nevezzük az objektum **állapotgépének**.
- ❑ Ez modellezhető egy olyan irányított gráffal, ahol a csúcsok a **logikai állapotokat**, az irányított élek az **állapot-átmeneteket** jelölik, és mind az állapotokhoz, mind az átmenetekhez **tevékenységek** is tartozhatnak.

- ❑ Az állapotgéppel nyomon kísérhetjük, hogy az objektumnak éppen melyik állapota **aktív** (ebből legfeljebb egy van), és hogy melyik eseményre hogyan fog reagálni.
- ❑ Az új aktív állapot **egyértelműen választódik ki** a régeből kivezető élek által mutatott állapotok közül.



# Állapotok

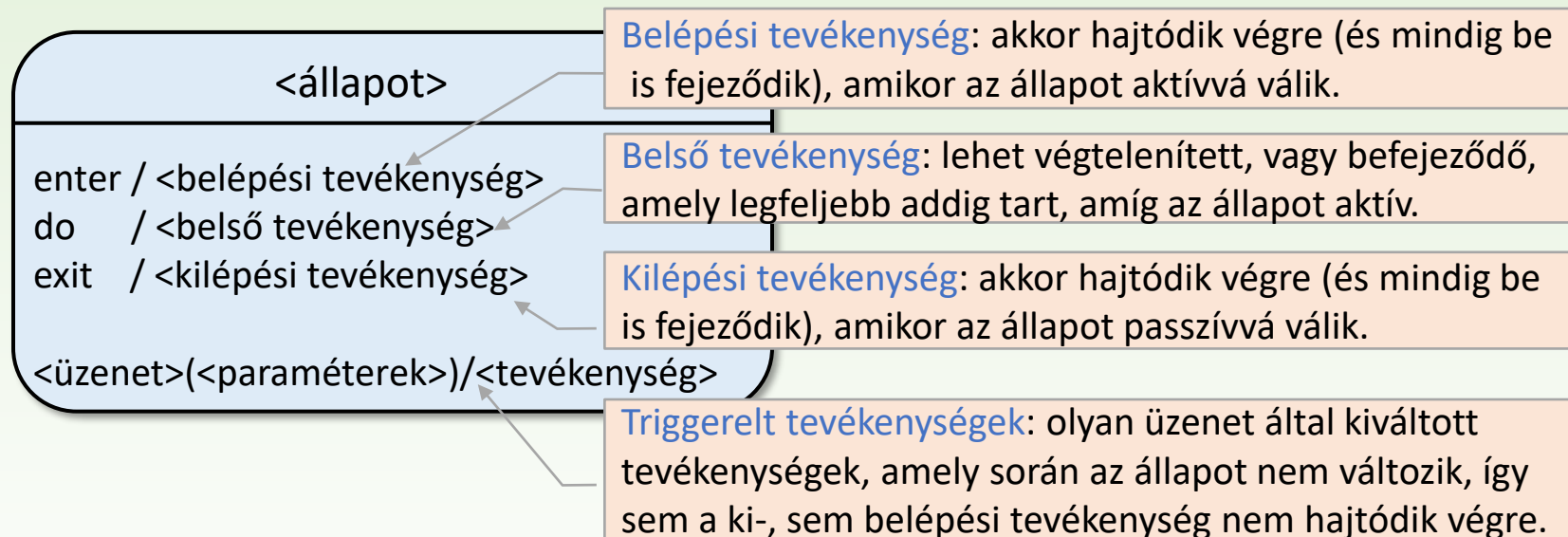
<állapot>

<állapot>  
[ feltétel ]

- ❑ Az állapot (csúcs) jele egy lekerekített sarkú téglalap, amelynek adhatunk nevet, de lehet anonim is.
- ❑ Az állapot neve mellett szögletes zárójelek közé írt logikai feltétellel leírhatjuk az állapot által képviselt fizikai állapotokat is. (Ez a feltétel az objektum adattagjaira megfogalmazott állítás.) Egy állapotgép állapotainak feltételei teljesen diszjunkt rendszert kell, hogy alkossanak.
- ❑ Az állapotot jelző téglalapban felsorolhatók az állapothoz rendelt különféle tevékenységek.

# Állapot tevékenységei

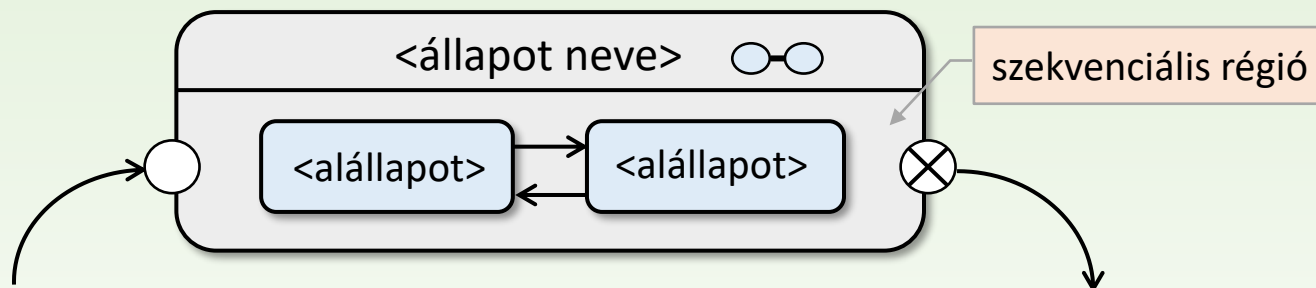
- Egy állapothoz négy féle tevékenység tartozhat, amelyek olyankor hajtódnak végre, amikor az állapot aktív.



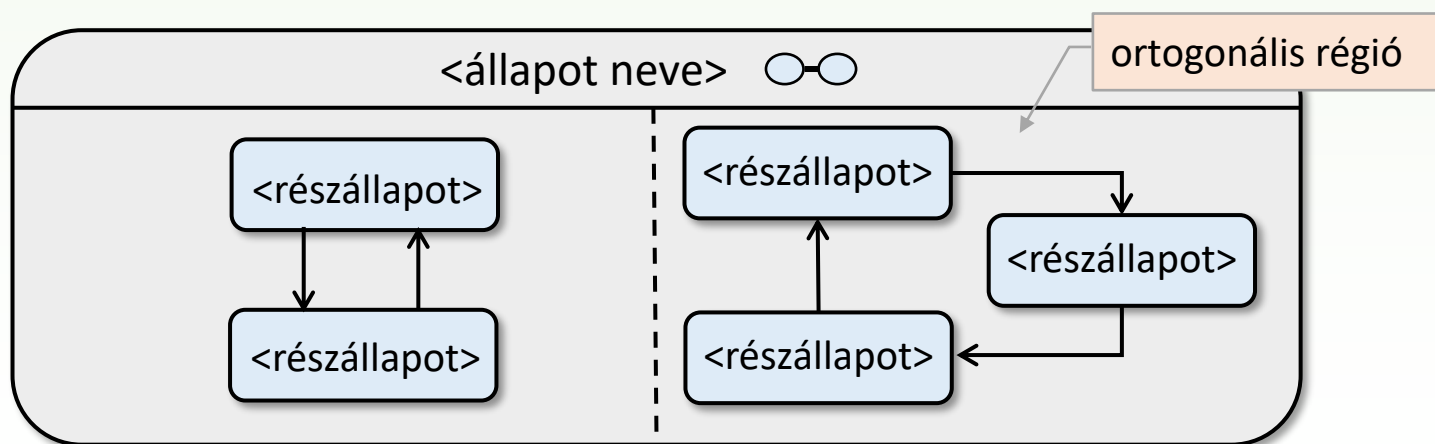
- Egy állapotból egy másikba történő átmenet (ami lehet reflexív is) során az aktív állapot kilépési tevékenysége fut le először, ezt követi az átmenethez tartozó tevékenység (ha van ilyen), végül az új állapot aktívvá válásával együtt annak belépési tevékenységére kerül sor, amit a belső tevékenységének végrehajtása követ.

# Hierarchikus állapotok



Szekvenciális állapotgép hierarchikus állapota:

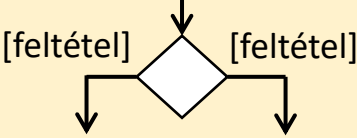
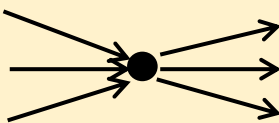







Párhuzamos állapotgép hierarchikus állapota:




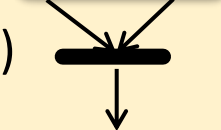


# Pszeudo állapotok

- kezdő állapot 
- végállapot 

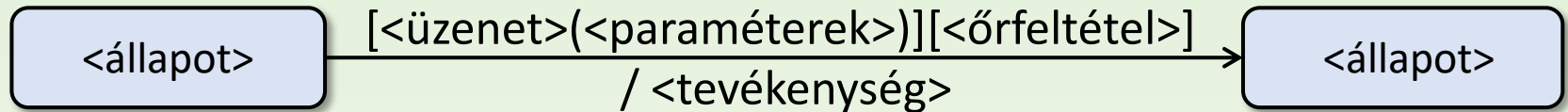
- elágazás (choice) 
- csomópont (junction) 

- belépés (entry) 
- kilépés (exit) 
- megszüntetés 
- shallow history 
- deep history 

- szétágazás (fork)   

- összefutás (join)   




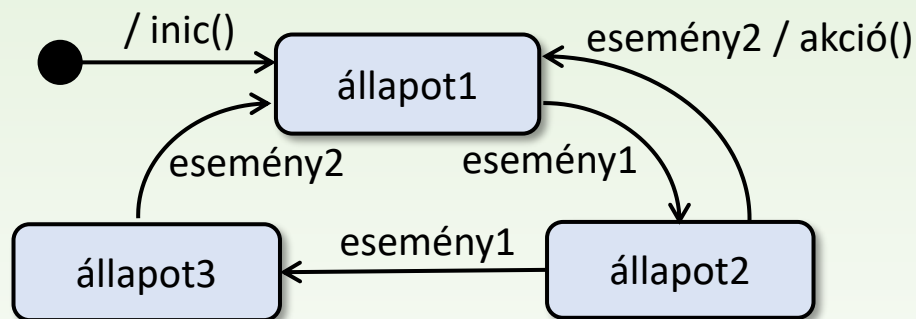
# Állapot-átmenet



- ❑ Egy állapotból egy másikba történő átmenet mindig valamilyen esemény hatására következik be.
- ❑ Ha az esemény egy üzenet (metódus-hívás vagy szignál küldés), akkor ennek nevét ráírjuk az átmenetet jelző nyílra.
- ❑ Egy üzenetnek lehetnek paraméterei, és kapcsolódhat hozzá őrfeltétel is. Amennyiben az őrfeltétel egy logikai állítás (*when*), akkor ez az üzenet paramétereitől és az objektum adattagjaitól függ.
- ❑ Az átmenethez rendelt tevékenység az objektum adattagjaival és a kiváltó üzenet (ha van ilyen) paramétereivel operáló program.

# Állapot-átmenet tábla

□ Az állapotgép diagram működése leírható egy táblázattal is.



állapot esemény	állapot1 / start: inic()	állapot2	állapot3
esemény1	állapot2	állapot3	
esemény2		állapot1 / akció()	állapot1

# Állapotgép megvalósítása

<div> <div>állapot</div> <div>esemény</div> </div>	állapot1 / start: inic()	állapot2	állapot3
esemény1	állapot2	állapot3	
esemény2		állapot1 / akció()	állapot1

```

inic()
állapot := állapot1
while állapot ≠ stop loop
  switch esemény
    case esemény1:
      switch állapot
        case állapot1 : állapot := állapot2
        case állapot2 : állapot := állapot3
        case állapot3 :
        endswitch
    case esemény2:
      switch állapot
        case állapot1 :
        case állapot2 : akció()
          állapot := állapot1
        case állapot3 : állapot := állapot1
      endswitch
    endswitch
  endloop

```

az elágazások kiküszöbölésére  
állapot-, illetve látogató  
tervezési mintát alkalmazzunk

```

inic()
állapot := állapot1
while állapot ≠ stop loop
  switch állapot
    case állapot1:
      switch esemény
        case esemény1 : állapot := állapot2
      endswitch
    case állapot2:
      switch esemény
        case esemény1 : állapot := állapot3
        case esemény2 : akció()
          állapot := állapot1
      endswitch
    case állapot3:
      switch esemény
        case esemény2 : állapot := állapot1
      endswitch
    endswitch
  endloop

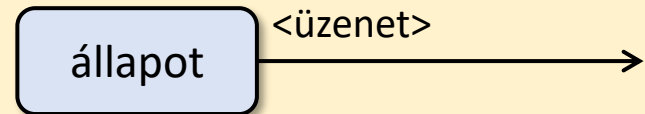
```

# Üzenet-vezérelt állapot-átmenetek

Amikor az átmenetet **üzenet váltja ki** (ami vagy egy metódus hívás vagy egy szignál küldés), akkor az átmenet

- **őrfeltétel hiányában azonnal** megvalósul (ha van belső tevékenység, az megszakad).

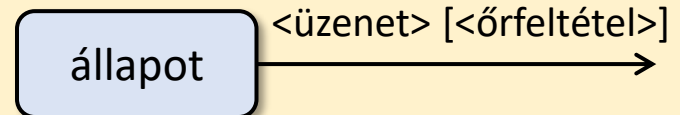
Egy állapotból azonos üzenettel nem vezethet ki egynél több őrfeltétel nélküli él.



determinisztikus

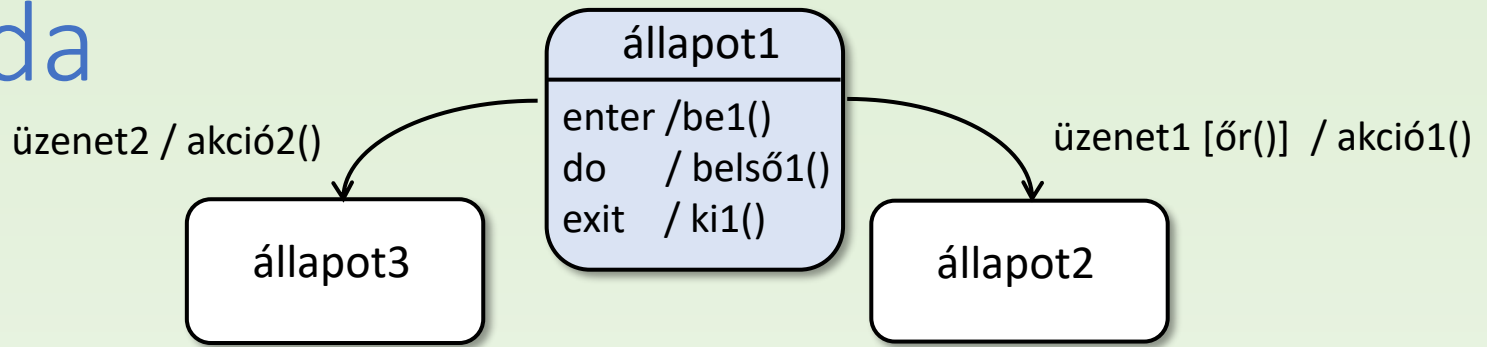
- **őrfeltétel esetén** csak akkor valósul meg, **ha az őrfeltétel éppen teljesül** az üzenet beérkezésekor.

Egy állapotból azonos üzenettel akkor vezethet ki egynél több él, ha azok őrfeltételei diszjunktak.



determinisztikus

# Példa



```
while állapot ≠ stop loop
```

```
    switch állapot
```

```
        case állapot1:
```

```
            be1(); start(belső1())
```

```
            switch esemény
```

```
                case üzenet1 :
```

```
                    if őr() then
```

```
                        stop(belső1()); ki1();
```

```
                        akció1()
```

```
                        állapot := állapot2
```

```
                    endif
```

```
                case üzenet2 :
```

```
                    stop(belső1()); ki1();
```

```
                    akció2()
```

```
                    állapot := állapot3
```

```
            endswitch
```

```
        case állapot2:
```

```
            ...
```

```
        endswitch
```

```
    endloop
```

elindul a belső tevékenység

megszakad a belső tevékenység

# Üzenet nélküli állapot-átmenetek

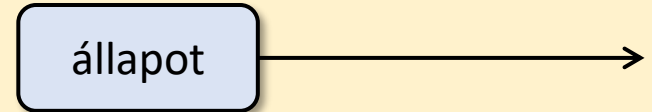
Amikor az átmenetet **nem üzenet váltja ki**, hanem az állapot belső tevékenységének befejeződése (ennek hiányában az állapot aktívvá válása), akkor

- az átmenet **őrfeltétel hiányában azonnal** megvalósul a belső tevékenység lefutása után.

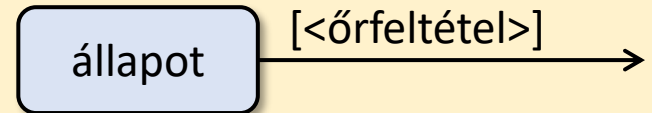
Ilyen átmenete egy állapotnak csak egy lehet.

- **őrfeltétel esetén várakozik** az átmenet arra, hogy az őrfeltétel igaz legyen. De a várakozás megszakad, és az átmenet megghiúsul, ha közben egy másik állapot-átmenet következik be.

Egy állapotból kivezető üzenet nélküli átmenetek őrfeltételei diszjunktak.

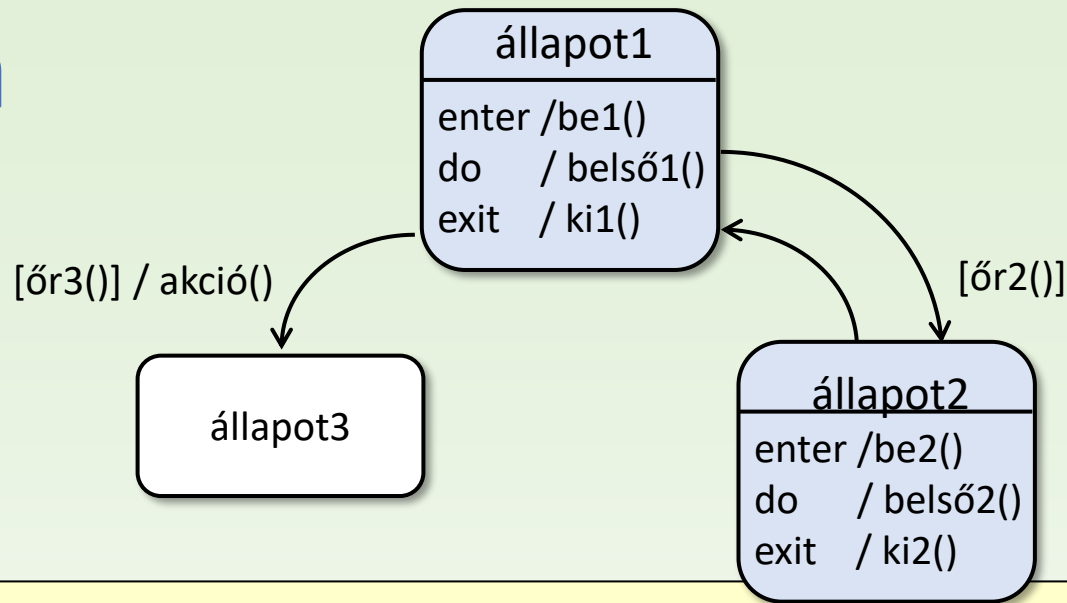


determinisztikus



determinisztikus

# Példa



```
while állapot ≠ stop loop
  switch állapot
    case állapot1 :
      be1(); start(belső1()); wait( vége(belső1()) )
      wait ( őr2() or őr3() )
      ki1()
      if      őr2() then      állapot := állapot2;
      elseif őr3() then akció(); állapot := állapot3;
      endif
    case állapot2 :
      be2(); start(belső2()); wait( vége(belső2()) ); ki2()
      állapot := állapot1
    case állapot3 :
      ...
  endswitch
endloop
```

# Nyomtató állapotgépe

## ❑ A nyomtató

- **aktív** állapotában a *current* adattagjában tárolt fájl *act*-edik blokkját nyomtatja, és egy *queue* tárolóban nyomtatásra váró fájlokat tárolhat.
- **passzív** állapotában nem nyomtat, és nyomtatásra váró fájl sincs.

## ❑ A *Send()* segítségével küldhetünk a nyomtatónak egy nyomtatandó fájlt.

- Ha a nyomtató aktív, akkor ez a fájl a *queue* tárolóba kerül.
- Ha a nyomtató passzív, akkor ez a fájl blokkokra bontva a *current*-be kerül, az *act* ennek első blokkjára mutat, a nyomtató állapota pedig aktív lesz.

## ❑ Szükség lesz még egy *endofprint* szignálra is, amelyet akkor küld a nyomtató saját magának, amikor a *current*-ben tárolt fájl utolsó blokkját kinyomtatta.

Ennek hatására

- üres *queue* esetén a nyomtató passzív lesz,
- nem üres *queue* esetén a soron következő fájl blokkokra bontva kerül a *current*-be, és az *act* az első blokkra mutat



# Nyomtató állapotgépe

**switch** state **do**

**case** passive: SplitToBlocks(file); state:=active

**case** active: queue.Enqueue(file)

**endswitch**

current(act) nyomtatása

++act

**if** act>|current| **then**

**send** endofprint **to** printer

**endif**

current, act := split(file), 1

**Printer**

**Printer**

- queue : Queue<File>

- current : Block[]

- act : int

- state : State = {active, passive}

+ Printer() ○---- state := passive

+ Send(file:File)

- SplitToBlocks(file:File)

- PrintBlock()

<<signals>>

endofprint



**passive**

Send(file) / SplitToBlocks(file)

endofprint [ queue.Empty() ]

ha a belső tevékenység végeztével  
nincs feldolgozandó üzenet, akkor  
ez az „üres” átmenet valósul meg

**active**

/ do <PrintBlock()>

Send(file)

/ queue.Enqueue(file)

endofprint [ **not** queue.Empty() ]  
/ SplitToBlocks(queue.Dequeue())

az üzenetek nem szakíthatják  
meg a belső tevékenységet:  
< > az atomi tevékenység jele

# Állapotgép

## 2.rész

### Garázskapu-vezérlés modellezése

Gregorics Tibor

[gt@inf.elte.hu](mailto:gt@inf.elte.hu)

<http://people.inf.elte.hu/gt/oep>

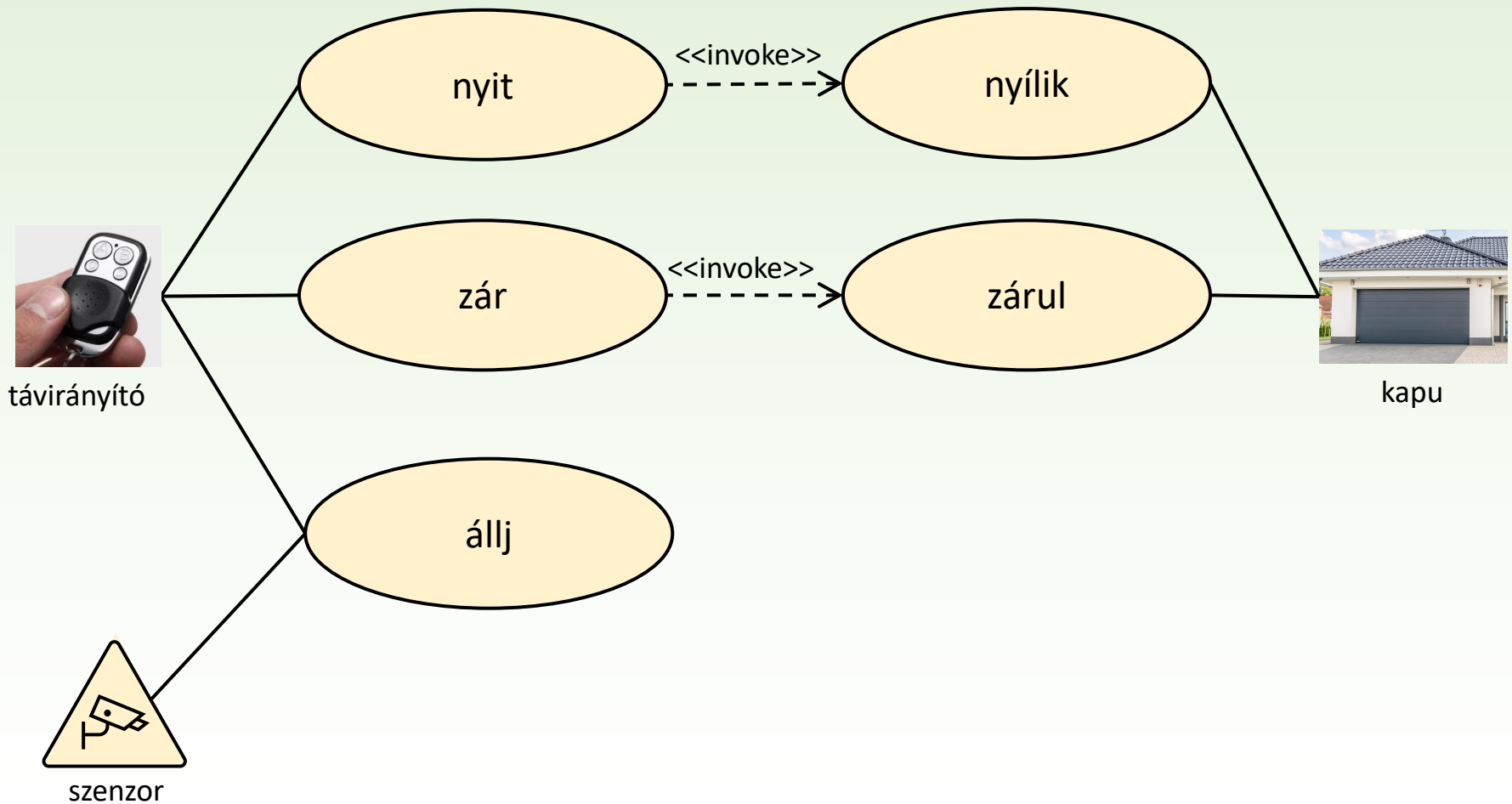
# Feladat

Tervezzük meg, és implementáljuk egy függőlegesen nyíló garázskapu működését!

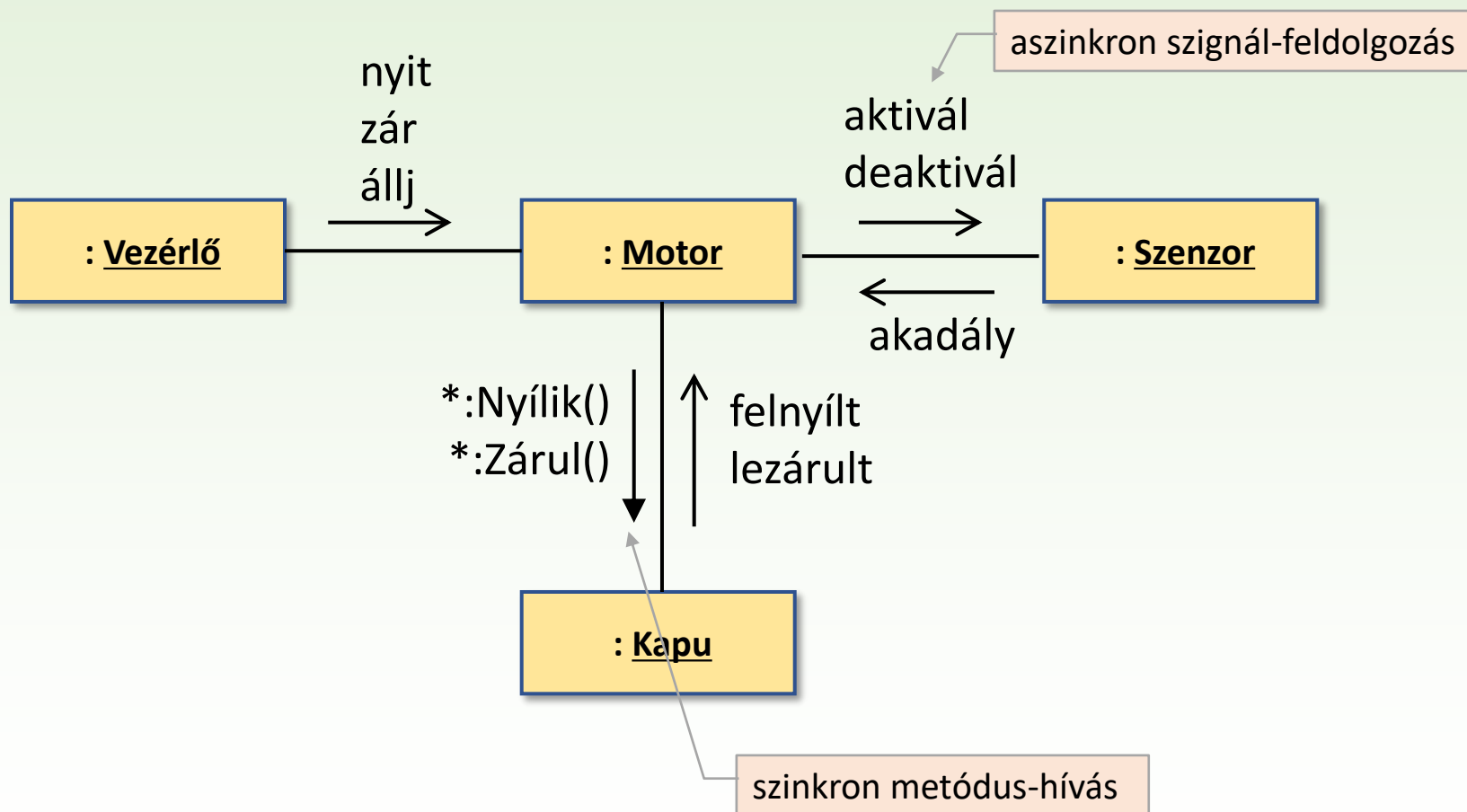
- A garázskaput egy távirányító vezérli, amely „nyílj”, „záródj”, „állj” parancsokat küldhet a kapu mozgató motornak.
- A motor az aktuális (a legutoljára) küldött parancs szerint mozgatja a hozzá kapcsolódó kapuszárnyat.
- A kapu mozgása leáll magától, ha a kapu felnyílt, vagy lezárult, vagy ha valamilyen akadályt jelez az a szenzor, amit a kapun helyeztek el.



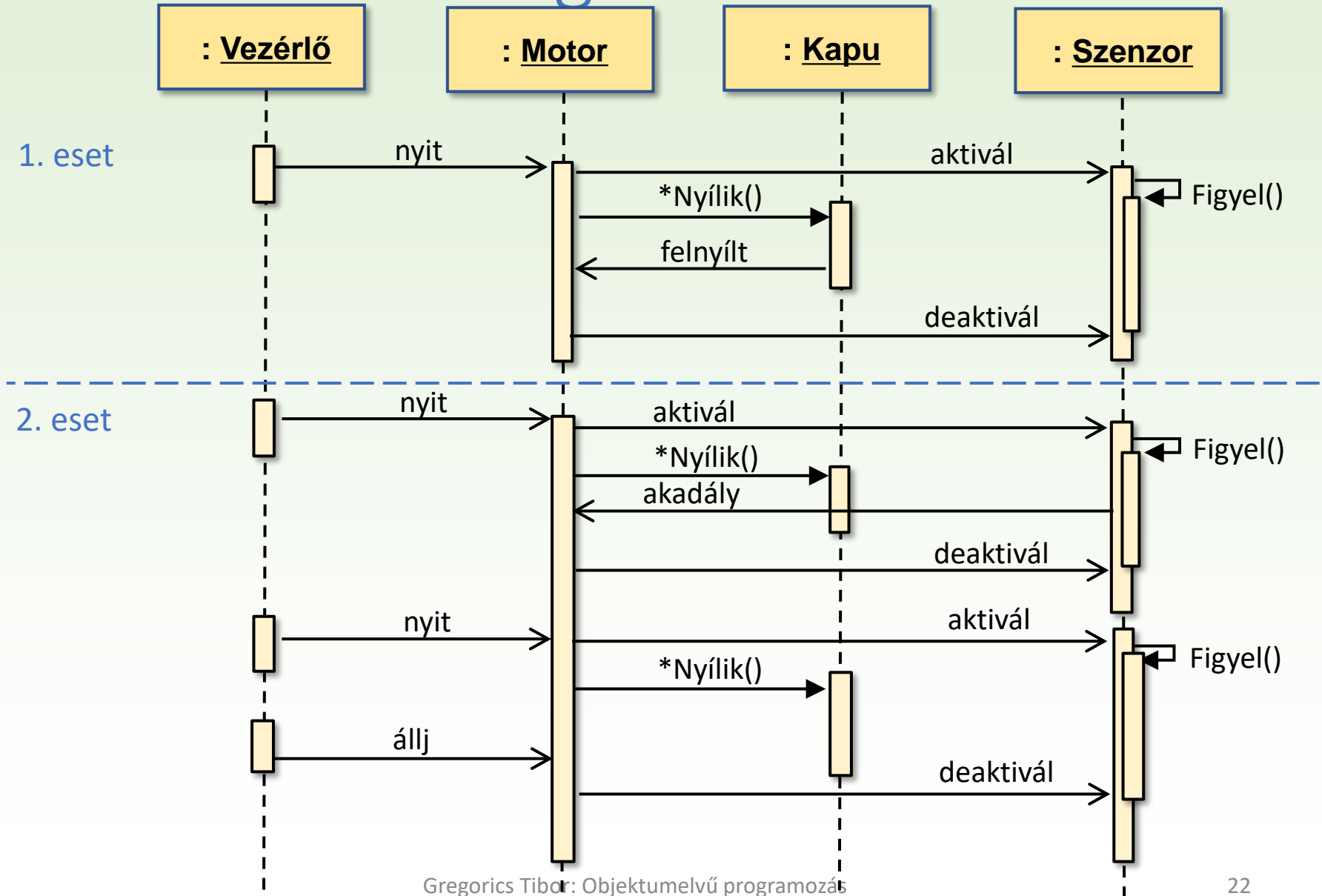
# Használati eset diagram



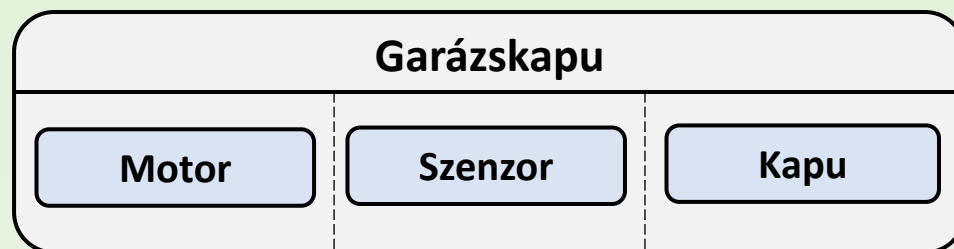
# Kommunikációs diagram



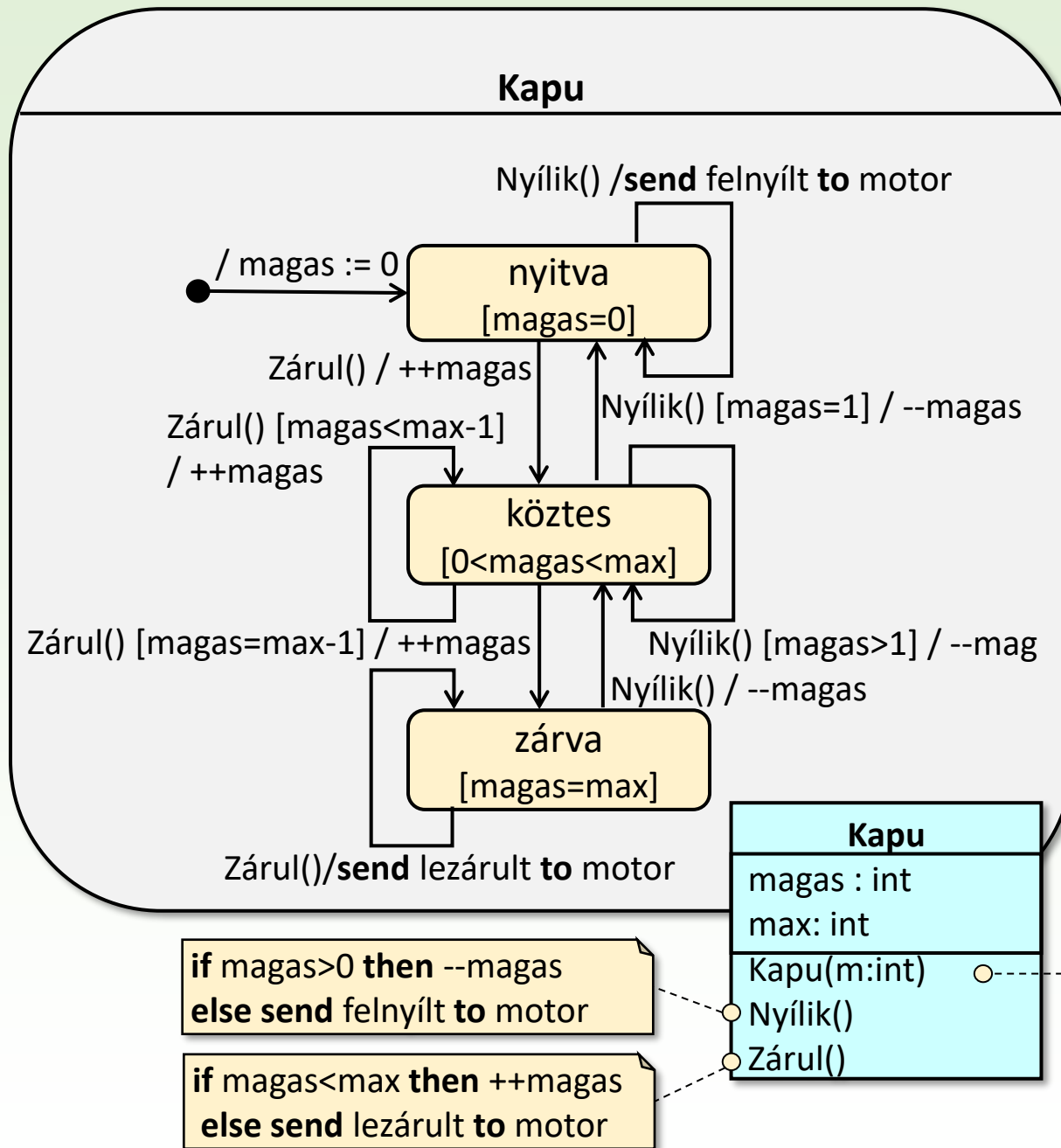
# Szekvencia diagram



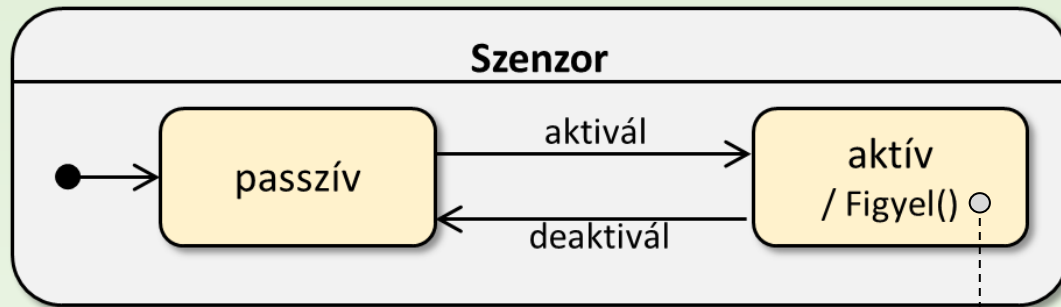
# Állapotgépek



- ❑ A garázskapu rendszer objektumai közül a motor, a szenzor, és a kapu esetében lehet különböző logikai állapotokat megkülönböztetni. Ezek az állapotok az adott objektumnak küldött üzenetek hatására változnak: ez a változás állapotgépekkel írható le.
  - A motor és a szenzor a neki küldött **szignálokat aszinkron módon** (azaz a küldő objektum működését nem várakoztatva) dolgozzák fel azok beérkezési sorrendjében, miközben maguk is küldenek üzeneteket más objektumoknak.
  - A kapu állapotát a motor által **szinkron módon hívható** `Nyílik()`, `Zárul()` **metódusok** módosíthatják.
  - A vezérlő egység a felhasználói szándékot továbbítja szignálok formájában a motor felé: nincsen saját állapotgépe.

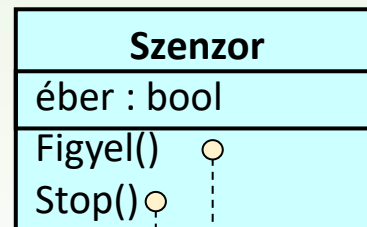






```

loop
  if "akadály" then send akadály to motor
endloop
  
```

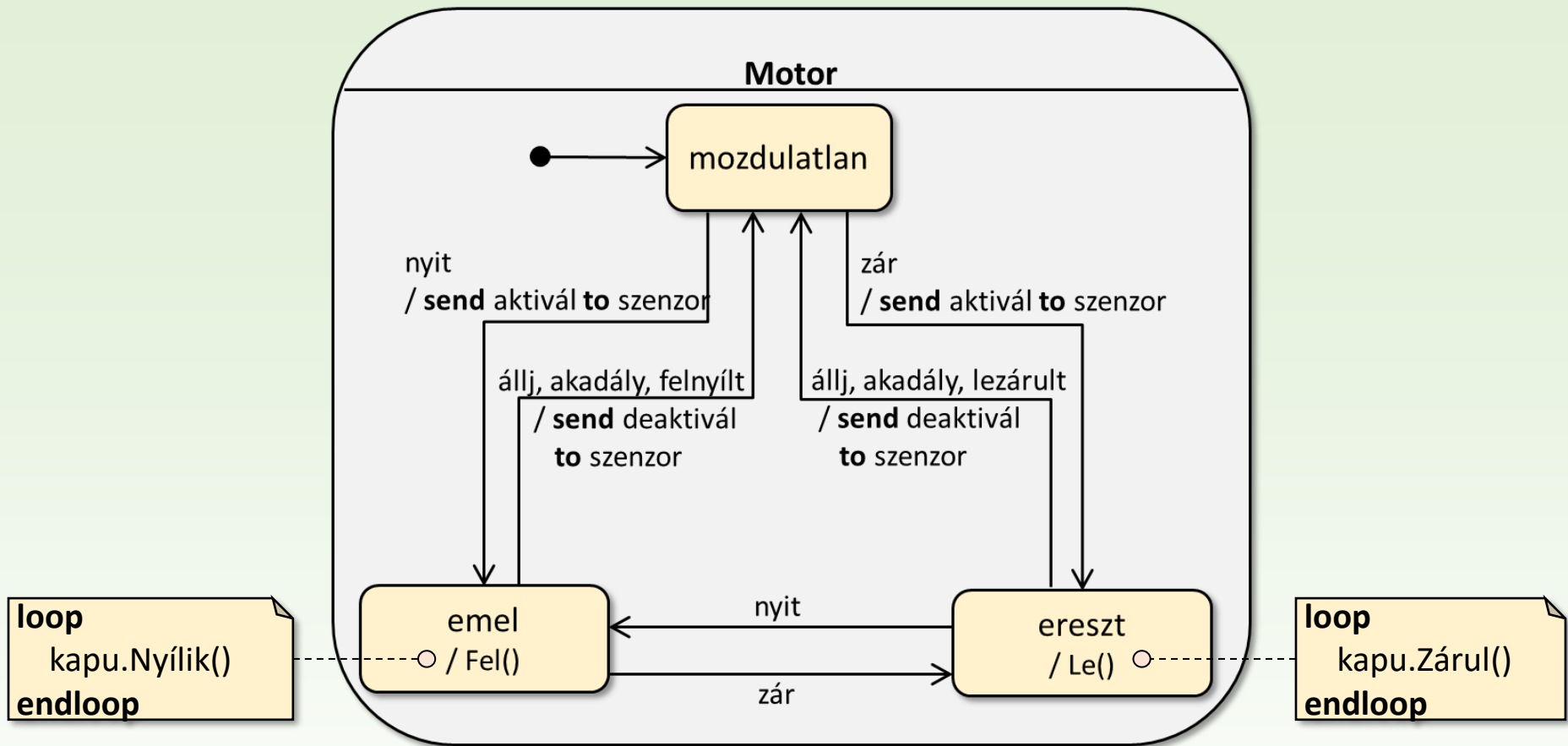


```

éber := false
  
```

```

éber := true
while éber loop
  if "akadály" then send akadály to motor
endloop
  
```



Motor	
mozgás : bool	
Fel()	○
Le()	○
Stop()	○

```

mozgás := true
while mozgás loop kapu.Nyílik() endloop
  
```

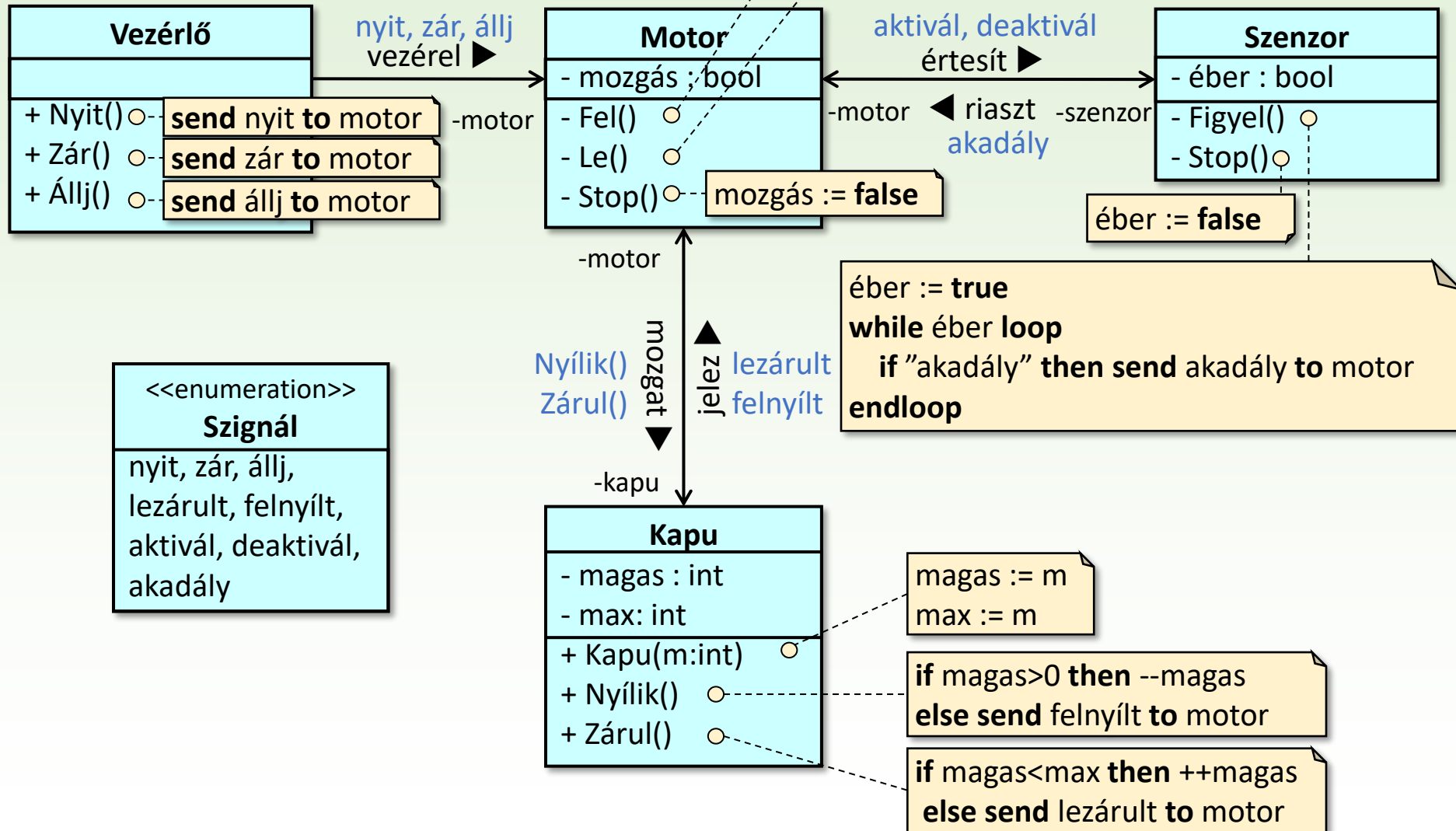
```

mozgás := true
while mozgás loop kapu.Zárul() endloop
  
```

```

mozgás := false
  
```

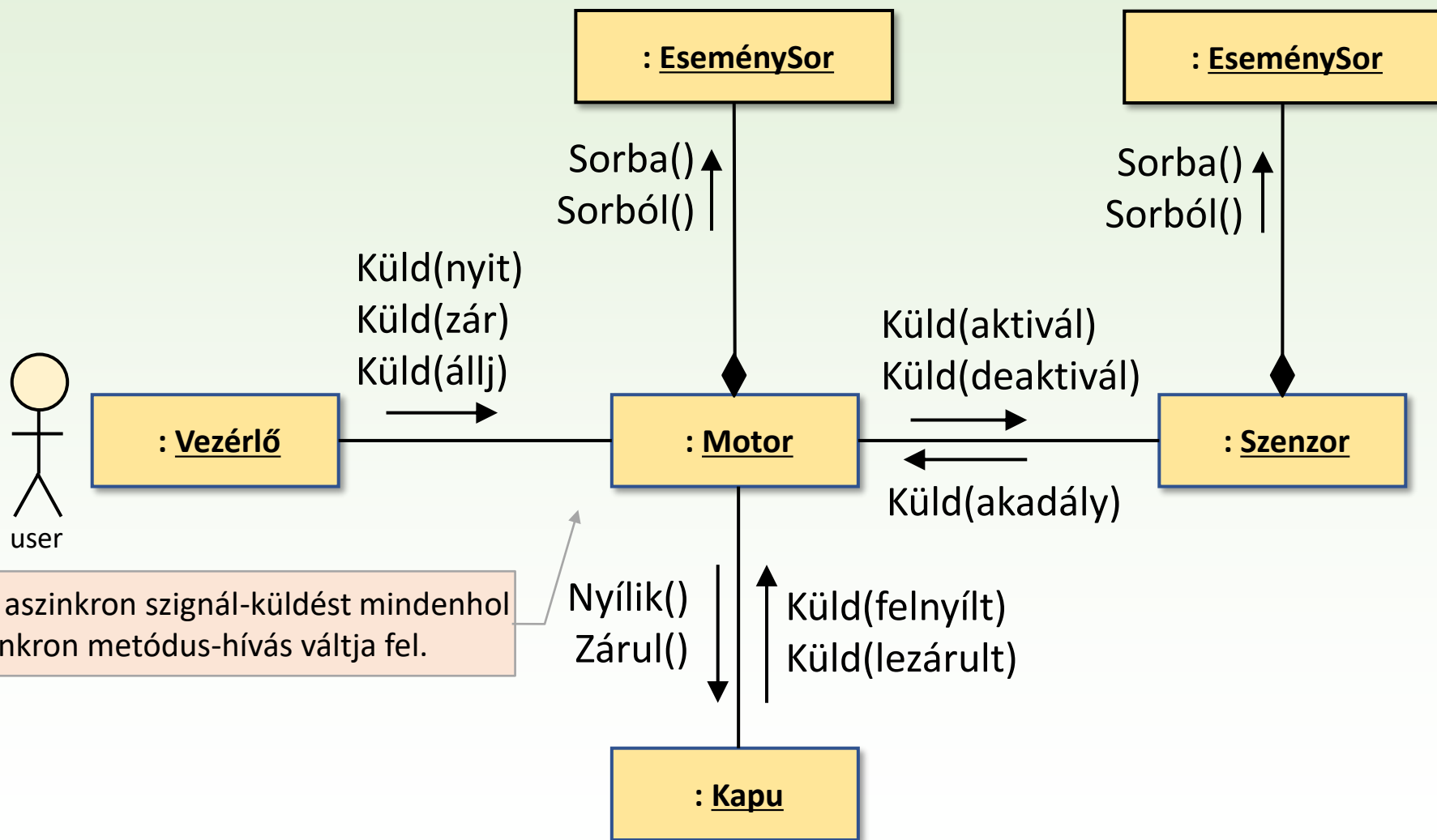
# Osztály diagram



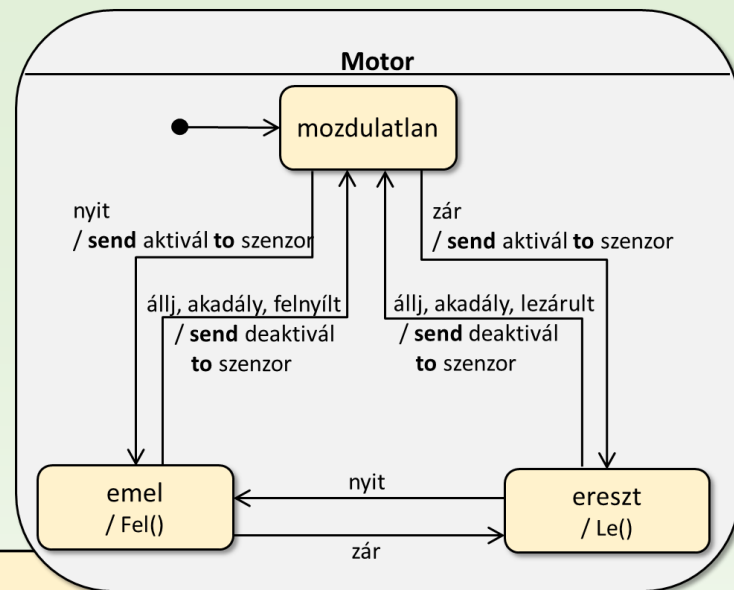
# Szignálok aszinkron feldolgozásával vezérelt állapotgép megvalósítása

- ❑ Ha egy objektumnak aszinkron módon kell feldolgoznia a neki küldött szignálokat, akkor az **állapotgépét párhuzamosan kell futtatni** a rendszer többi komponensének működésével.
- ❑ A beérkező szignálokat egy külön **eseménysorban** kell gyűjteni.
  - A **küldő objektum a szignált saját szálát használva helyezi el** a fogadó objektum eseménysorába úgy, hogy azt paraméterként átadja a fogadó objektum erre szolgáló Küld() metódusának.
  - A **fogadó objektum saját szálán futó állapotgépe veszi ki** a soron következő szignált az eseménysorból. Üres sor esetén ennek a műveletnek **várakozni kell** újabb szignál beérkezésére.
  - A fentiek miatt az eseménysor Sorba() és Sorból() metódusai egymással párhuzamosan hívódnak meg, ezért ügyelni kell arra, hogy csak **kölcsönösen kizárásos** módon használják a sort.
- ❑ Egy állapot esetleges **belső tevékenységét is külön szálon** kell futtatni, amikor az állapot aktuális lesz.

# Kommunikációs diagram (megvalósítási szint)



# Motor



## EseménySor

+ Sorba(Szignál)  
+ Sorból() : Szignál

- eseménysor

## Motor

- állapot : {emel, ereszt, áll}  
- szenzor : Szenzor  
- kapu : Kapu  
- mozgás : bool

+ Küld(szignál:Szignál)  
- Start(m:Metódus)  
- Fel() {külön szálon}  
- Le() {külön szálon}  
- Stop()  
- Állapotgép() {külön szálon}

eseménysor.Sorba(szignál)

az m elindítása külön szálon

**switched ( szignál )**

**case** zár: Stop(); Start(Le); állapot := ereszt

**case** állj, akadály, felnyílt:

Stop(Fel); szenzor.Küld(deaktivál); állapot := mozdulatlan

**default:** skip

**endswitch**

**switched ( szignál )**

**case** nyit: Stop(); Start(Fel); állapot := emel

**case** állj, akadály, lezárult:

Start(Le); szenzor.Küld(deaktivál); állapot := mozdulatlan

**default:** skip

**endswitch**

**switched ( szignál )**

**case** nyit: szenzor.Küld(aktivál); Start(Fel); állapot := emel

**case** zár: szenzor.Küld(aktivál); Start(Le); állapot := ereszt

**default:** skip

**endswitch**

állapot := mozdulatlan

**loop**

szignál := eseménysor.Sorból()

**switch ( állapot )**

**case** emel: ...

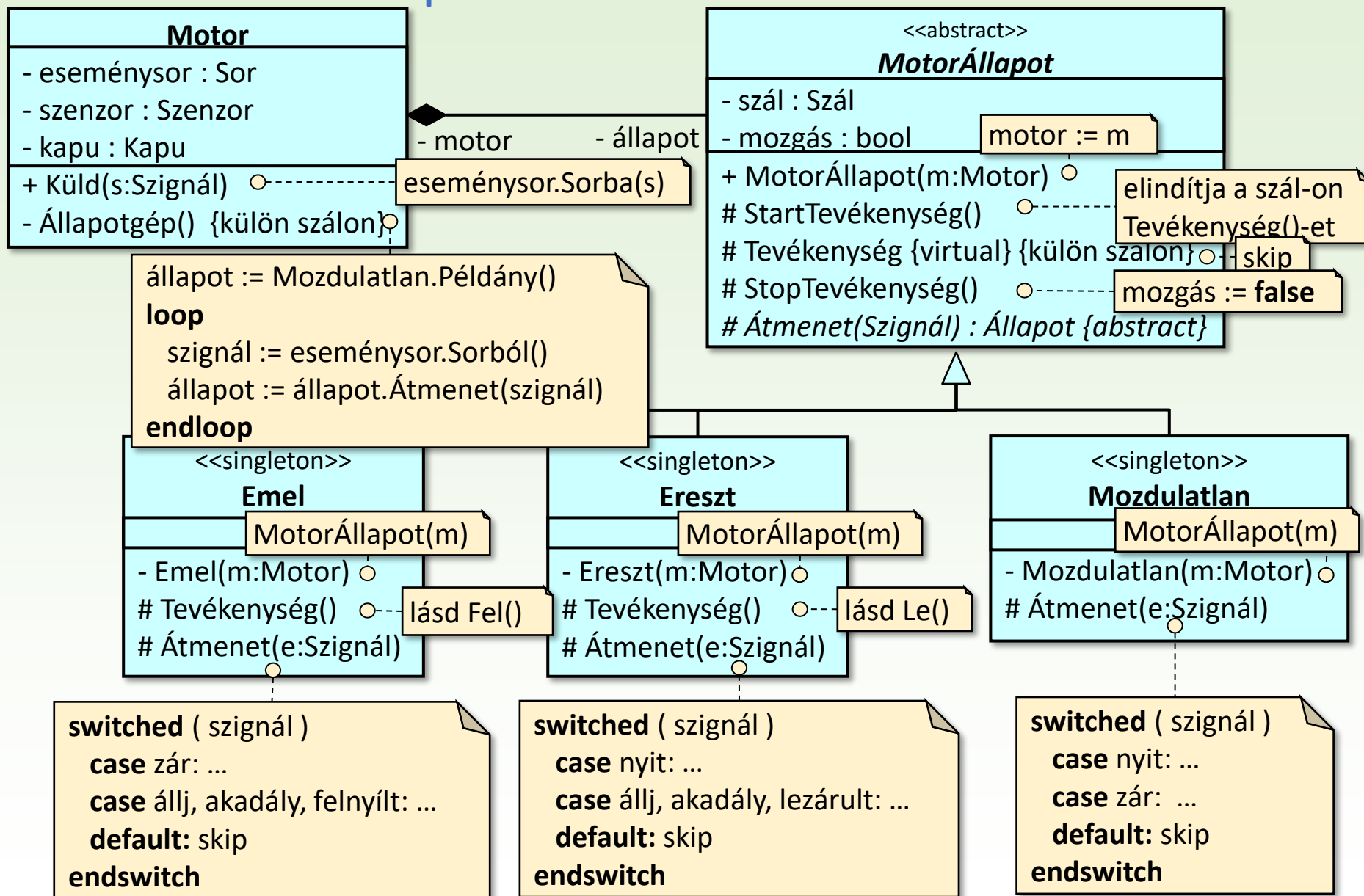
**case** ereszt: ...

**case** áll: ...

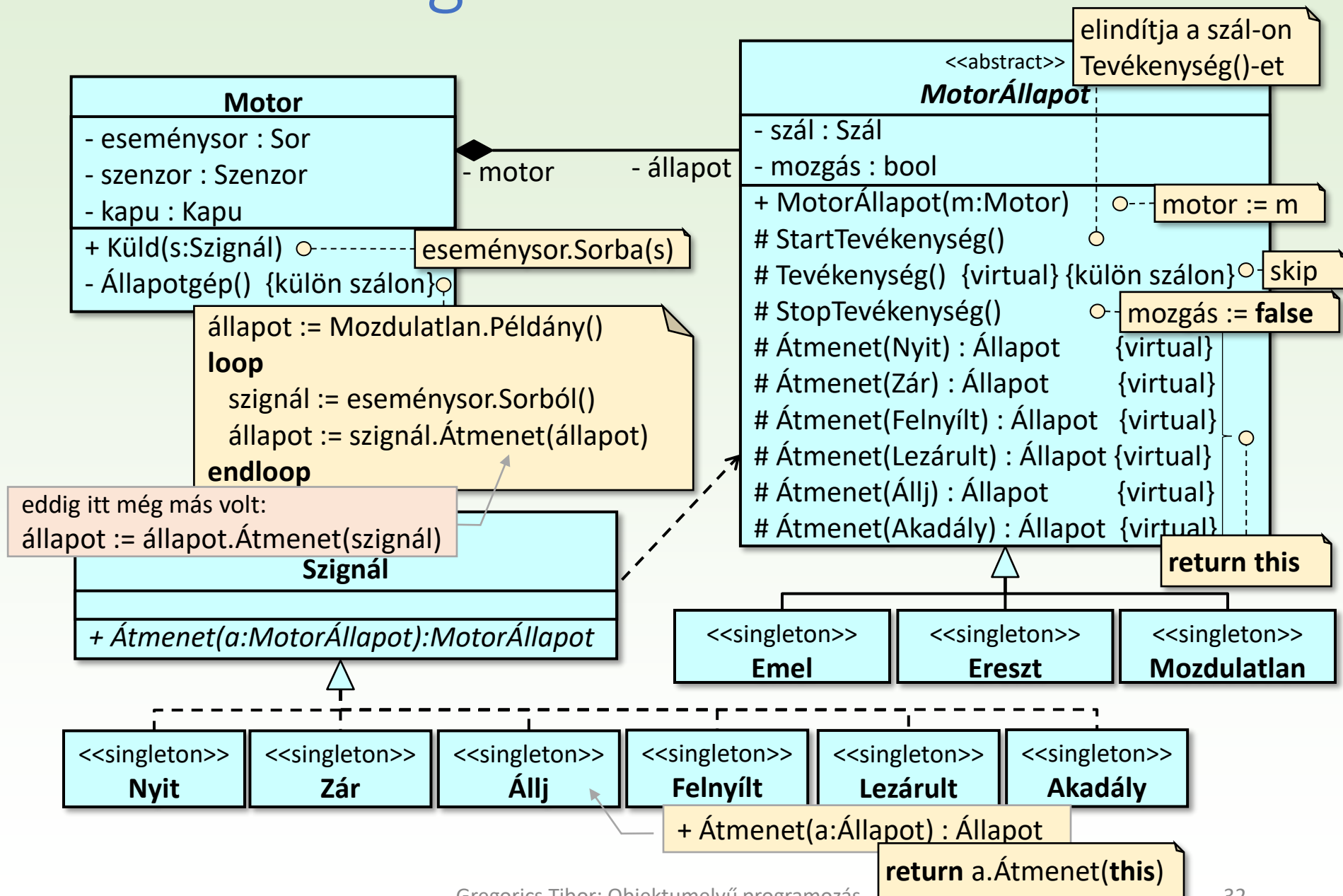
**endswitch**

**endloop**

# Motor – állapot tervmintával

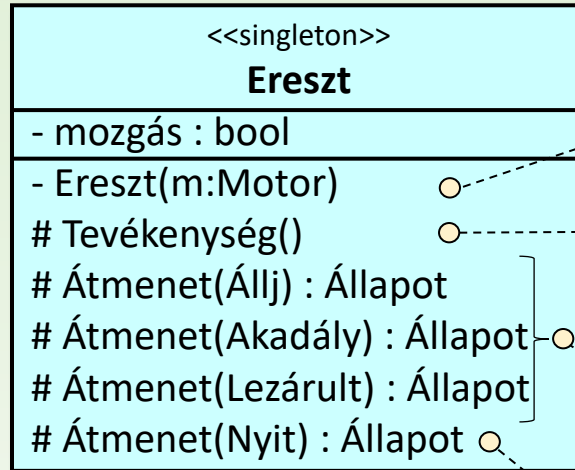


# Motor – látogató tervmintával





# Motor erezst állapota



MotorÁllapot(m)

ez a korábbi Le()

mozgás := **true**

**while** mozgás **loop** motor.kapu.Zárul() **endloop**

StopTevékenység()

motor.szenzor.Küld(Deaktivál.Példány())

**return** Mozdulatlan.Példány()

StopTevékenység()

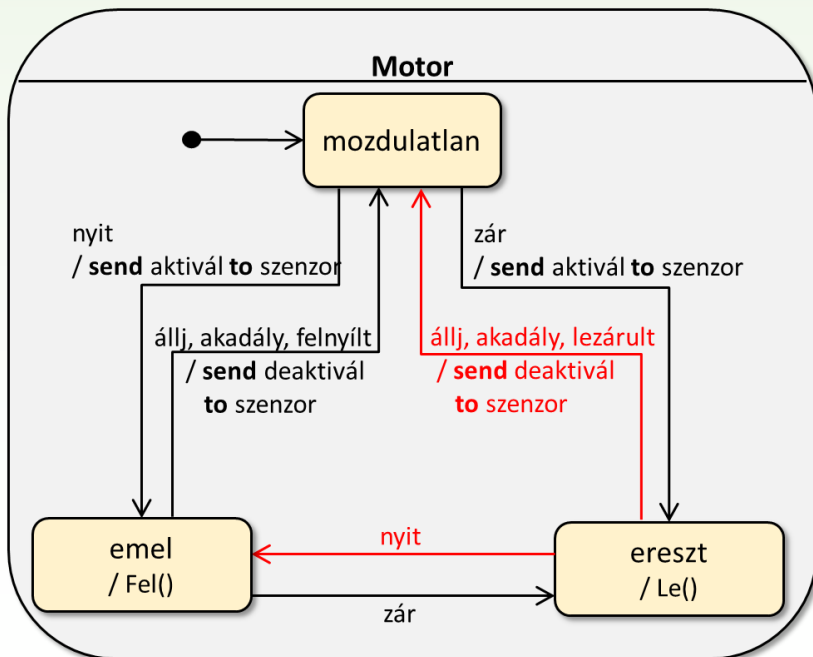
Emel.Példány().StartTevékenység()

**return** Emel.Példány()

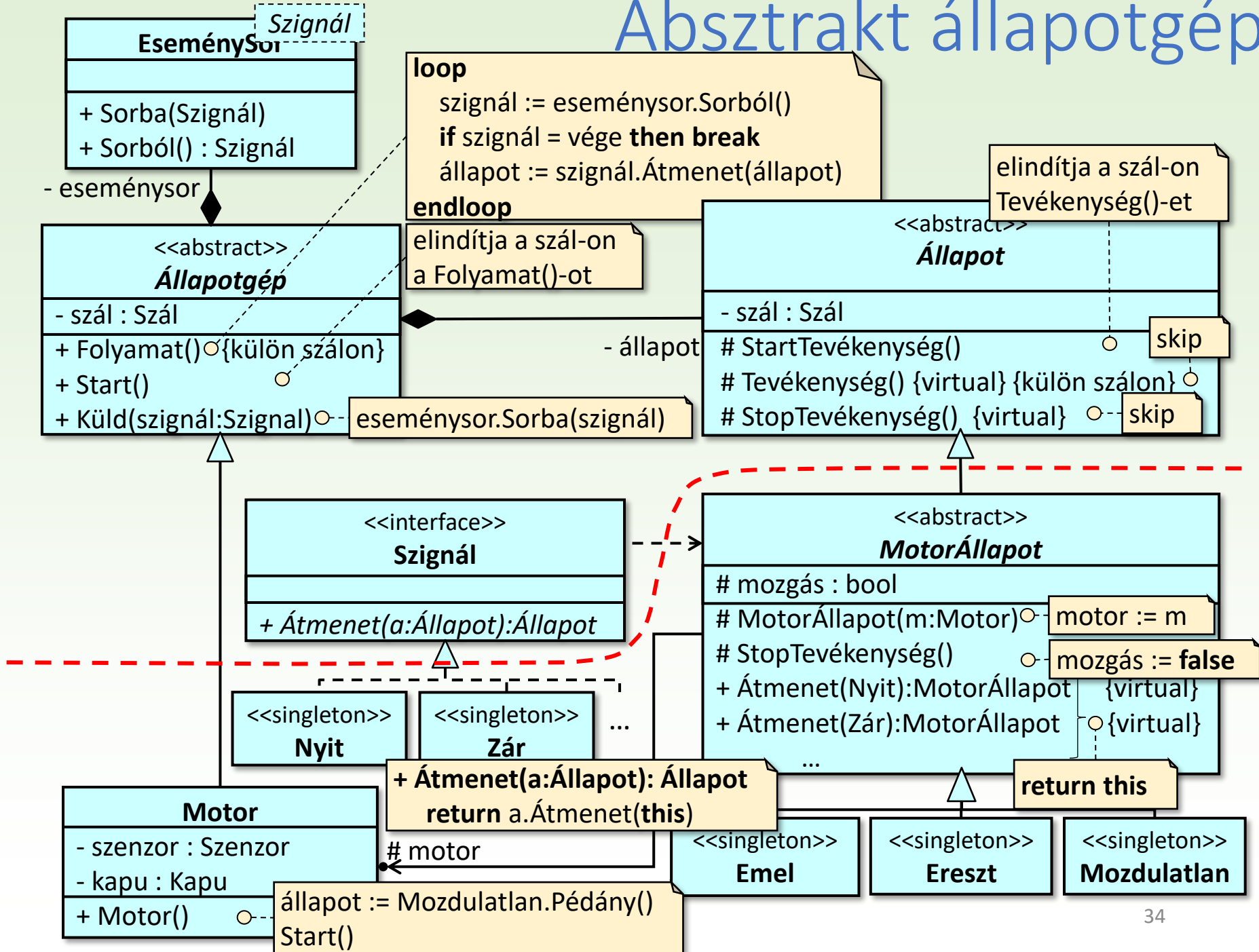
Rövidíthetnénk a kódon, ha az  
Emel.Példány() hívná meg a  
StartTevékenység()-et:

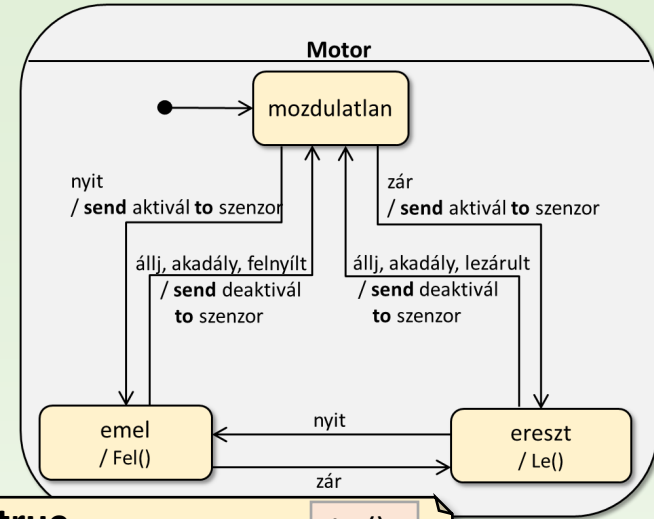
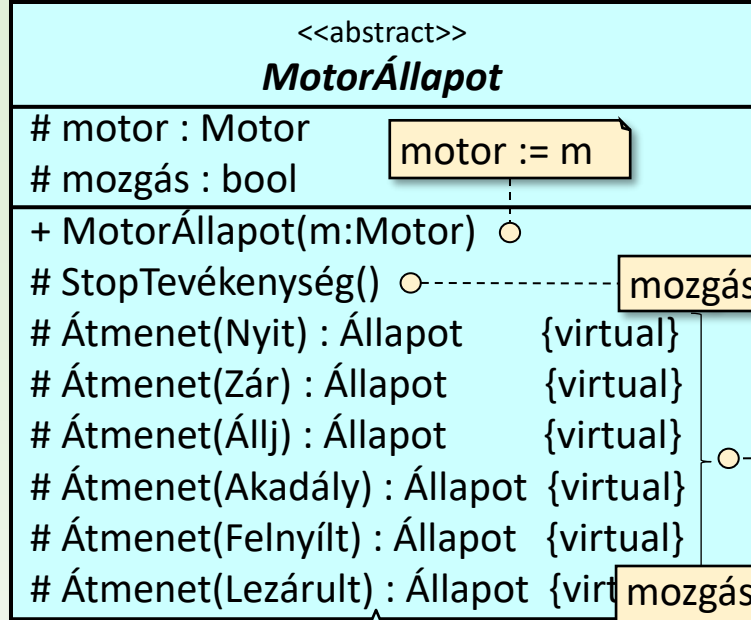
StopTevékenység()

**return** Emel.Példány()



# Absztrakt állapotgép





motor := m

mozgás := false

mozgás := true

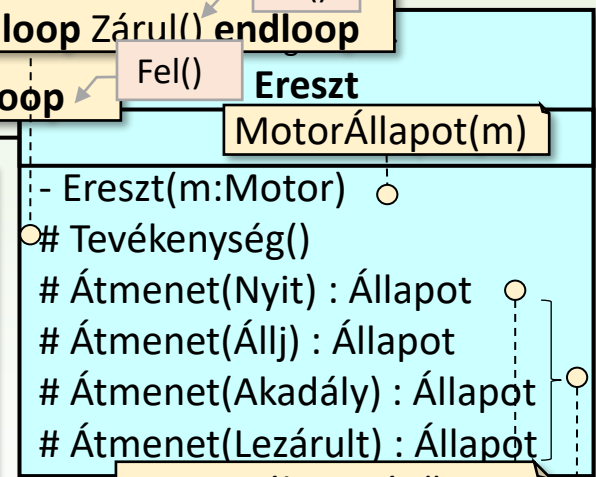
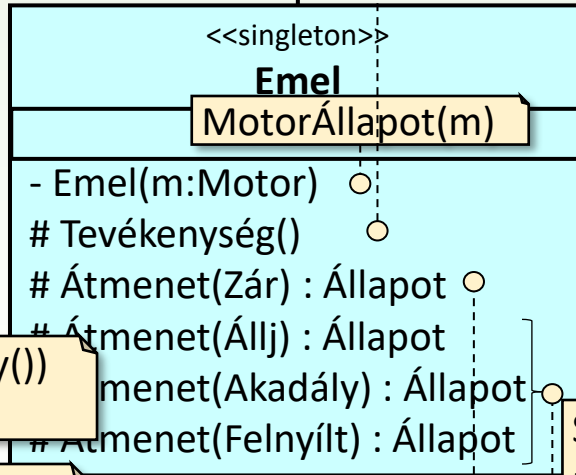
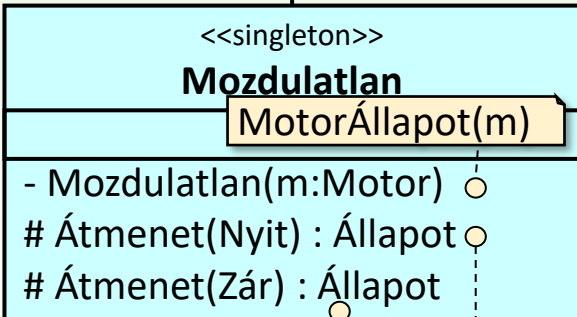
while mozgás loop Zárul() endloop

mozgás := true

while mozgás loop Nyílik() endloop

Ereszt

MotorÁllapot(m)



motor.szenzor.Küld(Aktivál.Példány())

return Ereszt.Példány()

motor.szenzor.Küld(Aktivál.Példány())

return Emel.Példány()

StopTevékenység()

return Ereszt.Példány()

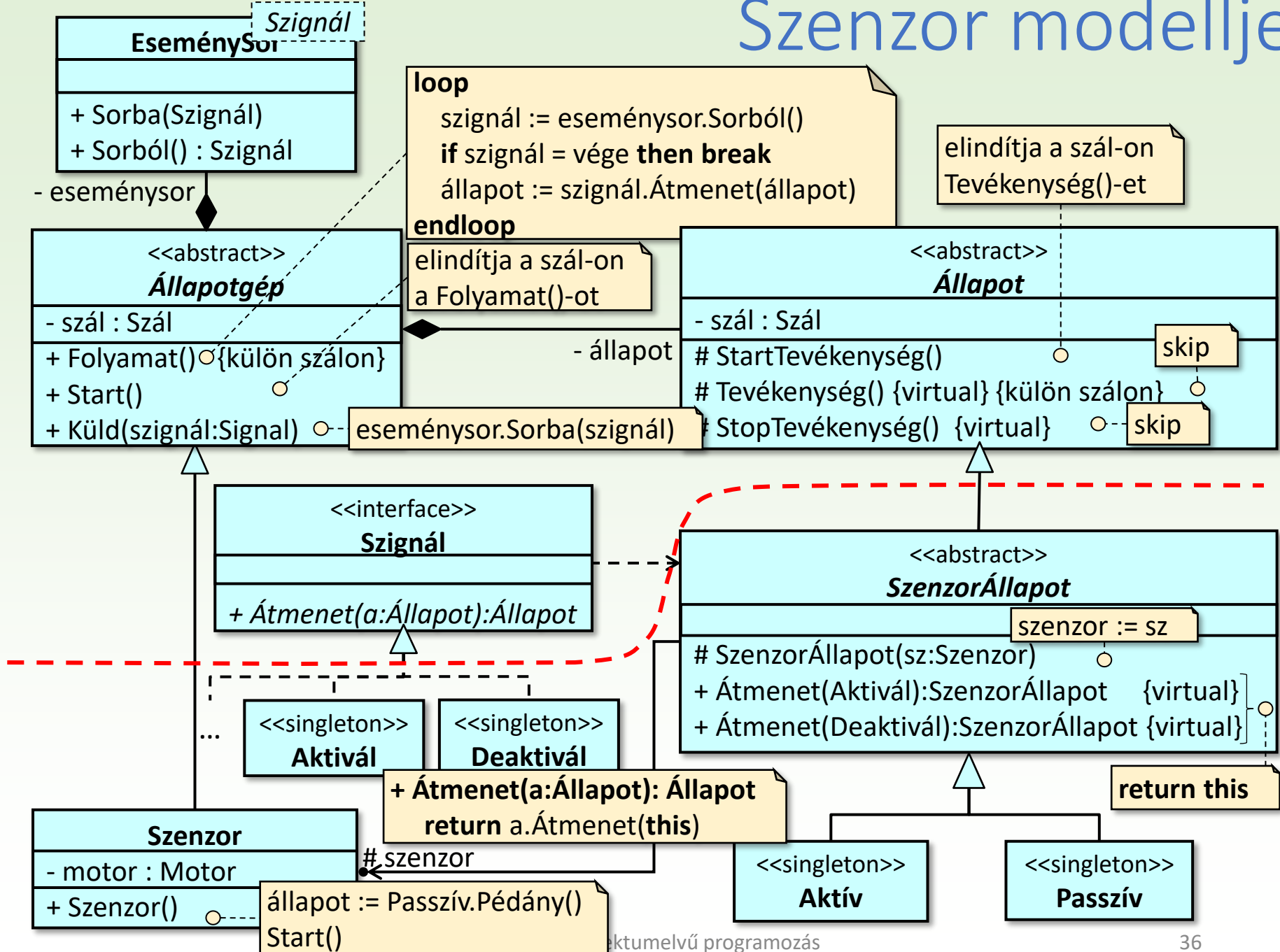
StopTevékenység()

motor.szenzor.Küld(Deaktivál.Példány())

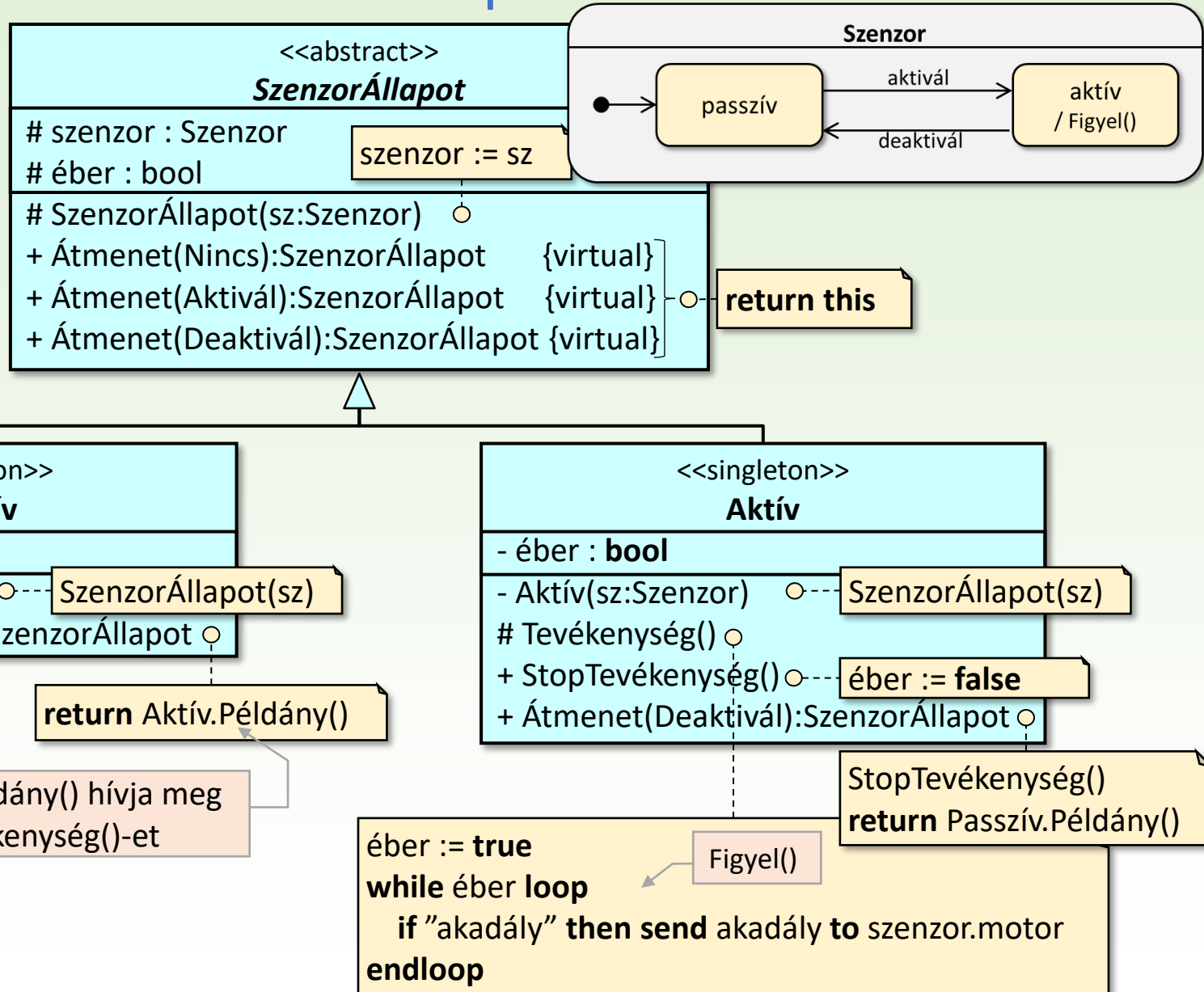
return Mozdulatlan.Példány()

Az egyiket lekérő Példány() hívja meg az örökölt StartTevékenység()-et

# Szenzor modellje



# Szenzor állapotai



# Állapotgép

## 3.rész

### Garázskapu-vezérlés megvalósítása

Gregorics Tibor

[gt@inf.elte.hu](mailto:gt@inf.elte.hu)

<http://people.inf.elte.hu/gt/oep>

# Rendszert leíró osztály (GarageGate.cs)

```
class Program
{
    static void Main( )
    {
        Garagegate system = new();
        system.Process();
    }
}
```

```
class Garagegate
{
    public readonly Engine engine = new();
    public readonly Sensor sensor = new();
    public readonly Gate gate = new(5);
    private readonly Controller controller = new();

    public Garagegate()
    {
        gate = new Gate(5);
        engine.Connect(gate, sensor);
        sensor.Connect(engine);
        gate.Connect(engine);
        controller.Connect(engine);
    }

    public void Process()
    {
        controller.Control();
    }
}
```

objektumok közötti kapcsolatokat  
felépítő metódusok

# Vezérlő osztály (Controller.cs)

```
public void Control()
{
    MenuWrite();
    int v;
    do
    {
        v = int.Parse(Console.ReadLine()); // ellenőrzés kell
        switch (v)
        {
            case 0: engine.Send(Final.Instance());
                    engine.Sensor.Send(Final.Instance());
                    break;
            case 1: engine.Send(Open.Instance());
                    break;
            case 2: engine.Send(Close.Instance());
                    break;
            case 3: engine.Send(Stop.Instance());
                    break;
        }
    } while (v != 0);
}
```

```
class Controller
{
    private Engine Engine { get; private set; }
    public void Connect(Engine engine) { Engine = engine; }
    public void Control() { ... }
    private void MenuWrite() { ... }
}
```

leállási szignál mindkét külön  
szálon futó állapotgépnek

```
private void MenuWrite()
{
    Console.WriteLine("Menupoints:");
    Console.WriteLine("0 - exit");
    Console.WriteLine("1 - up");
    Console.WriteLine("2 - down");
    Console.WriteLine("3 - stop");
}
```



# Szignálok (Signals.cs)

```
public interface ISignal
{
    State Transition(State state) { return state; }
}
```

az összes szignált ehhez hasonlóan definiáljuk

```
public class Open : ISignal
{
    private static Open instance = null;
    private Open() { }
    public static Open Instance()
    {
        instance ??= new Open();
        return instance;
    }

    public State Transition(State state)
    {
        return (state as EngineState).Transition(this);
    }
}
```

egyke tervezési minta

látogató tervezési minta

az Open szignált csak a motor állapotgépére hat

# Eseménysor (EventQueue.cs)

```
class EventQueue<ISignal>
```

```
{  
    private readonly Queue<ISignal> queue = new();  
    private readonly object criticalSection = new();  
    public void Enqueue(ISignal e)  
    {  
        queue.Enqueue(e);  
    }  
    public ISignal Dequeue()  
    {  
        ISignal e = queue.Peek(); queue.Dequeue();  
        return e;  
    }  
}
```

beépített `Queue<>` sablonra épül  
(using System.Collection.Generic)

objektum az egymást kölcsönösen  
kizáró szakaszok azonosítására

Az `Enqueue()` és `Dequeue()` külön szálakon fut, ezért  
hibát okozhat, ha egyszerre használják a queue-t.

`queue.Enqueue(e);`

`ISignal e = queue.Peek(); queue.Dequeue();`  
`return e;`

elindít egy felfüggesztett szálát

`Monitor.Enter(criticalSection);`  
`queue.Enqueue(e);`  
`Monitor.Pulse(criticalSection);`  
`Monitor.Exit(criticalSection);`

```
ISignal e;  
Monitor.Enter(criticalSection);  
if (queue.Count==0) Monitor.Wait(criticalSection);  
e = queue.Peek(); queue.Dequeue();  
Monitor.Exit(criticalSection);
```

felfüggeszti az adott szál végrehajtását

A `Monitor` osztály `Enter()` és `Exit()` metódushívásai  
– amelyek paramétere tetszőleges, de ugyanazon  
objektum – jelölik ki azokat a párhuzamosan futó  
kritikus szakaszokat, amelyek közül egyszerre csak  
legfeljebb csak egy lehet aktív.

# Állapotgép (StateMachine.cs)

```
public abstract class StateMachine
{
    private readonly Thread thread;
    protected State currentState;
    private readonly EventQueue<ISignal> eventQueue = new();
    private readonly ISignal finalSignal;

    public StateMachine(ISignal final)
    {
        finalSignal = final;
        thread = new Thread(new ThreadStart(StateMachineProcess));
    }

    public void Send(Signal signal) { eventQueue.Enqueue(signal); }

    public void Start() { thread.Start(); }

    protected void StateMachineProcess()
    {
        while (true)
        {
            ISignal signal = eventQueue.Dequeue();
            if (signal.Equals(finalSignal)) break;
            else { currentState = signal.Transition(currentState); }
        }
    }
}
```

külön szál a szignálok feldolgozásának

az eseményeket fogadó szálbiztos sor

speciális szignál

metódus hozzárendelése egy szálhoz

szál elindítása

a final szignál észlelésekor leáll

# Szenzor (Sensor.cs)

```
class Sensor : StateMachine.StateMachine
{
    public Sensor() : base(Final.Instance())
    {
        currentState = Passive.Instance(this);
        Start();
    }

    public Engine Engine { get; private set; }

    public void Connect(Engine engine)
    {
        Engine = engine;
    }

    public SensorState CurrentState
    {
        get { return (SensorState)currentState; }
    }
}
```

vége szignál

szenzor kezdő állapota

elindítja külön szálon az állapotgépet

# Szenzor passzív állapota és ősosztályai

```
public abstract class State
```

```
{
```

```
    private Thread thread;
```

```
    protected void StartActivity()
```

```
    {
```

```
        thread = new (new ThreadStart(Activity));
```

```
        thread.Start();
```

```
    }
```

```
    protected virtual void Activity() { }
```

```
    protected virtual void StopActivity() { thread.Join(); }
```

```
}
```

StateMachine névtér

sablonfüggvény tervezési minta

```
abstract class SensorState : State
```

```
{
```

```
    protected Sensor sensor;
```

```
    protected SensorState(Sensor s) { sensor = s; }
```

```
    public virtual State Transition(Activate signal) { return this; }
```

```
    public virtual State Transition(Deactivate signal) { return this; }
```

```
}
```

GarageGate névtér

állapot tervezési minta

```
class Passive : SensorState
```

```
{
```

```
    private static Passive instance = null;
```

```
    private Passive(Sensor s) : base(s) { }
```

```
    public static Passive Instance(Sensor s) {instance??.new Passive(s); return instance;}
```

```
    public override State Transition(Activate signal) { return Active.Instance(sensor); }
```

```
}
```

GarageGate névtér

egyke tervezési minta

állapotváltozás

látogató tervezési minta

# Szenzor aktív állapota

```
class Active : SensorState
```

```
{
```

```
    private static Active instance = null;
```

```
    private Active(Sensor s) : base(s) { }
```

```
    public static Active Instance(Sensor s)
```

```
    {
```

```
        instance ??= new Active(s);
```

```
        instance.StartActivity();
```

```
        return instance;
```

```
    }
```

```
    private readonly Random rand = new();
```

```
    private bool awake;
```

```
    protected override void Activity()
```

```
    {
```

```
        awake = true;
```

```
        while (awake) if (rand.Next()%100 < 15) sensor.Engine.Send(Blockage.Instance());
```

```
    }
```

```
    protected override void StopActivity()
```

```
    {
```

```
        awake = false; base.StopActivity();
```

```
    }
```

```
    public override State Transition(Deactivate signal)
```

```
    {
```

```
        StopActivity(); return Passive.Instance(sensor);
```

```
    }
```

```
}
```

indul az állapot belső tevékenysége

szignál küldése a motornak

leáll az állapot belső tevékenysége

átmenet másik állapotba

# Motor (Engine.cs)

```
class Engine : StateMachine.StateMachine
{
    public Gate Gate { get; private set; }
    public Sensor Sensor { get; private set; }

    public void Connect(Gate gate, Sensor sensor)
    {
        Gate = gate;
        Sensor = sensor;
    }

    public Engine() : base(Final.Instance())
    {
        currentState = Unmove.Instance(this);
        Start();
    }

    public EngineState CurrentState
    {
        get { return (EngineState)currentState; }
    }
}
```

vége szignál

kezdő állapot

elindítja az állapotgépet

# Motor állapotainak űsosztálya

```
abstract class EngineState : State
{
    protected Engine engine;
    protected EngineState(Engine e) { engine = e; }

    public virtual State Transition(Open signal)      { return this; }
    public virtual State Transition(Close signal)     { return this; }
    public virtual State Transition(Stop signal)      { return this; }
    public virtual State Transition(Unrolled signal)  { return this; }
    public virtual State Transition(Coiled signal)    { return this; }
    public virtual State Transition(Blockage signal)  { return this; }

    protected bool moving;
    protected override void StopActivity() { moving = false; base.StopActivity(); }
}
```

a motor állapotgépére ható szignálok



# Motor erezst állapota

```
class Downward : EngineState
{
    private static Downward instance = null;
    private Downward(Engine e) : base(e) { }
    public static Downward Instance(Engine e)
    {
        instance ??= new Downward(e);
        instance.StartActivity();
        return instance;
    }
    protected override void Activity()
    {
        moving = true;
        while (moving) { engine.Gate.Closing(); Thread.Sleep(500); }
    }
    public override State Transition(Open signal)
    {
        StopActivity();
        return Upward.Instance(engine);
    }
    public override State Transition(Stop signal)
    {
        StopActivity();
        engine.Sensor.Send(Deactivate.Instance());
        return Unmove.Instance(engine);
    }
    ...
}
```

indul az állapot belső tevékenysége

leáll az állapot belső tevékenysége

átmenet másik állapotba

szignál küldése

# Kapu osztály (Gate.cs)

```
class Gate
{
    public Engine Engine { get; private set; }

    private readonly int maxLength;
    public int currentLength;

    public Gate(int m)
    {
        maxLength = m;
        CurrentLength = m;
    }

    public void Connect(Engine engine) { Engine = engine; }

    public void Openning()
    {
        if (CurrentLength > 0) --CurrentLength;
        if (0 == CurrentLength) Engine.Send(Coiled.Instance());
    }

    public void Closing()
    {
        if (CurrentLength < maxLength) ++CurrentLength;
        if (maxLength == CurrentLength) Engine.Send(Unrolled.Instance());
    }
}
```