

Állapotgép

1.rész

UML állapotgép diagramja

Gregorics Tibor

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.

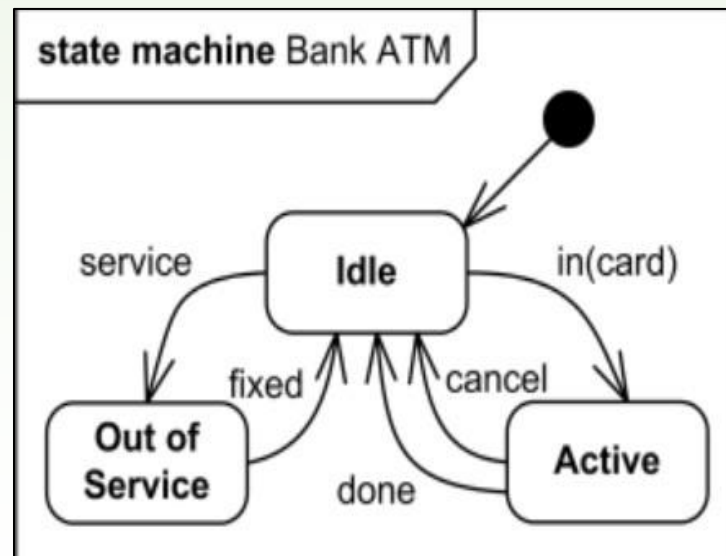
Esemény

- ❑ 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

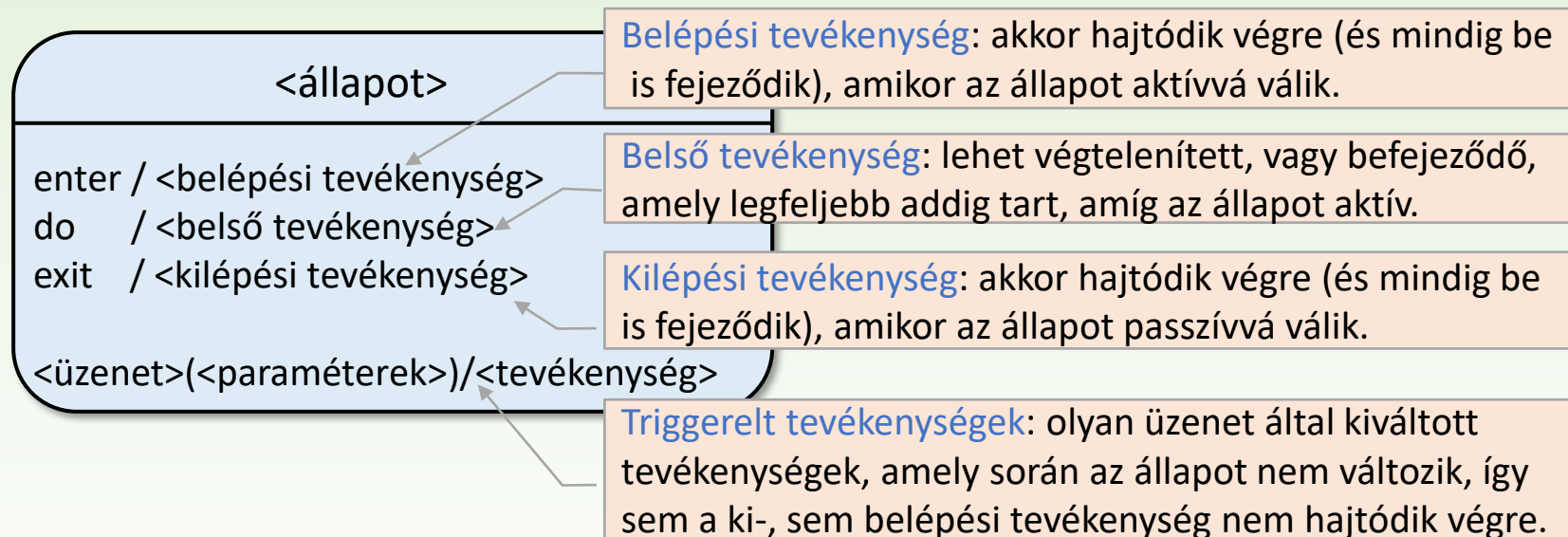
<állapot>

<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 írhatjuk le az állapot által képviselt fizikai állapotokat. (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

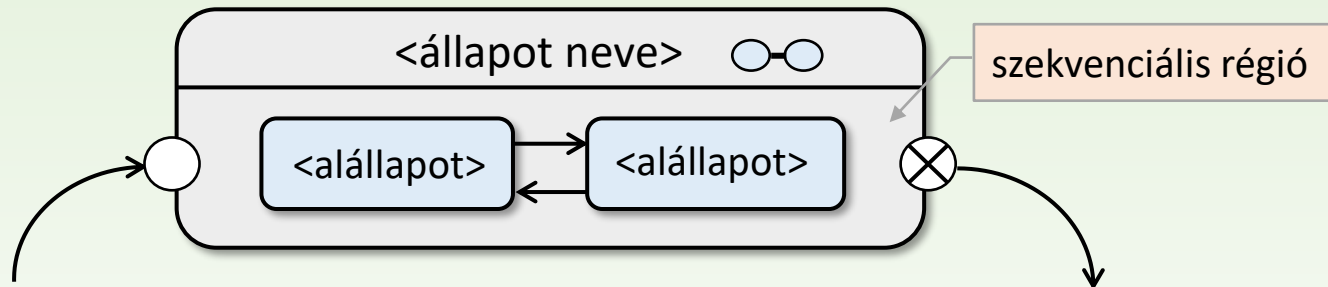
- Egy állapothoz négy féle tevékenység tartozhat, amelyek olyankor hajtódnak végre, amikor az állapot aktív.



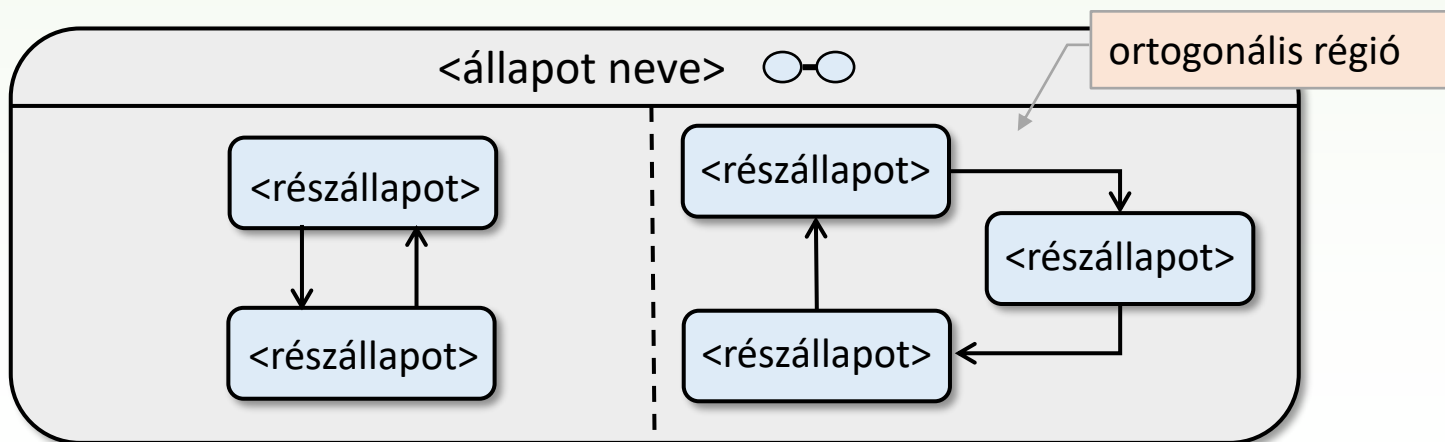
- Egy állapotból egy másikba történő átmenet (ami lehet reflexív is) során az aktív állapot kilépési tevékenysége fut le először, ezt követi az átmenethez tartozó tevékenység (ha van ilyen), végül az új állapot aktívvá válásával együtt annak belépési tevékenységére kerül sor, amit a belső tevékenységének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

Hierarchikus állapotok



Szekvenciális állapotgép hierarchikus állapota:

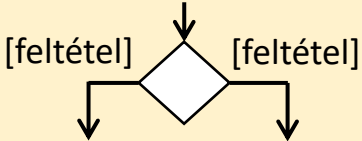
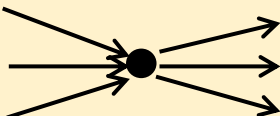







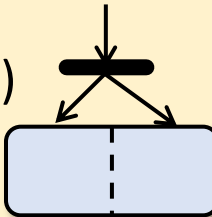
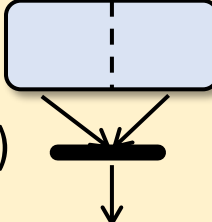
Párhuzamos állapotgép hierarchikus állapota:



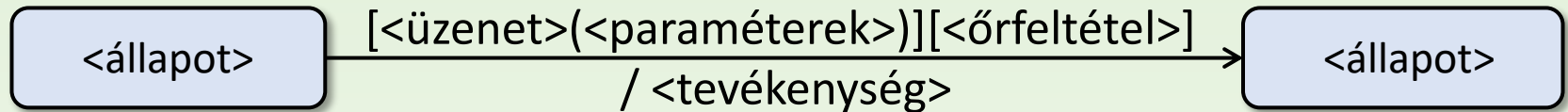
Pszeudo állapotok

- kezdő állapot 
- végállapot 

- elágazás (choice) 
- csomópont (junction) 

- belépés (entry) 
- kilépés (exit) 
- megszüntetés 
- shallow history 
- deep history 
- szétágazás (fork) 
- összefutás (join) 

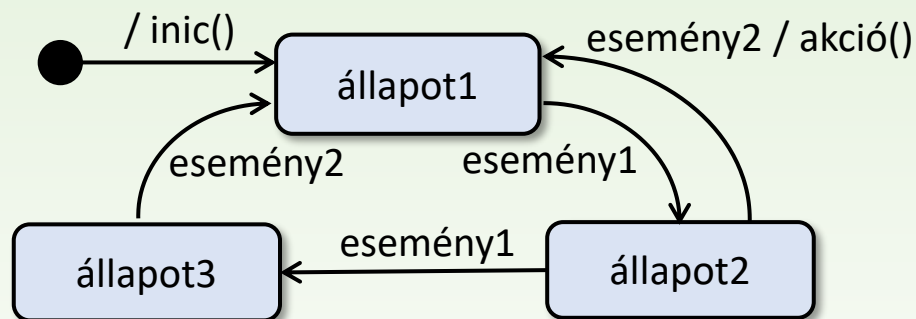
Állapot-átmenet



- ❑ Egy állapotból egy másikba történő átmenet mindig valamilyen esemény hatására következik be. (Ennek részletezését lásd később.)
- ❑ 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

az elágazások helyett az állapot tervezési mintát érdemes alkalmazni

```

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
    
```

```

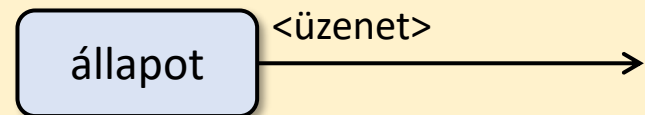
inic()
állapot := állapot1
while állapot ≠ stop loop
  switch állapot
    case állapot1:
      switch esemény
        case esemény1 : állapot := állapot2
        endswitch
    case állapot2:
      switch esemény
        case esemény1 : állapot := állapot3
        case esemény2 : akció()
            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

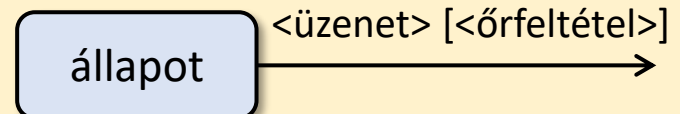
Egy állapotból azonos üzenettel nem vezethet ki egynél több őrfeltétel nélküli él.



determinisztikus

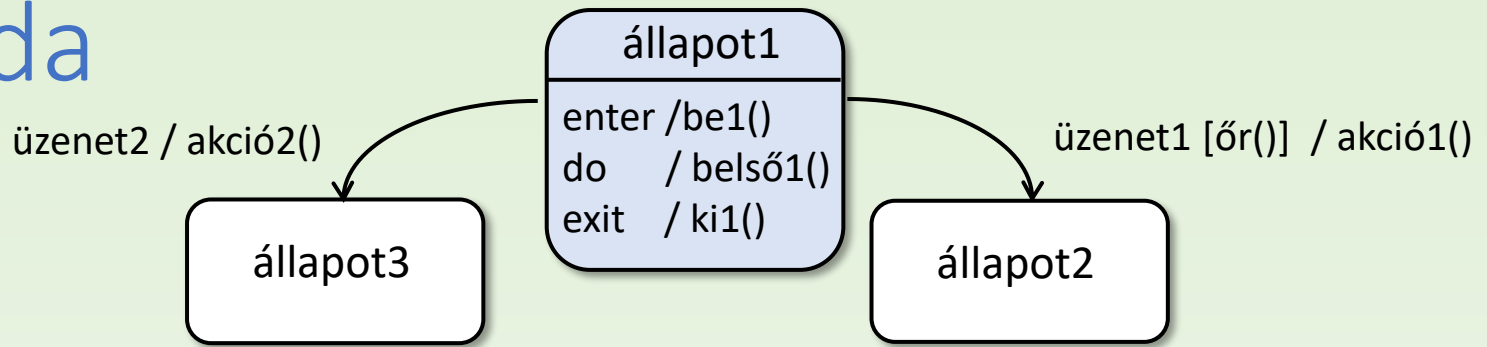
- **őrfeltétel esetén** csak akkor valósul meg, ha az **őrfeltétel éppen teljesül** az üzenet beérkezésekor.

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

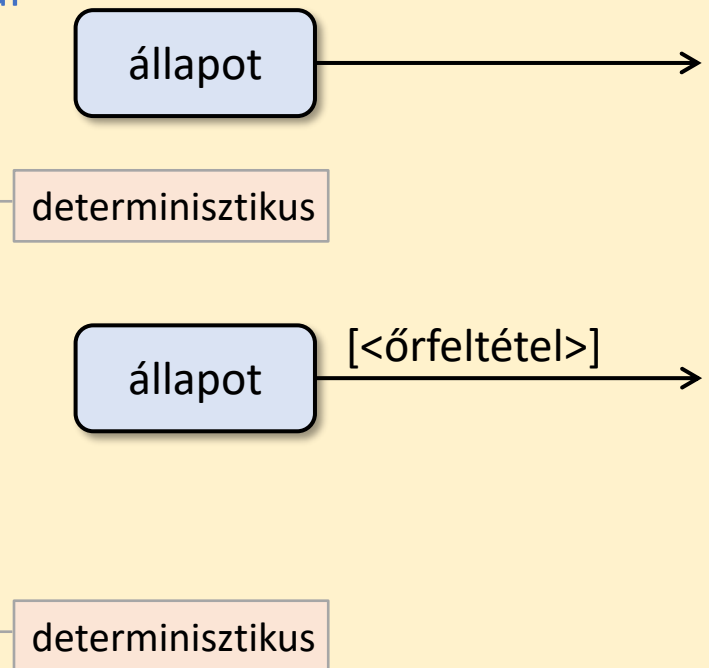
Amikor az átmenetet **nem üzenet váltja ki**, hanem az állapot belső tevékenységének befejeződése (ennek hiányában az állapot aktívvá válása), akkor

- az átmenet **őrfeltétel hiányában azonnal** megvalósul.

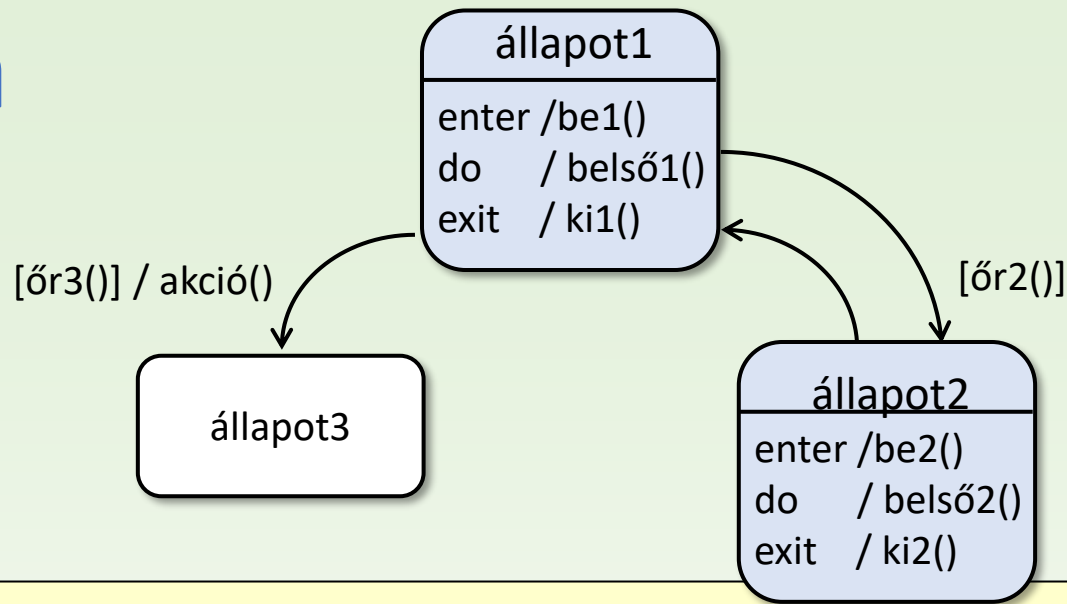
Ilyen átmenete egy állapotnak csak egy lehet.

- **őrfeltétel esetén várakozik** az átmenet arra, hogy az őrfeltétel igaz legyen, de ez megszakad, és nem történik semmi, ha közben egy másik állapot-átmenet következik be.

Egy állapotból kivezető üzenet nélküli átmenetek őrfeltételei diszjunktak.



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

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

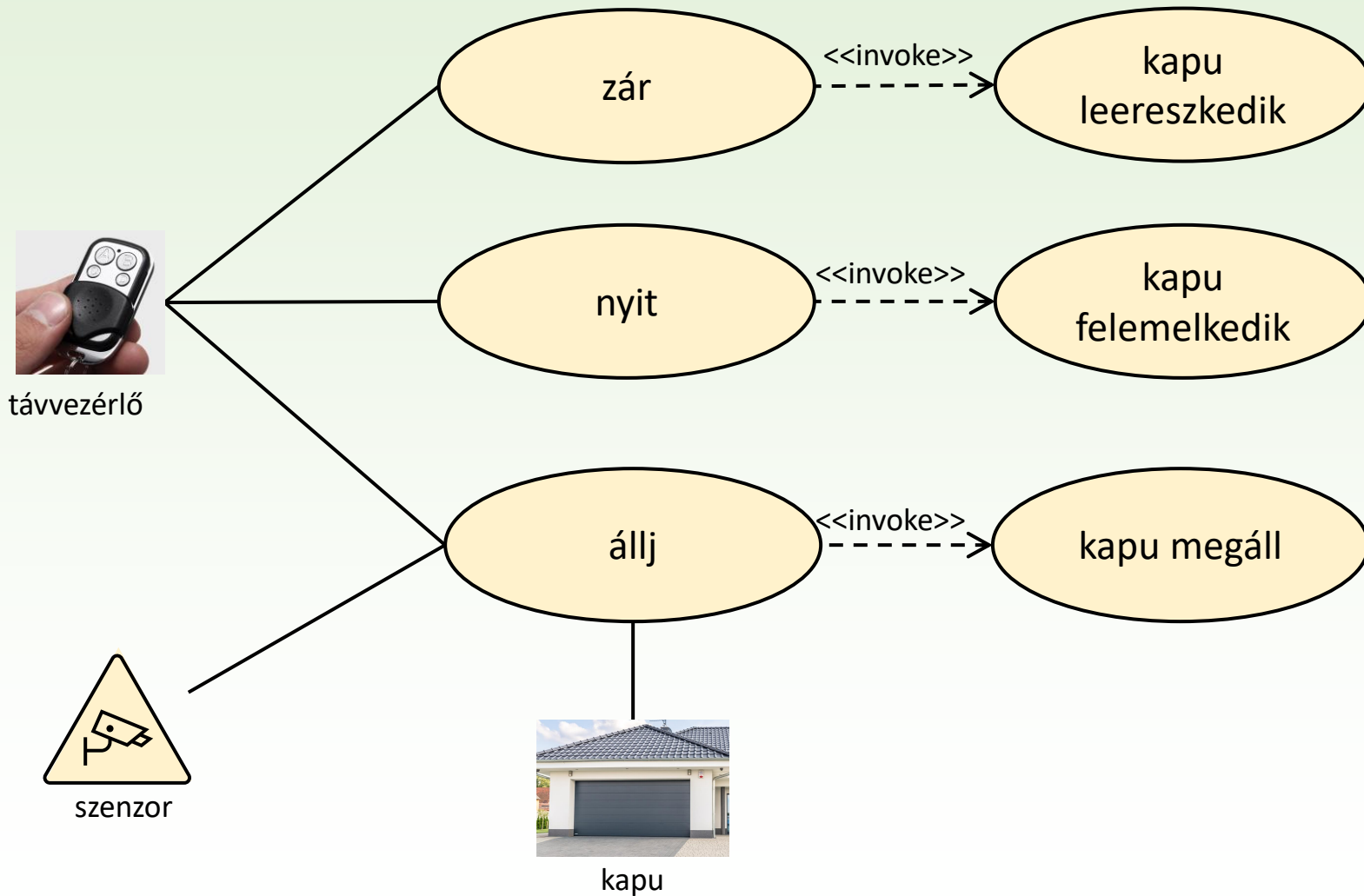
Feladat

Tervezzük meg, és implementáljuk egy függőlegesen nyíló garázskapu működését!

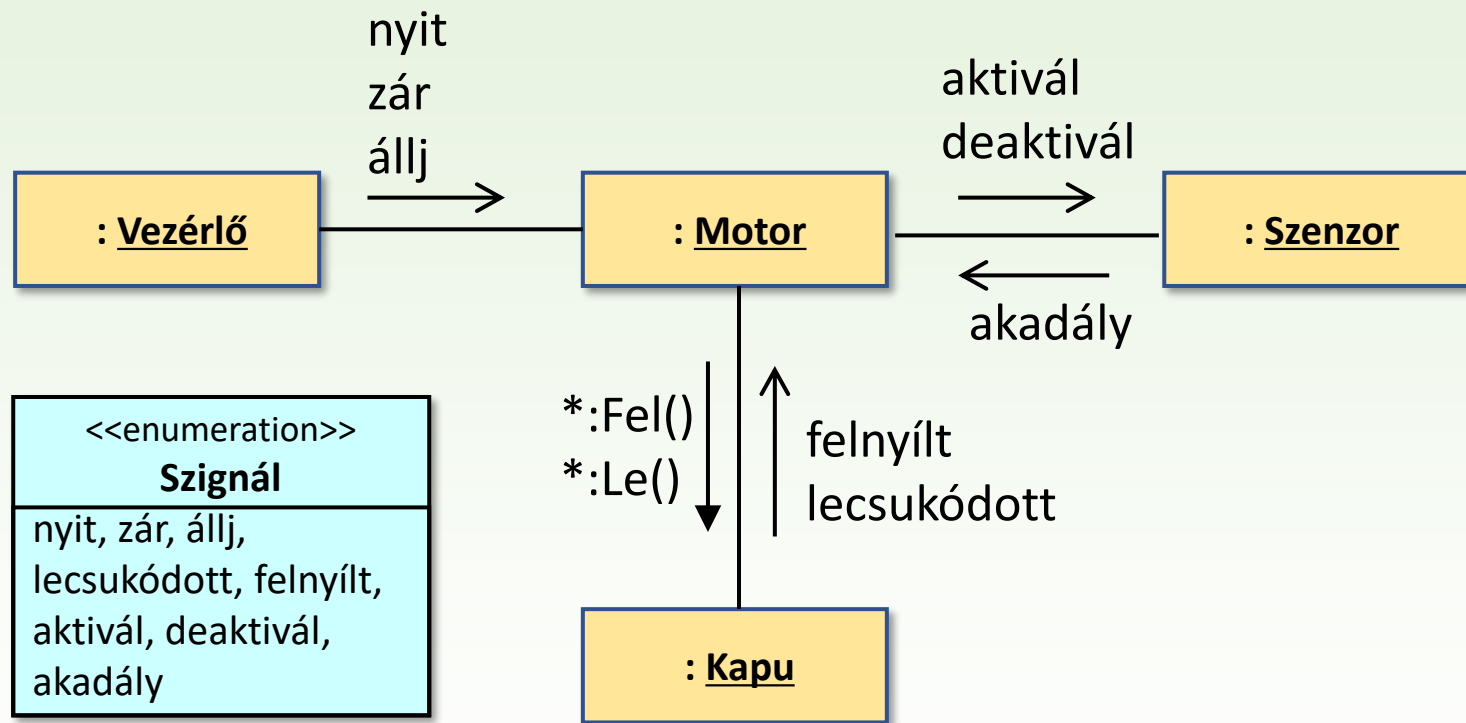
- A garázskaput, pontosabban az azt mozgató motort egy távirányító vezérli. A távirányító három féle parancsot adhat a kapunak: záródj, nyíljál, állj meg.
- A garázskaput a motor a legutoljára kiadott parancs szerint mozgatja.
- A kapu mozgása magától leáll, ha a kapu már teljesen kinyílt, vagy lecsukódott, vagy ha valamilyen akadályt jelet az a speciális szenzor, amelyet a kapun helyeztünk el.



Használati eset diagram



Kommunikációs diagram



: Vezérlő

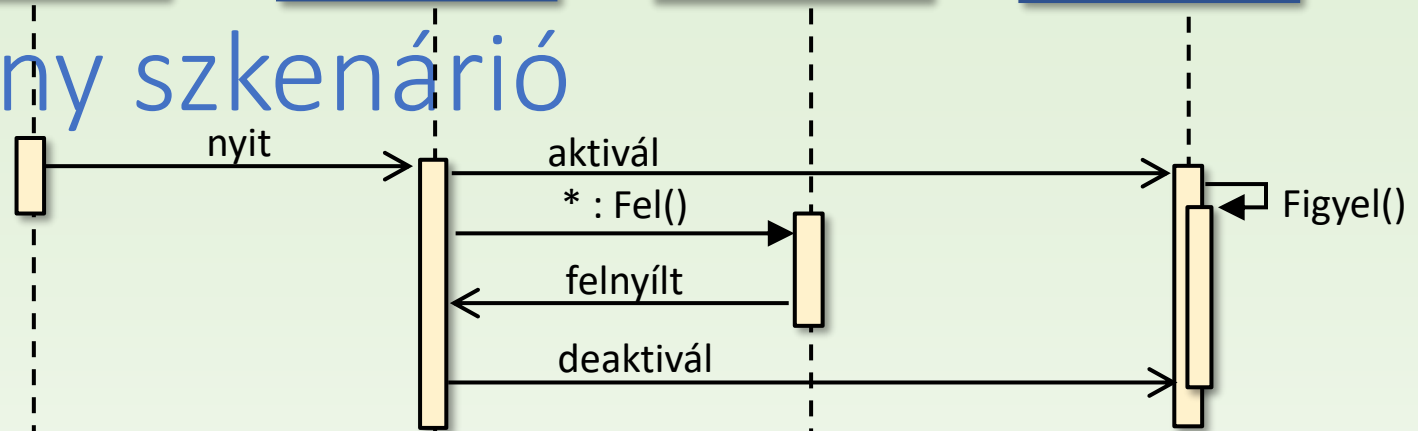
: Motor

: Kapu

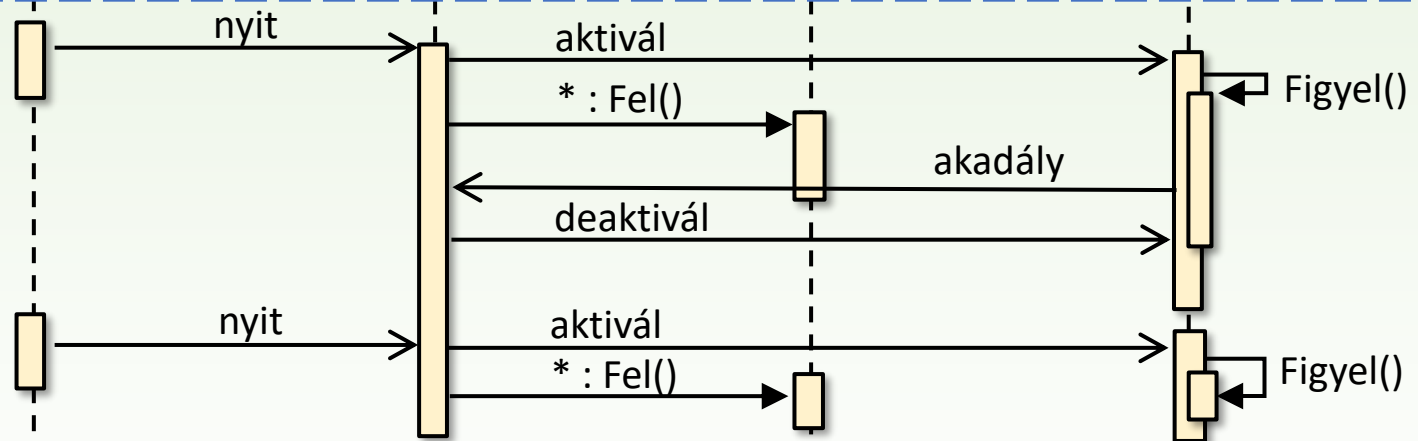
: Szenzor

Néhány szkenárió

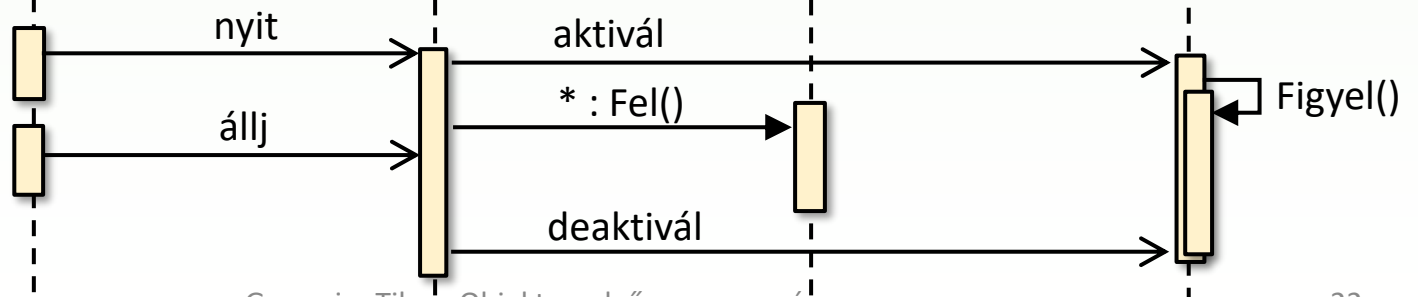
1. eset



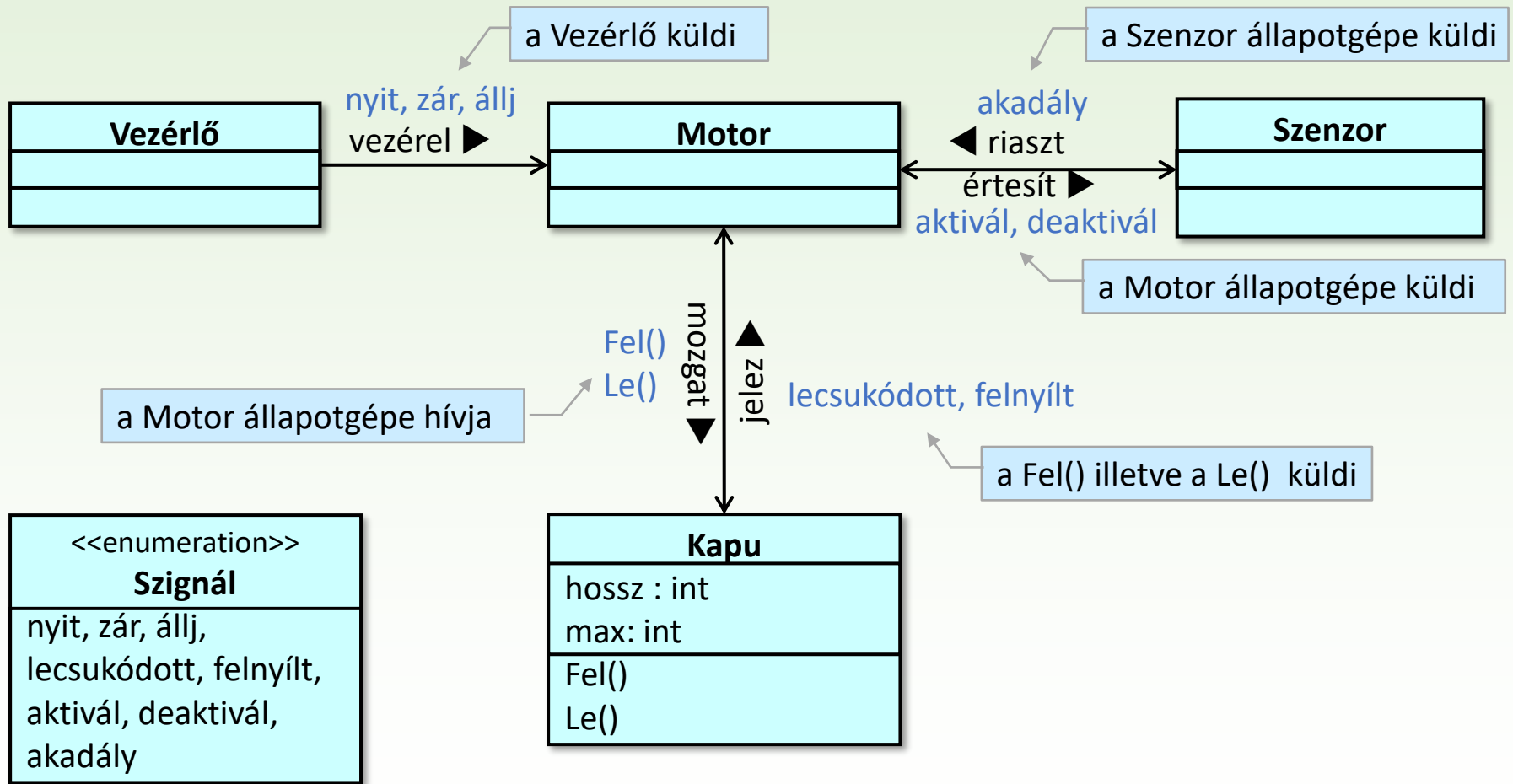
2. eset



3. eset



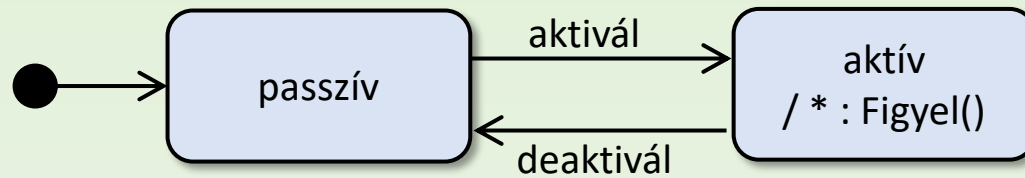
Osztály diagram



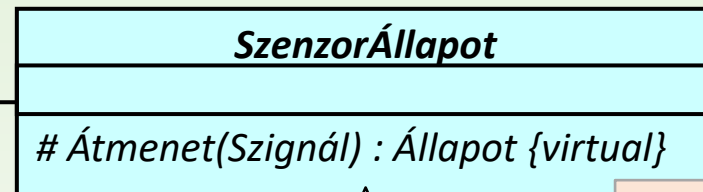
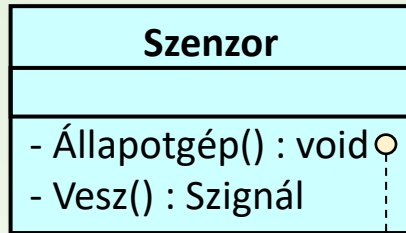
Állapotgépek

- ❑ Az objektumok egy része (motor, szenzor) aszinkron dolgozza fel a nekik küldött szignálokat, mások (kapu) szinkron metódus-hívásokat fogad.
 - a **motor**, a vezérlőtől és a szenzortól kapott szignálokat a beérkezésük sorrendjében, a saját működésével aszinkron módon dolgozza fel egy külön szálon megvalósított állapotgéppel
 - a **szenzor** külön szálon futó állapotgépe figyel az akadályokra, ha az állapota aktív, amelyet a motor állít be.
 - a **kaput** szinkron hívásokkal vezérli a motor; ezért nem készítünk hozzá külön szálon futó állapotgépet.
 - a **vezérlő** egység csak a felhasználó jelzéseit továbbítja a motor és a szenzor felé; nincsenek megkülönböztethető állapotai.

Szenzor állapotgépe



állapot esemény	aktív	passzív
aktivál		aktív
deaktivál	passzív	
nincs	/ Figyel()	

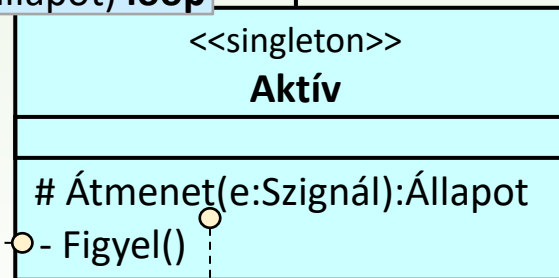


állapot tervminta

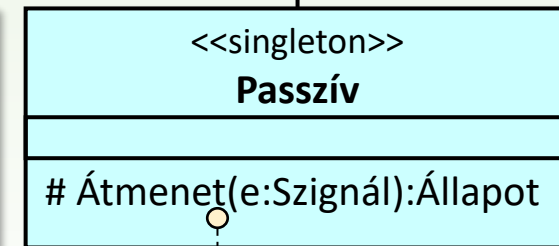
akt := Passzív.Példány(this)

loop while not (akt is a végállapot) loop
 akt := akt.Átmenet(Vesz())
endloop

if „akadály” then motor.Küld(akadály)

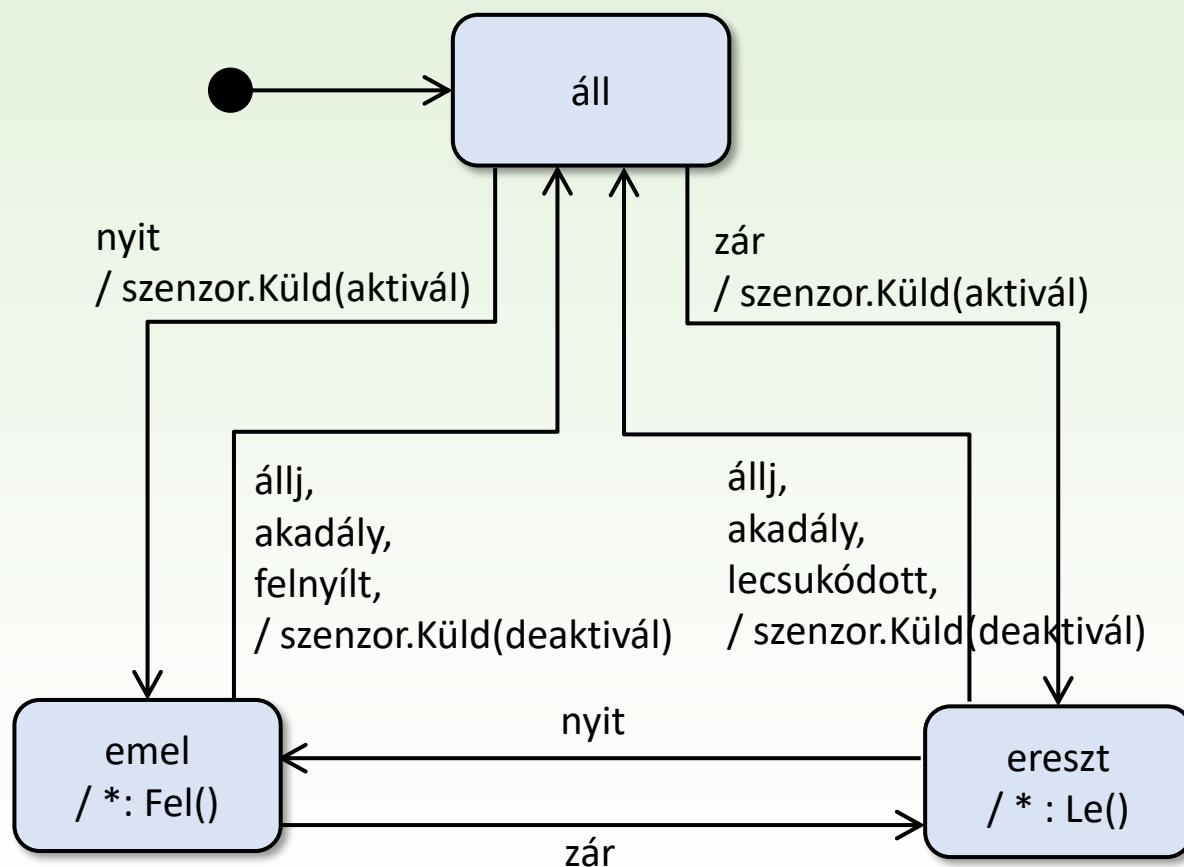


switch e
 case aktivál: return this;
 case deaktivál: return Passzív.Példány(szenzor)
 case nincs: Figyel(); return this;
endswitch

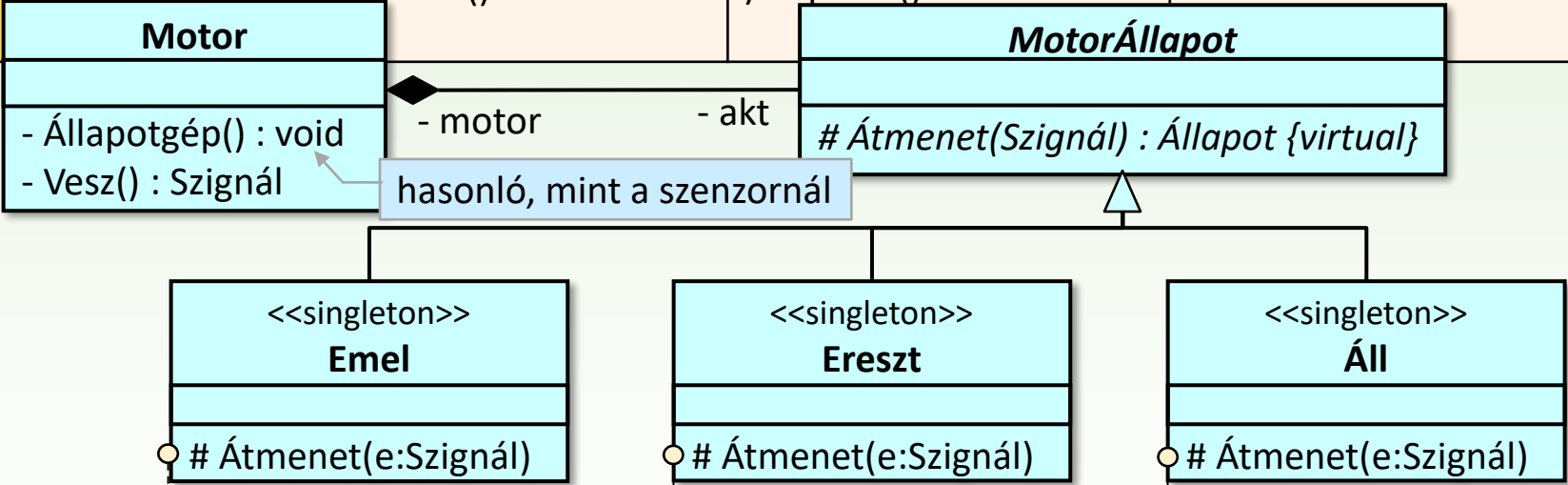


switch e
 case aktivál: return Aktív.Példány(szenzor)
 case deaktivál: return this;
 case nincs: return this;
endswitch

Motor állapotgép diagramja



állapot esemény	emel	ereszt	áll
nyit	/ kapu.Fel()	emel	emel /szenzor.Küld(aktivál)
zár	ereszt	/ kapu.Le()	ereszt /szenzor.Küld(aktivál)
állj, akadály, felnyílt, lecsukódott	áll /szenzor.Küld(deaktivál)	áll /szenzor.Küld(deaktivál)	
nincs	/ kapu.Fel()	/ kapu.Le()	



```

switch e
case nyit : kapu.Fel()
...
endswitch
  
```

```

switch e
case nyit:
return Emel.Példány(motor)
...
endswitch
  
```

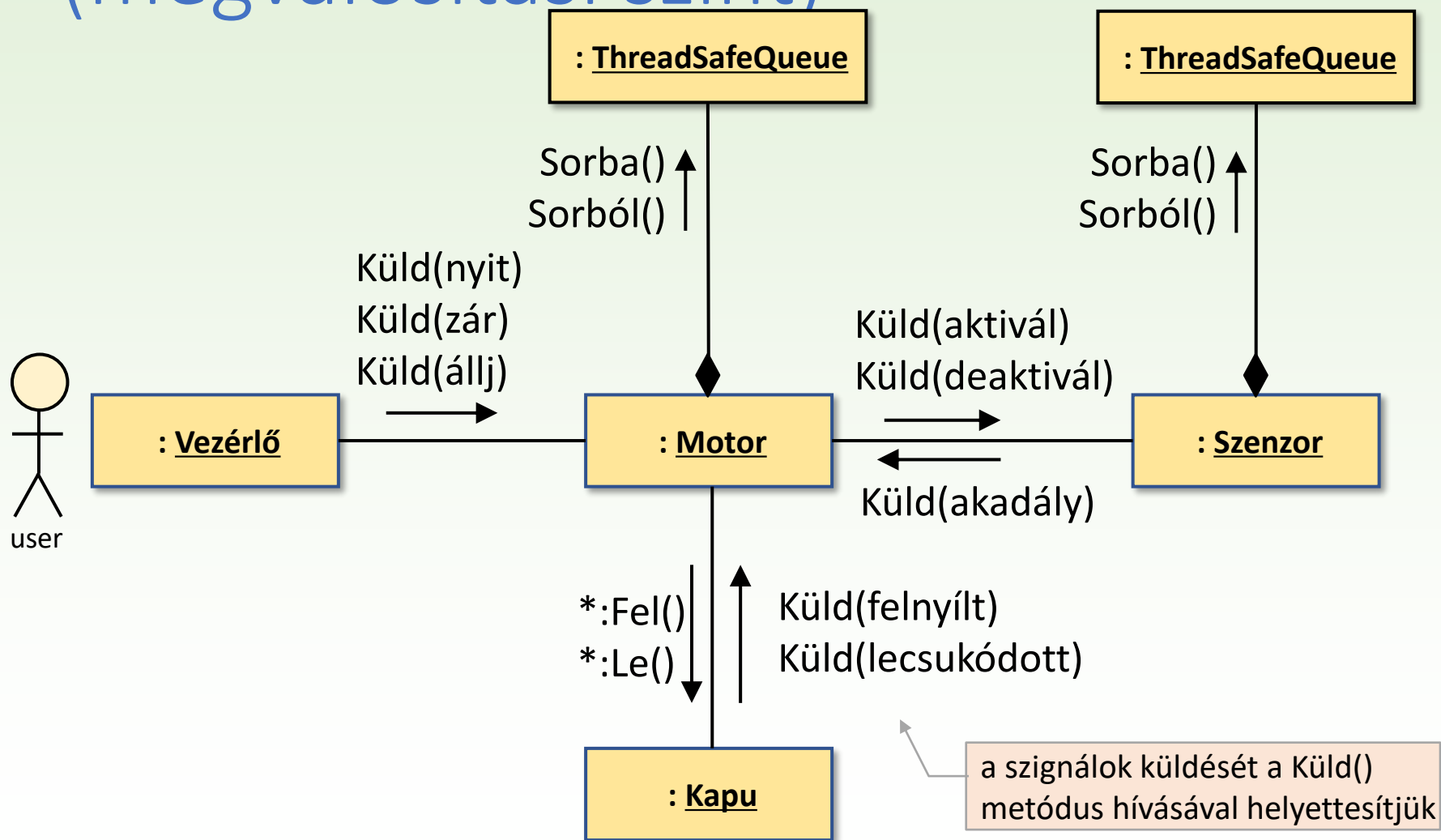
```

switch e
case nyit:
szenzor.Küld(aktivál);
return Emel.Példány(motor)
...
endswitch
  
```

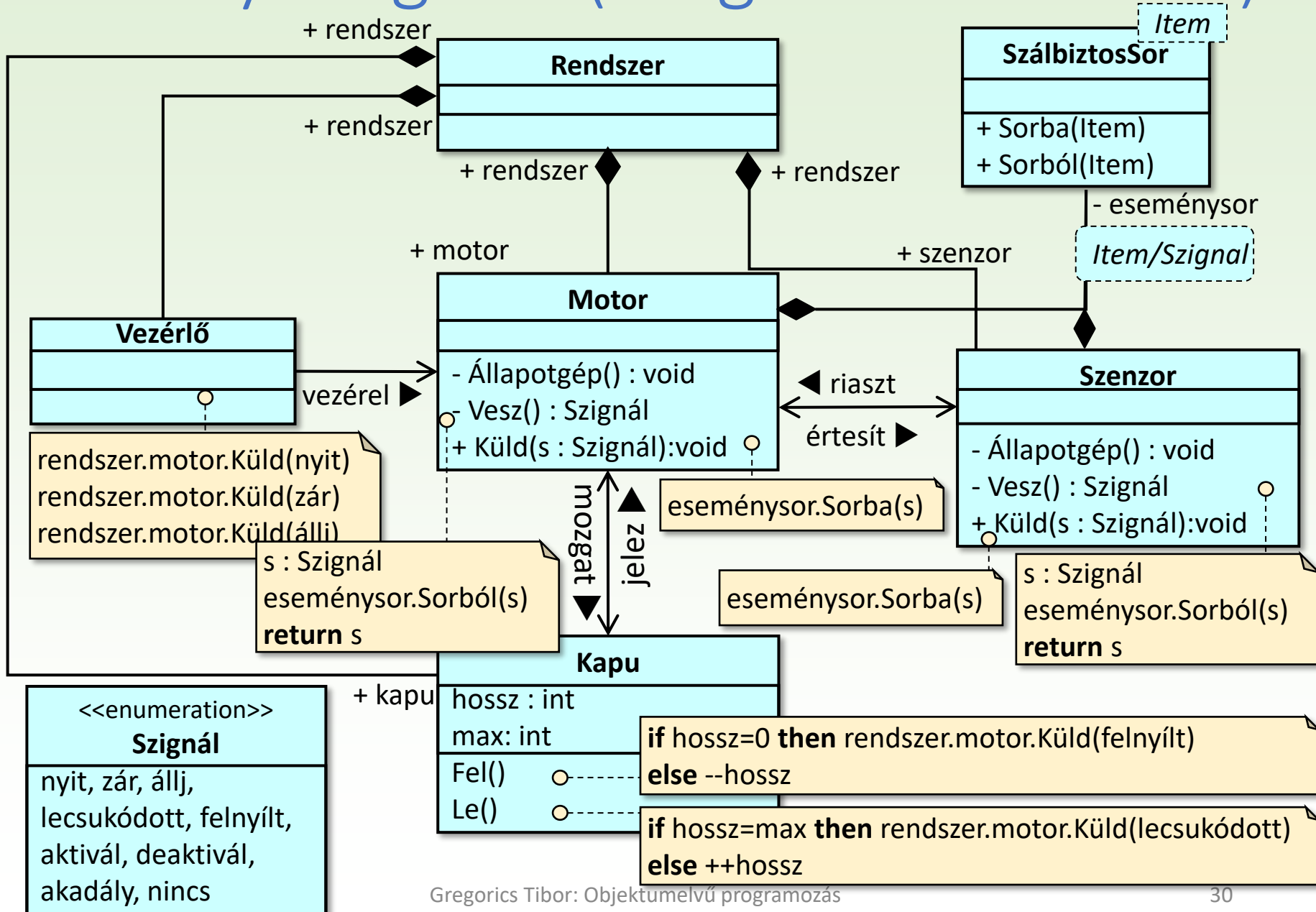
Aszinkron üzenetekkel vezérelt állapotgép megvalósítása

- ❑ Amikor az üzeneteket aszinkron módon dolgozza fel egy objektum állapotgépe, akkor azt az üzenetet küldő szálaktól **eltérő szálon** kell futtatni.
- ❑ Az objektum a hozzá több irányból érkező szignálokat egy **eseménysorban** gyűjti.
 - A küldő objektumok a fogadó objektum **Küld()** metódusának hívásával tehetnek bele szignált a fogadó eseménysorába annak **Sorba()** metódusával. Ezek a lépések a **küldő objektum szálán** hajtódnak végre.
 - A fogadó objektum **saját szálán** futó állapotgépe vesz ki egy szignált az eseménysorból annak **Sorból()** metódusával.
 - A Sorba() és Sorból() metódusokat **szinkronizálni** kell: ezek **kölcsönösen kizárásos** módon használhatják a sort, továbbá üres sor esetén a Vesz() műveletet **várakozó utasítással blokkolni** kell.

Kommunikációs diagram (megvalósítási szint)



Osztály diagram (megvalósítási szint)



Állapotgép

3.rész

Garázskapu-vezérlés megvalósítása

Gregorics Tibor

gt@inf.elte.hu

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

Rendszer osztálya (GarageGate.cs)

```
class Garagegate
{
    public readonly Engine engine;
    public readonly Sensor sensor;
    public readonly Gate gate;
    private readonly Controller controller;

    public Garagegate()
    {
        controller = new Controller(this);
        gate = new Gate(this, 5);
        sensor = new Sensor(this);
        engine = new Engine(this);
    }

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

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


Vezérlő osztály (Controller.cs)

```
public enum Signal
{
    up, down, stop,
    unrolled, coiled,
    activate, deactivate,
    blockage, none, final
};
```

két új szignál

```
class Controller
{
    private readonly Garagegate sys;
    public Controller(Garagegate s) { sys = s; }
    public void Control() { ... }
    private void MenuWrite()
    {
        Console.WriteLine("Menupoints:");
        Console.WriteLine("0 - exit");
        Console.WriteLine("1 - up");
        Console.WriteLine("2 - down");
        Console.WriteLine("3 - stop");
    }
}
```

```
public void Control()
{
    MenuWrite();
    int v;
    do
    {
        v = int.Parse(Console.ReadLine());
        switch (v)
        {
            case 0: system.sensor.Send(Signal.final);
                    system.engine.Send(Signal.final); break;
            case 1: system.engine.Send(Signal.up); break;
            case 2: system.engine.Send(Signal.down); break;
            case 3: system.engine.Send(Signal.stop); break;
        }
    } while (v != 0);
}
```

leállási üzenet annak a két komponensnek,
amelyek saját állapotgéppel rendelkeznek

szignálok küldése

Kapu (Gate.cs)

```
class Gate
{
    public readonly Garagegate system;

    private readonly int maxLength;
    public int CurrentLength { get; private set; }

    public Gate(Garagegate s, int m)
    {
        system = s; maxLength = m; CurrentLength = 0;
    }
    public void Up()
    {
        if (CurrentLength > 0) --CurrentLength;
        if (0 == CurrentLength) system.engine.Send(Signal.coiled);
    }
    public void Down()
    {
        if (CurrentLength < maxLength) ++CurrentLength;
        if (maxLength == CurrentLength) system.engine.Send(Signal.unrolled);
    }
}
```

Szálbiztos eseménysor (MvQueue.cs)

Egy állapotgéppel rendelkező objektum egy eseménysorban gyűjti a neki küldött szignálokat, hogy azokat a beérkezésük sorrendjében feldolgozza.

```
class MyQueue <Signal>
{
    private readonly Queue<Signal> queue = new ();
    private readonly object criticalSection = new ();
    private readonly Signal noneSignal;
    public MyQueue(Signal none) { noneSignal = none; }

    public bool Empty() { return queue.Count == 0; }
    public void Enqueue(Item e) { ... }
    public void Dequeue(out Item e) { ... }
}
```

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

összetartozó (egymást kölcsönösen
kizáró) szakaszok azonosítására

Mivel az Enqueue() és Dequeue() külön szálakon fut, hibát okozhat, ha egyszerre akarnak művelet végezni a soron. Ezért szinkronizálni kell őket: a **kölcsönös kizárás** elvén kell működniük.

```
public void Enqueue(Item e)
{
    Monitor.Enter(criticalSection);
    queue.Enqueue(e);
    Monitor.Exit(criticalSection);
}
```

A **Monitor** osztály Enter() és Exit() metódushívásai – amelyek paramétere egy tetszőleges, de ugyanaz az objektum – a jelölik ki azokat a külön szálakon futó **kritikus szakaszokat**, amelyek közül egyszerre csak legfeljebb csak egy lehet aktív.

```
public Signal Dequeue()
{
    Signal e;
    Monitor.Enter(criticalSection);
    if (!Empty())
    {
        e = queue.Peek();
        queue.Dequeue();
    }
    else e = noneSignal;
    Monitor.Exit(criticalSection);
    return e;
}
```

Állapotgép (StateMachine.cs)

```
public abstract class StateMachine<Signal>
{
    private readonly Thread thread;
    private readonly MyQueue<Signal> eventQueue;
    private readonly Signal finalSignal;

    public StateMachine(Signal none, Signal final)
    {
        finalSignal = final;
        eventQueue = new MyQueue<Signal>(none);
        thread = new Thread(new ThreadStart(StateMachineProcess));
        thread.Start();
    }

    public void Send(Signal signal) { eventQueue.Enqueue(signal); }
    private bool inprocess;
    private void StateMachineProcess()
    {
        inprocess = true;
        while (inprocess)
        {
            Signal signal = eventQueue.Dequeue();
            if (signal.Equals(finalSignal)) inprocess = false;
            else Transition(signal);
        }
    }

    protected abstract void Transition(Signal signal);
}
```

külön szál a szignálok feldolgozásának

az eseményeket fogadó szálbiztos sor

speciális szignálok

az állapotgép a final szignál megérkezéséig működik

sablonfüggvény tervezési minta

Szenzor (Sensor.cs)

```
class Sensor : StateMachine<Signal>
{
    public readonly Garagegate system;

    private SensorState currentState;

    public string CurrentState { get { return currentState.ToString(); } }

    public Sensor(Garagegate sys) : base(Signal.none, Signal.final)
    {
        system = sys;
        currentState = ... ;
    }

    protected override void Transition(Signal signal) { ... }

    private readonly Random rand = new();
    private void Observe()
    {
        if (rand.Next() % 100 < 1)
        {
            system.engine.Send(Signal.blockage);
        }
    }
}
```

speciális szignálok

a hiányzó részek a `SensorState`
definiálásától függenek

Szenzor állapotok felsorolásával

```
protected override void Transition(Signal signal)
{
    switch (currentState)
    {
        case SensorState.passive:
            switch (signal)
            {
                case Signal.activate: currentState = SensorState.active; break;
                case Signal.deactivate: case Signal.none: break;
                default: throw new UnknownSignalException();
            }
            break;
        case SensorState.active:
            switch (signal)
            {
                case Signal.activate: break;
                case Signal.deactivate: currentState = SensorState.passive; break;
                case Signal.none: Observe(); break;
                default: throw new UnknownSignalException();
            }
            break;
    }
}
```

a `SensorState` egy felsorolás

```
enum SensorState { active, passive };

private SensorState currentState;
```

```
public Sensor(Garagegate sys)
    : base(Signal.none, Signal.final)
{
    system = sys;
    currentState = SensorState.passive;
}
```

Szenzor állapot tervezési mintával

```
abstract class SensorState
```

```
{
```

```
    protected Sensor sensor;
```

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

```
    public virtual SensorState Transition(Signal signal) { return this; }
```

```
}
```

```
class Passive : SensorState { ... }
```

```
class Active : SensorState { ... }
```

a `SensorState` egy ősosztály



```
public Sensor(Garagegate sys) : base(Signal.none, Signal.final)
```

```
{
```

```
    system = sys;
```

```
    currentState = Passive.Instance(this);
```

```
}
```

```
protected override void Transition(Signal signal)
```

```
{
```

```
    currentState = currentState.Transition(signal);
```

```
    Thread.Sleep(100);
```

```
}
```

Szenzor állapot tervezési mintával

```
class Passive : SensorState
```

```
{
    private static Passive instance = null;
    private Passive(Sensor s) : base(s) { }
    public static Passive Instance(Sensor s)
    {
        if (instance == null)
            return new Passive(s);
    }
    public override void Transition(Signal signal)
    {
        switch (signal)
        {
            case Signal.activate: return this;
            case Signal.deactivate: return Passive.Instance(sensor);
            case Signal.none: Observe(); return this;
            default: throw new UnknownSignalException();
        }
    }
}
```

egyke tervezési minta

```
class Active : SensorState
```

```
{
    private static Active instance = null;
    private Active(Sensor s) : base(s) { }
    public static Active Instance(Sensor s)
    {
        if (instance == null) instance = new Active(s);
        return instance;
    }
    public override SensorState Transition(Signal signal)
    {
        switch (signal)
        {
            case Signal.activate: return this;
            case Signal.deactivate: return Passive.Instance(sensor);
            case Signal.none: Observe(); return this;
            default: throw new UnknownSignalException();
        }
    }
}
```


Motor (Engine.cs)

```
class Engine : StateMachine<Signal>
{
    public readonly Garagegate system;

    private EngineState currentState;

    public string CurrentState { get { return currentState.ToString(); } }

    public Engine(Garagegate sys) : base(Signal.none, Signal.final)
    {
        system = sys;
        currentState = ... ;
    }

    protected override void Transition(Signal signal) { ... }
}
```

speciális szignálok

a hiányzó részek az EngineState
definiálásától függenek