A vizsga szóbeli, a tételsor 2009-ben az alábbi:

1. Technológia, komplex szoftverrendszerek alapproblémái, szoftverminőség, fejlesztési folyamat és kézbentartása. Diszkrét rendszerek leírásának nehézségei, hibakezelés, tesztelés problémái.

A technológia: műveletek sorozata, amely egy adott kezdeti állapotból adott erőforrások felhasználásával megismételhető módon egy kívánt végállapot eléréséhez vezet.

Erőforrások: információ, anyagok, idő, munkaerő, szakértelem, eszközök → pénz (többé kevésbé lehet konvertálni)

Beágyazott szoftver (firmware): sok szempontból eltér a többi SW területtől:

- Szoros információs kapcsolatban áll a külvilággal
  - Humán interfész: beágyazott SW-rendszerekben sokkal változatosabb
  - Kommunikációs interfész (hálózat)
  - Érzékelés és beavatkozás a fizikai környezetbe
- Biztonságkitikus környezet (járműipar, energetika stb)
- Alacsony felhasználói hibatolerancia
- Kívülről látszólag egyszerű működés jellemző rájuk
- Szoftver és annak környezete stabil
- Árérzékeny: maga az eszköz olcsó, a hiba javítása drága (visszahívás)
- Eltérő fejlesztési módszerek

Hogyan állunk technológiával SW-es területen?

- A SW terület viszonylag új
- Nagyon sokszínű
- Nagyon flexibilis → **látszólag** a hibás döntés kisebb kárt okoz v nem okoz kárt

SW egyre fontosabb (kell a technológia)

- Egyre több és kritikusabb feladatot bízunk SW-re
- A népesség egyre nagyobb része lesz programozó, de a képességek adottak
  →"any idiot can write code"

- Képesség és motiváció: kb. 6 hónap alatt egy adott területen profi lehet

Komplex szoftverrendszerek alapproblémái, szoftverminőség:

- Komplex rendszerek, melyek többnyire hierarchikus felépítésűek
  Rendszer – alrendszer .. alrdsz – elemi komponens → erre kell felbontani a problémát
  - A világ hierarchikus viselkedése természetes emberi viselkedés
- Az elemi komponensek halmaza absztrakciófüggő: nézőpont és cél kérdése az elemi komponens pl.: adatbázisnál: a használó rekordokat, a programozó I/O műveletet lát

- A komponensen (alrendszeren) belüli kapcsolatok erősebbek, mint az azonos szinten levő komponensek közti kapcsolat. Pl.: tanszék és tanszék között kisebb, mint tanszéken belül → az alrendszerek felismerését segíti, a kapcsolatok erőssége
- A komplex rendszerek többnyire néhány alapvető komponens változatos kapcsolatából épül fel → ezért komplex pl.: számítógépes hálózat, internet (3 komponens: végeszköz, hálózati eszközök /router/, hálózati kapcsolatok)
- A komplex rendszerek többnyire nem előzmények nélkül jelennek meg, hanem egyre növekvő komplexitású elődökön keresztül fejlődik /evolúció/
    - 0-ról fejlesztett komplex rendszerek többnyire működésképtelenek (evolution not revolution)
    - be kell járni a fejlődési fokokat → inkrementális fejlesztés: képességben, méretben, teljesítményben, időben

Komplex SW rendszerekkel kapcsolatos problémák többnyire

Okok:

A, A megoldandó probléma bonyolultsága

B, A fejlesztési folyamat (process) kézbentartásának nehézségei

C, A SW eszközök rugalmassága

D, a diszkrét idejű rendszerek viselkedésének leírási nehézségei

A, A megoldandó probléma bonyolultsága

- felhasználó / megrendelő megad egy követelményt (requirements)
    - nagy számú
    - bonyolult
    - ellentmondóak
    - inkonzisztencia: követelmények következményei paradoxont okoznak
    - nehezen számszerüsíthető és mérhető

- megrendelő és megvalósító közti ellentét
    - megrendelő saját szakterületének nyelvén fogalmazza meg követelményeket
    - a megrendelő maga sem tudja mit szeretne, menet közben iterálnak
- a megvalósító programban gondolkodik, a számítógépek világában él
  → befolyásolják korábbi tapasztalatai

→ az a jó, ha a megrendelő és megvalósító együtt választják ki a specifikációt

Specifikáció használata kész program esetén:

- javítás/karbantartás: hiba (bug) specifikációtól eltérő működés → ingyen javítás
- fejlesztés (feature update): új, spec-ben nem megadott funkció → pénzért csináljuk
- konzerválás: SW környezete változik meg, ami a változatlan SW mellett hibákat okoz (Win XP-n is menjen) → pénzt kérünk érte

B, A fejlesztési folyamat (process) kézbentartásának nehézségei

- hosszú, bonyolult programok írása csak a kezdőket villanyozza fel
- a komplexitás és a fejlesztési folyamat hierarchikus felépítése kézben tartható
  - csak a feltétlenül szükséges részleteket kell megismerni
  - alrendszerek közti interfészek fontosak
- egyszerűségre, rövidségre törekvés (KISS: Keep It Simple Stupid)
- előre gyártott elemek használata /elemi komponens/
  → SW komponensek, példaprogik, minta (pattern): tervezési v. architektúrális minta

- modularizált, újrafeldolgozható elemekből való építkezés: feladatok szétosztása fejlesztők között
- inkrementális fejlesztés
  - kritikus részekkel való ismerkedés
  - funkcióiban inkrementális fejlesztés és tesztelés: unit teszt
  - aspektus orientált programozás: funkcionális és security: emberi hibák csökkentése

C. Szoftvereszközök rugalmassága

- A megfelelő absztrakciós szint megtalálása → feladatot a megfelelő eszközökkel
- A legalacsonyabb absztrakciós szint a gépi utasítás
- Programozási nyelv, környezet eltérő absztrakciós szintet tesz lehetővé
- Magasabb absztrakciós szintek <u>többnyire</u> magasabb futási idejű erőforrásigénnyel járnak
  - Nem csak a futási idejű erőforrás-felhasználás az érdekes /Labview ←→ C/
  - Fejlesztési erőforrások és a SW minőség is előtérbe kerülhet

D. Diszkrét idejű rendszerek leírásának nehézségei

SW:

- Állapottere:
  - Adatelemek + inputok
  - N bit
- Műveletek /állapotátmenetek/: az állapottérben milyen műveletek lehetnek
  - Pl.: 1. fokú szűrő, 32 bit belső áll, 16 bit bemenet → 2^16*2^32 állapot
  - 2. fokú szűrő → 2^16*(2^32) ^2 lehetséges állapot
  - → az állapottér óriási és a rajta végezhető műveletek bonyolultak

- →állapotszám nő → műveletek száma exponenciálisan nő

Tesztelés, Hibakezelés:

A programok működésének megértése és tesztelése nagyon nehéz

- Az összes lehetséges állapotra az összes lehetséges állapotátmenetet tesztelni kell
- A kérdés a tesztlefedettség

Beágyazott rendszerek:

- A beágyazó környezet modellje nehezen előállítható: nem csak a jó működés, hanem a rossz működés modelljével is rendelkeznünk kell
- A kód nagyon nagy része hibakezelés /70-90%/: hibakezelő kód tesztelése nehéz, mert a hibák előállítása nehéz, vagy lehetetlen /CPU jó-e, M jó-e/

→ E.W.Dijkstra /E.W.D/: „On the cruently of really teaching computer science”

→ nehéz a SW-s dolgok oktatása

SW-s fejlesztés: SW életciklus modell /vízesés modell/:

- Elemzés /MIT?/
- Tervezés /HOGYAN?/
- Implementálás
- Üzemeltetés

→ Intuíció*(mikor az informacioszerzes nem a gondolkodas es kovetkeztetes utjan tortenik*) elkerülése a folyamatban /persze elemzés tervezés során kell/

A sorrend fejlesztéskor: Üzemeltetés → implementáció → tervezés → elemzés

????

Az implementáció absztrakciós szintjének kiválasztása:

2. Programozási paradigmák, procedurális és deklaratív megközelítés, procedurális nyelvek fejlődése, strukturáltság megjelenése.

Az implementáció absztrakciós szintjének kiválasztása:

Programozási paradigmák:

- A programozás alapvető módja, stílusa, program belsejének definiálása
- Egy időben, egy programozási felületben akár több paradigmát használhatnak /ugyanazt a nyelvet használva/

Paradigma:

- **Procedurális programozás** /imperatív programozás/: → HOW?
  - nagyobb problémát részfeladatokra bontunk → elemi utasítások: függvénvek függvényeket hívnak
  - Matematikai függvény
  - A probléma megoldásához szükséges lépések megadása a feladat
  - Goto száműzése
  - Modularitás, strukturáltság, kód újrafelhasználás: /skalár, vektor, mátrix/
  - CPU nagyon mélyen is ezt támogatja /imperatív/: megmondjuk a CPU-nak a lépéseket
  - Objektum orientált programozás is ennek a kiterjesztése

- **Deklaratív programozás**: nem kell megadnunk hogyan csak azt hogy mit → WHAT?
  - Formális logikát és a következtetéseket használja
  - A feladat megoldásának logikáját adjuk meg, a megoldás részletei (vezérlési szerkezetek – control flow) nélkül
  - Pl.: bíró eldönti egy lépés szabálytalan-e, C-compiler, adatbáziskezelők motorja
  - Szabályokat értelmezi és végrehajtja őket a procedurálisan megírt motor

Kiegészítő paradigmák:

1. párhuzamos programozás
2. eseményvezérelt programozás

Procedurális / imperatív programozási nyelvek fejlődése:

0. generáció: bináris kód: gépi utasítások bináris bevitele

1. generáció /54 - 58/: matematikai kifejezések kezelése Fortran I, Algol 58, gépi típusok használhatók
2. generáció /59 - 61/: adattípus eljárás fv fogalma: Fortran II, Algol 60
3. generáció /62 - 70/: 2. generációs nyelvek finomítása, összetett adattípusok: Algol 68, Pascal

Generációs rés: PDP-11: könnyebb számítógép hozzáférés, evolúciós sokszínüség, C nyelv, UNIX OS

4. generáció: objektum orientált nyelvek: Smalltalk, C++, Java, C#, deklaratív nyelvek: SQL, Prolog

0: nincs strktúra

1-2: adattípus + műveletek

3: modularitás

4: adat és rajta végezhető műveletek szoros egymásbakapcsolása

3. Objektumorientált programozás, osztály és objektum viszonya, objektum típusa, az információ rejtés megvalósulása objektumorientált programozás során, az objektumorientáltság ismérvei. Java példák.

OOP: Object Oriented Programming

A programozási feladat megoldása olyan módon, hogy a létrejövő **OOP** együttműködő objektumok összessége. (maga is egy objektum)

Tipikus emberi gondolkodás, a világról alkotott képünk objektum orientált.

- **Objektum** = tulajdonságai + rajta végezhető műveletek
- objektumok **tartalmazhatnak** objektumokat (tulajdonság is lehet objektum)
- objektumok **hivatkozhatnak** egymásra, és azok is másra
  - referencia ≠ pointer → memóriaterületre mutató hivatkozás

Egyes programozási nyelvekben a pointer a referencia alkalmazható megvalósítása.

(C/C++, és közös memória)

JAVA: referenciához nem csatolható memória terület
- pl.: JAVA referencia, URL

- Az objektumok lehetnek hasonlóak
  - általánosítás és specializáció fogalma az objektumok között

(emberek / foglalkozások)

absztrakt és konkrét → ide szeretnénk eljutni

Az objektum SW szempontból:

- a program alapköve
  - független összetevők
  - (lehetőleg) rejtett belső állapota van (tulajdonság)
  - saját viselkedésük, működésük van
  - külső objektumok hatnak rájuk (belső állapot + változók)
- Az objektumokat egy vagy több osztályba tartozónak tekintjük

osztály (class): absztrakt adattípus, melynek példánya (egyede, instance) az **objektum**

tervezési idejű fogalom — futási idejű fogalom

Az objektumnak lesz élettartama is!

Definiáljuk az osztályt:
- megadjuk az objektum állapotát leíró adatszerkezetet – tulajdonságok listája
- megadjuk az objektumon végezhető alapműveleteket – metódusok

Az objektumorientáltság 3 fő vonása (ismérvei):

- (1) információ rejtés / összefoglalás
  encapsulation vagy information hiding
- (2) öröklődés (inheritace)
  ⇨ az osztályból létrehozunk újabb, speciálisabb osztályokat
- (3) poliformizmus (poliformism)
  rajzelemek mozgatása rajzelem – pont
  vonal
  stb.
  rajzelemekből álló listán mindegyik move(x,y) metódusa, megoldható.

```
pl. 2D rajzelemek osztály
               - move (dx, dy) metódus
               - származtatva: pont, vonal, kör : ugyanez a metódus (move)
meglesz ezekkel a paraméterekkel
               - legyen egy lista: rajzelem objektumokat tartalmazó lista
(illetve az ezekre mutató pointereket): pont, pont, vessző, kör
               - ha egy iterációval végigmegyünk ez a listán, akkor mindig az
objektumnak meghívódik a move metódusa

- a beágyazott C++ nyelvekben nincs polimorfizmus, mert futásidejű
következményei lehetnek

- a C egy nagyon veszélyes nyelv :)
```

(1) információ rejtés: - *KIEGÉSZÍTENI A TIPUSOS NYELVEKKEL.*

- a másik az objektumot használó programozó elől
- a komplexitás csökkentése
- interfészt definiálok public, protected, private ⟶ senki
  a metódusra a hozzáférés megadható (snope), friend módosító: a protectedet teszi elérhetővé
- a modern programozási nyelvek kivétel nélkül típusosak
  adattípus: - értékkészlet ⟶ lehetséges értékek
  - jelölés ⟶ hivatkozás szintaxisa
  - műveletek ⟶ típuson belül, v. más típusokba vezető konverzió
- fordítóprogram:
  - tárterület foglalás: - explicit típusdeklaráció (C/C++, JAVA)
    - dinamikus típus konverzió PHP
  - ellenőrzés

  szigorúan típusos nyelvek *TP?* ⟺ gyenge ellenőrzés PHP

  ⇓

  C/C++/JAVA automatikus konverzió

  Az objektum típusa az osztály + minden felette álló!!!

  ⇨ többféle típusunk van

- A programozó szempontjából is fontos az osztály és a típus
  magyar notáció ⟶ a változó nevében a típus legyen benne

JAVA PÉLDÁK

4.Objektum állapota, viselkedése, azonosság, értékadás, élettartam, interfész, objektumok közötti kapcsolatok, láthatóság, az objektumok közötti kommunikáció. Sablonok. Java példák. (élettartam után objektum tárolási módok)

Az objektum rendelkezik:

- Belső állapot /tulajdonságok, attributumok/
- Viselkedés /metódus,tagfüggvény/
- Élettertam
- Interfész

1. Belső állapot:

- Típusos osztály- és egyedváltozók
  - **Osztályváltozó:** az adott osztályhoz tartozó összes egyedben ugyanaz a memóriaterület van jelen
    - static a Java-ban: nem egy objektumhoz tartoznak, hanem az egész osztályra közösek (pl.: dolgozók osztályában a nyugdíjkorhatár)
    - osztály globális változó: pl.: adott osztályba tartozó elemek megszámlálása → konstans, ami idő közben változik
  - **Egyedváltozó:**
    - minden egyedben külön memóriaterület van jelen (egyedspecifikus)
    - jellegzetesen ezt használjuk
    - azonosságvizsgálat:
      - név szerinti (ahogy hívom): ritkán támogatott
      - referencia szerint „==” operátor: ugyanoda mutatnak-e
      - érték szerinti azonosságvizsgálat „===” operátor valahol: a két objektum tulajdonságai azonosak-e
      - Mi van ha:
        - A tulajdonság maga is objektum
        - A tulajdonság maga is referencia → nem egyértelmű a művelet (kereszthivatkozások), láncolt listánál a programozó dönti el, milyen mélységig vizsgálunk → polimorfizmus kell
      - Compare (obj) polimorfizmus
      - Az érték szerinti azonosság művelet nem egyértelmű

2. Viselkedés:

- Típusos osztály és egyedmetódusok /módszer, tagfüggvények/
- Osztálymódszer:
  - „Static” a JAVA-ban
  - Csak osztályváltozókat és más osztálymetódusokat használhat (nem használjuk gyakran)
- Egyedmódszer: egyed-és osztályváltozókat használ. Meg van határozva, hogy a metódus melyik adatterületen fut le.
  - overloading: többértelműség (ugyanolyan nevű metódus)
  - a módszereket a „nyoma” azonosítja
  - a metódus nyoma = a visszatérési érték típusa + metódus neve + paraméter típusa

3. Élettartam:

- az objektumokat egy speciális „konstruktor” nevű metódussal hozzuk létre. Fajtái:
    - default konstruktor: nincs paramétere
    - copy konstruktor: paraméterének típusa azonos a létrehozandó adatok típusával (deep copy)
    - egyéb paraméterezett konstruktorok
- fontos kérdés, hogy mi történik a szülőosztályok konstruktoraival, lefutnak-e maguktól, meg kell hívni őket, mi történik többszörös öröklődés esetén
- objektumok megszüntetése:
    - explicit felszabadítás: C++
        - destruktor
        - free utasítás hatására kerül meghívásra
        - hatékony, egyszerű
        - nagyon veszélyes, sok hibalehetőség van
        - túl korán, vagy nem jó objektumot szabadítunk fel
        - elfelejtettünk felszabadítani valamit (memory leak)
    - automatikus szemétgyűjtés: JAVA
        - a memória automatikus felszabadítása a nem használt objektumok esetén → az összes rá történő hivatkozás megszűnik
        - reference count nyilvántartása
        - erőforrás igényes, valósidejű környezetben az alkalmazása megkérdőjelezhető
        - finalize() metódus (JAVA): le lehet zárni dolgokat, mielőtt az objektum megszűnne

megszüntetés beágyazott és biztonságkritikus rendszerekben:

- ha lehet, egyiket se használjuk (tervezési időben megadható a memória felhasználás)
- explicit megoldás kimerítő technikát igényel
- automatikus szemétgyűjtés még nyitott kutatási terület

Az objektumok sok esetben elválhatnak az őket létrehozó programoktól:

- objektumok perzisztens tárolása: mentés / visszaállítás / adatbázis
- objektum továbbítása: copy / paste, hálózati másolás, elérés (architektúrális különbségek)

Tranziens objektum: a program elindulásától a megszűnéséig él

Perzisztens objektum: túlélheti az őt létrehozó programot vagy átkerülhet más programba.

- verziókövetés, platform-függetlenség, típus helyes helyreállítás (új word, régi word)

Objektum Tárolási módok:

- objektum-orientált adatbáziskezelő
- relációs adatbáziskezelő
    - tábla = osztály (adott verziószámú)
    - oszlop = tulajdonság (típusos)
    - sor (rekord) = egyed
    - referencia = idegen kulcs /másik táblában lévő kulcs/ (másik táblában adott sorra való hivatkozás)
- XML → lásd szoftvertervezés vagy web: definiálunk egy nyelvet, amiben el tudjuk menteni az objektumot (XML, XSLT, XPATH)
- bináris kódolás: Binary Encody Rube (BER)

Felhasználás:

- serialize, deserialize interfész: soros byte-streammé lehet alakítani meg vissza az objektumot
    - JAVA: implements serializable
    - C++: többszörös öröklléssel érik el, serializable absztrakt osztályból örökléssel
- convert to / from XML() interfész (saját magamnak kell megírni)

4. Interfész: objektum rendelkezzen metódussal és működéssel → rákényszerítem a programozót ezen interfész megvalósítására. Pl.: objektumot fájlban el akarok menteni → serialize rendesen letárolja.

- az objektumból létrehozott egyed egy vagy több jól definiált interfészt nyújt az őt használni kívánó objektumok felé
- hozzáférést beállíthatjuk: public, protected, private
- kérdés: egy osztályra hogyan tudunk „rákényszeríteni" egy interfészt? pl.:
    - C++: többszörös öröklődés, az interfész megadása absztrakt osztályként (direktbe nem lehet példányosítani)
    - JAVA (csak egyszeres öröklődés van): interfészek definiálása: impements "interface" → jobban szét van választva az interfész és az osztály, az ezután írt interfészt az osztálynak meg kell valósítani
        - csak publikus absztrakt fv-eket tartalmaz
        - nem lehet példányosítani

Objektumok / osztályok közötti kapcsolatok:

- komplex probléma hierarchikus dekompozíciója
- együttműködő objektumként képzeljük el a megoldást

Kapcsolatok:

- **1. tartalmazás** (aggregation): → rész egész (kutya - lába)

- **2. társítás** (association): → laza kapcsolat (kutya – harap - postás)
- **3. öröklés** (inheritance): → általánosítás / specializáció

3. Öröklés:

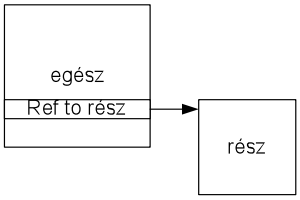
- általánosítás / specializáció
- osztályok között értelmezhető kapcsolat következményekkel az egyedre
- A örököl B-től:
    - A alosztálya B-nek (descendant) → leszármazott
    - B szülőosztálya A-nak (superclass)
    - hierarchia tetején a base ül (szűznemzéssel jön létre)
- öröklési hierarchia: többnyire gráf vagy fa, ha egyszeres öröklés van
    - többnyire az alsó legspeciálisabb osztályból hozunk létre egyedeket, vagy ezeket specializáljuk tovább (ezekből származtatunk le új osztályokat)
    - absztrakt osztály: nem tudunk egyedeket létrehozni: ha ebből leszármaztatok osztályt, csak akkor tudok egyedeket létrehozni
- alosztály örökli a szülőosztály tulajdonságait és viselkedését
- az alosztály elfedheti (felüldefiniálhatja) a szülőosztály tulajdonságait és viselkedését
    - többnyire van lehetőség a szülő tulajdonságainak és viselkedésének elérésére
    - egy adott osztálynak több azonos nevű metódusa lehet (overloading)
    - metódus nyomának kell különböznie
    - operátor overloading: értlemezhetőség pl.: vektor→mátrix, de ember??
- egyszeres vagy többszörös öröklődés: egyszeres (C++: oroszlán macskaféle és ragadozó is), többszörös (JAVA: öröklődési hierarchia fa)

Sablonok:

- tipikus probléma: hasonló viselkedés (pl.: FIFO), különböző típusú osztályokra. Pl.: int-et kezelő FIFO és adott (de tetszőleges) osztályt kezelő FIFO
- sablon→Osztály1, compiler→Osztály2: Osztály1 kezelésére→ egyed /osztály 2 ből/
- pl.: FIFO sablon → int → FIFO osztály int kezelésére → FIFO objektum int tárolására
- pl.: tárolási osztályok (container class)
    - C++: standard template library
    - JAVA: Generics

1. Összetétel, tartalmazás (aggregation):

- irányított reláció, az egész tartalmazza a részt
- érték vagy referencia szerint lehet csoportosítani:
    - érték: fizikailag egymásba ágyazott tárterület (Memória)
        - együtt jönnek létre és szűnnek meg
    - referencia szerint: egész egy referenciát tartalmaz a részre



- külön életet él a rész és az egész, de az egész kezeli a részt

2.Társítás (association): kutya és a postás

- 2 irányú összeköttetés: milyen postás
- szigorúan nem tartalmazás: az objektumok külön életet élnek
- tipikusan referenciákkal valósítjuk meg
- számosság: jellegzetes:
    - egy – egy: férj – feleség
    - egy – több: szultán
    - több – több: orgia

5.Párhuzamos eseményvezérelt rendszerek, alapfogalmak, ütemező és feladatai, ütemezési algoritmusok. Ütemezés hardware és software megvalósítása. Java példák. 04.01.text fajl

Klasszikus programozás (szekvenciális program):

- szekvenciálisan (egymás után végrehajtott) utasítások sorozata.
- a program utasításainak végrehajtási sorrendjét a vezérlési szerkezetek önmagukban definiálják
- a program nem mindig csinálja ugyanazt, az adatoktól is függ a működése
  - sok probléma nehezen oldható meg így
  - az erőforrásokon osztozni kell, az erőforrásokkal gazdálkodni kell
  - nagyon gyakran valójában párhuzamosan végrehajtandó feladataink vannak

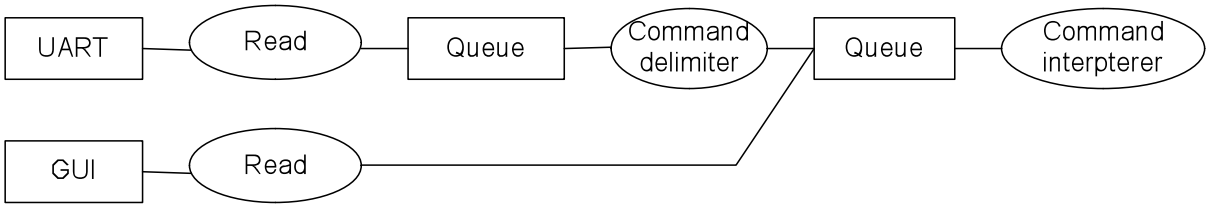
A világ párhuzamos → eleve sok feladat párhuzamos. Ez részfeladatok formájában kerül megfogalmazásra, és nehezen fogalmazható meg szekvenciális módon.

Alapfogalmak:

- **szekvenciális program** / **rendszer** → tudjuk mi ez
- **párhuzamos** / **konkurens program** / **rendszer**: olyan rendszer vagy program, amelyben több, egymással együttműködő részfeladat egészeként valósítjuk meg a teljes feladatot.
-

Feladat / részfeladat: Task / subtask:

A párhuzamos rendszerben egymással párhuzamosan futó, önmagában szekvenciális programrészleteket, amelyek egymás nélkül többnyire értelmes működésre nem képesek, részfeladatoknak nevezzük.



A részfeladatokhoz végrehajtó egységet kell rendelnünk:

N részfeladat ⟶ 1 végrehajtó egység

N részfeladat ⟶ M végrehajtó egység (Dual Core ☺)

Vagyis a szekvenciális részfeladatok együtt oldják meg a teljes feladatot, eközben kommunikálnak egymással és befolyásolják egymás működését, azaz együttműködnek.

- **eseményvezéreltség (event driven):** A program lefutása függ a program működése során lezajló történésektől (event).

- **<u>esemény / jelzés, event / signal:</u>** ezekre az eseményekre reagál a rendszer ⟶ megváltozik a működés ⟶ azonos információs kapcsolat.
  (fontos a válaszidő! (RT rendszerek))

  - **Külső esemény:** pl. user input, (A/D) vagy kommunikációs interfészen küldés / fogadás
  - **Belső esemény:** A program belső állapotában lezajló változások, explicit eseménykiküldés (send_event())
  - Az esemény többnyire egy összetett adattípusként jelentkezik a SW-en belül (struct / object)

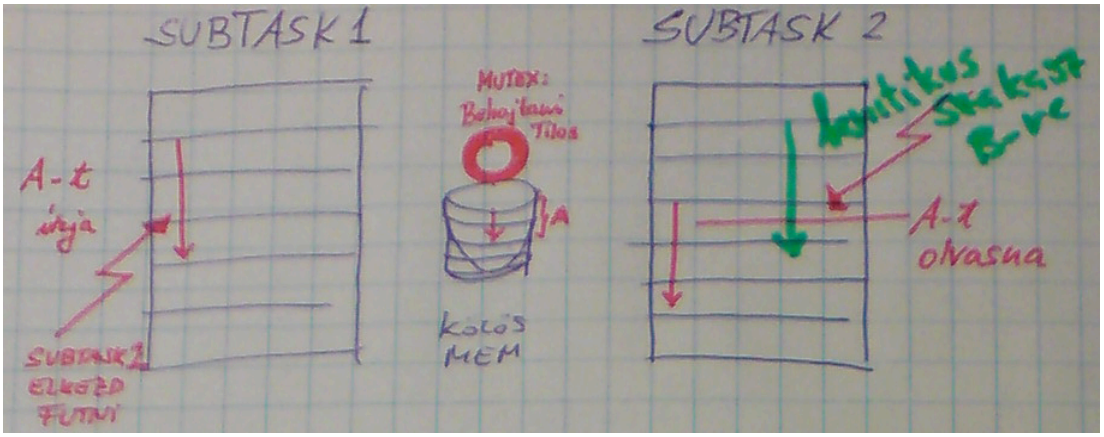
<u>Erőforrás:</u> minden olyan eszköz, amire a részfeladatnak futás közben szüksége van. A legfontosabb: CPU. Kell még: memória és annak tartalma, illetve különböző perifériák.

<u>Közös erőforrás (shared resource):</u> Egy időben több párhuzamosan futó részfeladatnak is szüksége van rá. Az ilyen erőforráson osztoznak. Többnyire egy, vagy maximum megadott számú részfeladat tudja helyesen használni.

Egy felhasználó: printer, soros port, bus, MEM írása

Több felhasználó: SCSI, SATA, HDD-nél NCQ

A rendszertervezőnek és a programozónak a legfontosabb feladata, hogy felismerje a közös erőforrásokat, és biztosítsa azoknak a helyes használatát.



⇩

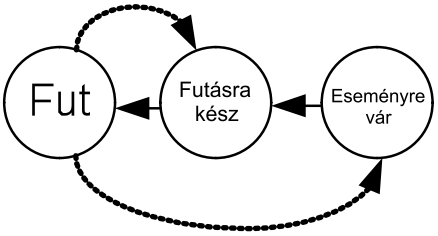
Inkonzisztens állapot lesz, mert a memóriát csak félig írtuk tele. Subtask2 rosszat olvas, rosszul fog működni.

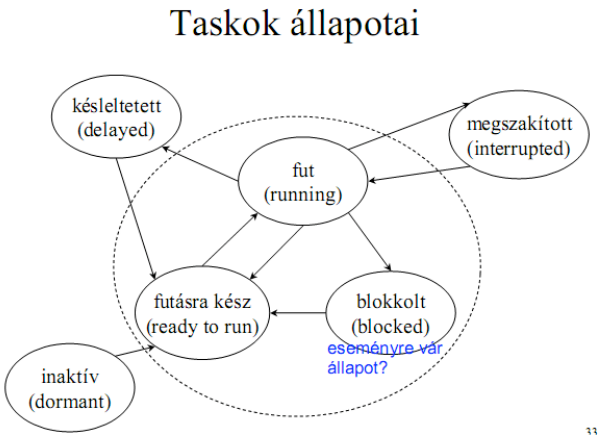
- **<u>Kölcsönös kizárás (mutual exclusion, MUTEX):</u>**
  - annak a biztosítása, hogy a közös erőforrást egy időben CSAK annyi részfeladat használja, amennyi mellett a helyes működés garantálható.
  - többnyire a használt erőforrást valamilyen módon lock-oljuk, lezárjuk.
  - nem engedjük hozzáférni a többi részfeladatot
  - Hogyan tudjuk megoldani és milyen részletességgel oldjuk meg? ⟶ mindenhol máshogy van (de csak kicsit ☺)
- **<u>Kritikus szakasz (critical section):</u>**
  - a részfeladat azon kódrészletei, ahol biztosítjuk a kölcsönös kizárást egy bizonyos közös erőforrásra
  - a kritikus szakasz a közös erőforráshoz tartozik.
- **<u>Atomi művelet (atomic operation):</u>**
  - nem megszakítható művelet, amelyet a processzor egyetlen utasítással hajt végre.
  - egyprocesszoros rendszerekben „bármilyen" műveletsorozat atominak tekinthető a műveletsor elején az IT tiltásával, majd a végén engedélyezéssel.
  - Test and Set vagy Read Modify Write modern processzorokban
  - utasítás prefixek alkalmazásával érhetünk el hasonló eredményt.
  - a közös erőforrásokhoz való hozzáférést (lock) atomi műveletekre fogjuk visszavezetni.

<u>Kommunikáció</u>: részfeladatok egymással információt cserélnek (együttműködve oldanak meg egy közös feladatot) → ez elengedhetetlen feltétele az egységbe foglalásuknak

- közös erőforrásokon keresztül történik
  - többnyire memóriaterületen keresztül történik, ha az architektúra lehetővé teszi
- de nagyon sokféle megoldás van

Részfeladatok állapotai (taszkot ért alatta, tehát nem job):





- FUT: Max annyi lehet ahány végrehajtó egység van
- FUTÁSRA KÉSZ: csupán a végrehajtó egység hiányzik a futáshoz, minden más erőforrás rendelkezésre áll

- ESEMÉNYRE VÁR: CPU-n kívül más erőforrás is hiányzik a futáshoz (pl.: más részfeladat által használt közös erőforrás)

Az állapotátmeneti ábra jellegzetes, szinte minden OS-ben megtalálható bizonyos kiegészítésekkel (extra állapotok, állapotátmenetek) (preemptív OS-ek esetén: FUT → Futásra kész átmenet)

Végrehajtó egységek száma?

- 2 szélsőség
  - sok végrehajtóegység és egy többnyire nehezen párhuzamosítható feladat (a feladatból a részfeladatok nem triviálisan határozhatók meg)
    - Példa: High Performance Computing: szuperszámítógépek, grid rendszerek, sokmagos CPU-k, GPGPU (Cuda OpenCL)
  - Egy vagy több végrehajtó egység: egy párhuzamos,vagy többé-kevésbé jól azonosítható részfeladatok
    - beágyazott rendszerek
      - erőforrások megosztása
      - A kihívás a párhuzamos részfeladatok és az egész probléma hatékony és helyes megvalósítása

Preemptív ütemezés: *Az oprendszer elveheti a futás jogát az éppen futó folyamattól, és "futásra kész" állapotúvá teheti. Közben egy másik folyamatot indíthat el.*

Nem preemptív ütemezés: *Az operációs rendszer nem veheti el a futás jogát a folyamattól. A folyamat addig fut, amíg az általa kiadott utasítás hatására állapotot nem vált, azaz csak maga a folyamat válthatja ki az új ütemezést. (Befejeződik, erőforrásra vagy eseményre vár, lemond a futásról)*

Ütemezés(): a futásra kész részfeladatok közül kiválasztani a futó részfeladatokat

- nagyon sokféle ütemező algoritmus van
  - **soft, hard ütemezők**
  - statikus vagy dinamikus ütemezés
  - nagyon nagy komplexitási különbségek
  - preemptív, nem preemptív,
  - valós v nem valós idejű
- az ütemező a munkája során szükséges adatokat különböző adatstruktúrákban tárolja
  - lesz memóriaigénye
- Az adatstruktúrával operálása miatt lesz CPU használata is
- kontextust kell váltani, ha egy új taszk kerül ütemezésre:
  - ütemező futása előtt:
    - le kell menteni a futó részfeladat állapotát (kontextus)
    - helyre kell állítanunk az ütemező állapotát
  - ütemező futása után
    - le kell menteni az ütemező állapotát
    - helyre kell állítani a futásra kiválasztott részfeladat állapotát

Ütemező választás:

- nincs univerzális ütemező algoritmus
- a feladatok ismeretében a feladatnak megfelelő ütemezési algoritmust kell választani
- a legtöbb beágyazott operációs rendszer erre lehetőséget ad
- ütemező is egy lecserélhető szoftver komponens (még Win XP-ben is)
- az ütemező lecserélhető, kibővíthető, vagy megkerülhető (Interval Zero RTX: Windowsból valós idejű ütemezőt csinál)

Kérdések:

késleltetés / válaszidő

- megszakításkésleltetés: interrupt latencyt ~ms, de RTOS-nél: ~us
- annak ingadozása (jitter)
- statisztikai jellemzők

ütemező és az idő kapcsolata:

- ütemező algoritmusok jelentős részének követnie kell az időt
- ütemezési feladat (részfeladatok periodikus, vagy adott időben történő futtatása)
- időmérés, időtúllépés meghatározása

HW időzítő: megszakítás periodikusan: **HW ütemező???**

- az egyik legfontosabb erőforrás, amit az OS-ben meg kell valósítani
- ezután az ütemező periodikusan fut
- fontos funkció: virtuális (SW) timer megvalósítása lehetséges így
  - egy időzítő a részfeladatok számára: kb mennyi időt szeretnének várni
  - virtuális SW timernek van periódusideje ~1-10 ms (régebben 20 ms) = system tick: ennyi időnként fut az OS ütemezője - *időszelet*
  - ha 40 ms-ot akarunk várni: vár 4 órajelet és a progi futásra kész állapotba kerül → nem tudjuk pontosan biztosítani mikor fogunk futni + jittere is van → nem használható mindenre (lehet: passw megadására 10 sec-e van vkinek, de pl nagy pontosságok elérésére nem jó) (egy láncolt listában vannak részfeladatok HW interrupt jön ütemező jön és kiválasztja ki fog futni)

Mit tudunk tenni, ha ennél rövidebb ideig szeretnénk várakozni?

- más ütemezési algoritmust választunk: idő alapú ütemezést választunk (SJF, SRTF, HRR)
- aktívan várakozni
  - CPU ciklusú várakozás:
    - az utasításvégrehajtási idő váltakozhat → bogomips
    - compiler kiszedheti az optimalizációtól függően
  - timestamp counter alapú
    - HW timer számlálója
    - Pentium Timestamp Counter

- más HW interrupt alapú várakozás

→ az aktív várakozást és a más HW alapúakat az ütemező befolyásolja

Mekkora system tick a megfelelelő?

- egyik oldalról minél kisebb felbontást szeretnénk
- másik oldalról minél kisebb, annál nagyobb az overhead
- ha nem realtime interruptot elfogadunk alkalmazásainkban az gáz lehet real time rendszerekben

Mikor fut az ütemező?

- ha van óraütés
- minden más külső HW interruptra
- belső esemény létrejöttekor
  - közös erőforrás felszabadul
  - egy részfeladat egy eseményt küld
- futó részfeladat várakozó állapotba helyeződik (erőforrásra kezd várni)
- futó részfeladat lemond a futási jogáról (yield)

04.08 TEXT fajl

6.Folyamatok leírásának eszközei, szálak, architektúrális és tervezési minták (általában és ebben a környezetben

`Hogyan tudunk párhuzamos folyamatokat leírni szoftverben - architektúrális minták.`

Párhuzamos eseményvezérelt programozás támogatása:

- néhány kísérleti rendszertől eltekintve ugyanazokon az elveken alapulnak, hasonló funkciókat támogatnak.
- ugyanakkor az interfész és az interfész mögött elrejtett funkció sok esetben eltérő (kis mértékben)

Tervezési / architektúrális minta (pattern):

- **újrafelhasználás szintjei:**
  - kód újrafelhasználás (kód vagy library használata)
  - példaprogram
- **ötletek, megoldások újrafelhasználását hogyan tudjuk biztosítani?**
  - **tervezési / architektúrális minta**: A szoftverfejlesztés során felmerülő problémákra adható általános megoldás. A minta leírása tartalmazza:
    1. minta neve
    2. a minta céljának és alkalmazási környezetének, követelményeinek egyértelmű megoldása
    3. a megoldás megadása
    4. követelmények felsorolása
    5. alkalmazási példa vagy példák
- **antiminták (anti pattern):**
  - Hogyan ne csináljuk…?
  - vita: jó ötlet-e rossz megoldásokat bemutatni? ☐ ☻
- ***Preemptív ütemezés:*** *Az oprendszer elveheti a futás jogát az éppen futó folyamattól, és "futásra kész" állapotúvá teheti. Közben egy másik folyamatot indíthat el.*
- ***Nem preemptív ütemezés:*** *Az operációs rendszer nem veheti el a futás jogát a folyamattól. A folyamat addig fut, amíg az általa kiadott utasítás hatására állapotot nem vált, azaz csak maga a folyamat válthatja ki az új ütemezést. (Befejeződik, erőforrásra vagy eseményre vár, lemond a futásról)*

Párhuzamos részfeladatok (vagy folyamatok) leírása: (Architekturális minták)

- alacsony szintű vezérlési szerkezetek
  - Round – Robin architektúra
  - Round – Robin megszakításokkal
  - függvény – sor alapú (FIFO-FCFS?)
- nem alacsony szintűek - operációs rendszerek:

- co – rutin vagy fiber
- process (folyamat)
- thread (szál)

Round – Robin architektúrák:

```
while (1) {

    if (  ) {subtask1 ( ) ; }

    if (  ) {subtask2 ( ) ; }

    if (  ) {subtask3 ( ) ; }

    }
```

- adott feladatokat feltételesen ciklikusan futtat
- nincs megszakításkezelés
- a perifériákat pollinggal kezeljük

valós idejűség → round – robin , a worst-case végrehajtási idő lényegében megadható

- törékeny architektúra
  - új részfeladatok hozzáadása felbontja az ütemezést és az időzítést
  - részfeladatok leírását a forrásnyelv nyelvi konstrukciói szeparálják → könnyű hibázni
- kölcsönös kizárási probléma egyszerű szabályok betartásával biztosítható
- előnye: egyszerű
- proc kihasználtság 100%
- HRT viselkedés lassú
- Taskok közötti komm. Megosztott változókkal
- Nem preemptív

Round – Robin architektúra megszakításokkal:

- megszakításokat használ, minden eszközhöz flaget rendelünk
- flaget billenti be a megszakításkezelő rutin
  - hardverspecifikus dolgokat is kezeli

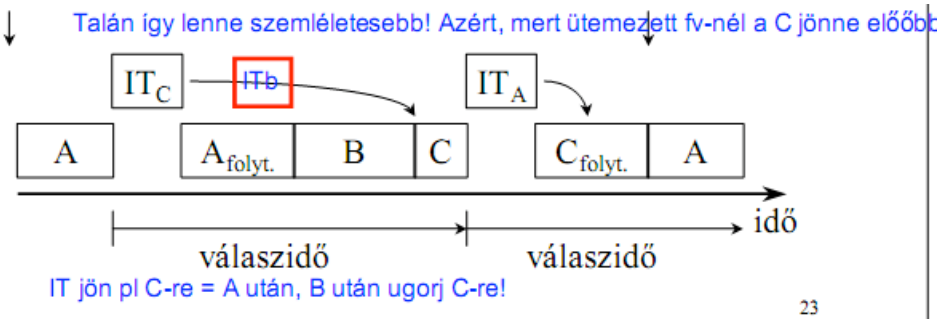
  egy végtelen ciklus a flaget ellenőrzi
- ha a flag be van billentve, akkor meghívja a kezelő függvényt
- utána visszabillentjük a flaget
- jobban kezeli az időkritikus részeket???
  - megszakításkezelő rutinban történik???
  - ha van prioritás a megszakításkor, akkor még kedvezőbb
  - a válaszidő szórása még mindig nagy (válaszidő: interrupt bejön, a teljes feldolgozás végéig tart)
- az architektúra törékeny
- ennél az architektúránál el lehet a processzort küldeni aludni, ha nincs munka (fogyasztás szempontjából kedvező: IT alapú)
  - a beérkező IT-re a processzor felébred
- függvénypointerek: milyen memóriacímtől kezdődő függvényt kívánunk meghívni

- - azt kell tudni, hogy milyen hívási paraméterű és visszatérési értékű függvényről van szó (stack miatt)
- időkritikus részfeladatot többször is felsorolhatunk, ezzel csökkentve a kiszolgálási időt
- alacsony fogyasztású rendszerben alkalmazható

IT-vel kiegészített ciklikus programszervezés

```
FLAG A, B;

void interrupt A_Handler() { Handle_HW_A(); A=TRUE; }

void interrupt B_Handler() { Handle_HW_B(); B=TRUE; }

void interrupt C_Handler() { Handle_HW_C(); C=TRUE; }

void main() {

while (TRUE){

if A  {A=FALSE; Service_A(); }

if B  { B=FALSE; Service_B(); }

if C  { C=FALSE; Service_C(); }

...

}

}
```

ciklushatárok



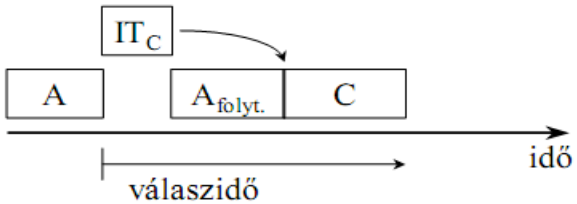
Függvény – sor alapú architektúra(FCFS):

- képes a prioritások kezelésére
  - ez a sor kezelésétől függ, alapvetően FIFO , CPU – 100%
  - IT szinten MCU / CPU IT vezérlő
- kölcsönös kizárás az IT szinten és a részfeladatok között
- preemptivitás nincs!
- nem annyira törékeny
- a „függvény” nyelv konstrukció erős, jól szeparál
- megszakítást kezelő rutinok kielégítik az eszköz sürgős igényeit

- további feldolgozáshoz egy várakozási sorban elhelyeznek egy függvényre mutató pointert
- a várakozási sort a main-ben dolgozzuk fel
- meghatározott algoritmus szerint a sorból kivesszük a függvényre mutató pointereket
- lényegében ez is interrupt alapú
- - interrupt: kezeli a hardvert (időkritikus hardver közeli dolgokat)
- - majd önmaga egy függvényekre mutató pointereket tartalmazó sorba behelyezi a magas szintű (nem időkritikus) feladatokat
- - végtelen ciklusban: ha a sor nem üres, levesszük a függvényt (első függvény) és meghívjuk
- - a kölcsönös kizárást meg kell oldani az IT-kezelő és a magas szintű (nem időkritikus) feladatokat ellátó függvény között
- - mert elképzelhető, hogy miközben a magas szintű feladat fut, befut még egy interrupt
- - válaszidő = a leghosszabb task + az IT végrehajtása + a kezelő függvény végrehajtása
- - nem borul fel annyira könnyen az architektúra (csak azok a kódok futhatnak le, amelyek egy függvényben benne vannak)
- - a függvény szeparálja a taskokat
- - az új feladatok csak akkor tudják felborítani a rendszer működését, ha magasabb prioritásúak (a magasabb prioritású task egyébként minden architektúrában felborítja a működést)
- - nem preemptív: ha már fut egy kezelő függvény, akkor hiába érkezik be egy újabb kérés (és lefut az IT kezelő függvénye), csak akkor futhat le, ha az előző befejezte

Ütemezett függvények módszere

```
void interrupt A_Handler() { Handle_HW_A(); PutFunction(Service_A); }
void interrupt B_Handler() { Handle_HW_B(); PutFunction (Service_B); }
void interrupt C_Handler() { Handle_HW_C(); PutFunction (Service_C); }
void Service_A();
void Service_B();
void Service_C();
void main() {
  while (TRUE){
     while (IsFunctionQueueEmpty()) ;
          CallFirstFromQueue();
          }
}
```



Kiemelés sorrendje lehet prioritásos is -> Hangolhatjuk a rendszert, akár az IT-vel kieg. ciklikus programszervezének a kiszolg idejét is elérhetjük, ami persze visszáesés, kisz. idő növekedés is lehet.

Coroutine (Cooperative OS): ez pontosan mi is?

- kooperatív feladatok leírására szolgál
- a szubrutin általánosítása

```
        - megvalósítható OS-ben, nyelvi elemként, saját magunknak is
megvalósítható
                (OS: fiber, nyelvi elem: coroutine)
        - stack alapú nyelvekben (C, C++) nehezen implementálható, de pl.
FreeEcos támogatja a coroutine-t
        - első belépési pontnál úgy működik, mint függvény v. szubrutin
        - további belépéseknél: utolsó kilépési pontnál folytatja
                - kilépés: yield to coroutine
                - mintha lenne egy függvény több returnnel, és a a függvény
újbóli meghívásakor az előző return utánról indulunk

var q := new queue;
coroutine produce {
loop while q is not full {
        create new item;
        add item to queue;}
        yield to consume;}

coroutine consume {
loop while q is not empty {
remove item from q;
use item;
}
yield to produce;
}
```

Operációs rendszerek (konkurens programozás):

Process (folyamat) és Thread (szál)

Process:

- saját stack és munkaterület (memória)
- komplex folyamatleíró: létrehozása és megszüntetése bonyolult és erőforrásigényes
- nem férhet hozzá más folyamatok munkaterületéhez
    - memóriavédelem hardver támogatással(MMU van a CPU-ban)
    - kommunikáció az OS-en keresztül történik
- Linux / Windows alkalmazás: Linux daemon(fork) vagy Windows Service(CreateProcess)
- Beágyazott rendszerek: Windows CE vagy XP Embedded, RT LINUX, Vx Works, QNX

Thread / szál (pehelysúlyú folyamat) fordítunk:

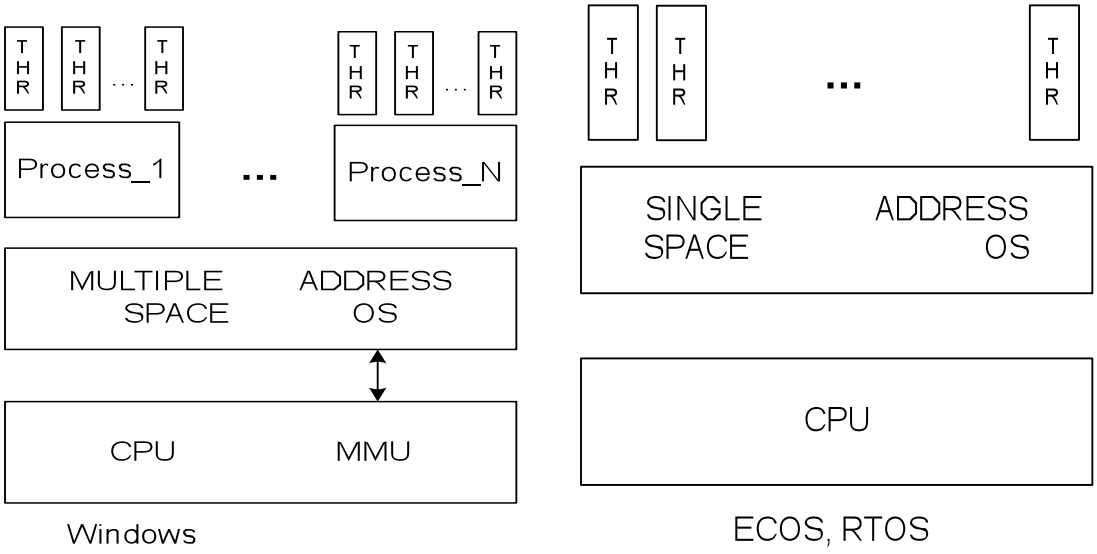
Általában minden program több szálból áll. Minden egyes szál egy részprogramként értendő, s ezeket külön futtatja az OS. (Szálakba szervezzük a részfeladatokat.)

- párhuzamosan futó részfeladatok kommunikálni tudnak
- spec függvényből hozható létre, az őket tartalmazó folyamat munkaterületén futnak
- szálaknak saját stack-jük van
- egyszerű „szálleíró”
    - alacsony erőforrású a létrehozásuk és megszüntetésük
    - definíció szerűen hozzáférhetnek a saját folyamatuk munkaterületéhez

        - a saját folyamatukban létrehozott szálakkal memórián keresztül kommunikálhatnak
        - kis erőforrásigény és késleltetés
    - más folyamatokkal csak OS hívásokon keresztül kommunikálhatnak
    - OS támogatás (jellemző) vagy thread library (pthreadunkat OS a mi folyamatunkat ütemezi, a szálakat pedig más fogja ütemezni)

Megvalósítás:

- **Windows és linux (esetleg beágyazott rendszerek MMU-val):**
    - **alkalmazás, service, daemon (Linux)= folyamat (folyamatokon belül futhatnak szálak)**
- **beágyazott rendszerek MMU nélkül:**
    - **csak szálakat tudunk létrehozni**



7.A beágyazott rendszerek gyakori rendszerarchitektúrái és azok hatása a párhuzamos eseményvezérelt szoftverre.

Párhuzamos részfeladatok (vagy folyamatok) leírása: (Ütemezési algoritmusok – Beágyazott rendszerek architekturális mintái)

- alacsony szintű vezérlési szerkezetek
  - Round – Robin architektúra
  - Round – Robin megszakításokkal
  - függvény – sor alapú (FIFO-FCFS?)
- nem alacsony szintűek - operációs rendszerek:
  - co – rutin vagy fiber
  - process (folyamat)
  - thread (szál)

Round – Robin architektúrák:

```
while (1) {

    if (  ) {subtask1 ( ) ; }

    if (  ) {subtask2 ( ) ; }

    if (  ) {subtask3 ( ) ; }

    }
```

- adott feladatokat feltételesen ciklikusan futtat
- nincs megszakításkezelés
- a perifériákat pollinggal kezeljük

valós idejűség ⟶ round – robin , a worst-case végrehajtási idő lényegében megadható

- törékeny architektúra
  - új részfeladatok hozzáadása felbontja az ütemezést és az időzítést
  - részfeladatok leírását a forrásnyelv nyelvi konstrukciói szeparálják ⟶ könnyű hibázni
- kölcsönös kizárási probléma egyszerű szabályok betartásával biztosítható
- előnye: egyszerű
-

Round – Robin architektúra megszakításokkal:

- megszakításokat használ, minden eszközhöz flaget rendelünk
- flaget billenti be a megszakításkezelő rutin
  - hardverspecifikus dolgokat is kezeli

  egy végtelen ciklus a flaget ellenőrzi
- ha a flag be van billentve, akkor meghívja a kezelő függvényt
- utána visszabillentjük a flaget
- jobban kezeli az időkritikus részeket???
  - megszakításkezelő rutinban történik???
  - ha van prioritás a megszakításkor, akkor még kedvezőbb
  - a válaszidő szórása még mindig nagy (válaszidő: interrupt bejön, a teljes feldolgozás végéig tart)
- az architektúra törékeny
- ennél az architektúránál el lehet a processzort küldeni aludni, ha nincs munka (fogyasztás szempontjából kedvező: IT alapú)
  - a beérkező IT-re a processzor felébred

- függvénypointerek: milyen memóriacímtől kezdődő függvényt kívánunk meghívni
- - azt kell tudni, hogy milyen hívási paraméterű és visszatérési értékű függvényről van szó (stack miatt)
- időkritikus részfeladatot többször is felsorolhatunk, ezzel csökkentve a kiszolgálási időt
- alacsony fogyasztású rendszerben alkalmazható

Függvény – sor alapú architektúra(FCFS):

- képes a prioritások kezelésére
  - ez a sor kezelésétől függ, alapvetően FIFO
  - IT szinten MCU / CPU IT vezérlő
- kölcsönös kizárás az IT szinten és a részfeladatok között
- preemptivitás nincs!
- nem annyira törékeny
- a „függvény" nyelv konstrukció erős, jól szeparál
- megszakítást kezelő rutinok kielégítik az eszköz sürgős igényeit
- további feldolgozáshoz egy várakozási sorban elhelyeznek egy függvényre mutató pointert
- a várakozási sort a main-ben dolgozzuk fel
- meghatározott algoritmus szerint a sorból kivesszük a függvényre mutató pointereket
- lényegében ez is interrupt alapú
- - interrupt: kezeli a hardvert (időkritikus hardver közeli dolgokat)
- - majd önmaga egy függvényekre mutató pointereket tartalmazó sorba behelyezi a magas szintű (nem időkritikus) feladatokat
- - végtelen ciklusban: ha a sor nem üres, levesszük a függvényt (első függvény) és meghívjuk
- - a kölcsönös kizárást meg kell oldani az IT-kezelő és a magas szintű (nem időkritikus) feladatokat ellátó függvény között
- - mert elképzelhető, hogy miközben a magas szintű feladat fut, befut még egy interrupt
- - válaszidő = a leghosszabb task + az IT végrehajtása + a kezelő függvény végrehajtása
- - nem borul fel annyira könnyen az architektúra (csak azok a kódok futhatnak le, amelyek egy függvényben benne vannak)
- - a függvény szeparálja a taskokat
- - az új feladatok csak akkor tudják felborítani a rendszer működését, ha magasabb prioritásúak (a magasabb prioritású task egyébként minden architektúrában felborítja a működést)
- - nem preemptív: ha már fut egy kezelő függvény, akkor hiába érkezik be egy újabb kérés (és lefut az IT kezelő függvénye), csak akkor futhat le, ha az előző befejezte

Coroutine (Cooperative OS): ez pontosan mi is?

- kooperatív feladatok leírására szolgál
- a szubrutin általánosítása
- megvalósítható OS-ben, nyelvi elemként, saját magunknak is megvalósítható
  (OS: fiber, nyelvi elem: coroutine)

- stack alapú nyelvekben (C, C++) nehezen implementálható, de pl. FreeEcos támogatja a coroutine-t
- első belépési pontnál úgy működik, mint függvény v. szubrutin
- további belépéseknél: utolsó kilépési pontnál folytatja
  - kilépés: yield to coroutine
  - mintha lenne egy függvény több returnnel, és a a függvény újbóli meghívásakor az előző return utánról indulunk

```
var q := new queue;
coroutine produce {
loop while q is not full {
        create new item;
        add item to queue;}
        yield to consume;}

coroutine consume {
loop while q is not empty {
remove item from q;
use item;
}
yield to produce;
}
```

8. Kölcsönös kizárás, szinkronizáció, kommunikáció módszerei közös memóriával rendelkező rendszerekben

Hogyan védjük a közös erőforrásokat? (Kölcsönös kizárás biztosítása (MUTEX))

1.Folyamatok leírásának eszközei??

- Kik férhetnek hozzá a közös ef-khoz?
  - ISR – Interrupt Service Routine
  - Részfeladatok
  - DMA – Direct Memory Access (CPU mentes memoriakezelo blokk)

- Lehetőségek:
  - IT tiltás és engedélyezés
  - ütemező tiltása és engedélyezése
  - locking (nem jó megoldás, de minden más rosszabb)
  - Új kutatási területek, megoldások:
    - Software Transactional Memory (STM)
    - Software Isolated Precesses (pl.: MS Singularity)

Ezek kifejtése:

- IT tiltás
  - IT rutin és részfeladatok közötti problémák megoldására az egyetlen lehetőség
  - könnyű hibázni (elfelejtünk tiltani vagy engedélyezni /lehet, hogy valamelyik feltételes utasítás ágon az IT rutin tiltva marad/)

- Ütemező tiltás:
  - OS hívásokkal (ha támogatott) egyszerűen megoldható
  - ütemező sokáig nem tiltható
    - értelmét veszti az ütemező
  - könnyű hibázni (tiltás/engedélyezés elhagyása)

- Lockolás:
  - behajtani tilos tábla az erőforrásra
  - erőforrás specifikus
  - az a kérdés hogyan várakozunk az erőforrásra:
    - aktív várakozás (live lock, spin lock, busy waiting, spining)
      - CPU erőforrás pazarlás történik (folyamatosan ellenőrizzük az erőforrás felszabadult-e)
      - a rendszer csak külső esemény hatására haladhat előre
      - ezt a fajta problémát javítani lehet:
        - Round Robin megszakítással mintája
        - aktív várakozás timerral
      - kis fogyasztású rendszerekben kerülendő
    - passzív várakozás(sleeplock)
      - az ütemező által karbantartott várakozási sorokban várnak a részfeladatok
      - mi történik, ha részfeladat közös erőforrást akar használni:
        - ha nem lockolt lockol és fut tovább
        - ha lockolt, akkor felfüggesztődik és az adott erőforráshoz tartozó várakozási sorba kerül

- lényegesen erőforrástakarékosabb, mint az aktív várakozás, de van overhead (rezsiköltség?)
- a felszabaduló erőforrásra váró részfeladat csak futásra kész állapotba kerül→időzítést vizsgálni kell emiatt-(CPU munka)
- alacsony fogyasztású rendszerekben a CPU aludhat, ha nincs futásra kész részfeladatrész
- van egy Idle részfeladat, ami a CPU sleep utasítását hajtja végre végtelen ciklusban→IT-re ébred fel a rendszer, hiszen belső esemény nem történhet, mert nincs futó részfeladat

Hogyan lockolhatunk?

- <u>lock bit</u>
  - többnyire egyszerű OS használata nélkül megvalósított szoftverekben használjuk
  - védendő erőforráshoz tartozó logikai változó (boolean)
  - lock bit jelentése:
    - FALSE: az erőforrás szabad (használható)
    - TRUE: az erőforrás használt (számunkra nem használható)
  - Belépés során:
    - teszt: Lock bit == FALSE → lock bit = TRUE → használjuk ez EF-t
    - if (lock_bit == TRUE) → wait amíg FALSE nem lesz
    - tiszta aktív várakozás
  - Kilépési művelet:
    - lock_bit = FALSE
  - gond: ha megvizsgáljuk és látjuk, hogy szabad,de jön egy IT ami előttünk lefoglalja az erőforrást az gond→
  - lock_bit figyelése, állítása atomi műveletként végzendő→IT tiltás, vagy CPU utasítás (Test and Set)

- <u>szemafor</u>
  - lehet bináris és counter típusú
  - bináris: egy időben egy részfeladat lehet a kritikus szakaszban
  - counter t<p: egy időben max N részfeladat lehet a kritikus szakaszban (pl.: ha winchi több parancs végrehajtására alkalmas, írhat olvashat is egyszerre)
  - ez már OS szolgáltatásként kerül megvalósításra
  - belépés:
    - P() , Wait() esetleg Pend() utasítással
  - kilépés:
    - V() , Signal() esetleg Post() utasítással
  - counter típusú esetén a belépés és a kilépés számmal paraméterezett (egy időben egynél több erőforrás lefoglalható és felszabadítható → jobb, mint ha egymás

után 2-t foglalok le, pl.: ha nekem 2 kell, de egyszerre csak egyet kapok meg, mert más megelőz, akkor holtpont lehet)
    - hány egység lép ki és be

- *kritikus szakasz objektum*
    - *CriticalSection objektum létrehozása*
    - *Enter() metódust meghívhatjuk*
    - *Leave() a kilépésre szolgál*
    - *az objektum megszüntethető*
    - *tulajdonképpen bináris szemafor szerű működés (WINDOWS)*

- *mutex*
    - *ugyanaz csak más a neve*
    - *Aquire(), Waitone()*
    - *Release() kilépésre*
    - *kritikus szakasz és objektum majdnem ugyanaz, csak más a neve*
    - *miért is jó a lockolás*
        - *kölcsönös kizárás*
        - *szinkronizációt visszavezetjük lockolásra*
            - *létrehozunk egy védendő objektumot, ami valójában nem közös erőforrás a rendszer elsődleges működése szempontjából*
            - *egy és kétoldalú randevú:*
                - *egyoldalú: van egy szemafor, egy taszk_1 és taszk_2: taszk_1 szólni akar taszk_nek(taszk2_nek?)→a taszk_2 waittel vár a szemaforra, a taszk_1 pedig szignalt küld a szemafornak ha azt szeretné, hogy taszk_2 fusson*

Taszk_1→signal()→szemafor←wait()←Taszk_2

- *kétoldalú ugyanaz csak 2 szemaforral*

A lockolás során elkövetett tipikus hibák:

- belépés / kilépés elmaradása: ha nem lépek be, de lefut → a kilépés erőforrás instabil állapotba kerülhet
- többszöri be vagy kilépés
- más erőforrásokat lockolunk (pl hasonló változóneveink vannak→ soros helyett a párhuzamos portot lockolom)
- deadlock v. livelock: erőforrások sorrendjének hibás kezelése: a rendszer részfeladatai képesek lennének működni, de a rendszer nem, livelock: a rendszer csinál valamit, de nem jut el a céljáig (hibás deadlock feloldás eredménye)
- az erőforrások indokolatlan lassan történő lezárása→kiéheztetődik

    - lockolás finomsága (conse or fire gain locking)
- prioritás inverzió problémája → oprendszer konfigurálásával kiküszöbölhető

Bizonyos hibák kivédhetők monitor alkalmazásával. Lokalizáljuk a lockolással kapcsolatos feladatokat a közös erőforrást körülvevő API-val.

- a lockolás nem szétszórtan történik a programban, hanem egyetlen, a közös erőforráshoz szorosan tartozó programrészletben (modulban)
- automatikus nyelvi szinten támogatott (JAVA): a compiler szúrja be a kölcsönös kizárást biztosító konstrukciókat
- hasonlót kézzel is készíthetünk → egy monitorszerű funkciót hozunk létre →jobban járunk: nem kell a párhuzamos programozással foglalkozni a programozónak
- Hoare és Meza szemantika (működés)
    - Charles Richard Hoare a monitor ötletadója és az elméleti alapok kidolgozója:
        - erőforrás felszabadítása után az erőforrásra váró folyamat azonnal futni kezd
        - nehéz megvalósítani
        - preemptív ütemezők működésével nehezen összeegyeztethető
    - Meza egy programozási nyelv volt, amit a Xerox Parc: párhuzamos programozást támogató programnyelv:
        - az erőforrás felszabadulása után az erőforrásra váró folyamat csak futásra kész állapotba kerül
        - „notofied all" broadcast jellegű üzenetek küldése is lehetséges
        - minden az adott eseményre várórészfeladat futásra kész állapotba kerül

<u>Kommunikáció:</u>

1. közös erőforrásként kezelt memóriaterületen keresztüli kommunikáció→"ha van közös Memória"
2. adatstruktúra:

b. procedurális nyelvek: tömb, struktúra
c. OO nyelvek: objektum
    1. HW analógia: dual portos RAM: párhuzamosan 2 adatvezeték és címvezeték van, kölcsönös kizárást HW úton biztosít
    2. kölcsönös kizárás biztosítása kötelező
    3. <u>mailbox és message queue</u>: operációs rendszer szolgáltatás, mely elsősorban kommunikáció célú: oprendszer által szolgáltatott kommunikációs célú adatstruktúra
d. referenciák (C/C++ pointer)-t rakunk a queueba: ha okosan használjuk, akkor nem is kell kölcsönös kizárás
e. message queue n darab ilyen pointert tud tárolni: akkor használható ha a termelő rövid idő alatt sok infót állít elő, fogyasztó szép lassan használja fel a dolgokat

f. ha elosztott rendszerbe kerülünk, akkor érték szerinti információ átadás történik→ az infót fizikailag küldjük el (Microsoft MSMQ, IBM WebShpere MQ, UNIX PIPE (ls –la /les)ilyeneknél üzenet nem veszhet el, pl.: banki tranzakcióknál)

9.Kölcsönös kizárás, szinkronizáció, kommunikáció módszerei elosztott rendszerekben, architektúrális minták elosztott rendszerekhez

Elosztott rendszerek:

- nincs közös memória

- valamilyen kommunikációs hálózat a node-k között
  - ez többnyire szoftvervezérelt
  - a HW a mai világban: CAN, FlexRay, TT Ethernet, AFDX
- az információküldés a kommunikációs interfészen keresztül történik
  - alacsonyabb megbízhatóságú, mint az osztott (megosztott??-ez kulcsszó) memórián keresztüli kommunikáció
  - nagyságrendsekkel nagyobb a késleltetés, és annak ingadozása (jitter)
  - nagyságrendekkel kisebb a sávszélesség
  - lehetnek architektúrális különbségek (endianess /little vagy big endian/)
- Üzenet küldés és fogadás:
  - SendMessage(to, from, message) jellegű függvénnyel
    - többnyire aszinkron, azonnal visszatér
    - FROM mezőt automatikusan az alacsonyabb SW réteg beszívhatja
      - unicast
      - multicast
      - broadcast
  - RecieveMessage(), Listen()
    - blokkoló, a megérkezett üzenettel tér vissza
    - opcionálisan megadhatunk szűrőfeltételeket (pl.: forráscím, üzenettípus)
    - opcionálisan lehet timeout (ha 5 másodpercig nem jött, akkor hibát küld)

rmi epc

Lehetséges-e egy nem megbízható kommunikációs csatornán 1 valószínűséggel megbízhatóan kommunikálni? → NEM

2 hadsereg probléma: 2 sereg 2 dombon, ellenség a völgyben: ha 2 hadsereg együtt támad, akkor tud csak győzni → futárokat küldenek, a közös támadási időpontról,de a futárokat az ellenség el tudja kapni. 1 valószínűséggel nem lehet megállapítani, hogy mikor lesz a támadás, mert az egyik fél soha nem lehet 100%-ig biztos, hogy az üzenete megérkezett-e a másik táborba vagy sem. → Gyakorlatban: 3 utas kézfogás (TCP is így veszi fel és bontja a kapcsolatot)

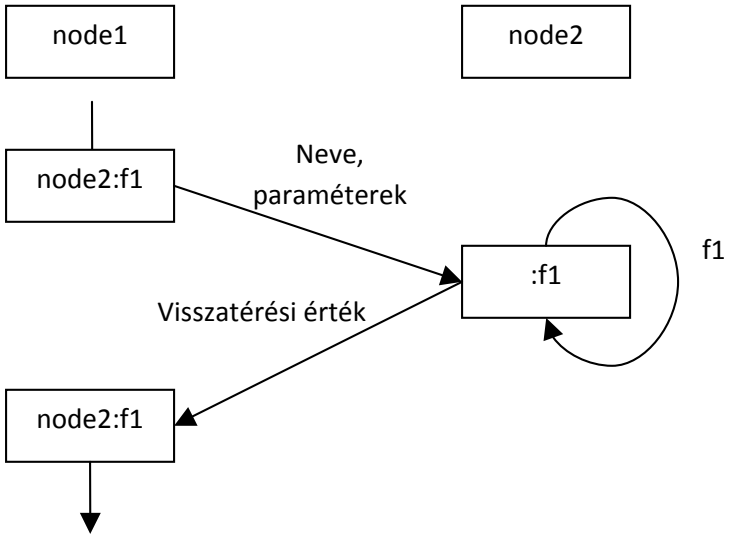
- **Virtuális osztott memória**:
  - Egy szolgáltatás, melyet üzenetekkel tudunk igénybe venni, és a nyújtott szolgáltatás jellegre hasonló a fizikai osztott memóriához → írás és olvasás leképzése üzenetekre
  - A modern buszok (pl. PCIe) is üzenet alapúak → a tényleges, fizikai memória üzenetekkel kezelhető
  - A WEB is felfogható ennek…

- **Mailbox és MessageQueue** (már korábban volt szó róla)

- **Távoli eljárás/metódus hívás**:
  Remote Procedure Call, Remote Method Invocation, operációs rendszerekben elterjedten alkalmazzák

- **Rendszerarchitektúrák**:
  - Publich-Subscribe
  - Virtual Databus
  - Proxy
  - Bróker (Broker)

Távoli eljárás hívás (Remote Procedure Call):



→ A hívó oldalon (node1) egy **Stub** (függvényfej) fut le
  - Paramétereket szabványos formába konvertálja (csomagolja)
  - Üzenetekben átküldi a függvényhívás tényét és paramétereit a node2-nek (tételezzük fel, hogy node2-n a függvény létezik)

→ A hívott féloldali Stub az üzenet vétele után
  - Üzenet vétele
  - Lokális formára hozza a paramétereket
  - Lokális f1 hívása (mellékhatások!): node2-n maradandó változások
  - Visszatérési érték szabványos formára hozása
  - Üzenetben elküldeni a visszatérési értéket a hívó félnek (node1)

→ A hívó oldalon a Stub visszatérésekor
  - Üzenet vétele
  - Visszatérési érték lokális formára hozása
  - Visszatérés a lokális függvénybe a lokális visszatérési értékkel

→ A hívott fél megtalálásának módszerei (Hogyan történik node2 kiválasztása?)
  - Manuális konfiguráció
  - Bróker architektúra
  - Broadcast vagy multicast üzenet

→ Távoli metódushívás:
  - adott node-n adott objektum adott metódusát hívjuk
  - Adott node adott objektum: referencia azonosítja

- HTTP: URL adja meg az objektumot

→ Tényleges technológiák:
  - JAVA RMI = távoli eljárás hívás virtuális gépek között
  - DCE/RPC (ezt használja némi módosításokkal a Microsoft)
  - SUN RPC – Remote Procedure Call

  A távoli eljárás híváson alapul a távoli metódus hívás is csak magasabb szinten működik.

  - DCOM (Microsoft)
  - WCF – Window Communication Fundation, .NET-ben használják
  - Új technológiák: elsősorban WEB-es technológiák
    - XML-RPC
    - SOAP

<u>Szabványos adatformátumok:</u>

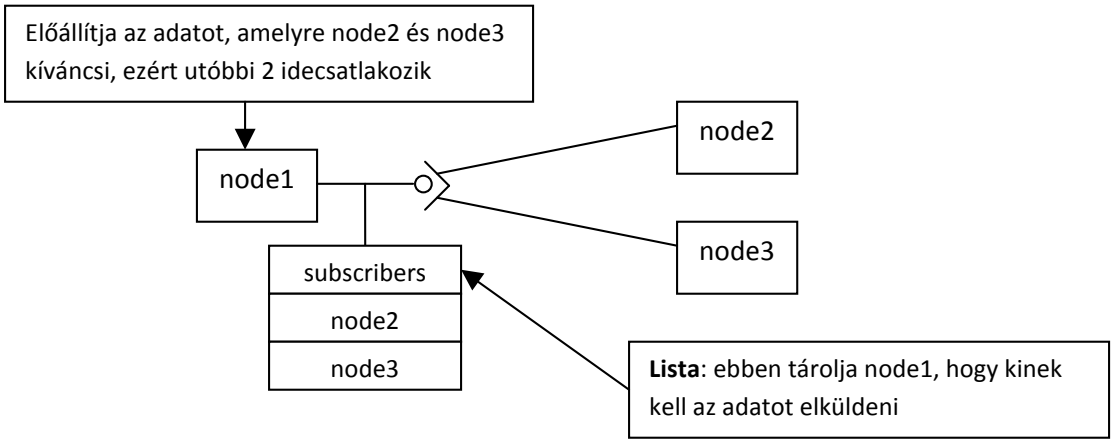
→ BER (Binary Encoding Rules)

  - Nagyszámú eltérő szabvány van
  - Média konténer formátumok – avi, mkv, …

→ XML (eXtendable Markup Language)
  - <int16>896</int16> → http jelölőelemekhez hasonló

<u>Publish-Subscribe:</u> (observer)

→ Résztvevők:
  - Publisher = termelő
  - Subscriber = fogyasztó



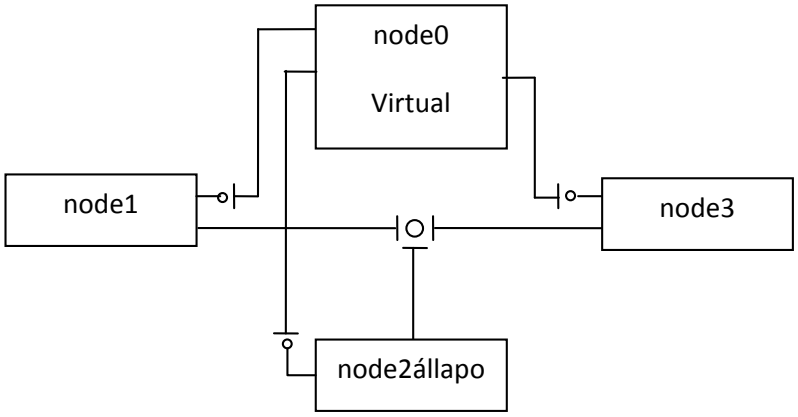
→ Bonyolult hálóztok építhetőek fel
  - Nem elosztott környezetben:

- MATLAB Simulink
- NI Labview

→ Adatvezérelt szerkezet (dataflow)
→ Push módszer (adat küldése) – node1 továbbnyomja az adatot node2-nek és node3-nak

Virtuális adatbusz: (Virtual Databus)

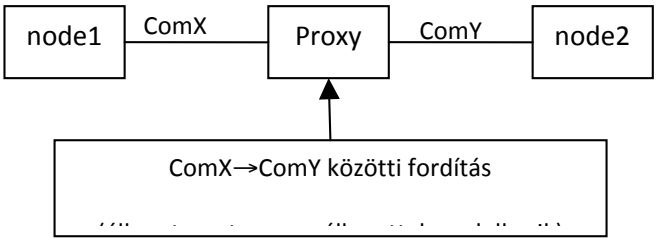
→ Publish-suscriber alapján
→ A busz egy központi node a rendszerben



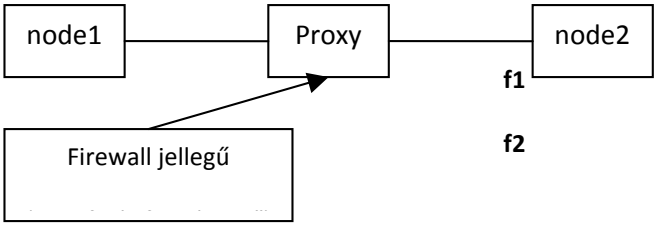
→ Ha valakitől olvas, azt utána kiírja mindegyik node számára
  - node1-3 adatot állíthat elő → node1-3 termelő, node0 fogyasztó
  - node1-3 értesül node0-tól minden információról → node0 termelő node1-3 fogyasztó

Proxy:

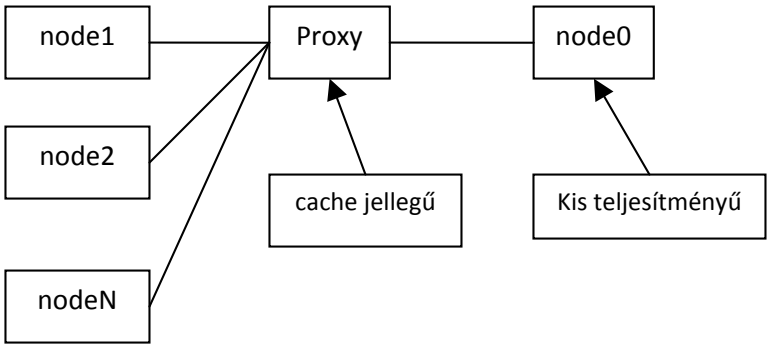
→ beáll két vagy több fél közé a kommunikáció során
→ cél:
  - kompatibilitás megteremtése



  - funkciók elrejtése



- Teljesítményillesztés: Σ(kérés node1-N) >> max. kérés node0



Bróker:

→ Központi szereplő, amely összepárosítja a többi szereplőt
→ Ismert a szereplők számára, egyszerű automatizmussal megtudható, vagy statikusan megadható konfigurációban
→ A felek megadják az igényeket és a képességeket
→ A bróker összeválogatja őket

10. A párhuzamos eseményvezérelt programozás architektúrális mintái, egyszerű párhuzamos architektúráktól a beágyazott operációs rendszerig.

`Hogyan tudunk párhuzamos folyamatokat leírni szoftverben - architektúrális minták.`

Párhuzamos eseményvezérelt programozás támogatása:

- néhány kísérleti rendszertől eltekintve ugyanazokon az elveken alapulnak, hasonló funkciókat támogatnak.
- ugyanakkor az interfész és az interfész mögött elrejtett funkció sok esetben eltérő (kis mértékben)

Tervezési / architektúrális minta (pattern):

- **újrafelhasználás szintjei:**
  - kód újrafelhasználás (kód vagy library használata)
  - példaprogram
- **ötletek, megoldások újrafelhasználását hogyan tudjuk biztosítani?**
  - **tervezési / architektúrális minta**: A szoftverfejlesztés során felmerülő problémákra adható általános megoldás. A minta leírása tartalmazza:
    6. minta neve
    7. a minta céljának és alkalmazási környezetének, követelményeinek egyértelmű megoldása
    8. a megoldás megadása
    9. követelmények felsorolása
    10. alkalmazási példa vagy példák
- **antiminták (anti pattern):**
  - Hogyan ne csináljuk…?
  - vita: jó ötlet-e rossz megoldásokat bemutatni? ☻
- ***Preemptív ütemezés:*** *Az oprendszer elveheti a futás jogát az éppen futó folyamattól, és "futásra kész" állapotúvá teheti. Közben egy másik folyamatot indíthat el.*
- ***Nem preemptív ütemezés:*** *Az operációs rendszer nem veheti el a futás jogát a folyamattól. A folyamat addig fut, amíg az általa kiadott utasítás hatására állapotot nem vált, azaz csak maga a folyamat válthatja ki az új ütemezést. (Befejeződik, erőforrásra vagy eseményre vár, lemond a futásról)*

Párhuzamos részfeladatok (vagy folyamatok) leírása: (Architekturális minták)

- alacsony szintű vezérlési szerkezetek
  - Round – Robin architektúra
  - Round – Robin megszakításokkal
  - függvény – sor alapú (FIFO-FCFS?)
- nem alacsony szintűek - operációs rendszerek:
  - co – rutin vagy fiber
  - process (folyamat)
  - thread (szál)

Round – Robin architektúrák:

```
while (1) {

    if (   ) {subtask1 ( ) ; }

    if (   ) {subtask2 ( ) ; }

    if (   ) {subtask3 ( ) ; }

    }
```

- adott feladatokat feltételesen ciklikusan futtat
- nincs megszakításkezelés
- a perifériákat pollinggal kezeljük

valós idejűség → round – robin , a worst-case végrehajtási idő lényegében megadható

- törékeny architektúra
  - új részfeladatok hozzáadása felbontja az ütemezést és az időzítést
  - részfeladatok leírását a forrásnyelv nyelvi konstrukciói szeparálják → könnyű hibázni
- kölcsönös kizárási probléma egyszerű szabályok betartásával biztosítható
- előnye: egyszerű
- proc kihasználtság 100%
- HRT viselkedés lassú
- Taskok közötti komm. Megosztott változókkal
- Nem preemptív

Round – Robin architektúra megszakításokkal:

- megszakításokat használ, minden eszközhöz flaget rendelünk
- flaget billenti be a megszakításkezelő rutin
  - hardverspecifikus dolgokat is kezeli

  egy végtelen ciklus a flaget ellenőrzi
- ha a flag be van billentve, akkor meghívja a kezelő függvényt
- utána visszabillentjük a flaget
- jobban kezeli az időkritikus részeket???
  - megszakításkezelő rutinban történik???
  - ha van prioritás a megszakításkor, akkor még kedvezőbb
  - a válaszidő szórása még mindig nagy (válaszidő: interrupt bejön, a teljes feldolgozás végéig tart)
- az architektúra törékeny
- ennél az architektúránál el lehet a processzort küldeni aludni, ha nincs munka (fogyasztás szempontjából kedvező: IT alapú)
  - a beérkező IT-re a processzor felébred
- függvénypointerek: milyen memóriacímtől kezdődő függvényt kívánunk meghívni
- - azt kell tudni, hogy milyen hívási paraméterű és visszatérési értékű függvényről van szó (stack miatt)
- időkritikus részfeladatot többször is felsorolhatunk, ezzel csökkentve a kiszolgálási időt
- alacsony fogyasztású rendszerben alkalmazható

IT-vel kiegészített ciklikus programszervezés

```
FLAG A, B;

void interrupt A_Handler() { Handle_HW_A(); A=TRUE; }

void interrupt B_Handler() { Handle_HW_B(); B=TRUE; }

void interrupt C_Handler() { Handle_HW_C(); C=TRUE; }

void main() {

while (TRUE){
```

if A {A=FALSE; Service_A(); }

if B { B=FALSE; Service_B(); }

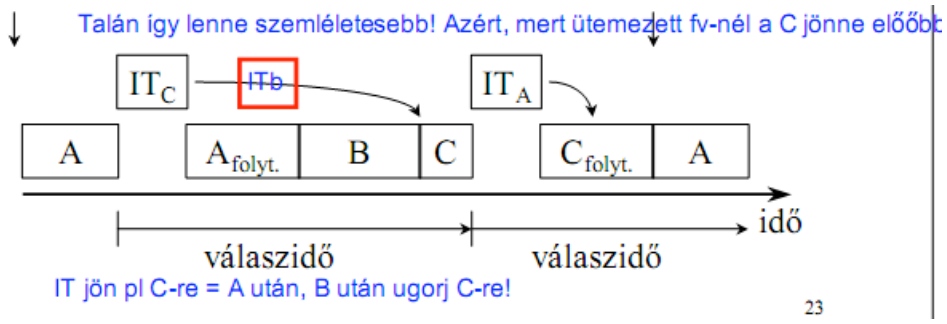
if C { C=FALSE; Service_C(); }

...

}

}

ciklushatárok



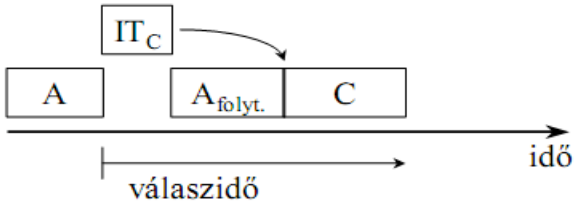
Függvény – sor alapú architektúra(FCFS):

- képes a prioritások kezelésére
  - ez a sor kezelésétől függ, alapvetően FIFO , CPU – 100%
  - IT szinten MCU / CPU IT vezérlő
- kölcsönös kizárás az IT szinten és a részfeladatok között
- preemptivitás nincs!
- nem annyira törékeny
- a „függvény” nyelv konstrukció erős, jól szeparál
- megszakítást kezelő rutinok kielégítik az eszköz sürgős igényeit
- további feldolgozáshoz egy várakozási sorban elhelyeznek egy függvényre mutató pointert
- a várakozási sort a main-ben dolgozzuk fel
- meghatározott algoritmus szerint a sorból kivesszük a függvényre mutató pointereket
- lényegében ez is interrupt alapú
- - interrupt: kezeli a hardvert (időkritikus hardver közeli dolgokat)
- - majd önmaga egy függvényekre mutató pointereket tartalmazó sorba behelyezi a magas szintű (nem időkritikus) feladatokat
- - végtelen ciklusban: ha a sor nem üres, levesszük a függvényt (első függvény) és meghívjuk
- - a kölcsönös kizárást meg kell oldani az IT-kezelő és a magas szintű (nem időkritikus) feladatokat ellátó függvény között
- - mert elképzelhető, hogy miközben a magas szintű feladat fut, befut még egy interrupt
- - válaszidő = a leghosszabb task + az IT végrehajtása + a kezelő függvény végrehajtása
- - nem borul fel annyira könnyen az architektúra (csak azok a kódok futhatnak le, amelyek egy függvényben benne vannak)
- - a függvény szeparálja a taskokat

- - az új feladatok csak akkor tudják felborítani a rendszer működését, ha magasabb prioritásúak (a magasabb prioritású task egyébként minden architektúrában felborítja a működést)
- - nem preemptív: ha már fut egy kezelő függvény, akkor hiába érkezik be egy újabb kérés (és lefut az IT kezelő függvénye), csak akkor futhat le, ha az előző befejezte

Ütemezett függvények módszere

```
void interrupt A_Handler() { Handle_HW_A(); PutFunction(Service_A); }
void interrupt B_Handler() { Handle_HW_B(); PutFunction (Service_B); }
void interrupt C_Handler() { Handle_HW_C(); PutFunction (Service_C); }
void Service_A();
void Service_B();
void Service_C();
void main() {
  while (TRUE){
     while (IsFunctionQueueEmpty()) ;
          CallFirstFromQueue();
          }
}
```



Kiemelés sorrendje lehet prioritásos is -> Hangolhatjuk a rendszert, akár az IT-vel kieg. ciklikus programszervezének a kiszolg idejét is elérhetjük, ami persze visszáesés, kisz. idő növekedés is lehet.

Coroutine (Cooperative OS): ez pontosan mi is?

- kooperatív feladatok leírására szolgál
- a szubrutin általánosítása
- megvalósítható OS-ben, nyelvi elemként, saját magunknak is megvalósítható
  (OS: fiber, nyelvi elem: coroutine)
- stack alapú nyelvekben (C, C++) nehezen implementálható, de pl. FreeEcos támogatja a coroutine-t
- első belépési pontnál úgy működik, mint függvény v. szubrutin
- további belépéseknél: utolsó kilépési pontnál folytatja
  - kilépés: yield to coroutine
  - mintha lenne egy függvény több returnnel, és a a függvény újbóli meghívásakor az előző return utánról indulunk

```
var q := new queue;
coroutine produce {
loop while q is not full {
        create new item;
        add item to queue;}
        yield to consume;}

coroutine consume {
loop while q is not empty {
```

```
remove item from q;
use item;
}
yield to produce;
}
```

Operációs rendszerek (konkurens programozás):

Process (folyamat) és Thread (szál)

Process:

- saját stack és munkaterület (memória)
- komplex folyamatleíró: létrehozása és megszüntetése bonyolult és erőforrásigényes
- nem férhet hozzá más folyamatok munkaterületéhez
    - memóriavédelem hardver támogatással(MMU van a CPU-ban)
    - kommunikáció az OS-en keresztül történik
- Linux / Windows alkalmazás: Linux daemon(fork) vagy Windows Service(CreateProcess)
- Beágyazott rendszerek: Windows CE vagy XP Embedded, RT LINUX, Vx Works, QNX

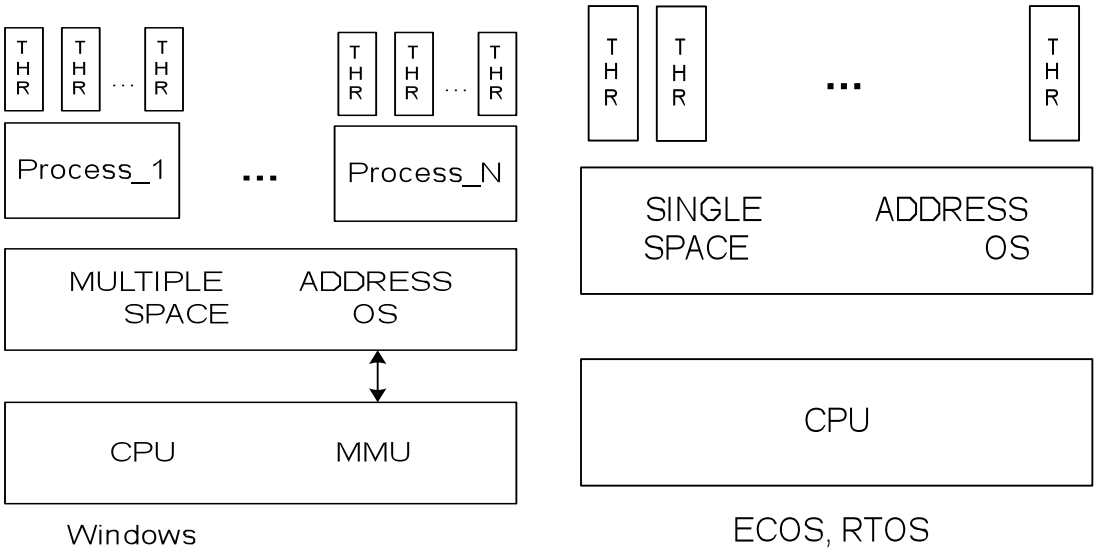
Thread / szál (pehelysúlyú folyamat) fordítunk:

Általában minden program több szálból áll. Minden egyes szál egy részprogramként értendő, s ezeket külön futtatja az OS. (Szálakba szervezzük a részfeladatokat.)

- párhuzamosan futó részfeladatok kommunikálni tudnak
- spec függvényből hozható létre, az őket tartalmazó folyamat munkaterületén futnak
- szálaknak saját stack-jük van
- egyszerű „szálleíró”
    - alacsony erőforrású a létrehozásuk és megszüntetésük
    - definíció szerűen hozzáférhetnek a saját folyamatuk munkaterületéhez
        - a saját folyamatukban létrehozott szálakkal memórián keresztül kommunikálhatnak
        - kis erőforrásigény és késleltetés
    - más folyamatokkal csak OS hívásokon keresztül kommunikálhatnak
    - OS támogatás (jellemző) vagy thread library (pthreadunkat OS a mi folyamatunkat ütemezi, a szálakat pedig más fogja ütemezni)

Megvalósítás:

- **Windows és linux (esetleg beágyazott rendszerek MMU-val):**
    - **alkalmazás, service, daemon (Linux)= folyamat (folyamatokon belül futhatnak szálak)**
- **beágyazott rendszerek MMU nélkül:**
    - **csak szálakat tudunk létrehozni**



11. Szoftverfejlesztés módszerei és módszertana, a modell szerepe. Az UML nyelv és helye a szoftverfejlesztés folyamatában.

Módszerek és módszertanok (method, methodology)

Módszer:

→ A szoftverfejlesztés során követendő szabályozott lépések sorozata, amely segítségével a fejlesztés alatt álló szoftver különböző aspektusait (azoknak a modelljét) írjuk le.
→ Jellegzetesen két részük van:
- **Folyamat**: lépések sorozata (túlzott szabadság korlátozása, minőségbiztosítása, határidők betartása)

- **Jelrendszer**: a fejlesztés alatt álló szoftver különböző aspektusait leíró jelölésrendszer (Cél: a szoftveren dolgozó emberek együtt tudjanak dolgozni, közös jelölésrendszernél mindenki ugyanazt gondolja, ha dobozt lát)
- **Implementációs szint**en maga a kód is ilyen
- A kód a fejlesztés korai fázisában nem alkalmazható jelölésrendszerként

Módszertan:

→ Azonos elvek alapján dolgozó módszerek (hasonlóak)
→ A fejlesztési modellek?? a fejlesztési folyamat átfogó, koncepcionális modelljét írják le
→ Pl.: ROPES (Rapid Object-based Process for Embedded Systems), Rational Unified Process
→ **Módszertan**:

- objektum orientált elemzés és tervezés, valamint megvalósítás
- modell-alapú fejlesztés (model driven architecture, model based design)

→ **Modell**:

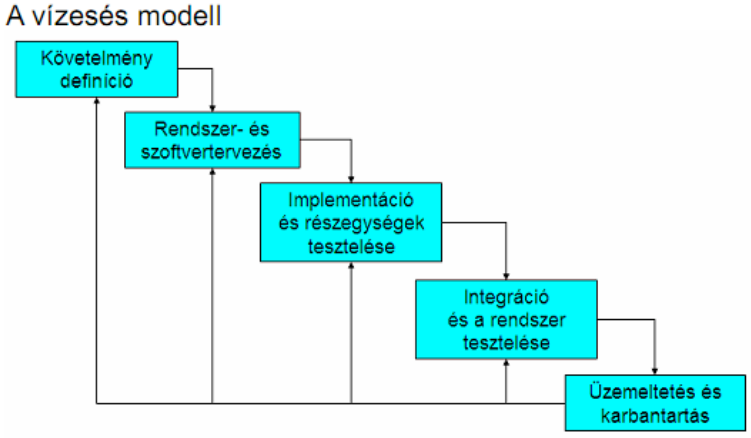
- a valós világról alkotott (többnyire annak egy jól meghatározott részéről) egyszerűsített konstrukció, amely elégséges információt hordoz a vizsgálat tárgyát képező dolog adott szempontból történő részleges megértéséhez.
- Az ember maga is modellt alkot a világról, és az alapján él.

SW gyártási modellek:

Vízesés modell: *Élesen elkülönülő specifikációs és fejlesztési fázisok.*
Evolúciós fejlesztés*: A specifikáció, fejlesztés és validáció átlapolódik.*
Komponens alapú fejlesztés: *A rendszert kész komponensekből állítjuk össze.*
Újrafelhasználás alapú fejlesztés
Iteratív SW gyártás
Inkrementális programozás
Extrém programozás

SW-s fejlesztés: SW életciklus modell /vízesés modell/:

- Elemzés /MIT?/
- Tervezés /HOGYAN?/
- Implementálás
- Üzemeltetés

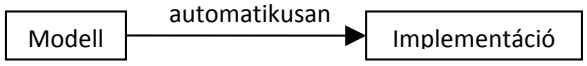


Miért fontos ez a SW szempontjából?

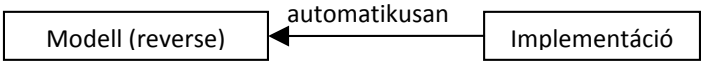
→ A probléma megoldása során így is-úgyis modelleket alkotunk, akkor miért ne tegyük ezt a folyamat részébe?
→ Komplex rendszerek nem hozhatóak létre kód szintű eszközökkel (hatékonyan).
→ Az absztrakt (implementációhoz direkt módon nem köthető) modellek lehetővé teszik a fontos tulajdonságok és azok viszonyának leírását.

Mit tesz lehetővé a modell alapú megközelítés?

→ Modellből kód generálása:

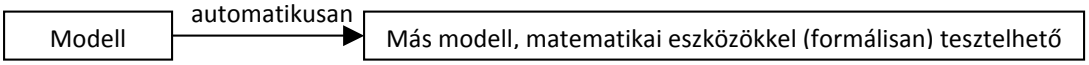


→ Modell ↔ kód kétirányú átjárhatósága



- Hibák:
  - A kód elválhat a modelltől
  - A folyamat még bonyolultabb lesz (ettől)

→ Végrehajtható modellek (korai tesztelés)
→ Modell automatikus ellenőrzése



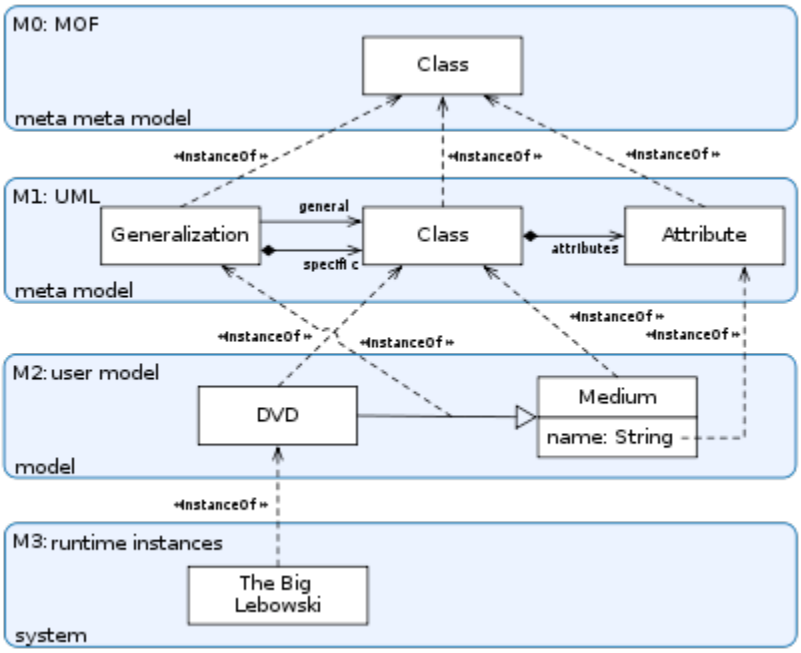
→ Utóbbi 2 elősegíti az elemzést a tervezés során elkövetett hibák gyors felderítésére.

Szükségünk van egy közös jelrendszerre: **UML**.

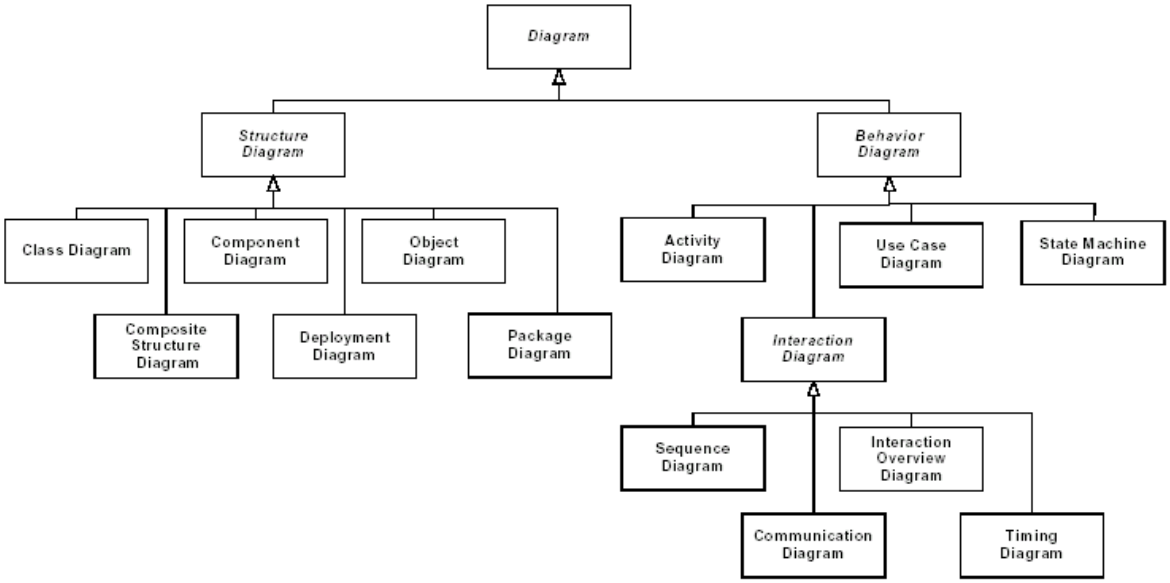
UML az oop szabványos specifikálciós nyelve.

UML – Unified Modeling Language:

Szöveges, grafikus modellek készítése szoftverekről, programokról, adatbázisokról, rendszerekről, szervezetekről, szereplőkről, üzleti tevékenységekről, folyamatokról, logikai összetevőkről



→ Nem mond semmit a folyamatról!
→ 80s, 90s.
→ OMG (97-Object Managment Group, www.omg.org) szabvány (széleskörű ipari támogatás)
→ Folyamatos fejlesztés alatt áll
  - Jelenleg UML 2.2 (2.3-on dolgoznak, 1.x → 2.x nagy változás)
→ Az UML nem csak jelrendszer, hanem tartalmaz egy úgynevezett meta-modellt is.
→ **Meta-modell**: egy olyan modell, ami más modellek leírására szolgál.
→ Az UML le tudja írni önmagát:
  - Jó hír: jó kifejezőképességű
  - A meta-modell nem formális, klasszikus matematikai módszerekkel a helyessége nem ellenőrizhető.
→ *Ajánlott könyv: Real Time UML Third Edition – Advanced int he IML for real-time systems, Bruce Powel Dougless*
→ *Wikipedián egy jó bevezető van az UML-hez*
→ *UML Profile-ok kibővítések és kiegészítések az UML-hez*
  - *SysML (System Modifying Language)*



Deployment : felfejlődés- diagram

Composite structure diagram: vegyes struktúra diagram

12. UML diagrammok, használati-eset diagramm és osztálydiagramok, alkalmazási köreik.

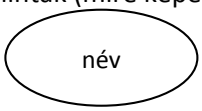
Use-Case Diagram:

→ Célja: A rendszertől vagy alrendszertől elvárt **viselkedési minták leírása. A követelményrögzítés** (a szoftverfejlesztés első fázisaiban, pl. a követelmény-definíciós fázisban használatos).
→ Funkcionális diagram**: középpontban a rendszer által végrehajtandó funkciók vannak.** A rendszer funkcióinak komplett leírására szolgál.
→ Megmondja, hogy
- ¨ **mit kell tudnia a rendszernek,**
- ¨ **milyen funkciói legyenek a megtervezett rendszernek.**

→ Tulajdonságai:
- ¨ szemléletes,
- könnyen áttekinthető.

→ **Részei:**
- **Rendszer (system)**: amit el akarunk készíteni

- **Használati eset:** elvárt viselkedési minták (mire képes a rendszer)

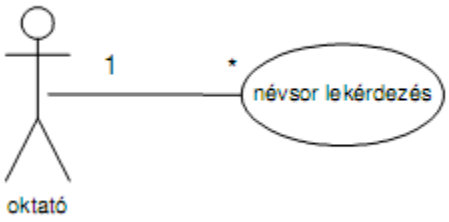


- **Aktor (actor)**
  - Ember a jele, de nem feltétlenül (automatikusan) egy humán aktorról van szó, lehet a rendszeren kívüli más rendszer is (gép-gép kapcsolat)
  - egy szerepkört reprezentál.
  - Környezet: a világ, ami a rendszert körülveszi
  - aktorok
    - A felhasználók akik a rendszert használják
- **Összefüggések (relations)**

  Kapcsolatot jelenti az előbbiek között
  - Szimpla vonal
  - Általános összefüggések

1. **Asszociáció (association):**

jele folytonos vonal
használati eset és aktor között
jelölhető a számossága
használati eset és aktor között:

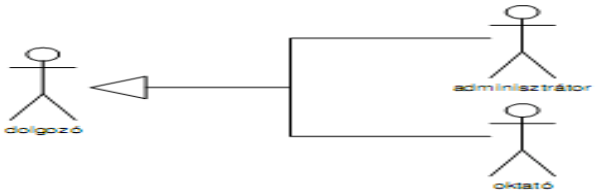


Általánosítás (generalization):

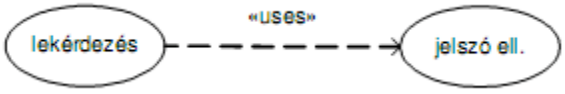
2.a: ***használati eset és használati eset között:***

oo

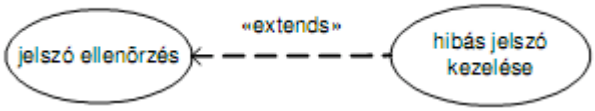
2.b: ***aktor és aktor között:***



2. **Include: <<include>>>** :(1.5 UML szabványban ez szerepel) vagy **<<uses>>**
   Két használati eset között áll fent, ha az egyik magában foglalja a másikat. (az egyik használati eset használja, és mindig használja a másikat)

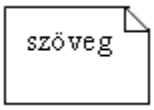


3. **Kiterjesztés: <<extend>>**
   kibvítés (kivételkezelés, hibakezelés)
   az egyik használati eset mködését kiegészíti egy másik használati eset. (



   a „jelszóellenőrzés” használati esetet kibővíthetjük egy olyan funkcióval, amely lekezeli azt, ha a felhasználó hibás jelszót ad meg

o. Megjegyzés (note)



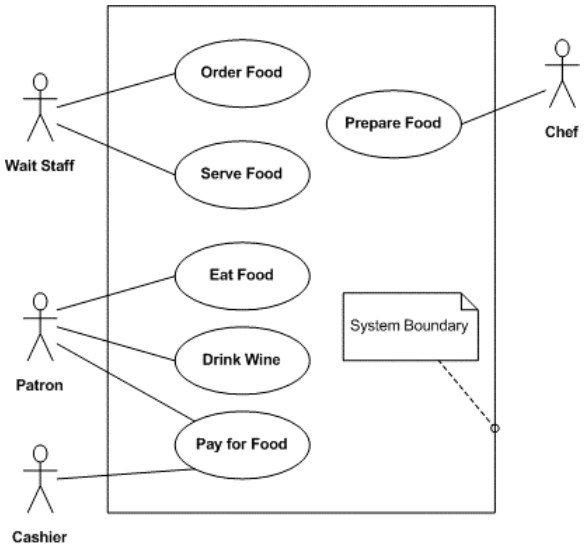
o.Kényszer (constrmint) – {szöveg}

o.Rendszerhatár

o.Alrendszerhatár

<<subsystem>>

Példa Use-case diagramra:

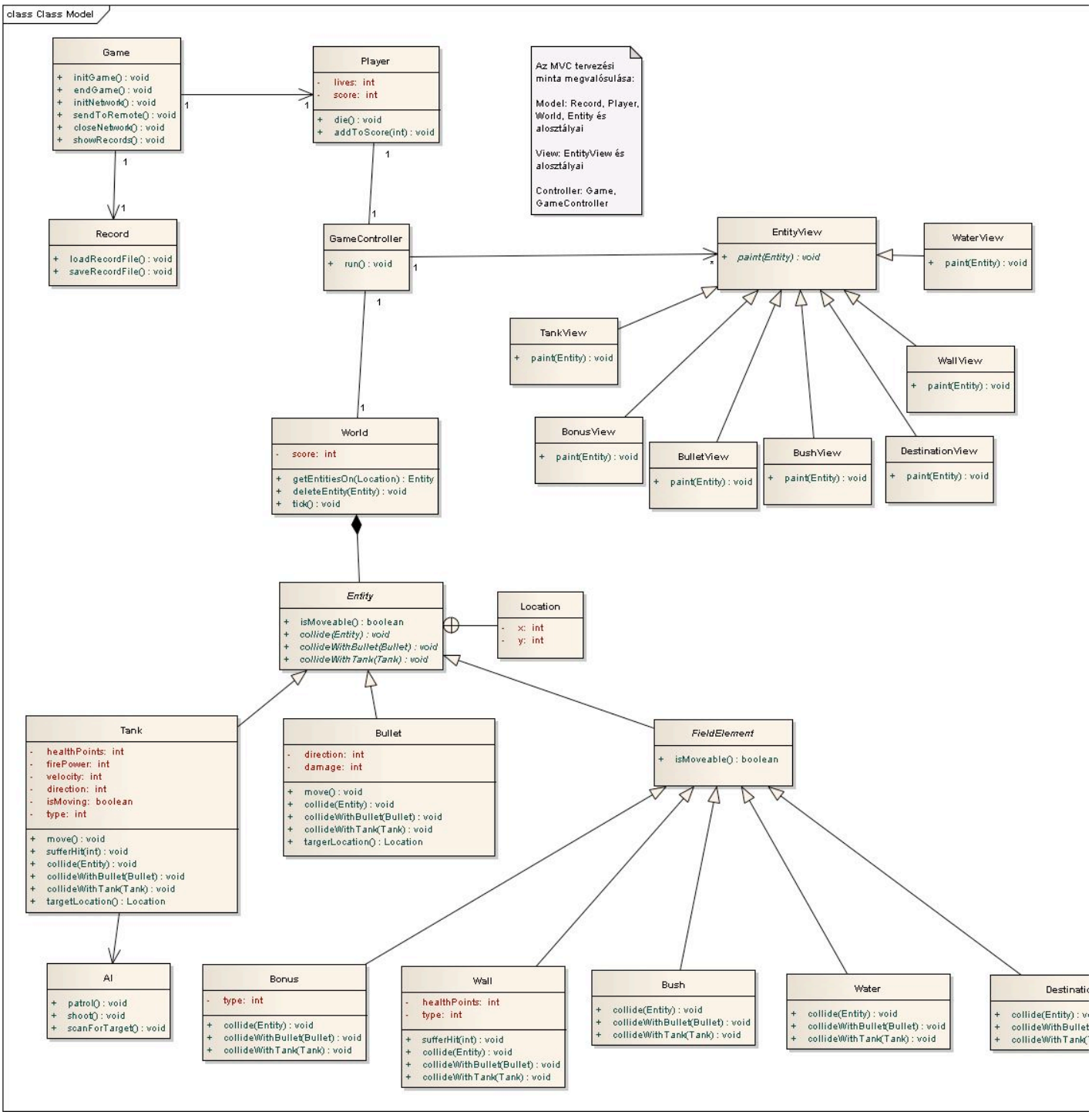


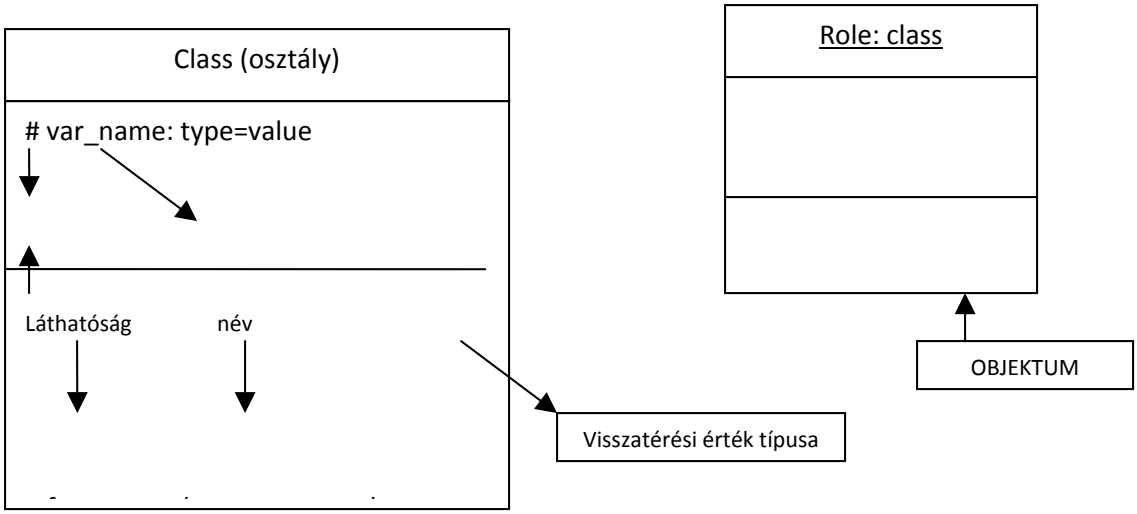
Nem túl jól használható (kódot nem generálhatunk belőle), DE szemléletes!
Az elemzés során készül, és az elemzés és a tervezés során használjuk fel.

Osztálydiagram:

→ Osztályok, objektumok és azok kapcsolatainak a leírására szolgál
→ A fejlesztési folyamat tetszőleges szakaszában használható.

- Elemzés: nem feltétlenül SW
  - A környezet leírására
  - A SW-rel szemben megfogalmazott elvárások leírására
- Tervezés, megvalósítás, stb.
- OO nyelvekre kód generálható belőle, OO kódból → UML diagram


class Class Model
Game
+ initGame() : void
+ endGame() : void
+ initNetwork() : void
+ sendToRemote() : void
+ closeNetwork() : void
+ showRecords() : void
Player
- lives: int
- score: int
+ die() : void
+ addToScore(int) : void
Az MVC tervezési minta megvalósulása:
Model: Record, Player, World, Entity és alosztályai
View: EntityView és alosztályai
Controller: Game, GameController
Record
+ loadRecordFile() : void
+ saveRecordFile() : void
GameController
+ run() : void
EntityView
+ paint(Entity) : void
WaterView
+ paint(Entity) : void
TankView
+ paint(Entity) : void
WallView
+ paint(Entity) : void
BonusView
+ paint(Entity) : void
BulletView
+ paint(Entity) : void
BushView
+ paint(Entity) : void
DestinationView
+ paint(Entity) : void
World
- score: int
+ getEntitiesOn(Location) : Entity
+ deleteEntity(Entity) : void
+ tick() : void
Entity
+ isMoveable() : boolean
+ collide(Entity) : void
+ collideWithBullet(Bullet) : void
+ collideWithTank(Tank) : void
Location
- x: int
- y: int
Tank
- healthPoints: int
- firePower: int
- velocity: int
- direction: int
- isMoving: boolean
- type: int
+ move() : void
+ sufferHit(int) : void
+ collide(Entity) : void
+ collideWithBullet(Bullet) : void
+ collideWithTank(Tank) : void
+ targetLocation() : Location
Bullet
- direction: int
- damage: int
+ move() : void
+ collide(Entity) : void
+ collideWithBullet(Bullet) : void
+ collideWithTank(Tank) : void
+ targerLocation() : Location
FieldElement
+ isMoveable() : boolean
AI
+ patrol() : void
+ shoot() : void
+ scanForTarget() : void
Bonus
- type: int
+ collide(Entity) : void
+ collideWithBullet(Bullet) : void
+ collideWithTank(Tank) : void
Wall
- healthPoints: int
- type: int
+ sufferHit(int) : void
+ collide(Entity) : void
+ collideWithBullet(Bullet) : void
+ collideWithTank(Tank) : void
Bush
+ collide(Entity) : void
+ collideWithBullet(Bullet) : void
+ collideWithTank(Tank) : void
Water
+ collide(Entity) : void
+ collideWithBullet(Bullet) : void
+ collideWithTank(Tank) : void

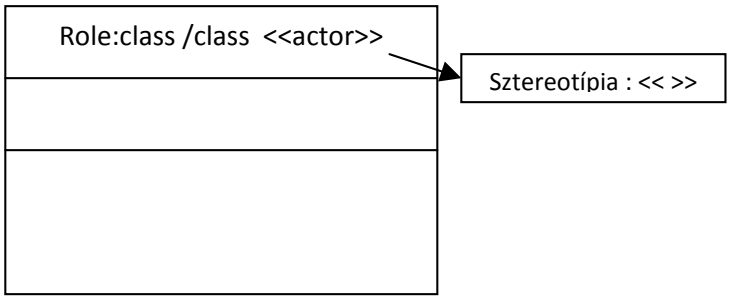


→ Láthatóság (Visibility):
- +: public
- -: private
- #: protected
- ~: package

→ #var_name:type=value – az aláhúzás jelentése, hogy osztályváltozó

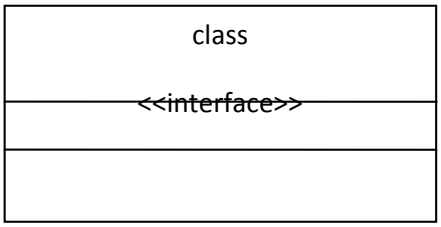
→ Objektumok a diagramban: a program futása közben egy adott időpillanatban rögzített kép (snapshot) található az ábrán.

→ Aktorok az osztálydiagramban:
- Az aktort nem kell megírni, az a rendszeren kívül van, kivéve a tesztelést

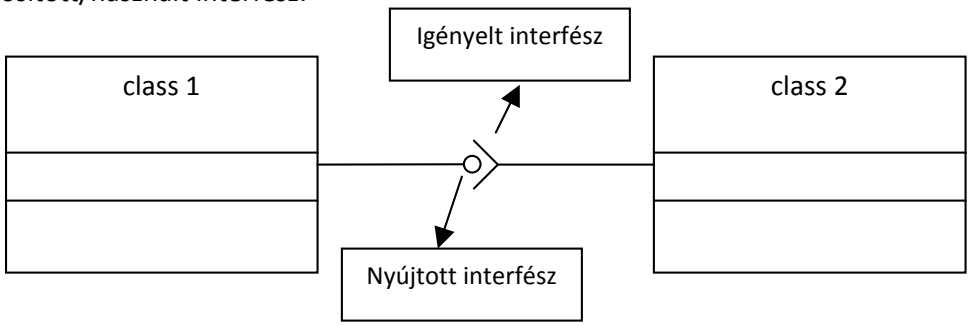


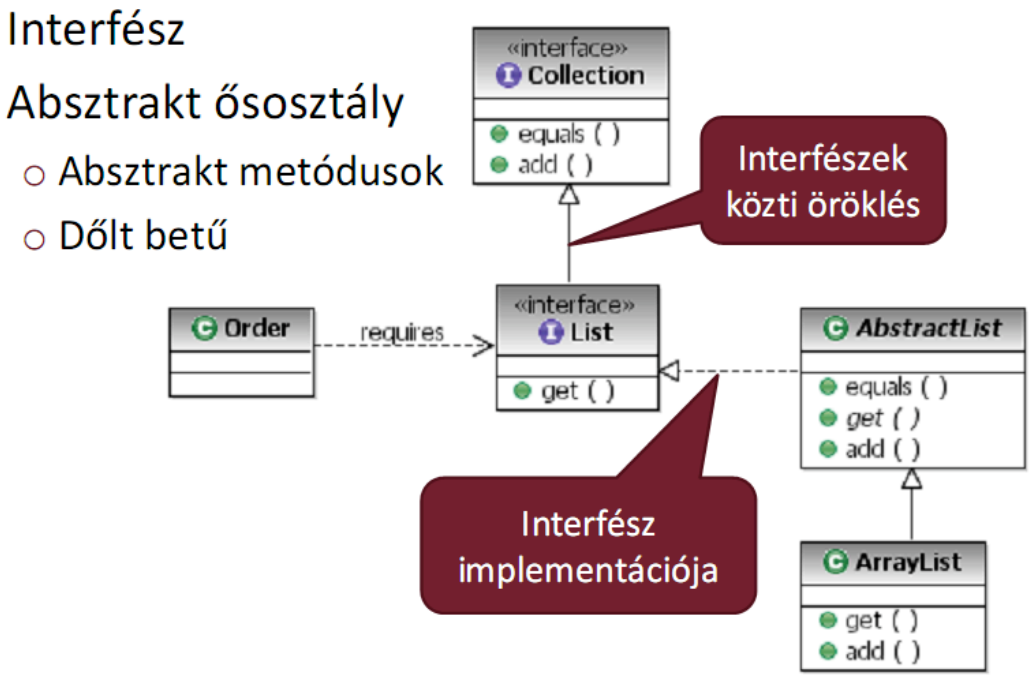
→ **Interfészek**

- Interfész osztályok:



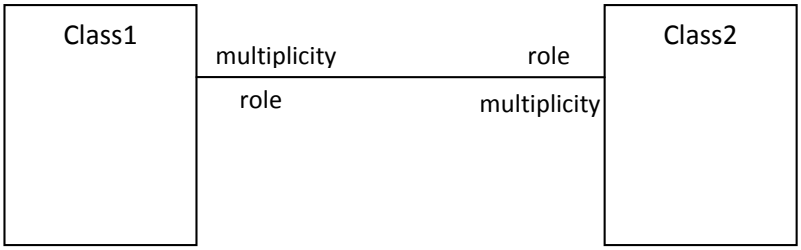
- Megvalósított/használt interfész:





→ **Társítás (association)**

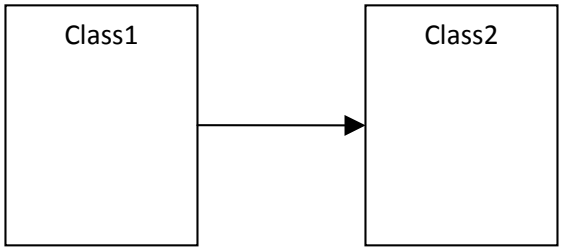
- Kétirányú (postás és kutya)



- Multiplicity = számosság
- Role = szerep
- Leggyakrabban használt számosságok:
  - Fix számok: 3
  - Lista: 0,2,3
  - Tartomány: 0…10
  - 0 vagy több: *
  - 1 vagy több: 1...n

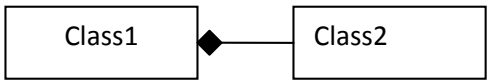
Ha csak egyirányú akkor is lehet mindkét oldalon számosság?? Ez mit jelent?

- Egyirányú társítás (ha szükséges) – *Tankos játékban nem tömör nyílfejjel, ilyen csak öröklésekor volt*

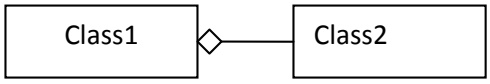


→ **<u>Tartalmazás (aggregation) –</u>** *Tankos játékban ref. Sz-i volt talán, egy kör és benne kereszt*

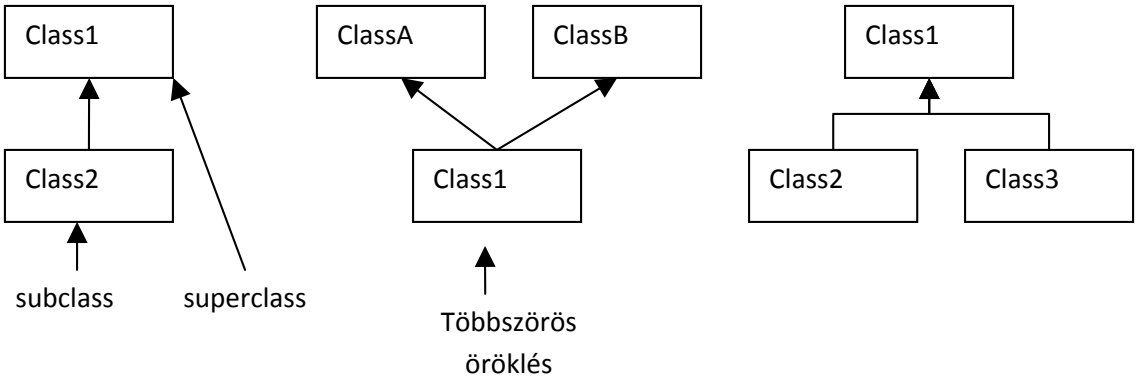
- Érték szerinti – Class1 tartalmazza Class 2-t.



- Referencia szerinti



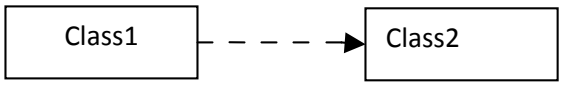
→ **Öröklés (inheritance)**



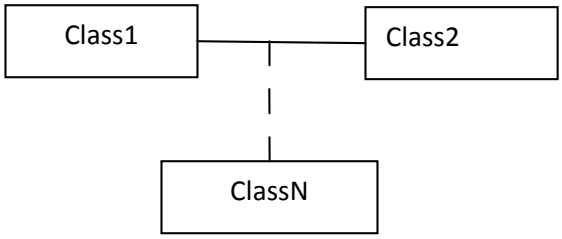
- Több pontosított változat esetén elágazást alkalmazunk, és az elnevezés a csomóponthoz kerül. –lenti ábra balra- Férfi - Nő

→ **Függés(Class 1 függ Class 2-től?)**

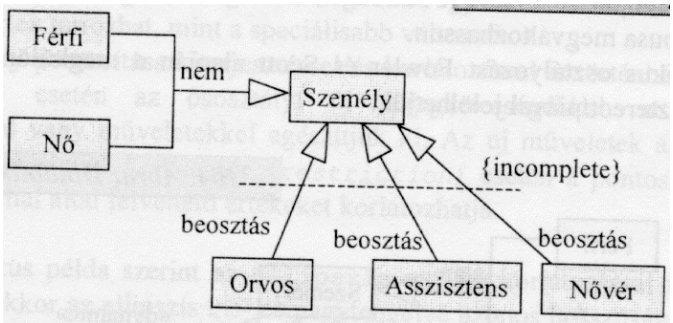
- Laza kapcsolat, ami a többivel nem írható le, például nem működik a hőmérő(class 1) I^2C (Class 2)nélkül

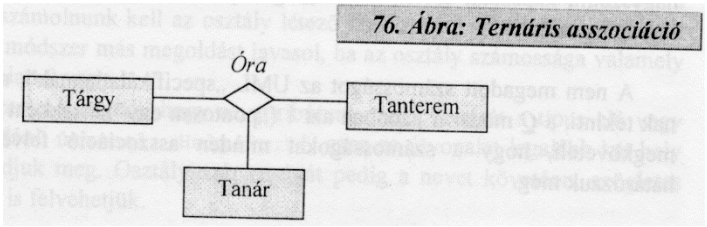


→ **Társítás (Asszociáció)**



- ClassN: például Hash tömb, amelyen keresztül teremtődik meg a kapcsolat, mint pl itt?



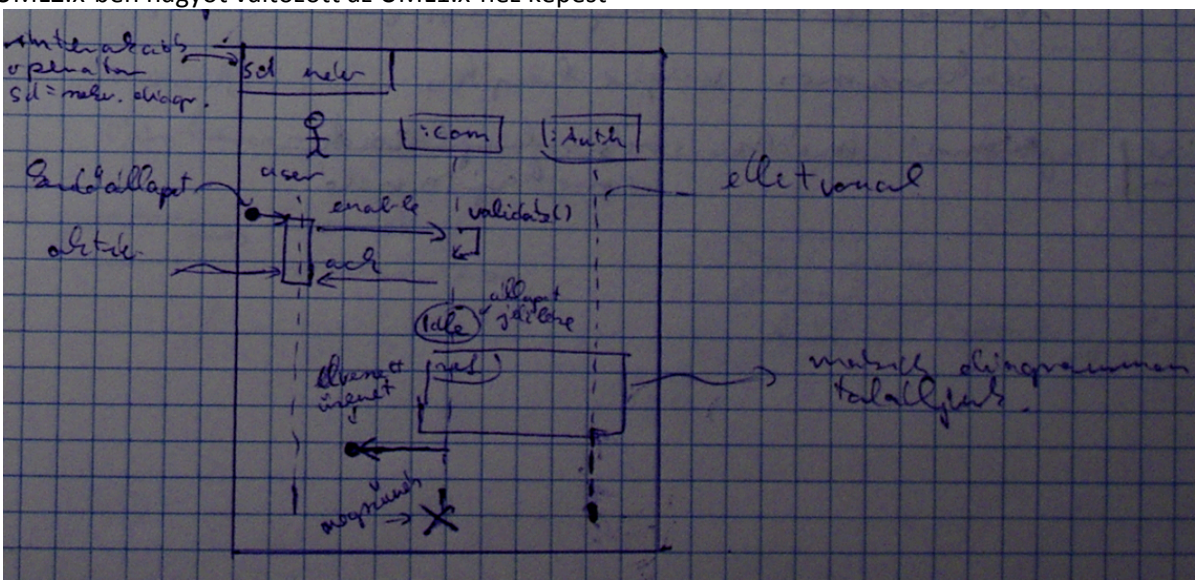


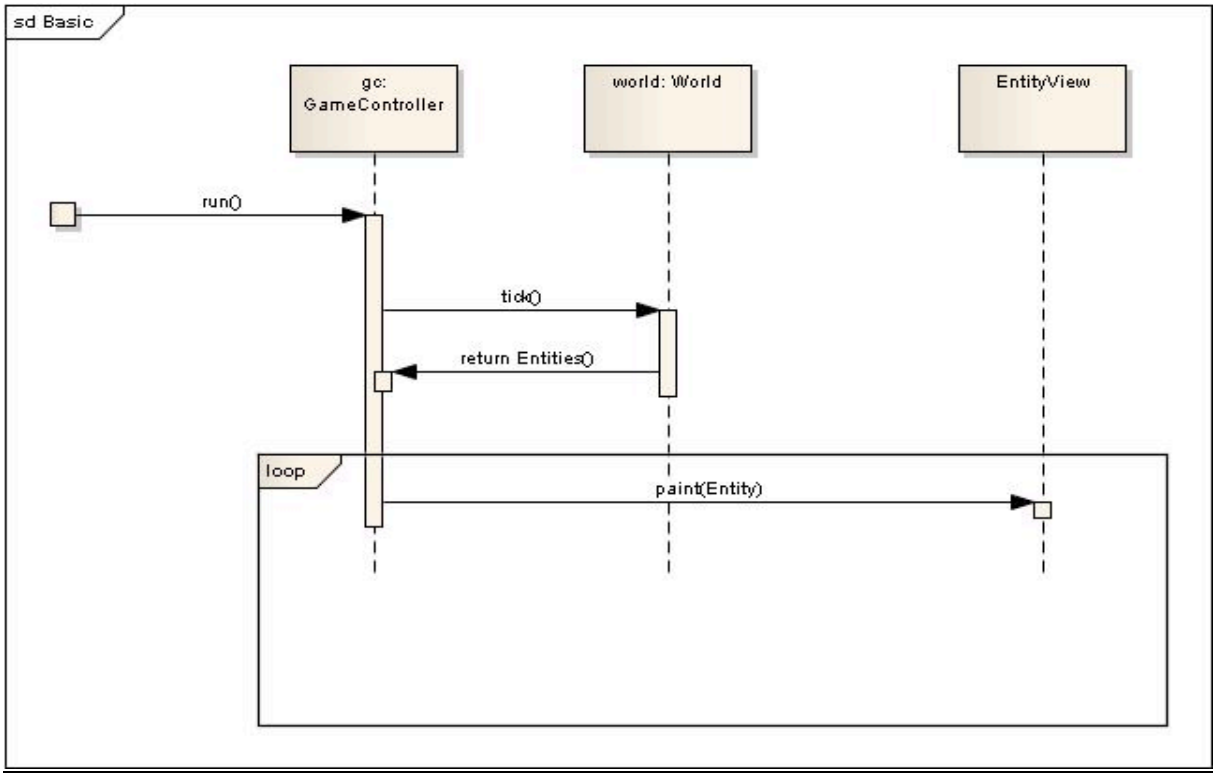
13. UML diagrammok, szekvencia és aktivitás diagrammok, alkalmazási köreik.

Szekvenciadiagram:

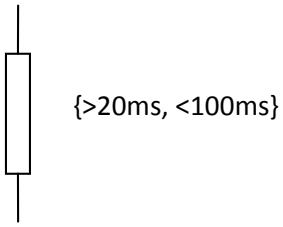
→ Külső viselkedést ír le osztályok vagy objektumok között
→ Scenáriók, viselkedés darabkák (fragments) leírására szolgál
- Pozitívak (így működik) vagy negatívak (nem így működik)
- A scenáriók sokasága írja le a teljes viselkedést

→ UML2.x-ben nagyot változott az UML1.x-hez képest





→ Egymásutániságot ad meg, nem idődimenziójú a függőleges tengely
→ Időkényszerek megadhatóak:



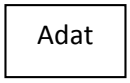
→ Instrukciós operátorok
- **sd** – egyszerű szekvenciadiagram
- **ref** – más ábrán részletesen megadott szekvenciadiagram (hierarchikus ábrák)
- **alt** – alternatíva, az alternatívákból csak egy kerülhet végrehajtásra (pl. switch)
- **opt** – vagy megtörténik, vagy nem
- **break** – alt eseteknél használjuk (C++ break utasításához hasonló)
- **loop** – ciklus
- **seq/struct (laza/szoros) sorrendiség** – adott dolgok tetszőleges sorrendben végrehajthatóak
- **neg** – negatív követelmények megfogalmazása
- **par** – párhuzamos végrehajtás
- **critical region** – atomi módon végrehajtandó

→ Szekvencia diagram alkalmazása:
- A fejlesztési folyamat tetszőleges szakaszában alkalmazható
- Tesztek generálására alkalmas

<u>Aktivitás diagram</u>

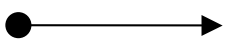
→ Lényegében egy kibővített, jól definiált folyamatábra
→ Működési „token flow” szemantikán alapul
- A token egy adott állapotban van, állapot tartalmazza a tokent
- A token áramlik a rendszerben
- „Petri hálók”

→ Objektumok belső viselkedésének a leírására szolgál
- Adat

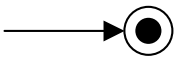


- Állapot

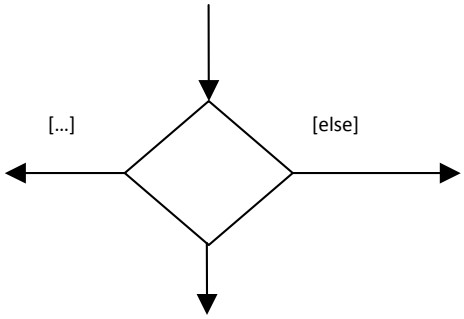
- Kezdő állapot



- Végállapot

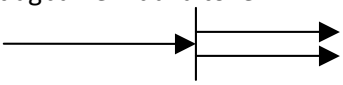


- Elágazás

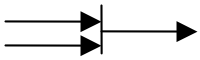


Kérdés: több igaz van?

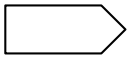
- Fork (elágazás) – mindkét ágban elindul a token



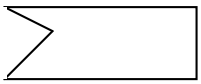
- Join (randevú)



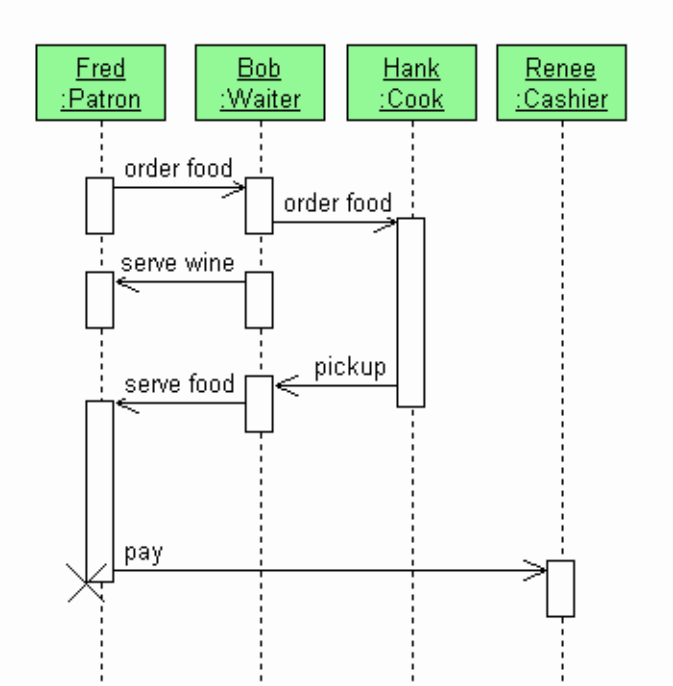
- Esemény generálása



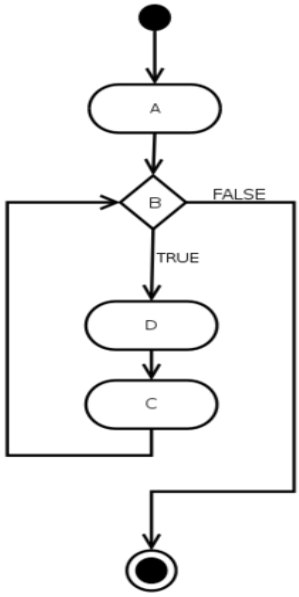
- Külső esemény fogadása



→ UML2.x-ben jelentősen kibővült
→ A fejlesztési folyamat tetszőleges szakaszán használható
→ Lehetne belőle kódot generálni, de többnyire nem szoktak
→ Az állapotdiagram elterjedtebb
Példa egy szekvencia és egy aktivitás diagramra:


Fred
:Patron
Bob
:Waiter
Hank
:Cook
Renee
:Cashier
order food
order food
serve wine
pickup
serve food
pay

```
for(A;B;C)
  D;
```


A
B
FALSE
TRUE
D
C

14. UML diagrammok, állapotdiagram, alkalmazási köre, implementációs lehetősége.

UML állapotdiagram (statechart, statediagram)

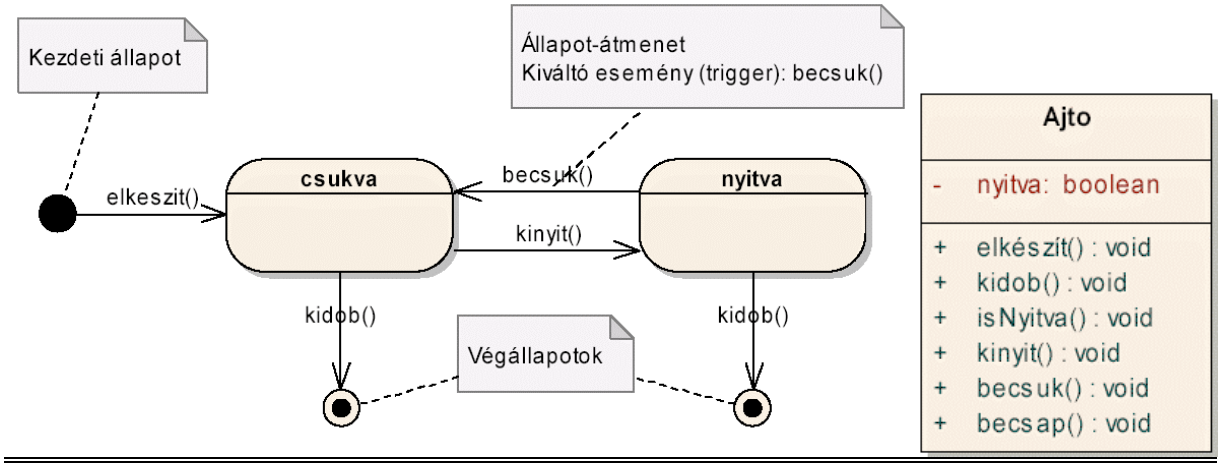
- Az állapot diagram (state diagram) egyetlen osztály (annak egy elő fordulásának) dinamikus viselkedését, a külvilággal való kapcsolatát ábrázolja.
- Az állapot diagram egy gráf, melynek csomó pontjai állapotok, élei pedig átmenetek.
- Megadja, hogy az objektum mely események hatására milyen állapotból milyen állapotba kerül.
- Ha egy objektumnak intenzív a dinamikus működése, akkor az objektum viselkedésének feltárásában az állapot-átmeneti diagram igen hatékony eszköz.
- Az állapot jele az UML-ben egy lekerekített téglalap, melynek részei (bármelyik rész elhagyható):
    - Név: a téglalap felső része, az állapot neve
    - Belső átmenetek: Olyan akciókat, illetve aktivitásokat tartalmazhat, melyek nem okoznak állapotváltozást.

Az állapot-átmenet lehetséges tulajdonságai:

- Kiváltó esemény (trigger vagy event).
    - Az átmenet a kiváltó esemény hatására következik be.
    - Az esemény egyaránt jöhet kívülről vagy belülről.
- Őrfeltétel (guard).
    - Egy logikai kifejezés, amely hivatkozhat az objektum adataira.
    - Az átmenet csak akkor következik be, ha az őrfeltétel igaz.
- Akció (action).
    - Átmenetkor végrehajtódik.
- E tulajdonságokat az átmenet vonalán tüntetjük fel (bármelyik elhagyható):

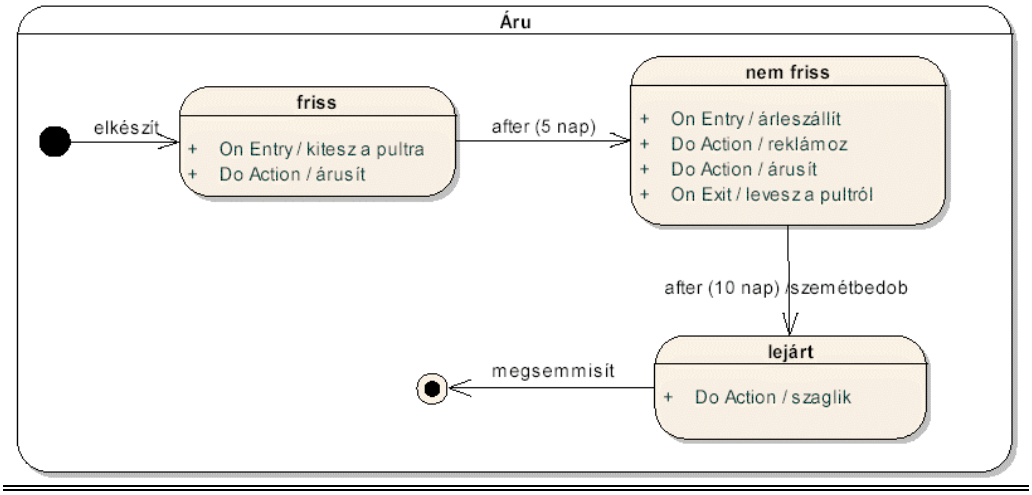
  trigger [őrfeltétel] / akció
- **Az objektumnak van**

- pontosan egy kezdeti állapota (initial state), melybe az objektum születésekor kerül. Jele egy tömör kör.
- valahány végállapota (final state), ahonnan további állapotváltozás nem lehetséges. Jele egy ökörszem.



Tevékenységek az adott állapotban

- Vannak események, melyek az objektumon nem okoznak közvetlen állapotváltozást.
- Ezeket az eseményeket az állapotdoboz akciórészébe(belső átmenetek részbe) írjuk a következő formában:
  Akciócímke / Akció-kifejezés (Action-list)
- Akciócímkék:
  - On Entry: az akció az állapotba való belépéskor kerül végrehajtásra; On Entry / Akció-kifejezés
  - On Exit: az akció az állapotból való kilépéskor kerül végrehajtásra; On Exit / Akció-kifejezés
  - Do Action: tevékenység, mely az adott állapot fennállása alatt folyamatosan fut; Do Action/ Akció-kifejezés

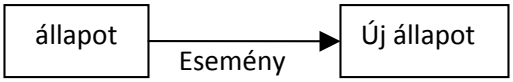


Kapcsolat a forráskóddal

- Ha az állapotváltozások figyelésénél kiderül, hogy hiányzik egy metódus, akkor az osztályt természetesen bővítjük.
- Az állapotdiagram nagy segítsget jelent az osztály adattagjainak és metódusainak meghatározásában, valamint az egyes metódusok kódolásában.
- A kiváltó esemény (trigger) lekezelőjének forráskódjában a következő tevékenységek szerepelnek, ebben a sorrendben:
  - az esemény forrásállapotának On Exit tevékenységei;
  - a trigger kísérő eseménye;

a célállapot On Entry tevékenységei;

→ Harel féle statechart az alapja
→ Osztály belső viselkedésének leírására alkalmas
→ a klasszikus véges állapotgép általánosítása
→ nagyon sikeres, különösen beágyazott rendszerekben
  - a beágyazott rendszerek reaktív rendszerek, az őket ért változásokra reagálnak



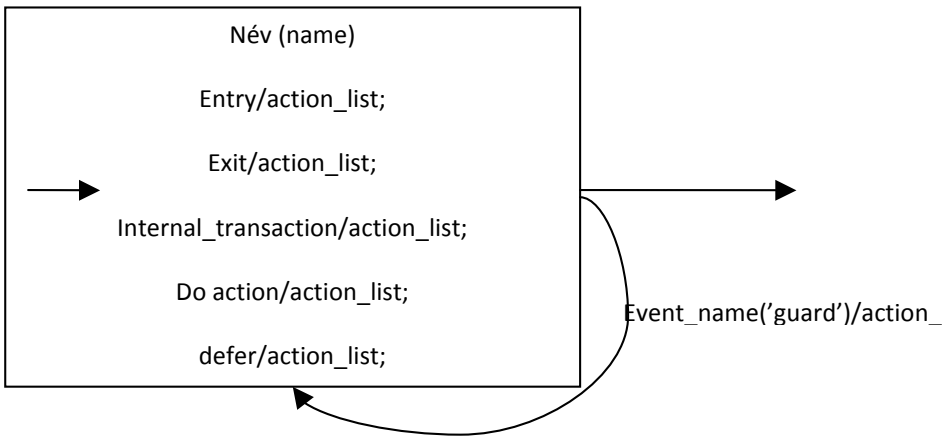
→ Az állapotdiagram főbb elemei:

- Állapot (state)
  - Részei:
    - Név
    - Belső átmenetek (nem okoznak állapotváltozást)
- Állapot átmenet (transition)
- Pszeudó-állapot (pseudostate)
  - Itt a rendszer tranziens jelleggel tartózkodik
  - Az állapot átmenetek kedvezőbb/áttekinthetőbb leírását tesznek lehetővé
- Esemény (event)

→ Az állapotgép egy irányított gráf

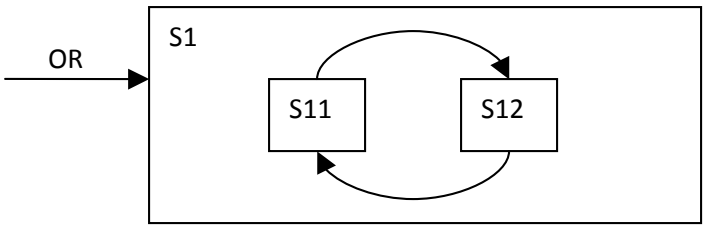
- Állapot és pszeudó-állapot: csomópont
- Állapotátmenet: él
  - Mindig van rajta nyíl (van iránya), hiszen a gráf irányított

<u>Az állapotátmenetek megadása</u>



- Entry – belépéskor
- Exit – kilépéskor
- Internal_transaction – belső átmenet
- Do – belül ciklusban végrehajtott
- Defer – azon események listája, amiket a kérdéses állapotban nem dolgozunk, hanem visszateszi a listába
- Az action_list-ek nem megszakíthatóan végtelen rövid idő alatt végrehajtódnak (nem lehet rekurzívan hívni, run to complition)

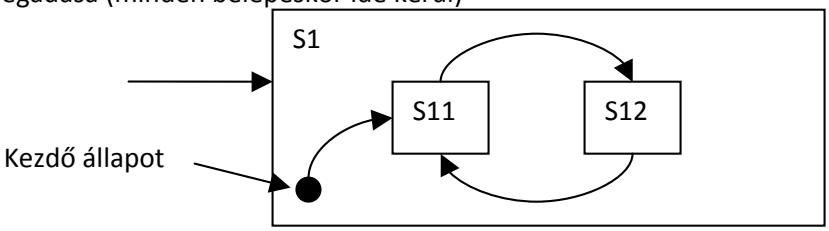
→ Internal_transaction: belső állapok vannak



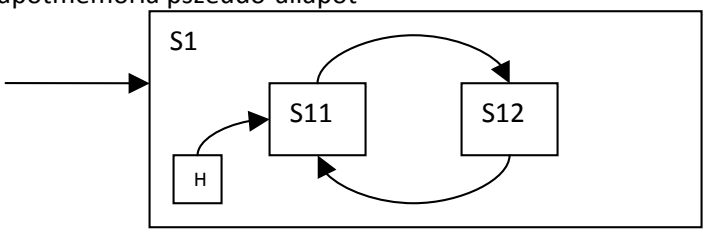
- Ha S1-ben vagyunk, akkor vagy S11-ben, vagy S12-ben vagyunk
- Valójában az állapot vagy S1:S11, vagy S1:S12
- Az állapotgép hierarchikus

Hierarchikus állapotgép

1. Ha S1-be belépünk, hogyan dől el, hogy S11-be vagy S12-be lépünk-e be?
   - Kezdőállapot megadása (minden belépéskor ide kerül)



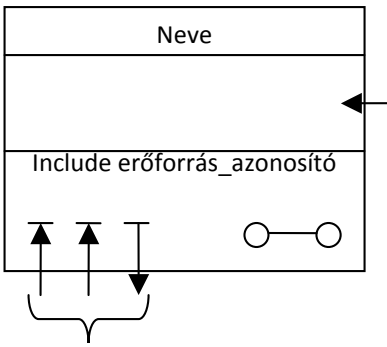
   - Sekély és mély állapotmemória pszeudó-állapot



     - Sekély: S11-re és S12-re igaz
     - H* a mély: a teljes hierarchiára működik az emlékezet
     - Ha szimpla H van: S11-gyel kezdünk, ha kilépünk megjegyezzük, hogy melyik állapotban voltunk, a következő belépéskor oda lépünk vissza
     - H* esetén nem csak ezen a szinten jegyezzük meg , hogy S12-ben voltunk, azt is megjegyezzük, hogy S12 melyik alállapotában voltunk, pl.: S121, S122 vagy S123

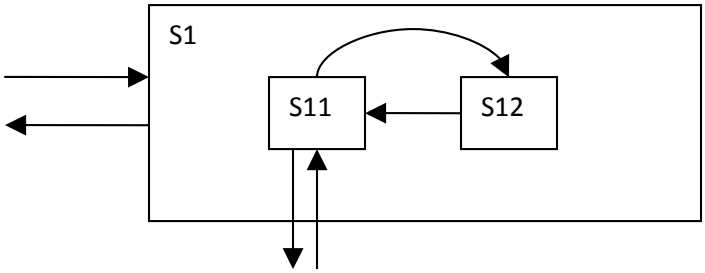
2. Végállapot (leállítás)

3. Külső file-ban megadott állapotgép részlet
   - Sokszor az állapotgép összetettsége miatt egy ábrára nem rajzolható le
   - Modularitás (állapotgép részletek újrafelhasználása)
   - Hivatkozás:



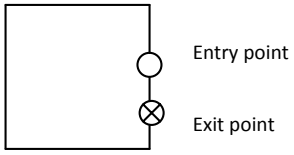
Hol találom meg az állapotgépet (pl. egy file)

Állapotgépen belülről induló vagy belülre mutató állapotátmenetek vannak



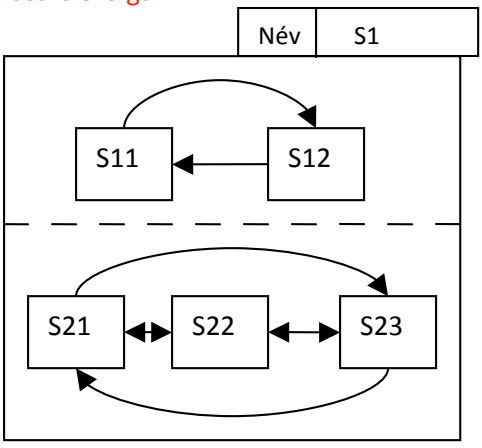
Ezt kell tudnia megadni

- Megadás:



AND állapot vagy ortogonális régiók

→ Konkurens aktív állapotok leírására szolgál



- Ha S1-ben vagyunk, akkor:
  - S11:S21 vagy S11:S22 vagy S11:S23, vagy

        - S12:S21 vagy S12:S22 vagy S12: S23 állapotokban vagyunk
    - Így 6 helyett 5 részállapot
    - a rendszert ortogonálisan szétbontva könyebben tudjuk kezelni
    - Pl. S11 módosításánál nem kell 3 állapotot módosítani, elég csak egyet

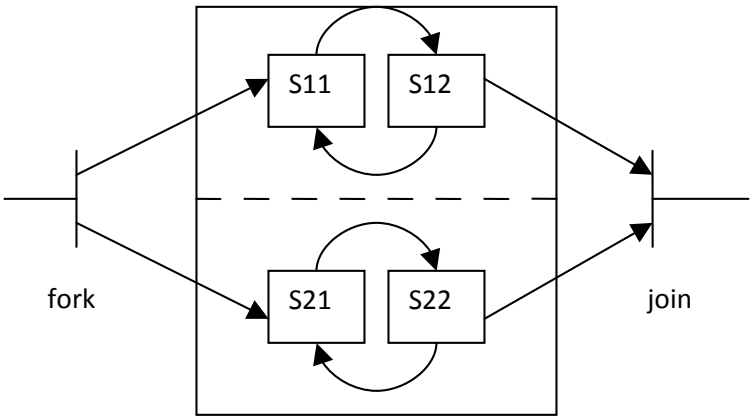
→ A megkapott eseményeket az AND állapotok külön-külön megkapják

- Pl. gomb megnyomására:
    - Háttérvilágítás bekapcsol
    - Menübe belép

→ AND állapot fontos elem, lényegében lehetővé teszi a konkurens, párhuzamos, eseményvezérelt programozás támogatását (tudni kell, hogy van, és hogyan működik ez az AND-es cucc vizsgára)

→ Hogyan lehet AND állapotokban működő konkurens állapotdiagram részleteket szinkronizálni?

- Join és fork pszeudó-állapotok



- Broadcast események
- Eseményküldéssel szinkronizálás
- is_IN operátor, ami igaz, ha a rendszer egy más, megadott AND részállapota az adott állapotban van

<u>Állapotdiagramok megvalósítása</u>

→ Miro Samek : „Practical Statecharts in C/C++”

1. állapotdiagram → véges automata → kód a target nyelvre, mindkét transzformáció automatikus
2. állapotdiagram → kód a target nyelvre (OO nyelvekre egyszerűbb, automatikus transzformáció)

→ 1.pont szerint 2 megoldás

- Beágyazott switch utasítás
- Állapottábla

→ <u>Beágyazott switch utasítás:</u>

- Állapot+esemény
- Régi állapotból egy esemény hatására új állapotba megyünk át
- Két switch szerkezet egymásba ágyazása
    - Switch állapot szerint
    - Egy bizonyos állapotban esemény szerint
    - Utasítás részben:
        - Új állapot beállítása

- Akciók végrehajtása

- Példa:

```
switch state

  case state 0

switch event

  case event 0

      state=new_state;

      action_list;

      break;

  case event N

break;

  case stateN
```

- Állapot és esemény tárolása ENUM-ként (felsorolás típus)

→ Állapottábla:

- Állapottábla (adatstruktúra) + SW modul annak a kezelésére (dispetcher)

	E0		EM
S0			
			Func_pointer +next_state
SN			null

- Func_pointer: action_list-et megvalósító függvényhez
- Next_state: hogy azt ne kelljen az action_list-en kezelni
- Null: ilyen nincs, az adott állapotban az adott eseményt nem kezeljük
- 2D tömb: a gyakorlatban ritka, mert sok NULL pointert tartalmaz, sok helyet foglal, és megvalósítási kérdéseket vet fel

→ Összehasonlítás:

Szempont	**Beágyazott switch**	**Állapottábla**
Kód	Generált kód	Kézzel írt kód (az állapottáblát generáljuk)
Kód méret	Függ az állapotdiagram komplexitásától	Nem függ az állapotdiagram komplexitásától
Kód újra felhasználhatóság	Nem használható fel újra	Újrafelhasználható
Adat memória	Kevés: állapot tárolása	Az állapotdiagram komplexitásától függ

Végrehajtási idő	Switch megvalósításától függ: - Feltételes utasítások sorozata - Ugrótábla - Stb.	Nem függ az állapotdiagram komplexitásától (indexelés a feladat)