13. előadás – 04. 22.

Megoldások:

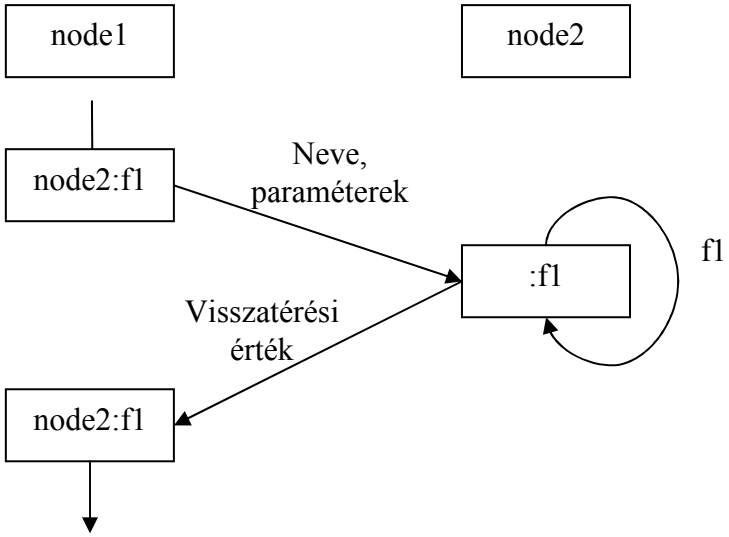
- **Virtuális osztott memória**:
  - Egy szolgáltatás, melyet üzenetekkel tudunk igénybe venni, és a nyújtott szolgáltatás jellegre hasonló a fizikai osztott memóriához → írás és olvasás leképzése üzenetekre
  - A modern buszok (pl. PCIe) is üzenet alapúak → a tényleges, fizikai memória üzenetekkel kezelhető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

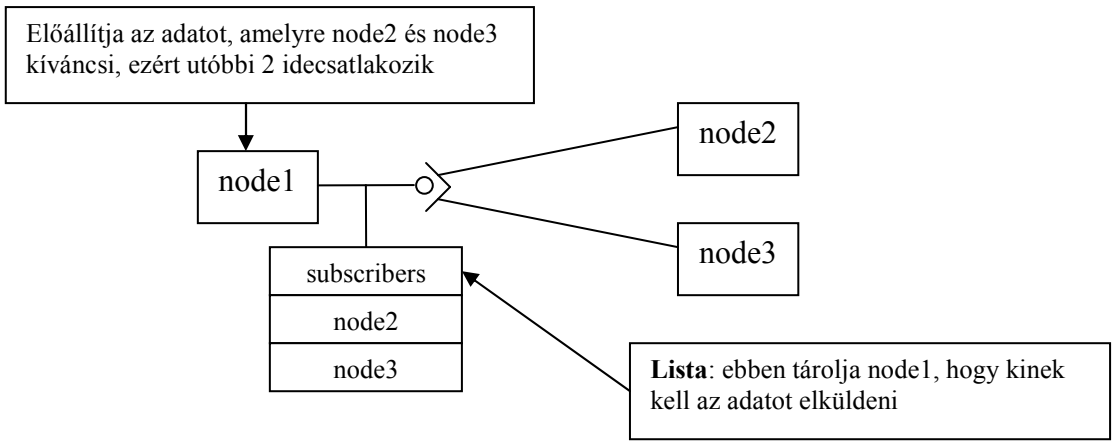
→ BER (Binary Encoding Rules)

- 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

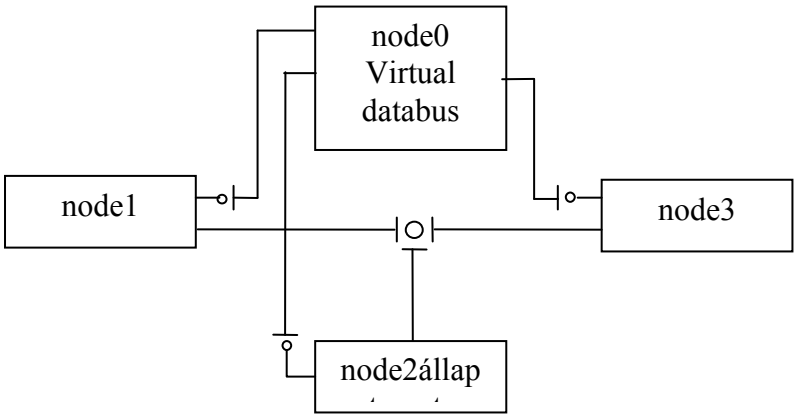
→ Adatvezérelt szerkezet (dataflow)

→ 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

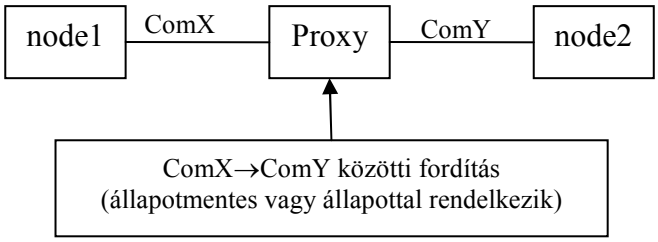


→ Ha valakitől olvas, azt utána kiírja mindegyik node számára
  - node1-3 adatot állíthat elő → node1-3 termelő, node0 fogyasztó
  - node1-3 értesül node0-tól minden információról → node0 termelő node1-3 fogyasztó

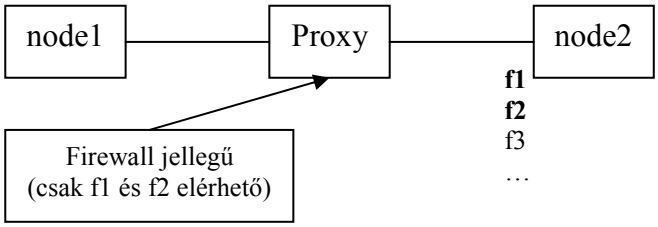
Proxy:

→ beáll két vagy több fél közé a kommunikáció során

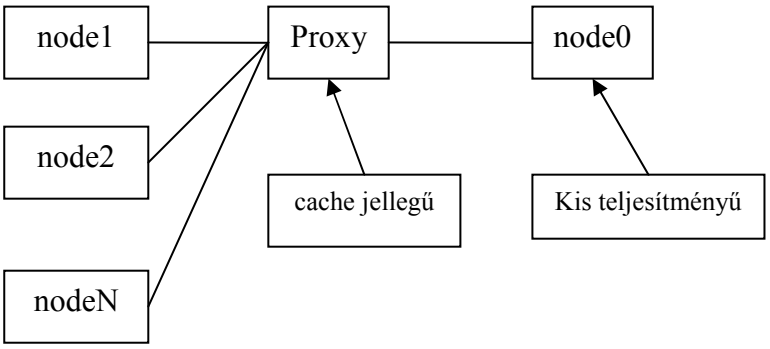
→ cél:
  - kompatibilitás megteremtése



  - 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

14. Előadás – 04. 27.
(Modellalapú SW tervezés)

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)

→ 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.

Miért fontos ez a SW szempontjából?

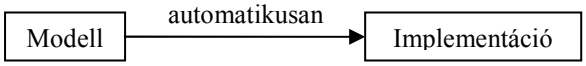
→ A probléma megoldása során így is-úgyis modelleket alkotunk, akkor miért ne tegyük ezt a folyamat részébe?

→ Komplex rendszerek nem hozhatóak létre kód szintű eszközökkel (hatékonyan).

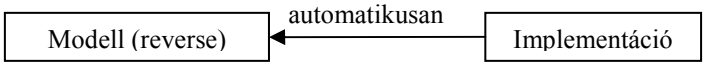
→ Az absztrakt (implementációhoz direkt módon nem köthető) modellek lehetővé teszik a fontos tulajdonságok és az viszonyának leírását.

Mit tesz lehetővé a modell alapú megközelítés?

→ Modellből kód generálása:



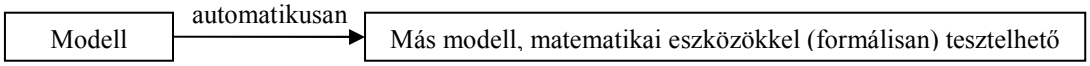
→ Modell ↔ kód kétirányú átjárhatósága



- 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.

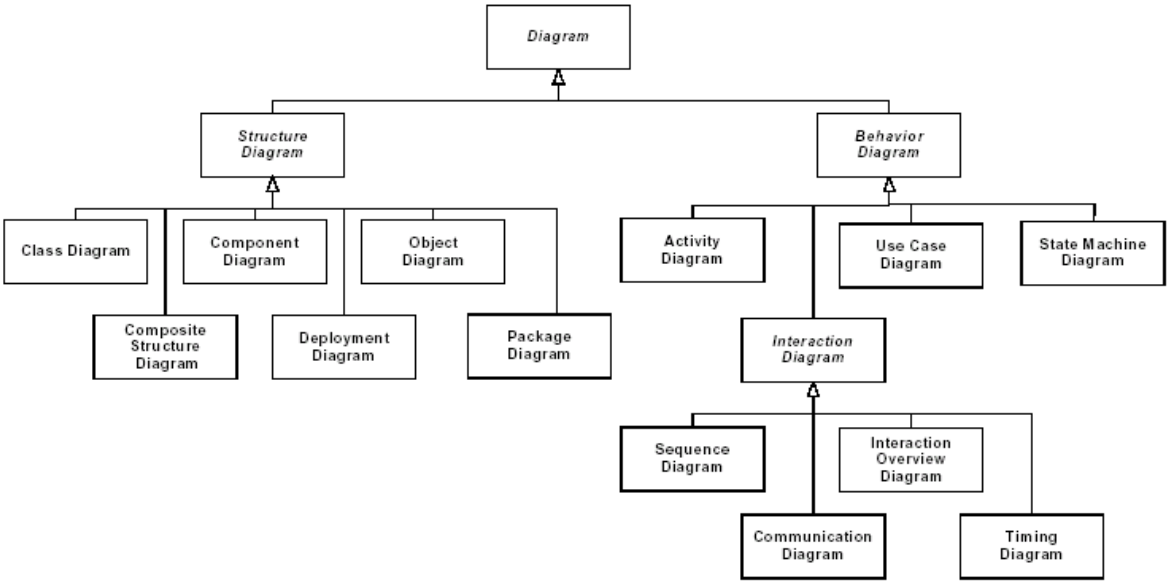
Szükségünk van egy közös jelrendszerre: **UML**.

UML – Unified Modeling Language:

→ Nem mond semmit a folyamatról!
→ OMG (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)

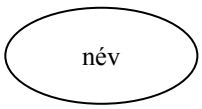
→ Az UML nem csak jelrendszer, hanem tartalmaz egy úgynevezett meta-modellt is.
→ **Meta-modell**: egy olyan modell, ami más modellek leírására szolgál.
→ Az UML le tudja írni önmagát:
- Jó hír: jó kifejezőképességű
- A meta-modell nem formális, klasszikus matematikai módszerekkel a helyessége nem ellenőrizhető.

→ 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)



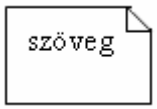
Use-Case Diagram:

→ A rendszertől vagy alrendszertől elvárt viselkedési minták leírása
→ Részei:
- Használati eset

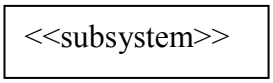


- Aktor (actor)
  - Ember a jele, d nem feltétlenül (automatikusan) egy humán aktorról van szó, lehet a rendszeren kívüli más rendszer is (gép-gép kapcsolat)
- Összefüggések (relations)
  - Szimpla vonal

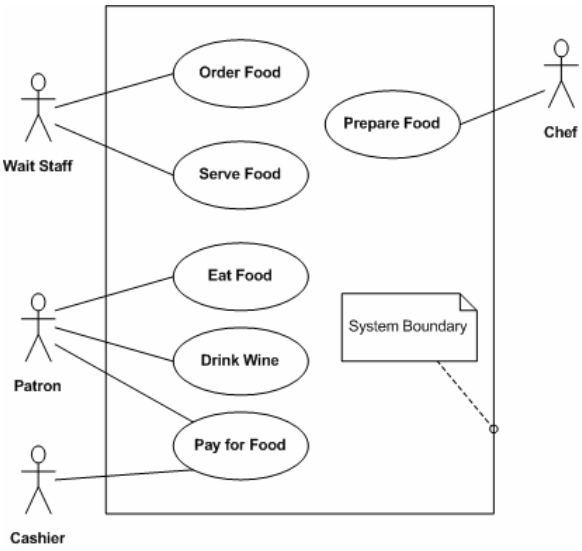
        - Általános összefüggések
        - <<uses>> sztereotípia a default
        - <<includes>>, <<extends>> – további sztereotípiák vonal fölé kell írni, azt mondja meg, hogy milyen a kapcsolat
    - Megjegyzés (note)



    - Kényszer (constrmint) – {szöveg}
    - Rendszerhatár
    - Alrendszerhatár



→ 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.

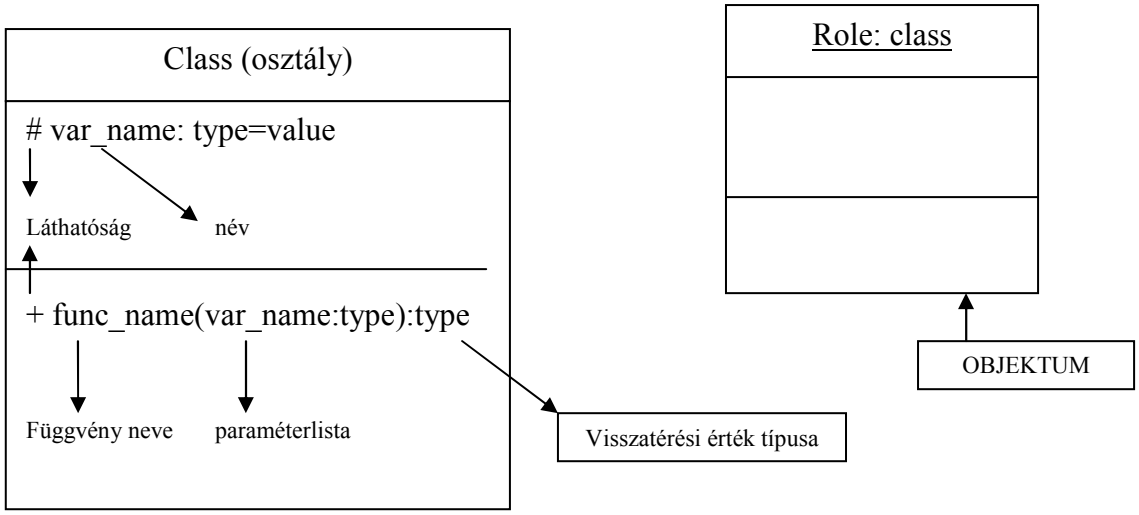
15. Előadás – 05. 04.

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



→ Láthatóság (Visibility):

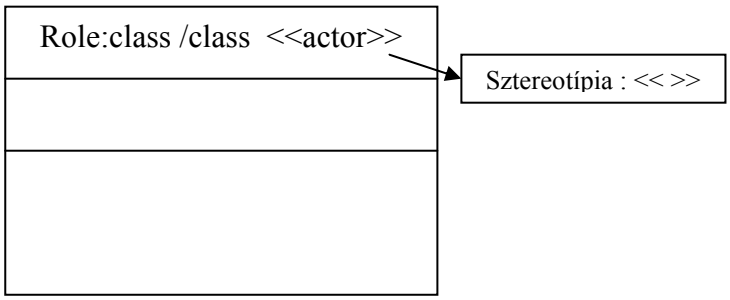
- +: public
- -: private
- #: protected
- ~: package

→ #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.

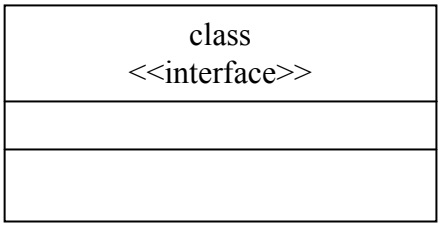
→ Aktorok az osztálydiagramban:

- Az aktort nem kell megírni, az a rendszeren kívül van, kivéve a tesztelést

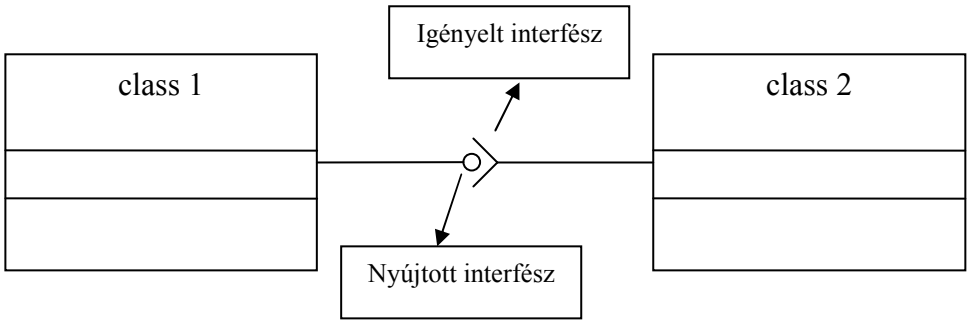


→ **Interfészek**

- Interfész osztályok:

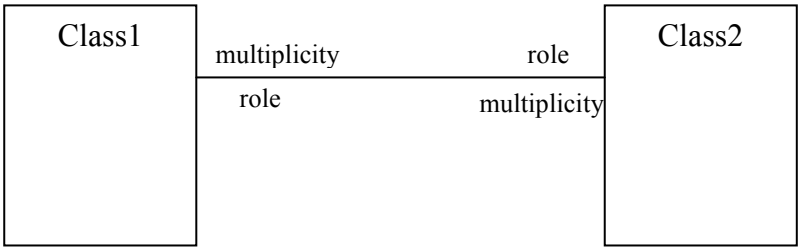


- Megvalósított/használt interfész:

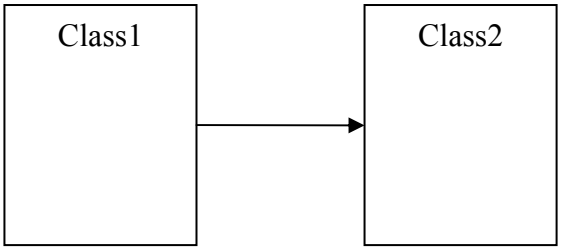


→ **Társítás (association)**

- Kétirányú (postás és kutya)

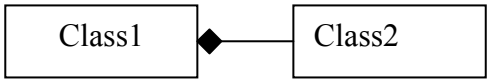


  - Multiplicity = számosság
  - Role = szerep
  - Számosság:
    - Fix számok: 3
    - Lista: 0,2,3
    - Tartomány: 0…10
    - 0 vagy több: *
    - 1 vagy több: 1...n
- Egyirányú társítás (ha szükséges)

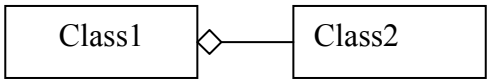


→ **Tartalmazás (aggregation)**

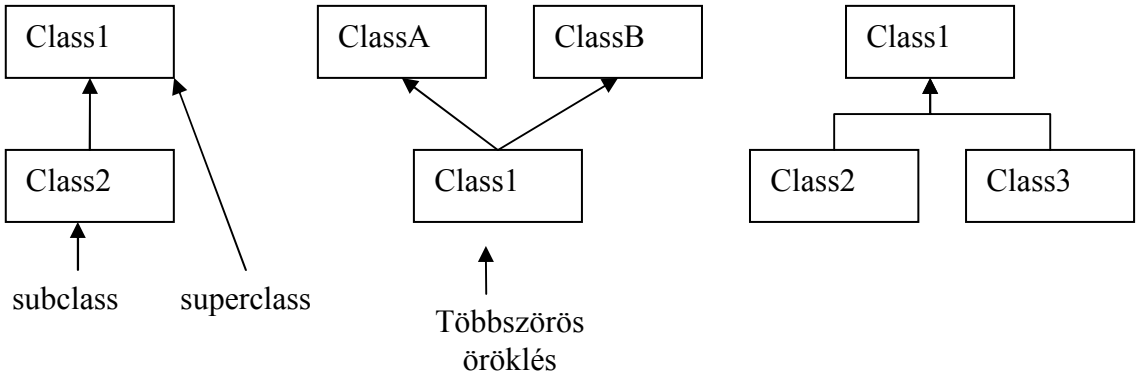
- Érték szerinti



- Referencia szerinti

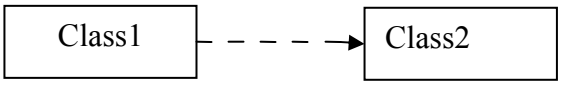


→ **Öröklés (inheritance)**

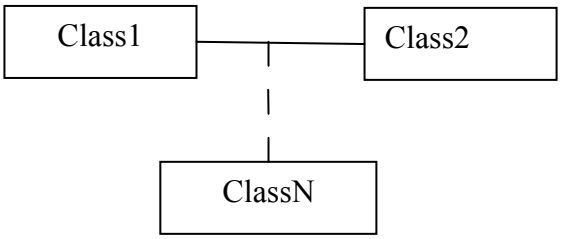


→ **Függés**

- Laza kapcsolat, ami a többivel nem írható le, például nem működik a hőmérő I^2C nélkül



→ **Társítás**



- ClassN: például Hash tömb, amelyen keresztül teremtődik meg a kapcsolat

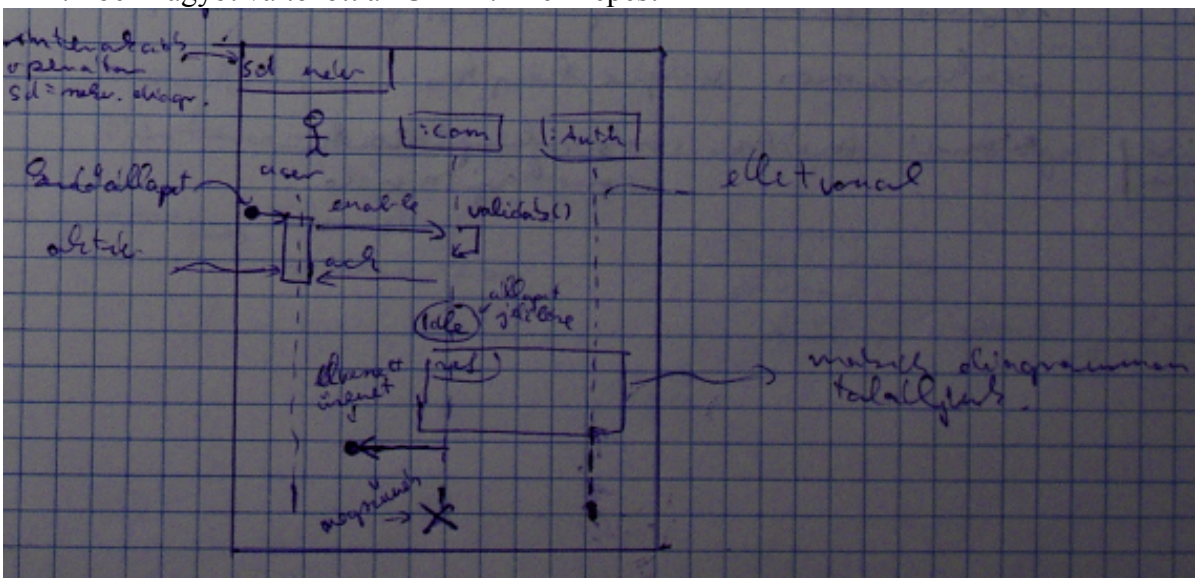
Szekvenciadiagram:

→ Külső viselkedést ír le osztályok vagy objektumok között

→ Scenáriók, viselkedés darabkák (fragments) leírására szolgál

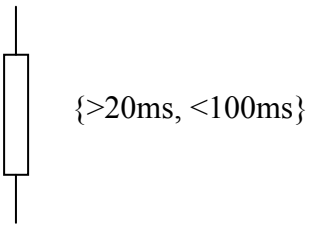
- Pozitívak (így működik) vagy negatívak (nem így működik)
- A scenáriók sokasága írja le a teljes viselkedést

→ 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

Aktivitás diagram

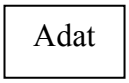
→ Lényegében egy kibővített, jól definiált folyamatábra

→ Működési „token flow” szemantikán alapul

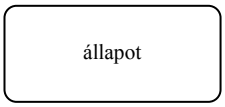
- A token egy adott állapotban van, állapot tartalmazza a tokent
- A token áramlik a rendszerben
- „Petri hálók”

→ Objektumok belső viselkedésének a leírására szolgál

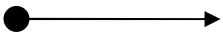
- Adat



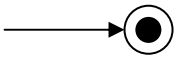
- Állapot



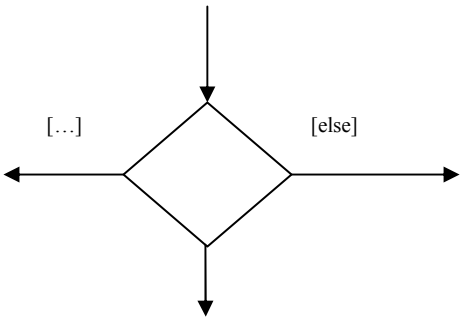
- Kezdő állapot



- Végállapot

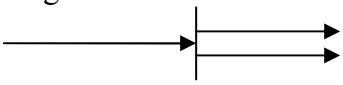


- Elágazás

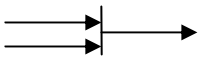


Kérdés: több igaz van?
Nincs igaz?

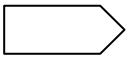
- Fork (elágazás) – mindkét ágban elindul a token



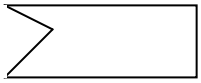
- Join (randevú)



- Esemény generálása



- Külső esemény fogadása



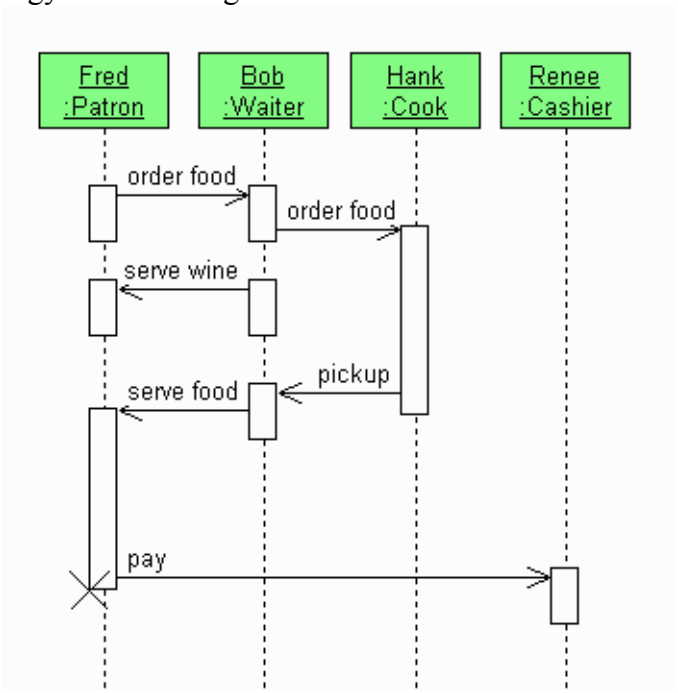
→ UML2.x-ben jelentősen kibővült

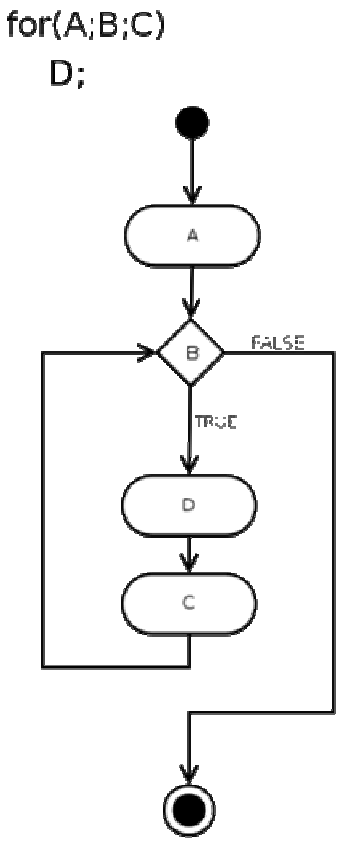
→ A fejlesztési folyamat tetszőleges szakaszán használható

→ Lehetne belőle kódot generálni, de többnyire nem szoktak

→ Az állapotdiagram elterjedtebb

Példa egy szekvencia és egy aktivitás diagramra:

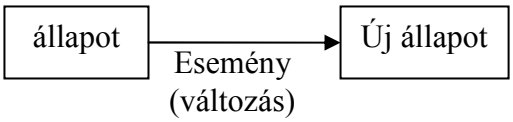




16. Előadás – 05.06.

UML állapotdiagram (statechart, statediagram)

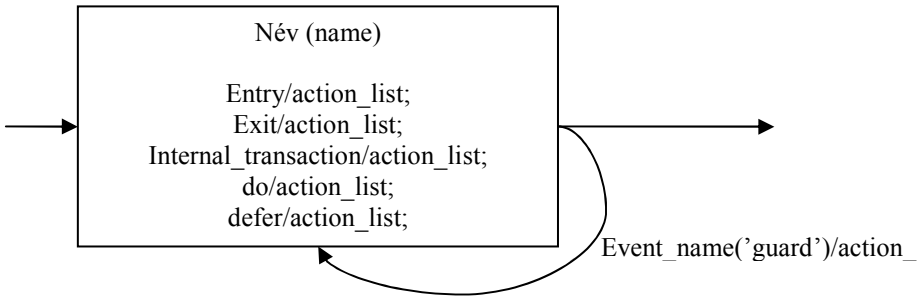
→ Harel féle statechart az alapja
→ Belső viselkedés leírására alkalmas
→ a klasszikus véges állapotgép általánosítása
→ nagyon sikeres, különösen beágyazott rendszerekben
- a beágyazott rendszerek reaktív rendszerek, az őket ért változásokra reagálnak



→ Az állapotdiagram főbb elemei:
- Állapot (state)
- Állapot átmenet (transition)
- Pszeudó-állapot (pseudostate)
  - Itt a rendszer tranziens jelleggel tartózkodik
  - Az állapot átmenetek kedvezőbb/áttekinthetőbb leírását tesznek lehetővé
- Esemény (event)

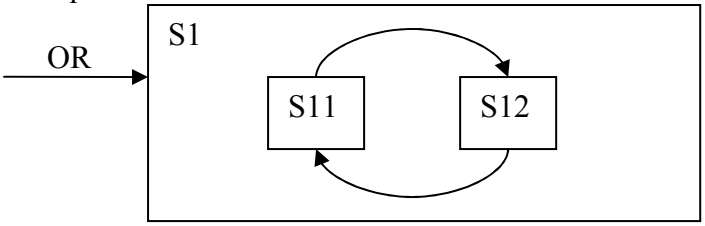
→ Az állapotgép egy irányított gráf
- Állapot és pszeudó-állapot: csomópont
- Állapotátmenet: él
  - Mindig van rajta nyíl (van iránya), hiszen a gráf irányított

Az állapotátmenetek megadása



- 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)

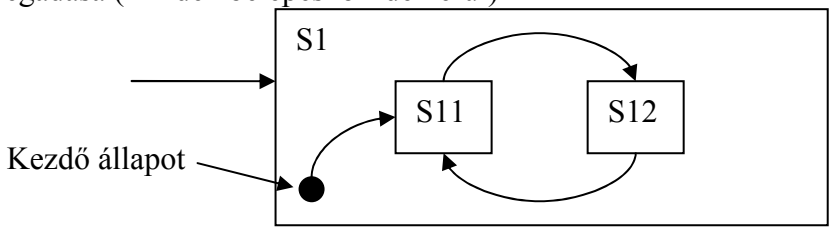
→ Internal_transaction: belső állapok vannak



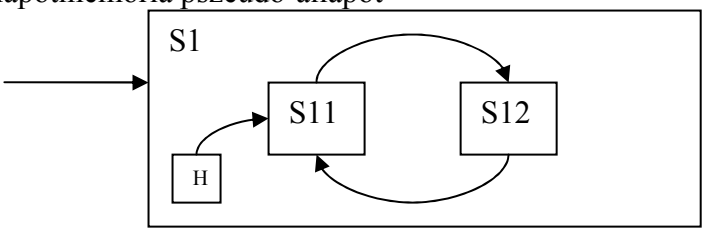
- Ha S1-ben vagyunk, akkor vagy S11-ben, vagy S12-ben vagyunk
- Valójában az állapot vagy S1:S11, vagy S1:S12
- Az állapotgép hierarchikus

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)

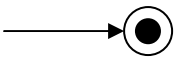


   - Sekély és mély állapotmemória pszeudó-állapot

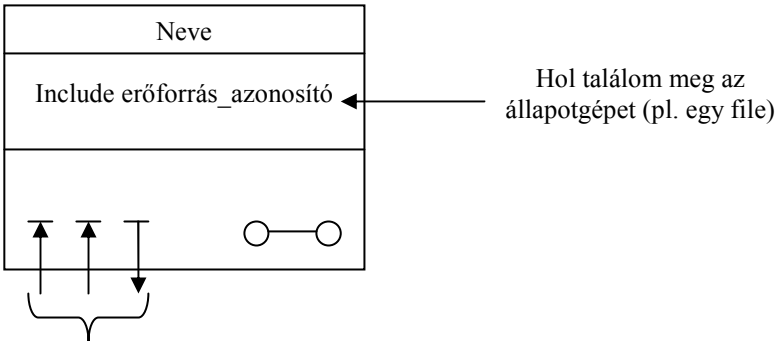


     - Sekély: S11-re és S12-re igaz
     - H* a mély: a teljes hierarchiára működik az emlékezet
     - Ha szimpla H van: S11-gyel kezdünk, ha kilépünk megjegyezzük, hogy melyik állapotban voltunk, a következő belépéskor oda lépünk vissza
     - H* esetén nem csak ezen a szinten jegyezzük meg , hogy S12-ben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

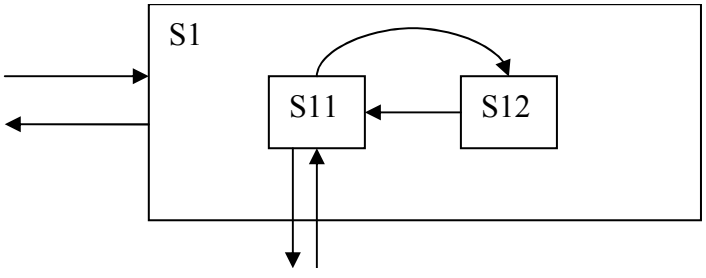
2. Végállapot (leállítás)



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

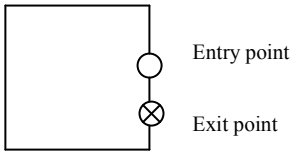


Állapotgépen belülről induló vagy belülre mutató állapotátmenetek vannak



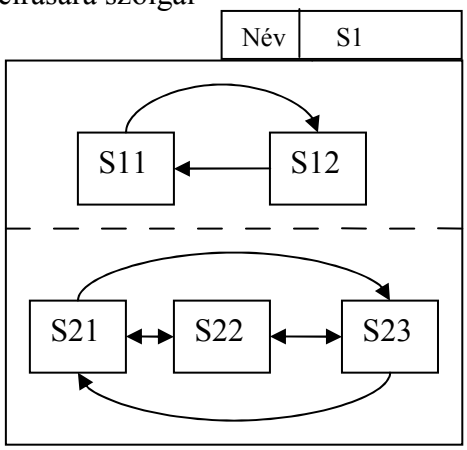
Ezt kell tudnia megadni

- 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

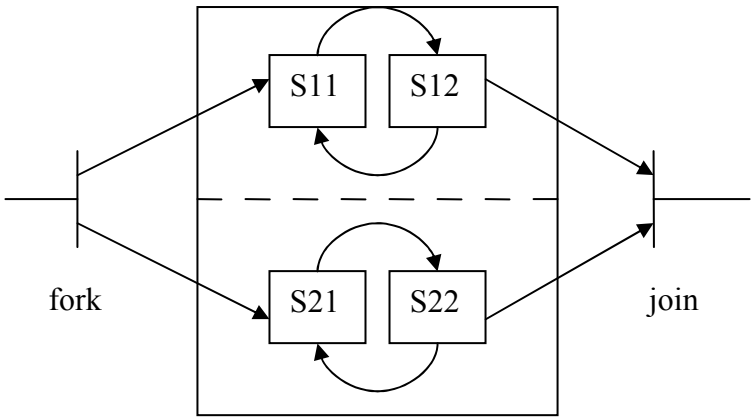
→ A megkapott eseményeket az AND állapotok külön-külön megkapják

- 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

Állapotdiagramok megvalósítása

→ 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

→ Beágyazott switch utasítás:

- 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

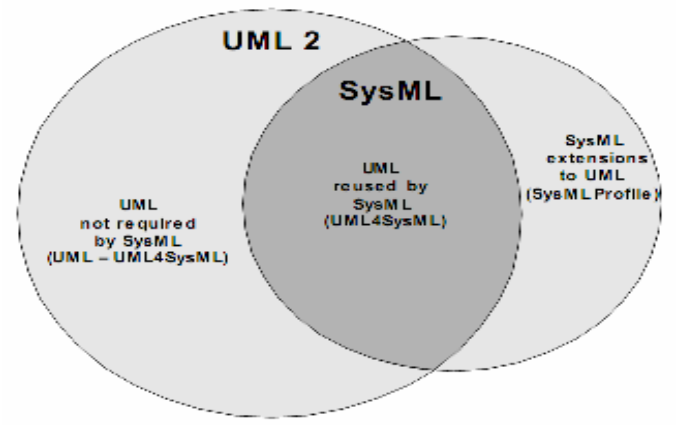
→ Összehasonlítás:

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)

17. Előadás – 05. 11.

SysML – System Modelling Language

→ www.omgsysml.org
→ Komplex rendszerek modellezése (elsősorban nem SW)
→ 1.0 verzió (1.1 kidolgozás alatt)
→ SysML és UML kapcsolata:
  - Valójában a SysML egy UML profil



→ SysML diagram típusok:

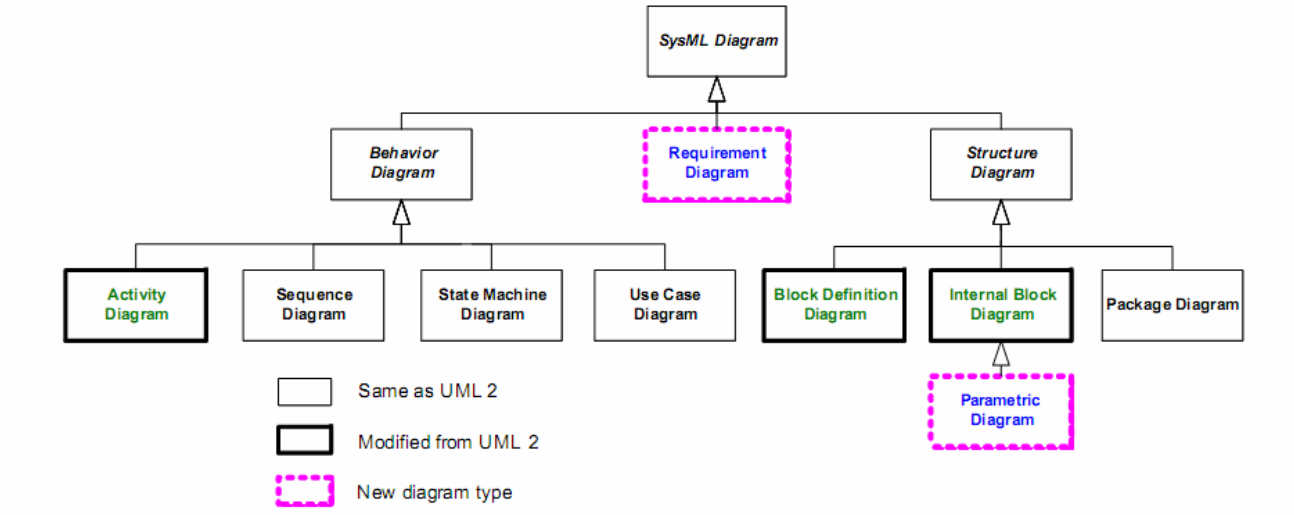
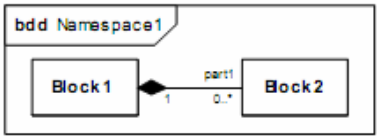


Figure 4.4 - SysML Diagram Taxonomy

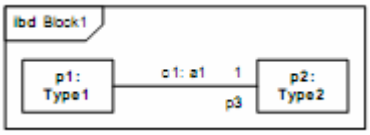
  - Behavior: viselkedést leíró diagramok
    - Activity
    - Sequence
    - State machine
    - USE-CASE
  - Structure: rendszer felépítését írják le
    - Block definition
    - internal structure
      - Parametric
    - Package
  - Requirements

→ SysML tutorial: http://www.omgsysml.org/INCOSE-2008-OMGSysML-Tutorial-Final_revb.pdf

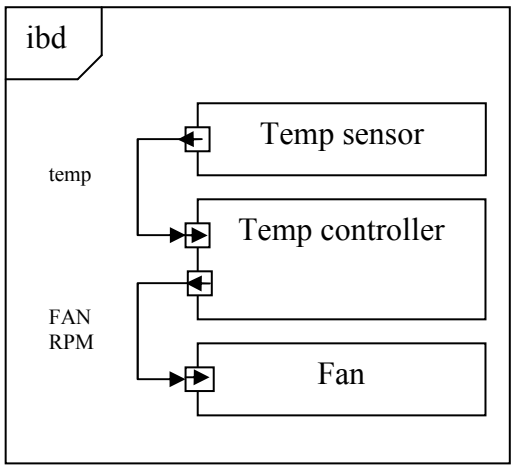
→ Block definition diagram:
- Tartalmazás, összetétel, stb.



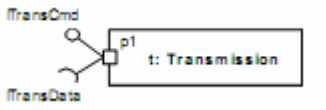
→ internal blokk diagram: információ, fizikai mennyiségek áramlása:

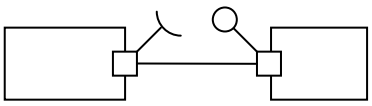


- egy példa:



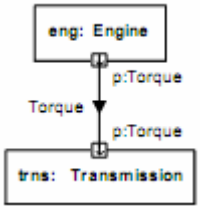
  - temp szenzor
  - temp controller
  - ventilátor
- az egyes blokkok közé: portok:
  - standard port



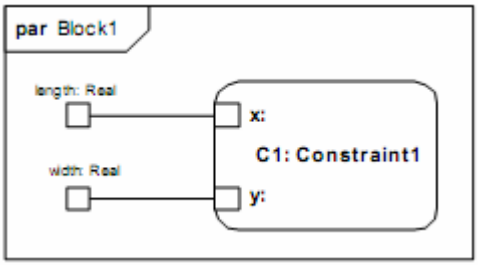


    - karika: megnyitja a funkciót
    - félkör: hívja az operációt

  - flow port
    - egyértelmű,
    - információ, vagy fizikai mennyiségek áramlása

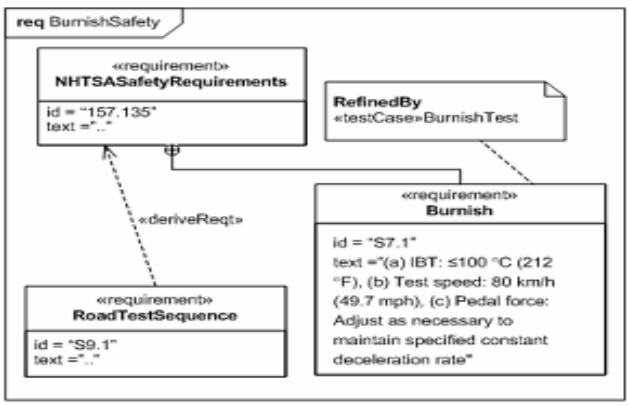


→ Parametrikus diagram:
- dobozok, amik leírják a különböző pontok között milyen összefüggések vannak
- ez hasonlít a Simulink adatfolyam hálójához
- sajnos az összefüggés megadó nyelv hiányzik: ez egy szöveg string, nincs semmi értelme
  - ha lehetne MathML: a problémánk meg lenne oldva
- példa:



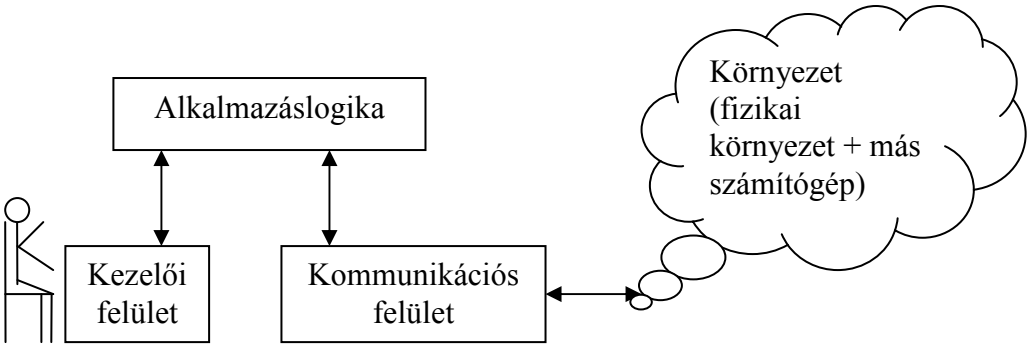
→ aktivitás diagram:
- hierarchikus
- inputok és outputok kezelése megoldott
- még mindig számos gond van
  - enhanced funktional flow block diagram a megoldás, ami egy UML profil, ha nem lenne a SysML aktivitás diagram elégséges

→ követelmény diagram:
- követelmények szoros összefüggésben vannak egymással, és ezek az alrendszerekre továbbterjednek
- amikor fejlesztünk, a folyamatban rengeteg változás történik, követni kéne egy komponens megváltoztatása hogyan mozog a követelményhierarchián → követelménydiagram
- követelménynek van:
  - azonosítója (neve)
  - tulajdonságai (felhasználó bővítheti)
  - követelmények között vannak kapcsolatok
- Milyen kapcsolatok lehetségesek:
  - derived (származtatott)
  - satisfied (kielégít)
  - verify (ellenőriz)
  - refine (finoít)
  - trace (visszavezet)
  - copy (másol)
- példa:



Generációs fejlesztőrendszerek

- 4GL: félrevezető, magas szintű nyelvekre épülő programfejlesztési környezetek ezek
  - általában grafikusak (Matlab Simulnik, National Labview, MS Visual Studio)
  - általános architektúra



    - alkalmazáslogika:
    - kezelőfelület: legnehezebben megírható része a rsz-nek (PC-n belül AD kártya kezelése, meg windows driver megírása →ezeknél nem írunk drivert jön magától)
    - kommunikációs felület: környezettel (fizikai környezet + más számítógépek) van kapcsolatban
  - itt grafikus módon kell programozni
  - minden komponenst grafikus objektumként használok a programfejlesztés során és ezeket huzalozom össze
  - SW fejlesztés abból áll, hogy vannak komponenseink és azoknak van valamilyen vizuális megjelenése, tehát vannak portjai:
    - portok (input, output futás közbeni konfiguráció)
    - konfigurációs felület:
      - tervezés idejű konfiguráció megadása
    - ezeket kell összekötni (plumbing)
  - **grafikus felület kialakítása**:
    - layout vagy dialog editor
    - objektumok felelnek mega GUI elemeinek
    - beágyazott rendszerben sokszínűbben a felhasználói felület elemek
    - tipikus elemek:
      - nyomógomb, kapcsoló, választó gomb, szövegmező, listák (állandó/listaelem/, beugró/legördülő/, vegyes(kombó), görgetősáv)
      - list (jól konfigolható eszköz: pl.: lista nézet explorerben) és tree (fa struktúrát mutató contol) control/view
      - menük, dalógudobozok, TAB control, date and time control (naptár),
  - grafikus felületeknél 2 fontos dolog:
    - fókusz: bizonyos controlt használva mit tudunk elérni
    - navigáció: egérrel való navigáció (billentyűzet, gyorbillentyűk, stb.)
  - **kommunikációs felület:**
    - OS szolgáltatás hívása
      - fájlkezelés stb.
    - adatbáziskezelés:
      - standard (ODBC vagy JDBC) és gyártóspecifikus (MS,Oracle, mySQL, stb.) programozási felületek
    - komplex alkalmazások programozott elérése (Word-ból elindul az Excel)
      - Windows → com/dcom
      - Linux → dbus, CORBA

- mérési adatgyűjtés és beavatkozás: általában egy speciális HW van, amihez kell egy driver és 4GL támogatás a driverhez: amikor az USB plug and play felismeri a cuccokat