Szoftvertechnológia jegyzet

Bevezetés:

A tárgy kettös célkitüzése:
- átfogó kép a szoftvertechnológiáról, arról szól lényegében hogy hogyan csinálunk szoftvert nagyban
- objektum-orientált gondolkozásmód és fejlesztés, java nyelven gyakorlatok

Ha hibát találsz akkor ide írj róla: sz.csabi.3@gmail.com

Szoftver

Szoftver definicíója:
- *def1:* utasítássorozatok melyek a hardvert müködésre késztetik
- *def2:* programok és a vele kapcsolatos dokumentációk, mindazok melyek szükségesek a kifejlesztéshez, teszteléshez, karbantartáshoz és müködtetéshez

IT service (információ technológiai szolgáltatások):
- hardver/szoftver karbantartás és támogatás
- alkalmazás fejlesztés
- telepítés
- rendszerinformáció
- helpdesk, ...

Aktuális problémák:
- szoftver iránti igény meghaladja a szoftver gyártási kapacitást
- minöség iránti igény
- régi (legacy) szoftverek melyek jók, de újra kéne írni öket
- régi szoftverek felújítása (újra ki kell találni vagy valahogy fel lehet használni?)

Szoftver minösége:
- korrektség (különbözö reprezentációknak meg kell felelniük egymásnak)
- megbízhatóság
- teljesítmény
- hordozhatóság

Szoftverkarbantartás:
- 5-15 év egy szoftver élettartama
    - van, hogy 35 éves vagy akár öregebb is és a mai napig használják (pl.: fizikában)
- az adatbázisok karbantartása
- hogyan tároljuk az adatokat hogy ne vesszenek el
- jó lenne úgy csinálni a szoftvert hogy az karbantartható legyen!
- követünk el hibákat, melyek a körülmények hatására súlyos hibákat okozhatnak

Software engineering céljai:
- szoftver termelékenység növelése
- minöség növelése
- fejlesztési és karbantartási költségek csökkentése
- költségek precíz mérése
- automatizált szoftver gyártás

Technológia

Mérnökség (technológia) definíciója:
- Költséghatékony megoldást kell készíteni ...
- ... gyakorlati problémákra ...
- ... tudományos ismereteket felhasználva ...
- ... dolgokat építve ...
- ... az emberiség szolgálatában (szabályoknak/szabványoknak/törvényeknek/elöírásoknak megfelelöen).

Szoftvertechnológia definíciója:
- *def1*: Tudományos ismeretek felhasználásával pratikus alkalmazás tervezése és készítése és a hozzá tartozó dokumentáció elkészítése ami szükséges a fejlesztéshez, használathoz és karbantartáshoz
- *def2*: Technológiai és vezetéstudományi ismeretek összessége amelyek a gyártáshoz és karbantartáshoz kellenek, idö és becsült költségek betartásával.

4 P a szoftvertechnológiában:
- people
- problem
- process
- product

Szoftver process (szoftver folyamat)

Minöséget meghatározó szervezetek:
- szabványalkotó testületek (IEEE, BSC, NATO, DoD, ...)

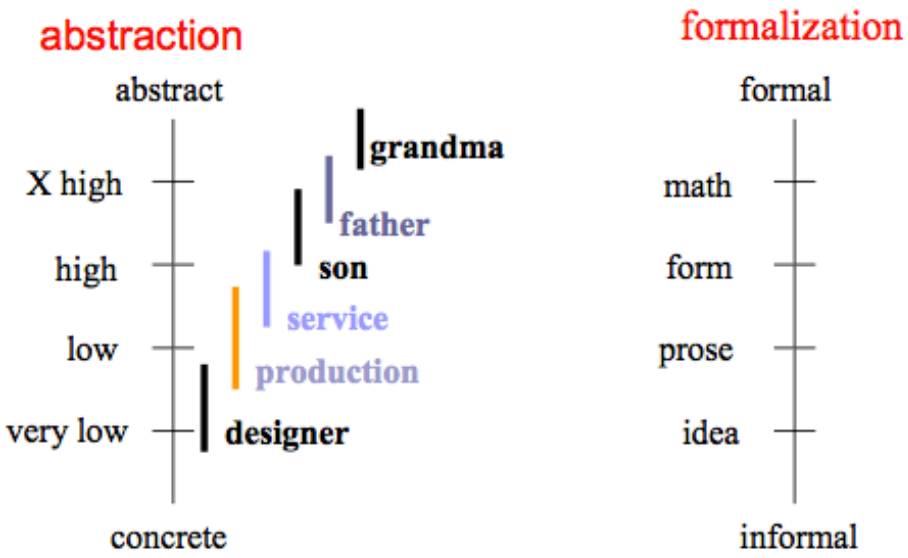
Minöségi képek:
- transzcendentális
    - nem mérhetö, mint például szépség
    - nem használjuk
- felhasználói
    - probléma: csak akkor ha már kész a termék
- gyártás
    - ha a gyártási eljárást betartjuk akkor jó szoftvert kapunk
    - ezt alkalmazzuk
- termék
    - alkalmazzuk, minöség ellenörzés kiadás elött
- értékarányosság
    - “ami drágább az jobb minöségü”

Processz definíciója:
- amit az emberek csinálnak, eljárások, módszerek, eszközök és berendezése amik ehhez a tevékenységhez kellenek olyan célból hogy az alapanyagot számunkra hasznos termékké transzformáljuk át

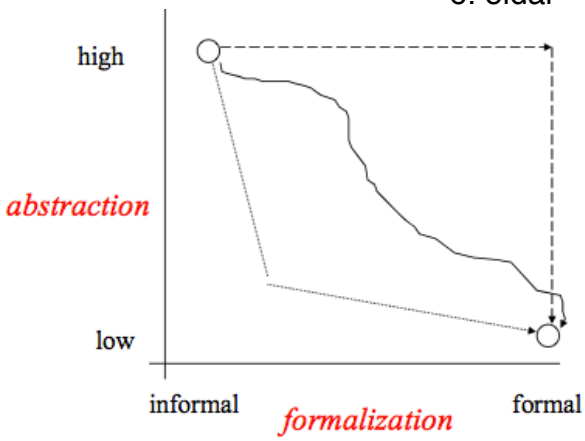
Gyártási elv:
- a szofter minöségét az határozza meg, hogy milyen folyamat során keletkezik

Absztrakció: információrejtéssel függ össze, minél több információt rejtünk el, annál absztraktabb

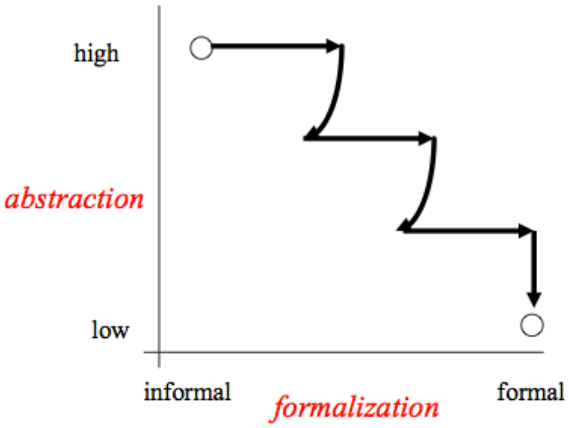
Formalizáltság: ha elegendöen formális a leírásom, akkor abból transzformációval elö tudok állítani egy másik leírást



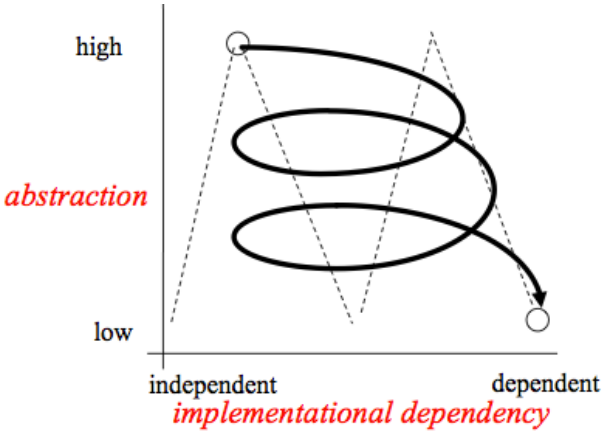
Fejlesztés diagramja:
- absztrakt dologból (leírásból) indulunk
- az ideális eset a szaggatott vonal lenne
- a formális leírás készítés a specifikálás
- a kérdés az útvonal



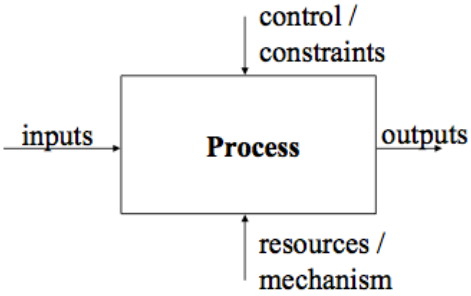
1) **Valós fejlesztési diagram**:
- a mai tendencia: 2 lépés
  - 1.: platform független model
  - 2.: platform specifikus dolgok



2) **Ward-Mellor modell**:
- absztrakció - implementációs függöség
- implementációs függöség: mennyire függ az informatikától és mennyiben függ az alkalmazástól
- magas absztrakt szintü alkalmazói fogalomrendszerböl indulunk ki és ehhez keresünk megfelelö informatikai fogalmakat
- jobb oldali háromszög: alkalmazói tér
- bal oldali háromszög: implementációs ter



3) **ICOM model**:
- input: követelmények
- kimenet: kód, dokumentáció
- kontrol: pénz, szabványok
- eröforrás: csapat, eszközök



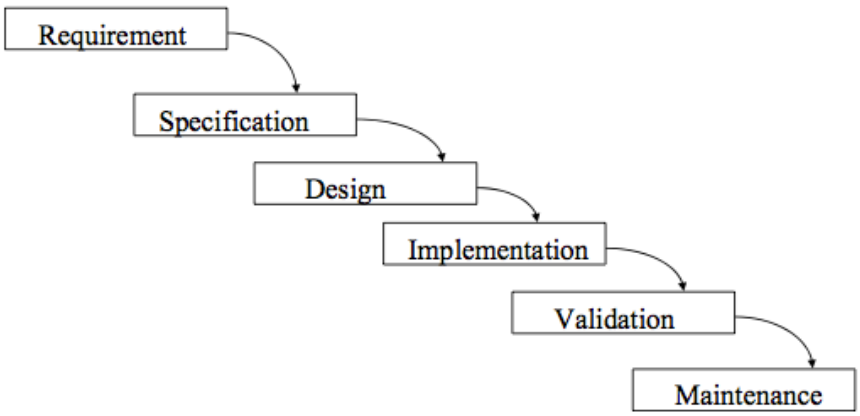
Folyamat specifikáció ürlapja:
- cél
- szerepek: kik a szereplök és ki miért felel
- folyamat elkezdésének kritériuma
- bemenetek
- folyamatban lévö aktivitások, mit csinálnak a szereplök
- kimenet
- kimenet kritériuma
- folyamat metrika, értékek amiket mérünk a folyamat közben

Szoftver fejlesztési folyamat:
- a legbonyolultabb folyamatok közé tartozik
- emberigényes, emberfüggö
- dokumentációt és kódot kell készíteni
- bonyolultsággal kell megküzdeni, fel kell bontani egyszerübb feladatokra (Work Breakdown Structure - WBS)

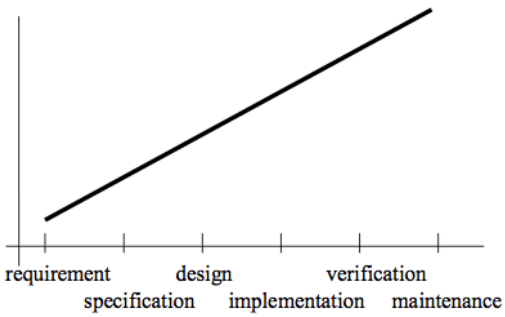
Szoftver fejlesztési folyamat életciklusa:
- követelmények tisztázása (requirement)
- specifikáció
    - formális leírás, megoldás amely kielégíti a követelményeket
- tervezés (design)
    - a jelenlegi semmiböl mit kell ahhoz csinálni hogy olyat kapjunk amit specifikáltunk
    - technikai részletek, felhasznált eszközök, pénz, munka szervezése, ...
    - a végén kapunk egy listát amiket végre kell hajtani
- konstrukció/gyártás (implementation/construction)
    - a tervezés során készített lista végrehajtása
- validáció/érvényesség ellenörzése
    - megfelel-e a termék a követelményeknek
- karbantartás (maintenance)

Szoftver életciklus modellek

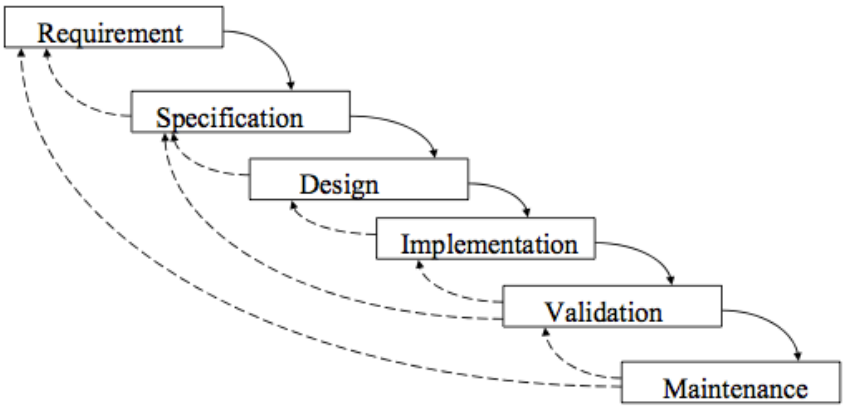
1) Lineáris szekvenciális



- a fejlesztési folyamat szekvenciális, lépcsöszerü végrehajtása
- statisztikák szerint a hibák nagyrésze implementáció elött van
- minél késöb vesszük észre a hibát, annál többe fog kerülni

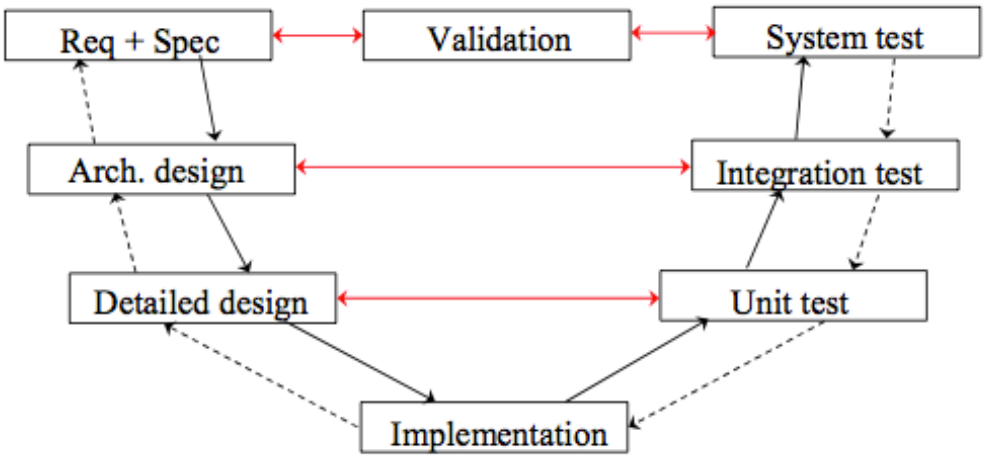


2) Vízesés modell



- szaggatott vonal: visszalépés
- a lineáris szekvenciális model hibáját javítja
- minél később lépünk vissza, annál költségesebb lesz

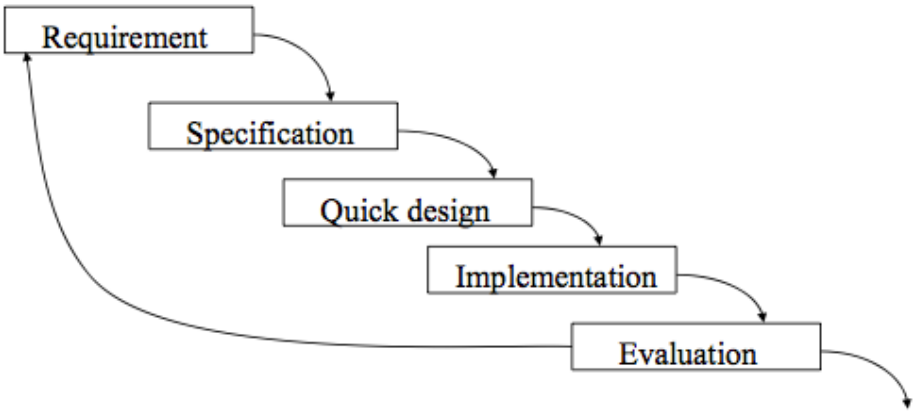
3) V model



- eddig: a megvalósításig finomítottunk, a végén elkészítettünk egy csomó kódot
- sok programozó esetén sok kisebb programrészt kell összerakni, ezt pont ellentétes a fejlesztési iránnyal
- unit test: egy programozó által elkészitett program ellenörzése

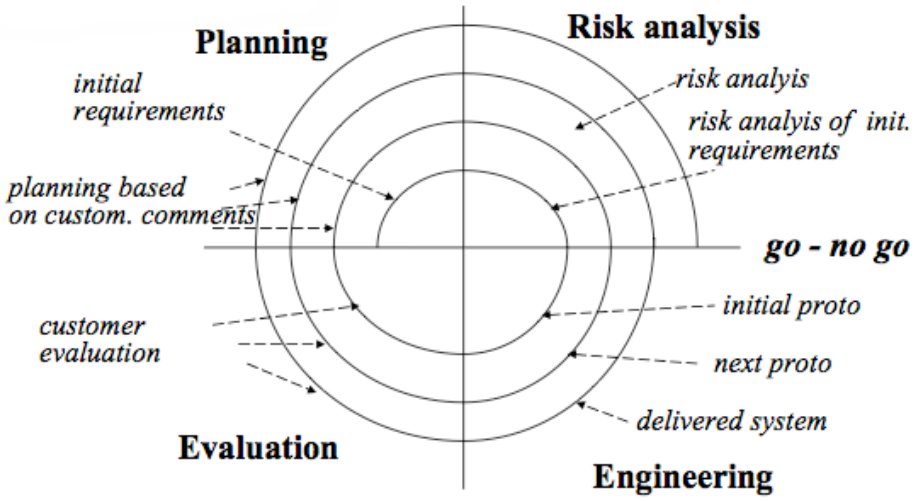
- integration test: komponensekböl összerakott architektúrális elemek tesztje
- system test: a kész egész rendszer

4) Prototípus és iteratív inkrementális



- prototípusokat készítünk (kritikus viselkedésekre)
- a gyors alkalmazásfejlesztés ennek egy változata
- például egy webes alkalmazás már gyorsan készíthetö

5) Spirál model
- 1. síknegyed: követelmények meghatározása [konstrukció, szintézis]
- 2. síknegyed: kockázat elemzés [analízis, elemzés]
- go / no go
- 3. síknegyed: technológia, ahol egy prototípust készítünk [konstrukciói, szintézis]
- 4. síknegyed: kiértékelés [analízis, elemzés]



Szoftver keletkezés fázisai végére mikor érünk, miket kell elkészíteni:
(nem kell pontosan tudni ezt a listát)
- követelmények
    - rendszerdefiníció, projekt terv
    - itt még születhet olyan döntés hogy nem megy tovább a project
    - PFR (Project Feasibility Review)
- specifikáció
    - specifikáció technikai része (Requirement specification)
    - felhasználói felületek/dokumentum tervei (Preliminary user's manual)
    - verifikációs terv elözetes változata (Preliminary Verification plan)
    - SRR (Software Requirement Review)
- architektúrális tervezés
    - architektúra terv: mik a nagy komponensek, hogyan kapcsolódnak össze
    - PDR (Preliminary Design Review)
- részletes tervezés
    - Work Breakdown Structure (WBS) - részfeladatokra bontás
    - Részletes tervek
    - felhasználói kézikönyv
    - verifikációs terv
    - CDR (Critical Design Review)
- implementálás
    - kód felülvizsgálat (code review)
    - átadási teszt tervei (acceptance test plan)
    - SCR (Source Code Review)
    - ATR (Acceptance Test Review)
- validáció
    - minden dokumantáció felülírt változata (update)
    - projekt kiértékelés
    - PRR (Product Release Review), PPM (Project Post Mortem)

Problémák az életmodellekkel:
- ritkán követi a valóságot
- a megbízó nem tudja a követelményeket pontosan definiálni
- a megbízó türelmetlen, gyorsan kell valamit mutatni a megbízónak

CMM (Capability Maturity Model):
- a fejlesztési folyamatot vizsgáljuk (SEI fejlesztette ki - Software Engineering Institute)
    - ha jó a fejlesztesi folyamat akkor jó a termék (mérni kéne a fejlesztési folyamatot)
- nem a szoftverre mondjuk ezt, hanem a cégre
- 5 szintes a CMM model
    - 1 - Kezdetleges
        - bárki aki nekilát és kódot ír
    - 2 - Ismétlödö
        - nagyvonalakban definiált dokumentációs minták
        - van egy lista ami alapján dolgoznak, nem szabványos
        - a vevöi igények és a munkatermékek már kontrolláltak
        - alapvetö projekt-menedzsment gyakorlatok kialakultak
        - lehetövé vált a költségek, az ütemterv és a funkcionalitás nyomonkövetése
    - 3 - Definiált
        - van egy fejlesztési folyamatuk ami megfelel valamilyen szabványnak
        - a szoftverfolyamat tevékenységei már beintegrálódtak a szevezet szabványos szoftver-folyamatában
        - minden projekt a szervezet szabványos fejlesztési és karbantartási folyamatának egy jóváhagyott, személyre szabott verzióját követi
    - 4 - Menedzselt
        - mérés van, a folyamat megjósolható
        - a folyamatból szerzett mérési eredmények vannak, ezek felhasznált mérési eredmények a szoftver folyamat tervezése és végrehajtása során
        - a szoftverfolyamatról módszeresen adatokat gyüjtenek
        - a vezetök képesek a folyamatok elörehaladásának és a problémáknak a mérésére
    - 5 - Optimalizált
        - magát a folyamatot tervezhetjük termékfüggöen
        - ellenörzött módon újés jaívtott szoftverfejlesztési eszközöket próbálnak ki
        - számszerü visszacsatolás segíti az állandó folyamatfejlesztést
        - innovatív öteletek és technológiák segítik az állandó folyamat-fejlesztést

    - Szinthet egyértelmüen nem besorolható állítások:
        - a folyamat számszerü mérése lehetövé teszi az ipari szabványok javítását
        - a termék életciklusának tervezése beépült a folyamatfejlesztés részfeladatai közé
        - a vezetök képesek a termék minöségének közvetlen ellenörzésére
        - a mérések eredményei lehetövé teszik a hatékony vezetöváltást

UML

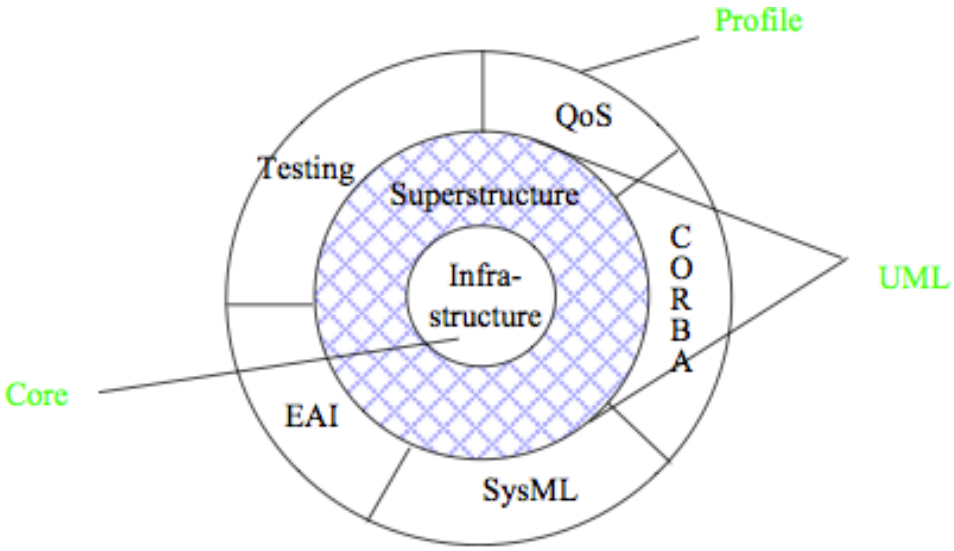
Mit specifikálunk, vizualizálunk?
- termékeket
- szoftver rendszerekben elöállított termékeket
- üzleti folyamatok modellezése és más nem szoftver termékek jellemzése

A legjobb mérnöki gyakorlatot próbálja bevezetni. Nagy és bonyolult rendszereket is lehet vele modellezni.

Mi van az UML-ben?
- nyitott dolog, “boríték amit ki lehet terjeszteni”
- standard modellezönyelv és nem egy folyamat
    - arról hogy mit kell tennünk, arról nem mond semmit
- szigorú jelölésrendszer

Amire nem terjed ki az UML:
- nem program nyelv
- nem eszköz
- nem fejlesztési folyamat

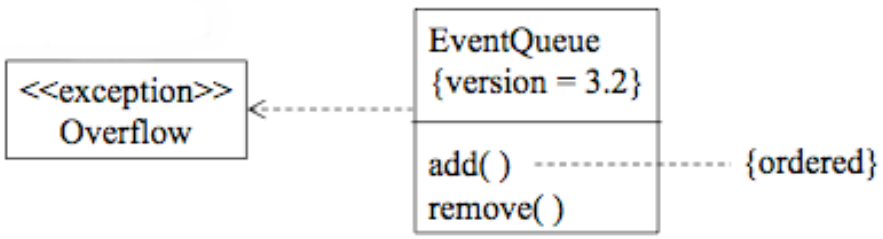
UML2 struktúrája:



Infrastruktúra:
MOF (Meta Object Facility): önmagát leíró nyelv

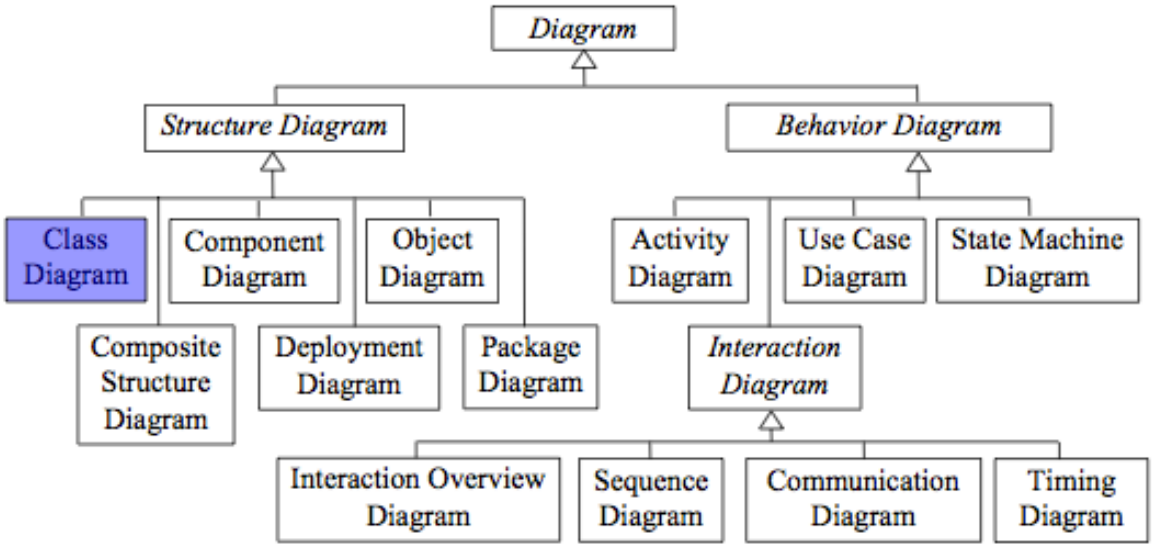
UML jellemzöi:
- az UML egy gráf topológia
- 3 relációt tartalmazhat
    - kapcsolat (vonalak a dobozok között)
    - tartozás (egyik doboz a másik dobozban van)
    - közelség (vonalra írunk valamit)
- grafikus elemeket is tartalmazhat
    - ikonok
    - 2D szimbólumok
    - összeköttetések
    - stringek

3 kiterjesztö mechanizmus:
- korlátozás (constraints)
    - egy elemhez kapcsolok egy szemantikus korlátot
- sztereotípia
    - az UML elemkészlet nem biztosít elegendö szelekciót
- tagged value
    - név-érték páros (property)



Ábra elemzés:
- korlátozás: add()-hoz korlátozást adunk, egy ordered korlátozást (rendezett marad a Queue hozzádás után is)
- sztereotípia: az <<exception>> egy sztereotípia, mely megmondja hogy ez egy kivétel
- tagged value: version = 3.2

UML Diagramok - Class Diagram



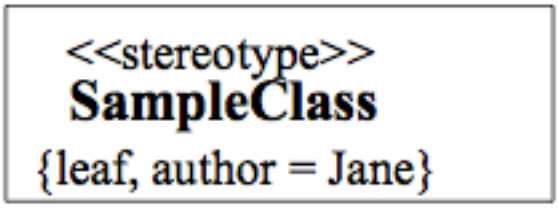
Class:
- olyan objektum, aki képes magát megvalósítani
- tartalmaz egy specifikációt ami leírja, hogy milyen attribútumok és operációk tartoznak hozzá
- különbözö szinteken különbözö bonyolultságú kifejtés lehet:

analysis level

implementation level

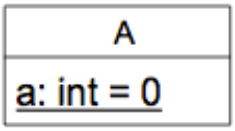
Class név doboz:
- (sztereotípia), név, (property)
- ha *dölt betüvel* van szedve akkor absztrakt
- standard sztereotípiák:
    - *utility:* aminek csak statikus attribútumai és operációi vannak
    - *metaclass:* olyan osztály melynek példányai osztályok
    - *interface, enumeration*

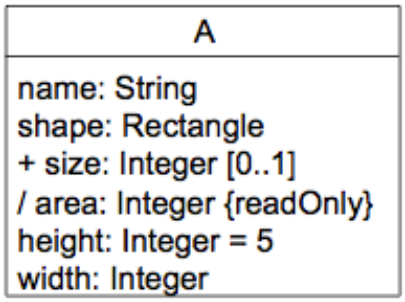
Példa: SampleClass, mely stereotype sztereotípiával rendelkezik, leaf jellemzöje van (leaf = true) és a szerzöje Jane



Class attribútum:

- [<visibility>] [‘/’] <name> [‘:’ <type>] [‘[‘ <multiplicity> ‘]’] [‘=’ <default>]
  [‘{‘ <prop-modifier> [‘,’ <prop-modifier>]* ‘}’]
- visibility:
  - public : ‘+’
  - protected: ‘#’
  - private: ‘-’
  - package: ‘ ‘

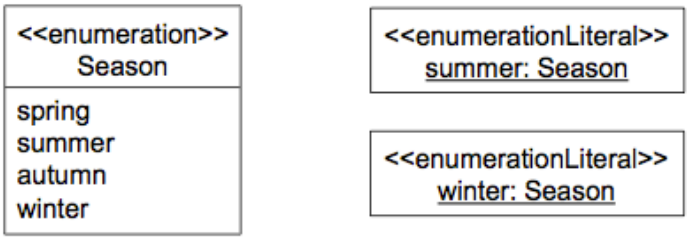




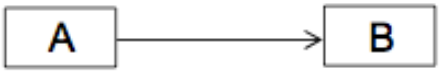
- attribútumok
  - ha alá van húzva, akkor class-scope (static)
- prop-modifier
  - readOnly
  - subsets <property-name>
    - attribútumoknak részhalmaza szerepel
  - union
    - fölszármazásnál fordul elö
    - pl.: az ösosztály attribútuma a leszármaztatott osztályok attribútumai
  - redefines <property-name>
    - property újradefiniálása
  - ordered (rendezett)
  - unique (egyedi értékek vannak benne)
  - <prop-constraint>
- kollekció típusok

isOrdered	isUnique	kollekció
FALSE	TRUE	Set
TRUE	TRUE	OrderedSet
FALSE	FALSE	Bag
TRUE	FALSE	Sequence

- enumeráció

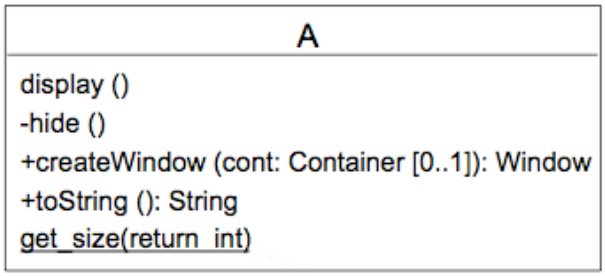


- lehet attribútoum egy másik osztály is, ez megfelel egy asszociációnak

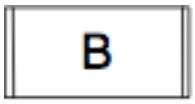


Class operáció:
- [<visibility>] <name> ‘(‘ [<parameter-list] ‘)’ [‘:’ [<return-type>]
  [‘{‘ <oper-property> [‘,’ <oper-property>]* ‘}’]]
- <parameter-list> ::= <parameter> [‘,’ <parameter>]*
- <parameter> ::= [<direction>] <paremeter-name> ‘:’ <type-expression> [‘[<multiplicity’]’]
  [‘=’ <default>] [‘{‘ <parm-property> [‘,’ <parm-property>]* ‘}’]
- <direction> ::= ‘in’ | ‘out’ | ‘inout’ | ‘return’

- oper-property
  - query (nem változtatja meg a szerkezetet)
  - redefines <oper-name>
  - ordered
  - unique
  - <oper-constraint>

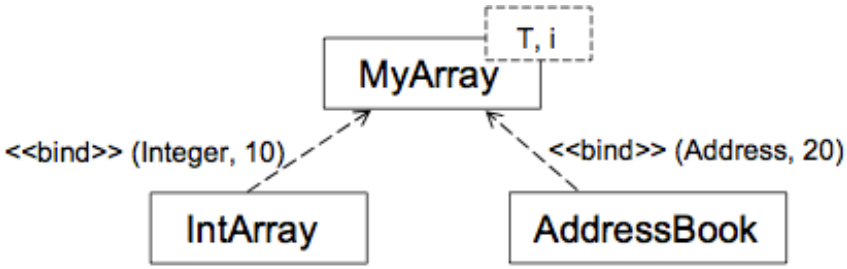


- **szemantika** (több utasítás érkezik be egyszerre)
  - *szekvenciális*: garantált hogy nem lehet verseny
  - *őrzött (guarded)*: a verseny elöfordulhat, de kezelve van
  - *konkurens*: a verseny elöfordulhat és próbálja kezelni mindet



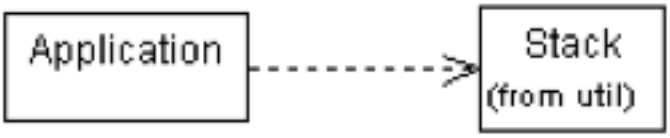
Aktív objektum:
- olyan objektum melynek saját szála van.

Template osztályok:
- parametrizált osztályok, pl.: típussal parametrizált



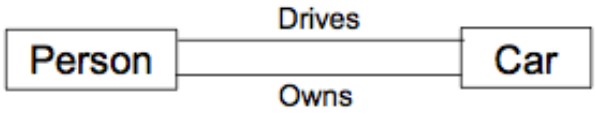
Classifier:
- “osztályszerü”
- osztálynak kinézö dolgok feletti metaclass, ez a classifier

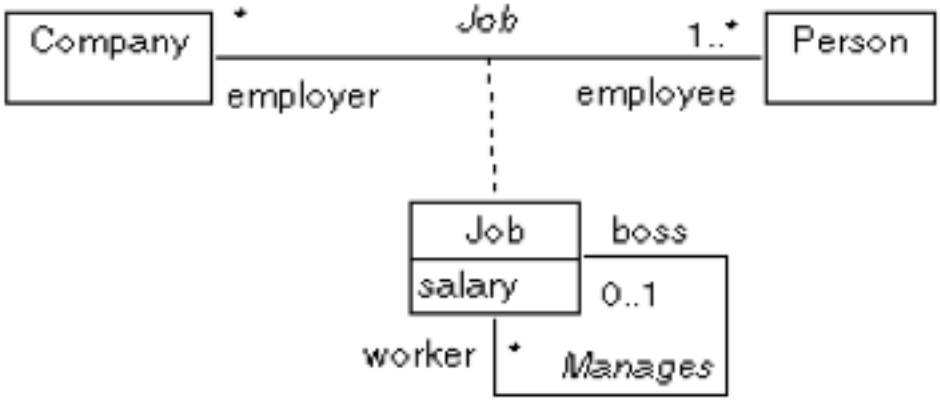
Relációk:
- függöség (defendencia)
  - ha használok egy osztályt akkor az függöséget jelent
  - szaggatott, irányított vonal a jele



- asszociáció

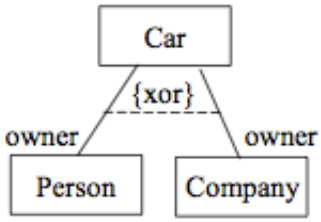
- szemantikus reláció
- osztály jelleggel viselkedik, példányosodik az asszociáció
- példánykapcsoladok (link) halmaza
- folytonos vonal a jele
- az asszociániónak neve is lehet





Példa: Állás asszociáció, melynek két vége van, ezeket is elnezevhetjük (role - szerepek). Multiplicitásuk is van. Egy cégnek egy vagy több alkalmazottja van. Egy személyhez 0 vagy több állása lehet. Hova kapcsolódjon a fizetés?
Asszociációs osztályt vezetünk be, melynek saját attribútuma van és az lesz a fizetés. A manages asszociációnál minden munkavállalónak van 0 vagy 1 fönöke. Minden fönöknek van beosztottja. Ez a jobok közötti kapcsolat. Itt fontos a szerep (role), hogy melyik végén vagyunk.

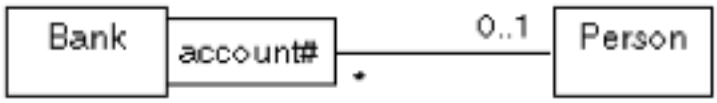
- lehet neve az asszociációnak
- ha '/'-t írunk elé, akkor örökölt asszociáció
- lehet constrait-et adni az asszociációnak -------->
- property string tartozhat hozzá
- n-es asszociáció
- specializálás, újradefiniálás is tartozhat hozzá



- multiplicitás
  alsó korlát .. felsö korlát
- navigálhatóság
  - a classifier egy példányából van közvetlen hatékony út az asszociációban álló másik példány eléréséhez
  - a nem navigálhatóságot X-et jelöljük (nem érhetö el hatékonyan)

Példa: A-ból irányított B felé, azaz A-ból el tudom érni B-t, például van A-ban egy pointer B-re. Itt a role fontos, mert a pointert role-nak tudok elnevezni (endB), és a role elé tudom írni a láthatóságot.

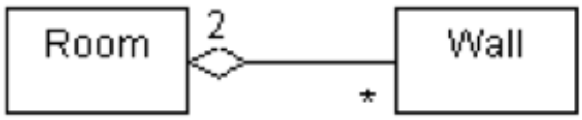
- asszociáció tulajdonos
    - alapaesetben az asszociáció a tulajdonos
    - ha pont áll a vona l végen akkor a classifier a tulajdonos
- qualifier, minösítö
    - az asszociáció multiplicitását csökkentjük ezzel
    - az asszociáció része a qualifier



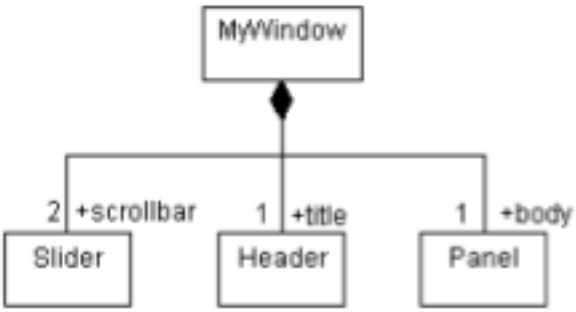
Példa: A bank-szemely között több-több kapcsolat. Ha felvesszük minösítönek a bankszámla számát, akkor egy számlához 0 vagy 1 személy tartozik.

- aggregáció

- rész-egész viszonyt jelenti (pl: autó és az ajtaja, az ajtó az autóval együtt születik és azzal együtt is hal meg, együtt élnek-halnak
- bináris asszociáció
- osztott reláció
    - egy dolog nem egy egésznek a rész, hanem több egésznek a része (egy fal több 2 szobához tartozik)

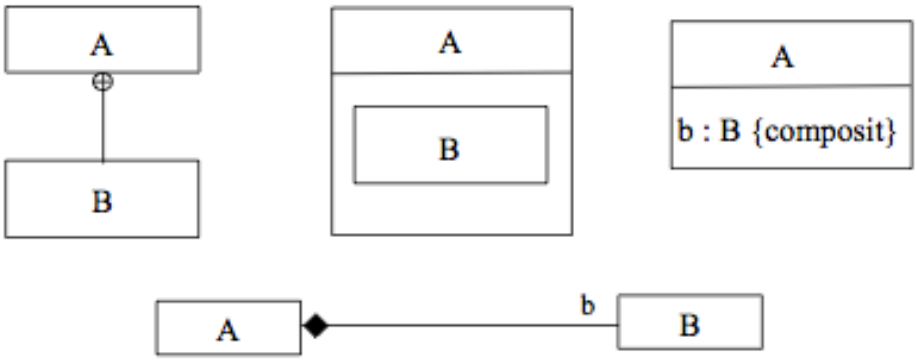


- komponens reláció (autó - ajtaja)
- kompozíció:
    - egy egésznek sok kisebb része van
    - propagációs szemantika: az egészen értelmezett müvelet a részeken végzett müveleket összessége

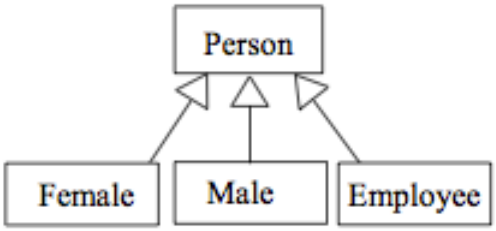


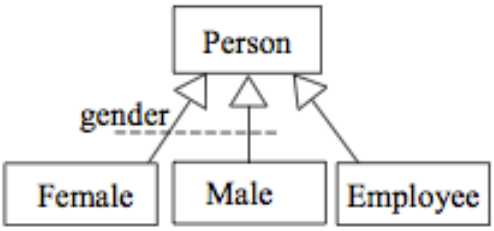
- beágyazott osztályok (nested class)

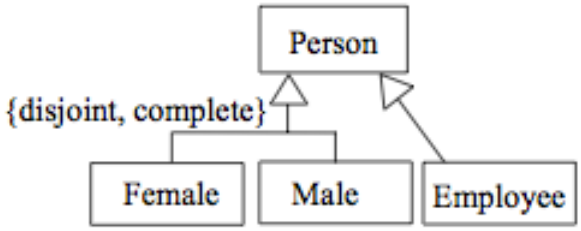
- a beágyazott osztály az az öt tartalmazó osztályon belül látható

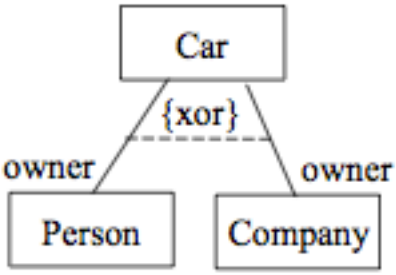


- generalizáció (öröklés)

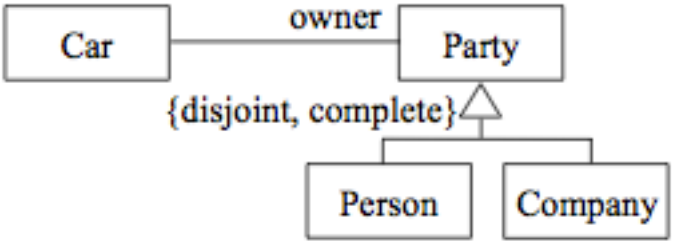




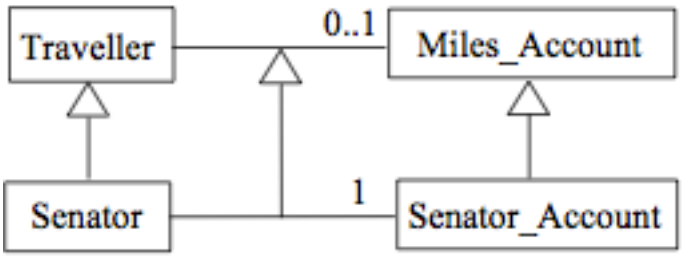




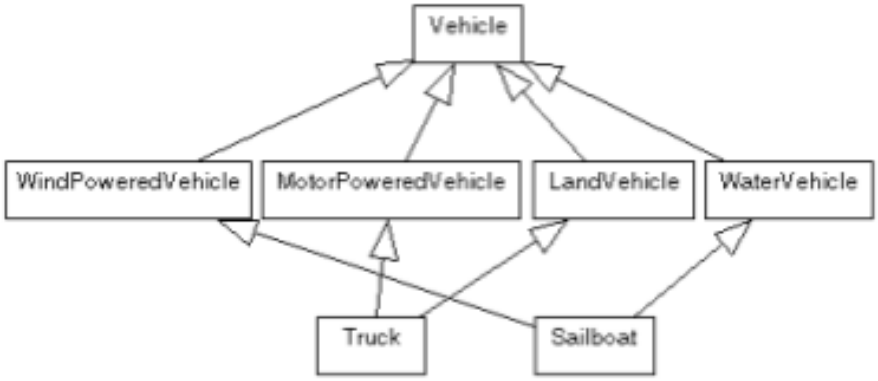
Példa: Autónak lehet személy vagy cég tulajdonosa. Az xor egy korlátozás.



Példa: Az autónak Party a tulajdonosa, és a Partyt örököltetjük le Person és Company-re.

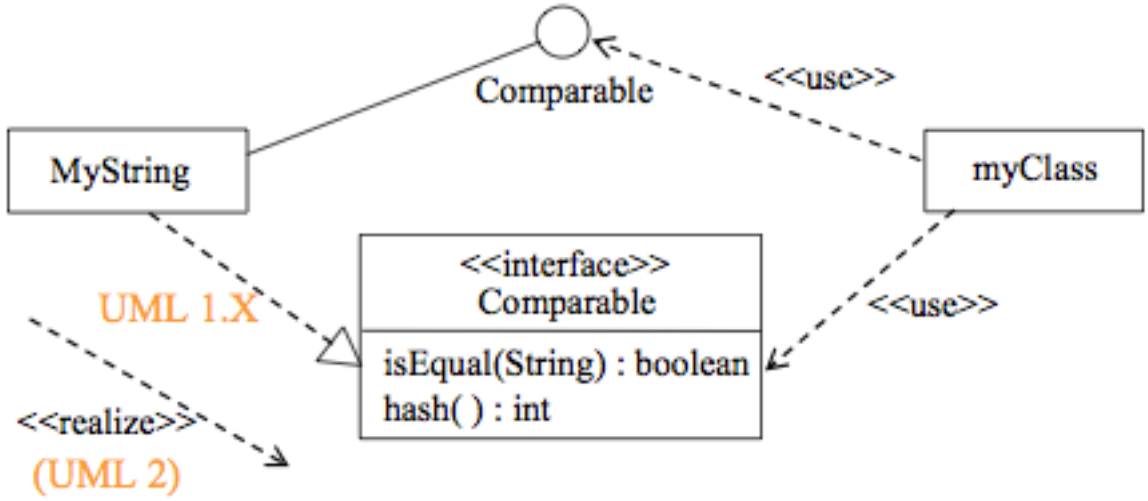


Példa: Az utashoz tartozik mérföld számla vagy nem. A szenátor egy speciális utas, és hozzá is tartozik egy szenátor számla. A szenátor és szenátor számla asszociációja örököltetve van az utas és a mérföld számla közötti asszociációból.

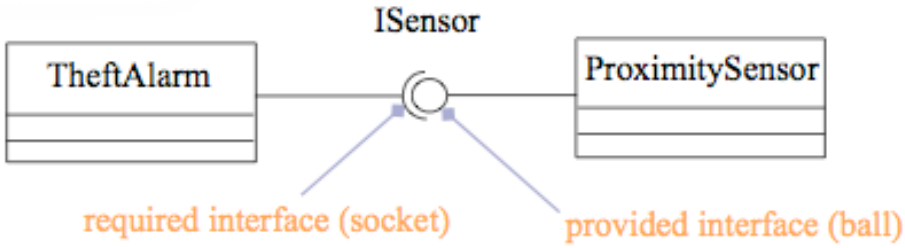


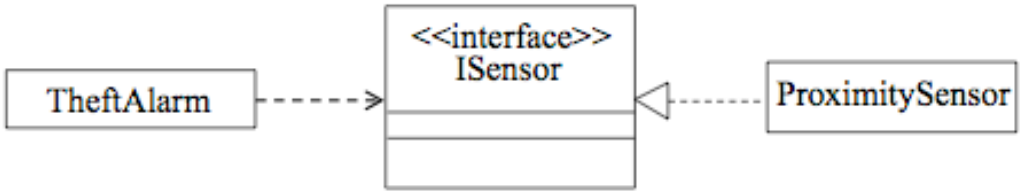
Példa: Az UML-ben is létezik többszörös öröklés.

Interface:
- az objektum által nyújtott szolgáltatás halmaz, de belevesszük az attribútumokat is
- ha az interface attribútumokat deklarál, abból nem következik az, hogy az attribútumok attribútumként érhetöek el (pl.: get vagy set metódussal csak)



Példa: A myClass használni akarja a Comparable interfészt, abban van két müvelet. A myClass használni akar egy komparálható valamit. A MyString implementálja ezt az interfészt (szaggatott vonal, és öröklési háromszög fej, mi ez a jelölést használjuk). A fenti jelölés a nyalóka jelölés.

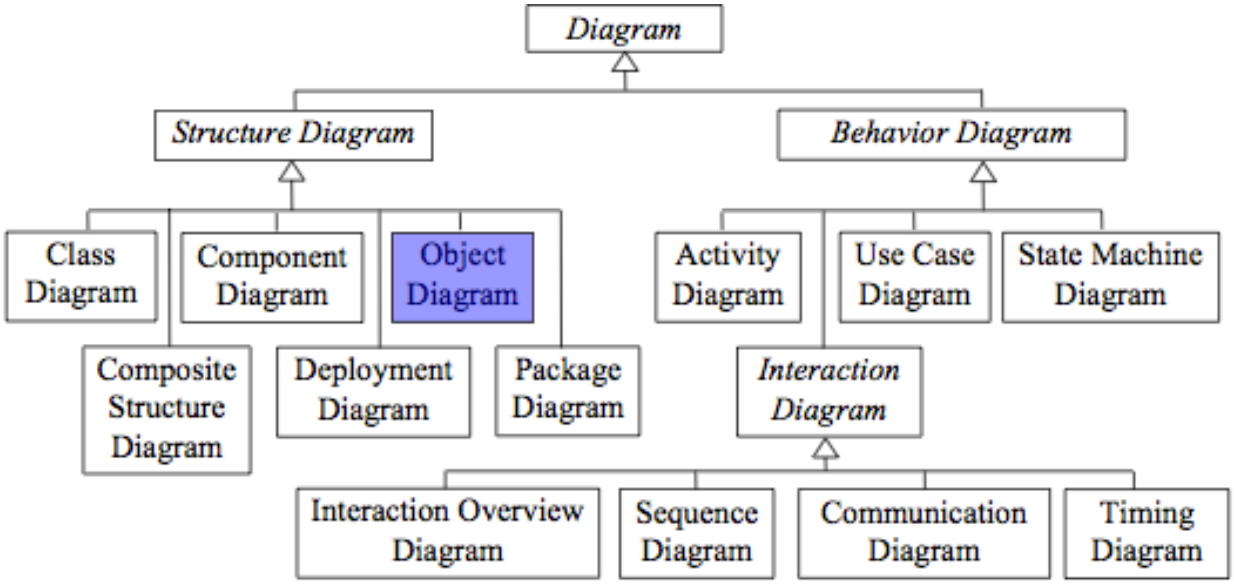




Példa: A szolgáltatott interfész a gombóc. Az elvárt interfész (socket) a félkör. Így ábrázolni tudjuk az elvárt interfészt.

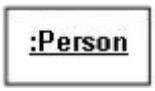
Feladatok Class diagram témakörböl (KATT IDE)

UML Diagramok - Object Diagram



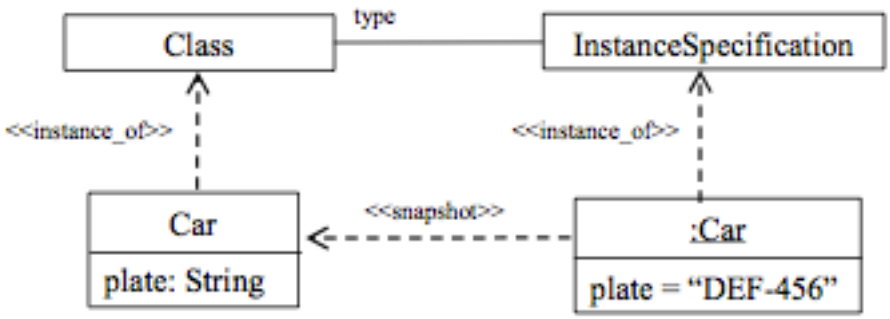
Példa: Jim akinek Person az osztálya.

Példa: Jim, akinek nem tudom az osztályát.

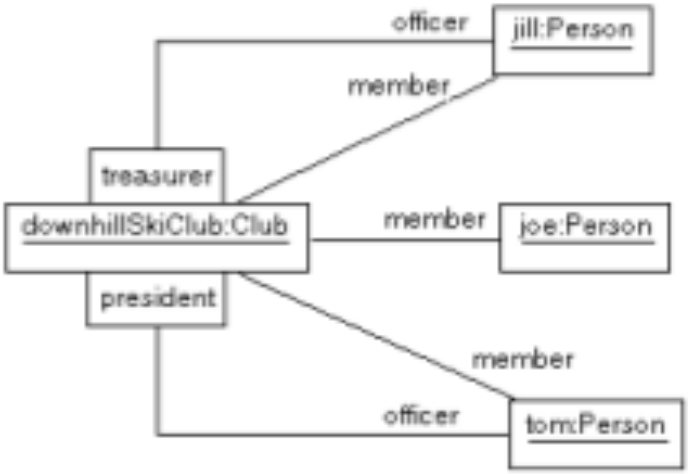


Példa: Egy Person osztálynak a példánya, akinek nem tudom a nevét.

Object diagram:
- objektum példányokról beszélünk
- pillanatfelvételeket ábrázolnak ezek a diagramok

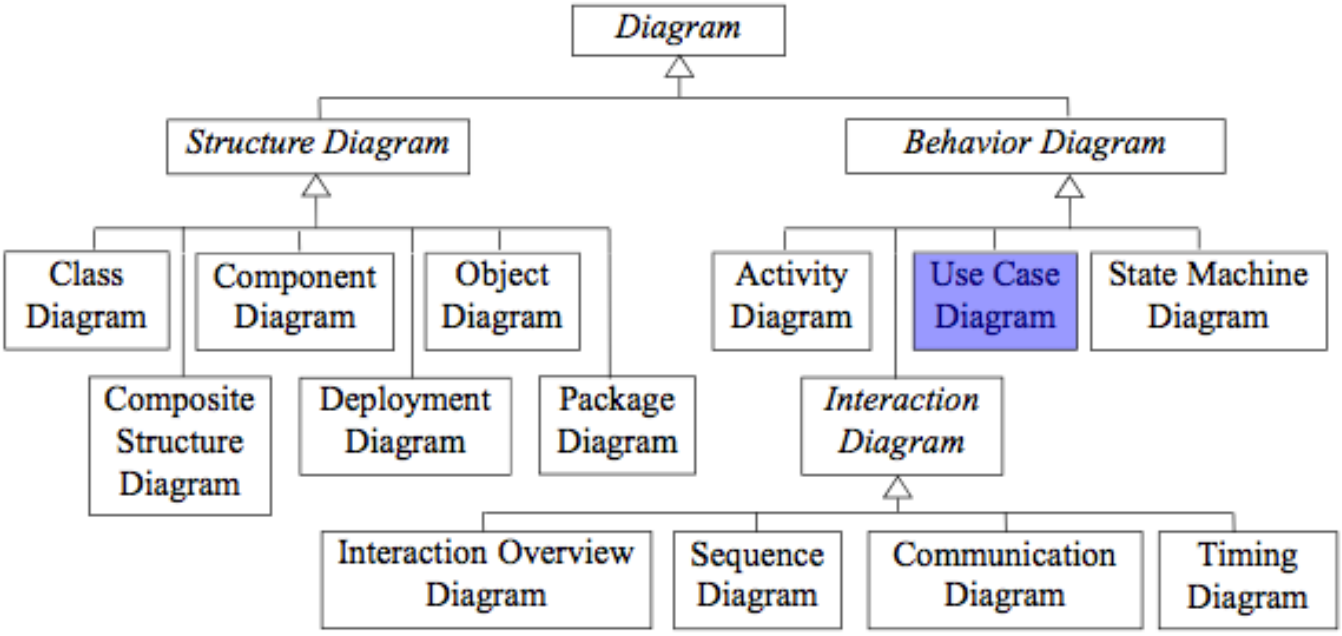


Példa: Az UML Class-nak egy példánya a Car osztály. (A Class egy metaclass).
A Car osztálynak példánya a Car, aminek DEF-456 a rendszáma. A Car pedig a példányspecifikáció (InstanceSpecification) példánya. Ez egy modellbeli példány.

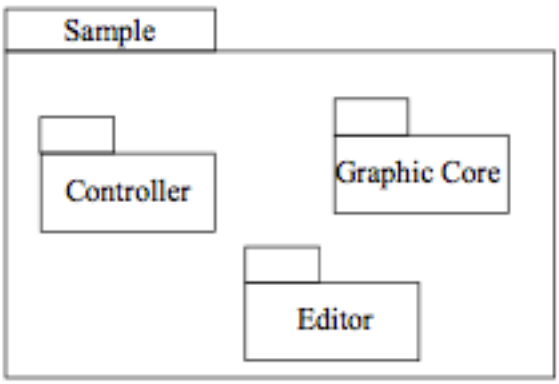


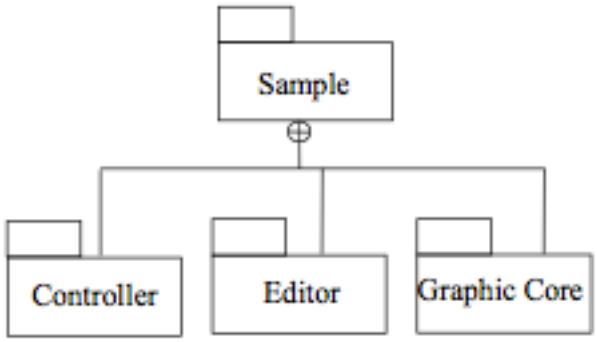
Példa: Háromtagú klub, melyben vannak tisztviselök (pénztáros és elnök - qualifier).

UML Diagramok - Package Diagram



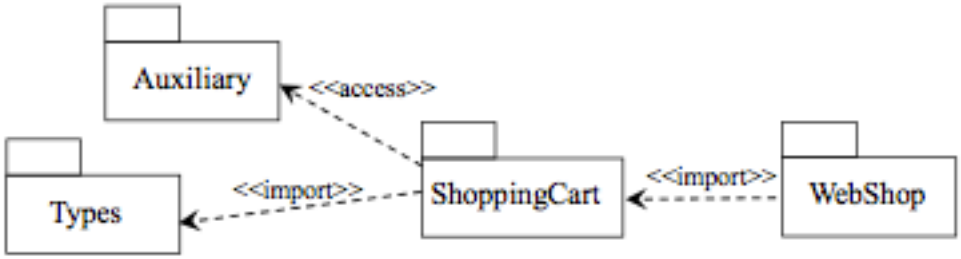
Package:
- meghatároz egy szerkezetet
- azon elemek melyek egy bizonyos névteret alkotnak
- minden elem csak egy package-ben lehet benne





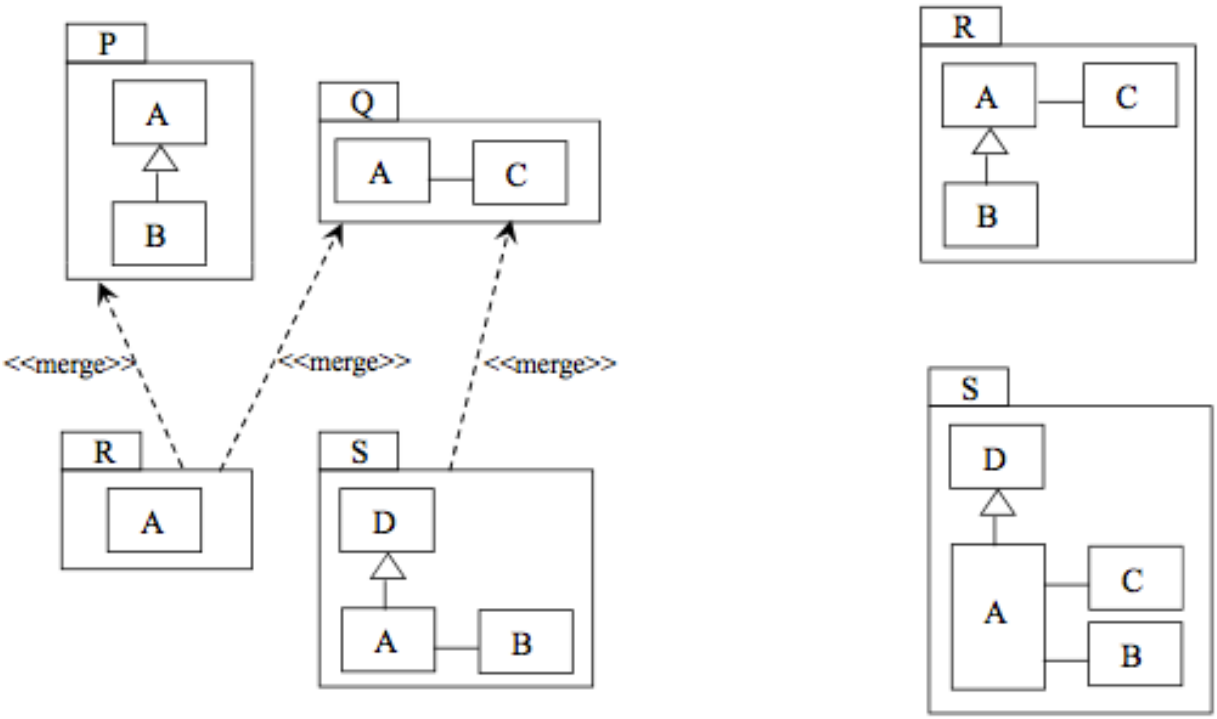
Láthatóság:
- private
- public

Package import:
- egyik package tartalma egy másik package-ben is látható
- 2 sztereotípia:
    - <<import>> - public
    - <<access>> - private

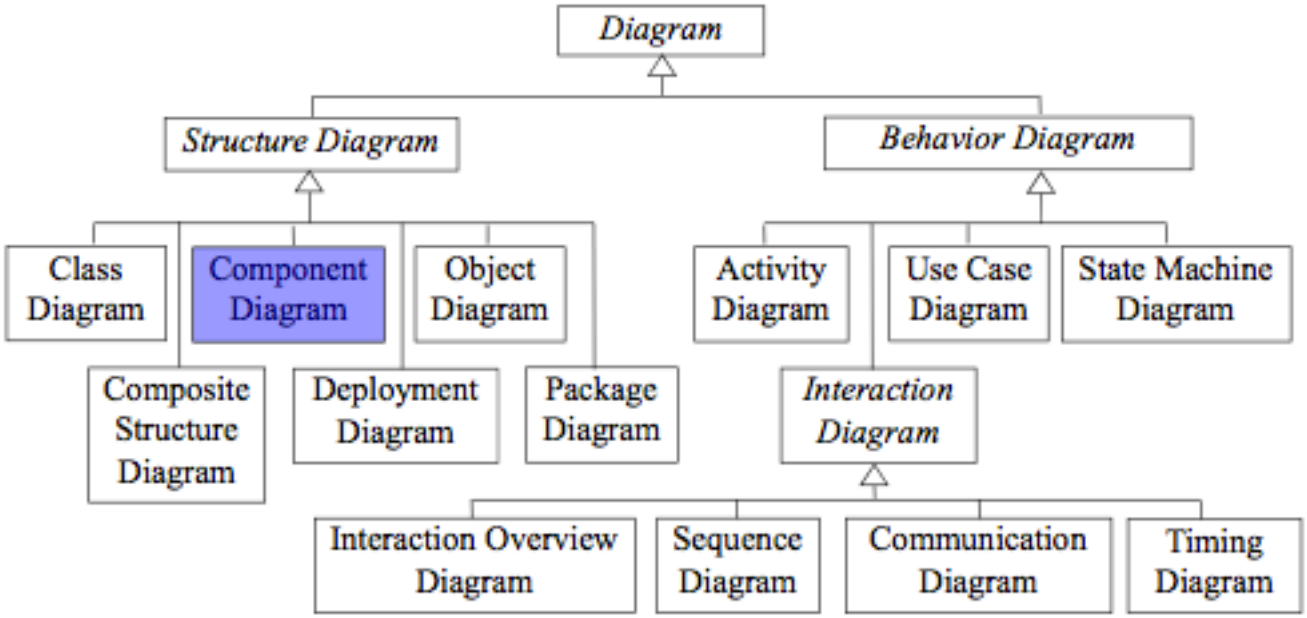


Példa: A ShoppingCart-ban használhatjuk az Auxiliary és Types elemeket. Az access és import ezt valósítja meg. A WebShop szempontjából érdekes. A ShoppingCart importálta a Types-t ezért a WebShop is látja a Types elemeket, de az Auxiliary-t nem.

Package merge:
- az azonos elemeket megpróbáljuk összehozni

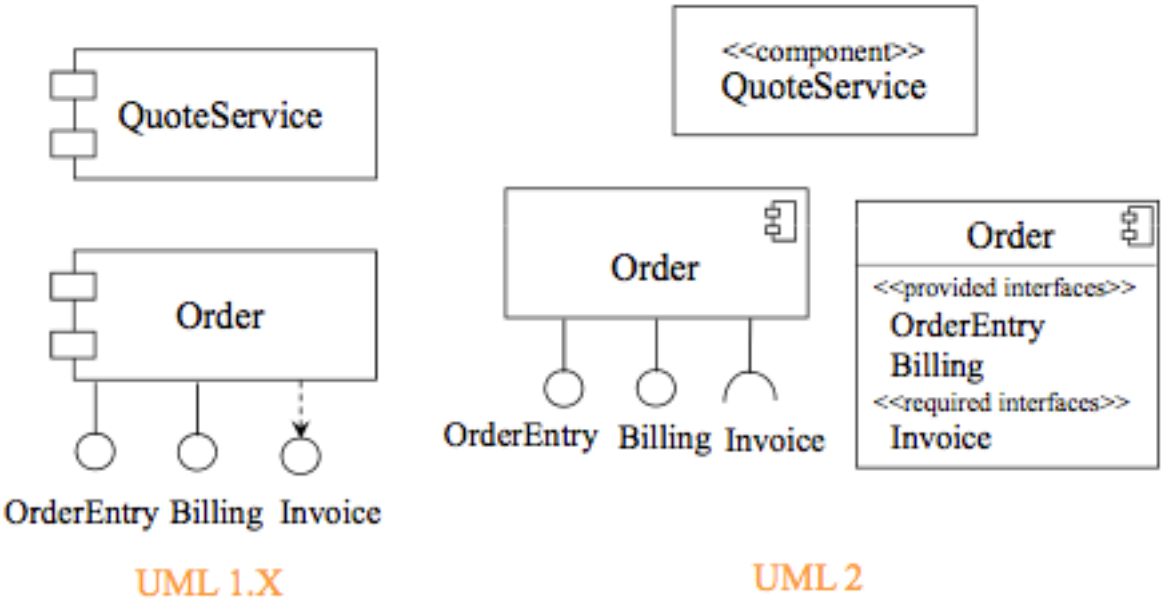


UML Diagramok - Component Diagram

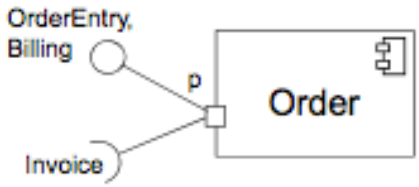


Komponens:
- fizikailag létezö, helyettesíthetö része a rendszernek, mely valamilyen interfészt szolgáltat és számít bizonyos interfészekre

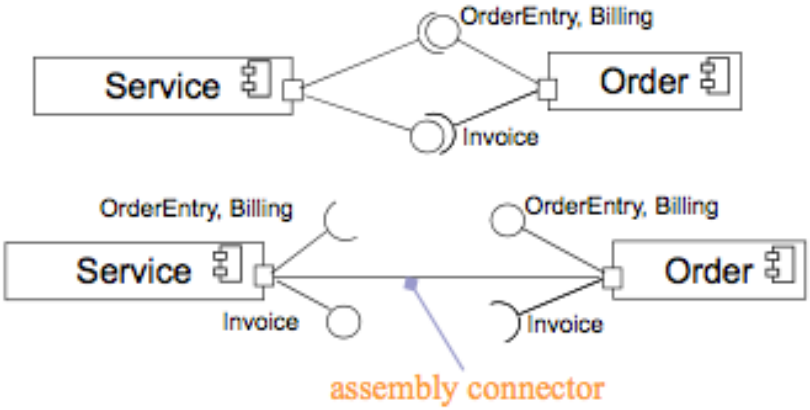
- deployment komponensek
  - program telepítéséhez szükséges elemek
- work product komponensek
  - fejlesztés során keletkezö elemek
- execution komponens
  - müködés közben létrejövö elemek



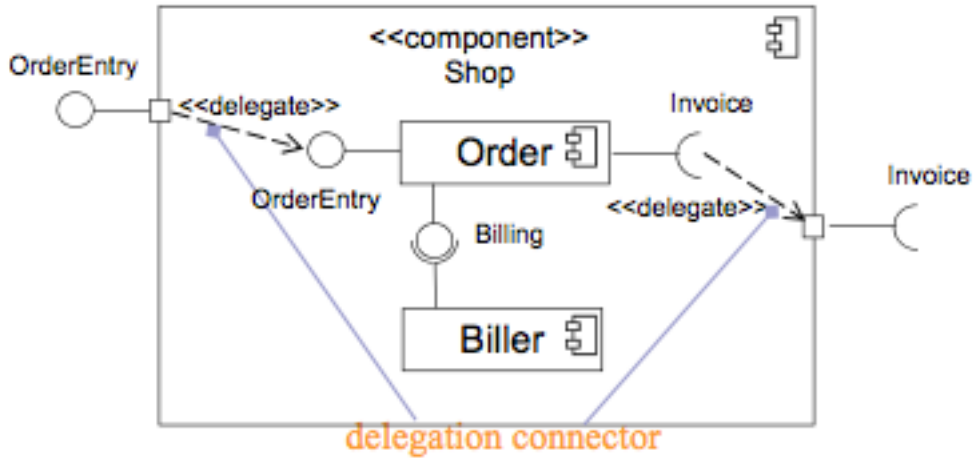
Port:
- definiált csatlakozó pont amelyen keresztül bizonyos interfészek elvártak és elérhetöek
- pl.: USB port ami elvár interfészeket és megvalósít interfészeket - így van a szoftvernél is



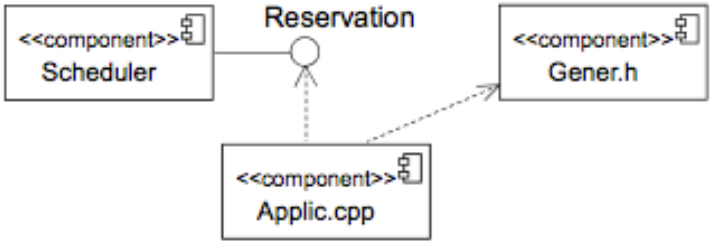
Konnektorok:
- assembly konnektor: összeköt két egymásnak megfelelö portot



- delegation konnektor: egy bonyolultabb dolgot egy ‘dobozba’ zár

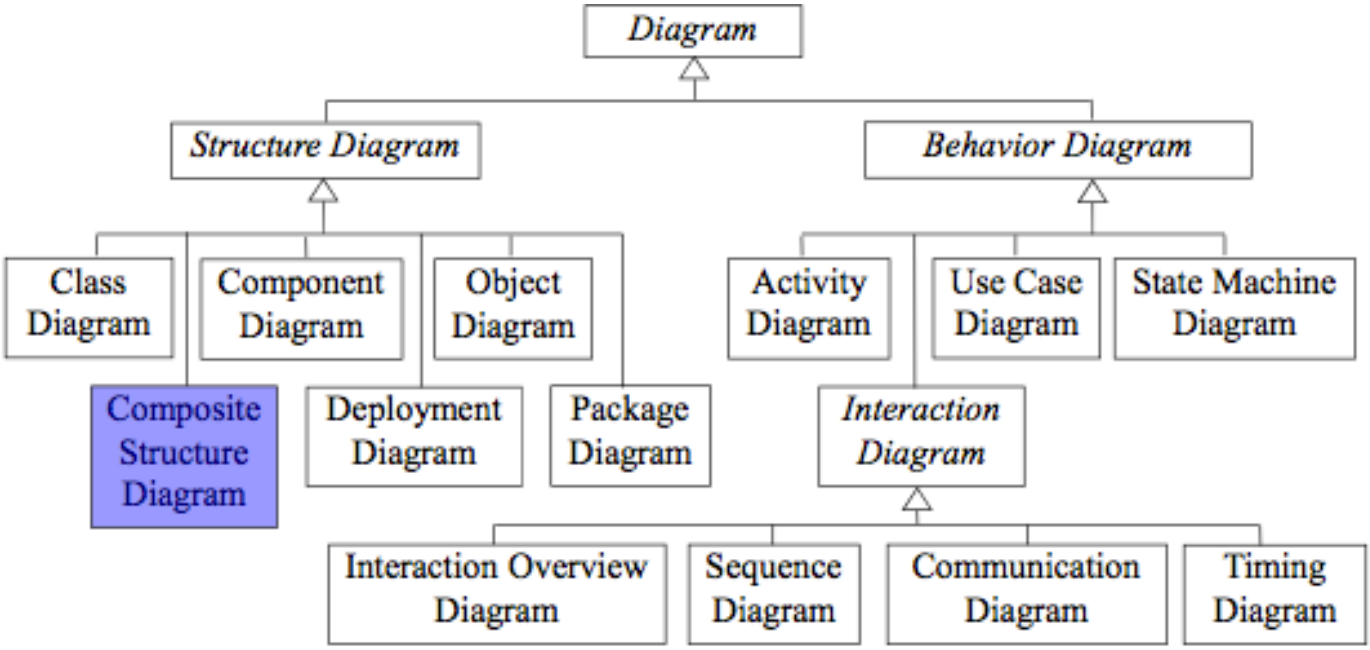


Komponens diagram:
- komponenseket, interfészeket és a köztük lévö kapcsolatokat írjuk le vele



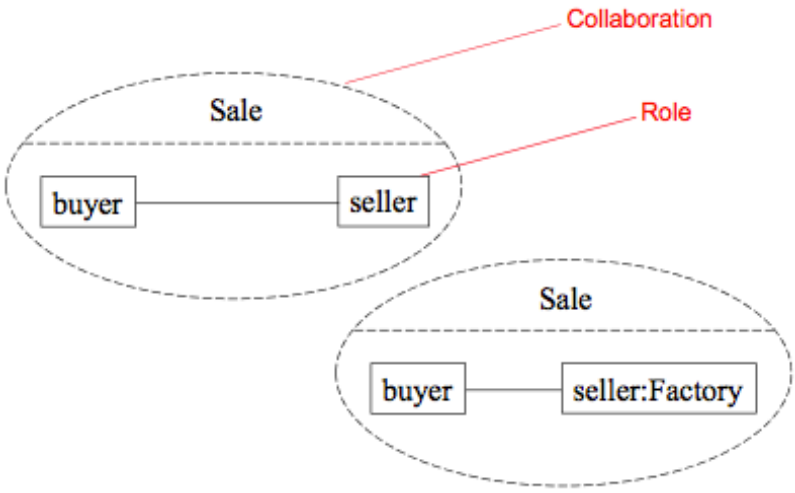
[Feladatok komponens diagram témakörböl (KATT IDE)]

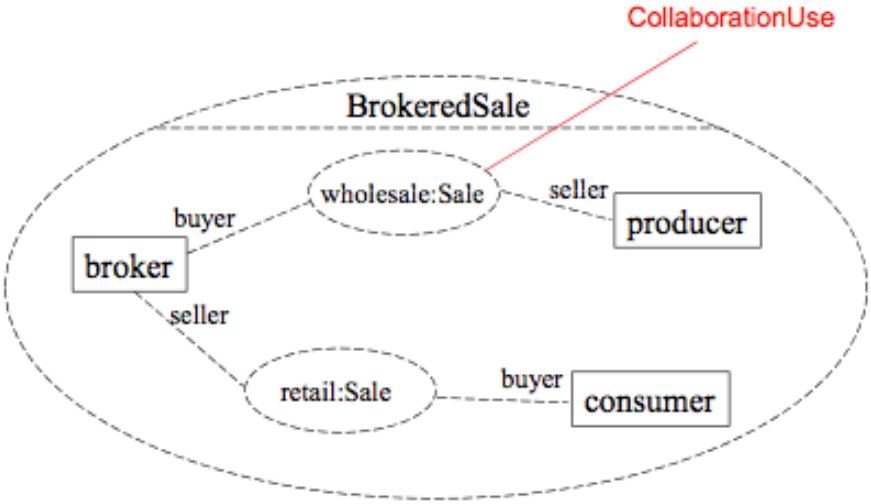
UML Diagramok - Composite Structure Diagram



Kompozit struktúra:

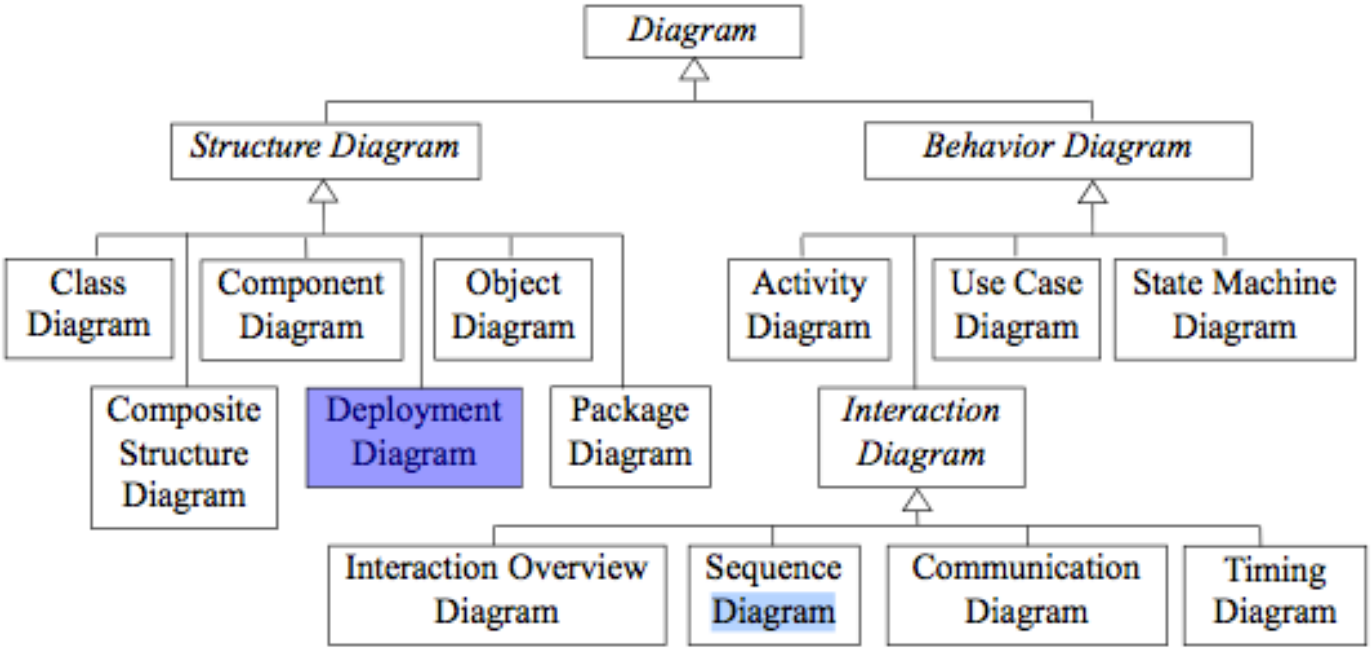
- Mire jó?: Ha van egy nagy class diagramunk, és meg akarjuk nézni hogy egy funkcióhoz milyen osztályok tartoznak - ezt ábrázolja a kollaborációs diagram
- **kollaboráció:** egy meghatározott funkcionalitás megvalósításához szükséges együttműködö elemek



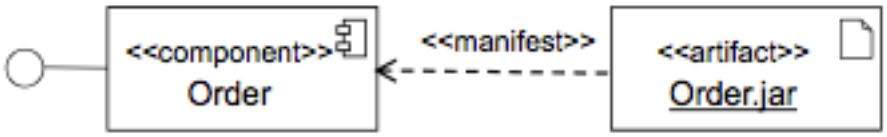


Példa: Kollaboráció használat (szaggatott vonallal). A közvetítön keresztül kereskedelemben felhasználjuk a Sale kollaborációt.

UML Diagramok - Deployment Diagram

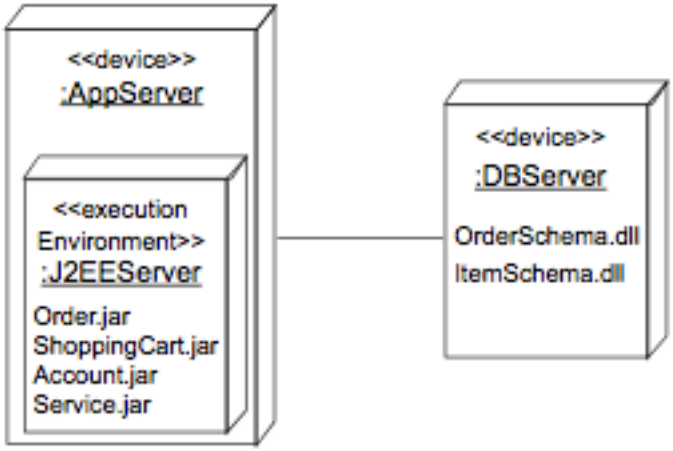


Artifact (termék, produktum):
- egy komponenshez tartozik egy termék (pl.: megjelenik egy .jar formátumban)

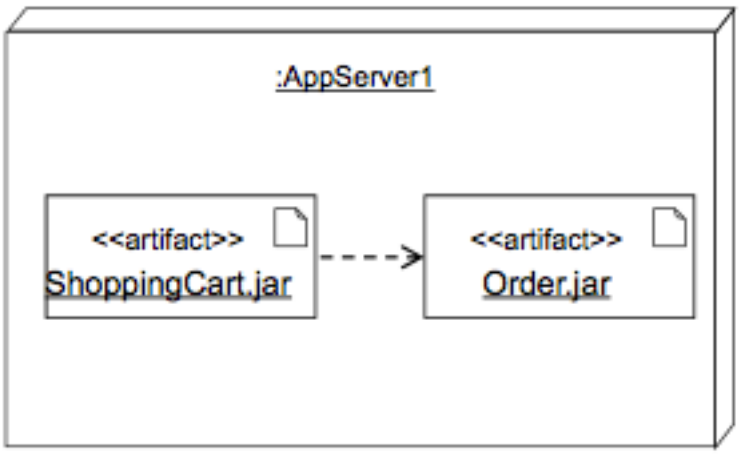


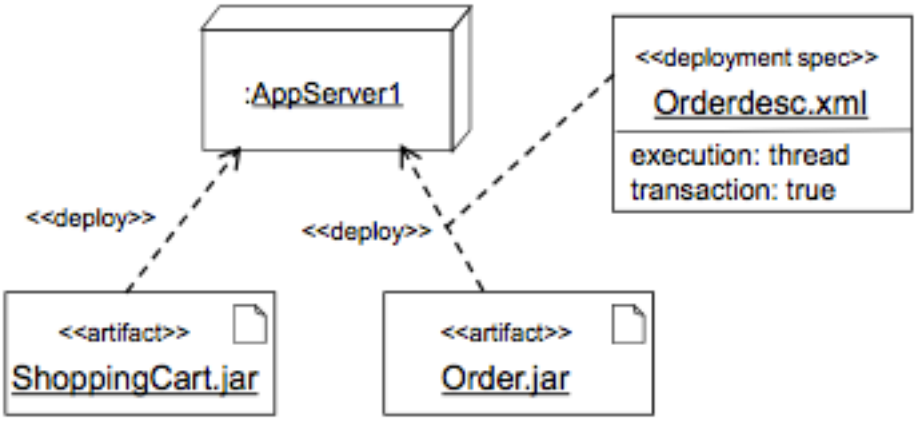
Node:
- számítástechnikai eröforrás, melyen termékek telepíthetök végrehajtás céljából
- logikai értelmü szerkezet (a device lesz a fizikai szerkezet)

Device:
- fizikai számítási képességekkel rendelkezö rendszer, amelyen az elemek telepíthetöek végrehajtás céljából
- használjuk osztály és objektum példányként is
- fizikai példánya a node-oknak
- hozzá tudunk rendelni egy végrehajtási környezetet (executionEnvironment)



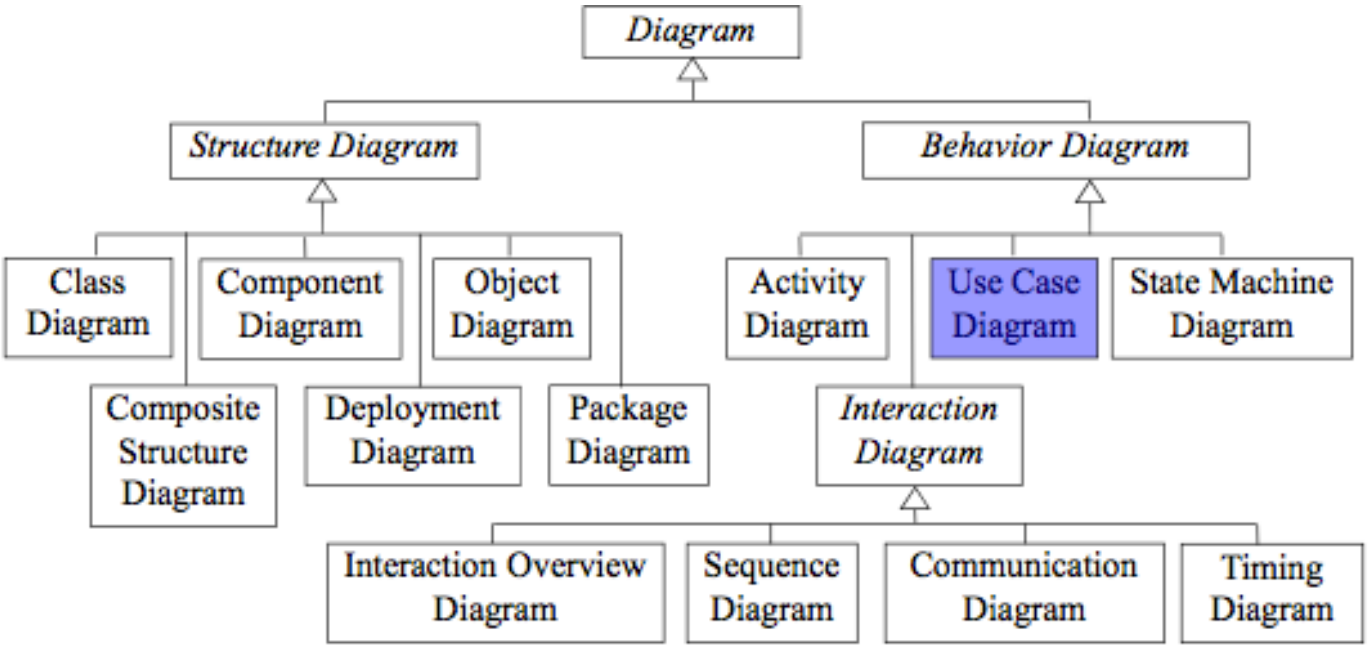
Deployment diagram:
- hogyan helyezzük el a node-okon és a device-okon az egyes termékeket (artifract-okat)
- mind típus, mind instance szinten szerepel





Példa: Elözö ábra úgy hogy az AppServer1-re telepítettük a ShoppingCart.jar-t és az Oder.jar-t. Így külön megjelenítjük a deploy-t, egy deployment-et meg tudunk adni, leírás arról hogy hogyan történik a leírás (tipikusan XML).

UML Diagramok - Use Case Diagram



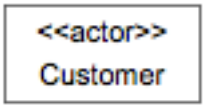
User Case (használati eset):
- elemi funkcionalitás
- actor tartozik egy use case-hez, az actor kezdeményezi és bonyolítja le a use case-t
- leírás az actor és a rendszer közötti együttmüködésröl
- az actor egy viselkedést definiál,mindegy ki végzi, csak úgy viselkedjen ahogy le van írva

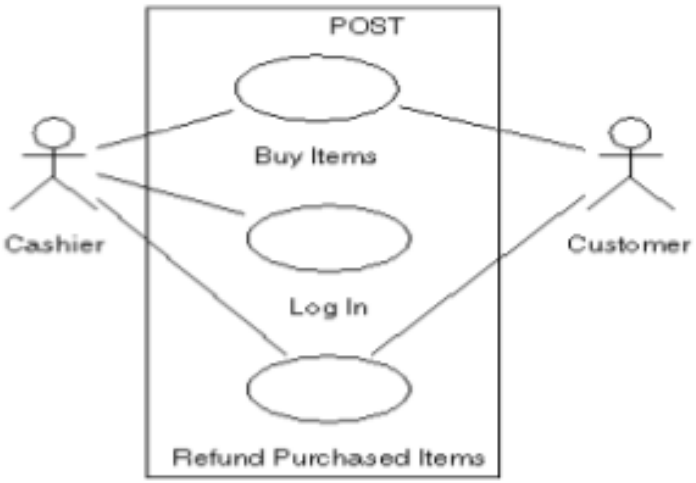
- a use case egy táblázatos leírás lényegében + szereplök cselekedetei

Use Case	Vásárlás
Actor-ok	Ügyfél, kasszás
Leírás	Az ügyfél megérkezik a termékekkel. A kasszás összeszámolja az árkat, az ügyfél fizet.
Követelmények (kereszt ref.)	Referenciák a szükséges dokumentumokra

+ táblázat arról hogy az ügyfél egyes viselkedéseire mit reagál a kasszás és a rendszer

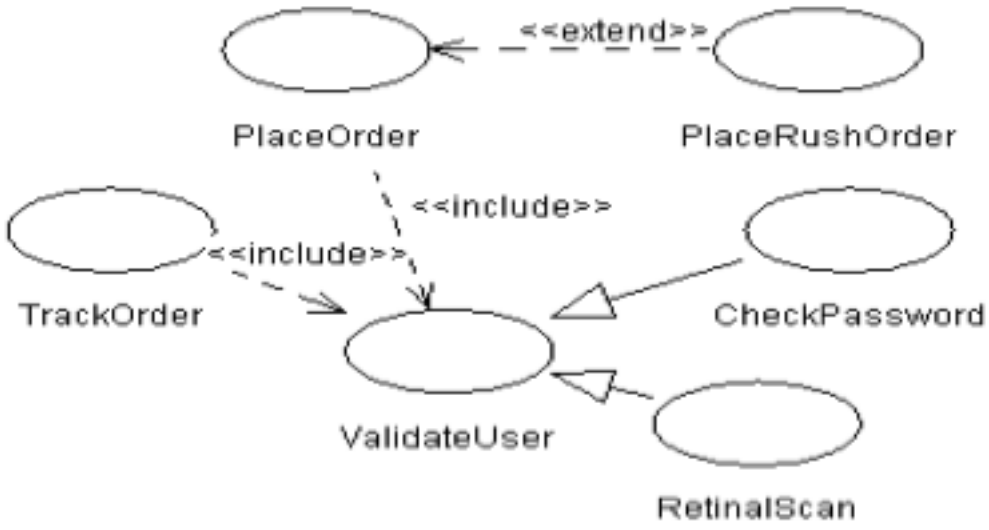
Use Case Diagram:
- statikus képet ad hogy kik az együttmüködök és a szerepek
- mik vannak a diagramon?
    - use case
    - actor - - - - - - - - >
    - relációk



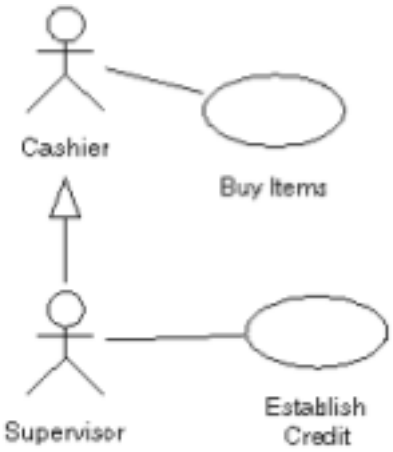


Use Case relációk:

- asszociáció
  - tipikusan nyíl nélkül, két irányí asszociáció
  - multiplicitás is elöfordulhat
    - több use case: megnyitunk laptopon több browsert és használjuk öket
    - több actor: (idöbeliségben lehet különbség)
      - atomtámadáshoz 3 tábornok beleegyezése kell
      - nagyival telefonálás, kézröl-kézre adjuk a telefont
- include
  - a nyíl irányába mutató use case bele van építve a nagyobb use case-be
  - kötelezöen tartalmazza az include-olt elemet
- extend
  - a megcélzott elemet bizonyos körülmények között kiegészítjük bizonyos dolgokkal



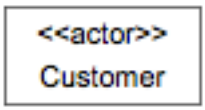
Példa:
Internetes webáruház. A vásárló azonosítása a ValidateUser, a PlaceOrder és a TrackOder is használja ezt, mindkettö azonosítja a vásárlót. A ValidateUser-nek két fajtája van, egy jelszó és egy retina ellenörzés, mindkettö a ValidateUser egy specifikus megvalósítása. Ez egy leszármaztatás mert mindkettö állhat a ValidateUser helyében.
A sürgös megrendelés kiegészíti a sima rendelést. Ez nem kötelezö, csak egy lehetöség.



-generalizálás (leszármazás)
-actorok közötti leszármazás

Példa: Van egy kassza felügyelö, aki hitelt is tud adni a vásárlónak. A felügyelö a kasszás leszármaztatottja. A kasszába beülhet a felügyelö, de a kasszás nem adhat hitelt.

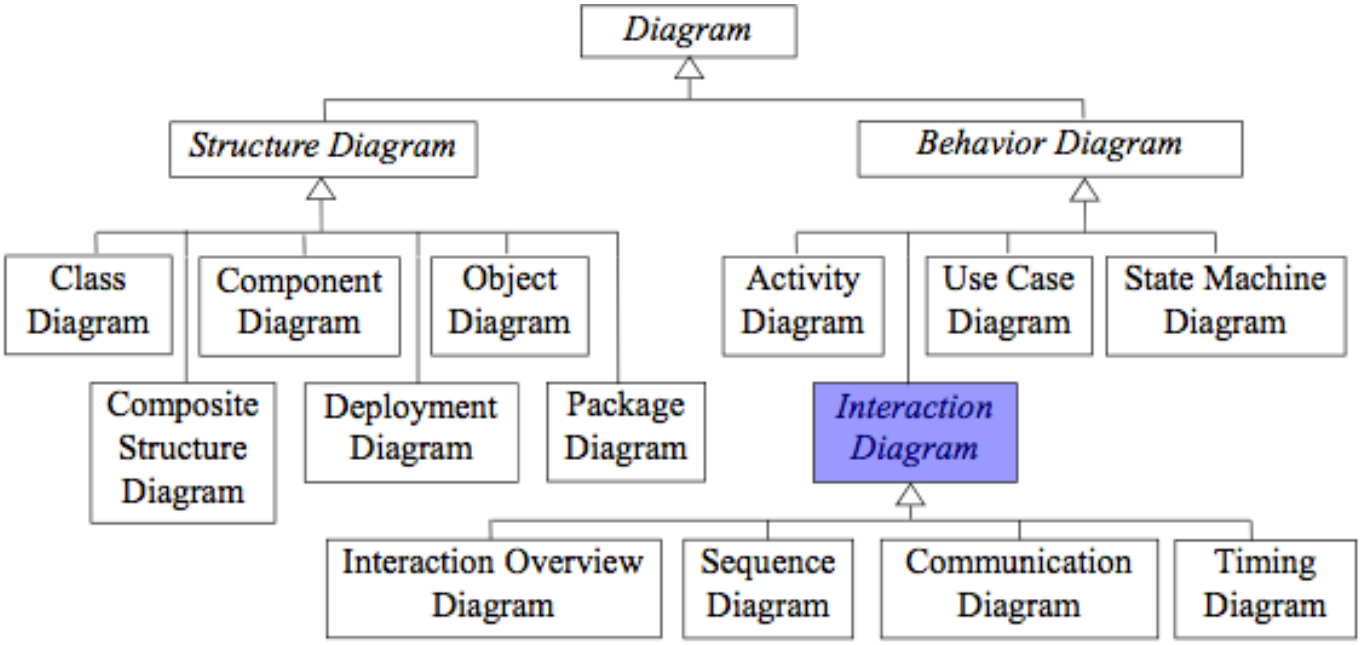
Belsö actor:
- létezik olyan a rendszerben hogy belsö actor
- pl.: tetrixben amikor esik le az elem azt egy actor végzi, az az actor egy belsö actor
- belsö actort szoktuk diagramon az osztály szerü jelöléssel jelölni --->



Use case (táblázatos leírás) egyéb jellemzök:
- speciális követelmények (biztonság, titkosság)
- elöfeltétel (precondition)
- utófeltétel (postcondition)
- kiterjesztési pontok (extension point)
- relációk
- activity diagram (flow chart)

Feladatok Use Case Diagram témakörböl (KATT IDE)

UML Diagramok - Interaction Diagram



Interakció diagramok:
szekvencia / kommunikációs / interaction overview / timing diagram és interaction table

Az objektumok együttmüködnek. Eddig a szerkezettel foglalkoztunk, most pedig a dinamikával foglalkozunk, azaz hogy milyen sorrendben cselekednek.

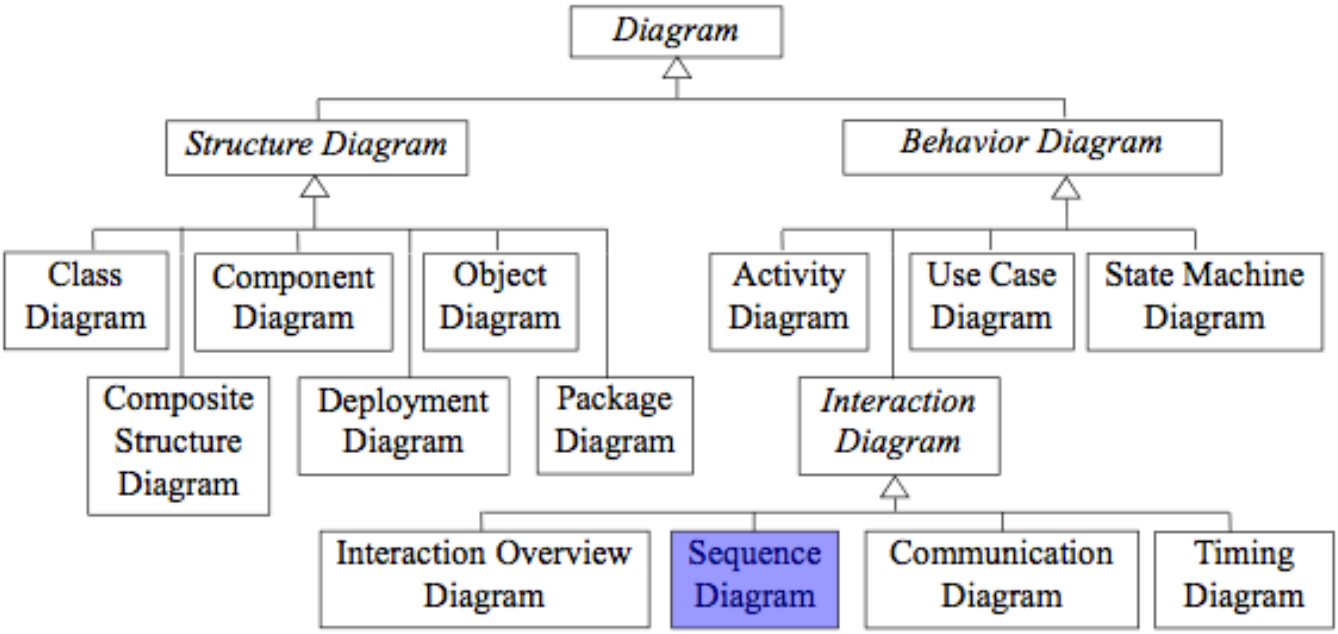
Interakció: együttmüködés melyet egy classifierhez rendelünk. Trace-ekböl áll elö a viselkedés (trace = esemény sorozat). Az esemény pedig ugyanazt jelenti mint eddig, egy rendszert érö külsö esemény érkezik melyre reagálni kell.

Események:
- ExecutionEvents: minden operáció kezdete és vége
- CreationEvent: objektum létrehozása event
- DestructionEvent: objektum pusztulása event
- MessageEvents: absztrakt event
    - message: egy specifikáció mely leírja hogy milyen kapcsolatnak kell lenni a kliens és a szerver között amely meghatároztta, hogy milyen operációt fogunk végrehajtani
    - SendOperationEvent: kliens oldalon
    - RecieveOperationEvent: kiszolgáló oldalon

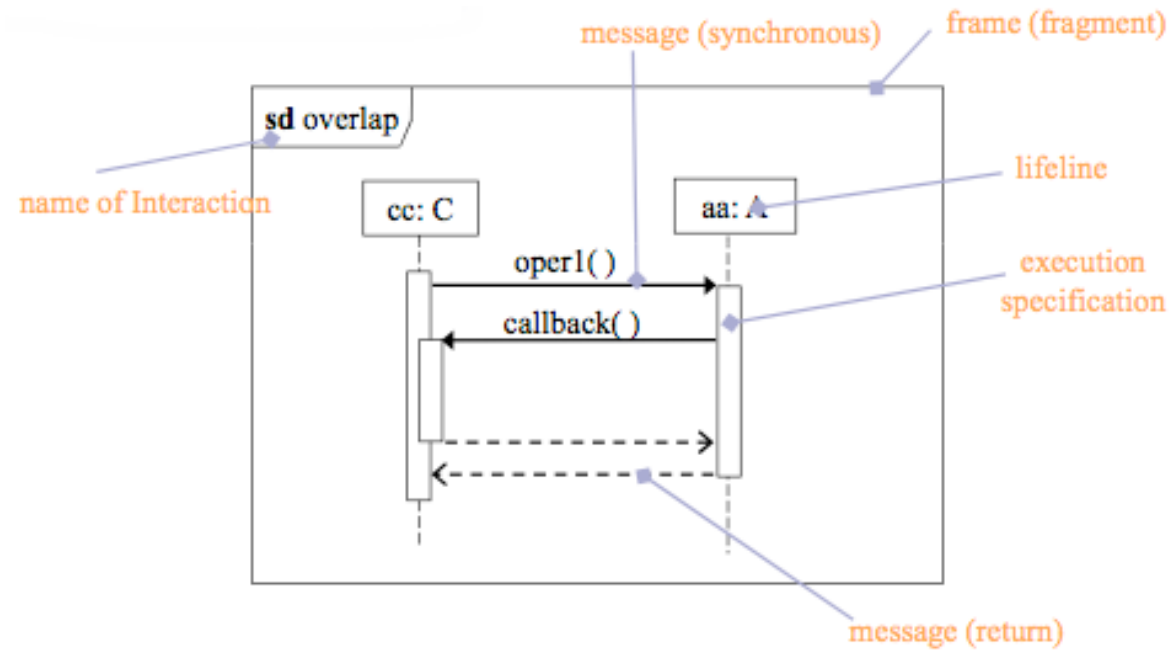
Signal: Üzenet célja az, hogy operációt hajtsunk végre, a signal pedig állapotváltozást fog okozni. Mindig aszinkron hívás. SendSignalEvent, RecieveSignalEvent.

Interakció:
- trace-ek halmazaiból egy pár, 2 db eseménysorozat pár. Egyik halmaz: érvényes esemény sorozat, másik halmaz: érvénytelen esemény sorozat.
- kompozíció elv: ha van két esemény sorrendben, akkor ebböl összekomponálhatok egy újabb összetett eseménysorozatot
- átlapoló szemantika: azt megtehetem, hogy összekomponálok két eseménysorozatot úgy, hogy csak az eredeti eseménysorozatok sorrendje maradjon meg

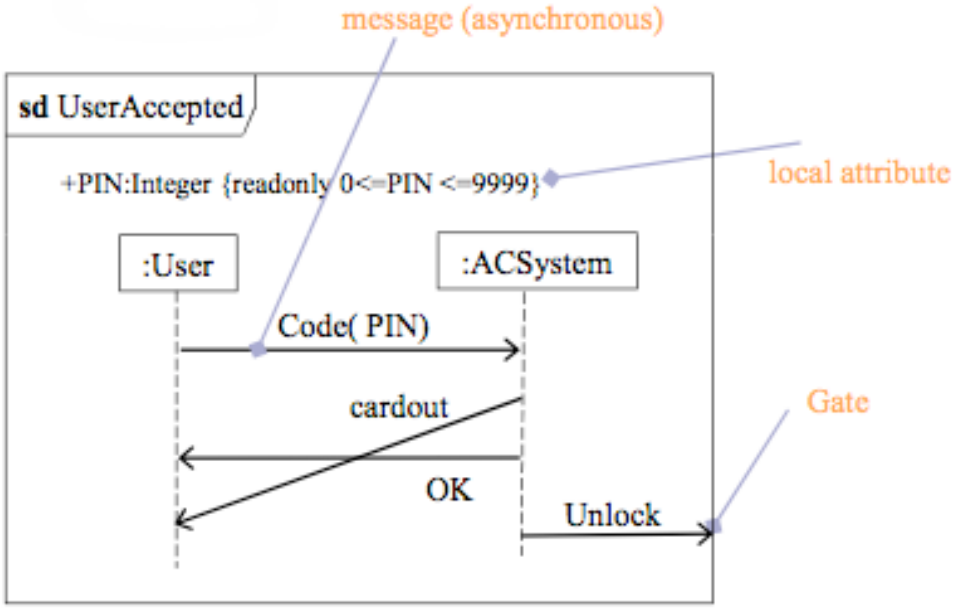
UML Diagramok - Szekvencia Diagram



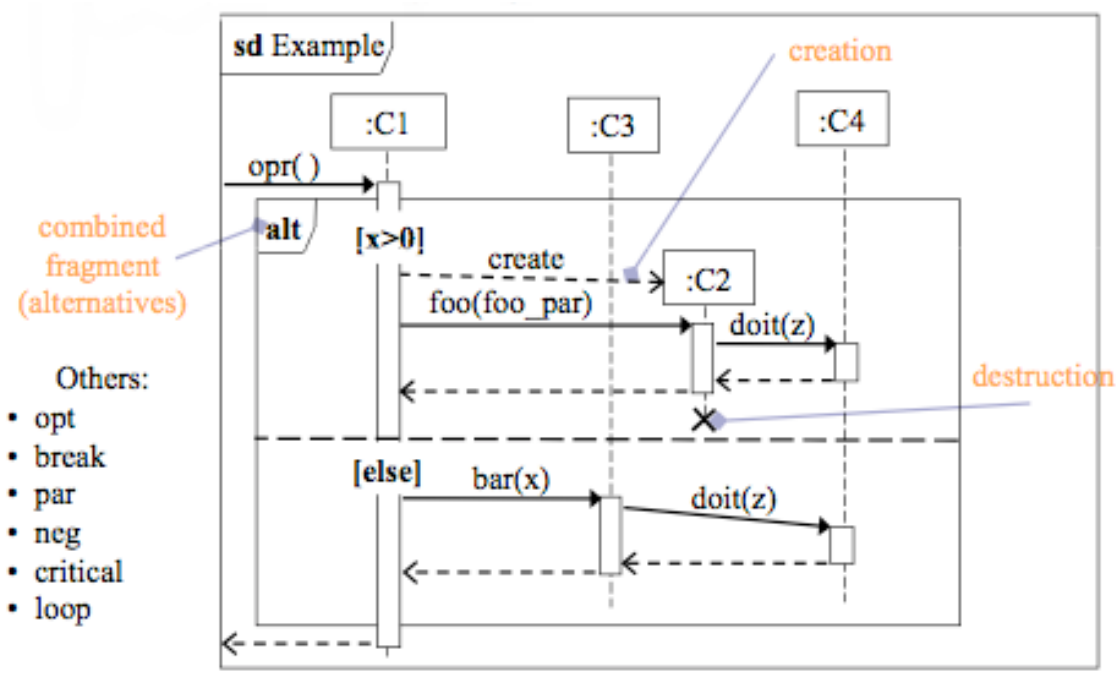
UML2-ben nincs aláhúzva az osztály/objektum, mert nincs eldöntve hogy objektum vagy osztály. UML1-ben még alá volt húzva.
Lifeline (életvonal): idöbeli elrendezés, egyes osztályok között menö üzenetek hogy helyezkednek el. Az egész körül van egy frame, aminek neve van.



Példa: C elkezd müködni és oper1 szinkron üzenetet küld A-nak (tele fejü nyíl = szinkron üzenet). C átadja a vezérlést A-nak, majd visszatér alul a szaggatott nyitott fejü nyíllal. Akkor kötelezö abrázolni a visszatérést ha fontos visszatérési érték van. Az A a visszatérés elött még végrehajtja a callback szinkron függvényt. Leolvasható a hívási stack a dobozokból.

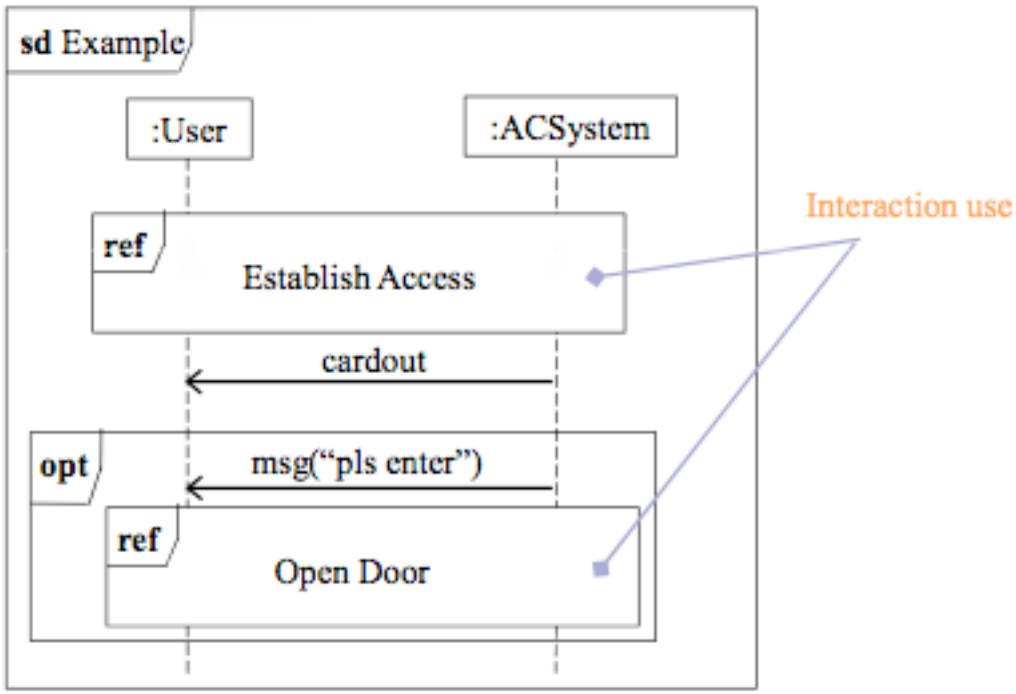


Példa: Itt nincs ábrázolva az execution bar, mert nem kötelezö ábrázolni. Aszinkron üzenetet küld a felhasználó az ATM-nek. Elküldök egy cardout üzenetet és egy OK üzenetet. A cardout üzenet mechanikus, ezért tovább tart kiadni, minthogy csak kiírjuk az OK-t a képernyöre. A felhasználó elöször az OK-t látja, utána az üzenetet, de a rendszer fordítva küldte el. Tudunk üzenetet küldeni a doboz határára, mert több doboz egymásba lehet építeni.



Példa: C1en opr szinkron módon meg lett hívva és majd alul visszatér. A dobozon belül van egy nagy doboz, cím az hogy ‘alt’. Ez azt jelenti hogy a dobozban alternatívák vannak, a doboz szaggatott vonallal régiókra vannak bontva. Különbözö esetekben különbözö régiók hajtódnak végre. Szögletes zárójelben van a feltétel (x>0), ide olyan feltétel írható ami interakciót nem igényel. Ha interakciót igényel akkor ki kell emelni és ki kell számolni. Ha x>0 akkor létrehozunk egy C2 objektumot, melyet úgy jelölünk hogy szaggatott vonal sima fejjel. A C2 megszünik ha minden végrehajtódik, ez jelölve van X jellel.

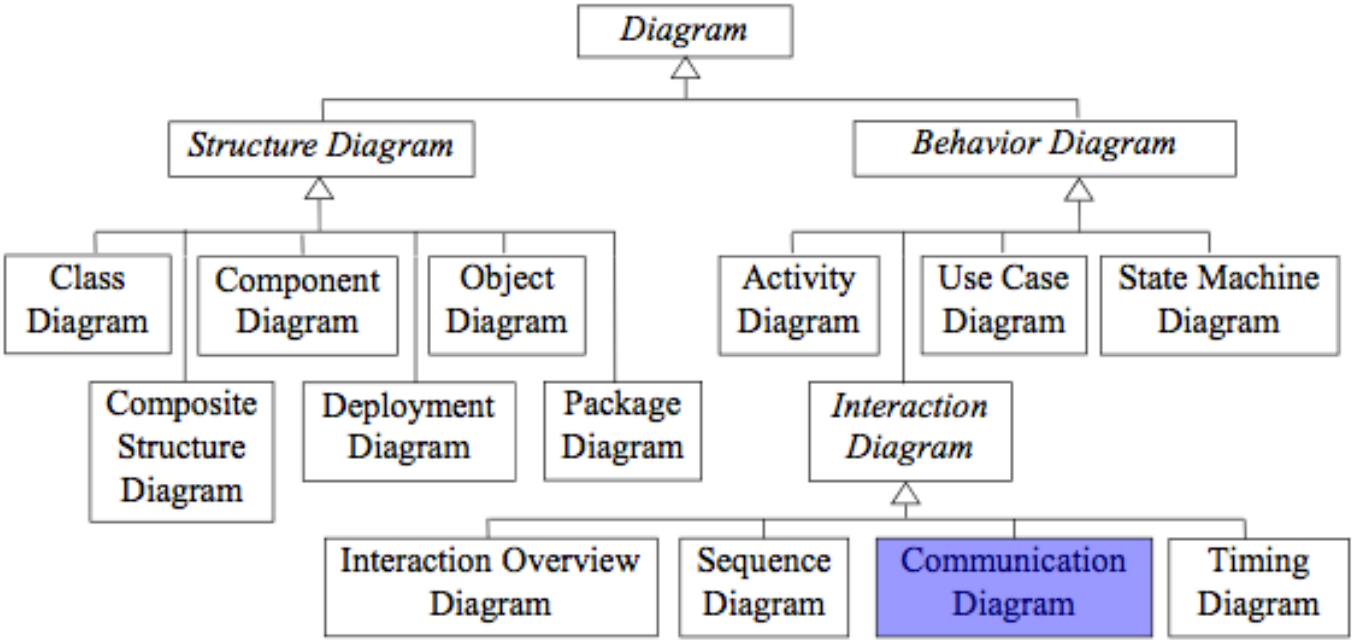
Doboz feliratok:
- **alt**: alternatíva, választás
- **opt**: olyan alt melyben csak egy régió van
- **break**: olyan szekvencia mely valami megszakadáskor van, pl.: exception
- **par**: régiók vannak, az egyes régiók párhuzamosan hajtódnak végre
- **neg**: olyan szekvencia mely nem megengedett
- **critical**: kritikus régió, kölcsönös kizárás megvalósítására, oszthatatlan
- **loop**: ismértlödik, melyben valami feltétel van



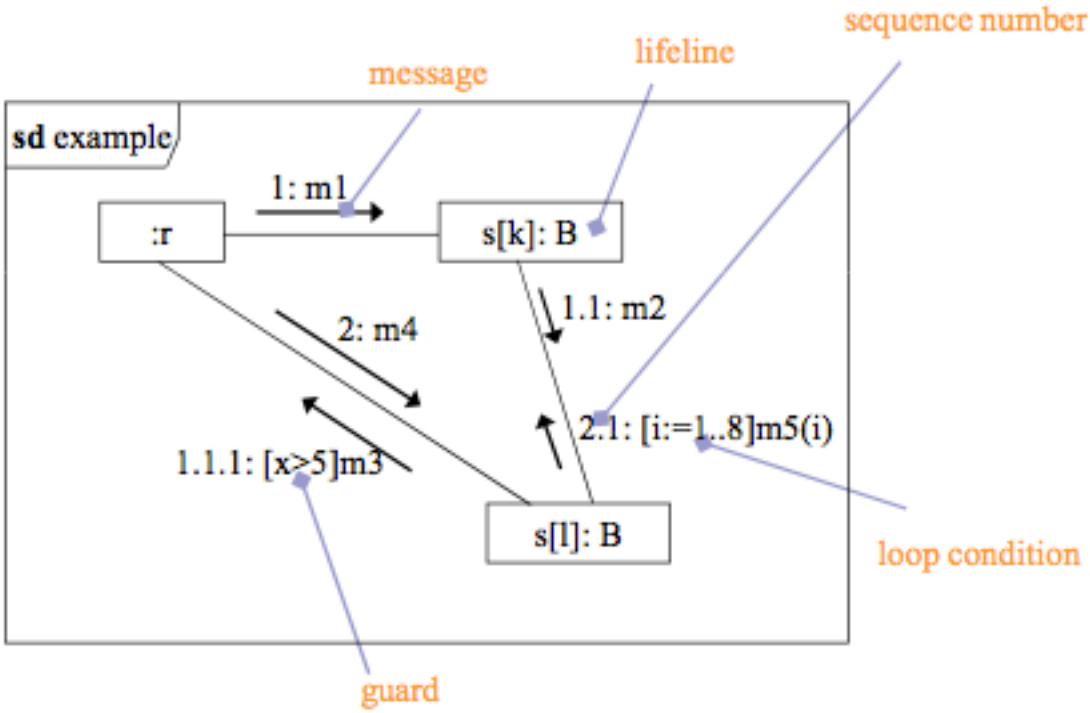
Példa: Van rajta egy idegen szekvencia diagram (ref). Ez egy interaction use, olyan szekvencia diagram melyben van két életvonal és valami történik benne. Utána van egy opcionális doboz, melyben mégegy beágyazott interaction use szekvencia diagram.

Feladatok Szekvencia diagram témakörböl (KATT IDE).

UML Diagramok - Kommunikációs Diagram



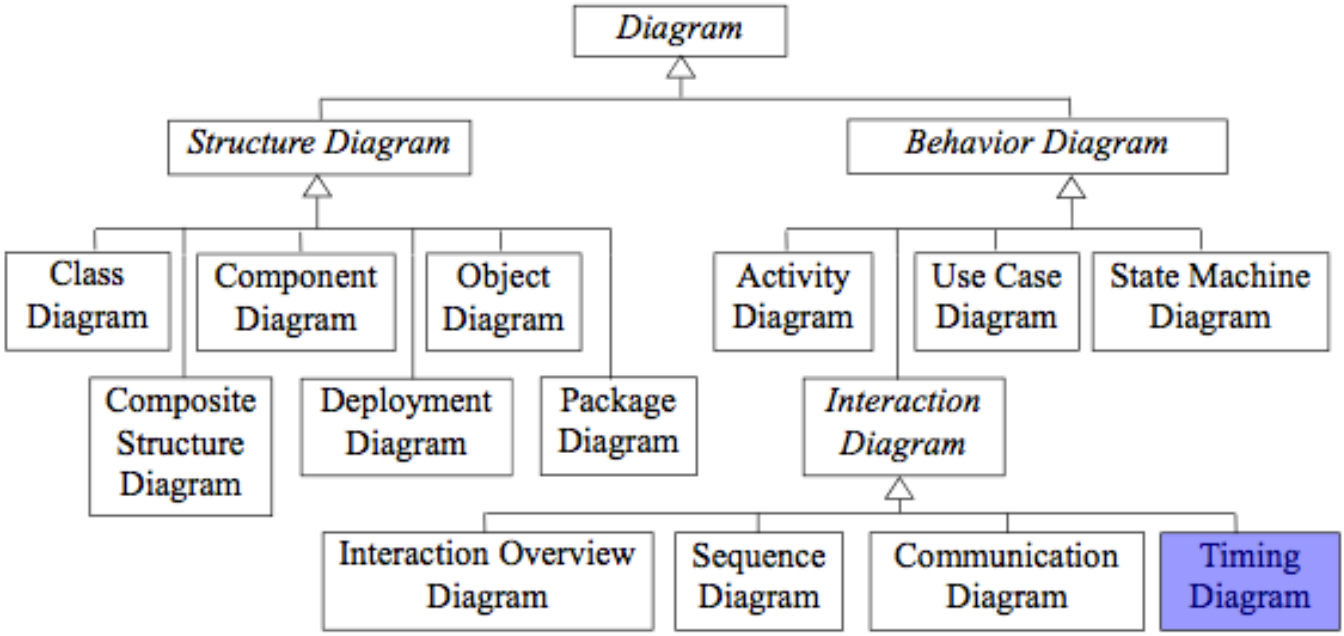
Ugyanazt írja le mint a szekvencia diagram, csak más nézöpontból.
Fogunk egy szekvencia diagramot és az életvonalak tetejét nézzük.



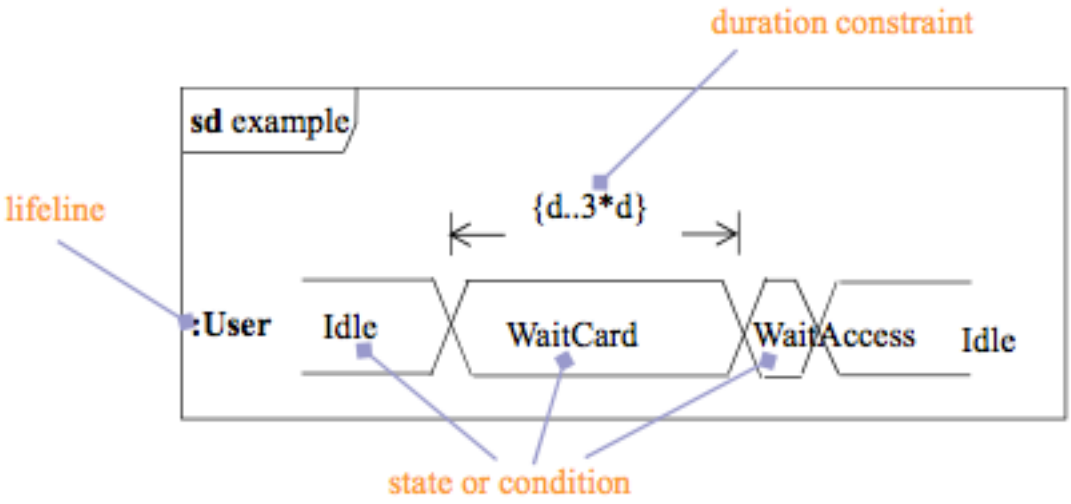
Példa: A három elemet felülröl nézem. Összekötöm azokat az objektumokat amik kapcsolatban vannak, az üzeneteket pedig jelölöm rajta és számozom öket. Az elsö üzenet (1) az m1, amit tovább küld az s[k] s[1]-nek aminek 1.1 a számozása. Az üzenetet tovább küldi s[1] r-nek, melynek már 1.1.1 a számozása.

Feladatok Kommunikációs Diagram témakörböl (KATT IDE)

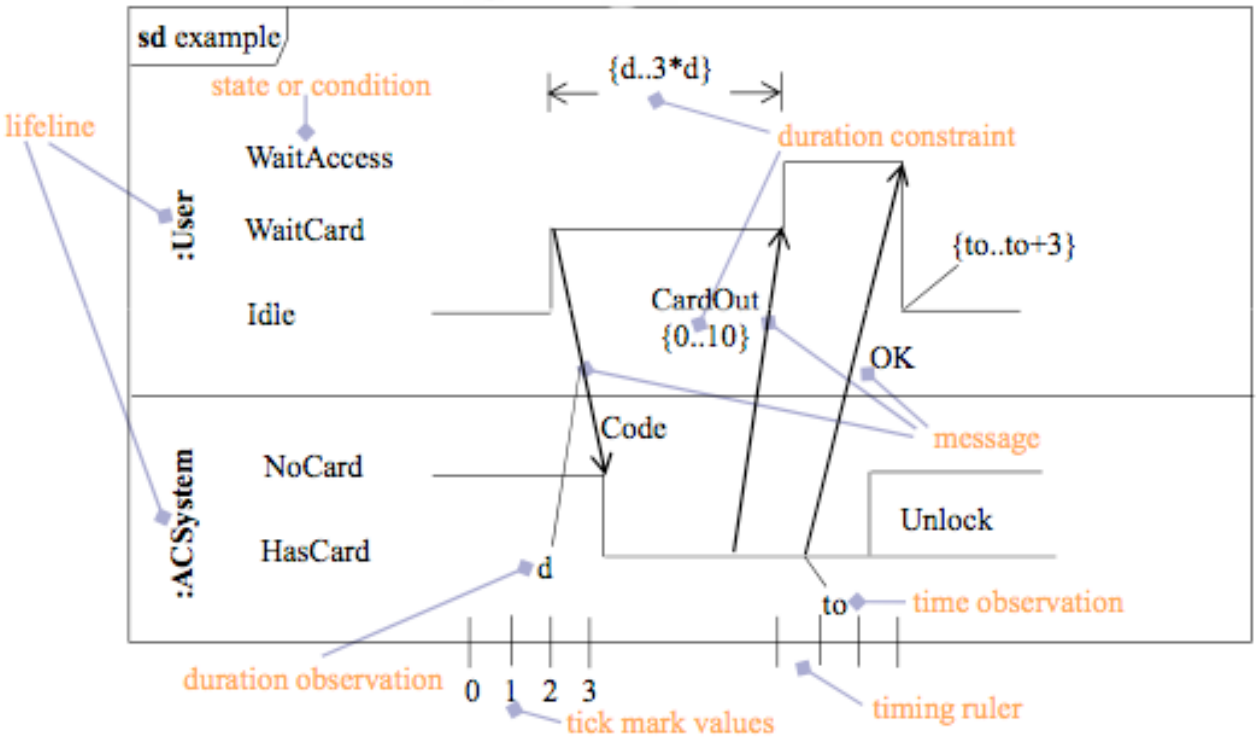
UML Diagramok - Timing Diagram



Interrakciókat ír le, idözítéseket tudunk kifejezni vele.



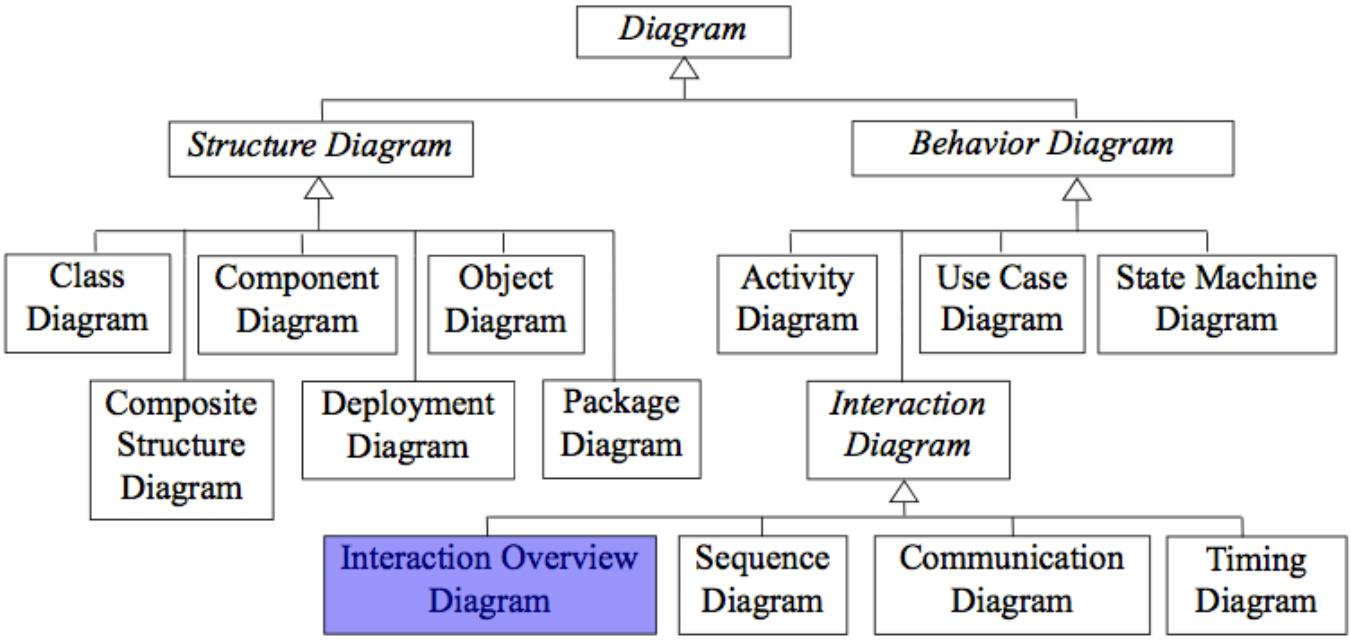
Példa: User objektumra vonatkozik. Idözítési elöírásokat tudok tenni különbözö idötartamok között.



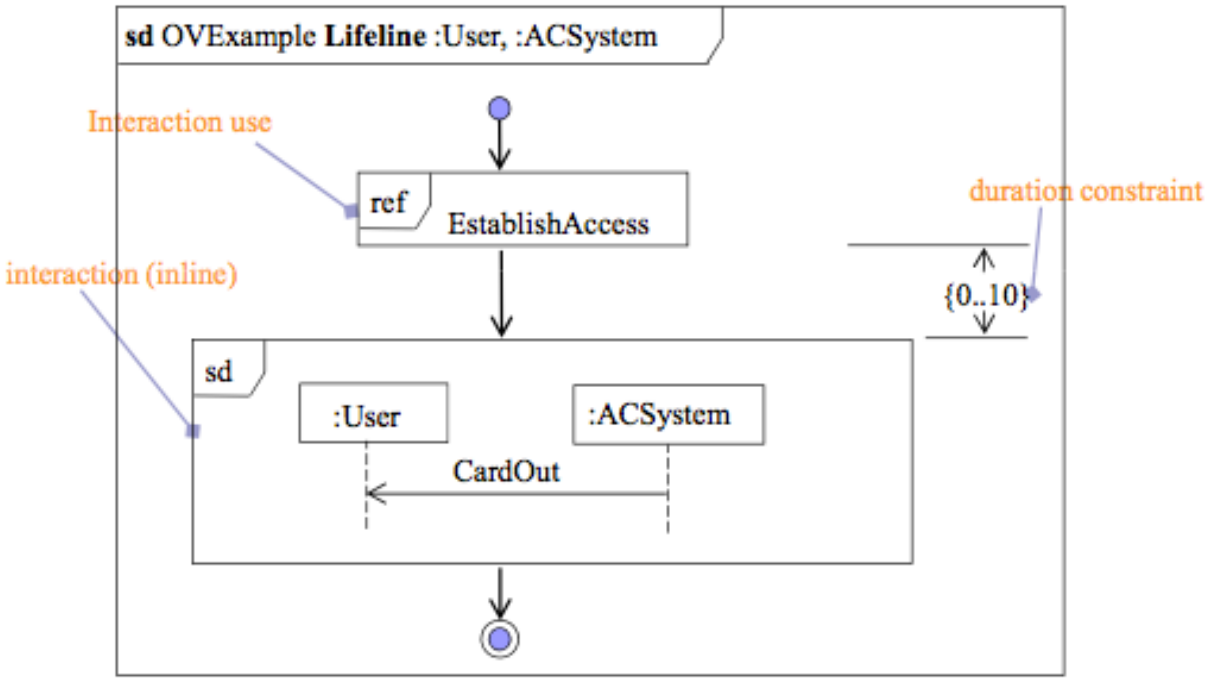
Példa: Két lifeline van és egyes állapotoknak egy-egy szint felel meg. Annyi szint van ahány állapot. Van indiferens állapot is (dont care), mindegy melyik állapotban van. A nyilak üzeneteket/eseményeket jelentenek. Időpillanatot tudok megjelölni, mint például a to és ezt felhasználom hogy to-tól to+3ig. Időpontra és időtartamra is tudok mintát venni a rajzból és azt felhasználhatom. Alulra vonalzót fel lehet venni.

Feladatok Timing Diagram témakörböl (KATT IDE)

UML Diagramok - Interaction Overview Diagram

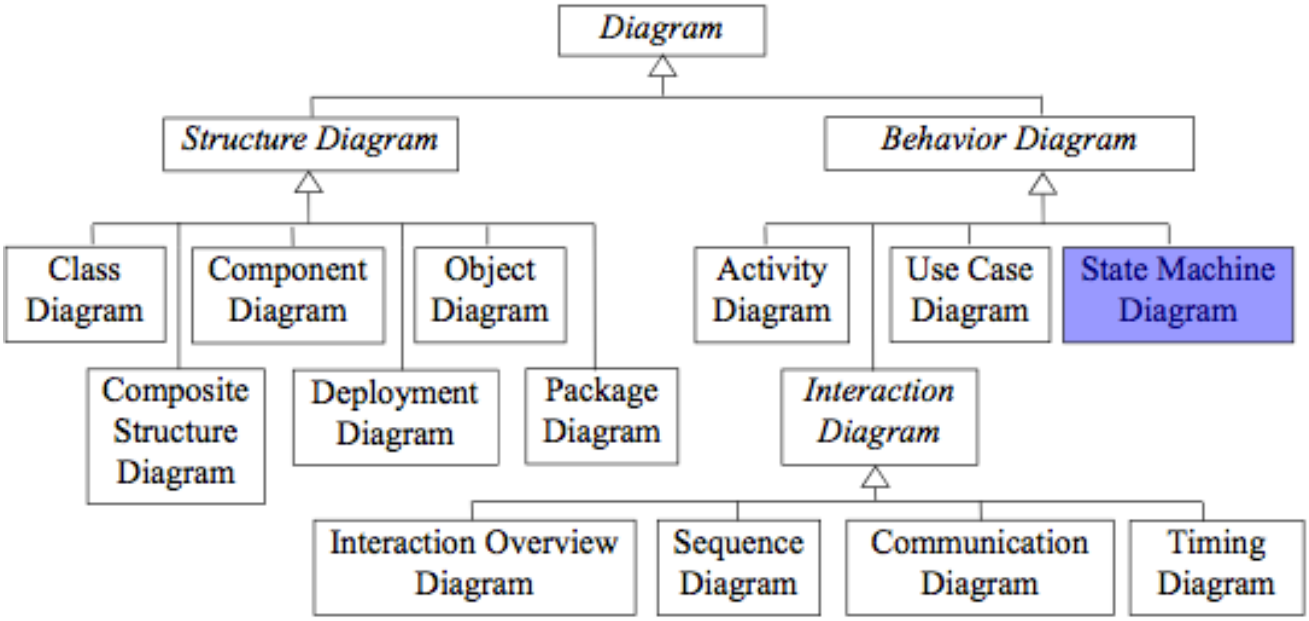


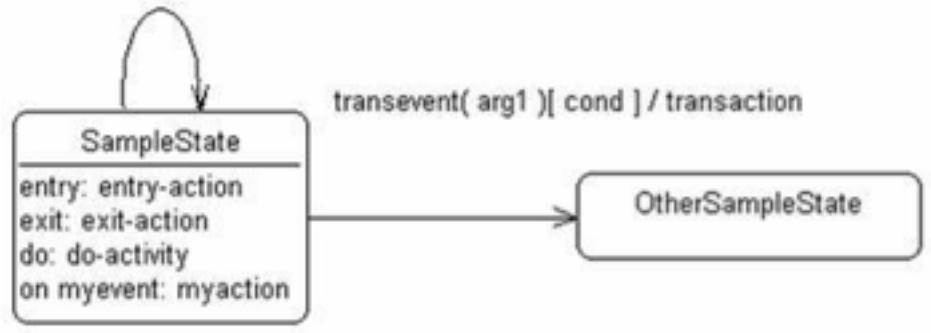
Interakciós áttekintö diagram:
- az Activity Diagram-hoz hasonlít, ami pedig a Flow Chart UML-es változata
- magasszintü kép, ahol egy-egy doboz lehet akármilyen interakció és idözítési elöírás is lehet köztük



Példa: Felül lépünk be, alul lépünk ki. Timing diagramhoz hasonlóan az idökorlátot is fel lehet venni.

UML Diagramok - State Machine Diagram

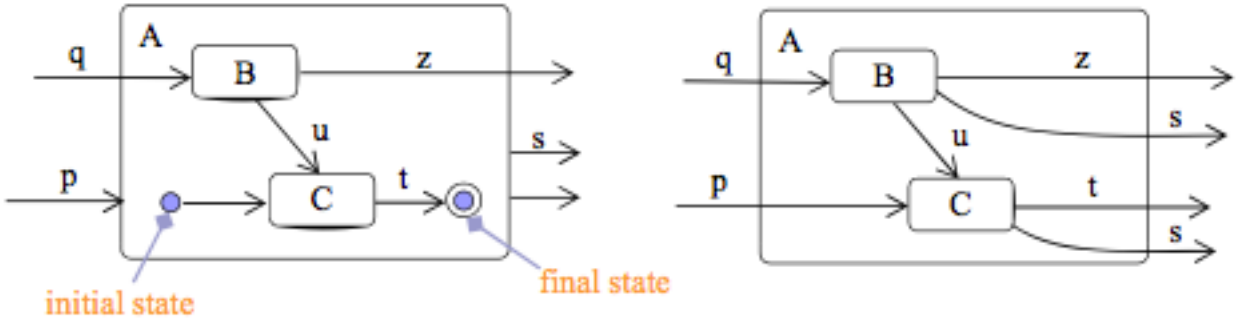




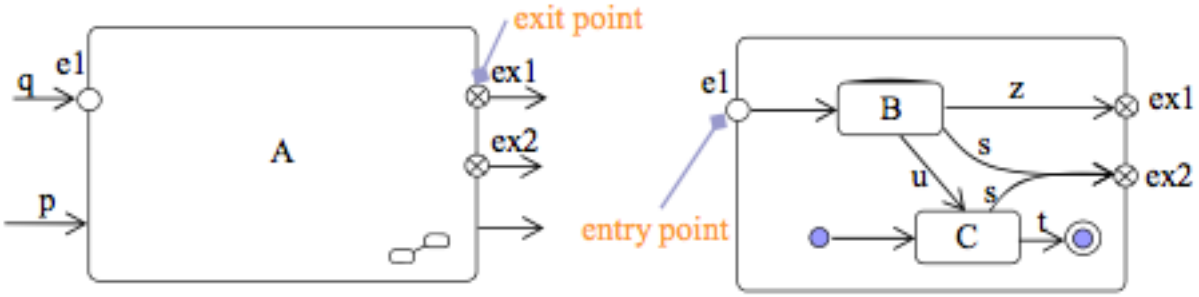
UML állapotgép model (State Machine):

- megegyezik a normál állapotgép fogalmával
- vannak UML specialitások
- állapot: lekerekített sarkú doboz
    - van olyan tevékenység amit nem azért hajtunk végre mert az eseményhez tartozik, hanem mert egy állapotot elhagyunk vagy egy állapotba lépünk
      pl.: ha belépek a boltba akkor boltban vagyok és köszönök, mindegy hogy honnan megyek a boltba, a boltba mindig köszönök. A köszönés nem az átmenethez tartozik hanem az állaptoba kerüléshez. -> ezek az entry-action és exit-action-ok
      Ha önmagába vezet az esemény akkor nem az entry/exit-actiont kell megtenni, hanem a myevent eseményre myaction-t hajtok végre --> ekkor el lehet hagyni a hurkot
    - do: arra való hogy ebben az állapotban valamit számol/tesz a program, megengedett az hogy ezt a tevékenységet mi hosszú távon tesszük. Ha valami megszakítás/következö esemény jön akkor mi történik? Ezt nem szabja meg az UML, a diagram rajzoló döntheti el
- állapotátmenet: rá van írva az esemény és az állapotátmenet közben végrehajtott tranzakció
- egy eseménynek lehetnek argumentumai, szögletes zárójelben meg tudom adni utána a guard-ot
- lehet olyan esemény mely valamit csinál, de ugyanabba az állapotba ér vissza amibe volt

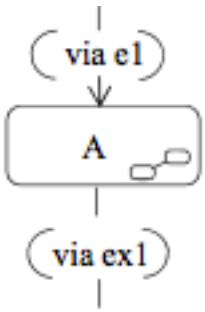
Kompozit állapotok (composite states):



Példa: Ha A-n belül vagyok akkor vagy B-ben vagy C-ben vagyok. Ha B-ben vagyok akkor A-ban is vagyok. Ha C-ben vagyok, akkor A-ban is vagyok. Az UML megengedi hogy átmeneteket is vezessünk a belsö állapotokhoz. A ‘q’ belemegy a B-be és ki is jön belöle. A ‘p’ belép az A-ba, és az initial state képzi le a külsö burkolatát a fedö állapotnak, azaz itt lesz a p. Ebben a példában a p-vel a C-be léptünk. Ha elhagyjuk a C-t és kilépünk a C-böl (final state), az leképzi a szélét. C-böl a t hatására elhagyjuk az állapotgépet. Felvehetünk egy s-t, s hatására bármelyik belsö állapotból kilépünk.

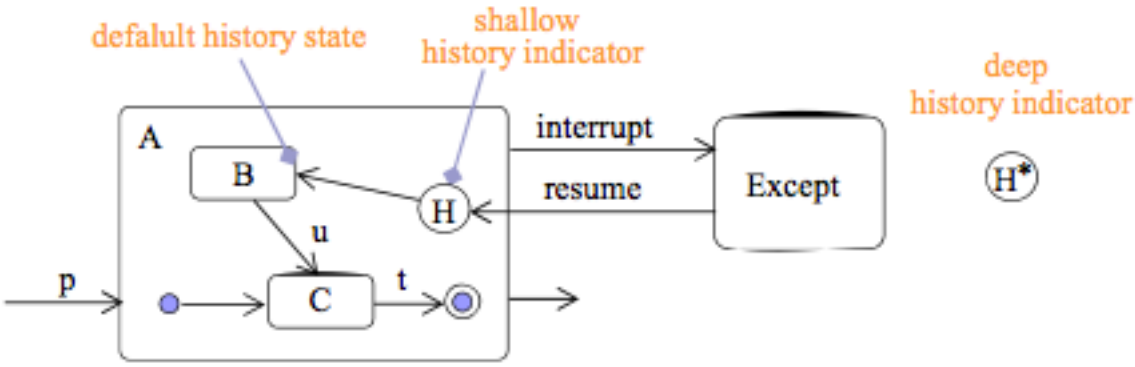


Példa: Van lehetöség entry és exit pontokat definiálni, csatlakozópontokat, ha nem a kezdö és végállapoton keresztül lepek be/ki. A q-val jövök e1-be, onnan B-be megyek. A z-vel az ex1-en lépek ki, az s-el bármelyik állapotból az ex2-ön lépek ki.



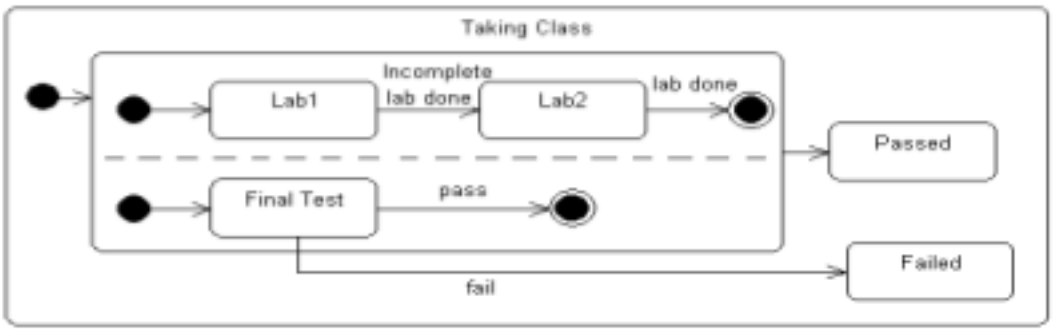
<-- Az elözö példára lehet ez egy alternatív jelölési mód.

History:

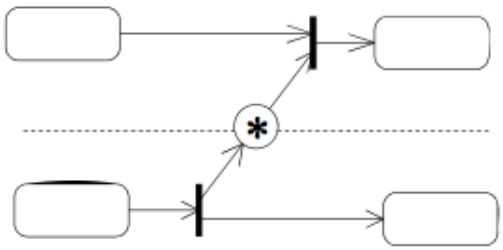


Példa: Van egy interrupt event, akár B-ben akár C-ben voltam interrupt hatására kilépek és Except állapotba lépek, ott megteszem azt amit ott kell es szeretnék visszalépni. De ha visszalépnél simán akkor a belépö ponton lépnék vissza, de én oda szeretnék visszalépni ahol a kivétel miatt kiléptem. Erre való a history, amelyik megjegyzi hogy honnan léptem ki. Ha mégsincs history de mégis kéne lenne akkor alkalmazunk egy elöre definiált history állapotot. Két fajtája van a historynak, **Shallow history** és **Deep history**. A B-n belül is lehetnek alállapotok amelyekre emlékezni kellene. A Deep history a teljes stacket tárol és B-n belüli alállapotokat is tárolja, amíg a Shallow csak a default értékü alállapotokat hozza létre.

Konkurens alállapotok:
- a konkurrens alállapotok egymással párhuzamosan futnak, melyek egymástól függetlenek, de mindhárom van valamilyen állapotban



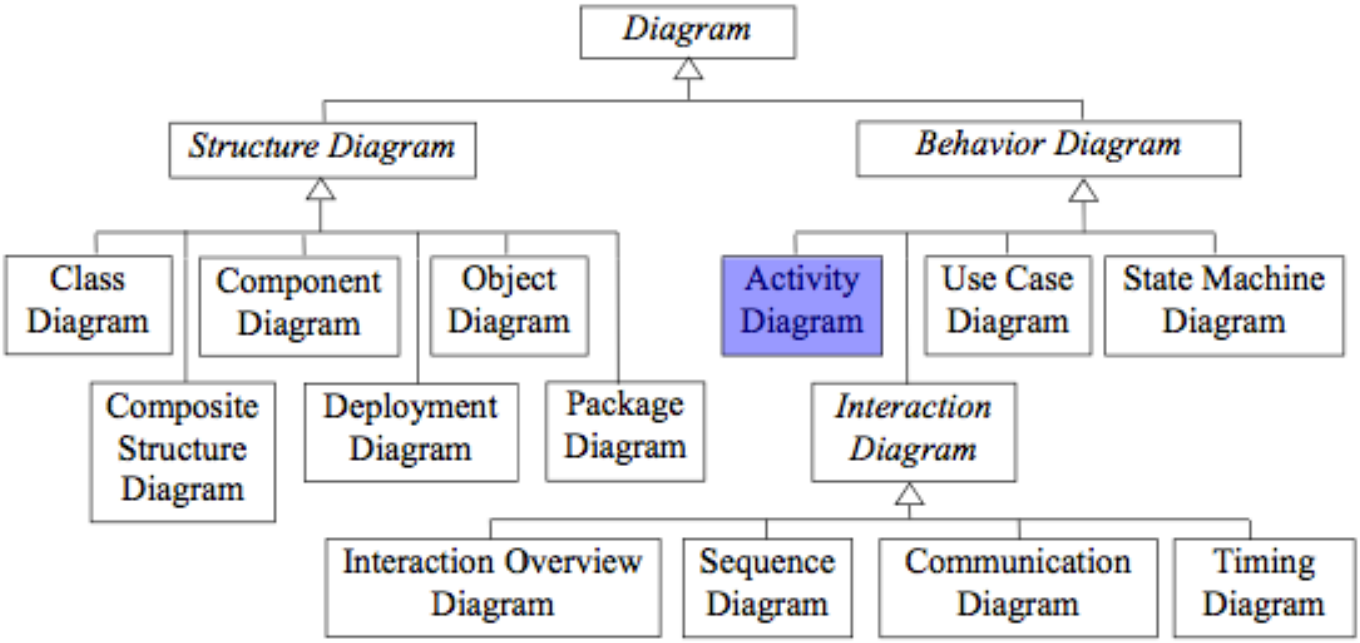
Példa: Két régió van. Van egy tárgy aminek a követelménye kettös (két laborfeladat + zh). A zh és a feladatok sorrendje tetszöleges lehet, ezzel szemben a laborok sorrendje fix (elsö labor majd második). Ha valaki megbukik a zh-n akkor megbukik, nem érdekes hogy van-e laborja vagy nincs. Kezdö állapot kell mindkét régióban.
A labor ágon ha beléptünk a lab1-be akkor azt jelenti hogy labor1 kötelezettségünk van, ha kiléptünk a lab1-böl akkor a labor1-et teljesítettük. Akkor tudunk kilépni a párhuzamos ‘bikaszemekböl’ ha mindkét ‘bikaszem’-en átmentünk.



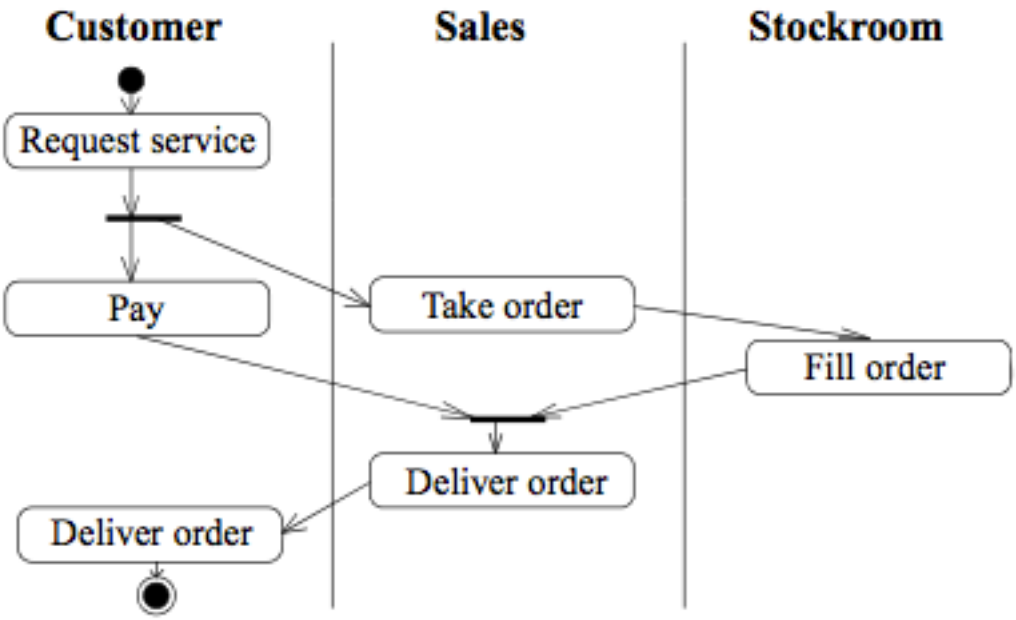
Szinkronizáció:
két régió van. Csak akkor mehetünk át egyik állapotból a másikba, ha a csillagon is átmentünk, ami bizonyos állapotokat kér hogy teljesüljenek.

Feladatok Állapot Diagram témakörböl

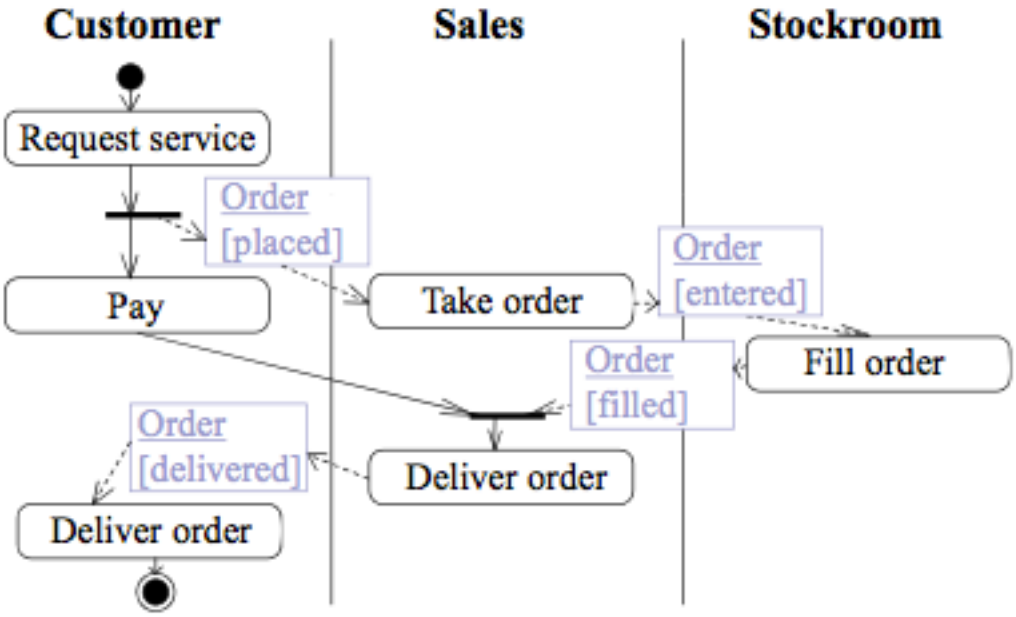
UML Diagramok - Activity Diagram



Activity diagram:
- megfelel a Flow Chart-nak ahol egy doboz=egy állapot, amiben egy tevékenységet végzünk, ez egy do activity. Ha végeztünk akkor automatikus állapotátmenet a következöre. Ez egy speciális állapotgép.
- lehet eljárásokat, alaktivitásokat bevezetni
- lehet döntéseket beletenni (klasszikus rombusz)
- megengedett hogy fork-oljunk és join-oljunk
- bevezetünk úszósávokat és object action flow-t



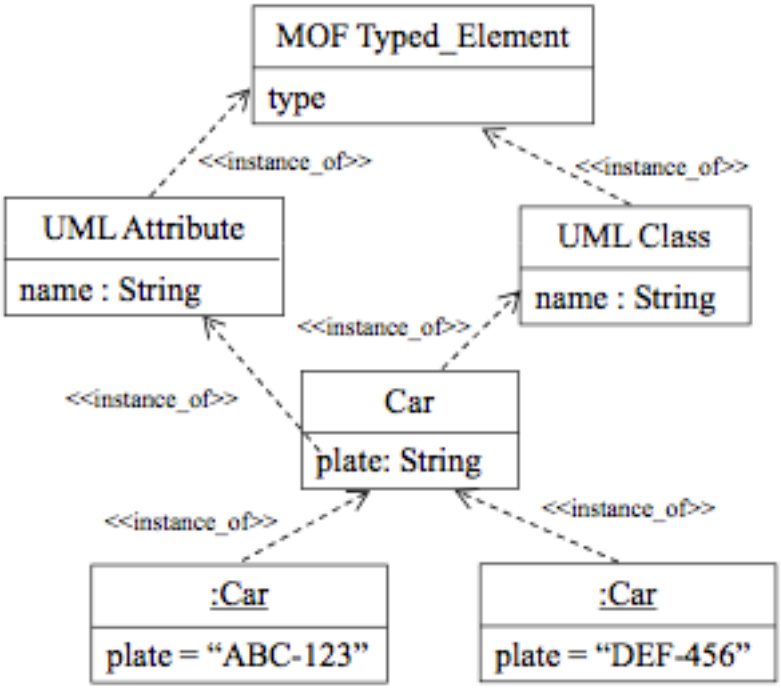
Példa: Tekinthetö folyamatábrának, mely szét lett húzva oldal irányba, mert külön-külön classifierekhez vannak rendelve.



Példa: az átadott objektumokat is tudjuk jelölni a diagramon.

Feladatok Activity Diagram témakörből (KATT IDE)

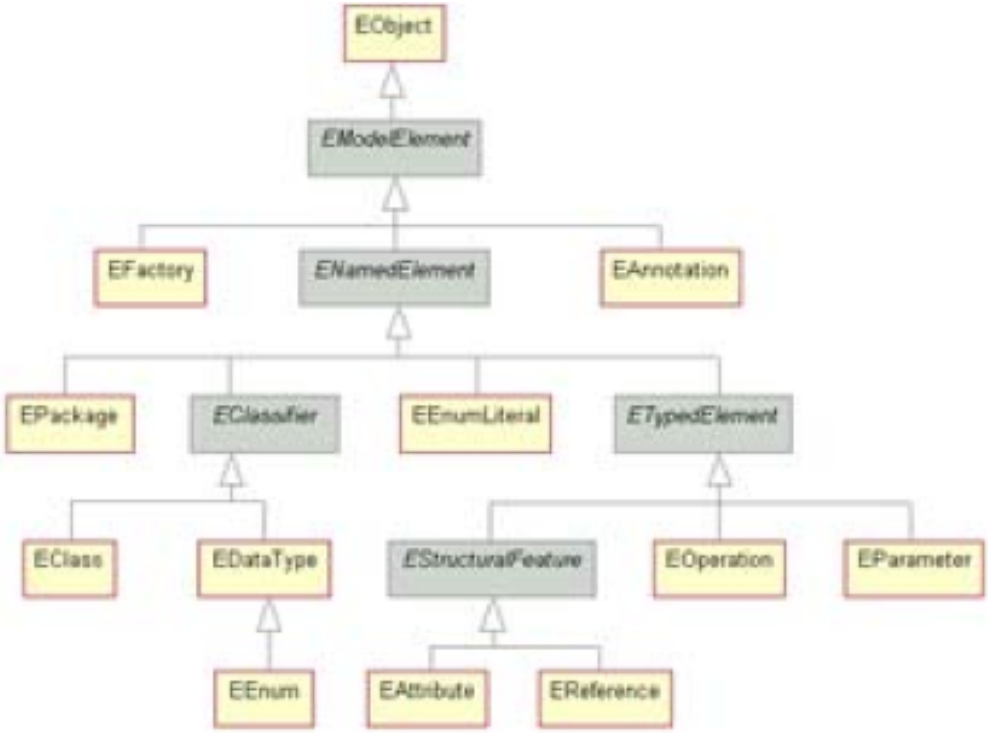
“élet az UML után”



Példa: A Car osztály az UML Class leszármaztatottja, a Car osztály plate attribútuma pedig az UML Attribute leszármaztatottja. Mindkét UML meta model leszármaztatottja a metameta model a MOF Typed_Element
M0 user object: itt :Car
M1 model: Car osztály
M2 meta model: klasszikus UML attribútum
M3 meta-meta model: attribútum az egy típusos eleme az UMLnek

MOF = Meta Object Facility
- OMG standard
- UML - általános célú modellezö eszköz
- MOF - implementálható nyelv metaobjektumok leírására

EMF = Eclipse Modeling Framework:

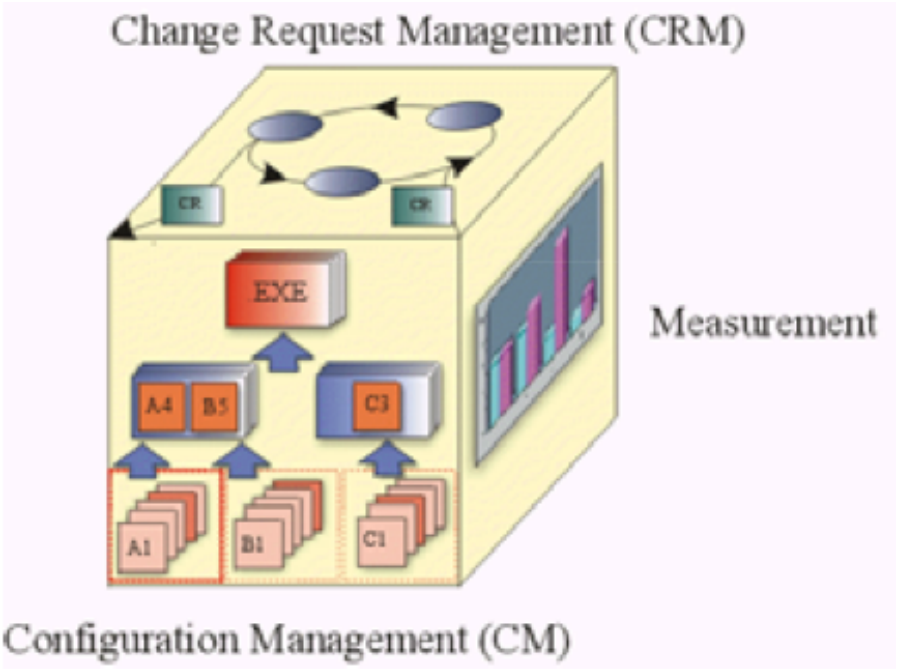


XMI = XML Metadata Interchange

- XML leíró amelyben a tageket arra találtunk ki, hogy vele UML modeleket tudjunk leírni
- az egyik eszközben elmentett modelt tudjuk máshol is használni
- ennek alapján könnyen lehet java kódot generálni
- ha én írom a compiler-t akkor a model elveszti a modelezö képességét, mert kifejez valamit és az tetszöleges szerkezeteket állít elö

Konfiguráció Menedzsment

Konfigurációs menedzsment definíciója:
- **egy komplex rendszer fejlödése követésének tudománya, a szoftver esetében számítógépes programok és dokumentációk együttesét jelenti**
- **a szoftver életciklus közben sok anyag keletkezik, ezeket az összefüggö anyagokat kézben kéne tartani**



Rajz elemzés: Néhány nézete a CM-nek.
1.: különbözö dokumentációk/programok/modulok vannak, melyekböl különbözö elemek állnak elö, majd ebböl egy exe sorozat
2.: Megjelennek változás igények (CR change request), különféle kérések valami életúton végigmennek és lecsapódnak a CM szerkezetben és ott változtatásokat okoznak
3.: Mérés: mérni kell (mi micsoda, miböl hány van, mihez mi tartozik, mi mennyire készült el, miben hány hiba volt)

Probléma:
- hogyan azonosítjuk az egyes dokumentumokat (pl.: hogy nevezzük öket)
- hogy irányítjuk az egészet
- milyen állapotok vannak, hogyan tartjuk számon, hogyan kezeljük

Azonosítás:
- mik leszenek a Configuration Item-ek (melyeket bevonunk a konfigurációs menedzsmentbe)
- meg kell mondani azt hogy hogyan kommunikálunk
- hogy néznek ki a standard emlékeztetök
- milyen dokumentációk vannak és ezeket hogyan készítjük el
- implementáció: milyen források vannak, hogyan kezeljük a különbözö változatokat, ...

Milyen dokumentumokat tekintünk Configuration Item-eknek RUP-ben:

System Requirements	Models	Use Case Model	Use Case Package
		User Interface Prototype	
	Database	Requirements Attributes	
	Documents	Vision	
		Glossary	
		Stakeholder Requests	
		Supplementary Specifications	
		Software Requirement Specs	
	Reports	Use-Case Model Survey	
		Use-Case Report	
System Design and Implementation	Models	Analysis Model	Use Case Realization

Konfiguráció Menedzsment alapvetö folyamatai:
- Storage Configuration Item
- Build Management
- Change Management
- Release Management

Státusz:
- Hogy nézzük meg azt hogy hogyan halad egy projekt?
- A projekt követésére milyen módszerek vannak

Storage Configuration Item-ek:
- verzió kontroll rendszerek: a source változatok között legyen valami áttekintés

Change menedzsment:
- hogyan kezeljük a változásokat
- Change Coltrol Board - változás kezelö bizottság
- defect management: mikor hogyan ki milyen prioritással kezeli a hibákat

Build menedzsment:
- milyen alapanyagokból hogyan rakjuk össze a programot
- vonatkozik a forrásokra, külsö eszközökre, bináris állományokra, stb.
- van egy alap szerkezet és ezeken változások vannak
- mindig az alapból hozzáadjuk a válzotásokat, ha csak a változást tároljuk akkor könnyü visszalépni
- **baseline**: alapként, inkrementum: változás
    - baseline + inkrementum a jelenlegi állása a programunknak
    - el kell dönteni hogy mikor kell új baseline-t csinálni
    - baseline elötti múlt már nincs

Release menedzsment:
- verzió: funkcionálisan különböző
- variáns: funkcionálisan egyezik, nem funkcionálisan különböző
- release: verzió amit kiad a team
- revision: verzió + variáns (nem szükségesen release)

- a release-hez tartoznak egyéb plusz dolgok (pl.: release notes)
- verzió név probléma
    - addig nincs gond amíg nem ágazik szét a számozás

Tools:
- vezió kontrol
    - cvs, subversion, ClearCase
- change kontrol
    - Bugzilla, ClearQuest
- build menedzsment
    - Apache ANT, CruiseControl
- kommunikáció
    - email, calendar, wiki

Verzió kontrol (version control):
- betesz/kivesz elven müködik
    - check-out: kezelt elem kiemelése felhasználásra a közös tábból
    - check-in: kezelt elem visszahelyezése a közös tárba
- ha konkurrens módosítástok vannak akkor van gond (párhuzamos hozzáférés)
    - reserved checkout
        - amíg valaki kiveszi addig más nem nyúlhat hozzá
    - modify-update-merge
        - mindenki kivehet és ha mindenki visszateszi akkor próbáljuk összerakni az egyes verziókat

Konfigurációs menedszment terv:
- labor4-en érdemes valami eszközt használni ehhez

Verifikáció és Validáció

Verifikáció:
- Jól készítettük-e el a terméket?
- elöírásnak megfelel-e a program/dokumentum/termék

Validáció:
- Jó terméket készítettünk-e el?
- ez-e az a produktum amire a felhasználó vágyik

A verifikákás és a validálás nem egyszeri cselekedet, hanem folyamatosan kell figyelni minkettöt. Egy szoftver fejlesztési folyamatban folyamatosan kell visszacsatolás, ez a két eszköz alkalmas erre.

Szoftver áttekintés (software review):
- veszünk valamilyen anyagot, melyet vizsgálat alá veszünk
- könnyen lehet alkalmazni
- a fejlesztési folyamat bármely fázisában elöfordulhat
- az adott anyagnak megfelelö specifikációságot kell ellenörizni
- a nem funkcionális dolgokat nehéz megvizsgálni
- csak azt lehet nézni hogy bizonyos elöírásoknak megfelel-e az anyag

Szoftver teszt (software testing):
- a programot futtatjuk, müködés közben vizsgáljuk
- azt csinálja-e amit elvárunk?

Review

Software review definíciók:
Review: általános kifejezés a felülvizsgálatra/áttekintésre.
- valamilyen anyagot átnézünk, de végrehajtás/dinamika nincs benne

Review item: a vizsgálat anyaga
Audit: általában az adott cégtöl független emberek végzni. Vannak jól definiált szakmai elöírasok, és ezeknek való megfelelést vizsgálják.
Walkthrough: általában valamilyen anyagnak a készítöje bemutatót készít az anyagáról amit készített, többen ezt vitatják
Inspection (bevizsgálás): elöre kiosztanak anyagokat a reviewer-eknek (ellenöröknek) kiadják, ök jelentést tesznek az anyagokról egy bizottságban megvitatják, kifogásokat tehetnek, ahol a szerzö is ott van (egyetemi diploma védés is hasonló)

Before the meeting:
- meghatározott idöszakonként van review, ez egy cégnél elöre van ütemezve
- meg kell mondani hogy mik lesznek az ellenörzött anyagok
- szereplök:
    - koordinátor: lehet az anyag szerzöje vagy csoportvezetö, ö felelös a meetingért, ö felelös a következményekért
    - lead reader: az adott anyag felépítésével/müködésével tisztában van
    - jegyzökönyvvezetö: korrektül dokumentálja a meeting-et
    - megbízó/felhasználó képviselöi

During the meeting:
- 15 perc és 2 óra közötti idötartamú
- a hiba megtalálása a cél, a személy értékelésére nem szabad ezt felhasználni
- a kulcskérdés a hibát megtalálni, nem kijavítani
- **minden egyes hibát amit találnak:**
    - **ki kell jelölni egy felelös személyt, akin ez a hiba számonkérhetö**
    - **meg kell mondani hogy mit kell csinálni pontosan, mi a teendö**
    - **megmondjuk azt hogy ez a probléma típusás és a súlyát**
- a következöket kell meghatározni
    - vagy az anyag el van fogadva
    - vagy az anyag feltételesen van elfogadva, amiben kisebb hibák vannak és a felelösöknek ezt meg kell oldani, ha azok meg vannak oldva akkor nem kell újabb meeting
    - vagy major action, ha a problémák meg vannak oldva akkor újabb review-t kell tartani

After the meeting:
- jegyzökönyvvezetönek ki kell osztnai a jegyzökönyvet
- a felelös személynek dolgozni kell a problémáján
- komoly probléma estén újabb meeting kell

Jegyzökönvy template:
- mikor volt
- csoport név
- termék név
- milyen dolgokat vizsgáltak
- kik vettek részt, milyen szerepben
- mi volt ennek a célja
- akciók: felelös, mit kell tenni, a probléma nehézsége (minden problémára)

Felülvizsgálók szabályai:
- illik rá felkészülni
- a szerepnek megfelelöen kell viselkedni
- legyen együttmüködö a felülvizsgáló (nem a személyröl szól, hanem a hibáról)
- ell kell fogadni a bizottság/vezetök parancsait, nem lehet szembeszegülni (még ha nekünk is van igazunk)

Szoftver tesztelés

Szoftver teszt:
- konformancia vezérelt tesztelés: a szoftvernek szabályoknak kell megfelelnie, ezt a megfelelöséget kell igazolni
- hibadetektálás: a program futtatása hibakeresés céljából

Teszttel kapcsolatos hiba fogalmak:
- *error*: emberi tevékenység (tévedés, elhanyagolás) hiánya, ezért keletkezik a program kódjában valami hiba (bug)
- *bug*: kódbeli hiba, ami el van rontva (debug = ezeket a hibákat akarjuk megtalálni), valami ki van hagyva a kódból. A bug failure-t okoz.
- *failure*: kívülröl látható hiba (pl.: lefagyás)

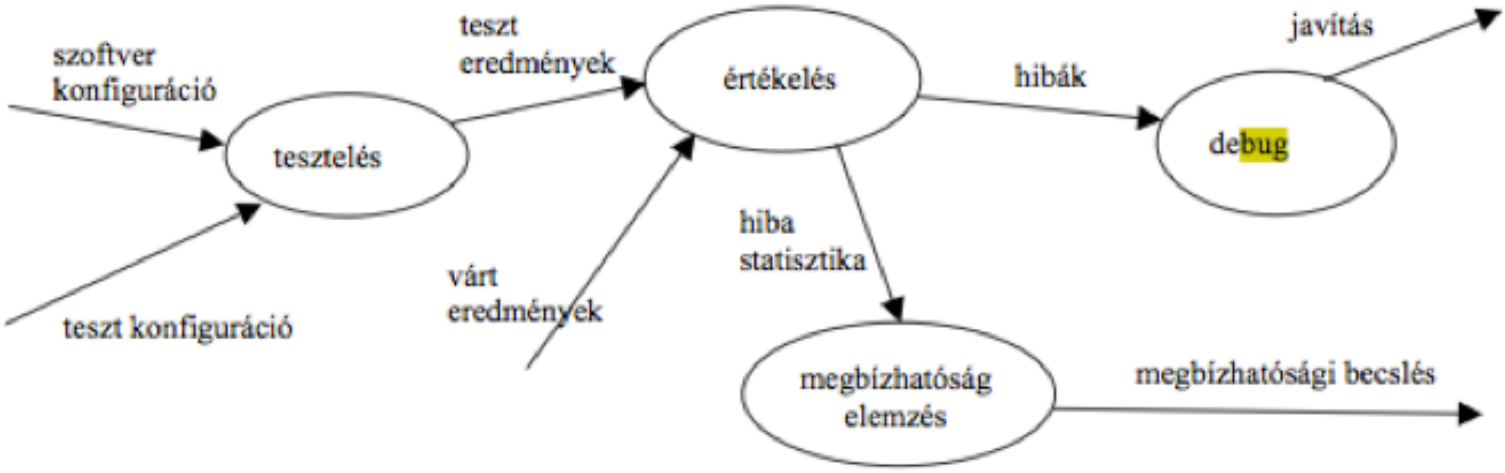
A failure részével szembesülünk. Ehhez kell megtalálni a bug-ot, és ebböl ki kell deríteni az error-t.

Teszt célja:
- objektumok közötti interakciók tesztje
- jól vannak-e összerakva az egyes elemek
- kielégítjük-e a követelményeket
- szoftver telepítés elött ellenörizzük hogy van-e benne hiba

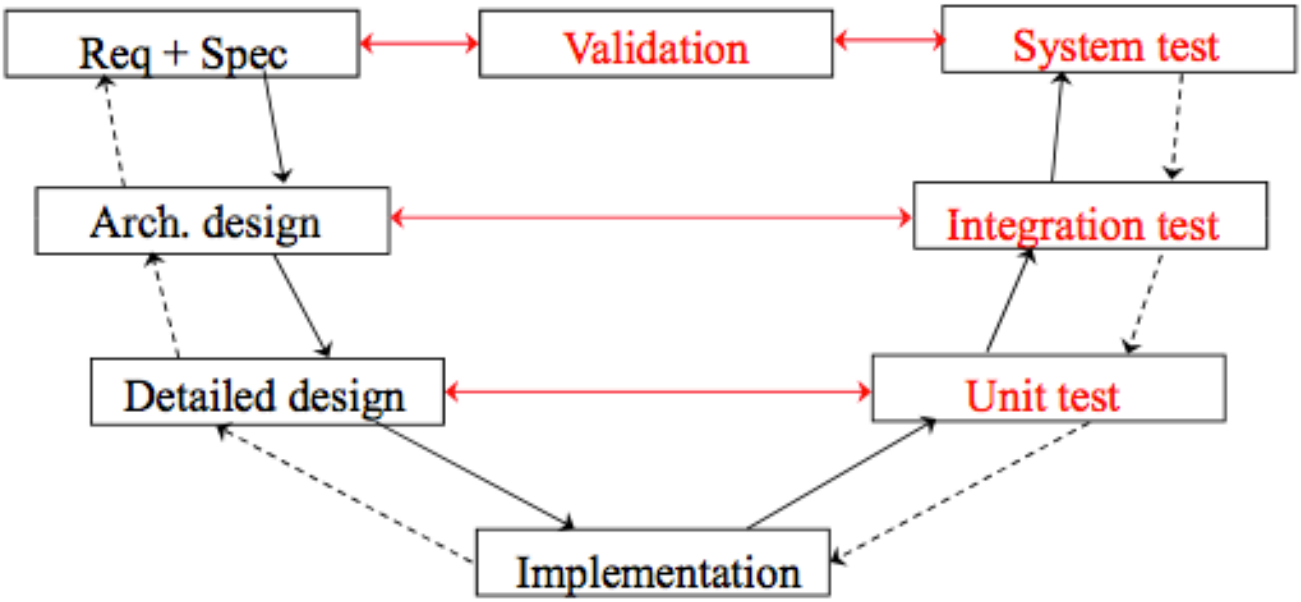
Miért speciális a szoftver teszt?
- a szoftver nem folytonos
- finomításos módon nem lehet a hibát javítani (“kicsit itt javítok rajta és kicsit jobb lesz a szoftver” -> ilyen nincs!)
- “eddig eljutottam, majd továbbjutok és meglesz a hiba” -> ilyen nincs! (interpolálás)
- nem nagyon van olyan program ami megírás után egyböl fut

Teszt:
- azt kimutatni hogy egy programban van hiba, az kimutatható
- nem lehet biztosítani a hiba hiányát
- a fejlesztés költségeinek 30-50%-a tesztelésre megy el
- bonyolult dolog a tesztelés
- milyen skálán lehet mérni a jó és rossz programozót? : a jó programozó 3* annyi programot/kódot ír ugyanannyi hibával
- a tesztelésnél ugyanez a szám 10*-es, bonyolult gyorsan eredményt hozni
- nagyon emberigényes a tesztelés
- az érdekel hogy a szoftverben mennyi hiba lehet még?
    - tervszerüen hibákat tesznek a kódba, ebben a tesztelök hibákat keresnek, ha megtaláltak pár hibát akkor a tervszerüen beletett hiba arányából következtetni lehet a hibák számára

A tesztelés információs folyamatai:



Van egy szoftver konfiguráció és egy teszt konfiguráció. Ezt futtatjuk és kijönnek az eredmények. A kapott teszt eredményeket összevetjük a várt eredményekkel. Meg kell tudni mondani, hogy milyen eredményeket várunk. A kiértékelésböl debug felé megy ha hiba van, illetve megbízhatósági elemzés megy, hogy ki tudjuk értékelni a fejlesztést, ebböl pedig kiértékelések lesznek.



Klasszikus V model:
Ahogy szétszedjük és eljutunk az implementációig, ugyanúgy fordított irányba össze is kell rakni a dolgokat.
- a teszt és az integráció összetartozik
- az egység teszt (unit test) a részletes tervben megfogalmazottakkal tart egyensúllyt
- a részletes terv az architektúrális tervvel
- a renszer terv a specifikációval
- a validáció a követelményekkel

Egység teszt (unit test):
- olyan elem melyet egy programozó elkészít
- aki megírta az le is teszteli
- elemekröl szól (egy vagy egy-két osztály OO esetén)

Integrációs teszt (integration test)
- független tesztelök végzik
- összerakunk elemei teszten átment egységekböl egy összetettem alrendszert vagy rendszereket
- a fö cél annak a vizsgálata hogy az egyes egységek jól müködnek-e
- az interakciók, a kommunikáció vizsgálata folyik
- **két stratégia van erre:**
    - **top-down**: föntröl rakunk össze egy programot. (pl.: grafikus felület esetén elöször a menüt teszünk össze majd egyes funkciókat teszünk bele, ami még nincs benne oda “not yet implemented” ablakot teszünk - test betét (test stub))
    - **bottom-up**: alulról készülünk el elemekkel, alacsony szintü funkciók/metódusokat rakunk össze, majd építkezünk fel. az a problémája hogy hiányzik a keret amiböl az egészet tudjuk használni, ezért kénytelenek vagyunk egy teszt keretet (**test bed**) készíteni amibe az eddig elkészült alulról elkészített elemeket tudjuk beletenni

Rendszer teszt (system test):
- fokuszban a rendszer, hogy helyesen müködik-e
- megvizsgáljuk hogy a rendszer képességeit, jellemzöit tudjuk-e mutatni

Elfogadási teszt (acceptance test):
- kész a program, mielött átadjuk a felhasználónak, azelött tesztelünk
- teljesen formálisan le van irva hogy milyen teszteket kell végrehajtani
- informális tesztek
    - alfa teszt
        - házon belül ad-hoc vizsgálat, nincs elöírás
    - beta teszt
        - a fejlesztök kiadják valamilyen körben (internetes beta változat), automatikus vagy manuális visszajelzés alapján keresi a hibákat, felelösség nélkül adják ki

Tesztek típusai (FURPS):
- **funkcionalitás (Functionality)**
- **használhatóság (Usability)**
- **megbízhatóság (Reliability)**
- **teljesítmény (Performance)**
- **támogatottság (Supportability)**

Funkcionalitás teszt:
- rendszer funkciói
- security: nem csak az azonosítás, hanem van-e feljogosítása hogy valamilyen adatot elérhet-e
- mennyiségi teszt: nagy mennyiségü adatmennyiséggyel való teszt, új hibát nem nagyon hozhat létre

Használgatóság teszt:
- emberi tényezök
- user dokumentációk, helpek
- nehéz emberi szempontból leteszteltelni pl. a felhasználói felületet

Megbízhatóság teszt:
- integritási teszt: robosztusság, kibírja-e az idétlenségeket
- struktúra teszt: ismerem a program belsejét és abból következtetve próbálok szélsöséges eseteket tesztelni
- stressz test: szélsöséges értékek kipróbálása, tesztelése

Teljesítmény teszt:
- benchmark teszt: összehasonlításhoz
- contention teszt: stressz teszt változata, több actor versenyez ugyanazokért az eröforrásokért
- load teszt: müködési limit környérén müködés
- teljesítmény profil: olyan teljesítmény profil amelyet az éles müködés során gyüjtünk, gyakran futtatott metódust így például gyorsítani lehet jobb algoritmussal

Támogatottság teszt:
- hogyan konfigurálható
- hogyan installálható

Teszt metrikák (mértékek):
- meg kell mondani hogy mikor hagyjuk abba a tesztelést
- követemény-alapozott mérték
    - ha van egy RfT (total number of Requirements for Test) számunk, amiböl tudjuk hogy a követelmények teszteléséhez hány tesztesetet kell lefuttatni helyesen
    - ha tudjuk T-vel hogy hány tesztesetet futtatunk le sikeresen
    - T / RfT értékkel tudjuk hogy hány százalékos a tesztünk
- kód-alapozott mérték
    - megmondjuk hogy milyen kódelemeket kell mérnünk = Tlic (total number of items in the code)
    - I = tesztelt kódelem
    - T / Tlic értékkel tudjuk hogy hány százalékos a tesztünk
- hiba analízis
    - hibákkal kapcsolatos bizonyos paramétereket érdemes rögzíteni
    - milyen prioritásu a hiba
    - mi okozta a hibát
    - ebböl különféle riportokat tudunk készíteni (eloszlás/sürüség/stb)

Teljesítmény riportok:
- hol, melyik rész fut
- a program 90%-a az idö 10%-ában fut, a program 10%-a az idö 90%-ában fut

Teszt stratégiák:
- megmondja az általános megközelítéseket
- megmondja hogy milyen technikákat/eszközöket használunk
- megmondja hogy mi a teszt sikeressége
- hogyan teszteljük a számunkra külsö dolgokat (milyen szimulátor programok, milyen egyéb teszt segédprogramok)

Teszt eljárások:
- teszt eset csoportokat foglal össze
- milyen teszt esetek vannak
- milyen input, milyen elöfeltevés
- egy teszt eset csoportra van ez definiálva

Teszt esetek:
- egy tesztelendö elem ellenörzése
- ezt al-teszt esetekre tudjuk bontani
- hogyan készítünk teszt esetet (unit test esetén) ?
    - **white-box test**
        - amit tesztelünk azt tudjuk hogy mi, tudjuk mi van benne, ismerjük a program szerkezetét
        - a program szerkezetéböl kiindulva tesztelek (pl minden utasítássor legyen végrehajtva)
        - minden döntési ágon végigmegyünk értelmesen, minden hurkot is végigpróbálunk
        - kimerítö teszt: összes lehetséges értelmes/értelmetlen adattal kipróbálni (nem igazán van rá esély)
    - **black-box test**
        - ekvivalencia osztályozás: tudunk azonosítani bizonyos viselkedéseket
        - boundary value analízis: ekvivalencia osztály határok tesztelése
        - speciális érték tesztelés

Objektum-oriantált program tesztelése:
- lazán csatolt objektumok, megpróbálni interface szinten kezelni
- beállítani az attriútumokat
- megnézni mennyire zártak
- öröklést kipróbálni
- polimorfizmus tesztelése
- állapotfüggés tesztelése

JUnit:
- ingyenes tesztelö eszköz java programokhoz
- elvárt eredmény/produkált eredmény összehasonlítása

```
public class MyInt {
    private int value;
    public MyInt(int aValue) {
        value = aValue;}
    public void add(MyInt anInt) {
        value += anInt.getValue();}
    public int getValue() {
        return value;}
}
```

Teszteléshez készített objektum.

```
public class MyTest {
   public static void main(String[] args) {
       MyInt m1 = new MyInt(5);
       MyInt m2 = new MyInt(30);
       m1.add(m2);
       if (m1.getValue() != 35)
         System.out.println("Sum failed");
       if (m2.getValue() != 30)
         System.out.println("m2 failed");}}
```

Egyszerü tesztelés, objektum létrehozása. JUnit nélküli megoldás.

```
public class MyIntTest1 {
    MyInt m1, m2;
    @Before
    public void setUp() {
        m1 = new MyInt(5);
        m2 = new MyInt(30); }
    @Test
    public void testAddInt() {
      m1.add(m2);
      assertEquals("sum Test", 35, m1.getValue());
      assertEquals("m2 Test", 30, m2.getValue());}
}
```

JUnit teszt eset:
-@Before inicializál egy kezdö állapotot
-@Test maga a végrehajtandó teszt
- assertEquals függvénnyel teszteljük az elvárt és megkapott eredményt
- eclipse-ben való futtatása
    - Java Build Path/Libraries/Add Library/Junit4
    - Rus As/Junit Test
- minden teszt egy metódussal van implementálva (nincs paraméter/visszatérési érték)
- a teszt metódusnak public-nak kell lennie, @Test-tel jelölve
- egyéb fixture:

```
@BeforeClass - oneTimeSetUp
@Before - setUp
@Test - test
@After - tearDown
@AfterClass - oneTimeTearDown
```

Assertion-ok:

```
static void assertTrue([String msg,] boolean condition)
static void assertFalse([String msg,] boolean condition)

static void assertNull([String msg,] Object object)
static void assertNotNull([String msg,] Object object)

static void assertSame([String msg,] Object unexp, Object act)
static void assertNotSame([String msg,] Object unexp, Object act)

static void assertEquals([String msg,] X exp, X act)
static void assertArrayEquals([String msg,] X exp, X act)

static void fail([String msg])
```

Teszt eredmények:
- lefut
- failed: szándékos hiba amit mi tesztelünk
- error: olyan hiba amire globálisan nem számítunk (pl NullPointerException, de mi nem ezt teszteltük)

Exception teszt:

```
public class TestException {

        private Collection<Object> collection =
                                new ArrayList<Object>();

        @Test(expected=IndexOutOfBoundsException.class)
        public void testIndexOutOfBoundsException() {
                Object o = ((ArrayList<Object>)collection).get(0);
        }
}
```

Timeout teszt:

```
@Test(timeout=100) public void infinity() {
        while(true);
    }
```

Parametrizált teszt:

```
@RunWith(value = Parameterized.class)
public class ParamTest {
        private int a;
        private int b;
        public ParamTest(int a, int b) {
                this.a = a;
                this.b = b;
        }
        @Parameters
        public static Collection<Object[]> data() {
                return Arrays.asList(new Object[][]
                        { {1, 5}, {4, 9}, {2, 7}, {2, 3} });
        }
        @Test
        public void runTest() {
                Assert.assertTrue(a < b);
        }
}
```

Feladatok Verifikáció és Validáció témakörből (KATT IDE)

Dokumentációhoz készített tervek:

- Requirement
  - System definition, Project plan, PFR
- Specification
  - Requirement specification, Preliminary user's manual, Preliminary verification plan, SRR
- Architectural design
  - Architectural plan, PDR
- Detailed design
  - Detailed design plans, User's manual, Verification plan, CDR
- Implementation
  - Code reviews, Acceptance test plan, SCR, ATR
- Validation
  - All updated, Project evaluation, PRR, PPM

Verifikációs terv:

1.0 Introduction and overview

2.0 Reviews, walkthroughs, inspections, and audits

3.0 Component test plans and procedures

4.0 System test plans and procedures

5.0 Traceability from SRS to SDP

6.0 Test-requirements cross-reference matrix

7.0 Acceptance test and preparation for delivery

8.0 Additional information

1.0 Introduction and overview

- An overview of the entire V&V document.

1.1 Purpose of this Document

- Full description of the main objectives of the V&V Plan.

1.2 Scope of the Development Project

- This summarizes the project's goals and objectives.

1.3 Definitions, Acronyms, and Abbreviations

1.4 References

- Use proper and complete reference notation. Give links to documents as appropriate.

1.5 Overview of Document

- A short description of how the rest of the V&V Plan is organized and what can be found in the rest of the document.

2.0 Reviews, walkthroughs, inspections, and audits

- This section updates the planning for reviews, walkthroughs, inspections, and audits that was started in the team's Project Plan. The section should address both the procedures that the team has followed and the schedule of reviews, walkthroughs, inspections, and audits.

3.0 Component test plans and procedures

- This section describes the test planning for each individual software components.

3.1 Component test strategy overview

3.2 Component test case descriptions for Comp. k

4.0 System test plans and procedures

- This section describes the testing for the software product as a whole.

4.1 System test strategy overview

4.2 System test case descriptions for test 1

5.0 Traceability from SRS to SDP

This section is used to show (and discover) the specific connections between the SRS and the SDP. It is here that the unique identification for requirements and design sections becomes very important.

6.0 Test-requirements cross-reference matrix

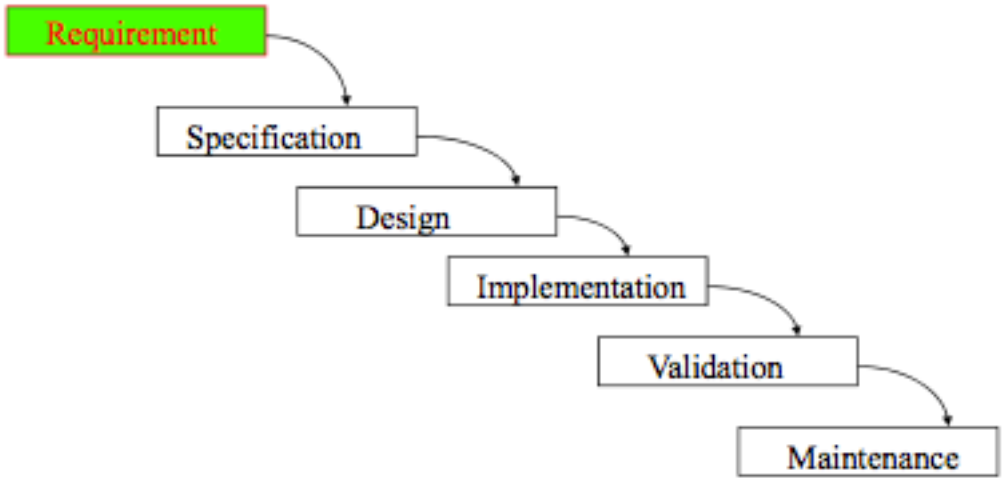
- The test-requirements cross-reference matrix shows how every requirement listed in the SRS will be tested. Information that can be provided for each requirement in the matrix:
  - The unique identifier for the requirement
  - A brief description of the requirement
  - The means by which the requirement will be tested (e.g. analysis, inspection, demonstration, test data)
  - Specific variables that are associated with the requirement
  - Ranges of values for the variables
  - System functionality associated with the requirement
  - Any comments

7.0 Acceptance test and preparation for delivery

- 7.1 Procedure by which the software product will be acceptance tested
  - Acceptance test cases should be based on the description of the product in the SRS document.
- 7.2 Specific acceptance criteria
  - This section will list the exact criteria that will be used to determine acceptance of the product by, or on behalf of, the client.
- 7.3 Scenario by which the software product will be installed
  - The installation may be a transfer process, it may involve creating a disk or CD for the client, or it may be a URL.

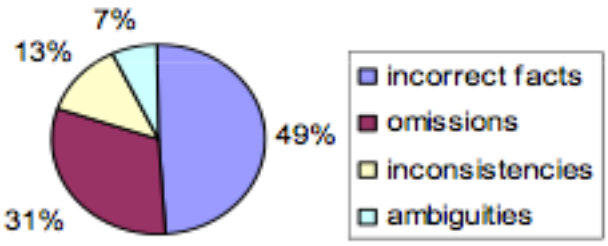
Követelmények

Szoftver életciklusának elsö fázisa:
- mik az elvárások, mik az igények és mik a korlátozások
- ezeket rögzíteni kell



Minél később veszünk észre egy hibát, annál drágább lesz kijavítani.
A hibák kb. fele a követelmény fázis környékéről származik. A hibák háromnegyede nem elírás, hanem valamit nem értettünk meg.

A követelmények során milyen hibákat vétünk:

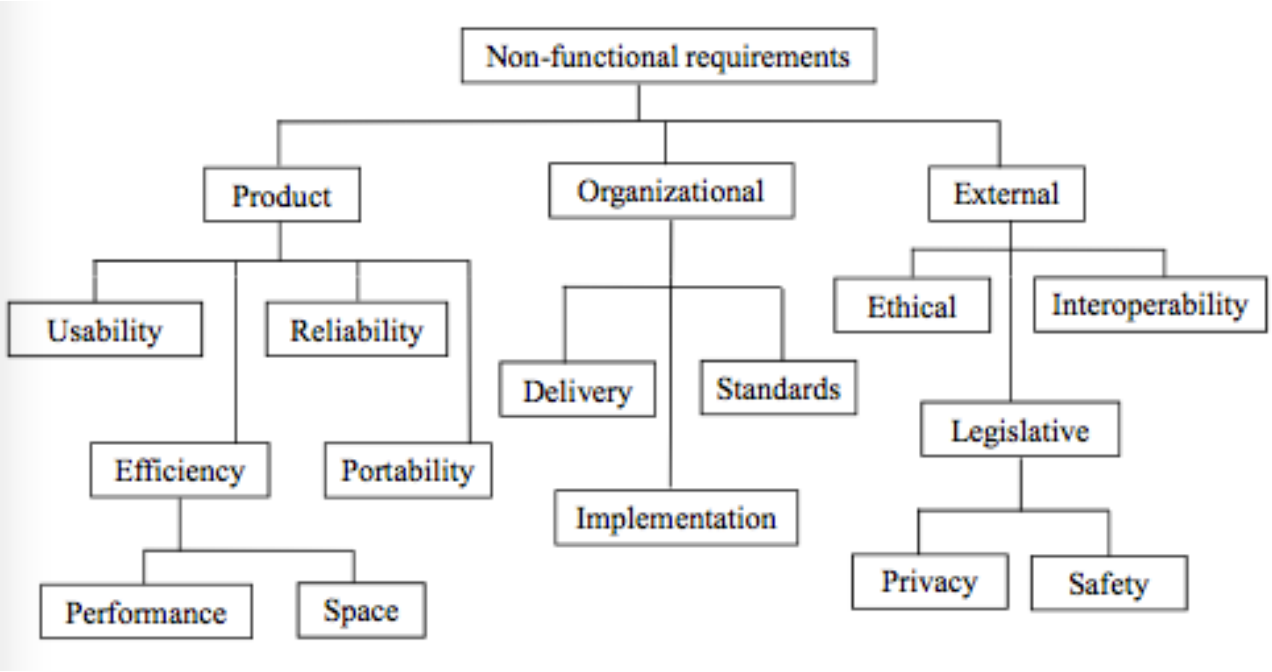


- hamis tények
- elhanyagolás
- nem egységesség
- többértelmüség

Követelmény definiálás: kevésbé formális
Követelmény specifikálás: komolyan formalizált, szerződés melléklete lehet

A követelmény azt mondja meg, hogy amit csinálunk az hogy viselkedik és mi a jellemzöje
- funkcionális követelmények
- nem funkcionális követelmények

Nem funkcionális követelmények:



- használhatóság (pl: valamilyen felkészültségü felhasználó tudja kezelni)
- megbízhatóság (müködjön éjjel-nappal, kiesés évi 1 nap lehet)
- használhatóság (válaszidök, felhasználási kapaciás)
- helytakarékosság (memória használat)
- portability
- szoftver terjesztése
- szabványok, melyeket be kell tartani
- programozási/implementációs elöírások
- interoperability/kikkel kell együttmüködni
- biztonság (hol lehet nyílt/titkos)
- személyiségi jogok (milyen adatok, hogyan féhetök hozzá)
- etikai, erkölcsi kérdések

Követelmény definíció jellemzöi:
- természetes nyelvünek kell lenni
- Minél ellenörizhetöbbnek kell lenni
- korrekt
- egyértelmü
- teljesnek kell lenni
- verifikálható (igazolható) - valami módszert kell adni arra hogy a követelmény teljesülését vizsgálni lehessen
- megérthetö
- módosítható
- követhetö

Mit kell a követelményekben rögzíteni?
- mi az információtartalma (az adatok mit jelentenek)
- az információ adatok hogyan mozognak
- az információ struktúrája mit jelent
- adatok, folyamatok
- hogyan vezérlünk
- egyes egységek hogyan kommunikálnak, kik kommunikálnak
- párhuzamosság, konkurencia
- idözítés, szinkronizáció
- viszonyok, relációk, egyéb függöségek
- korlátozások
- összeállítás (aggregáció)
- ésszerüség, magyarázat a szerkezetekre
- history, történetiség

Adat fluxus:
- Idöegység alatt beérkezö, feldolgozandó, kezelendö adatok, tranzakciók száma
- pl.: 2millió adóbevallást kell összeszen feldolgozni, de az utolsó 24 órában érkezik 1.3millió adóbevallás

Hogyan szerzünk információkat (követelmények gyüjtése)?
- interview-kat kell készíteni az érdekeltekkel
- kérdöíveket kell kiadni
- csoportokkal együttes találkozás
- kapott anyagok tanulmányozása

- Requirement
  - System definition, Project plan, PFR
- Specification
  - Requirement specification, Preliminary user's manual, Preliminary Verification plan, SRR
- Architectural design
  - Architectural plan, PDR
- Detailed design
  - Detailed design plans, User's manual, Verification plan, CDR
- Implementation
  - Code reviews, Acceptance test plan, SCR, ATR
- Validation
  - All updated, Project evaluation, PRR, PPM

- 1.0 Introduction
- 2.0 Functional and Data Description
- 3.0 Subsystem Description
- 4.0 System Modeling and Simulation Results
- 5.0 Project Issues
- 6.0 Appendices

1.0 Introduction

- This section provides an overview of the entire system or product. This document describes all subsystems including hardware, software, human activities, documents, and process.

1.1 Goals and objectives

- Overall business goals and project objectives are described.

1.2 System statement of scope

- Major inputs, processing functionality and outputs are described

1.3 System context

- The system is placed in a business or product line context.

1.4 Major constraints

- Any business or product line constraints that will impact the manner in which the system is to be specified, designed, implemented or tested are noted here.

2.0 Functional and Data Description

- This section describes overall system function and the information domain in which it operates.

2.1 System architecture

- A context-level model of the system architecture is presented.
- 2.1.1 Architecture model
- 2.1.2 Subsystem overview

2.2 Data Description

- Top-level data objects that will be managed/manipulated by the system or product are described in this section.
- 2.2.1 Major data objects
- 2.2.2 Relationships
- 2.2.3 System level data model

2.0 Functional and Data Description (cont.)

- 2.3 Interface Description
  - The system's interface(s) to the outside world are described.
  - 2.3.1 Machine interfaces
    - Interfaces to other machines (computers or devices) are described.
  - 2.3.2 External system interfaces
    - Interfaces to other systems, products or networks are described.
  - 2.3.3 Human interface
    - An overview of any human interfaces to be designed for the system/product is presented

3.0 Subsystem Description

- A description of each subsystem is presented.

- 3.1 Description for Subsystem n
  - A detailed description of each subsystem is presented
  - 3.1.1 Subsystem scope
  - 3.1.2 Subsystem flow diagram
  - 3.1.3 Subsystem n components
    - 3.1.3.1 Component k description (processing narrative)
    - 3.1.3.2 Component k interface description
  - 3.1.4 Performance Issues
  - 3.1.5 Design Constraints
  - 3.1.6 Allocation for Subsystem n

4.0 System Modeling and Simulation Results

- If system modeling and simulation and/or prototyping is conducted, these are specified here.

- 4.1 Description of system modeling approach (if used)
  - The system modeling approach (including tools and/or mathematical models) is described.
- 4.2 Simulation results
  - The results of any system simulation are presented with specific emphasis on data throughput, timing, performance, and/or system behavior.
- 4.3 Special performance issues
- 4.4 Prototyping requirements
  - If a system prototyping is to be built, its specification and implementation environment are described here.

5.0 Project Issues

- An overview of the overall system/product project plan is presented.

5.1 Projected development costs

- The results of system-level cost estimates are presented.

5.2 Project schedule

- A top-level schedule for the development project is proposed.

6.0 Appendices

- Presents information that supplements the System Definition

6.1 Business Process Descriptions

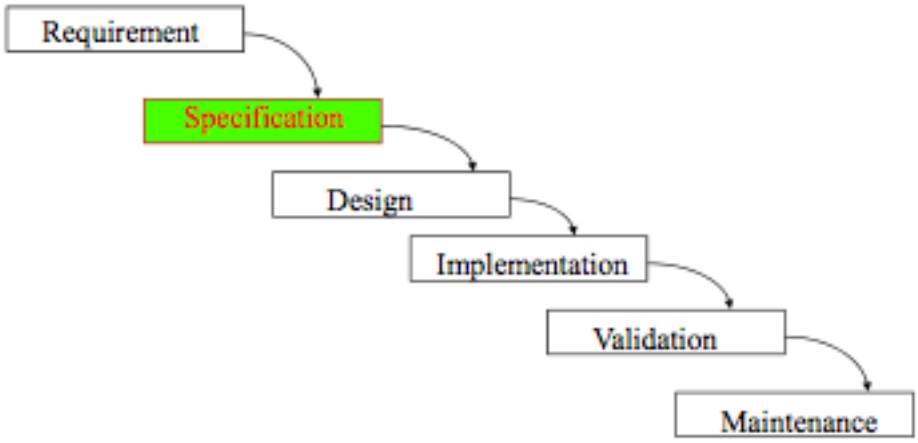
- If the specification is developed for a business system, a description of relevant business processes is presented here.

6.2 Product Strategies

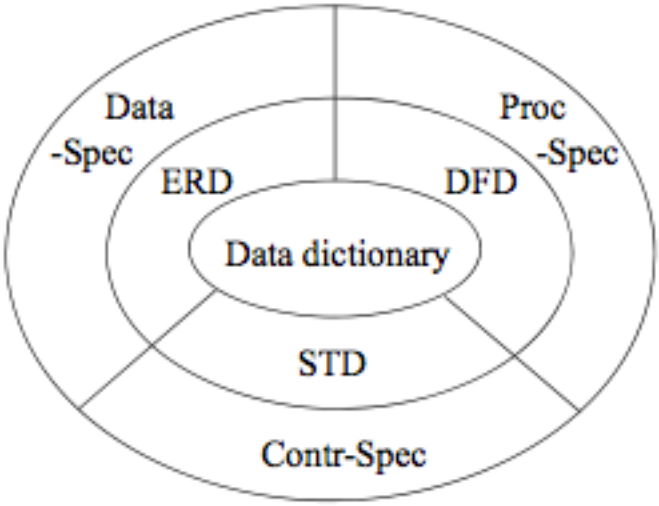
- If the specification is developed for a product, a description of relevant product strategy is presented here.

6.3 Supplementary information (as required)

Specifikáicó



Specifikáció: Formális leírás készítése, mely eleget tesz a követelményeknek.

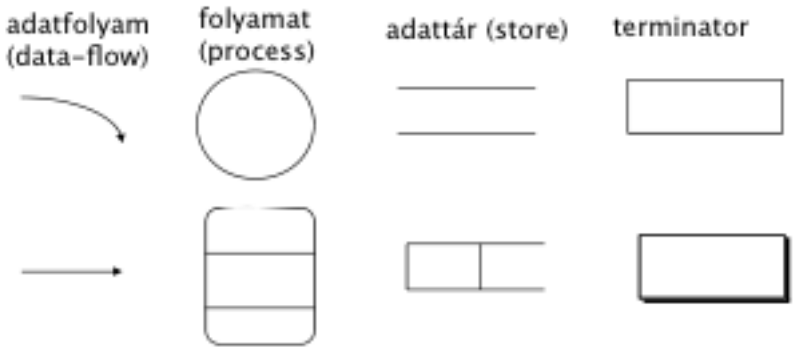


3 kép:
- **<u>funkcionalitás</u>: mit csinálunk**
  - **Data-flow diagram**
  - **folyamat specifikáció (process specifikáció)**
- **<u>szerkezet</u>: kik a szereplök**
  - **ERD: entitás-reláció diagram**
  - **struktúrális leírás**
- **<u>dinamika (idöbeliség)</u>: milyen sorrendben**
  - **állapotgép (state-transition diagram)**
  - **kontroll specifikáció**

Specifikáció - Funkcionális specifikáció

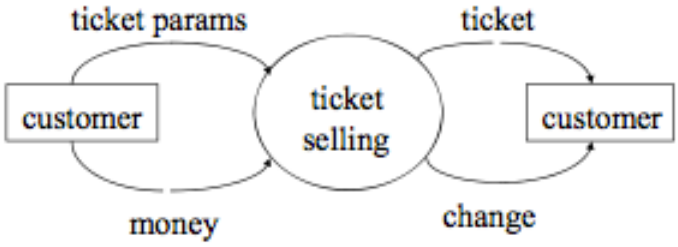
Funkcionális kép:
- inputból outputot generálunk
- részfeladatokra kell bontani

Data-Flow diagram:
- le lehet rajzolni vele a transzformációt
- jelölések:

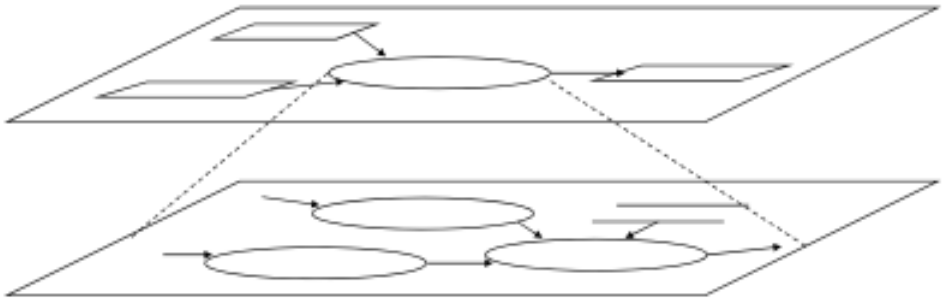


- process: adatot transzformál (legalább egy bemenet és kimenet), de nincs emlékezetet
- az emlékezet az adattár

Context-diagram (környezeti diagram):
- el kell tudnunk határolni hogy mi van a rendszerünkön belül és kívül

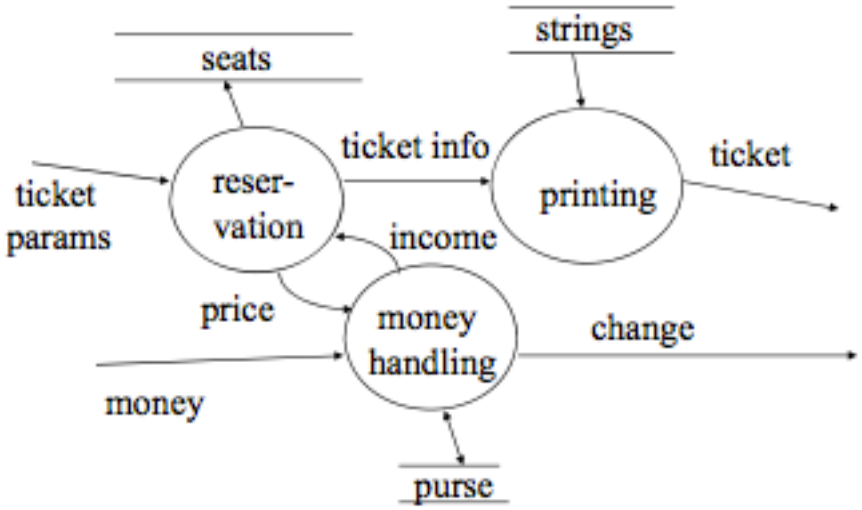


Példa: Az eladási folyamaton kívül egy szereplö van, a vásárló. Megadja a jegyparamétereket és a pénz. Ez át lesz transzformálva mozijeggyé és visszajáróvá.
A ticket selling elég bonyolult, szét kéne szedni.



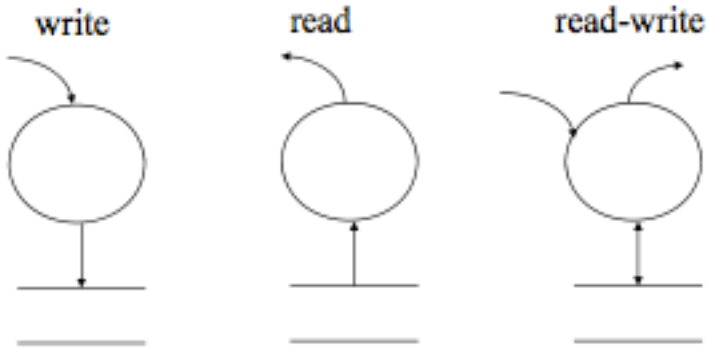
Egy gombócnak egy önálló DFD felel meg.
Hol van a kibontás vége? -> Amikor azt mondjuk egy gombócra hogy primitív gombóc.

Átlalában készítünk egy Context-diagramot és egy kibontását. A context-diagramban lévö gombóc a 0. gombóc, és az annak megfelelö DFD a 0. szintü kibontás. A DFD-ban a gombócokat is sorszámozzuk, onnantól pedig hierarchikusan (2.1, 2.1.1, stb...).



Példa: Mozijegy vásárlás. Bement a jegyparaméter és a bejövö pénz. Kimenet pedig a jegy és a visszajáró. Felbontottuk 3 lépésre.
- foglalás: lényege, hogy a rendelkezésre álló székeket és az igényeinket összevetjük
- nyomtatás: jegy adatokat és egyéb stringeket nyomtatunk ki (mozi neve, idö, ...)
- pénz kezelés: bejövö pénzt kasszába tesszük, a visszajárót visszaadjuk

Adattárak (store-ok):
- mindig valamilyen process-hez kapcsolódik, process nélkül nem íródik át
- context-diagramon nincs store, csak a data-flow diagramon
- ugyanaz a store különbözö szinteken/gombócoknál megjelenhet
- az idöalap is egy store



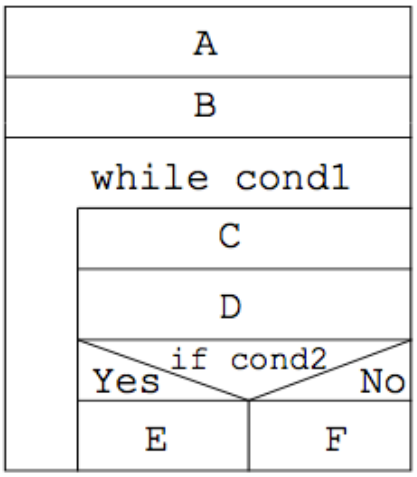
- a fájl és store írás feltételezi hogy az íráshoz szükséges olvasást végrehajtjuk (1. ábra)
- az olvasás egyszerü, olvassuk és továbbvisszük (2. ábra)
- jön egy bemenet, és ebböl mindenképpen van írás, de van egy plusz funkció. van a processnek egy kimenete, amiben részt kell vennie a storenak és lehet hogy a bemenetnek is (3. ábra)

Funkcionális kép lényege:
- a szoftvert adattranszformációként képzeljük el, ennek az ábrázolására van a context-diagram
- a context-diagramot bontottuk fel data-folyam diagramra

Process specifikáció:
- minden process gombóc kibontható egy data-flow ábrával
- megjegyzésekkel ellátott eljárás/funkció fej

- input/output lista
- processzben zajló tevékenység
    - struktúrált angollal
    - alapvetö szerkezeti konstrukciókkal
        - szekvencia
        - szelekció (választás)
        - iteráció

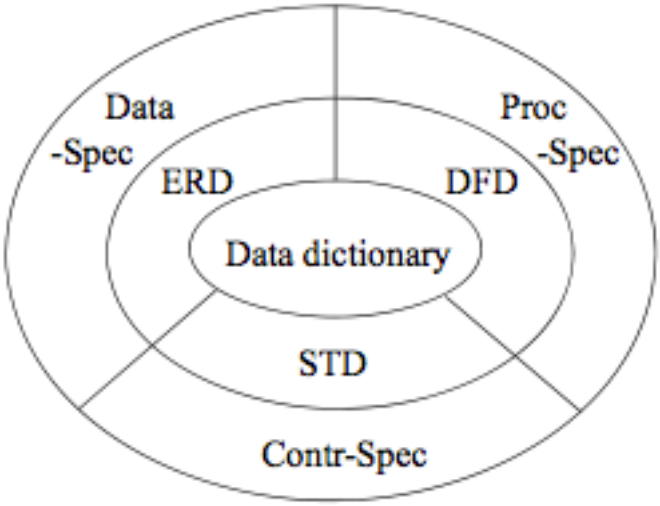
- flowchart




```
begin
  A;
  B;
  while cond1 do
    begin
      C;
      D;
      if cond2 then E else F
    end
end
```

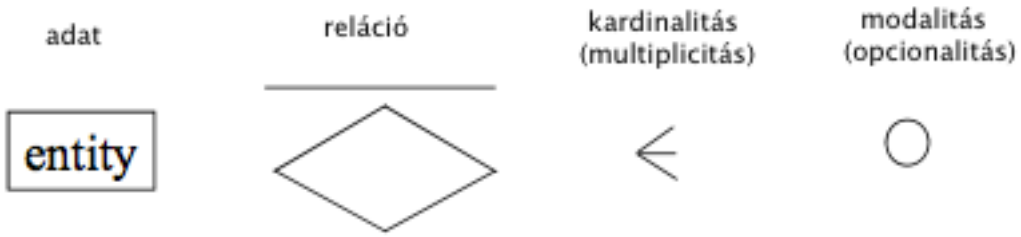
Feladatok Funkcionális Specifikáció témakörböl (DFD, CD) (KATT IDE)

Specifikáció - Szerkezeti leírás, adatmodellezés

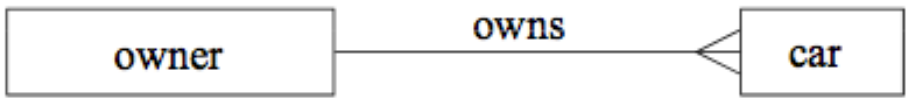


Adat:
- A világ valamely dolgáról valamiféle kép. Nem egy dolgot értünk rajta, hanem egy gyüjtönevet
- **attribútumok**: az adat jellemzöi. Az adat egy struct, az attribútumok pedig a struct-ban lévö mezök. Az attribútumok együttesen jelentik az adatelemet, mely az **entitás**.
- az entitások egymáshoz kapcsolódása a fontos

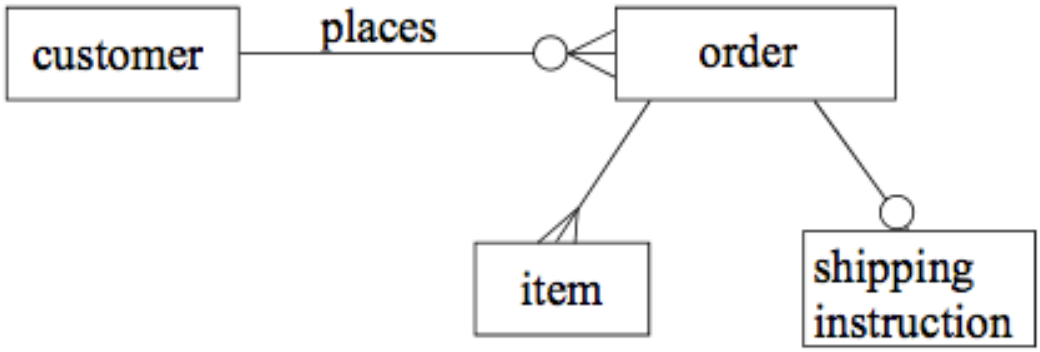
ERD (entitás-reláció diagram);
- az adatok és az adatok kapcsolatának leírására használjuk
- jelölések:



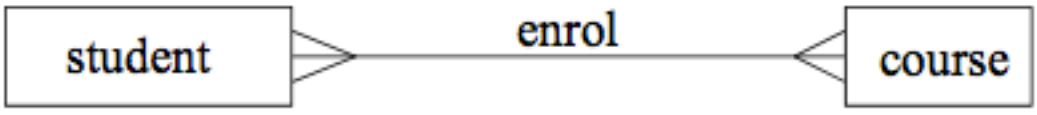
Kardinalitás (multiplicitás): két összekapcsolt adat közül az egyik egy példányához a másik hány példánya tartozik.
Modalitás (opcionalitás): a kapcsolat kötelezöségét jelenti (kötelezö-e a kapcsolat)

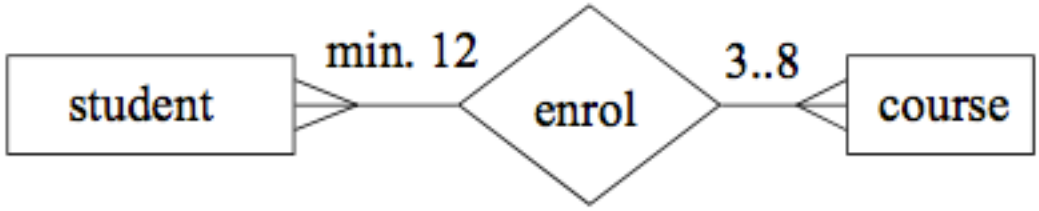


Példa: Tulajdonos-autó kapcsolat. A birtoklás a kapcsolat. Egy gazdához több autó tartozhat (csirkeláb), ebbe nem tartozik bele a 0, 1 vagy több autó tartozik egy gazdához. Egy autónak egy gazdája van (normális esetben).



Példa: Internetes vásárlás adat leírása. A rendeléshez tartozik több cikk, olyan rendelés nincs amihez nem tartozik semmi cikk. A megrendeléshez tartozhat kiszállítási információ, nem kötelezö. Egy kiszállítási információ egy rendeléshez tartozik. Egy megrendelési elem is szigorúan egy rendeléshez tartozhat. Egy ügyfelnek 0 vagy több megrendelése van.
Az egyedek közötti kapcsolatoknál fontos az egyedek közötti kapcsolat idöbelisége. A megrendelés az örökké megmarad, vagy csak átmeneti? Ki nem szolgált megrendelést tartjuk meg csak vagy az összes már kiszolgált megrendelést is?





Példa: Több-több kapcsolat a diák és tantárgy között. Az alternatív jelölés a rombuszos jelölés. Módomban áll konkrét számokat oda írni, hogy pl. egy kurzusra minimum 12 diák iratkozik be.

Eddig páronkénti kapcsolatokról beszéltünk. Több elem kapcsolatát általában felbontjuk páronkénti kapcsolatra.

Feladatok Entitás-Reláció Diagram témakörböl (KATT IDE)

XML

Adatok leírása - az XML:
- Extensible Markup Language
- HTML volt az ötlet amiböl ez alakult
- saját tag-eket tudunk készíteni

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Jack</to>
  <from>John</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
  <!-- This is a comment -->
</note>
```

Példa: Elsö sor a prologe, verziószám és kódolás a tartalma. Egy note-ot csinálunk.

XML szintaktika:
- mindig kell záró tag
- case sensitive (kis/nagybetü)
- korrekt skatulyázás kell
- kell a root elem
- spacek megmaradnak
- CR/LF-ek konvertálódnak LF-re (kocsivissza, sorlezárás)
- nevek: betüt/számot/egyéb karakterek, nem kezdödhet számmal/ponttal/.. a név, az XML-t nem lehet használni a nevek elején, nem tartalmazhat szóközöket

XML elemek:
- nyitó tag, záró tag
- benne lehet minden féle: más elemek, kevert, egyszerü, üres
- szülö/gyerek viszony a tag-ek között

```
<from>John
  <tel>12 34 56 78</tel>
  <e-mail />
</from>
```

Példa: John az egy szimpla text. A tel egy sima string. Az email egy üres tag, egyenértékü egy nyitó és egy záró taggel. (<email> </email>). A from kevert, van benne text (John), van egy tel ami egy element és egy email ami empty.

Attribútumok:
- <megnezés = “érték”>
- több attribútum is tartozhat hozzá
- az attribútumnak nincs belsö szerkezete, csak egy szöveg -> az attribútumok kezelése nem egyszerü

```
<note date="12/11/2002"> … </note>
```

XML element vagy attribútum:
- ha valami jellemzö a szerkezetben levö dologra vonatkozik, az element
- attribútum nem az ábrázolt dolog lényege, hanem az XML leírással függ össze (pl. sorszámozás)

```
<person sex="male">
  <firstname>John</firstname>
  <lastname>Lennon</lastname>
</person>
```

```
<person>
  <sex>male</sex>
  <firstname>John</firstname>
  <lastname>Lennon</lastname>
</person>
```

XML namespace:
- név konfliktus elkerülése (azonos nevü tag-ek)
- prefix (`<a:person>`)
- namespace attribútum (`xmlns="namespaceURI"`)

XML validáció:
- az XML lehet szintaktikailag helyes (jól formált)
- lehet érvényes, szemantikailag helyes
- `<![CDATA[" ..... "]]>` , a CDATA kivétel
  - CDATA = (unparsed) Character Data, olyan adat melyet az XML nem elemez ki, csak esetleg egyéb (külsö) programok
- PCDATA, minden XML által elemzett adat (Parsed Character Data)

DTD (Document Type Definition) séma:
- inline megoldás: az XML leírás elött kell lennie
  - `<!DOCTYPE root-element [element-declarations]>`
- external megoldás: include-olás
  - `<!DOCTYPE root-element SYSTEM "filename">`

- DTD példa:
  (PCDATA - parsed character data, ezt ellenörizni fogjuk)

```
<!DOCTYPE note [
    <!ELEMENT note      (to,from,heading,body)>
    <!ELEMENT to        (#PCDATA)>
    <!ELEMENT from      (#PCDATA)>
    <!ELEMENT heading   (#PCDATA)>
    <!ELEMENT body      (#PCDATA)>
]>
```

- építö elemek
  - element
  - tag
  - attribútum
  - entitás (különleges esetek, pl olyan szinte ahol ‘<‘ is szerepel)
  - PCDATA
  - CDATA

- DTD element

```
<!ELEMENT element-name category>
     category = EMPTY, (#PCDATA), ANY
```

- szekvencia `<!ELEMENT element-name (child-name, child-name,..)>`
- iteráció `(child-name+) - 1 or more,  (child-name*) - 0 or more`
- szelekció `(child-name?) - 0 or 1,  (choice1|choice2|choice3)`
- vegyes `<!ELEMENT note (#PCDATA|child-name)*>`

- DTD attribútumok:

```
<!ATTLIST element-name attribute-name
     attribute-type default-value>
```

- attribute-type: `CDATA  , (en1|en2|..)`
- default-value: `value, #REQUIRED, #IMPLIED, #FIXED`
- példa: `DTD: <!ATTLIST person number CDATA #REQUIRED>`

```
XML: <person number="5677" />
```

Feladatok XML témakörből (KATT IDE)

Algebrai axiómák

Formális leírás. Objektum-orientáltság matematikai hátteréhez közelít.

Absztrakt adatszerkezet:
- legyen az adatszerkezet rejtve, azzal foglalkozunk hogy müveleteket lehet rajta végzeni
- egy halmaz, melyen értemezve vannak müveletek, a részletek nem érdekelnek
- megmondjuk hogy milyen szabályok érvényesek a müveletek
- a müveleteknek szignatúrái vannak (fgv prototípus: neve, paraméter, visszatérési érték)
- axiómák - kifejezés = kifejezés'

Példa: *Stack*
- elemek: stack, item (amit beleteszek a stack-be), boolean
- szignatúrák (müveletek):

```
op params -> ret.value
NEW() -> stack
PUSH(stack, item) -> stack
POP(stack) -> stack
TOP(stack) -> stack
EMPTY(stack) -> boolean
```

- axiómák (sarok igazságok, két művelet egymás után amik oszthatatlanok)

```
EMPTY(NEW()) = true        // az új stack üres
EMPTY(PUSH(s, i)) = false  // ha egy stackbe teszünk valamit akkor
                              nem üres
TOP(NEW()) = undefined     // egy új stack teteje üres, ezért nem
                              értelmes hogy mi van a tetején
TOP(PUSH(s, i)) = i        // amit betettünk azt a tetejére tettük
POP(NEW()) = NEW()         // üres stack-böl még üresebbet akarunk
                              csinálni, üres stack-nél nincs üresebb
POP(PUSH(s, i)) = s        // nem változtatja a push/pop müveletekkel
                              a tetején kívüli elemeket
```

Müveletek az adatszerkezeteken:
- *konstuktor*: szükséges ahhoz hogy az összes lehetséges adathalmazt elöállíthassam (`PUSH és NEW`)
- *modifier*: a művelet eredménye maga az adatszerkezet, változtatja az adatszerkezetet és enélkül is megélhetek (`POP`)
- *behavior*: visszatérési értéke az adatszerkezten nem változtat, csak egy információt ad vissza (`EMPTY, TOP`)

Hány axiómát kell felírni?
- a behavior és a modifier müveleteket alkalmazzuk a konstruktorokon

Példa: *Tömb*
- elemek: tömb, int, item
- müveletek:

```
op params -> ret. value
NEW(int, int) -> array          (array-t létrehozás indexkorláttal)
ASSIGN(array, int, item) -> array
BOUND(array) -> int, int        (indexkorlát lekérdezése)
EVAL(array, int) -> int
```

- axiómák:

```
BOUND(NEW(x, y)) = x, y
BOUND(ASSIGN(a, n, v)) = BOUND(a) //egy elem berakása nem változtat
                                    az index határokon
EVAL(NEW(x, y), n) = undefined
EVAL(ASSIGN(a, n, v), m) = if (n==m) v
                           else EVAL(a, m)
// a tömb n. pozícióra v-t teszünk, majd megnézzük hogy milyen érték van
   az m. pozición. ha átírok egy elemet, akkor nem szabad megváltoznia a
   többi elemnek
```

Példa: *Lista (FIFO)*
- elemek: lista, int, item
- müveletek:

```
op params        -> ret.value
CRT()            -> list [konstruktor]
ADD(list, item)  -> list [konstruktor]
TAIL(list)       -> list  (farok képzés, elsö elemet kihagyjuk)
HEAD(list)       -> item [behavior]
LGTH(list)       -> int  [behavior]
```

- axiómák:

```
HEAD(CRT()) = undefined // üres listának nincs elsö eleme
HEAD(ADD(l, v)) = if (LGTH(l)==0) v
                  else HEAD(l) // ha a lista hossza 0, akkor v, ha
                                  nem 0 akkor marad a HEAD
LGTH(CRT()) = 0 // üres listánál nincs üresebb
LGTH(ADD(l, v)) = LGTH(l)+1
TAIL(CRT()) = CRT()
TAIL(ADD(l, v)) = if (LGTH(l)==0) l
                  else ADD(TAIL(l), v))
// egy listához adjunk egy elemet majd farok képzés, ha a lista
   üres akkor visszakapjuk az üres listát, ha nem üres a lista,
   akkor hozzáadjuk a v értéket a megfarkalt listához
```

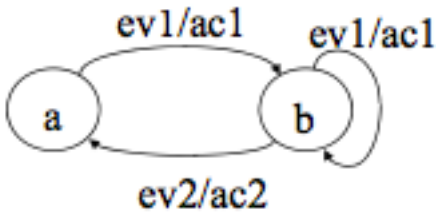
Feladatok Algerbai axiómák témakörböl (KATT IDE)

Specifikáció - Dinamikus leírás

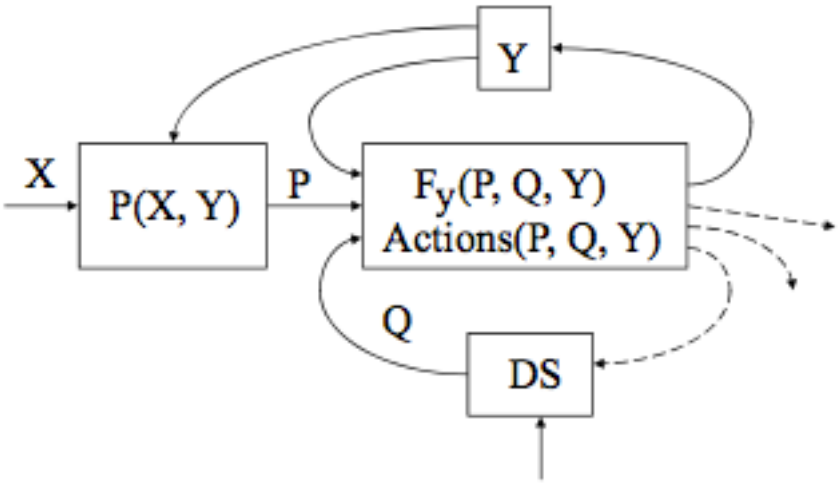
A szoftverrendszer reaktív rendszer:
- ha belerúgunk a szoftverbe akkor reagál valamire (belerúgás = event, esemény)
- az esemény 0 idö alatt játszódik le, oszthatatlan
- reagálás = valami tevékenységet végez, ami kívülröl vagy látszik vagy nem
- a reagálást oszthatatlannak tekintjük a modellezésnél
- ennek az aktivitásnak (milyen eseményre hogy reagál) a leírása az állapotgép

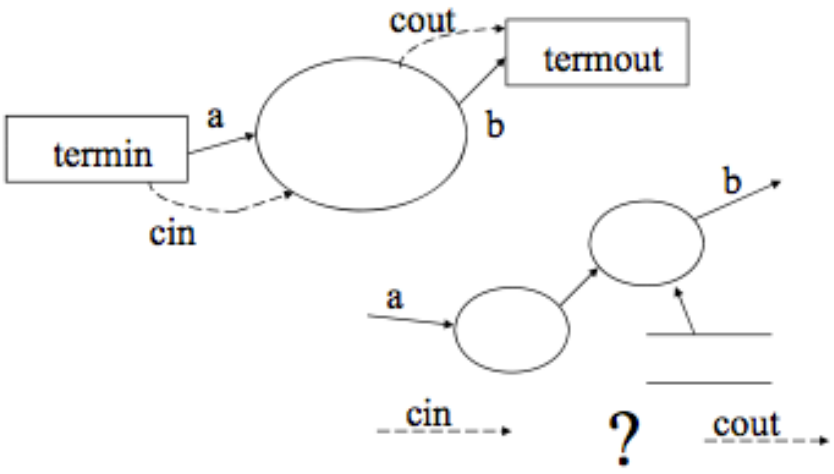
Állapotgép:
- egy eseményre nem mindig ugyanúgy reagálok, többféleképpen reagálhatok
- megmondjuk mit kell csinálni és hogy milyen állapotba lépünk
- állapottáblát érdemes állapotgép mellé készíteni
- az állapottábla alapján egyszerüen felrajzolható az állapotgráf

	ev1	ev2
a	b/ac1	
b	b/ac1	a/ac2



- az állapotgép az egy visszacsatolt memória
  - a visszacsatolt ág maga a processzor
  - itt a processzor olyan egyszerü, hogy csak az állapotot szállítja



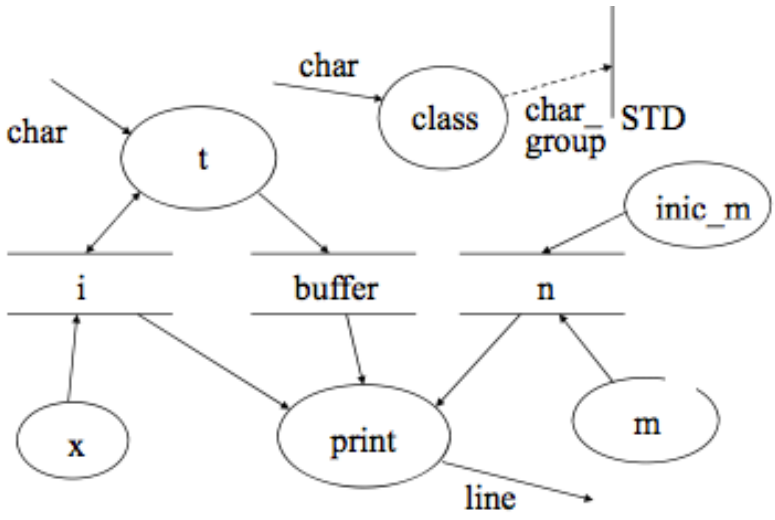


Példa: Context-Diagram. A-ból B-t transzformálunk. A CD csak azt mondja meg hogy A-ból B-t állítunk elö, de nem mondja meg hogy mikor van A.
A terminator adja azt hogy mikor érvényes A, illetve a terminator mondja B-nek hogy elveheti A-t. A kérdöjel helyére jön lényegében az állapotgép.

	letter	*separator*	*newline*
begofline	word1	begofline	begofline/m
word1	word1	afterw1	begofline/m
afterw1	word2/x,t	afterw1	begofline/m
word2	word2/t	throw/k	begofline/k,m
throw	throw	throw	begofline/m

x i = 0; **m** n ++ (inic n = 1)
t buffer[i++] = c; **k** if (i > 10) print(n,buffer);

Példa: minden sorból ki kell nyomtatni a sorszámot, illetve a 2. szót ha az 10 karakternél hosszabb.



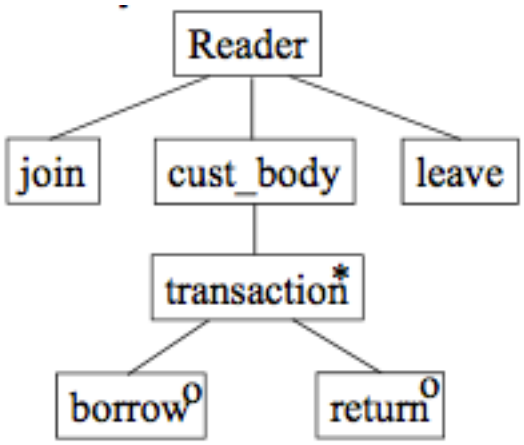
Feladatok Állapotgép témakörböl (KATT IDE)

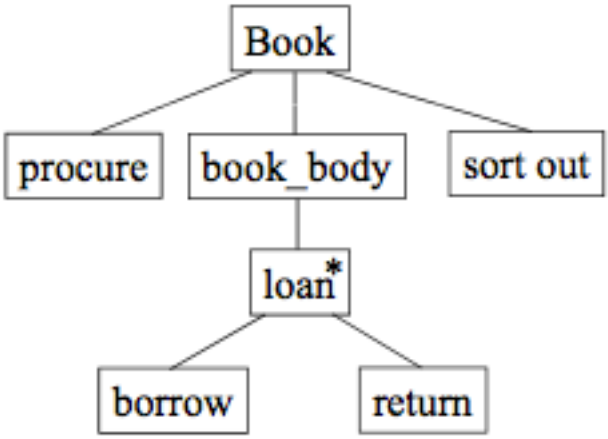
Jackson-ábra

A JSD egy entitások élettörtének megjelenítésére alkalmas leírási mód. Maga a leírás egy faszerkezet, melynek gyökér eleme az entitás. A fa szintekre osztható. A közös őselemből származó elemek alkotnak egy szintet. Egy szinten belül az elemek időbeli sorrendjét a helyzetük határozza meg. Két azonos szinten lévő elem közül az történt előbb, amely balra van.

A fában különböző típusú kapcsolat lehet az ős, és az alatta lévő elemek között.
- Szekvenciális - ekkor az alatta lévő elemek balról jobbra hajtódnak végre.
- Választás - a rajzon ilyenkor kör van a dobozok jobb felső sarkában. Ez azt jelenti, hogy csak az egyik hajtódik végre.
- Iteráció - a rajzon csillag, ekkor az elem valahányszor végrehajtódik.

Fontos szabály: egy szinten csak azonos típusú elemek lehetnek, ezért ha különböző típusokra van szükség, akkor azok egy alsóbb szinten helyezkednek el, ilyenkor egy üres dobozból származnak.
Tehát tömören arról van szó, hogy az entitás élettörténete a fa meghatározott módon történő bejárása, egy adott elem leszármazottait mindig balról jobbra járjuk végig, és ezt minden elemre megismételjük, amelynek vannak leszármazottai.





Feladatok Jackson-ábra témakörből (KATT IDE)

Specifikáció - összefoglalás

- Requirement
  - System definition, Project plan, PFR
- Specification
  - Requirement specification, Preliminary user's manual, Preliminary verification plan, SRR
- Architectural design
  - Architectural plan, PDR
- Detailed design
  - Detailed design plans, User's manual, Verification plan, CDR
- Implementation
  - Code reviews, Acceptance test plan, SCR, ATR
- Validation
  - All updated, Project evaluation, PRR, PPM

- 1.0 Introduction
- 2.0 Usage scenario
- 3.0 Data Model and Description
- 4.0 Functional Model and Description
- 5.0 Behavioral Model and Description
- 6.0 Restrictions, Limitations, and Constraints
- 7.0 Validation Criteria
- 8.0 Appendices

- 1.0 Introduction
  - An overview of the entire requirement document.
  - 1.1 Goals and objectives
    - Overall goals and software objectives are described.
  - 1.2 Statement of scope
    - Major inputs, processing functionality and outputs are described
  - 1.3 Software context
    - The software is placed in a business or product line context. Strategic issues relevant to context are discussed. The intent is for the reader to understand the 'big picture'.
  - 1.4 Major constraints
    - Any business or product line constraints that will impact the manner in which the software is to be specified, designed, implemented or tested are noted here.

2.0 Usage scenario

- This section provides a usage scenario for the software. It organized information collected during requirements elicitation into use-cases.
  - 2.1 User profiles
    - The profiles of all user categories are described here.
  - 2.2 Use-cases
    - All use-cases for the software are presented.
  - 2.3 Special usage considerations
    - Special requirements associated with the use of the software are presented.

3.0 Data Model and Description

- This section describes information domain for the software

3.1 Data Description

- Data objects that will be managed/manipulated
- 3.1.1 Data objects
  - Data objects and their major attributes are described.
- 3.1.2 Relationships
  - Relationships among data objects are described using an ERD-like form.
- 3.1.3 Complete data model
  - An ERD for the software is developed
- 3.1.4 Data dictionary
  - A reference to the data dictionary is provided. The dictionary is maintained in electronic form.

4.0 Functional Model and Description

- A description of each major software function, along with data flow or class hierarchy (OO) is presented.

4.1 Description for Function n

- A detailed description of each software function is presented. 4.1.1 Processing narrative (PSPEC) for function n
- 4.1.2 Function n flow diagram
- 4.1.3 Function n interface description
- 4.1.4 Function n transforms

  A detailed description for each transform (subfunction)
  - 4.1.4.1 Transform k description (narrative, PSpec)
  - 4.1.4.2 Transform k interface description
  - 4.1.4.3 Transform k lower level flow diagrams
  - 4.1.4.4 Transform k interface description

4.0 Functional Model and Description (cont)

- 4.1.5 Performance Issues
  - Special performance required for the subsystem is specified.
- 4.1.6 Design Constraints
  - Any design constraints that will impact the subsystem are noted.

4.2 Software Interface Description

- The software interface(s)to the outside world is(are) described.
- 4.2.1 External machine interfaces
  - Interfaces to other machines (computers or devices)
- 4.2.2 External system interfaces
  - Interfaces to other systems, products or networks
- 4.2.3 Human interface

4.3 Control flow description

- The control flow for the system is presented

5.0 Behavioral Model and Description

- A description of the behavior of the software is presented.

5.1 Description for software behavior

- A detailed description of major events and states is presented in this section.
- 5.1.1 Events
  - A listing of events (control, items) that will cause behavioral change within the system is presented.
- 5.1.2 States
  - A listing of states (modes of behavior) that will result as a consequence of events is presented.

5.2 State Transition Diagrams

- Depict the overall behavior of the system.

5.3 Control specification (CSpec)

- Depict the manner in which control is managed by the software.

6.0 Restrictions, Limitations, and Constraints

- Special issues which impact the specification, design, or implementation of the software are noted here.

7.0 Validation Criteria

- The approach to software validation is described.

7.1 Classes of tests

- The types of tests to be conducted are specified, including as much detail as is possible at this stage. Emphasis here is on black- box testing.

7.2 Expected software response

- The expected results from testing are specified.

7.3 Performance bounds

- Special performance requirements are specified.

8.0 Appendices

- Presents information that supplements the Requirements Specification

8.1 System traceability matrix

- A matrix that traces stated software requirements back to the system specification.

8.2 Product Strategies

- If the specification is developed for a product, a description of relevant product strategy is presented here.

8.3 Analysis metrics to be used

- A description of all analysis metrics to be used during the analysis activity is noted here.

8.4 Supplementary information (as required)

- Requirement
  - System definition, Project plan, PFR
- Specification
  - Requirement specification, Preliminary user's manual, Preliminary verification plan, SRR
- Architectural design
  - Architectural plan, PDR
- Detailed design
  - Detailed design plans, User's manual, Verification plan, CDR
- Implementation
  - Code reviews, Acceptance test plan, SCR, ATR
- Validation
  - All updated, Project evaluation, PRR, PPM

- 1.0 General information
- 2.0 System summary
- 3.0 Getting started (tutorial)
- 4.0 Using the system
- 5.0 Advanced features
- 6.0 Error messages
- 7.0 How to
- 8.0 Appendix, dictionary

1.0 General information

- 1.1 System Overview
  - Explain in general terms the system and the purpose for which it is intended.
- 1.2 Project References
  - A list of the references that were used in preparation of this document
- 1.3 Authorized use permission
  - A warning regarding unauthorized usage of the system
- 1.4 Points of Contacts
- 1.5 Organization of the Manual
- 1.6 Acronyms and Abbreviations

2.0 System summary

- should outline the uses of the system in supporting the activities of the user and staff
- 2.1 System Configuration
- 2.2 Data Flows
  - Depicts graphically the overall flow of data in the system
- 2.3 User Access Levels
  - Describe the different users and/or user groups and the restrictions placed on system accessibility or use for each
- 2.4 Contingencies and Alternate Modes of Operation
  - Explain the continuity of operations in the event of emergency, disaster, or accident. Explain what the effect of degraded performance will have on the user.

3.0 Getting started (tutorial)

- provides a general walkthrough of the system from initiation through exit

3.1 Installation

3.2 Logging On

3.3 System Menu

- describes in general terms the system menu first encountered by the user, as well as the navigation paths to functions noted on the screen
  - 3.3.x System Function x

3.4 Help

3.5 Exit System

4.0 Using the system

- provides a detailed description of basic system functions.

4.1 System Functions

- 4.1.x. System Function x

4.2 Special Instructions for Error Correction

- describes all recovery and error correction procedures, including error conditions that may be generated and corrective actions that may need to be taken

4.3 Caveats and Exceptions

- If there are special actions the user must take to insure that data is properly saved or that some other function executes properly, describe those actions here. Include screen captures and descriptive narratives

5.0 Advanced features

- provides a detailed description of advanced system functions in the structure of chapter 4.0

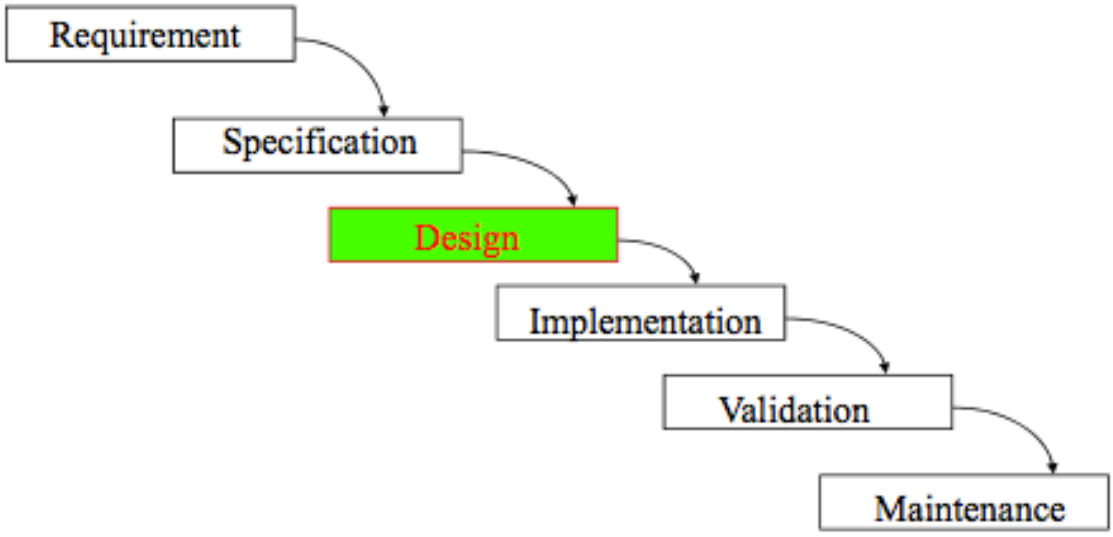
6.0 Error messages

7.0 How to

8.0 Appendix, dictionary

Design

A design-nak olyan kimenetet kell produkálnia, hogy a programozóknak ki tudjuk adni a feladatokat, had csinálhassák



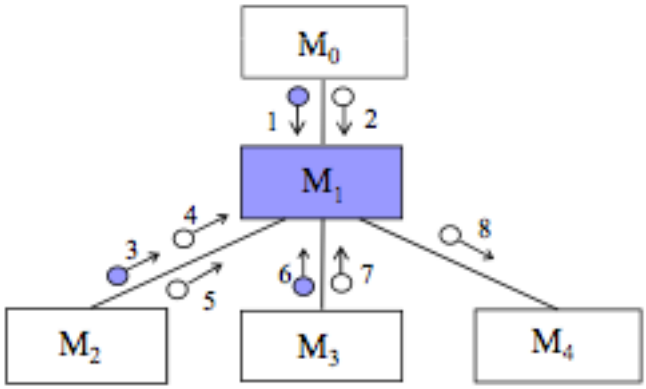
A tervezés (design):
- meg kell érteni a szoftver struktúráját és meg kell határozni a lépéseket amelyekkel ez létrehozható
- a tervezöi döntéseket dokumentálni kell
- döntéstechnikai kérdések
- elkészítendö dokumentumok; test plan, user manual
- koncepció kell arról hogy hogyan fogjuk implementálni, tesztelni, integrálni
- útmutatás a karbantartáshoz
- két része van
    - **architektúrális tervezés** (architectural design)
        - mik a fö komponsek és ezek hogyan kapcsolódnak egymáshoz
        - olyan szerkezetet kell adni amivel a feladat megoldható
    - **részletes tervezés** (detailed desing)
        - egyes eljárásokat, metódusokat, ürlapokat meg tudjuk határozni

Alapfogalmak, jótanácsok:
- hasznos, ha nem vagyunk csölátóak (“itt ez a probléma, csak ezt oldjuk meg”)
- követhetöség: minden egyes tervezési lépésnek követhetönek kell lennie
- nem kell újra feltalálni a kereket
- a tervezésnek összhangban kell állnia az alkalmazási területhez
- fel kell készülni a változásra
- a tervezés nem kódolás
- a tervezésnek is van minösége
- a terveket is felül kell vizsgálni

Asztrakció:
- részletekkel való játék
- érdemes végiggondolni, hogy mikor minek a részleteivel foglalkozok
- részletek finomítása
    - alkalmazási terület
    - informatika szemlélet
        - funkcionalitás köré
        - adat köré
        - kontrol köré

Encapsulation (összecsomagolás, struktúrálás):
- az elemeket hierarcikus rendben szeretjük tartani, de többféle szempotból és ez inkonzisztens
- próbáljunk meg olyan dobozokat létrehozni, amibe nem kell belelátni, szándékosan rejtsük el az egység tudásait az egység használója elöl (ha többet tudna ö belöle az csak zavarná), pl.: elektronikában kis fekete chip, foglalmunk sincs róla hogy belül hogy müködik, csak azt tudjuk mit csinál

Moldularizálás (modularity):
- hogy szedek szét bonyolultabb dolgokat modulokra
- nem szabad véletlenül csinálni
- ha egy modult szétszedek két modullá, akkor külön-külön a komplexitás kisebb mint az egy modul komplexitása (300 kg fa elhordása 10 lépésben könnyebben megy)
- struktúra diagram

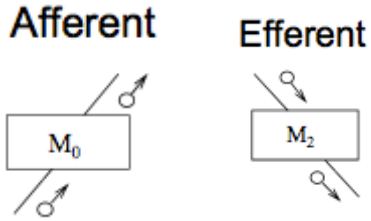


Példa:
Van egy vezérlömodul (M1), ami használja az M2, M3 és M4 modult. A vonal a <<uses>>-t valósítja meg.
Adat: üres gombóc
Vezérlöjel: teli gombóc

Ez egy egyszerü szoftver architektúra.

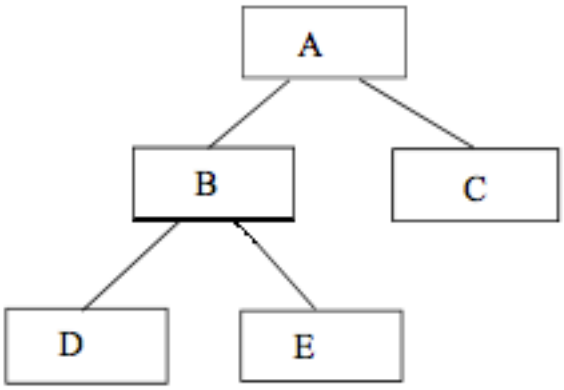
Ha alulról fölfelé áramlanak az adatok: afferens

Ha felülröl alulra áramlanak az adatok: efferens



Fan-out: “egy fönök alá hányan tartoznak”. A legjobb a 7.
Fan-in: “hány fönököm van”. Nincs legjobb.
- kevesen használnak minket: nem vagyunk fontosak
- sokat használnak minket: fontosak vagyunk, de ha hibázunk akkor sokminden múlik rajta

Vezérlési határkör: önmaga illetve minden alá tartozó modul
Döntési hatáskör: azon más modulok, melyeket érintenek a modul döntése

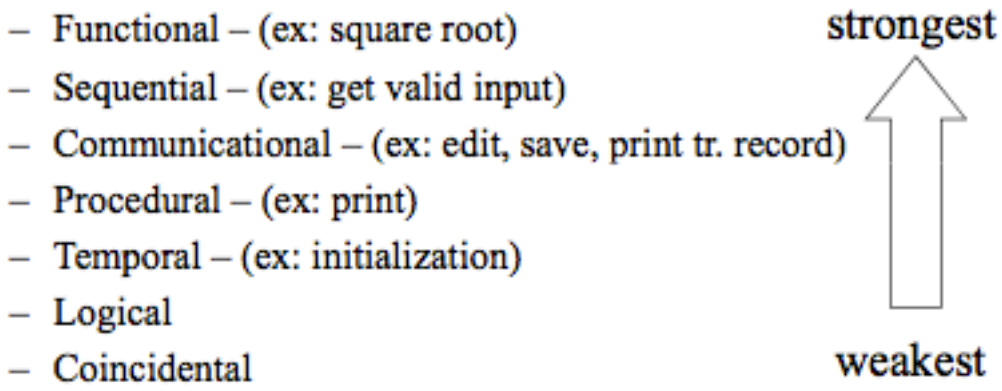


Példa: B vezérlési hatásköre: B,D,E. A döntési hatáskört szemantika nélkül nem tudjuk megmondani, de még a C is benne lehet.

Döntés hasítás: döntési hatáskör szélesebb mint a vezérlési hatáskör
(pl: B-ben van egy döntés hogy meg kell hívni C egy funkcióját, A-nak adunk egy vezérlést hogy hívja meg C-t. A vagy meghívja C-t vagy nem. 3 év után egy új ember módosítja C-t dokumentáció hiányában, de ö nem tudja hogy C-t meg kell hívni, ezért B el foghasalni)
- el kell kerülni a döntés hasítást

Csatolás:
- elemek közötti kapcsolat kérdése
- minél jobban össze vannak kötve az elemek, annál nehezebb az elemeket cserélni
- laza csatolás esetén könnyen lehet módosítani egy elemet
- dimenziók:
    - kapcsolat tárgya
        - laza csatolás: primitív adatokat, primitív adatok tömbjét adom át
        - stamp csatolás: kompozit (összetett) adatot adok át
        - control csatolás: vezérlést adunk át, olyan információt adunk át mely a fogadót vezérli, kívülröl beavatkozunk a vezérlésébe, kezdek belelátni
        - közös adatok: nem tudjuk hogy kinek a kezében van, bárki hozzáférhet -> veszélyes használni
        - tartalmi jellegü csatolás: egyik modulból a másik programrészlet kódját mint adat kezelem -> kód átírása, “vírus”
    - kapcsolat mérete
        - minél több adatot adunk át, annál nagyobb a valószínüsége annak hogy baj lesz, könnyü paramétert felcserélni, rosszul sorrendezni, stb...
    - kapcsolat ideje
        - program írás során
        - compiler kapcsolja össze
        - linker, kapcsolati idöben
        - load time, program betöltödése során
        - futás idöben (pl.: polimorf metódushívás OO programozási módban)

Kohézió (cohesion):
- valamiféle összatartó erö
- egy objektumban mennyire állnak közel a metódusok
- mennyire kohézív két metódus
- mennyire kohézív egy adott package-ben lévö osztályok, mi az összetartó erö
- kohézió intuitív osztályozása



Funckionális kohézió: A gyökvonásnál nincs olyan pont ahol szét lehet törni a funkciót.
Szekvenciális kohézió: A get valid input-nál már a get és a valid között szét lehet törni a funkciót, tudjuk csökkenteni a kohéziós eröt.
Még gyengébb kohézió a *kommunikációs kohézió*. Egy adatszerkezeten müveleteket végzünk, és a müveleteket az tartja össze, hogy ugyanazon az adatszerkezeten végznek müveleteket.
A *procedurális kohézió*: C-s printf függvény, mellyel string-et és double-t is be tudunk olvasni, de nincs sok közük egymáshoz. Típusfüggö switch szerkezet mindig procedurális.
Temporális kohézió: olyan elemek közötti kapcsolat, melyeket az idöbeliség tart össze.
Logikai kohézió: valami logika kapcsolja össze az elemeket.

Objektum-Orientáltság tervezés szempontból

“Tervezzünk szerződések szerint” (design by contact)
- mi a szerződés elöfeltétetle
- szerződés után mi lesz az állapot
- vannak kötelezettségek és hasznok

provide_service	Obligations	Benefits
(Client)	(Satisfy precondition)	(From postcondition)
Subscriber	Pay bill	Get telephone service
(Supplier)	(Satisfy postcondition)	(From precondition)
Company	Provide telephone service	No need to provide anything if bill not paid

Példa: Kliens - cég elöre számlabefizetés. Kliens szempontjából: kötelezettség az hogy ki kell fizetni a számlát, és az a haszna hogy neki telefonszolgáltatást kell kapnia. Cég szempontjából: kötelezettsége az hog ha valaki fizet akkor köteles szolgáltatást biztosítani, és az a haszna hogy csak annak kell szolgáltatni aki fizetett is.

Prekondíció (elöfeltétel): kliens számára kötelezettség, a céget megvédi
Postkondíció (utófeltétel): kliens haszna, kötelezettség a cég számára
Class invariants: alapvetö szabályokat ír elö osztályokra

Eiffel programozási nyelv:
- design by contract elvet megvalósító nyelv

```
class interface ACCOUNT
feature -- Access
    balance: INTEGER -- Current balance
    deposit_count: INTEGER -- Number of deposits
feature -- Element change
    deposit (sum: INTEGER) -- Add sum to account.
         require
            non_negative: sum >= 0
         ensure
            one_more_deposit:
            deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
invariant
    consistent_balance: balance = all_deposits.Total
end -- class interface ACCOUNT
```

Példa: Bankszámla, melynek balance (számlaösszeg) és befizetés számláló attribútumai vannak. Van olyan operáció benne hogy befizetés, az elöfeltétel az hogy a befizetésnek nem szabad negatívnak lenni, ez az elöfeltétel. A deposit szám az az olddeposit szám+1, és bevezetek egy újabb postkondíciót az updatet.
Bevezetek egy invariáns, a balansz egyenlö az összes befizetések összességével.

- Az elö és utófeltételek nem utasítások, hanem megállapítások. Csak állításokat közlök, melynek logaikai értékei vannak, melyek a szerződés feltételeit teljesítik. Ennek a meglétének meg lehet vizsgálni ezzel a programmal. Leírom deklaratív programként is.

- Debug közben fut általában
- ha futás közbenen egy szabály megsérül
    - prekondíció sérül: a kliens hibázott
    - postkondíció, inveriáns sérül: az objektum aki szolgáltat nem úgy viselkedik ahogy kell

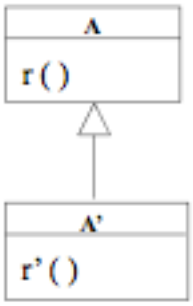
Öröklés:
- az osztályokon leszármaztatást végzünk, új osztályokat hozunk létre
- ha az elö- és utófeltételeket figyelembe vesszük akkor nem egyértelmü az öröklés, szabálymegmaradási tétel
- class öröklés: brutális, felhasználjuk és nem törödünk többel, kódújrahasznosítás
- interface öröklés: ezt szeretjük, a helyettesíthetöségre építünk

- “az ellipszis egy kör fajtája? elipszissel helyettesíthetö a kör? a kör egy ellipszis?”
    - az ösosztály egy speciális eset
- “John örökölte a szeme színét a papájától”
    - két entitás, van egy reláció köztük, van örökölt tulajdonság (szemszín) és egy mechanizmus (genetika)
    - ezeket a gondolatokat végig kell vinni az OO világba, ha örököltetni akarjuk

Példa: A1 A-t létrehozta, C dolgozik az A-n, A2 változtatni akarja A-t. A-nak nyitottnak kell lennie hogy C dolgozhasson rajta biztonságosan, de nyitott hogy A2 változtathasson rajta.

- alváltozatok, verziók létrehozása örökléssel
    - a hármas verzió az eredetiböl, vagy a kettesböl öröklödik?
    - van egy alma osztályom. a piros alma a leszármaztatott osztály?
- heterogén kollekció
    - közös ösosztályokat hozunk létre a heterogén kollekcióhoz
- aggregáció szimulálása (nem biztos hogy jó rá az öröklés)
    - autó: ha a kerék sebesség az autó sebességböl örökölödik, akkor a kerék az autóból öröklödik?

Öröklés és szerzödés viszonya:



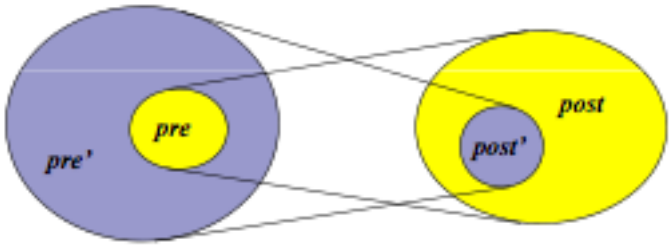

```
r is
   require
     pre          inv
     …
   ensure
     post

r' is
   require
     pre'         inv'
     …
   ensure
     post'
```

Ha r-ben van elöfeltétel, inveriáns és posztkondíció. Akkor a pre és pre’, inv és inv’, post és post’ között mi a kapcsolat?

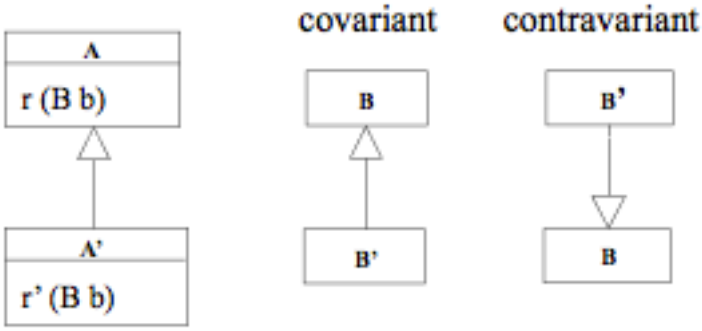
Def: P állitás erösebb vagy egyenlö a Q állítással, ha P implikálja Q-t.

Invariánsra vonatkozóan: ha több helyröl öröklünk, akkor az invariáns a többszörös öröklésböl származó kondíciók és kapcsolata.

Redeklarációs szabály: Az elöfeltételnek gyengébbnek vagy egyenlönek kell lennie az ösnél. Az utófeltételnek erösebbnek vagy egyenlönek kell lennie az ösnél.



Paraméter átadás esetén:



Állhat-e a leszármazott müveletnél paraméterként leszármazott érték? -> problémák vannak, mert az ösosztállyal együtt kell futtatni, veszélyes szerkezet

Ez a kovariancia hátránya.

Kontravariáns: Az ösosztálynál van a B leszármaztatottja, és a leszármazott osztályban van a B osztály. Ilyen nem nagyon tud elöfordulni.

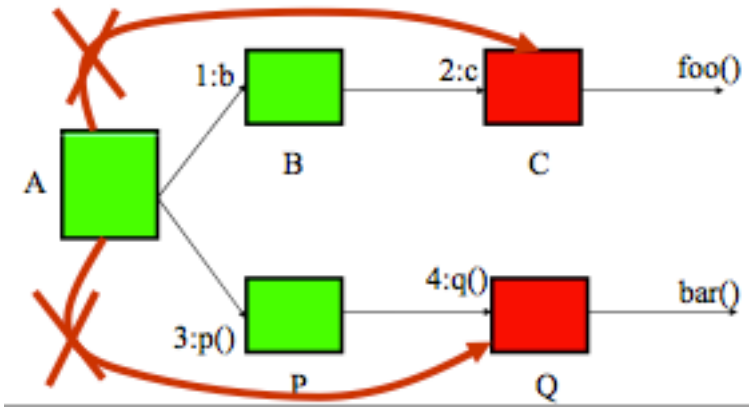
Példa:
(tegyük fel hogy A-ból le van származtatva B)

```
A[] aa = new A[10];
aa[2] = new A();     // korrekt

A[] aa = new A[10];
aa[2] = new B();     // korrekt

A[] aa = new B[10];
aa[2] = new B();     // korrekt

A[] aa = new B[10];
aa[2] = new A();     // a compiler lefordítja, compile time a második sok nem nagyon
                        ellenörizhetö, de runtime nem müködik, B elemeknek
                        foglaltunk helyet, de A-t akarunk tenni oda
```

Lemeter törvénye (the Law of Demeter, LoD):
- “dont talk to strangers”
- ne kössünk össze idegen objektumokat egymással
- ki nem idegen?

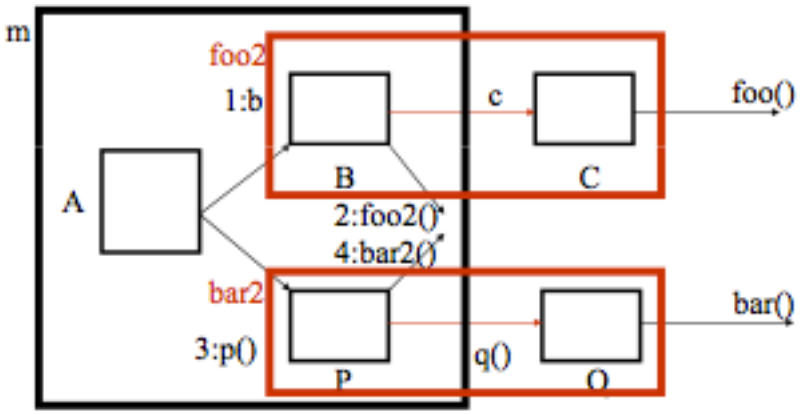
Példa: C osztálynak M metódusa van. M metódusban el kell érni a paramétereket amelyeket kaptunk, this-t, super-t, az osztályban deklarált attribútumokat, az átmeneti lokális válzotókat, globális változókat, M átlal kreált más osztályok példányait.

```
class A {public void m(); P p(); B b; };
class B {public C c; };
class C {public void foo(); };
class P {public Q q(); };
class Q {public void bar(); };
void A::m() {
    this.b.c.foo();
    this.p().q().bar();}
```



Példa: Túl messzire nyúlunk, sokat tudunk A-ban, A-ba beépül az egész szerkezet, ami nem biztos hogy mindig így marad.

Megoldás: Beveztünk foo2 és bar2 metódust. Azokat hívjuk meg b-n és p-n. Próbáljuk meg rejteni a kapcsolatokat.



Design - Szoftver archiketúrák

Architektúra definiíció:
- művészettudomány építészetre vonatkozóan, tervezés és stilus [Oxford distionary]
- a számítástehnikai komponensek gyüjteménye, komponensek közötti együttmüködés megadásával [Garlan & Shaw]
- a programnak a szerkezetéhez kapcsolódik, együttmüködö szoftver egységek, és definiálja az együttmüködéshez a kapcsolatokat [Schwanke, Altucher & Platoff]

A szoftver architektúra döntéseket is követel:
- elemek megválasztása
- együttmüködési mód (kollaboráció)
- hierarchikus rendszer az együttmüködö elemekböl
- architektúrális mintákat használhatunk
    - definiál egy szótárat amit használunk
    - szótárho tartozó korlátozás
    - szemantikai modell
- architektúrális kép
    - a specifikációs modelleket összevágva kapunk egy speciális képet

Jó architektúra jellemzöi:
- rugalmas, könnyen változtatható
- egyszerü
- jól elválasztott szempontok
- felelösségek kiegyensúlyozása

Milyen elemek lesznek benne?
- üzleti (business) elemek
- fontos mechanizmusok, eljárások
- processzorok, processzek
- ezek mind hierarchikusan vannak

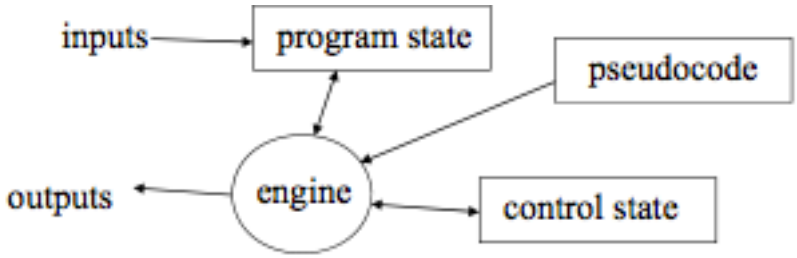
Architektúrális minták

Fö kategóriák:
- Adatfolyam rendszerek
- Call-and-return rendszerek
- Adatközpontú rendszerek
- Virtuális gép alapú rendszerek
- Független komponensek

Pipes and filters (csövek és szürök):
- megvalósítása a UNIX
- a filterek transzformálnak, a pipeok összekötik öket
- elöny:
    - erösen támogatja az újrahasználhatóságot
    - egyszerü karbantartani, mert jól szeparáltak
    - konkurrens megoldást támogatja
- hátrány
    - batch-hez vezet el

Blackboard (hagyományos adatbázis-kezelö rendszer);
- középen egy nagy adatbázis (osztott adatok sokasága)
- az adatbázishoz kapcsolnak körben processek
- az adatbázis egy nagy közös adatszerkezet
- megtestesíti az összes bajt, amit a globális adat okozhat
- elöny:
    - jól el vannak különítve a felelösségek
    - az adatbáziskezelö gyárilag tudja (vagy felokosítjuk) hogy a problémás adatok kezelését segítse, illetve a zavaros helyzeteket lemenedzselje
- hátrány:
    - nem könnyü tesztelni

Interpreter:
- virtuális gép
- állapotgép, mely interpretálja a táblázat tartalmát
- egyenes folyatása az állapotgépes specifikációnak
- bejár egy táblát, a táblázatot tudom címezni az eventtel és az aktuális állapottal, oda beírom a következö állapotot és annak a kódját hogy mit kell végrehajtanom

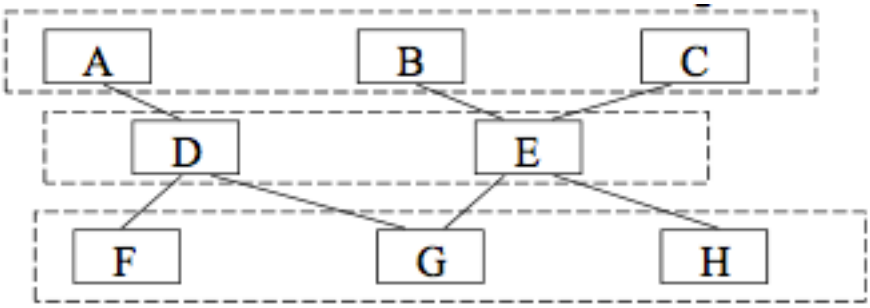


engine = gép, maga a program
állapottábla = pseudocode
control state = aktuális állapot
program state = változók, adatelemek melyek kellenek a végrehajtáshoz és az inputok

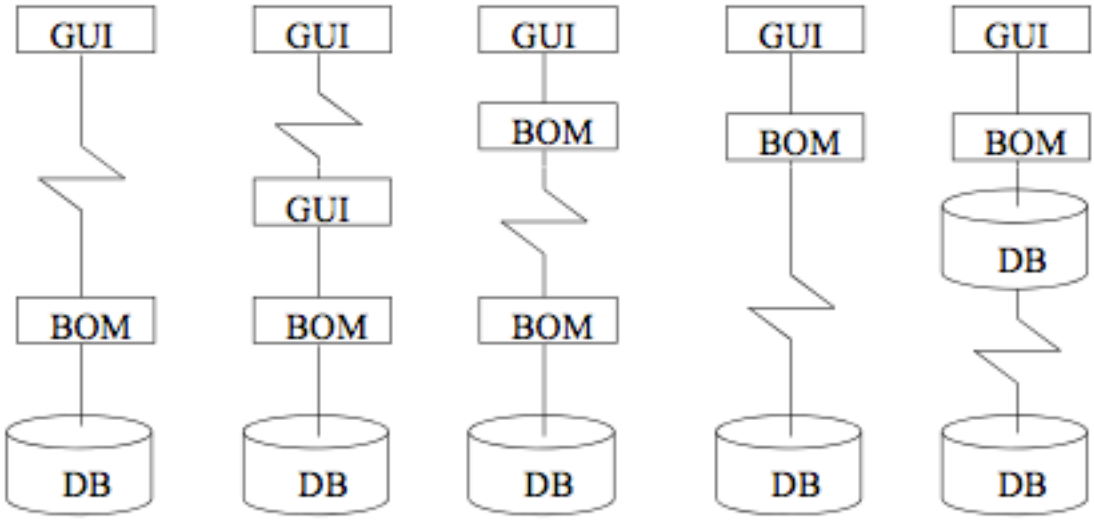
Objektum-orientált minta:
- szolgáltatásokat nyújtó szerkezeteket hozunk létre, melyek adatokat is tartalmaznak
- az objektumok egymás müveleteit hívják
- az objektum felelös a saját metódusáért
- kivülröl ne lehessen látni az objektum szerkezetét
- hátrányok:
    - ahhoz hogy az objektum szolgáltatásait el tudjuk érni, ahhoz ismerni kell valamennyire az objektumot

Event-based, implicit invocation (public subscriber, observer):
- van valamilyen eseményforrás
- érdekelt elemek beregisztrájják magukat az eseményforrásra, ha az bekövetkezik akkor értesítést kap róla

Rétegelt szerkezet (layered):
- hagyományos modul szerkezet, de észreveszünk belle layereket (vízszintesen)
- egy-egy layer (modulok együttese) kicserélhetö, és helyettesíthetö egy másik layerrel
- klasszikus példa erre: hálózati protokol stack (TCP/IP)
- egy adott rétegen modulok együttmüködnek
- az a célszerü ha egy réteg csak az alattalévö rétegre támaszkodik
- egyszerüen karbantartható
- hátrány:
    - nem mindig lehet ilyen rétegelt struktúrát csinálni
    - hatékonysági okokból át kell nyúlni a rétegeken (TCP/IP-ben is átnyúl)

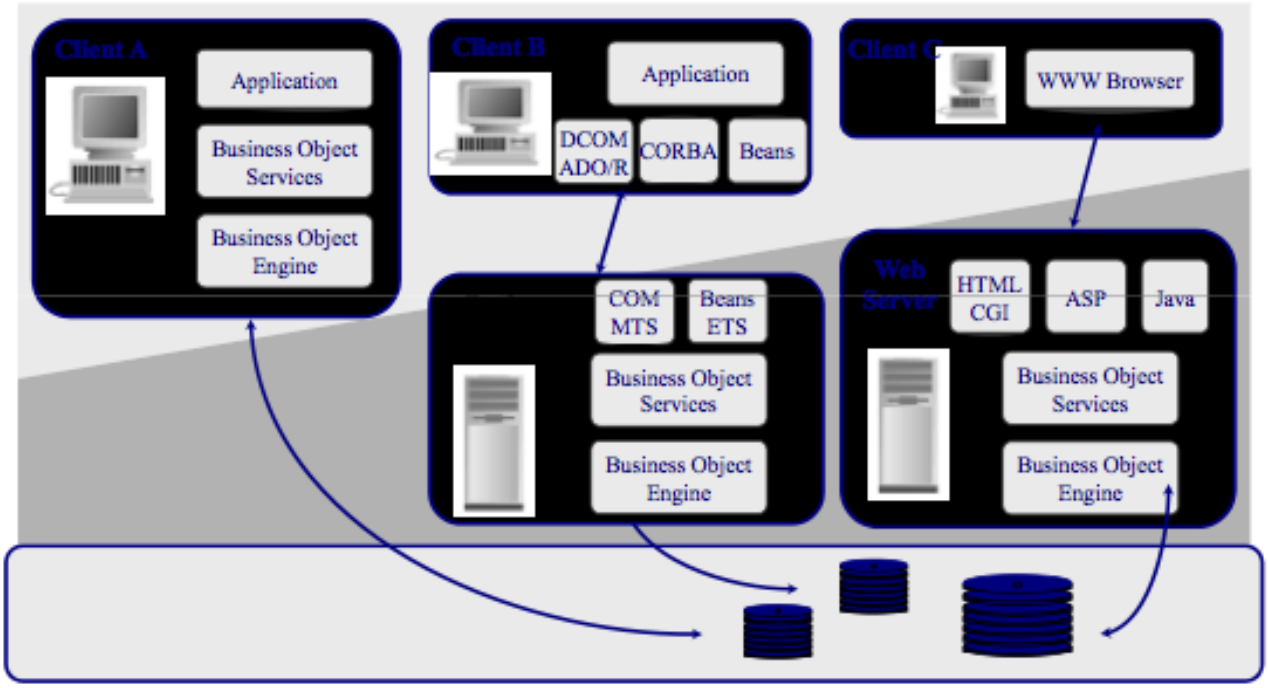


Kliens-szerver modell:
- 3 logikai réteg
    - GUI grafikus felület
    - BOM (Business Object Model): business objektumok
    - adatbázisok
- a 3 réteget gyakran két fizikai rétegen helyezzük el, példák erre:



- ha 3 fizikai rétegen osztjuk meg: egy-egy fizikai rétegen egy-egy logikai réteg

- történelem:



1. Kezdetben volt egy központi adatbázis, ahhoz hozzáfértek a kliensek, ahol telepített programok voltak. A kliens össze volt kötve az adatbázissal.

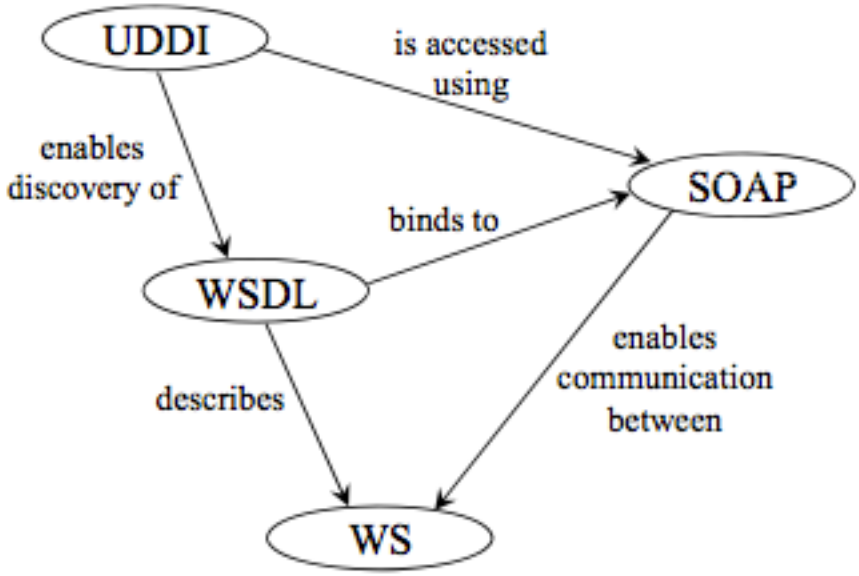
2. Később távoli kapcsolatokat hoztak létre, okosabb lett a kliens, távoli adatbázis kapcsolat jött létre. A kliensnek nagyon okosnak kellett lenni.

3. Megjelent a második réteg, a kliens egyszerüsödött. Az adatbázis és a kliens közé egy modelt kellene tenni. A kapcsolat standard módon volt megoldva. Középen megjelent egy business model, valójában az alkalmazásszerver elözménye.

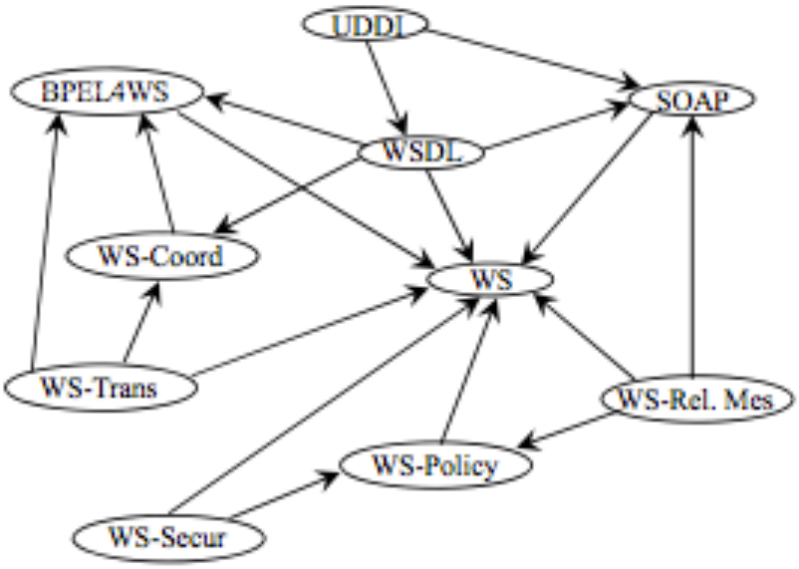
4. Mostmár a kliens helyén egy böngészö van csak. Van egy erös középsö szerver, webszerver, alkalmazásszerver.

SOA - Service Oriented Architecture:

- ilyen bróker az árgép (www.argep.hu) és a netrisk is (www.netrisk.hu)
- webszolgáltatás felületen el tudjuk érni a szolgáltatásaikat, nem klasszikus HTML megoldást kapunk
- mi XML-eket küldünk, majd erre XML-ben kapunk valami választ amiböl felépül az oldal
- szolgáltatás-orientált architektúra, központja a WS (web sercive)
- WSDL:
    - webszolgáltatásokat leíró nyelv,
    - leír interfészeket, üzeneteket, interfészek közötti kapcsolatokat, összerendeléseket
- SOAP:
    - üzenetformátum (fejléc, törzs)
    - RPC-t takar el (távoli eljárás hívás)
    - XML-es kommunikáció webszolgáltatások elérésére, WSDL kötödik ehhez
    - node-okat tud definiálni (sender = küldö, reciever = vevö, intermediary = áteresztö node)
- UDDI:
    - ‘yellow-pages’ = ‘aranyoldalak’, business registry
    - elö lehet fizetni rá
    - ha tudom hogy van az aranyoldal akkor rajta keresztül el tudom érni a webszolgáltatásokat



SOE - Service Oriented Enterprise:
- standard szolgáltatásokat vezetünk be
- üzleti élet bizonyos fogalmait szabványosítani kell, hogy mindenki tudja használni
- bevezethetünk elektronikus folyamatok is
- írjuk le a folyamatokat (technológiai, adminisztrációs, közigazgatási)
- BPEL (Business Process Execution Language)
    - le lehet írni üzleti folyamatokat
    - orchestration vezérlési mód
        - van egy process, ami leírja a folyamatot, tartozik hozzá egy végrehajtó gép, mely a process egyes lépéseit végrehajtja
        - egy kottában le van írva hogy hogyan kell müködni annak az egy processnek (a karmesternek)
    - choreograpgy vezérlési mód
        - ha elkezdödik egy folyamat (beérkezik egy megrendelés), azt továbbítjuk az elsö állomásra ahogy az ügyrend elöírja, az ügyrend szerinti tevékenységet ott végrehajtják
        - azt hogy hova kell tovább küldeni, azt az elsö állomás tudja, azt az elsö állomás küldi tovább
        - mindenki annyit tud hogy mi a dolga, és hogy hova kell tovább küldeni
        - nehéz utánanézni hogy hol tart az ügy (iktatás lehet még megoldás rá)



Az architektúrák egymásba ágyazhatóak, általában nem külön csak egy-egy modelt használunk!

Software Design Document

1.0 Introduction
2.0 Data design
3.0 Architectural and component-level design
4.0 User interface design
5.0 Restrictions, limitations, and constraints
6.0 Testing Issues
7.0 Appendices

1.0 Introduction

- This section provides an overview of the entire design document.
- 1.1 Goals and objectives
  - Overall goals and software objectives are described.
- 1.2 Statement of scope
  - A description of the software is presented. Major inputs, processing functionality, and outputs are described without regard to implementation detail.
- 1.3 Software context
  - The software is placed in a business or product line context.
- 1.4 Major constraints
  - Any business or product line constraints that will impact he manner in which the software is to be specified, designed, implemented or tested are noted here.

2.0 Data design

- A description of all data structures including internal, global, and temporary data structures.
- 2.1 Internal software data structure
  - Data structures that are passed among components the software are described.
- 2.2 Global data structure
  - Data structured that are available to major portions of the architecture are described.
- 2.3 Temporary data structure
  - Files created for interim use are described.
- 2.4 Database descriptio
  - Database(s) created as part of the application is(are) described.

3.0 Architectural and component-level design

- A description of the program architecture is presented.
- 3.1 Program Structure
  - A detailed description the program structure chosen for the application is presented.
  - 3.1.1 Architecture diagram
    - A pictorial representation of the architecture is presented.
  - 3.1.2 Alternatives
    - A discussion of other architectural styles considered is presented. Reasons for the selection of the style presented in Section 3.1.1 are provided.
- 3.2 Description for Component n
  - A detailed description of each software component contained within the architecture is presented. Section 3.2 is repeated for each of n components.
  - 3.2.1 Processing narrative (PSPEC) for component n
    - A processing narrative for component n is presented.
  - 3.2.2 Component n interface description.
    - A detailed description of the input and output interfaces for the component is presented.
  - 3.2.3 Component n processing detail
    - A detailed algorithmic description for each component is presented. Section 3.2.3 is repeated for each of n components.
    - 3.2.3.1 Interface description
    - 3.2.3.2 Algorithmic model
    - 3.2.3.3 Restrictions/limitations
    - 3.2.3.4 Local data structures
    - 3.2.3.5 Performance issues
    - 3.2.3.6 Design constraints

3.3 Software Interface Description

- The software's interface(s) to the outside world are described.
- 3.3.1 External machine interfaces
  - Interfaces to other machines (computers or devices) are described.
- 3.3.2 External system interfaces
  - Interfaces to other systems, products, or networks are described.
- 3.3.3 Human interface
  - An overview of any human interfaces to be designed for the software is presented. See Section 4.0 for additional detail.

4.0 User interface design

- 4.1 Description of the user interface
  - A detailed description of user interface including screen images or prototype is presented.
  - 4.1.1 Screen images
    - Representation of the interface form the user's point of view.
  - 4.1.2 Objects and actions
    - All screen objects and actions are identified.
- 4.2 Interface design rules
  - Conventions and standards
- 4.3 Components available
  - GUI components available for implementation are noted.
- 4.4 UIDS description
  - The user interface development system is described.

5.0 Restrictions, limitations, and constraints

- Special design issues which impact the design or implementation of the software are noted here.

6.0 Testing Issues

- 6.1 Classes of tests
  - The types of tests to be conducted are specified, including as much detail as is possible at this stage.
- 6.2 Expected software response
  - The expected results from testing are specified.
- 6.3 Performance bounds
  - Special performance requirements are specified.
- 6.4 Identification of critical components
  - Those components that are critical and demand particular attention during testing are identified.

7.0 Appendices

- Presents information that supplements the design specification.
- 7.1 Requirements traceability matrix
  - A matrix that traces stated components and data structures to software requirements is developed.
- 7.2 Packaging and installation issues
  - Special considerations for software packaging and installation are presented.
- 7.3 Design metrics to be used
  - A description of all design metrics to be used during the design activity is noted here.
- 7.4 Supplementary information (as required)

Rational Unified Process - RUP

Mire jó a RUP?
- kísérlet arra hogy próbáljunk meg egy olyan módszertant adni ami egybeépül az UML-el, olyan szempontból, hogy az egyes UML technikákat hogyan kell használni
- “hogy lesz projekt az UML projektekböl?”
- “milyen sorrendben kell az UML diagramokat elkészíteni?”

RUP jellemzöi:
- iteratív és inkrementális
- use-case vezérelt
- architektúra centrikus

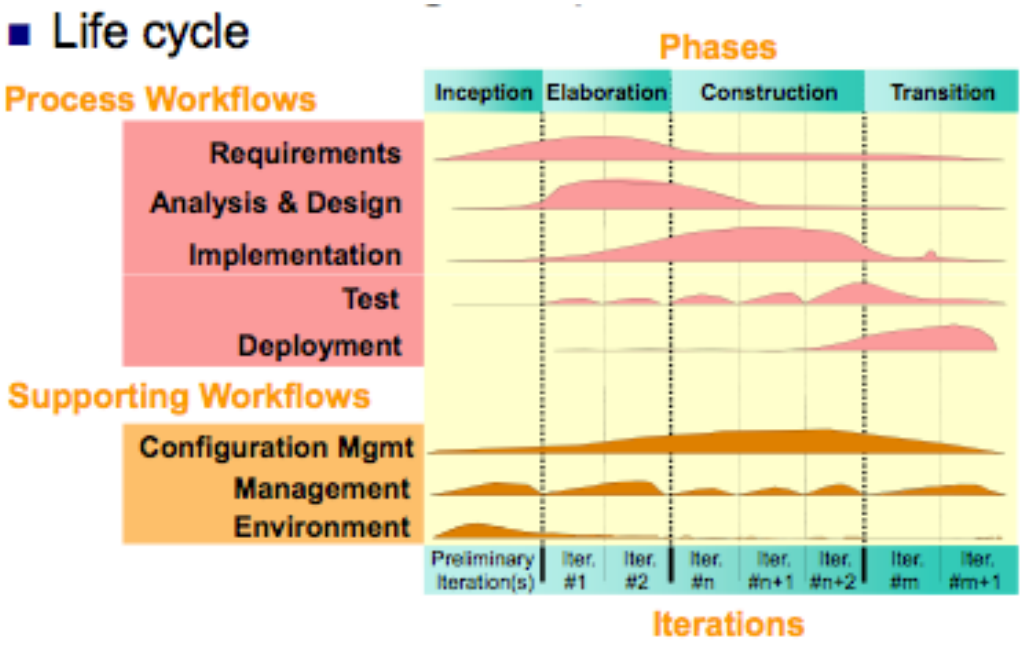
Iteratív és inkrementális:
- megóv attól, hogy hogy is kezdjük el
- korai visszacsatolást ad a usernek
- a csapat aki dolgozik ezek, azoknak van ideje megismerni az eszközöket amiket használni kell
- korai tapasztalatok a rendszer integrálásában és tesztelésében

Use-case vezérelt:
- use-case: koherens funkció halmaz, táblázatban
- vannak módszerek amelyek a use-case-ekböl becsléseket tudunk készíteni
- projekt ütemezéseket tudunk készíteni

Architektúra centrikus:
- az architektúra megválasztás jelentös ebben a módszertanban
- hamar kiderül hogy milyen architektúra köré épül
- klasszikusan rétegelt szerkezet
- alacsony csatolás az egyes komponensek között

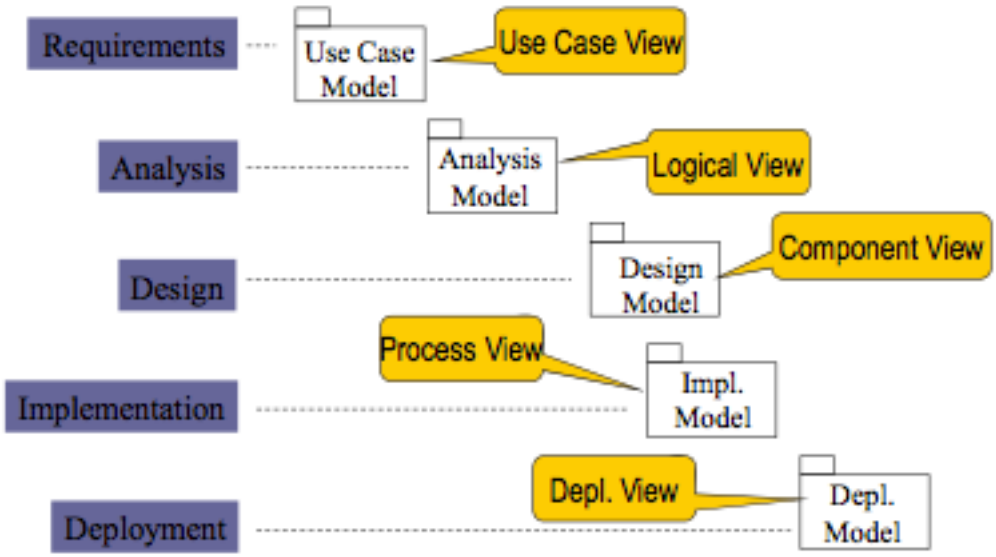
A process struktúrálása:
- idözítés
- milyen tevékenységek
- mit kell készíteni
- ki fogja csinálni

A RUP életciklusa:
- kezdeti fázis
    - probléma felfogása, feldolgozása
    - vége: legyen egy projekt vision (megvalósíthatóság)
- kidolgozási fázis
    - hagyományos analízis és tervezés fázis
    - a probléma feltárása és tervezés
- konstrukciós fázis
    - implementálás iteratív módon
- transition, átmeneti fázis
    - felkészíteni a szoftvert a kiadásra
    - felkészíteni az embereket a használatra

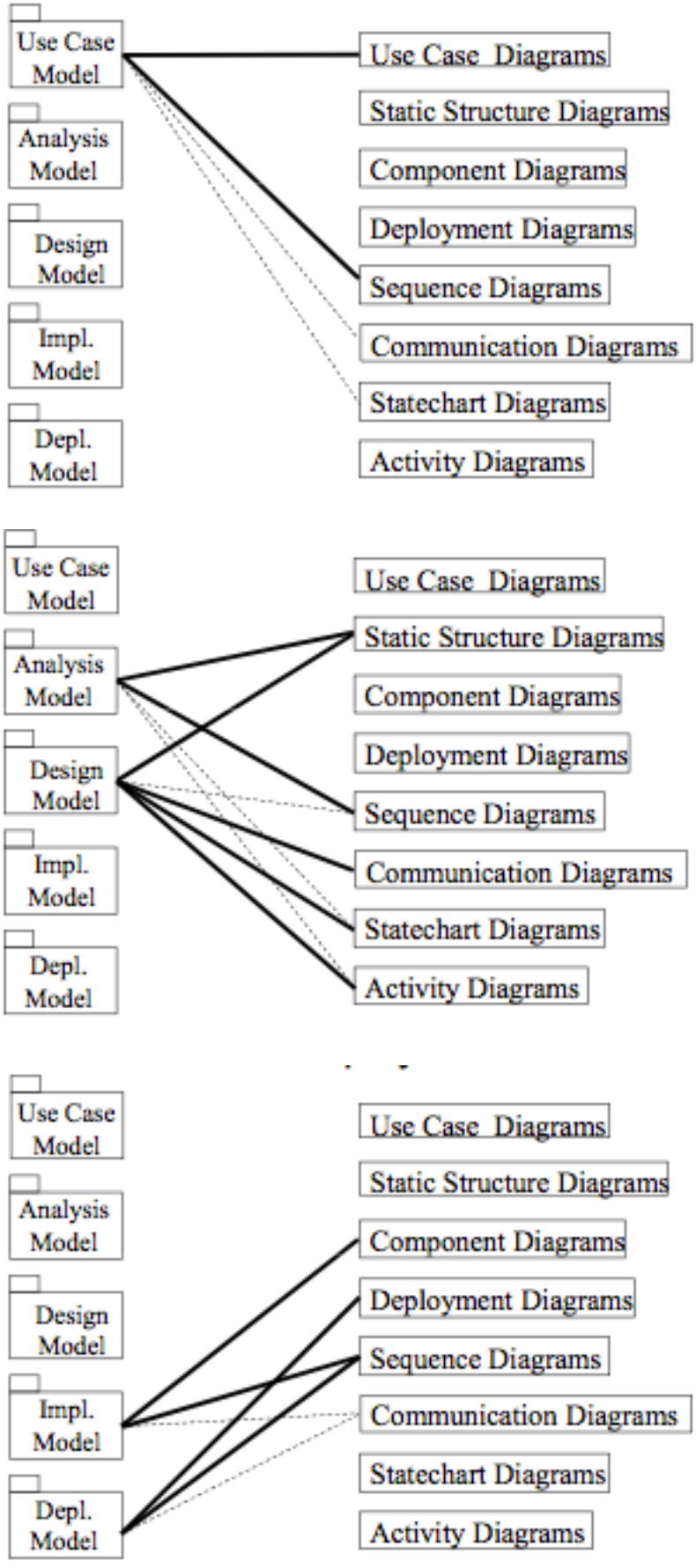


Ábra:
Függőlegesen életciklus fázisok, melyek további részekre vannak osztva (iterációk). A hagyományos folyamat lépései vízszintesen vannak, alatta pedig a RUP támogató workflow-ja. Az ábra azt mutatja meg, hogy a RUP életciklusai és a hagyományos workflow-k hogyan helyezkednek el. A dombok magassága az erőfeszítés mértékét jelöli.

Workflow-k és modellek:



Az egyes modellek milyen diagram technikákat használnak:
- szaggatott vonal: ritkábban elöfordul hogy kiegészítjük a modellt



RUP követelmény fázis:
- vázlattervet kell készíteni
- elözetes vizsgálati jelentést kell készíteni
- meg kell határozni a követelményeket
- rögzíteni kell a követelményeket szójegyzékbe
- implementálni kell a prototípust
- definiálni kell a use casce-eket (magas szintü és lényegeseket)
- vázlatos fogalmi modell (osztály diagram elözetes vázlata)
- vázlatos kép az architektúráról
- finomítani kell a terveket

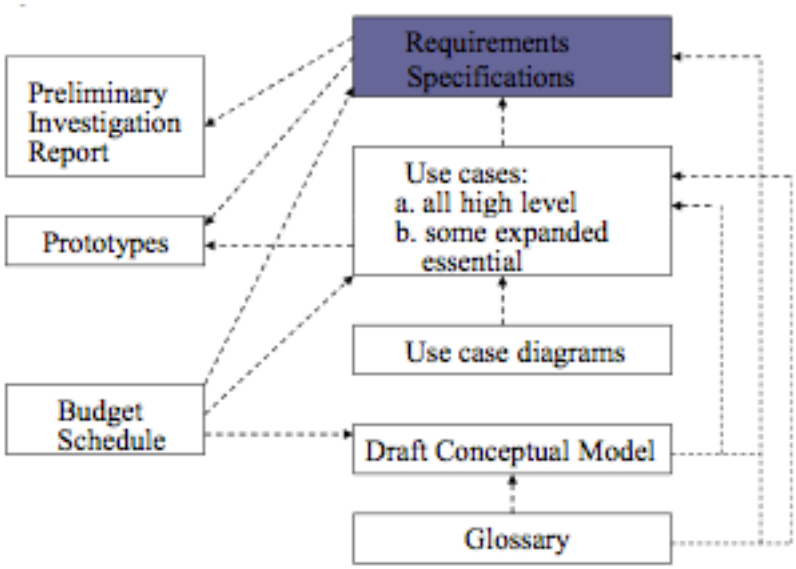
RUP analízis fázisa:
- lényeges use case-eket kell meghatározni
- use case diagram
- használható fogalmi model
- szójegyzés finomítása
- szekvencia diagramok
- szerzödések definiálása (elö- és utófeltételek)
- állapotdiagramokat kell az egyes szerkezeti elemekhez rendelni

RUP tervezés fázisa:
- valóságos use case-ek elkészítése
- nyomtatott riportok, felhasználói felületek, storyboard
- rendszerarchitektúra
- interakciós diagram
- class diagram
- adatbázis sémák

RUP implementációs fázisa:
- osztályok, interfészek definiálása
- metódusok megírása
- ablakok készítése
- riportok írása
- adatbázis sémák írása
- teszt programok

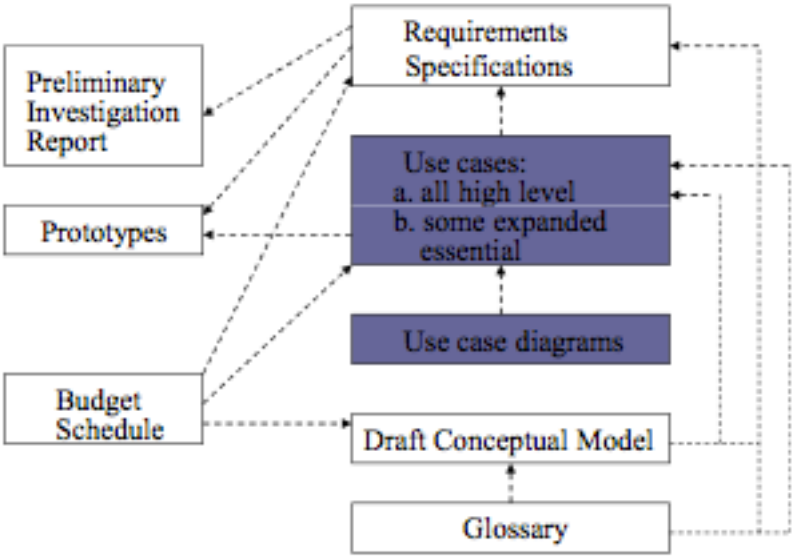
RUP deployment fázisa:
- definiálni kell azt hogy mik fognak párhuzamosan futni
- hogyan felelös a konkurenciáért
- véglegesíteni kell a technikai és user dokumentációkat
- rendszer teszt
- tréningek szervezése
- mögöttes támogatás szervezése (support)
- installálás

Követelmény temékek és függések



RUP tartalomjegyzék:

- áttekintés hogy kik a felhasználók
- mik a célok
- mik a funkciók
- rendszer attribútumok
- kockázatok
- glossary



Use case-eket hogy csinálunk?

1. actor bázisú: mik a szereplök, és ök mit csinálnak

2. esemény bázisú: milyen események történnek a rendszerünkkel

Use case formátumok 3 dimenzió szerint rendezhetöek:

- leírás részletezettsége:
    - magas szintü use case
    - kiterjesztett use case
- fogalmi vagy valóságos: a use case logikailag értelmezhetö vagy valóságosan
- prioritás alapján:
    - elsörendü (a rendszer ezek nélkül használhatatlan)
    - másodrendü (kis ideig ezek nélkül is megy a rendszer)
    - opcionális (jó ha van)

Use case készítés elött meg kell húzni a rendszer határait!

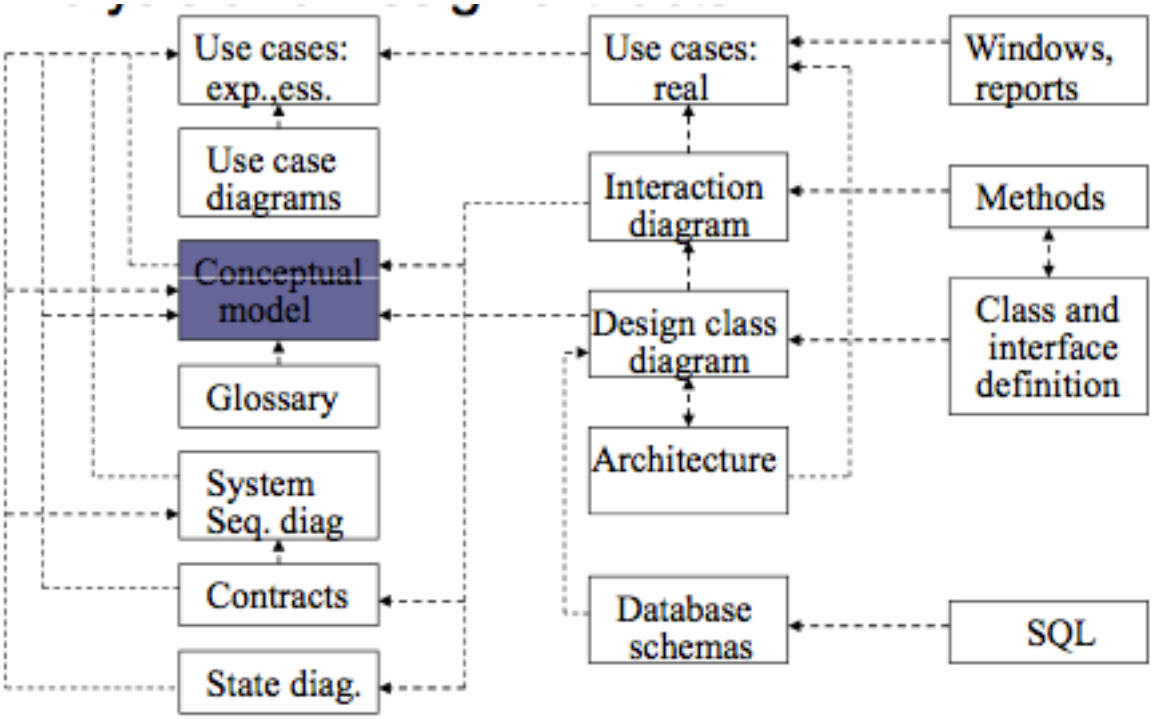
Hogyan csinálunk use case modelt?

- a követelményekböl meghúzzuk a határt, azonosítjuk az actorokat és az eseményeket
- írjuk meg mindegyik use case-t magas szintü formában, legyen egy áttekintésünk
- csináljuk use case diagramot
- keressünk kapcsolatokat a use case-ek között
- részletesen megírjuk a legkockázatosabb use case-eket
- rangsoroljuk a use case-eket

Hogyan rangsorolunk use case-eket?

- idöben kritikus
- bonyolultság funkciókat tartalmazó, kockázatosak
- alapvetö kutatást igénylöek
- alkalmazás szempontjából legfontosabbak
- üzleti élet szempontjából fontosak (bevétel növelése, kiadás csökkentése)

Analízis és tervesés termékek és függések



Fogalmi modell (conceptual model):
- alkalmazó tér szereplöit kell megkeresni
- komoly alkalmazói gyakorlat kell, szakértökkel kell egyezetetni
- osztály diagramokat készítünk
- tilos implementációs részleteket beletenni
- inkább túlspecifikált legyen, mint alulspecifikált
- fontos elv: térképész elv
    - a területen alkalmazott neveket használjuk, ne nevezzük át önkényesen
    - hagyjuk figyelmen kívül az értelmetlen részleteket
    - ne adjunk olyan dolgot hozzá ami nincs ott a valóságban

Hogyan csináljuk meg a fogalmi modellt?
- fogalmak összegyüjtése, kilistázása, kategorizálása (fönevek)
- tartsuk meg a helyes fogalmakat
- rajzoljuk meg
- ismerjük fel a közöttük lévö kapcsolatot (asszociáció)
- ki kell dobni a szükségtelen asszociációkat
- adjunk hozzá attribútumokat
- felismerjük az örökléseket
- szójegyzék (glossary)

Kiket veszünk fel az elemlistára?
- fizikai objektum
- secifikációk, tervek, dolgoknak a leírásai
- helyek, átmenetek, történések
- szerepek
- tartalmazási viszony, konténerek
- elvont fönévi fogalmak
- eljárás elvek
- katalógusok
- pénzügyi dokumentumok
- szerződések, jogi elemek
- pénzügyi instrumentumok
- szolgáltatások
- manual-ok, könyvek

Szükségtelen dolgok kihagyása:
- redundáns dolgok kihagyása
- értelmetlen fogalmak
- körülhatárolatlan fogalmak
- attribútumok
- implementációs konstrukciók

Asszociációk felismerése:
- az érintett elemeknek tudniuk kell egymásról
- fontosabbak az osztályok felismerése, mint az asszociációké
- óvatosan kell bánni az asszociációkat
- a kommunikációs diagramnál lesz visszacsatolás, hogy jók-e az asszociációk

Asszociáció lista, hol lehet asszoiáció:
- tartalmazás
- tagsági viszony
- birtoklás
- következö elem, szomszédság
- menedzselés, kommunikáció
- tranzakció
- ismeret alapján, ha tud valamit róla

Asszociációk kihagyása:
- redundáns, leszármazott asszociációk
- irreleváns, implementációs asszociációk

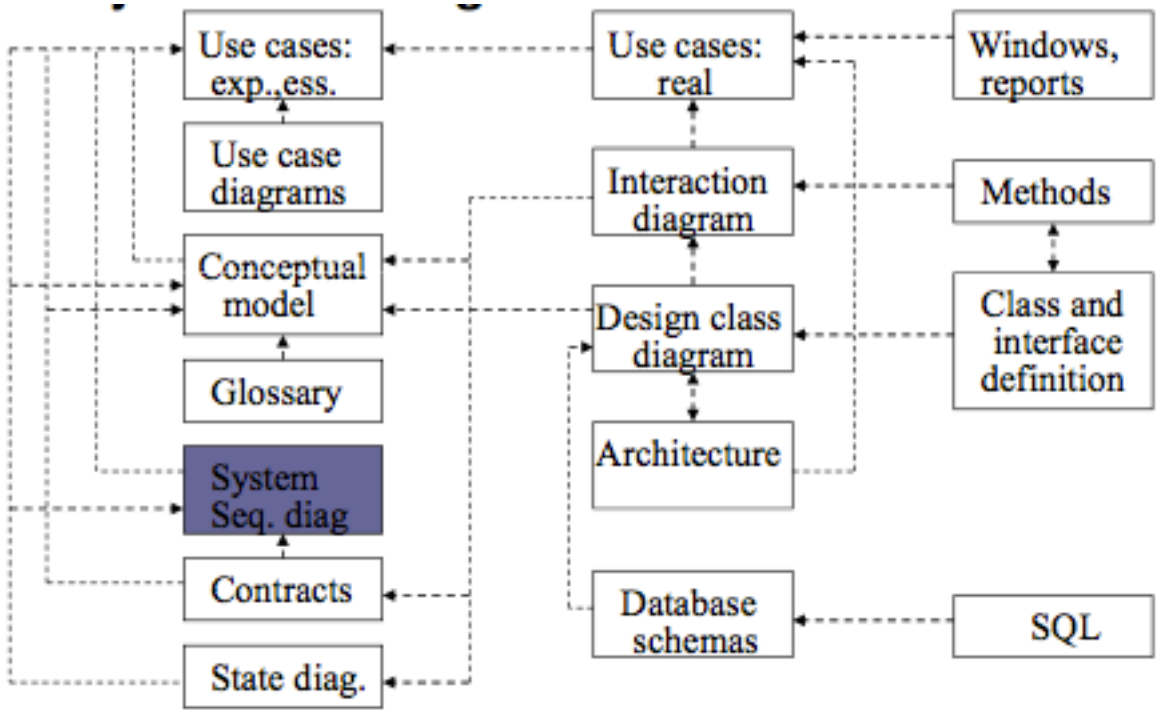
Attribútumok:
- tulajdonságokat fejeznek ki
- az attribútumoknak nincs önálló identitásuk, egyediségük (pl.: szín)
- tipikusan melléknevek
- vita esetén inkább mondjuk azt hogy osztály

Inkorrekt attribútumok elhagyása:
- származtatott attribútumok
- idegen kulcs (egy elemböl kihivatkozás egy másik helyre)

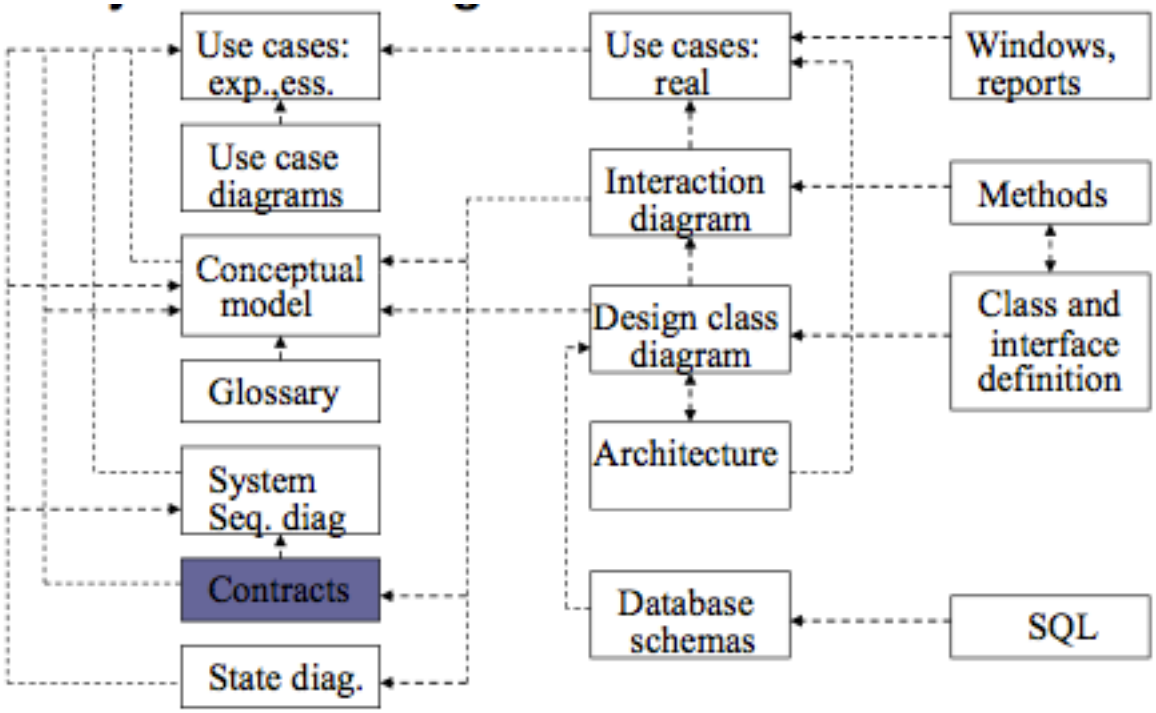
Típus megfeleltetés:
- ös és leszármaztatott típusok összehozása
- helyettesíthetöség egy alapvetöen fontos dolog

Asszociatív osztály létrehozása:
- ha egy asszociációnak vannak saját attribútumai

Delegáció és kompozíció:
- nem örökítünk, hanem dinamikusan hozzá tudok kapcsolni egy másik objektumot
- delegálás: felvállalok egy munkát és továbbadom egy kisebb valakinek

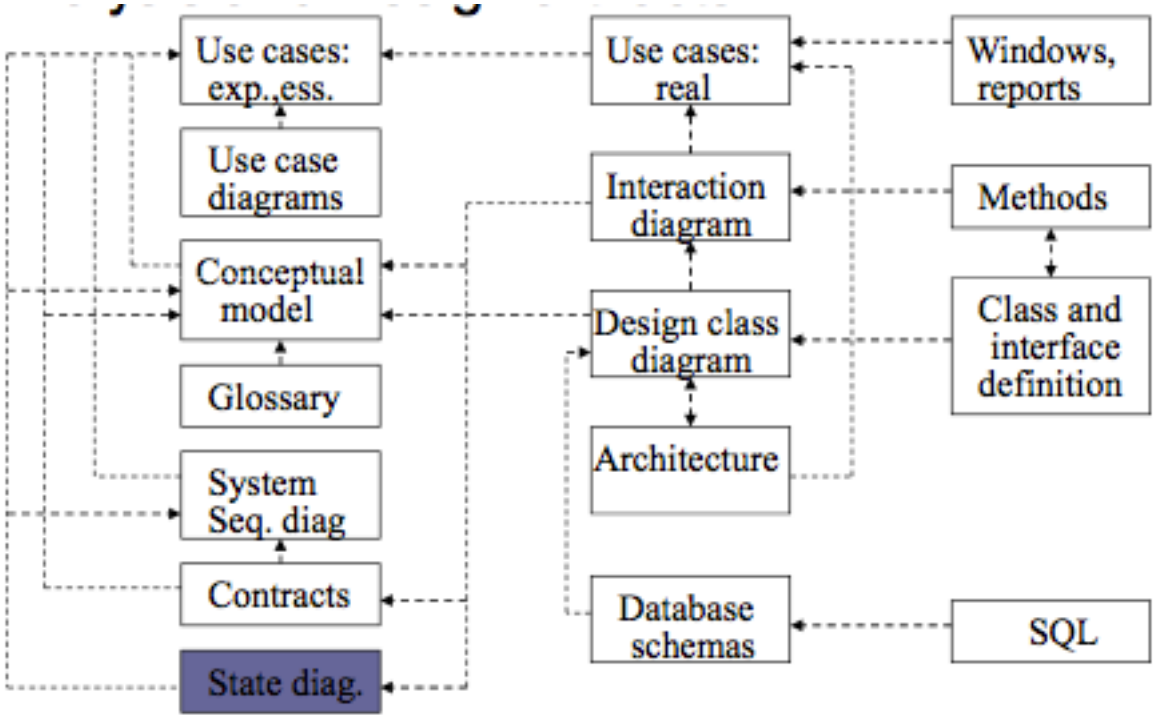


Szekvencia diagramok:
- egy diagram az egy use case-hez tartozó forgatókönyv
- egy use case-hez sok szekvencia diagram tartozik



Szerződések: (design by contracts elv)
- név
- felelösségek
- típusok, referenciák, exceptionok
- elö- és utófeltételek (pre- és postkondíciók)

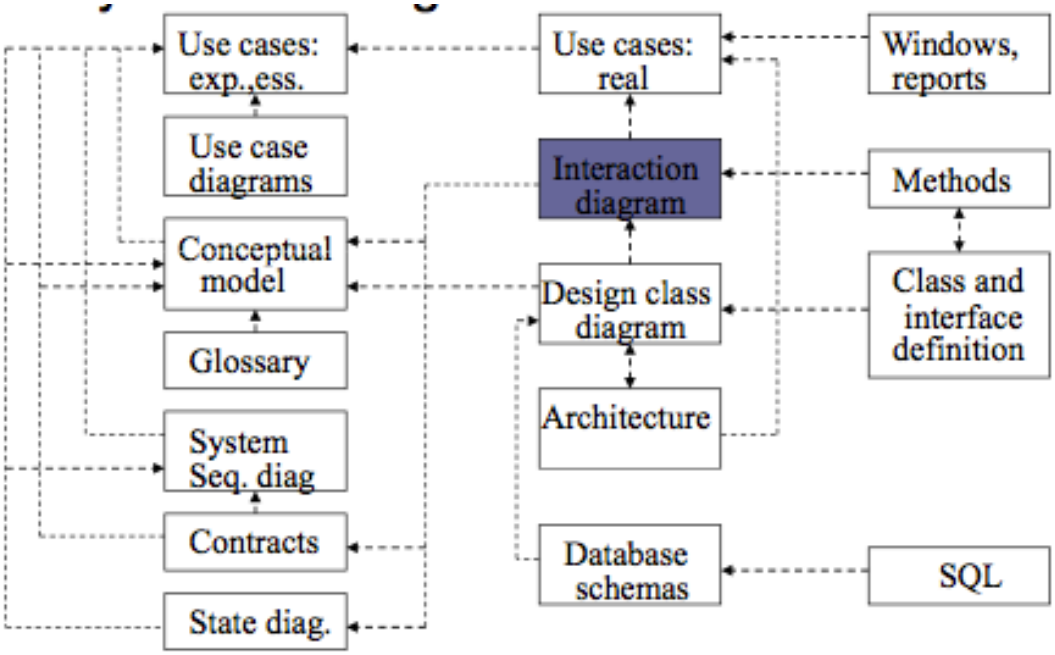
Szerződés készítése:
- kik között definiáljuk a szerződést
- felelösség definiálása
- elöször a postkondíciót (utófeltételt) határozzuk meg
- utána a prekondíciót (elöfeltételt) határozzuk meg



Állapot diagram:
- a viselkedés idöben változik, állapotfüggö
- külsö viselkedés

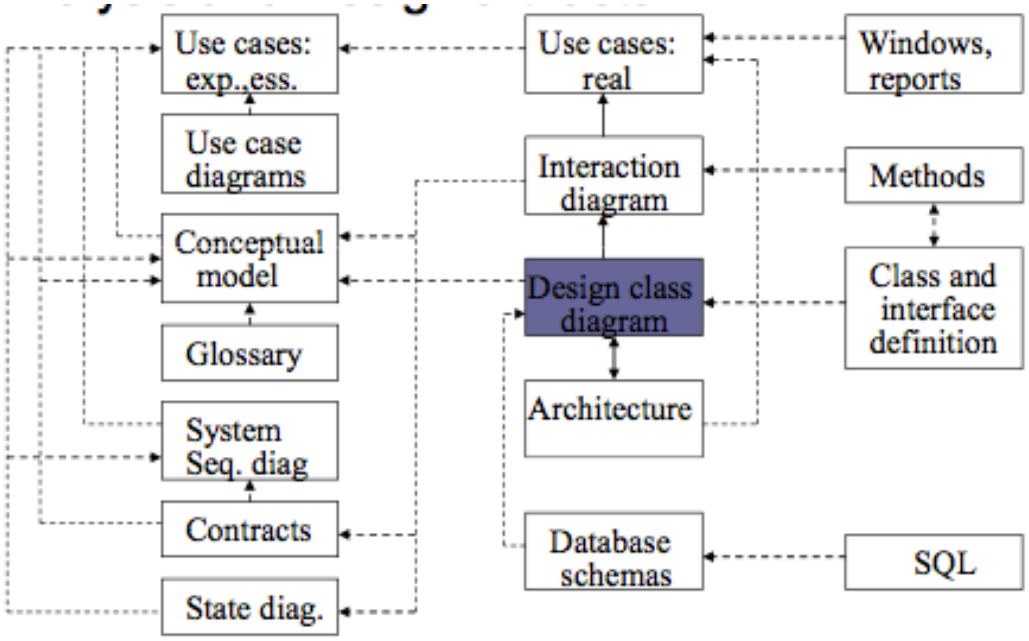
Mihez tartozhat állapot diagram?
- use case-hez
- osztályokhoz
- metódusokhoz

Mik állapotfüggöek?
- rendszer
- ablakok
- koordinátorok
- appletek
- eszközök
- mutátorok (olyan objektumok, melyek képesek megváltoztatni hogy melyik nyelvböl származnak)

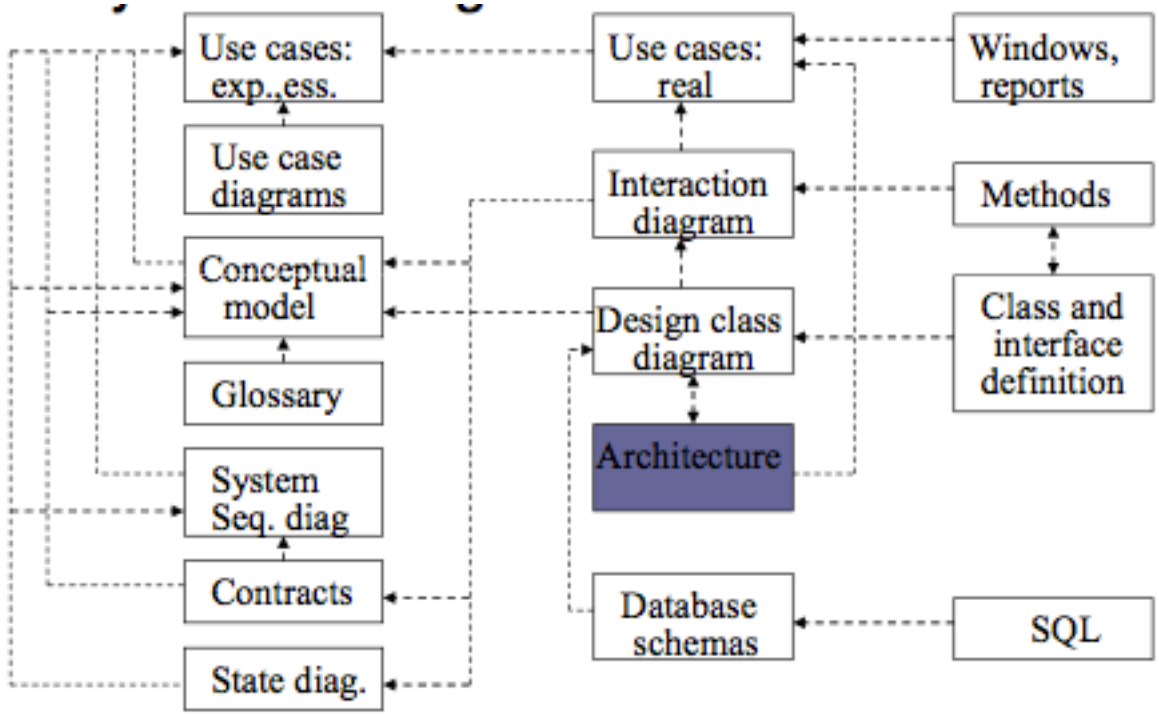


Interakciós diagram:
- szekvencia diagram, de inkább kommunikációs diagram

Interakció:
- felelösség: valamit kell tudni, kiröl mit tud

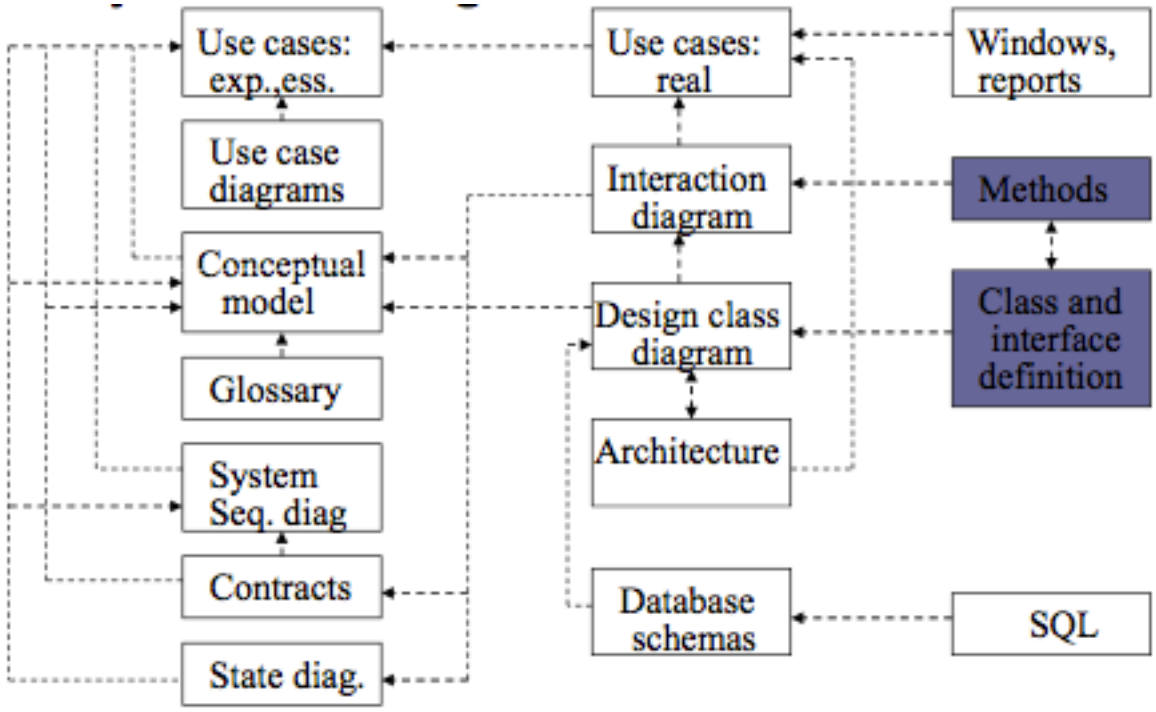


Design class diagram:
- részletekbe belemegy
- kigenerálható kódrészlet készíthetö belöle
- kik vesznek részt az adott megoldásba
- áthozni az analízis modell elemeit, metódusok típusát, navigálhatóságot, dependenciákat, mit hogy paraméterezünk



Architektúra:
- szét kell a rendszer rakni, package-ekbe és alrendszerekbe kell struktúrálni
- egy alrendszer komponensen belül szorosan csatolt elemek legyenek
- különbözö komponensek között laza csatolás legyen
- tervezési minták
    - hasznos dolgok, melyek a tervezési döntésekbe belejátszana

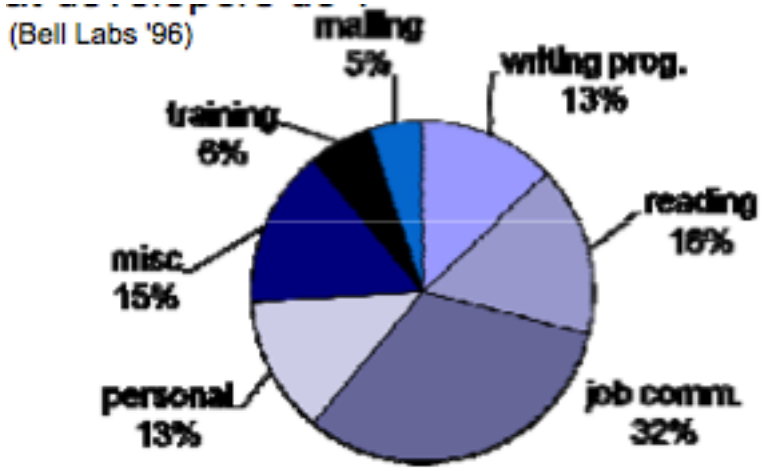
Milyen tervezési döntéseink vannak?
- csomagolás
- milyen konkurrenciák vannak, hogyan kezeljük öket
    - egyszálú rendszer
    - saját ütemezö, meglevö ütemezö
    - valóságos konkurencia
    - kölcsönös kizárás, kritikus szakasz
- package-ek allokálása processzorokra, közöttük milyen fizikai kapcsolatok lesznek
- storage és perzisztencia
    - az állapotra emlékezni kell
    - adatbázisokkal kötjük össze
    - adatbázis támogatás
    - a programunkat annotációval kell ellátni
- globális eröforrások kezelése
    - milyen örök vannak
    - lokkolás stratégiája
- eseményvezérelt rendszer legyen-e? (pl.: ablakozás szempontjából)
- ütemezés (preemptív, non-preemtív)

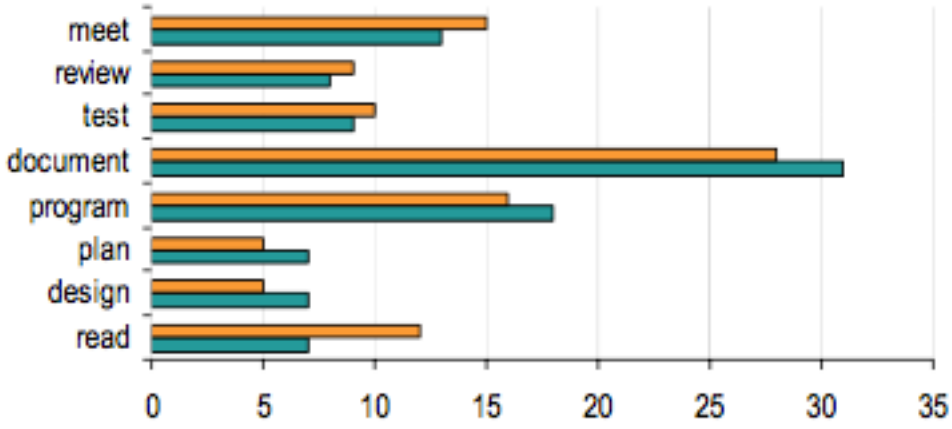


Metódusokat, interfészeket kell létrehozni.
Algoritmusok részleteit kell tisztázni.

Menedzsment

Mit csinálnak a szoftverfejlesztök?





Hogyan szervezzünk csoportokat?
- kevesebb, de jobb emberek
- testhezálló feladatokat adjunk az embereknek
- a csoportok harmonikusan illeszkedjenek
- késésben lévö projekthez ha ember csatlakozik, azzal csak nö a késés (Brooks törv.)
- magic seven: 7 ember legyen egy csoportban

Tervezni kell a szoftver projektet:
- kompromisszumokat kell keresni

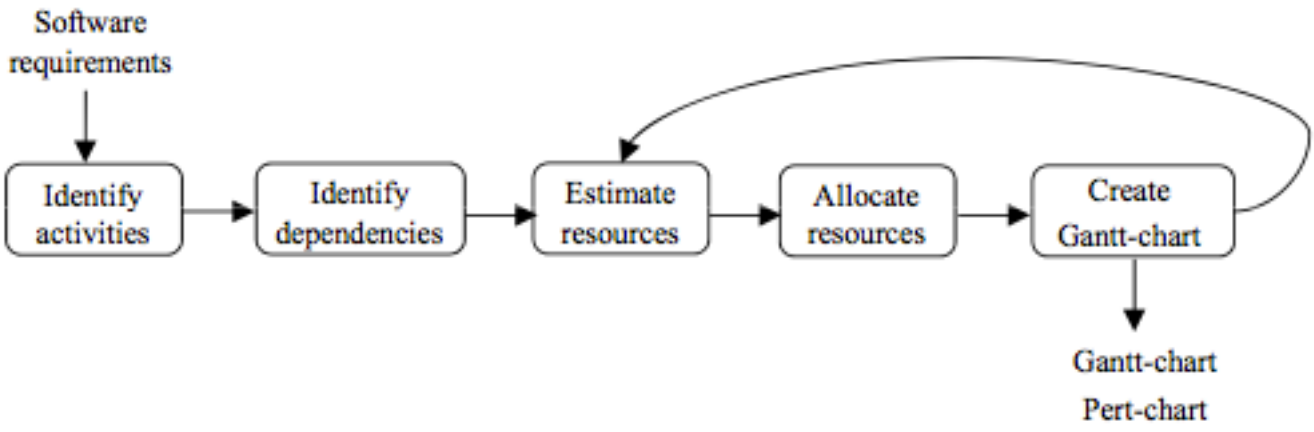
Projekt menedzsment feladata:
- felelösség vállalás
- eröforrás újraosztása
- ütemezés
- követelményeken lazítani
- oktatásra kell küldeni a munkatársakat

Projekt plan:
- long term: átfogja az egészet
- short term: akár napi projekt tervek

Eröforrások:
- emberek
    - legnagyobb költsége a szoftvernek
- hardware
    - ha a fejlesztögép és célgép elválik, akkor azt ki kell építeni
- szoftver
    - drága szoftver licenszek

Menedzselés:
- idöt: lehetöleg minden munkatársnak dolgoznia kell
- információt: melyik munkatárs milyen anyagot kapjon, feleslegesen ne terheljük
- szervezet
- minöség
- pénz

Ütemezés:
- elemi feladatokra bontjuk a nagyobb feladatokat (work breakdown structure)
- elemi feladatok közöti kapcsolatokat fel kell ismerni
- feladatok nagyságát és hozzárendelt eröforrásokat kell kezelni



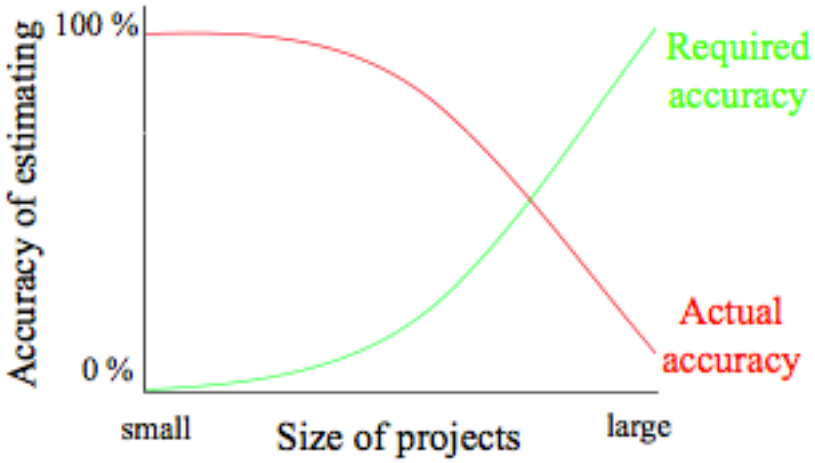
Kockázatok elemzése:
- kockázat = nem kívánatos esemény bekövetkezik
- példák:
    - itthagy a stáb
    - nem elérhetö a hardver
    - megbízó változtatja a követelményeket
    - a technológia kiszalad alólunk
- kockázat menedzsment lépései:
    - kockázatok azonosítása
    - kockázat elemzése
    - hogyan kerüljük el a kockázatot, mit tegyünk ha bekövetkezik
    - kockázat monitorozás
- kategóriák:
    - technológiai
    - stáb
    - szervezet
    - követelmények
    - becslések, ...

Kockázat elemzés:
- minden egyes kockázathoz két tényezöt rendelek
    - bekövetkezés esélye (1-5 ig például)
    - kockázat súlyossága (katasztrofális, komoly, elviselhetö, semleges)

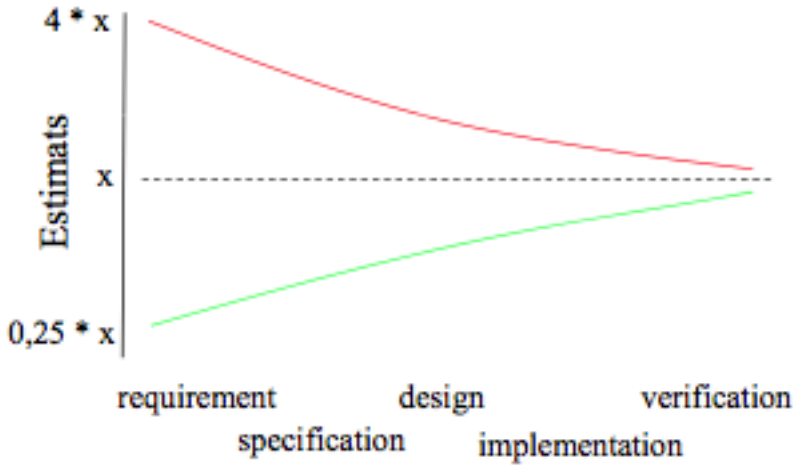
Kockázat tervezés: 3 stratégia
- elkerülés stratégiája
- minimalizálás: ha nem tudjuk megakadályozni a kockázat bekövetkezését, legalább csökkentsük a következményét
- folytatás: ha bekövetkezett a kockázat, akkor hogy tudunk felállni

Kockázat monitorozás:
- eddigi kockázatok újbóli áttekintése
- új kockázatok keresése

Becslés:
- ha a projekt mérete kicsi, akkor nagyon jól tudunk becsülni
- ahogy nö a projekt, úgy csökkent a pontossága a becslésnek, de az igény fordított

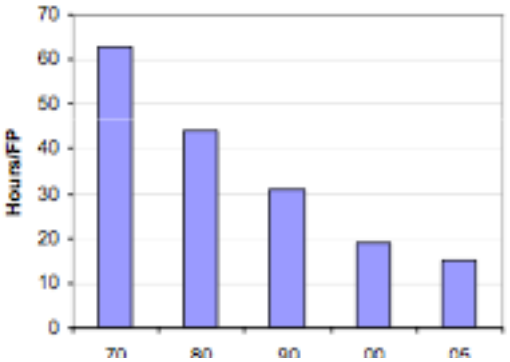


- a becslés nem hasraütés, indokolhatónak kell lenni
- a projekt végén nincs becslés, ott már tények vannak, ez az adat X, a projekt kezdetén a 0.25X - 4X becslés korrekt volt, ez a range csökkent a projekt elörehaladásával

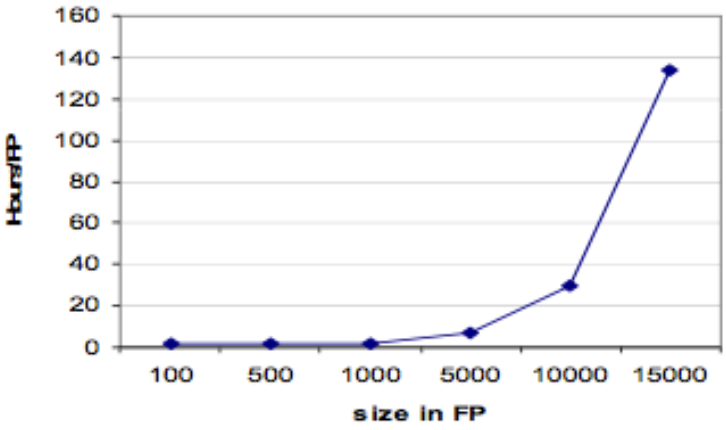


- ha már volt ilyen projektünk, akkor a meglevö dolgokból tudunk becsülni
- ha még nem volt ilyen projektünk, akkor a köztudatban lévö adatokból tudunk becsülni

Mérések:
- fejlesztéshez mekkora eröfeszítés szükséges (hány ember kell, mibe fog kerülni -> mennyi verejték kell hozzá)
    - összefügg a projekt méretével és bonyolultságával
- mértékek:
    - line of code (LOC)
    - funkció pont elemzés (FPA)
- komplexitás
    - hogyan lehet a bonyolultságot mérni?

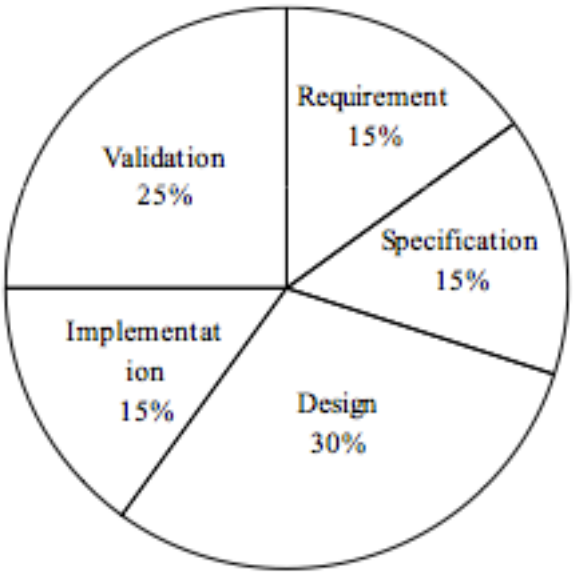


Ábra: hány óra volt egy funkciópontnyi program elkészítése

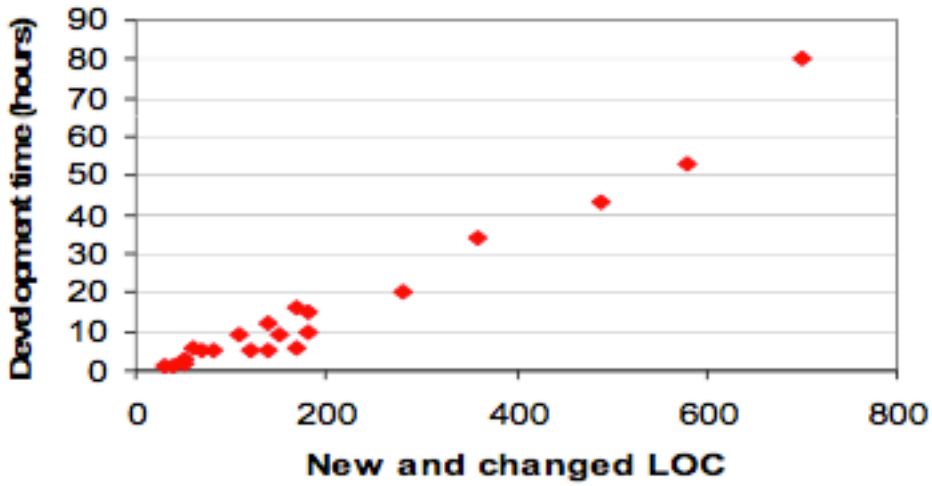


Ábra: program mérete funkciópontban / hány óra egy funkciópontnyi kódot megírni

Idöbeliség becslése:



Program mérete és a fejlesztési idö:



LOC és a funkciópont közötti összefüggés nyelvenként:

Language	*LOC*
Assembler	320
C	150
COBOL, FORTRAN	106
Pascal	91
Ada	71
Prolog	64
4 GL	40
APL	32
Smalltalk	21
Spreadsheet Languages	6

Projekt plan

- Requirement
  - System definition, Project plan, PFR
- Specification
  - Requirement specification, Preliminary user's manual, Preliminary Verification plan, SRR
- Architectural design
  - Architectural plan, PDR
- Detailed design
  - Detailed design plans, User's manual, Verification plan, CDR
- Implementation
  - Code reviews, Acceptance test plan, SCR, ATR
- Validation
  - All updated, Project evaluation, PRR, PPM

1.0 Introduction
2.0 Project Estimates
3.0 Risk Management
4.0 Project Schedule
5.0 Staff Organization
6.0 Tracking and Control Mechanisms
7.0 Appendix

1.0 Introduction

- provides an overview of the software engineering project.

1.1 Project scope

- A description of the software is presented. Major inputs, processing functionality and outputs

1.2 Major software functions

- A functional decomposition of the software

1.3 Performance/Behavior issues

- Non-functional requirements

1.4 Management and technical constraints

- Any special constraints that affect the manner in which the project will be conducted (e.g., limited resources or 'drop dead' delivery date) or the technical approach to development are noted here.

2.0 Project Estimates

- This section provides cost, effort and time estimates for the projects

2.1 Historical data used for estimates

- historical data that is relevant to the estimates presented

2.2 Estimation techniques applied and results

- A description of each estimation technique and results
- 2.2.1 Estimation technique m
  - Tables or equations associated with estimation technique m
- 2.2.2 Estimate for technique m
  - Estimate generated for technique m.

2.3 Reconciled Estimate

- The final cost, effort, time (duration) estimate

2.4 Project Resources

- People, hardware, software, tools, and other resources required

3.0 Risk Management

- This section discusses project risks and the approach to managing them

3.1 Project Risks

- Each project risk is described.

3.2 Risk Table

- The complete risk table is presented. Name of risk, probability, and impact are provided

3.3 Overview of Risk Mitigation, Monitoring, Management

- An overview of RM3 is provided here. The Complete RM3 is provided as a separate document or as a set of Risk Information Sheets.

4.0 Project Schedule

- This section presents an overview of project tasks and the output of a project scheduling tool.

4.1 Project task set

- The process model, framework activities and task set that have been selected for the project are presented in this section.

4.2 Functional decomposition

- A functional breakdown to be used for scheduling

4.3 Task network

- Project tasks and their dependencies are noted in this diagrammatic form.

4.4 Timeline chart

- A project timeline chart is presented. This may include a time line for the entire project or for each staff member.

5.0 Staff Organization

- The manner in which staff are organized and the mechanisms for reporting are noted

5.1 Team structure

- The team structure for the project is identified. Roles are defined

5.2 Management reporting and communication

- Mechanisms for progress reporting and inter/intra team communication are identified

6.0 Tracking and Control Mechanisms

- Techniques to be used for project tracking and control are identified.

6.1 Quality assurance and control

- An overview of SQA activities is provided. Note that an SQA Plan is developed for a moderate to large project and may be a separate document or included as an appendix.

6.2 Change management and control

- An overview of SCM activities is provided. Note that an SCM Plan is developed for a moderate to large project and may be a separate document or included as an appendix.

Agilis szoftverfejlesztés

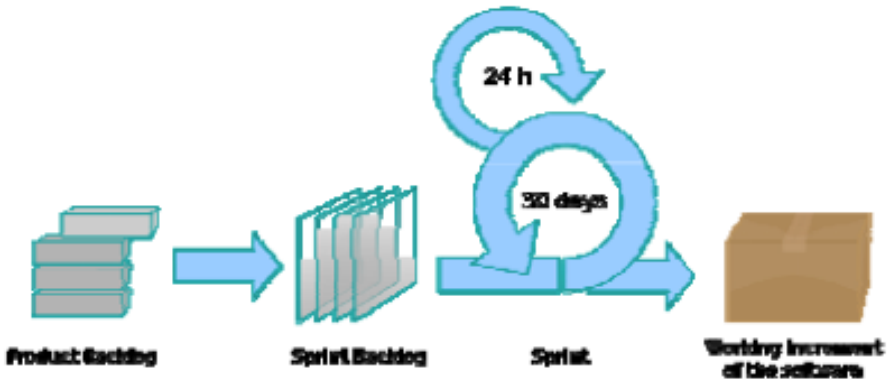
Kiáltvány melyet 2001-ben mindenféle szoftverfejlesztő guru fogadott el:
- az egyének és az együttműködés a folyamatok és az eszközök helyett
- a működő szoftver fölülmúl mindenféle dokumentációt is
    - elsösorban a szoftver a lényeg, hogy müködjön
- az ügyféllel való együttműködés a szerződéses megállapodás helyett
    - egyszerübb megállapodni valakivel, minthogy szerzödést írjunk
- ahelyett, hogy követknénk valamiféle tervet, folyamatosan reagálunk az igényekre
    - előzetesen halálkemény követelmények, tervek helyett majd alakul valahogy

Agilis módszertanok:
- scrum
- extreme programming

Ábra: Jól leírja az agilis programozást. Az agilitás egyik legfontosabb ismérve a borzasztó iterativitás. A RUP 3 alapelve, usecase vezérelt, iteratív inkrementális és architektúra centrikus. Ezek annyira iteratívak, hogy napi illetve napon belüli fejlesztési ütemeket definiálnak számunkra. Van napi vagy napon belüli iteráció is. Az iterációk szépen egymásba ágyazódnak és az agilitásnak ez a központi eleme.
Fontos dolog az átláthatóság.

Agilitás általános alapelvei:
- az ügyfél legyen elégedett attól, hogy nagyon gyorsan és számára használható szoftvert kap. Az ügyfél azonnal kap valamit, messze nem tökéletes, de valamivel már elkezdhet dolgozni
- teljesen értelmesnek és megszokottnak tartjuk azt, hogy a követelmények menet közben változnak, még a fejlesztés előrehaladott fázisában is. A fejlesztés nem egyéb, mint az új követelmények folyamatos kielégítése
- kulcskérdés, hogy működő szoftvert, rendszeres időközönként új változata jöjjön ki. Ez a rendszeres időközönként havi vagy ilyesmi. Az előrehaladás mértékét pont ez mondja meg, hogy jövünk-e új működő szoftverváltozatokkal
- fontos dolog, hogy fenntartható legyen a fejlesztés és lényegében kvázi hasonló vagy közel azonos inkrementumokkal fejlesszünk
- fontos, hogy a fejlesztői csapat működjön és időről időre tegyünk le valami új szoftvert. A szoftver is él, fejlődnie kell, ennek a fejlődésnek többé kevésbé egyenletesnek kell lenni
- a business szakemberekkel kell együtt lennünk
- a projekt motivált, elkötelezett emberekkel működik. Sokkal fontosabb, hogy jól meglegyünk, olyanokkal dolgozzunk, akikkel elmennénk inni, stb
- folyamatosan törekszik mindenki arra, hogy egyre okosabb legyen. Olyan csapat, ahol mindenki megpróbálja a munkát maga alá kaparni, fejlődni akar mindenki
- mindig a legegyszerűbb megoldásra törekszenek
- egy önszervező csoport van. Nincs vagy nem szükséges a főnök
- folyamatosan alakítják magukat a körülményekhez. Ha valaki becsöppen egy kis cégbe, 8-10 ember dolgozik együtt, van valami jó munkájuk, jó közösséget alkot

Scrum:
- egy framework, egy keretrendszer
- iteratív megközelítéssel, gyorsan értékes terméket állít elő
- ultrakönnyű, egyszerű megérteni, de meglehetősen bonyolult vezetni
- a 3 alappillér az, hogy minden áttekinthető, világos
- aaz átláthatóságnak alapja az, hogy közös nyelvet használ a business világ és a fejlesztői világ
- folyamatos felülvizsgálat van, ha eltérünk és nem tudjuk hozni azt amit kell, akkor a lehető legrövidebb időn belül felszámoljuk ezet az eltérést.



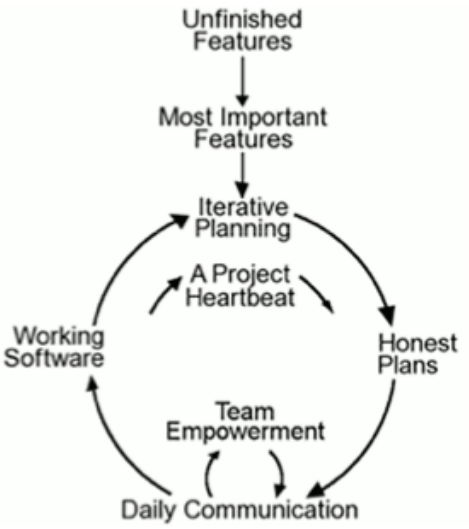
Árba: A legfontosabb lépések. Van a product backlog: elvárt termékjellemzők összessége. Tulképpen mindazon jellemzők, követelmények, usecasek formájába rögzítve, gyakorlatilag egy praktikus követelménylista. Ebből a listából majd megadott szereplők előállítanak sprint (következő tevékenység sorozatra specifikációkat). A központi elem a sprint, a futam, ami 2 és 1 hónap között szokott lenni. Kijön egy új szoftver. Lényegében havonta kijön egy új szoftver, egy új változat.

Sprint:
- a scrum központi eleme, egy scrum projekt a sprinteknek a sokasága, hossza 2-4 hét
- mindent csinálok: tervezés, tesztelés, integrálás
- sprint közben nem változtatunk
- lesz egy felügyelet, hogy ez egy értelmes cél legyen és ne egy szamárság
- a sprinthez tartoznak szerepek, a szerepek két csoportra osztható
- core szerepek/disznók (pigs): ők az életükkel fizenek a projektért
    - product owner
    - scrummaster
    - team
- nem elkötelezettek/csirkék (chickens): csak hozzájárulnak a projekt sikeréhez
- a sprintet meg kell tervezni, van egy áttekintés, van egy visszatekintés és minden nap van egy scrum meeting

Keretrendszer:
- szerepek
    - product owner
        - ö az ügyfél, a megbízó
        - dolga a product backlog, a doksi ami megfelel a követelményeknek
        - elfogadja, átveszi a munkát
    - scrum master
        - scrum értékekért és gyakorlatért felelös
        - ö dolga hogy a team tudjon dolgozi boldogan
        - adminisztrációt ö csinálja
    - team
        - felelösség, hogy idöben elkérszüljön a termék
        - 5-9 ember, önszervezö, önvezérlö csapat
        - mindenki mindent csinál
    - kiegészítö szerpek
        - ügyfélterjesztök
- események
    - sprint tervezés
        - a sprintet indító dokumentum a spring backlog
    - sprint áttekintés
        - felülvizsgálja azt hogy mit készítettünk, módosíthatjuk a követelményeket
        - owner megmondja mi készült el vagy mi nem
    - sprint visszatekintés
        - egy sprint bezárása és a következö között szokták csinálni
        - cél: a csapat fejlessze magát
    - napi scrum meeting
        - 15 perces, tényszerü, csak az core szerepek beszélnek
- termékek
    - product backlog
        - termék lista, benne van egy priorizált lista
        - mit kell tudnia a terméknek
        - finomítják (grooming)
    - sprint backlog
        - tartalmazza hogy mi mennyibe fog kerüli
        - fel vannak sorolva a taskok és az áruk
    - burnout chart
        - minden nap tudjuk hogy még hány óra van hátra a sprintböl

Scrum konklúzió:
- kis és egyszerű projekteknél nagyon jól használható
- jól működik, nem nagy gyári úthenger
- fontos, kirtikus szoftvereknél, illetve hatalmas projekteknél nem működik
- ha a team nem szilárd, akkor baj van
- a scrum masternek a dolga baromi nehéz
- az ügyfél hajlamos a kívánságait a végtelenségig növelni

Extreme programming (XP):
- egy últrakönnyű metódus OO szoftverfejlesztésre
- mitöl extrém? a gyakorlat van összekombinálva



- a központja egy iterációs ciklus, amibe tisztességes tervezés van, napi kommunikáció, a teamnek a megerősítésére, működő szoftvert kell produkálni és ezt csináljuk körbe
- fontos értékek: kommunikáció, bátorság, tisztelet, a tiszteletet csak pár éve teték oda
- előbbre helyezzük a kommunikációt, mint az írott dolgot
- minél gyorsabb, precízebb a visszacsatolás, annál kevésbé tévedünk el
- az egész szoftverfejlesztés nem lesz más, mint egy folyamatos újratervezés. bejön egy új követelmény, egy új igény, egy új indító event a körbejáráshoz és a kérdés, hogy mit kell nekünk refaktorálni a programon
- a változásnak örülünk, míg normális fejlesztésben megőrülünk tőle hamenet közben változás van
- sokat épít a páros munkára, a unit test perc/óra szinten van

XP project:
- user storiesból indulunk
- van az architectural spike, egy architektúrális kihívás, csúcs. Vannak olyan jellemzők, amiknek a megvalósítása nehéz és ezeknek csak prototipusokat tudunk csinálni. Csináljunk prototipusokat, amiben kipróbáljuk
- kiadatjuk a tervet. Itt jön egy iterációs hurok. Az elkészült programot teszteljük. A teszt forgatókönyvek jönnek a stori könyvből, elvétezzük a tesztet, a bugok miatt visszamegyünk iterálni. Ha az ügyfél is elfogadta, akkor lényegében túlvagyunk a releasen.

XP day:
- Standup meeting 9kor
- Párosodás: párok összeállnak
- A minőségbiztosítók tesztelnel. A teszt nem lesz jó, mert az új feladatot nem oljda meg a program, kódolunk, refaktoráljuk.
- Integrálunk. Mindig összerakjuk az egészet, nincsenek alternatív fejlesztési útvonalak.
- 5-kor elmegy a csapat. Ez fontos. Egy heti munkaidő 40 óra. Gyakori releasek vannak.

Tesztek:
- hamarabb elkészítjük a teszteket, mint a kódot. Ez teljesen normális, ott ahol megmondjuk előre hogy mit akarunk tesztelni
- aktív együttműűködést igényel. drágább, de egyszerűbb lesz a kód
- folyamatos integrációról is beszéltek. Az ügyfél ott van a munkahelyen. Különböző kódolási sztenderdeket alkalmaznak. NIncs minden megengedve. Nem lehet disznóságokat csinálni (egy adott programozási nyelvnél).
- A teamworkre fókuszál
- ellenjavallatok: Egy tipikus probléma itt is, hogy rosszul becsülünk. Nyilvánvalóan nagy programok esetén nem használható.

A scrum meg az xp nagyon hasonlítanak egymáshoz. Mind2 olyan hogy kis cégeknél kis, gyors egyszerű feladatok megoldásánál nagyon jól használható. Teljesen életszerű dolog lesz a mi életünkben is.

Java - Java alapjai

Java:
- szerkezetileg C-re hasonlít, szintaktikailag C++-ra
- hatalmas osztálykönvytár, olyan mint a legózás, nem kell láncolt listát, bináris fát csinálni csak tudnunk kell összerakni
- egyszerübb mint a C++ szintaktikailag
    - nincs: copy constructor, konstans függvény, virtual destructor, globális függvény
    - minden metótud vagy osztály metódus vagy osztály statikus metódusa
    - namespace helyett package
    - ugyanazok az operátorok, kivéve:
        - vannak logikai operátorok, értékük igaz vagy hamis, C++-ban ez int volt
        - nincs `delete`, `->` operátor (csak `.` operátort használunk)
        - `>>` (elöjel shiftel), `>>>` (0-át hoz be, úgy shiftel)
    - ugyanolyan vezérlö struktúrák (`for`, `while`, `switch`, `stb`...)
        - integert nem lehet ott használni ahol logikai kifejezést vár
        - lehet label-eket használni
            ```
            loop: while(!asleep()) { sheep++;} break loop;
            ```
    - nincs pointer, a java referencia olyan mint a C++-os pointer, de nincs pointer aritmetika
    - nincs: operator overload, `goto`
    - tömbök is objektumok, lekérdezhetö a mérete a tömbnek, nem lehet változtatni a méretét, ha egyszer lefoglalom akkor nem változtathatom a méretét
    - memória felszabadítást a Garbage collection végzi

Típusok:
- primitív típusok
    - új: `boolean`, `byte`
    - deklaráció ugyanaz maradt (`int a = 13;`)
    - közvetlenül elérjük, érték szerinti átadás
- komplex típus: osztály vagy annak példánya (objektum)
    - csak referenciánk van rá, csak a referenciát tudjuk átadni
    - ha a referenciát nem tároljuk el, akkor azt a garbage collecor összeszedi (`String s = "12345";  s = "hell";`), 12345-re mutató ref. elveszett
    - tömbök is komplex típusok
        - nem lehet túlcímezni, de 0 elemü tömböt lehet csinálni
        - `int a[] = new int[13];` vagy `char[] b = new char[20];`
        - többdimenziós: `int a[][] = new int [10][20];`
          pl: `int b[][] = new int[4][];`
          `for(int i=0; i<b.length; i++) b[i] = new int[i*2];`

`Hello.java`: (kiterjesztés mindig `.java`)

```
public class Hello {
    static public void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

Fordítás és futtatás parancssorban:

```
> javac Hello.java
> java Hello
```

Elég egyszer lefordítani, akkor egy bytecode készül ami mindnehol fut (OS X, Linux, stb), C++-nál a forráskód volt hordozható, mindnehol le kellett fordítani, a java mindenhol fut. Futtatható állomány hozhatunk létre ha `.jar` fájlba csomagoljuk össze.

Objektum, osztály, interface:

- láthatóságot osztály szinten kell definiálni és minden egyes metódus fejlécében is kell
  `(public, private, protected, package)`
  `package` - leszármazottak, illetve az ugyanabban a package-ben lévö osztályok
- nincsenek operátorok, csak a stingnél müködik az összeadás (+)
- objektumnak csak referenciáját adjuk át (nincs másoló konstruktor, inicializáló lista)
- nincs default paraméter, nincs többszörös és virtuális öröklés
- nincs destruktor
- `static`: osztály szintü metódusok és változók
  - a static metódust bármilyen metódusból elérhetjük, de ö már metódust nem ér el, nem tud az objektumokról
- `final`: a metódust a leszármazottak nem írhatják fölül, egyszer lehet neki értéket adni a konstuktorban, többször nem
- `abstract`: metódusok és osztályok elött állhat, ha metódus elött áll akkor nem szabad neki implementációs adni, ha osztály elött akkor legalább egy abstract metódusa van

```
public class Something {
  int a; // package visibility
  private double d;
  protected long l;
  public String s;

  public Something(int a) {
      this.a = a;
  }

  public Something() {
      this(10);
      l = 14l;
  }

  public void finalize() {
      ...
  }

  private int increment(int i) {
      a += i;
  }
  public long add(int i) {
      increment(i);
      l += i;
      return l;
  }
}
```

String osztály:

- szokásos metódusok, illetve + és += operátor
- `immutable`: módosíthatatlan
- `StringBuffer, StringBuilder`: mutable string reprezentáció
  - StringBuffer: gyorsabb, többszálú
  - StringBuilder: lassabb, egyszálú

Öröklés:

- `::`-ból `extends` lett
- ös konstruktorát a `super()`-rel hívhatjuk meg a `this` helyett, mindig elsö sorban kell lennie a konstruktorban a `super()`-nek
- minden metódus virtuális, nincs többszörös öröklés
- minden ösosztály ösosztálya az `Object` osztály

```
class A {
  int k;
  public A(int i) { k = i; }
  public void foo() { System.out.println("A"); }
  public void bar() { foo(); }
}

class B extends A {
  public B() { }
  public B(int j) { super(j); }
  public void foo() { System.out.println("B"); }
}
```

- == operátor helyett az Object `equals()` függvényét írjuk felül
- nincs `toString()`, helyette `clone()` - másoló konstruktornak felel meg, másolatot hoz létre

Interface:
- olyasmi mint az osztály csak a metódusoknak nincs implementiációja, lehet attrubútuma
- interfacekböl többet is meglehet valósítani

```
interface A {
  void foo();
  int bar(String s);
}
abstract class B implements A {
  ...
  public void foo() { System.out.println("B"); }
  abstract public int bar(String s);
}
class C implements A {
  ...
  public void foo() { System.out.println("B"); }
  public int bar(String s) { return s.length();}
}
```

Wrapper osztály: (olyan osztály amely primitív típust csomagol be)
- heterogén kollekcióban primitív típust nem tudunk belerakni, arra ez megoldás
- minden primitív típushoz létezik wrapper osztály, plusz dolgokat is tudunk hozzá rakni
- boxing: macerás lenne mindig case-el, new-al létrehozni, helyette létezik

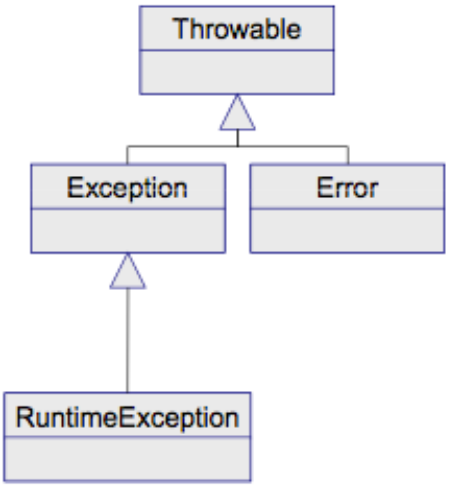
```
int a = 2; Integer b = 3; a = a + b;
Integer d = 3+3; // int -> Integer
System.out.println(d*3); // Integer -> int
```

```
public static void main(String args[]) {
      Integer i1 = Integer.parseInt(args[0]);// boxing
      Integer i2 = Integer.parseInt(args[0]);// boxing

      i1.equals(i2);       // true, no boxing
      i1 == i2;            // only if -128<i1≤127
      i1 <= i2;            // true, boxing
      i1++;                // incr, boxing
      i1 > 120;            // boxing
}
```

Kivétel kezelés:
- ha eldobunk egy metódust vagy exception-t akkor azt a metódus neve után fel kell sorolni a nevét vagy a típusát
- java-ban throws jelzi hogy mit dobhatunk, a throw pedig explicit dobja a kivételt
- catch ága után egy finally-t berakhatunk, ez akkor is lefut, hogyha bármelyik exceptiont elkaphatjuk

```
void foo(String s) throws IOException {
        …
}
```

```
FileInputStream fis = new FileInputStream("a");
try {
        ...
        foo(s);
        ...
} catch (IOException e) {
        ...
        throw e;
} finally {
        fis.close();
}
```



`Error`: rendszer error, nem lehet lekezelni, nem tudunk vele mit kezdeni (pl: hardver hiba), a rendszer le fog állni, de azért van hogy lássuk mi történt
`RuntimeException`: nem kötelező elkapni, ha pl. túlcímezzük a tömböt

Kódolási stílus:
- változók: camelCase `(getMaxValue(), int initValue)`
- osztályok: CamelCase `(StringBuffer)`
- package: kisbetü `(java.util)`
- zárójel:

```
while (true) {
    if (a<b) {
        ...
    } else {
        ...
    }
}
```

Java - Java input/output

System osztály:
- 3 statikus attribútum: `in, out, err` - standard input/output/error
  pl: `system.out.println(...);`
- `gc()` - garbage collector, mely kézzel hívható
- exit(int) - kilép a Java virtuális gépből (JVM)
- getProperty/getProperties, setProperty: hozzáférhetünt a rendszerdolgokhoz

Input/output:
- `java.io` package
- alacsony szinten karaktereket és bájtokat tudunk kiírni és beolvasni, van egy filter mechanizmus, tudunk szüröket definiálni, ezek becsomagolják ezeket és be tudunk olvasni stringeket és egyéb dolgokat
- saját típusokat is bájttá tudjuk alakítani anélkül hogy plusz kódot kéne írnunk
- `Reader` osztály - karakter beolvasáshoz
- `Writer` osztály - karakter kiíáshoz
- `InputStream` osztály - bájtokat beolvasása
- `OutputStream` osztály - bájtok kiírása

Reader osztály metódusai:
- `read()`, `read(char[] buf, int off, int len)` - beolvas *len* darab karaktert
- `mark(int limit), reset(), markSupported()` - könyvjelzö(mark), ugrás(reset)
- `skip(long n)` - markkal ellentétben ez elöre ugrik
- `close()` - reader bezárása
- `ready()` - meg tudjuk nézni hogy olvashatunk-e a fájlból (pl. hálózati kommunikációhoz)

Writer osztály metódusai:
- `write(int c)` - egy karakter kiírása
- `write(char[] bug, int off, int len)` - egy tömbnyi karakter kiírása
- `write(String s, int off, int len)` - string kiírása
- `flush()` - a pufferböl küldje ki azt ami összegyült
- `close()`
- `Writer append(...)` - fájl végére füzünk dolgokat és nem írja felül

Speciális Reader osztályok:
- `BufferedReader` - az összegyüjtött karaktereket megkapjuk
- `CharArrayReader / StringReader` - ha egy meglévö stringböl akarjuk olvasni a karaktereket, akkor ezt használhatjuk
- `FilterReader` - a leszármazottai összetettebb karaktereket tudnak olvasni
- `FileReader` - fájlból karakterek olvasására használhatjuk

```
FileReader fr = new FileReader("hello.txt");
BufferedReader br = new BufferedReader(fr);
while (true) {
      String line = br.readLine();
      if (line == null) break;
      System.out.println(line);
}
br.close();
```

Speciális Writer oszyályok:
-`BufferedWriter` - az összegyüjtött karaktereket írhatjuk ki
-`CharArrayWriter / StringWriter` - charArray-be vagy String-be ír
-`FilterWriter` - a leszármazottai összetettebb karaktereket tudnak írni
-`FileWriter` - fájlba karakterek írására használjuk
-`PrintWriter` - print, printf, println formátumban történö kiírásra alkalmas

```
FileWriter fw = new FileWriter("squares.txt");
PrintWriter pw = new PrintWriter(fw);
//PrintWriter pw =
// new PrintWriter("squares.txt", "ISO-8859-1");
for (int i = 1; i <= 10; i++) {
      pw.println(i+"*"+i+" = "+(i*i));
      //pw.printf("%d*%d = %d\n", i, i, i*i);
}
pw.close()
```

InputStream osztály metódusai:
-`read(), read(byte[] buf, int off, int len)` - beolvas *len* darab bájtot
-`mark(int limit), reset(), markSupported()` - könyvjelzö, visszaugrás oda
-`skip(long n)` - n darab bájt átugrása
-`close()` - fájl bezárása
-`ready()` - írható-e a fájl
-`available()` - hány bájt olvasható be blokkolás nélkül

OutputStream osztály metódusai:
-`write(int c)` - egy int kiírása
-`write(byte[] buf, int off, int len)` - egy tömbnyi bájt kiírása
-`flush()` - pufferböl kiküldi azt ami ott összegyült
-`close()`

Speciális InputStream osztályok:
-`ByteArrayInputStream` - bájtok olvasása bájt tömbböl
-`FileInputStream` - bájtok olvasása fájlból
-`FilterInputStream` - összetettebb dolgok olvasására, absztrakt

Speciális OutputStream osztályok:
-`ByteArrayOutputStream` - bájtok írása bájt tömbbe
-`FileOutputStream` - bájtok írása fájlba
-`FilterOutputStream` - összetettebb dolgok írására, absztarkt

Standard IO:
-`java.lang.System`: alap osztály sok statikus taggal
-`gc()`: garbage collector hívás
-`InputStream in`
    -standard input, byte alapú
-`PrintWriter out, err`
    -standard out, err, char alapú
-asszimetrikus input/output

Stdin-röl olvasás, stdout-ra írás:
- egy input stringböl tud bájt alapú karaktereket elöállítani, van egy InputStreamReader és egy BufferedReader

```
InputStreamReader isr =
      new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
while (true) {
      String line = br.readLine();
      if (line == null) break;
      System.out.println(line);
}
br.close();
```

Java IO filterek:
- egy input stringböl tud bájt alapú karaktereket elöállítani, van egy InputStreamReader és az van a BufferedReader-en belül
- a legoknak közös interfészeit össze tudjuk illeszteni, ezeket összeilesztve egy delegációs mechanizmust fogunk látni
- ezt az összefüzést úgy szoktuk megjeleníteni, hogy konstruktorokat füzünk össze
- char és byte konverzió
    - olvasás
        - forrás: InputStream
        - kliens elvárása: Reader
        - megoldás InputStreamReader
    - írás
        - OutputStreamWriter

Tömörítés:

```
BufferedReader br = new BufferedReader(
      new InputStreamReader(new GZIPInputStream(
            new FileInputStream("test.gz"))));
while (true) {
      String line = br.readLine();
      if (line != null) System.out.println(line);
      else break;
}
br.close()
```

```
BufferedReader br = ...;
PrintWriter pw = new PrintWriter(
   new OutputStreamWriter(
      new GZIPOutputStream(
         new FileOutputStream("test.gz"))));
while (true) {
      String line = br.readLine();
      if (line != null) pw.println(line);
      else break;
}
br.close(); pw.close();
```

Saját filter:
- FilterInputStream, FilterOutputStream, FilterReader, FilterWriter
- ugyanaz az intefésze mint az ösosztályának

```
public class MyInFilter
extends java.io.FilterInputStream {
      protected MyInFilter(InputStream arg0) {
            super(arg0);
      }
      ...
}
```

- kapunk paraméterként egy InputStream-em, ezt az ösosztálynak átadjuk, így tudunk saját filtert csinálni

Speciális IO osztályok:
- Olyan szövegfeldolgozást szeretnénk csinálni, hogy beolvassuk a szöveget és ezt kiíratjuk valakivel, kapunk egy bemenetet és valamit kiadunk a kimenetnek. Ez elegánsabb mint várni, hogy meghívja valaki és arra csináljunk valamit.
- Csináljuk olyan programot, ami valami readerböl olvas, megnézzük mi van benne és ha megfelelö mintára illeszkedik (pl. tartalmazza az alma szót), akkor kiírjuk. Lesz egy reader amiböl olvasunk, lesz egy writer amibe írunk és megkapjuk a mintát.

```
public class Grep {
      Reader in;
      Writer out;
      String pattern;

      public Grep(Reader in, Writer out, String pat) {
            this.in = in;
            this.out = out;
            this.pattern = pat;
      }
```

```
      public void run() {
            BufferedReader br = new BufferedReader(in);
            PrintWriter pw = new PrintWriter(out);
            while (true) {
                  String line = br.readLine();
                  if (line != null) {
                        if (line.matches(pattern)) {
                              pw.println(line);
                        }
                  } else break;
            }
            pw.close();
      }
```

```
      static public Reader
      grep(Reader r, String pattern) {
            PipedWriter pw = new PipedWriter();
            PipedReader pr = new PipedReader(pw);
            Grep grep = new Grep(r, pw, "hello");
            // starting grep in the background
            return pr;
      }
}
```

File:

- Szeretnénk tudni hogy milyen fájljaink vannak, mit tudunk csinálni velük.
- FILE osztály: File-nak a reprezentációja, megmondja mi a neve, milyen hosszú
- Konstuktorokkal tudom létrehozni, megadhatom a fájl nevét, elérési útját. Ekkor létrejön egy fájl objektum, meg tudjuk töle kérdezni, hogy a konstruktorban megadott stringez tartozik-e fájl.
- Ha megadunk egy másik fájl objektumot, könyvtárat és ebből keressük az adott nevü fájl.

```
void lsdir(File f, String tab) {

   File[] list = f.listFiles();

   for (int i = 0; i < list.length; i++) {
      System.out.println(tab+list[i].getName());

      if (list[i].isDirectory()) {
         lsdir(list[i], tab+" ");
      }

   }
}
```

Szerializáció:

```
import java.io.Serializable;
public class SerializableClass
                  implements Serializable
{ ... }
```

```
import java.io.*;
try {
      FileOutputStream f =
                  new FileOutputStream("filename");
      ObjectOutputStream out =
                  new ObjectOutputStream(f);
      out.writeObject(_SerializableClass);
      out.close();
}
catch(IOException ex) { ... }
```

```
import java.io.*;
try {
      FileInputStream f =
                  new FileInputStream("filename");
      ObjectInputStream in =
                  new ObjectInputStream(f);
      o_ins = (SerializableClass)in.readObject();

      in.close();

} catch(IOException ex) {
      ...
} catch(ClassNotFoundException ex) {
      ...
}
```

Szerializáció szabályai:

- szerializálhatónak kell lenni az osztálnak (implements Serializable)
- vannak nem szerializálható osztályok, pl. socket
- ami szerializálódik: attribútumok, primitív attribútumok és azok amik nem primitívek feltéve, hogy az adott tipus szerializálható
- ha egy mezöt szeretnénk nem szerializáhatóvá tenni, akkor az `transient`

Java - generic, collection, utility, thread

Ha az **Object** ösosztályt használjuk örökléshez, akkor kasztolásra kell figyelnünk.

```
public class Store {
  Object[] os; int size;
  public Store(int i) {
    os = new Object[i];
    size = 0;
  }
  public void put(Object o) {
    os[size] = o; size++;
  }
  public Object get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

```
Store s = new Store(10);

s.put("hello ");
s.put("world");
s.put("!");

for (int i = 0;
     i < s.size();
     i++) {

  String l = (String)s.get(i);
  System.out.print(l);

}
```

C++-s **template** benne van az 5-ös javatól. Amikor Java templateket használunk, akkor nem generálódik minden egyes templatehez külön osztály. Javaban egy darab osztaly generálódik egy generikus osztályhoz. Template kulcsszót nem használjuk. Egy helyen nem használhatjuk a T típust, nem hívhatunk konstruktort a T típuson, nem lehet olyan hogy new T(). T típusú tömböt létre tudunk hozni.

```
public class Store<T> {
  T[] os; int size;
  public Store(int i) {
    os = new T[i];
    size = 0;
  }
  public void put(T o) {
    os[size] = o; size++;
  }
  public T get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

```
Store<String> s =
       new Store<String>(10);

s.put("hello ");
s.put("world");
s.put("!");

for (int i = 0;
     i < s.size();
     i++) {

  String l = s.get(i);
  System.out.print(l);

}
```

Generics and inheritance:

```
Store<String> sf = new Store<String>(10);
Store<Object> of = sf;
of.put(new Integer(13));
```

- of Objecteket tárol, értékül adhatom neki a Stringeket tartalmazó store-t, majd Integert próbálok beletenni, itt gond lesz
- az of dinamikus típusa a Store<String>-nek és nem fognak szeretni ha valami mást akarok beletenni, a Store<Object> nem ösosztálya a Store<String>-nek, bár az object öse a stringnek, ahol én object-et várok ott nem adhatok paraméterül Stringet

```
Object[] oa = new String[10];
oa[0] = "Hello";
oa[1] = new Integer(2);
      //ArrayStoreException
```

- ez igaz lesz tömbökre is, ha van egy Object tömböm, értékül adhatok stringeket és elvileg integert is, ezt a fordító megeszi, szeretni fogja de futtatáskor ArrayStoreException-t fog dobni, generikus típusok esetén már fordítási idöben se fut

Wildcard:
- bevezetek egy Joker karaktert (?)

```
void printStore(Store<?> of) {
 for (int i = 0; i < of.size(); i++) {
    System.out.println(of.get(i));
  }
}
```

```
Store<String> sf = new Store<String>();
Store<?> of = sf;
of.put(new Integer(13));  // CT error
of.put("Hello");          // CT error
```

- a <?> tekinthetö bárminek, akár az <Object>, ki tudok venni az ilyen store-ból viszont be nem tudok rakni

```
void putAll(Store<E> s) {
  for(int i = 0; i < s.size(); i++) {
      put(s.get(i));
}
```

- szeretnék egy metódust implementálni, ami paraméterül kap egy Store-t, akármilyen store-ban lévö paramétereket lepakolja, látszik hogy ez csak akkor müködhet, hogyha az E leszármazottja annak a típusnak, ami a Store-nak az eredeti paramétere

```
void putAll(Store<? extends E> s) {
  for(int i = 0; i < s.size(); i++) {
      put(s.get(i));
}
```

- ha fordítva szeretném, hogy az egyik storeból átpakolni a másikba akkor super kell

```
void put2All(Store<? super E> s) {
  for(int i = 0; i < size(); i++) {
      s.put(get(i));
}
```

- Multiplate generic parameters:

```
<T extends X & Y>
```

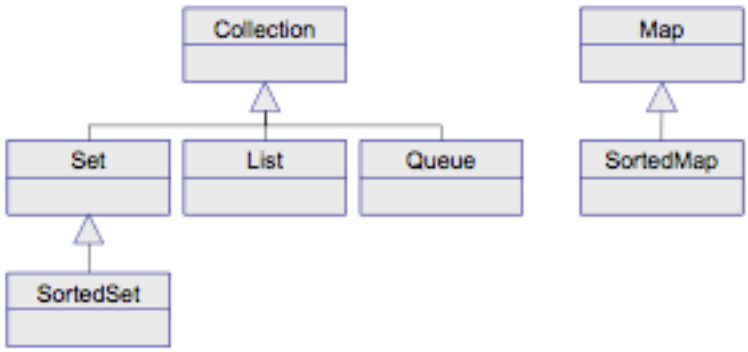
```
public static
<T extends E & Comparable<? super F>>
T max(Collection<? extends T> coll)
```

T típus legyen leszármazottja X-nek és Y-nak is. Olyat is lehet, hogy olyan T típussal akarok egy fv-t definiálni, ahogy az alsó képen látszik. (gyári java API)

Java Collections:
- alapja a tömb, minden kollekció öse a tömb
- mit várunk a kollekciótól: beszúrás, új elem, törlés, néha végigmenni (közös tulajdonságok), de vannak kollekciókra specifikus metódusok is

```
List<Integer> l = new ArrayList<Integer>();
// since J2SE 7
// List<Integer> l = new ArrayList<>();

for (int i = 0; i < args.length; i++) {
      l.add(Integer.parseInt(args[i]));
}

for (int i = 0; i < l.size(); i++) {
      System.out.println(l[i]+10);
}
```

- egészeket tároló lista, erre egy implementáció az ArrayList



- Collection a legösibb kollekció: tárol, végigmegyünk elemeket, keresünk
- ha plusz funkcionalitás akkor kibövítjük + dolgokkal
- Set: egy elemet egyszer tárol, List: sorba tároljuk az elemeket, Queue: két végét érjük el
- `void add(E e), void addAlL(Collections<? extends E> c), boolean remove (E e), boolean removeAll(Collections<? extends E> c), boolean contains(E e), int size, boolean isEmpty(), void clear(), boolean equals(E e), boolean retainAll(Col<? ext E> c), ...`

Iterator:
- tisztán objektumorientált iterátorok, 3 metódusuk van, konstuktor: `Iterator<E>`

```
Collection<Integer> c = ...;
...
Iterator<Integer> i = c.iterator();

while (i.hasNext()) {
      int a = i.next(); // outboxing
      if (a < 0) {
            i.remove();
      }
}
```

-`boolean hasNext(), E next(), void remove()`
- ha két iterátor megy a kollekción és töröl egyik a másik elöl akkor ConcurrentModificationException kivételt kapunk, iterálás közben ne bántsuk a kollekciót

Set:
- minden elemet csak egyszer tartalmaz, nem tudjuk hogy megy végig rajta az iterátor
- nincs plusz metódus, csak szemantikailag bövíti a kollekciót

SortedSet:
- rendezett halmaz, rendezést azért igényel hogy össze tudjuk hasonlítani az elemeket
- ha nincs természetes rendezés akkor saját komparátort tudunk létrehozni
  `(Comparator int compare(Object o1, Object o2)`

List:
- egy elemet többször is lehet tárolni, minden elemnek rögzített helye van a listában
- egy elemet csak úgy érhetünk el ha végigiterálunk a listán - ListIterator
- ArrayList-et kell alkalmazni ha dinamikus tömb kell (Vektor osztály is hasonló)

`-add(int index, E e), E get(int index), int indexOf(Object), int lastIndexOf(Object) E, remove(int index), boolean remove(Object o) E set(int index, E e), List<E> subList(int from, int to)`

Map:
- nem a Collection leszármazottja, speciális kollekció
- kupcs-érték párokat tárol, leképzése a valóságnak
- tipikus implementáció a hashMap
- létrehozás: Map<K, V> , K - kulcs, V - érték
- hasonló metódusok mint a Collection-ben (clear, equals, isEmpty, size,...)
- SortedMap - rendezett map, tipikus implementáció: TreeMap

For each loop:

```
Collection<Integer> c = ...;
...
for (Integer i : c) {
      System.out.println(i);
}
```

```
static public void main(String[] args) {
  for (String s : args) { System.out.println(s); }
}
```

Collections osztály:
- rendezés, min-max kereses, sorrend megfordítás, elemek összekeverése, stb...

Utility osztályok:
- StringTokanizer: feldarabolja a stringet általunk megadott formátumban
- Calendar/GregorianCalendar: dátum kezelés
- Random: véletlenszerüen akarunk dolgokat használni
- Scanner: kényelmes beolvasás a standard inputról
- Marh/StrictMath: matematikai függvények

Szálkezelés (threading):
- szálak: egymássalpárhuzamosan futnak, közös memótia, egymástól függetlenül hajtódnak végre alapvetöen
- szinkronizálni a szálakat, adatokat átvinni segédobjektumokkal tudunk
- hogyan csinálunk szálakat?
    - `run()` metódusnak kell lenni az osztályban és ebbe kerül a kód ami a többivel fog párhuzamosan futni, a run-ból hívható majd más metódus, de ez a kiinduló pont
    - ahhoz hogy a szál objektumot elindítsuk akkor a `start()` metódust hívjuk meg, ez indítja el a run-t
    - az osztálynak vagy a `Thread`-böl kell származnia vagy a `Runnable` interface-t kell megvalósítania

```
public class MyThread extends Thread {
  int a;
  int b;
  public MyThread(int i) { b=i; }
  public void run() {
    for (a = 0; a < b; a++) { System.out.println(a); }
  }
}
```

```
MyThread mt = new MyThread(1000);
mt.start();
...
```

```
public class MyThread implements Runnable {
  int a;
  int b;
  public MyThread(int i) { b=i; }
  public void run() {
    for (a = 0; a < b; a++) { System.out.println(a); }
  }
}
```

```
MyThread mt = new MyThread(1000);
Thread t = new Thread(mt); // kell egy szál, ami futtat
t.start();
...
```

- `yield()` - a CPU-t más szálnak átadja, lemond a futás jogáról
- `sleep(long milis [, int nanos])` - a szál vár megadott idöt, aludni küljdük
- `interrupt()` - megszakítja a szálat miközben vár

-`getId()`, `set/getName()`, `set/getPriorit()`, `isDaemon()`, `isAlive()`
- `setDaemon(boolean on)` - beállítja a daemon flaget, ha a JVM megáll az felfüggeszti véglel a daemon szálakat, a többit csak várakoztatja
- `static Thread currentThread()` - a futó szál referenciáját adja vissza
- `ThreadGroup getThreadGroup()` - a threadgroup-ot adja vissza
- `static int activeCount()` - aktív szálak száma a threadgroup-ban

```
private volatile Thread blinker;
MyThread(){ blinker =  Thread.currentThread(); }
public void stop() { blinker = null; }
public void run() {
    Thread thisThread = Thread.currentThread();
    while (blinker == thisThread) {
        try {
            thisThread.sleep(interval);
        } catch (InterruptedException e){}
        repaint();
    }
}
```

- szálak megállítására a stop metódus nem szép, beragadhatnak eröforrások, helyette egy flaget ellenörzünk, hogy futhat-e

Mutual exclusion:
- a szálaink ugyanazt az eröforrást akarják elérni (pl. egy kollekcióba belerakni, kivenni)
- minden java objektumnak van egy monitorta, ez egy speciális eszköz, ami lehetövé teszi hogy az objektumon csak egy szál dolgozhat
- ha más is szinkronizált, akkor a blokkon belüli dolgokat nem teheti meg a szál

```
Hashtable<String, Integer> ht = ...;
public void increment(String s) {
        ...
        synchronized (ht) {
                int i = ht.get(s);
                i++;
                ht.put(s,i);
        }
        ...
}
```

- lehet egész metódust vagy csak kódrészletet szinkronizálni

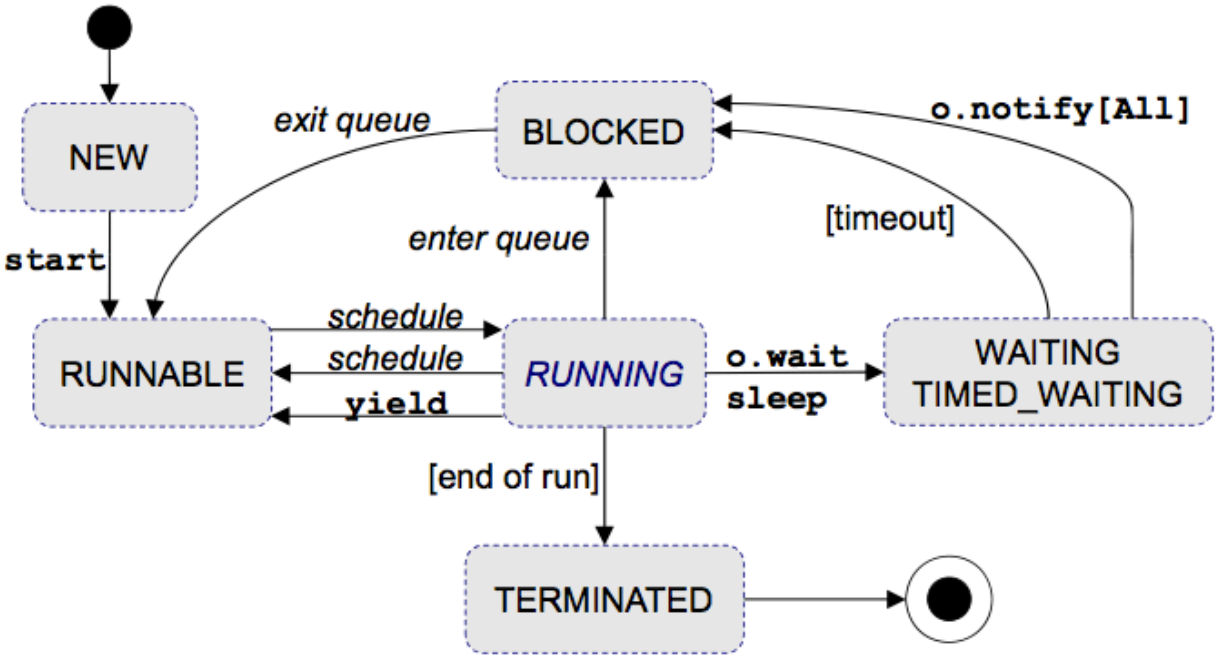
```
void foo() {
        synchronized(this) {
                ...
        }
}
```

```
synchronized void foo() {
        ...
}
```

- elöfordulhat, hogy két szál jelezni akar, egyik a másiknak hogy vmi lefutott, hogy ez-azt lehet csinálni
- `Object.wait([long milis [, int nanos]])` - a szál signal-ra vár az adott ideig, a szálnak szinkronizált blokkban kell lennie

```
synchronized (obj) {
        ...
        try {
                obj.wait(); // monitor szabad
        } catch (InterruptedException ie) {...}
        ...
}
```

- signal: `Object.notify()` - értesíti a szálat amelyik vár
- `Object.notifyAll()` - értesíti az összes várakozó szálat
- szálak állapotai:
    - `NEW`: most lett létrehozva, nincs elindítva
    - `RUNNABLE`: fut, már el van indítva (`start`)
    - `BLOCKED`: monitorra vár (`synchronized`)
    - `WAITING, TIMED_WAITING`: vár (`Object.wait, Thread.sleep`)
    - `TERMINATED`: meg lett állítva (`stop`), nem újraindítható


NEW
start
exit queue
BLOCKED
o.notify[All]
enter queue
[timeout]
RUNNABLE
schedule
schedule
yield
RUNNING
o.wait
sleep
WAITING
TIMED_WAITING
[end of run]
TERMINATED

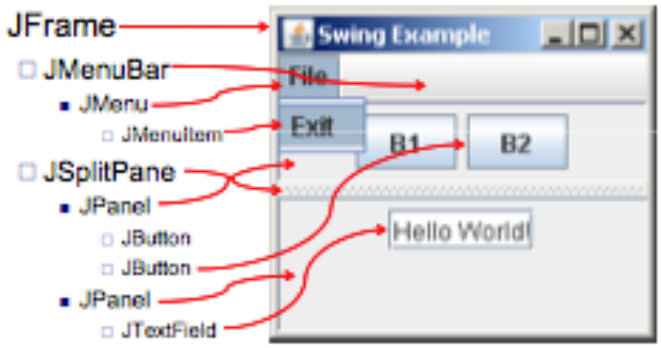
Java - Java GUI és SWING

AWT: java alapkészlete, elsö ablakkezelö könyvezet ez volt, heayv-weight widgeteket használ, oprendszer ablakkezelö könyvtárát hozza fel java szintre, natív használat miatt nagyon különbözöek a kinézetek

SWING: most ezt tanuljuk meg, egyszerübb toolkitet tenni egy elemre, van táblázat és fanézet, nem az oprendszer elemeit használja, minden javaban van megírva, egységes kinézet minden platformon, java.swing csomagban van.

SWT: eclipse framework az SWT-re épül, az eclipse gombjai nem swingre épülnek

Ablakozó architektúra:



Programozottan össze kell építenünk az ablakot. Létrehozzuk az ablakot, felépül az ablak. Következö lépés hogy beregisztráljuk az eseménykezelö dolgainkat: ha ráklikkelünk egy gombra, akkor történjen is valami. Majd az ablakot láthatóvá kell tenni.

Komponensek, konténerek:
- egyszerü funkcionalitású
    - JButton, JTextField, ...
    - levél elemek az ablakunkon, rájuk más nem helyezhetö
- konténerek
    - JPanel, JFrame, ...
    - újabb komponenseket lehet beléjük rakni
- komplex funkcionalitású
    - nem egy dolgot csinálnak, hanem több mindent
    - MVC-t alkalmaznak, több objektum dolgozik együtt
    - pl.: JList + JScrollPane

Widget-ek:
- a megjelenítés automatikus, magát kirajzolja
- lekérdezhetö: méret (min, max, preffered), láthatóság, enable/disable, event handling
- hozzá lehet adni öket konténerekhez, egy konténer egy másikbais rakható (fa hierarchia)

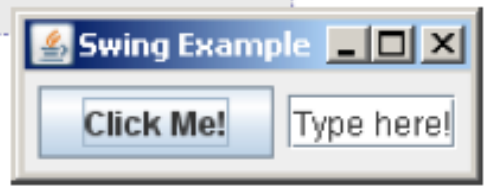
JButton:
- klasszikus gomb, megnyomható
- gombon lévö szöveg/kép beállítható

`-setEnabled(boolean), get/setText()`

JTextField:
- alap kontérner, ö felelös azért hogy a benne lévö widgetek jó méretüek legyenek
- elrendezés stratégiája beállítható, (x,y) koordinátán lévö komponens lekérdezhetö
-`add(Component[, param]), setLayoutManager(LayoutManager), getComponentAt(int, int)`

JFrame:
- keretes ablak, minden ablakozó müveletet támogat
- meg kell hívni egy metódust, hogy X-re bezáruljon (`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`)
-`add(Component, int where), pack(), setVisible(boolean)`

```
JFrame f = new JFrame("Swing Example");
JPanel p = new JPanel();
JButton b = new JButton("Click Me!");
JTextField t = new JTextField("Type here!");

p.add(b);
p.add(t);
f.add(p, BorderLayout.NORTH);

f.pack();
f.setDefaultCloseOperation(f.EXIT_ON_CLOSE);
f.setVisible(true);
```



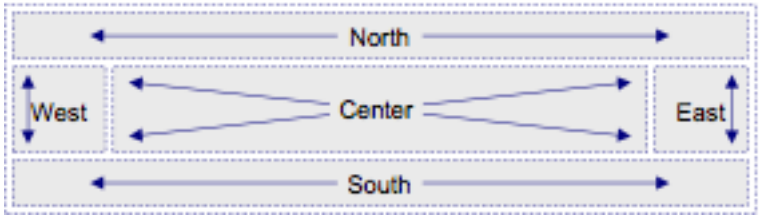
Event handling:
- eseményt kell kezelnünk hogy történjen is valami, observer mintának megfelelö módon fog történni, objektumorientáltan fogjuk kezelni
- két modelt kell definiálni:
    - listener: figyel arra hogy mi történik
    - subject: akivel történik
    - elözö példában subject a gomb, a listener az aki beregisztrálja hogy történt valami
- az esemény is egy objektum, egy interfacet kell implementálni
- a komponenseknek van egy `XEventListener` metódusa, a metódus paramétere lesz az az objektum, aki megvalósítja az esemény feldolgozásához szükséges metódusokat
-`java.awt.event.MouseEvent, java.awt.event.WindowsEvent, ...`

```
final class MyActionListener implements ActionListener {

  public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equals("date")) {
      t.setText((new Date()).toString());
    }
  }
}
```

```
...
JButton b = new JButton("Click Me!");
b.setActionCommand("date");
ActionListener al = new MyActionListener();
b.addActionListener(al);
...
```

Layout kezelés:
- hogyan tudunk a kinézetröl gondoskodni, kivesszük a konténerek kezéböl, a konténernek adunk valamit ami segít neki, ez a layout manager
- a konténereknek van default layout managere, le lehet cserélni (`setLayoutManager`)
- `add(...)` metódussal tudok hozzádani

BorderLayout:
- ezzel születik a JFrame, 5 terület (észak, dél, kelet, nyugat, közép) 1 terület -1 widget
- árbán a nyíl jelentése = arra nyúlik szét a terület



FlowLayout:
- ezzel születi a JPanel, sorban lehet helyezni az elemeket
- rendezés beállítható: jobb, közép, bal

CardLayout:
- egyes komponenseket egyes kártyákra tudjuk rátenni, meg tudjuk adni hogy melyik kártya látszódjon
- ha olyan akartunk hogy több konténert rakunk össze, de csak az egyik legyen fölül, akkor ezzel lehet megoldani

GridLayout:
- N*M elemet tudunk elhelyezni mátrix formában, de minden ugyanakkora,

GridBayLayout:
- mátrix, de minden elemhez definiálhatjuk hogy a mátrixnak hány kockáját foglalja el

BoxLayout:
- hasonló a FlowLayouthoz, csak máshogy rendezi az elemeket

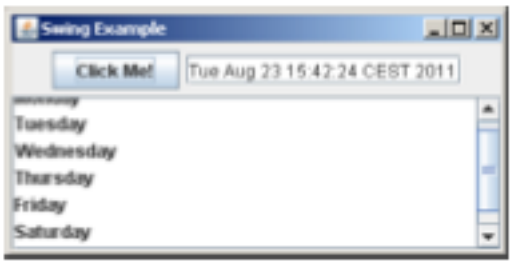
SpringLayout:
- hasonló a FlowLayouthoz, csak függölegesen rendezi
- minden mátrix sor és oszlop a legnagyobb elemnek megfelelö szélességre fog nyúlni

GroupLayout:
- ezzel mindent lehet, összetettebb layoutokat is lehet csinálni
- elég macerás megírni kézzel, designerrel érdemes ilyen csinálni

Komplex Widgetek:
- JList, JComboBox, JTable, JTree, ...
- ezekhez MVC van (Model-View-Controller)

JList:
- feladata az hogy elemek listáját mutassa
- legprimitívebb lista az objektumtömb, ha megjelenik egy új tömbelem akkor újrarajzol
- mindent megjelenít, ha nincs elég hely akkoris
- scrollozáshoz külön model van: JScrollPane

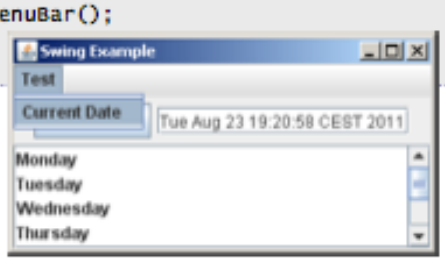
```
DefaultListModel model = new DefaultListModel();
model.addElement("Monday");
model.addElement("Tuesday");
...
JList l = new JList(model);
JScrollPane sp = new JScrollPane(l);
f.add(sp, BorderLayout.CENTER);
```



Alacsony szintü grafika:
- hogyan tudok köröket mozgatni meg négyzeteket?
- csinálok egy `JComponent` objektumot, ennek egy leszármazottja kell, lesz a paint meg egy repaint metódusa
- `Graphics`: primitív dolgok rajzolásához (vonal, szöveg, stb...)
- `Graphics2d`: bonyolultabb grafika (négyzet, kör, stb...)
- bufferelt rajzoláshoz `bufferedImage` kell
- blablabla... (nem hiszem hogy vizsgaanyag, akinek házihoz kell az majd megnézi diából)

Speciális komponensek: Menü készítés

```
ActionListener al = new MyActionListener();
b.addActionListener(al);
JMenuItem mi1 = new JMenuItem("Current Date");
mi1.setActionCommand("date"); // setting action command
mi1.addActionListener(al);    // adding listener
JMenu m1 = new JMenu("Test");
m1.add(mi1);
JMenuBar bar = new JMenuBar();
bar.add(m1);
f.setJMenuBar(bar);
```

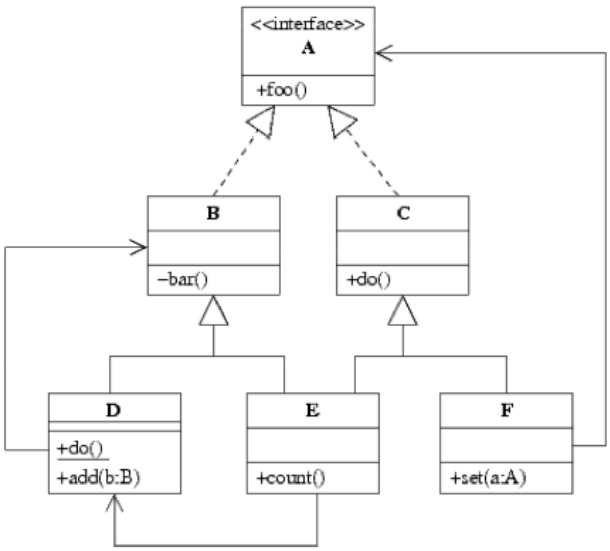


Feladatok Class diagram témakörből

1. feladat típus (mi van az ábrán):

A mellékelt ábrán milyen UML elemek láthatóak ?

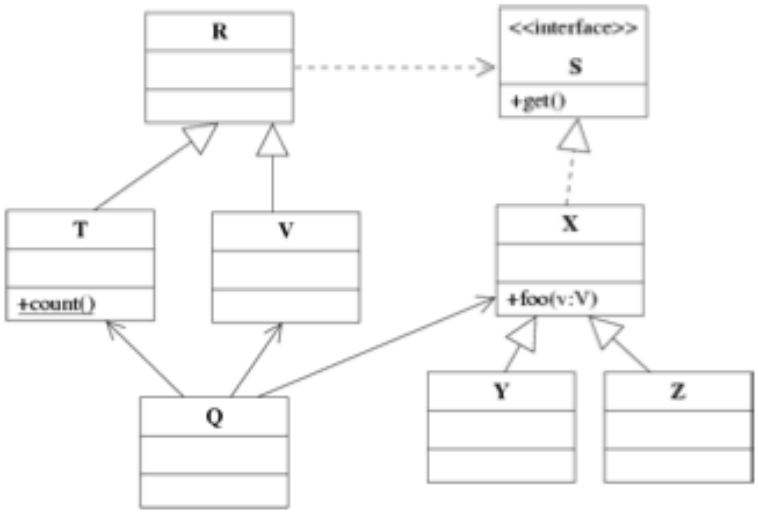
- ☐ aggregáció
- ☐ függőség
- ■ osztály metódus
- ☐ példányosítás
- ☐ multiobject
- ■ realizálás (impl)
- ☐ kollaboráció
- ■ navigáció
- ☐ qualifier (minősítő)
- ☐ absztrakt osztály
- ■ sztereotípus



Aggregáció: rész-egész kapcsolat (rombusz)
Függőség (dependencia): szaggatott vonal, sima nyíllal
Osztály metódus: metódus van rajta, pl C osztály do metódusa
Példányosítás: ezen az ábrán csak osztályok vannak, példányosítással objektumot hoznánk létre belölük
Multiobject: nem tanuljuk a félévben és nincs is rajta
Realizálás (impl): van, B és C is megvalósítja az A interfészt
Kollaboráció: osztály diagramon nem is lehet kollaboráció, bullshit
Navigáció: természetesen, van pl F-böl A-ba menö irányított (navigált) asszociáció
Qualifier (minösítö): nincs rajta, kis doboz lenne a nyíl vége és az osztály között
Absztrakt osztály: nincs, dölt betüvel van írva (*Italic*)
Sztereitípus: van, az <<interface>> sztereotípia

Jelölje be, hogy a mellékelt ábrán milyen UML elemek láthatóak

- ☐ aggregáció
- ■ függőség
- ■ osztály metódus
- ☐ példányosítás
- ☐ qualifier (minősítő)
- ☐ objektum
- ■ asszociáció
- ☐ kollaboráció
- ☐ iteráció
- ☐ protected metódus
- ■ realizálás (implementálás)



2. feladat típus (static):

```
x2.a = x1.a + 5;
x1.a +=5;
```

Mennyi lesz `x1.a` és `x2.a` ?

`x1.a` = 10 `x2.a` = 10

X
int a = 0
private aX(): int

Az a változó X osztályvan statikus (mert alá van húzva), tehát az az osztályhoz tartozik és csak 1 van belöle (akkor is létezik ha X osztályt nem is példányosítjuk).
x2.a = x1.a + 5 azt jelenti, hogy a = a+5, tehát elsö sor lefutása után a=5
x1.a += 5 azt jelenti, hogy a = a + 5, tehát második sor lefutása után a=10
ezért x1.a és x2.a értéke is 10 lesz.

Elkészítjük az alábbi `O` osztály két példányát, `o1`-et és `o2`-t. Ezt követően végrehajtjuk a következő műveleteket:

```
o2.x = -3; o1.x = 4;
o1.y = o2.x + 4;
o2.y = o2.x + o1.y;
```

Mennyi lesz a változók értéke ?

`o1.y` = 8 `o2.y` = 12

O
int x = 5 int y = -3
private aX(): int

Elkészítjük az alábbi `C` osztály két példányát, `c1`-et és `c2`-t. Ezt követően végrehajtjuk a következő műveleteket:

```
c2.a = 8; c1.a = -2;
c1.b = c2.a + 4;
c2.b = c2.a + c1.b;
```

Mennyi lesz a változók értéke ?

`c1.b` = 2 `c2.b` = 0

C
int a = 12 int b = 5
private aX(): int

Elkészítjük az alábbi `Y` osztály két példányát, `x1`-et és `x2`-t. Ezt követően végrehajtjuk a következő két műveletet:

```
x2.a = x1.a * 2;
x1.a +=5;
```

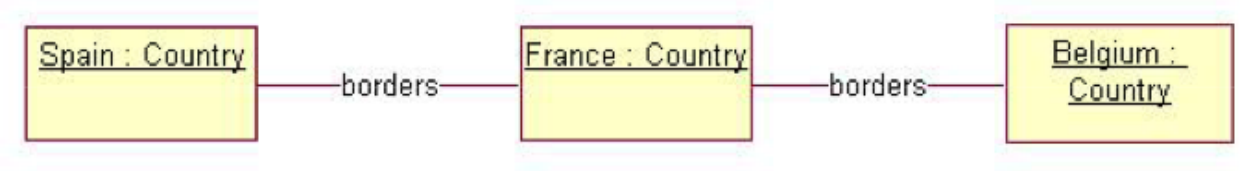
Mennyi lesz `x1.a` és `x2.a` ?

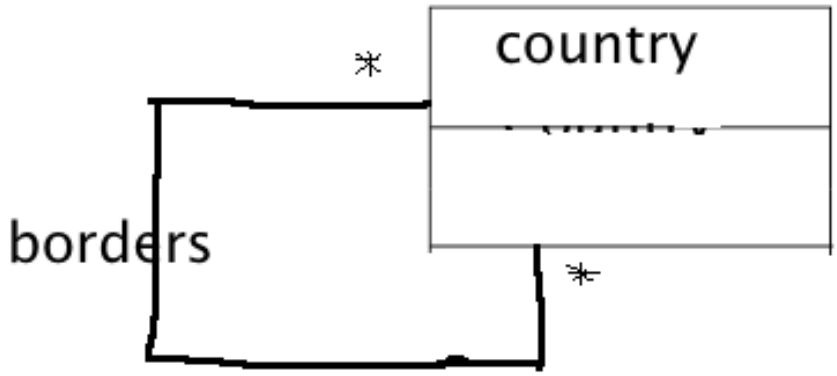
`x1.a` = 11 `x2.a` = 11

Y
int a = 3
aX()

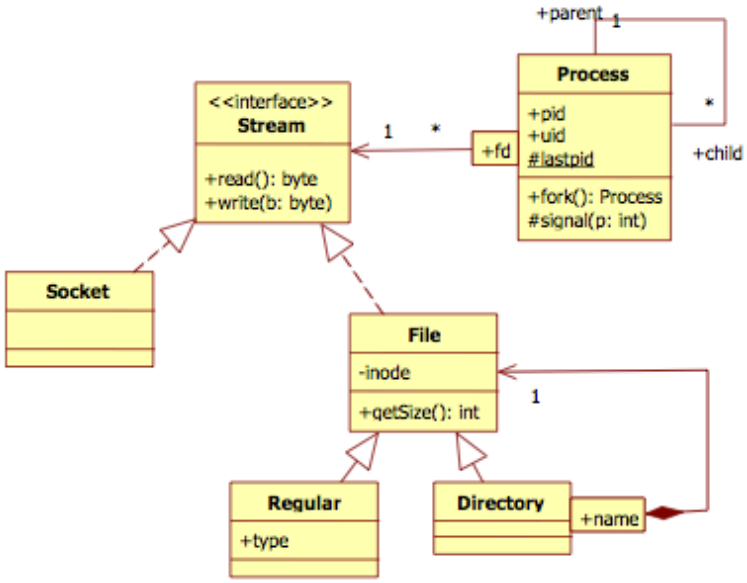
3. feladat típus (rajzolj osztálydiagramot):

Rajzoljon UML osztálydiagramot az alábbi objektumdiagram alapján !



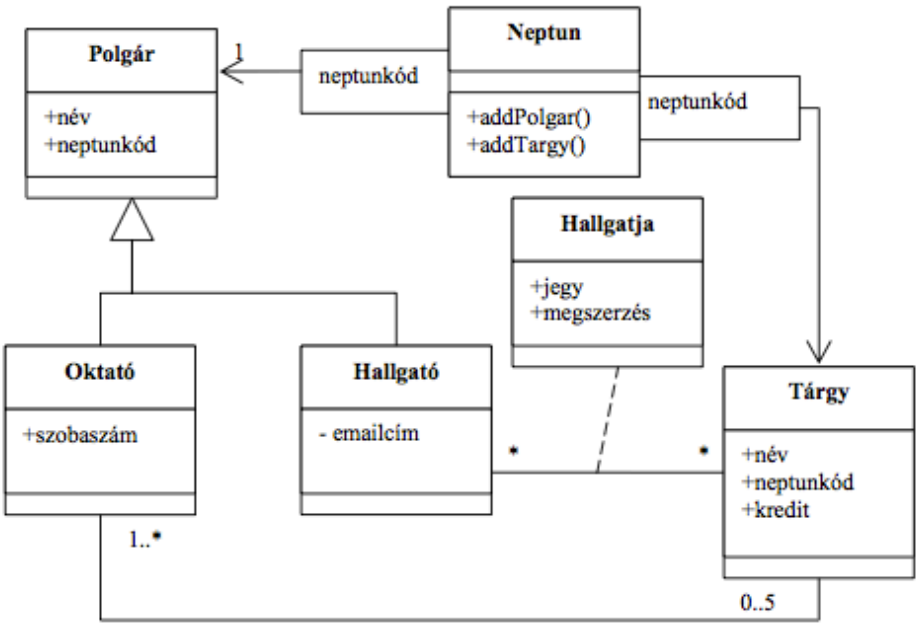


Készítsen UML 2 osztálydiagramot (class diagram) az alábbi leírás alapján! Használja a kövéren szedett kifejezéseket! Ahol lehet, adja meg a paraméterek, attribútumok, stb típusát is!
Az OOniX operációs rendszerben **folyamatok**, **fájlok** és hálózati kapcsolatok (**socket**) vannak. A folyamatoknak van azonosítója (**pid**), tulajdonosa (**uid**). Egy folyamatból a **fork**() metódussal lehet újat létrehozni. Minden folyamat ismeri a közvetlen **ősét** és a **gyerek**eit. A fájloknak van egy senki más által nem látható **inode** száma, és lekérdezhető a **méret**ük. A fájlok többfélék lehetnek: **könyvtárak**, amelyek más fájlokat tartalmazhatnak (a **nevük** alapján), **reguláris** fájlok, amiknek van **típus**a, stb. Minden fájl egy könyvtár része. A folyamatok egyformán kezelhetnek socketet és fájlt is, de csak egy közös interfészt (**stream**) látnak belőlük, amiken bájtokat lehet **olvasni** és **írni**. Az ilyen objektumokról a folyamatnak van egy listája, aminek az elemeit fájlleíróval (**fd**) azonosítja. A folyamatok létrehozásakor egy (a folyamatok számára közös) számláló (**lastpid**) növekszik, ez lesz az újonnan létrehozott folyamat azonosítója. A folyamatoknak más folyamatok tudnak üzenni a **signal**() üzenet meghívásával (egy darab egész típusú paramétere van). A lastpid és a signal csak folyamatból (és esetleges leszármazottjából) látható.



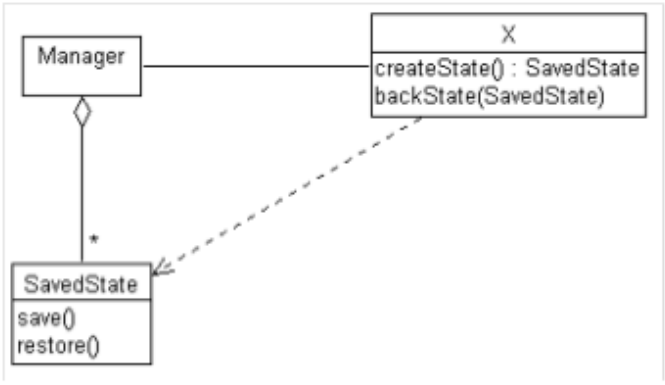
Rajzoljon UML2 osztálydiagramot az alábbi történet alapján! Jelölje a számosságokat is!

Az egyetemi polgárokat, akiknek nyilvános a neve és a neptunkódja, a Neptun rendszer tartja nyilván, mégpedig a polgáronként egyedi neptunkód alapján. Polgár az oktató és a hallgató is. Az oktatónak nyilvántartjuk a szobaszámát, a hallgatónak az emailcímét (privát adat). A Neptunban tároljuk a tárgyakat is egyedi neptunkóddal. A tárgyaknak szintén ismerjük a nevét és neptunkódját, valamint az értük kapható kreditek számát. Egy hallgató több tárgyat is felvehet (hallgatja), egy tárgyra több hallgató is járhat. Ezen kívül egy adott hallgató egy adott tárgyra kapott jegyét és a megszerzés évét is nyilvántartjuk. Egy tárgyat legalább egy oktató oktat, egy oktatónak pedig lehet több tárgya is (maximum 5), de van akinek egy sincs. A Neptunba fel tudunk venni új tárgyat és polgárt.

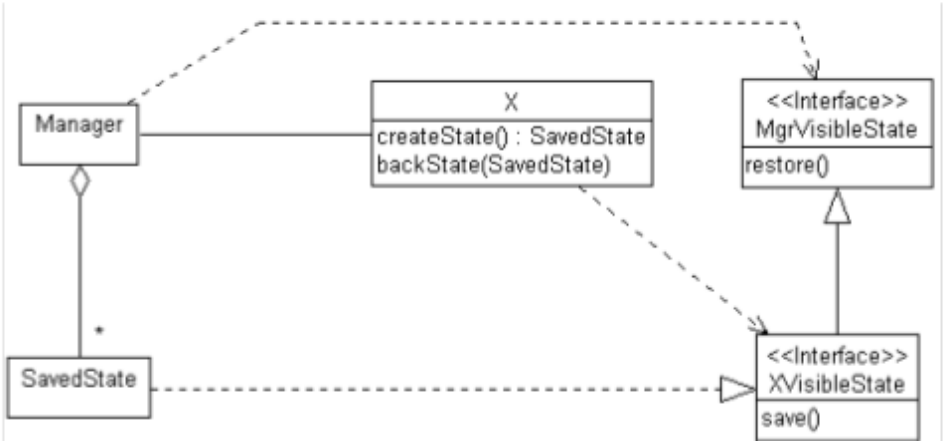


Legyen egy `X` objektumunk, amelynek állapotát időnként menteni kell egy (`SavedState` objektumba, annak a `save()` metódusát meghívva), hogy az később visszaállítható legyen (a SavedState `restore()` metódusával). A SavedState objektumokat egy `Manager` objektum kezeli, amely utasítja `X`-et egy mentésre (`createState()`), vagy az átadott SavedState szerinti állapot visszaállítására (`backState(SavedState)`).

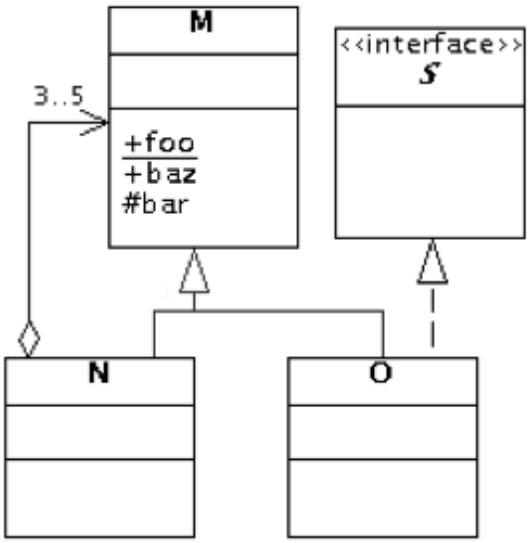
Rajzoljon **UML osztálydiagramot**, az operációk feltüntetésével !



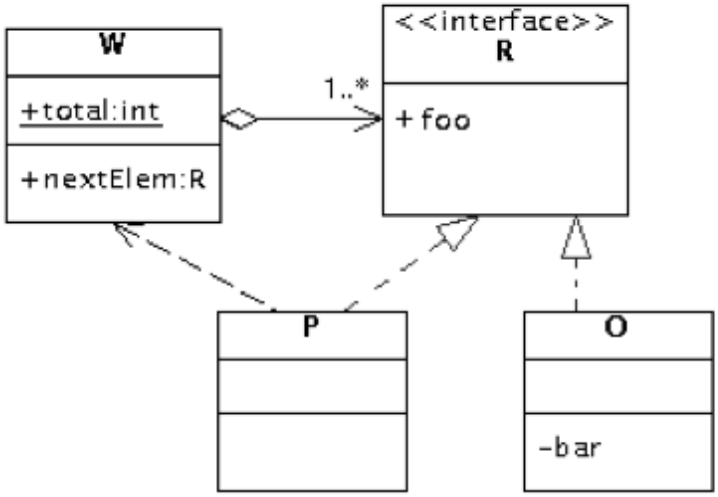
Legyen `MgrVisibleState` interfész, amelyből származtatjuk az `XVisibleState` interfészt úgy, hogy ez utóbbi interfésznek van `save()` operációja, amit `Manager` nem lát.



Az `M` osztálynak három metódusa van: `foo()`, `bar()` és `baz()`. `foo()` és `baz()` publikus, `bar()` protected. `foo()` osztályszintű.
Az `M` osztálynak két leszármazottja van: `N` és `O`. `N` tartalmaz legalább 3, de legfeljebb 5 `M` osztályú objektumot. `O` megvalósítja az `S` interfészt.
Készítsen a fenti leírás alapján **UML osztálydiagramot** !

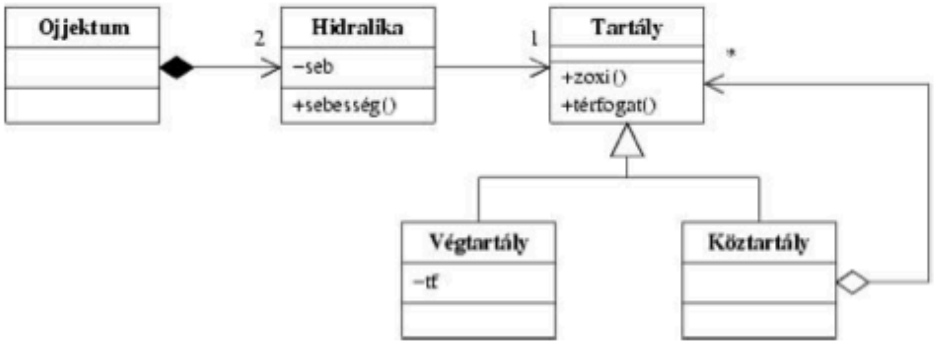


A `W` osztálynak van egy publikus, `total` nevű, osztályszintű attribútuma, valamint egy publikus `nextElem` metódusa, amely `R` interfészű objektumot ad visszatérési értékként. A `W` osztály tetszőlegesen sok (de legalább egy) `R` interfészű objektumot tartalmazhat.
Az `R` interfésznek egy publikus `foo()` metódusa van. Az interfészt két osztály valósítja meg, `O` és `P`. `O`-ban van egy privát `bar()` metódus, míg `P` függ `W`-től.
Készítsen a fenti leírás alapján **UML osztálydiagramot** !

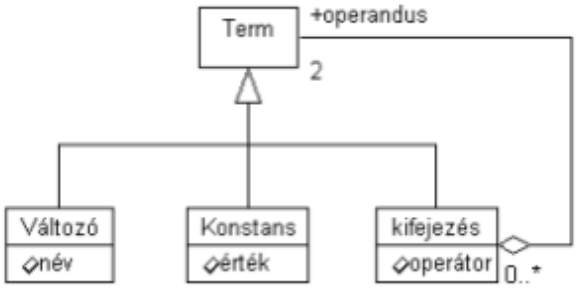


Árpád ojjektuma két hidralikából áll. A hidralika pontosan egy tartályhoz kapcsolódik, belőle a zoxigént tudja felvenni (a tartály zoxi metódusának meghívásával). Teccik érteni ? Hogy ezt milyen sebességgel tudja csinálni, az a neki (mármint a hidralikának) küldött sebesség() üzenettel állítható, amit az a privát seb attribútumában tárol. A tartály vagy végtartály, vagy köztartály. A köztartályban további tartályok lehetnek. A végtartály térfogatát (pl. 30000 vagy 300000 liter) egy privát attribútum (tf) tárolja. A tartályoktól le lehet kérdezni, hogy mekkora a térfogatuk (térfogat() metódus). A köztartály térfogata a benne levő tartályok térfogatának összege.

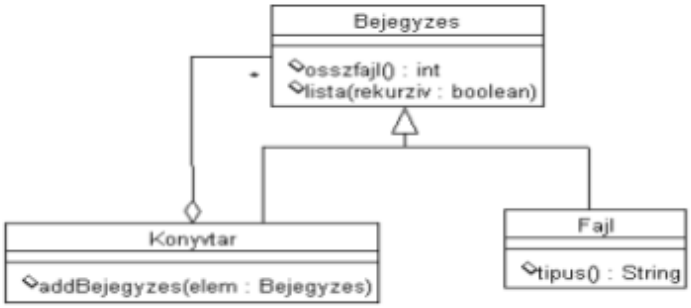
Rajzolja meg a fenti leírásnak megfelelő **UML** osztálydiagramot az attribútumok és a metódusok, valamint azok láthatóságának feltüntetésével ! Teccik érteni ?



Egy matematikai term lehet változó, konstans vagy kifejezés. A változót megadhatjuk a nevével, a konstanst az értékével. Egy kifejezés pontosan két operandust és egy bináris operátort tartalmaz. Operandus lehet bármely term. Rajzoljon osztálydiagramot



Egy fájlrendszerben fastruktúrában könyvtárak és fájlok vannak. A könyvtárakban további könyvtárak és fájlok lehetnek. Egy könyvtárba fel lehet venni újabb könyvtárakat és fájlokat. A könyvtárakat ki lehet listázni (mind rekurzívan, mind csak egyszeres mélységben), a fájloktól meg lehet kérdezni a típusukat. A könyvtárak meg tudják mondani, hogy összesen (minden alkönyvtárat beleértve) hány fájl található bennük.
Készítsen osztálydiagramot és tüntesse fel a metódusok szignatúráit !



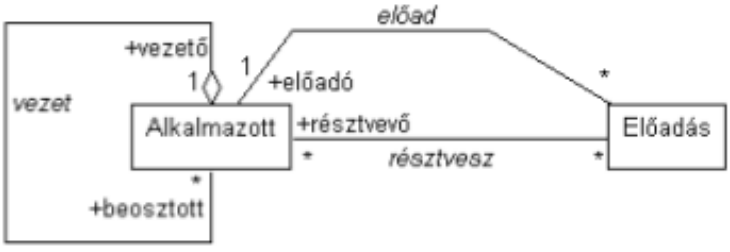
Készítsen UML osztálydiagramot és azon adja meg az osztályok metódusainak szignatúráit is !

A Vidékelőremozdító Minisztérium felépítése sok rétegű hierarchiát alkot. A minisztérium államtitkárságokból, az államtitkárság divíziókból, a divíziók főosztályokból, a főosztályok … stb. állnak. Minden egységnek van egy vezetője, és számos tisztviselője.

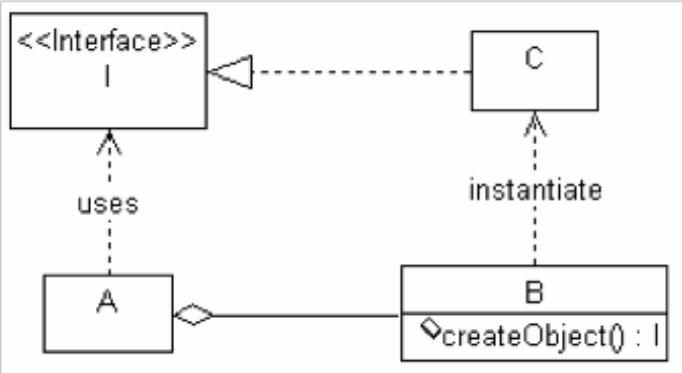
Az egységtől lekérdezhető az elnevezése, alakulásának éve, a vezető repült óráinak száma valamint az egységhez tartozó összes alkalmazotti (vezető + tisztviselői) létszám. Az alkalmazottaktól lekérdezhető az adószámuk, a TAJ-számuk és a fizetésük.

Egy cég az alkalmazottai részére továbbképzés céljából szakmai előadásokat szervez, amelyeket a cég egy-egy hozzáértő munkatársa tart. Az egyik előadás előadója lehet a másiknak hallgatója. Egy előadáson csak azok vesznek részt, akiket erre a vezetőjük kijelölt. Tanfolyamra küldhetik a vezetőket is.

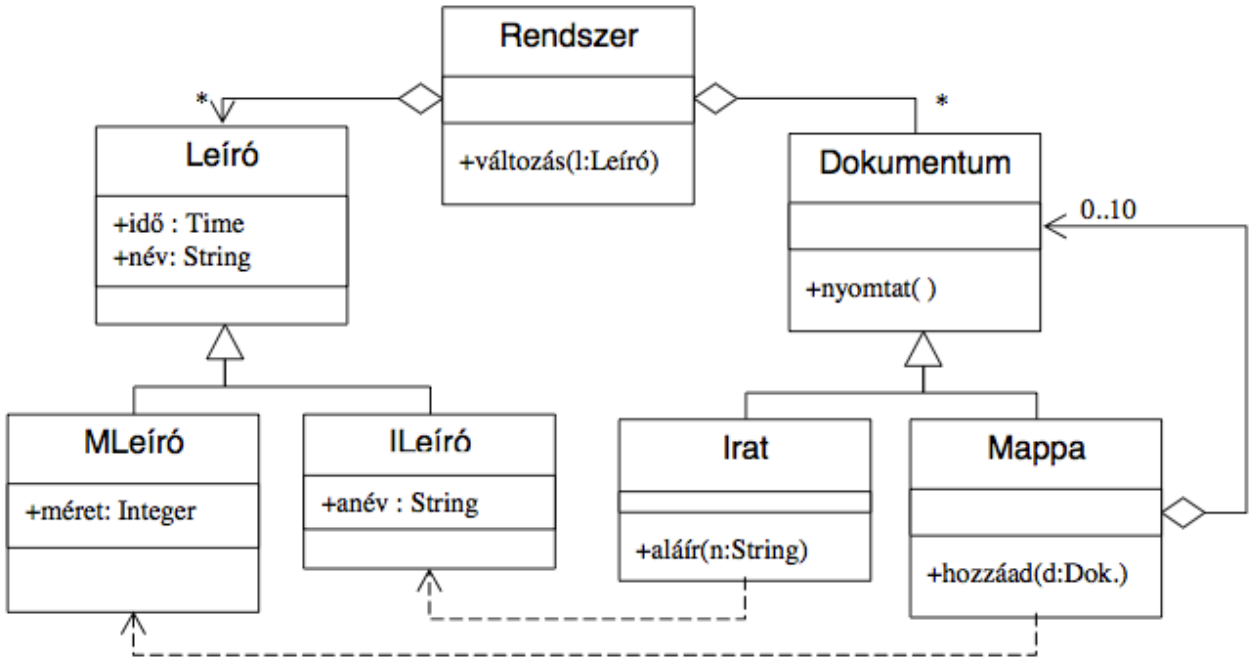
Rajzoljon **osztálydiagramot** a szerepek és a multiplicitások feltüntetésével.



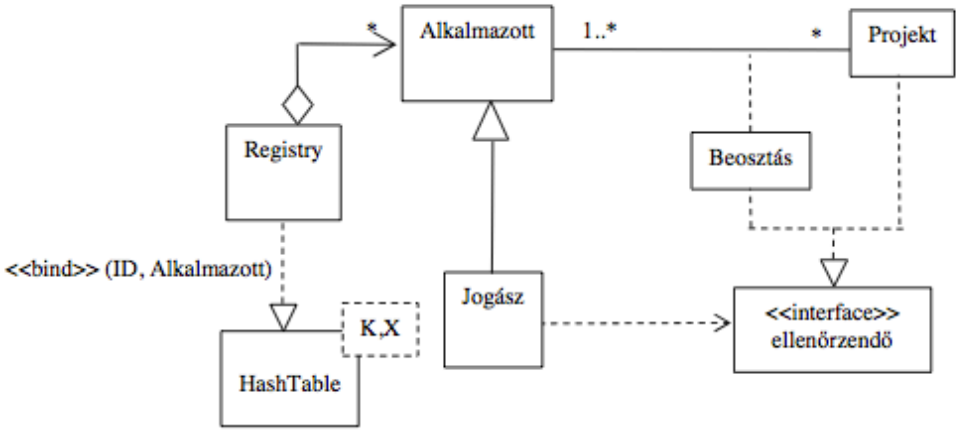
Az `A` osztályú objektum használni akar egy `I` interfészt megvalósító objektumot, de `A` nem ismer olyan osztályt, amelyik implementálná `I`-t. Viszont `A`-nak van egy `B` osztályú komponense, amely ismeri az `I`-t megvalósító `C` osztályt. `A` meghívja `B` `createObject()` metódusát, amely konstruál egy az `A` elvárásainak megfelelő objektumot. Rajzoljon UML osztálydiagramot !



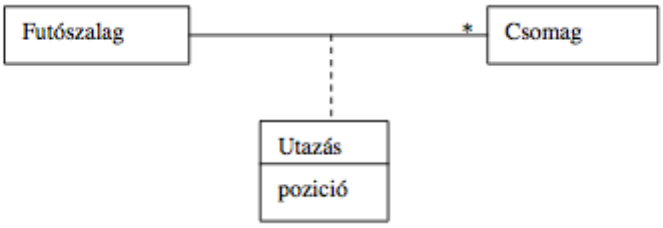
Készítsen UML2 **osztálydiagramot** **(class diagram)** az alábbi leírás alapján !
Egy dokumentumkezelő rendszerben heterogén kollekcióként névvel ellátott dokumentumokat tárolunk. A dokumentum lehet irat vagy mappa. A mappa további dokumentumokat tárolhat, de maximum 10-et. A rendszernek képesnek kell lennie arra, hogy később további dokumentumfajtákkal (fotó, hangszalag stb. bővítsük). A dokumentumokat ki tudjuk nyomtatni (*nyomtat*), az iratokat alá lehet írni (*aláír*), a mappákba újabb dokumentum helyezhető (*hozzáad*). A rendszer nyilvántartja a dokumentum-módosításokat is. Minden dokumentum a módosításról értesíti a rendszert (*változás* metódus), és átadja a módosítás adatait leíró objektumot. Ebben a módosuló dokumentum megadja a nevét és a módosítás idejét, de ezen kívül a dokumentum típusától függő egyedi adatok is szerepelhetnek (irat esetén az aláíró neve, mappa esetén a mappa mérete).



Rajzoljon UML 2 osztálydiagramot! A metódusokat és attribútumokat nem kell jelölje ! Csak a **vastagon** szedett classifier-ek szerepeljenek a diagramon !
Egy cégnél nyilvántartják az **alkalmazott**ak és a **projekt**ek adatait. Az alkalmazottak projektekhez vannak rendelve, minden projektnél különböző beosztásban. Minden projekten dolgozik legalább egy alkalmazott, de lehet olyan alkalmazott, aki éppen nincs projekthez rendelve. A **beosztás** határozza meg például, hogy mennyi bónuszt kap az alkalmazott az adott projekt sikere esetén. Mind a projekt, mind a beosztás jogilag **ellenőrzendő** (megvalósítják az **ellenőrzendő** interfészt). A **jogász** (aki persze a cég alkalmazottja is egyben) dolga, hogy az ellenőrzéseket elvégezze. A **registry**ben tárolják az alkalmazottak azonosítóinak és az alkalmazotti adatoknak az összerendelését. A registry az egyszerűség kedvéért a **HashTable** (K kulcs és X érték paraméterű) template osztályt példányosítja, ahol a kulcs az azonosító, az érték az alkalmazott adata.

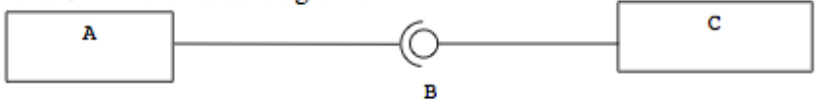


Egy futószalagon csomagok utaznak. A szalagot elhagyó csomagot a későbbiekben is használni akarjuk. Ki tudja (kinek a felelőssége), hogy mi a csomag pozíciója a futószalagon ? Legyenek az osztályok kohézívek és lazán csatoltak ! A probléma modellezésére rajzoljon UML osztálydiagramot a pozíció feltüntetésével !

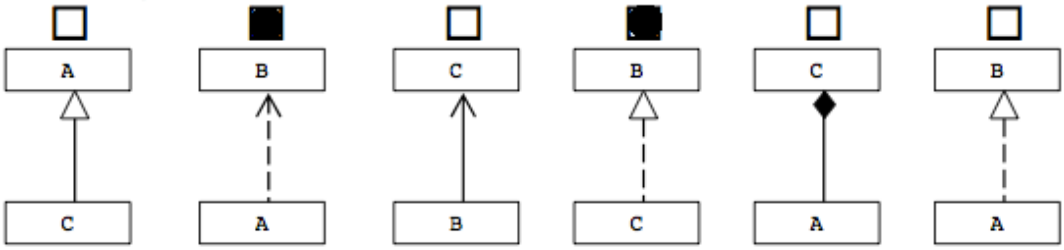


4. feladat típus (szerkezeti diagram):

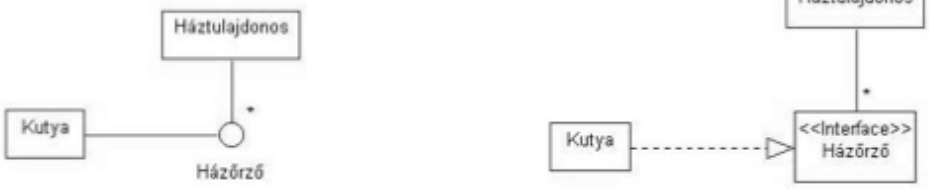
Legyen a következő UML2 szerkezeti diagram !



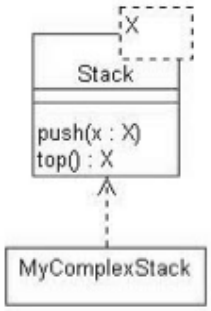
Feltételezve, hogy a fenti szerkezeti diagramon szereplö elemek között egyéb kapcsolat nincs, jelölje meg az alábbiak közül az igaz állítás(oka)t !



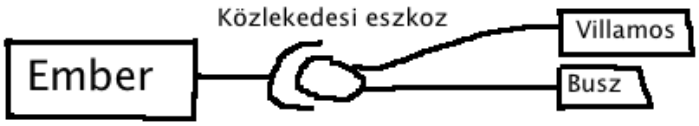
A Kutya osztály megvalósítja a Házőrző interfészt. A Háztulajdonos a kutyáit házőrzésre használja. Rajzoljon UML szerkezeti diagramot !



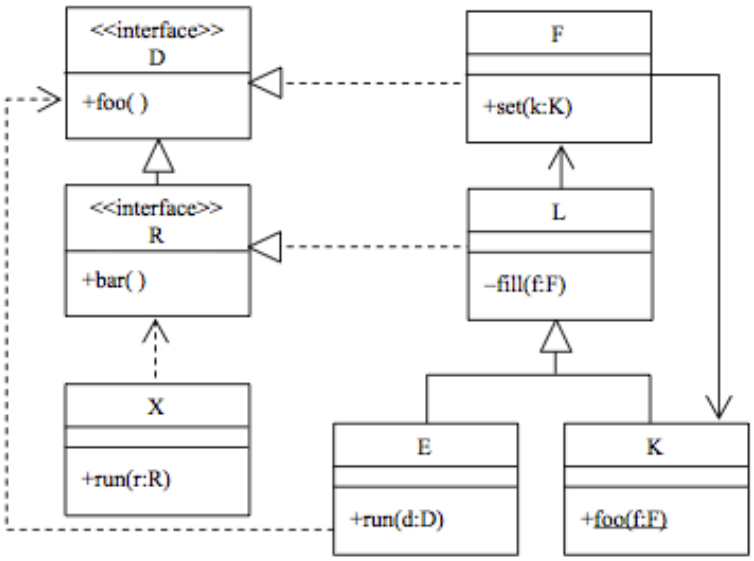
Legyen egy *Stack* paraméteres osztályunk, amelybe a paraméternek (*X*) megfelelő elemek tehetők. Legyen a stacken egy *push* művelet a szokásos jelentéssel, és egy *top* művelet, ami visszaadja a stack tetején található értékeket. A *Stack* template felhasználásával elkészítjük a *MyComplexStack* osztályt. Rajzoljon **UML szerkezeti diagramot** !



A Villamos és Busz osztályok megvalósítják a Közlekedési_eszköz interfészt. Az Emberek közlekedési eszközöket használnak az utazáshoz.
Rajzoljon UML szerkezeti diagramot !



5. feladat típus (diagram + állítások)



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] **E** bárhol helyettesíthető **K**-val, mert van közös ősük.

[B] **L** bárhol helyettesíthető **F**-fel, mert mindketten megvalósítják a **D** interfészt.

[B] **L** nem helyettesíthető **E**-vel, mert **L**-nek van privát metódusa.

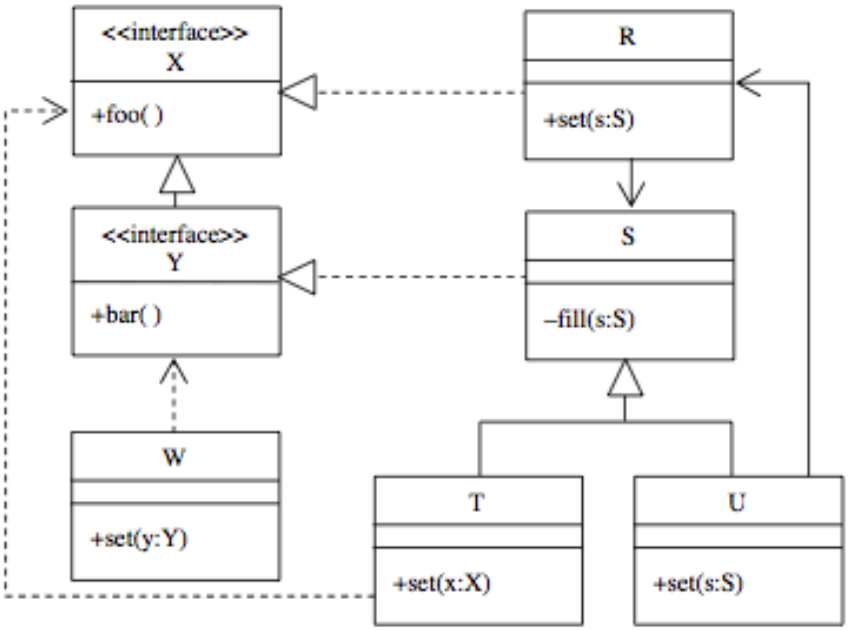
[B] **F set(k:K)** metódusa meghívhatja egy paraméterül kapott **K fill(f:F)** metódusát, mert **K** függ **F**-től.

[B] **X run(r:R)** metódusa kaphat paraméterül **F** osztályú objektumot, mert **X** függ **R**-től.

[E] **K**-nak nincs **foo()** szignatúrájú metódusa, mert **K**-t nem lehet példányosítani.

[B] **X run(r:R)** metódusa nem kaphat paraméterül **K** objektumot, mert **K**-nak van statikus metódusa.

[B] **K foo(f:F)** metódusa nem hívhatja meg a paraméter **foo( )** metódusát, mert az utóbbi metódus nem statikus.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **Y** bárhol helyettesíthető **W**-vel, mert **W** az **Y** leszármazottja.

[D] **U** bármikor lehet **T** **set(x:X)** paramétere, mert **U** megvalósítja az **X** interfészt.

[B] **R** meghívhatja saját **set(s:S)** metódusából egy **W** **set(y:Y)** metódusát, mert **S** megvalósítja **Y**-t.

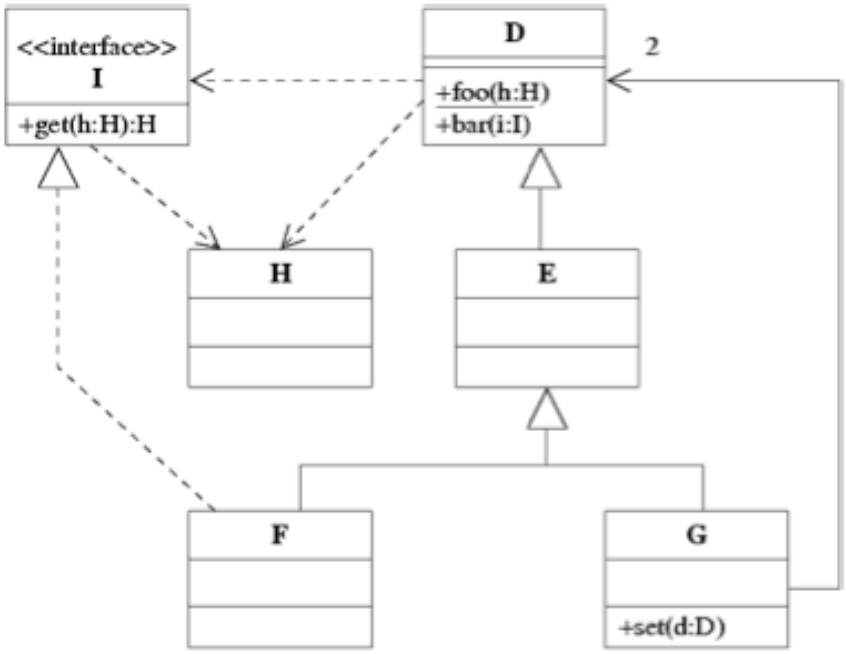
[E] **S** **fill(s:S)** metódusa nem kaphat paraméterül **T**-t, mert a metódus protected.

[A] **T** megvalósítja az **X** interfészt, mert **T** az **R** leszármazottja.

[B] **T** pontosan egy **U**-t tartalmazhat, mert csak egy közvetlen ősük van.

[E] **T** bárhol helyettesíthető **U**-val, mert egyforma az interfészük.

[B] **U** meghívhatja **S** **fill(s:S)** metódusát, mert **R** asszociációban van **S**-sel.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[D] `F` nem helyettesíthető `E`-vel, mert `E` nem leszármazottja `F`-nek.

[E] `G` helyettesíthető `F`-fel, mert `D` nem közös ősük.

[A] `H` nem hívhatja meg `D` `foo` metódusát, mert a metódus absztrakt.

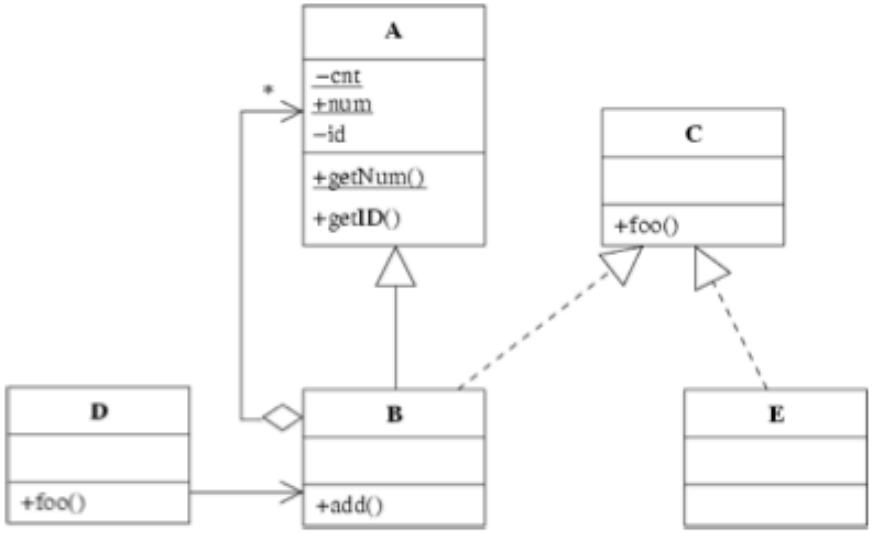
[E] `D` `bar` metódusa nem kaphat paraméterül `F`-et, mert `F` nem leszármazottja `D`-nek.

[A] `D` `foo` metódusa nem kaphat paraméterül `I`-t, mert az `I` valósítja meg `H`-t.

[B] `G` konstruktora pontosan kétszer köteles meghívni `D` `bar` metódusát, mert pontosan két `D`-t ismer.

[A] `G` `set` metódusa meghívhatja a paraméterül kapott `D` objektum `bar` metódusát, mert a metódus statikus.

[C] `F` nem hívhatja meg `G` `set` metódusát, mert mindketten függnek az `I` interfésztől.



A - csak az első tagmondat igaz
B - csak a második tagmondat igaz
C - mindkét tagmondat igaz, de a következtetés hamis
D - mindkét tagmondat igaz és a következtetés is helyes
E - egyik tagmondat sem igaz
(**F** - az ki van zárva, mert nem tudom)

[B] egy `D` osztályú `d` objektum nem mindig hívhatja meg egy `B` osztályú `b` objektum `getNum()` metódusát, mert előfordulhat, hogy `b` nem aggregál egy `A` osztályú objektumot sem

[B] `B` nem örökli `A` `num` attribútumát, mert az `A.num` statikus

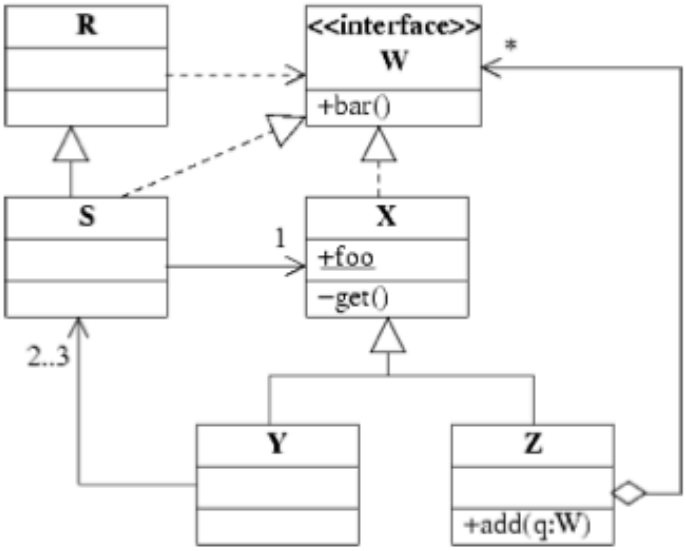
[B] `B` osztályú objektum helyén bárhol használhatok `E` osztályút, mert mindkettő a `C` interfészt valósítja meg

[D] `B` osztályú objektum nem látja `D` osztályú objektum `foo()` metódusát, mert a navigáció iránya `D`-től `B`-re mutat

[C] `E` nem látja `B` `add()` metódusát, mert `E`-nek nem őse `A`

[A] `B` nem látja az aggregált `A`-k `id` attribútumát, mert az `id` statikus

[B] `D` nem hívhatja `B` `foo()` metódusát, mert a `C` interfészt nem ismeri



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[D] *R* helyettesíthető *S*-sel, mert az *S* osztály az *R* leszármazottja.

[D] *Z* aggregálhat *S*-t, mert *S* megvalósítja a *W* interfészt.

[C] *S* módosíthatja *X foo* attribútumát, mert az attribútum statikus.

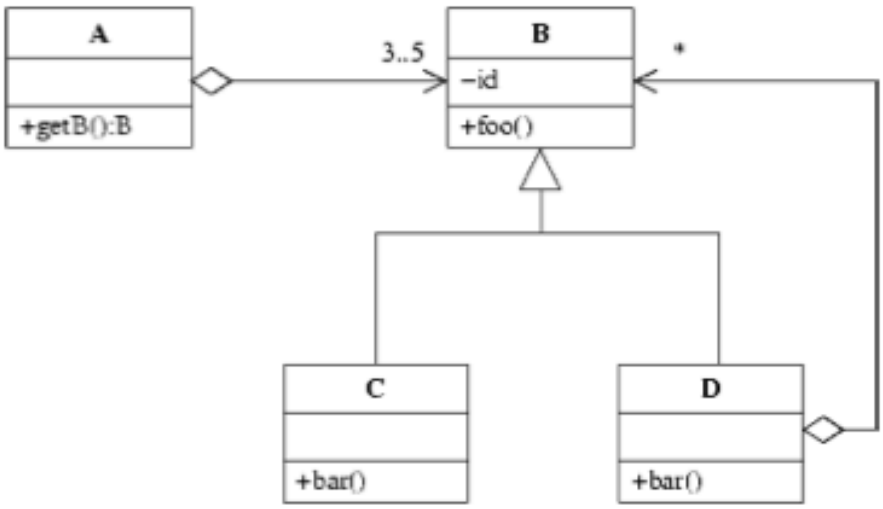
[D] Előfordulhat, hogy *Z* egy *W* interfészű objektumot sem aggregál, mert a '*' megengedi a nullát is.

[C] *S* ismeri a *W* interfészt, mert kapcsolatban áll egy megvalósításával.

[A] *S* nem látja *X get* metódusát, mert a metódus *protected*.

[C] *S* ismeri *X*-t, mert ugyanazt az interfészt valósítják meg.

[D] *Z add* metódusa kaphat paraméterül *Y*-t, mert *Y* (közvetetten) megvalósítja a *W* interfészt.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[B] *A* meghívhatja *C bar( )* metódusát, mert *B* minden leszármazottjában van *bar( )*

[B] *D* ismeri *C bar( )* metódusát, mert közös az ősük.

[E] *A* látja *B id* attribútumát, mert a kapcsolat *B*-ből *A*-ba irányított.

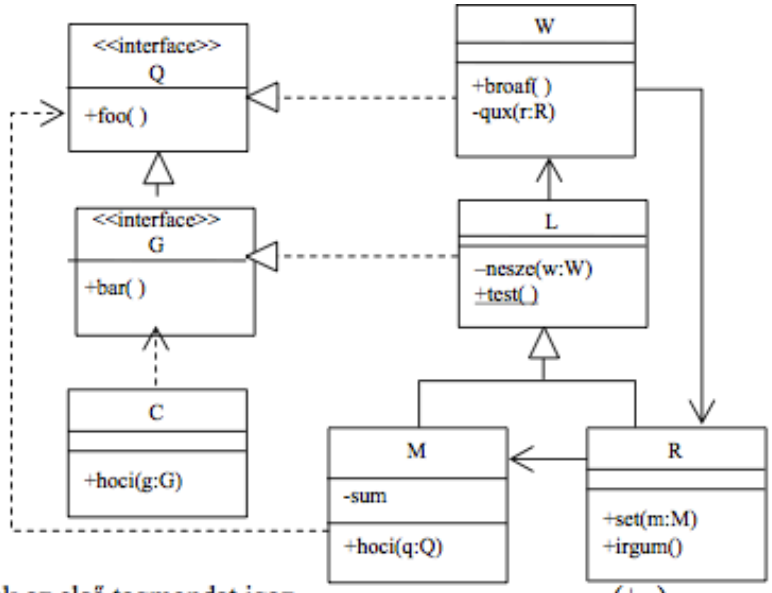
[E] *C* helyettesíthető *B*-vel, mert *B* a *C* leszármazottja.

[B] *A getb( )* metódusa nem adhat vissza *C* osztályú objektumot, mert *C* nem ismeri *B*-t.

[B] *C* helyettesíthető *D*-vel, mert *D* ismeri *C* ősét.

[C] *D* nem látja *A getB( )* metódusát, mert mind *A,* mind *D* tartalmazhat *B*-t.

[C] *B foo( )* metódusa látja *B id* attribútumát, mert *foo( )* publikus.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **M hoci(q:Q)** függvénye meghívhatja egy paraméterül kapott **W broaf()** metódusát, mert a **broáf** metódus statikus.

[B] **R set(m:M)** metódusa kaphat paraméterül **L** objektumot, mert **M** az **L** leszármazottja.

[B] **L nesze(w:W)** metódusa meghívhatja a paraméterül kapott objektum **qux(r:R)** metódusát, mert mindkét metódus privát.

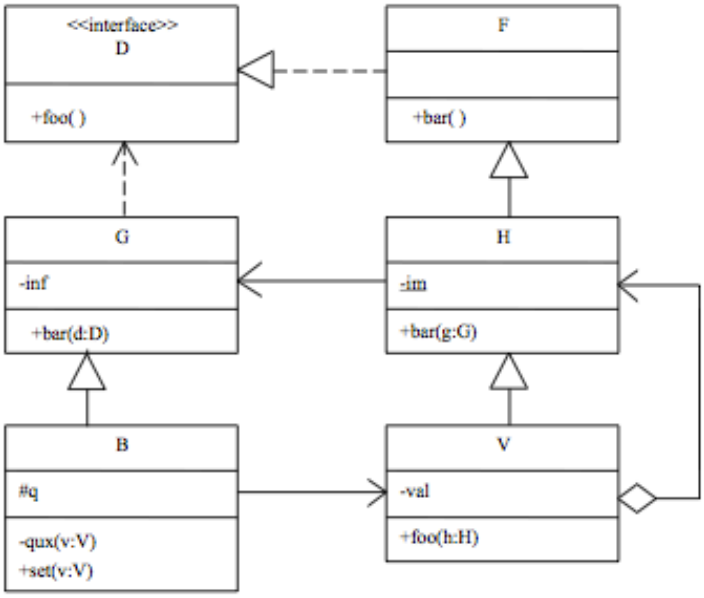
[B] **W** bárhol helyettesíthető **L**-lel, mert mindketten megvalósítják a **Q** interfészt.

[E] **R**-nek nincs **foo()** szignatúrájú metódusa, mert nem valósítja meg a **G** interfészt.

[A] **C hoci(g:G)** metódusa kaphat paraméterül **M** objektumot, mert **M hoci(q:Q)** metódusa is kaphat paraméterül **C**-t.

[B] **L nesze(w:W)** metódusa nem hívhatja meg a **test()** metódust, mert a **test()** statikus.

[A] **W qux(r:R)** metódusából bármikor meghívható a paraméter **irgum()** metódusa, mert a két osztály nem függ egymástól.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **G bar(d:D)** metódusa kaphat paraméterül **B** objektumot, mert **G** a **B** leszármazottja

[B] **H bar(g:G)** metódusa kaphat paraméterül **V** objektumot, mert **V** megvalósítja a **D** interfészt.

[B] **B qux(v:V)** metódusa módosíthatja a paraméter **val** attribútumát, mert mind a metódus, mind az attribútum privát.

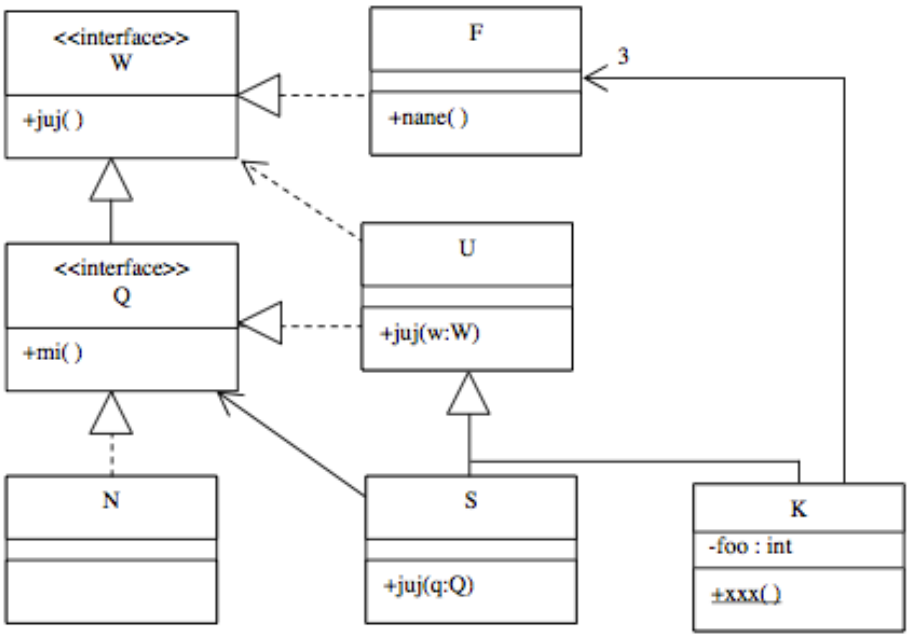
[E] **H bar(g:G)** metódusa nem módosíthatja az **im** attribútumot, mert az attribútum konstans.

[E] **B** objektum nem hívhatja meg egy **V** objektum **foo()** metódusát, mert **V**-nek nincs ilyen szignatúrájú metódusa.

[E] **G bar(d:D)** metódusa meghívhatja egy paraméterül kapott **F** objektum **bar()** metódusát, mert a két metódus azonos szignatúrájú

[B] **B set(v:V)** metódusa nem módosíthatja a **q** attribútmot, mert a láthatóságuk különböző.

[E] **B**-nek van **foo()** szignatúrájú metódusa, mert **B** megvalósítja a **D** interfészt



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] **K** helyettesíthető **S**-sel, mert közös ősük **U**.

[A] **K juj(w:W)** metódusa kaphat paraméterül **S**-t, mert **K** az **F** leszármazottja.

[B] **K xxx()** metódusa módosíthatja bármely **K** objektum **foo** attribútumát, mert a metódus statikus.

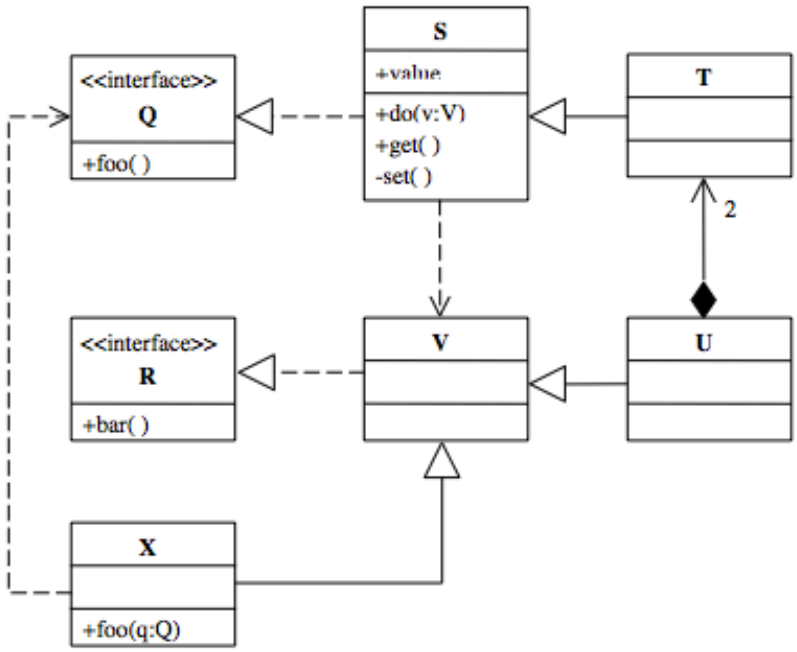
[B] **F** nem implementálja a **juj()** metódust, mert nem **U** leszármazottja.

[C] **S juj(q:Q)** metódusában meghívható egy paraméterül kapott **N** objektum **mi()** metódusa, mert **N** megvalósítja a **W** interfészt.

[E] **S**-nek nincs **juj(w:W)** metódusa, mert a **juj(q:Q)** metódusnak ugyanaz a szignatúrája.

[E] **F** helyettesíthető **U**-val, mert **K** mindkettejük leszármazottja.

[E] **U juj(w:W)** metódusából meghívhatjuk egy paraméterül kapott **F nane()** metódusát, mert **F** megvalósítja a **Q** interfészt.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[A] **S** létrehozhat **V** osztályú objektumot, mert **V** függ az **S**-től.

[D] **X foo(q:Q)** metódusa kaphat paraméterül **T**-t, mert **T** megvalósítja a **Q** interfészt.

[B] **X foo(q:Q)** metódusa meghívhatja a paraméterül kapott **S get()** metódusát, mert **S** megvalósítja a **Q** interfészt.

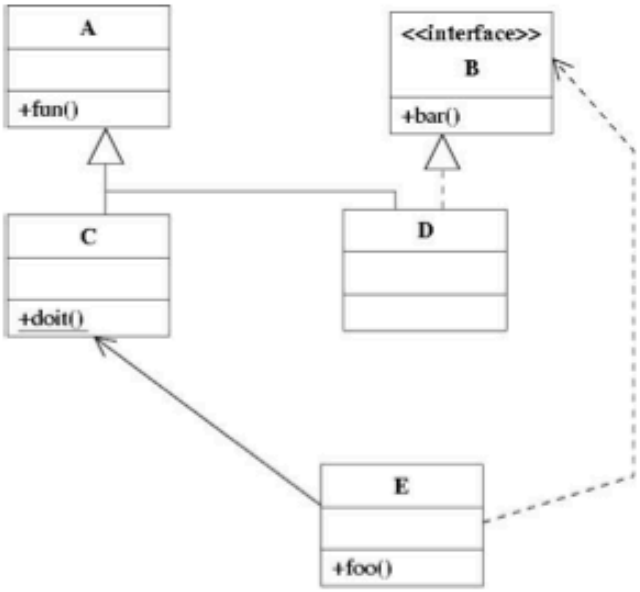
[D] **T**-ből legalább kétszer annyi példány van, mint U-ból, mert egy **T** példány nem tartozhat két **U**-hoz.

[B] **T** meghívhatja **U bar()** metódusát, mert **U**-nak van **bar** () metódusa.

[A] **X** meghívhatja egy **Q** interfészű objektum **foo( )** metódusát, mert **X** implementálja **Q**-t

[C] **V** helyettesíthető **U**-val, mert mindketten megvalósítják az **R** interfészt

[B] **S set( )** metódusa nem módosíthatja a **value** attribútumot, mert a láthatóságuk különböző



A - csak az első tagmondat igaz
B - csak a második tagmondat igaz
C - mindkét tagmondat igaz, de a következtetés hamis
D - mindkét tagmondat igaz és a következtetés is helyes
E - egyik tagmondat sem igaz
(**F** - az ki van zárva, mert nem tudom)

[A] Egy `A` osztályú objektum helyettesíthető `C` osztályú objektummal, mert az `A` osztály `C` leszármazottja.

[B] Egy `C` osztályú objektum helyettesíthető `D` osztályúval, mert van közös ősük.

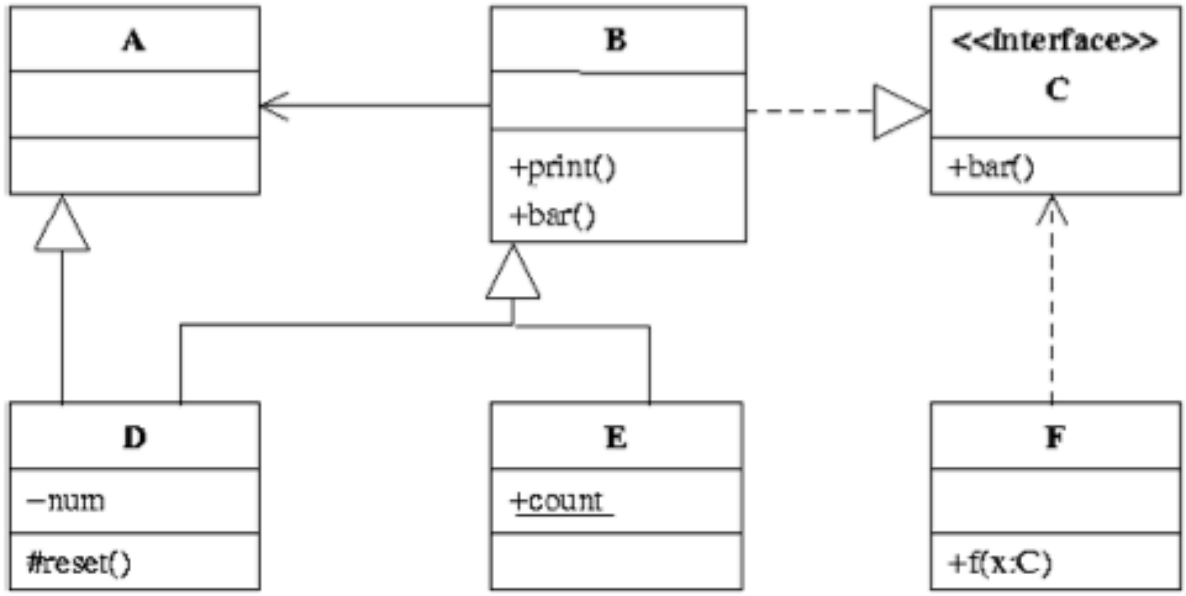
[E] Egy `C` osztályú objektum meg tudja hívni egy `E` osztályú objektum `foo()` metódusát, mert a metódus privát.

[B] Egy `E` osztályú objektum létrehozhat egy `D` osztályú objektumot, mert `D` megvalósítja `B`-t.

[D] Egy `E` objektum egy paraméterül kapott `D` osztályú objektumnak meg tudja hívni a `bar()` metódusát, mert ismeri a `D` által megvalósított `B` interfészt.

[B] Egy `E` objektum nem tudja meghívni `C` `doit()` metódusát, mert a metódus osztály-metódus.

[C] Egy `D` osztályú objektum nem ismeri `E`-t, mert az `E` osztály nem `A` leszármazottja.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[B] ***D*** helyettesíthető ***E***-vel, mert közös az ősük.

[B] ***E*** ismeri ***D***-t, mert egyik őse (***B***) ismeri ***D*** egyik ősét (***A***-t).

[E] ***F foo( )*** metódusa nem kaphat paraméterül ***D***-t, mert ***D*** nem valósítja meg ***C*** interfészt.

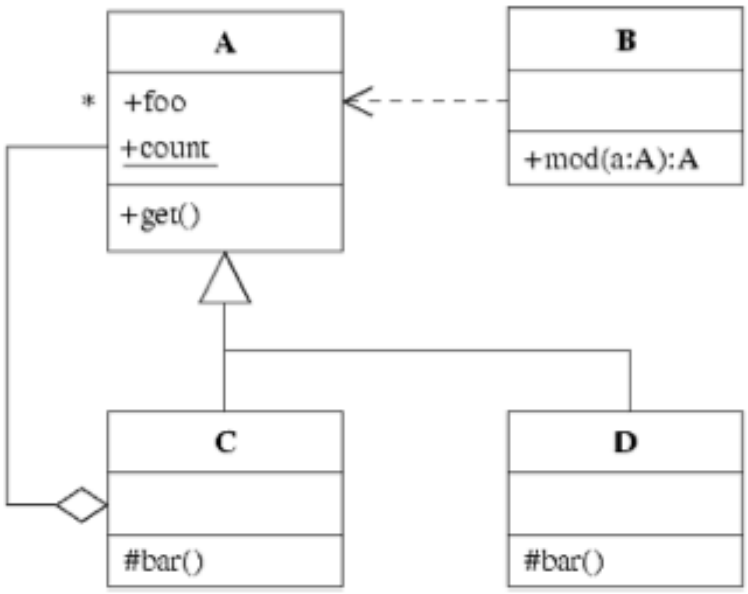
[E] ***A*** ismeri ***B***-t, mert ***B*** aggregálja ***A***-t.

[D] ***F*** meghívhatja ***B bar( )*** metódusát, mert ***B*** megvalósítja a ***C*** interfészt.

[B] ***A*** megvalósítja ***C*** interfészt, mert ***B***-vel közös leszármazottja van.

[B] ***E***-t nem lehet példányosítani, mert van statikus attribútuma.

[B] ***D reset( )*** metódusából nem érjük el ***D num*** attribútumát, mert a ***num*** privát.



A - csak az első tagmondat igaz
B - csak a második tagmondat igaz
C - mindkét tagmondat igaz, de a következtetés hamis
D - mindkét tagmondat igaz és a következtetés is helyes
E - egyik tagmondat sem igaz
(**F** - az ki van zárva, mert nem tudom)

[E] `A` osztályú objektum létrehozhat `B` osztályút, mert `A` függ `B`-től.

[B] `B` nem látja `A` `foo` attribútumát, mert `A`-ban van publikus metódus.

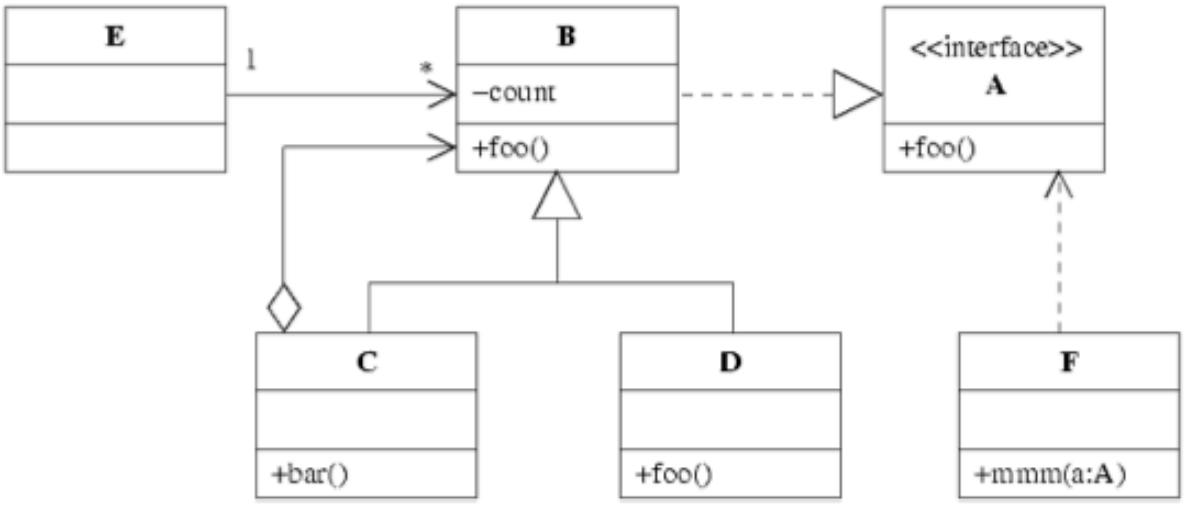
[E] `C` osztályú objektum nem tartalmazhat `C` osztályút, mert `C` `bar()` metódusa privát.

[D] `B` osztályú objektum `mod()` metódusa kaphat `C` osztályú objektumot paraméterül, mert `C` `A` leszármazottja.

[D] `B` az `A` `count` attribútumát `A` osztályú objektum létrehozása nélkül is módosíthatja, mert az attribútum statikus.

[E] `D` nem tartalmazhat `bar()` metódust, mert az ábrán többszörös öröklődés szerepel.

[B] `C` ismeri `D`-t, mert `B` ismeri az ősüket.



A - csak az első tagmondat igaz
B - csak a második tagmondat igaz
C - mindkét tagmondat igaz, de a következtetés hamis
D - mindkét tagmondat igaz és a következtetés is helyes
E - egyik tagmondat sem igaz
(**F** - az ki van zárva, mert nem tudom)

[B] `D` osztály nem definiálhatja felül a `foo()` metódust, mert nem ismeri az `A` interfészt.

[D] `E` osztályú objektum nem hívhatja meg `C` osztályú objektum `bar()` metódusát, mert csak a `B` osztályt ismeri.

[B] `C` osztály helyettesíthetõ `D`-vel, mert közös az õsük.

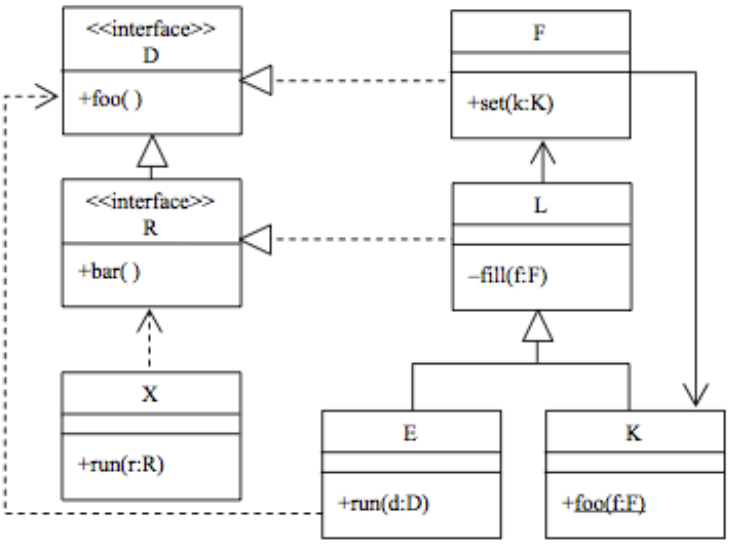
[B] `E` osztály ismeri az `A` interfészt, mert ismeri utóbbi egy megvalósítását.

[A] `C` osztály nem ismeri `D` osztályt, ezért `C` osztályú objektum nem hívhatja meg `D` osztályú objektum `foo()` metódusát.

[D] `F` osztály `mmm(a:A)` metódusa paraméterül kaphat `D` osztályú objektumot, mert utóbbi megvalósítja az `A` interfészt.

[A] `C` osztály nem látja `B` osztály `count` attribútumát, mert az attribútum protected.

[E] `F` osztályú objektum tartalmazhat tetszőleges számú `A` interfészű objektumot, mert `F` megvalósítja `A` interfészt.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] **X run(r:R)** metódusa kaphat paraméterül **F** osztályú objektumot, mert **X** függ **R**-től.

[E] **K**-nak nincs **foo()** szignatúrájú metódusa, mert **K**-t nem lehet példányosítani.

[E] **L** bárhol helyettesíthető **F**-fel, mert mindketten megvalósítják az **R** interfészt.

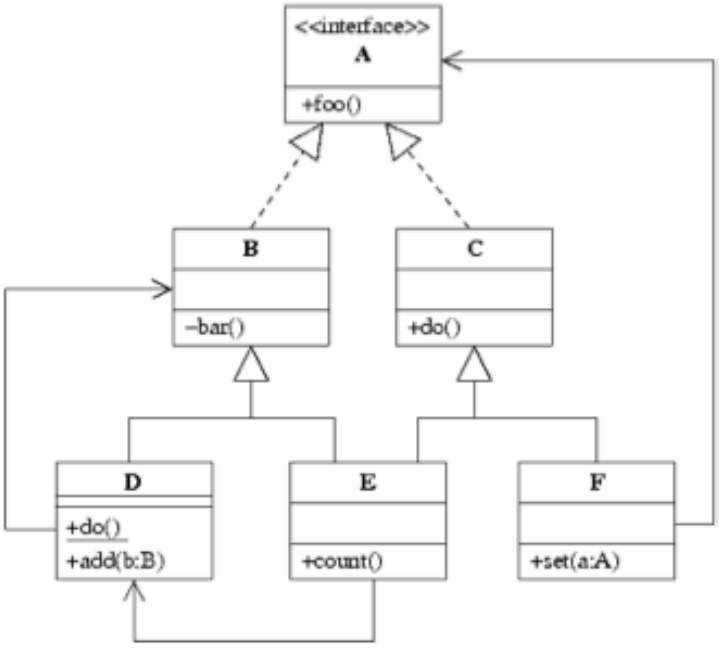
[B] **L** nem helyettesíthető **E**-vel, mert **L**-nek van privát metódusa.

[B] **X run(r:R)** metódusa nem kaphat paraméterül **K** objektumot, mert **K**-nak van statikus metódusa.

[B] **K foo(f:F)** metódusa nem hívhatja meg a paraméter **foo( )** metódusát, mert az utóbbi metódus nem statikus.

[B] **E** bárhol helyettesíthető **K**-val, mert van közös ősük.

[C] **F set(k:K)** metódusa nem hívhatja meg egy paraméterül kapott **K fill(f:F)** metódusát, mert **K** függ **F**-től.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[D] ***B*** bárhol helyettesíthető ***E***-vel, mert az ***E*** a ***B*** leszármazottja.

[E] ***C*** bárhol helyettesíthető ***D***-vel, mert a ***D*** a ***C*** leszármazottja.

[D] ***F set(a:A)*** függvénye kaphat paraméterül ***D***-t, mert a ***D*** megvalósítja az ***A*** interfészt.

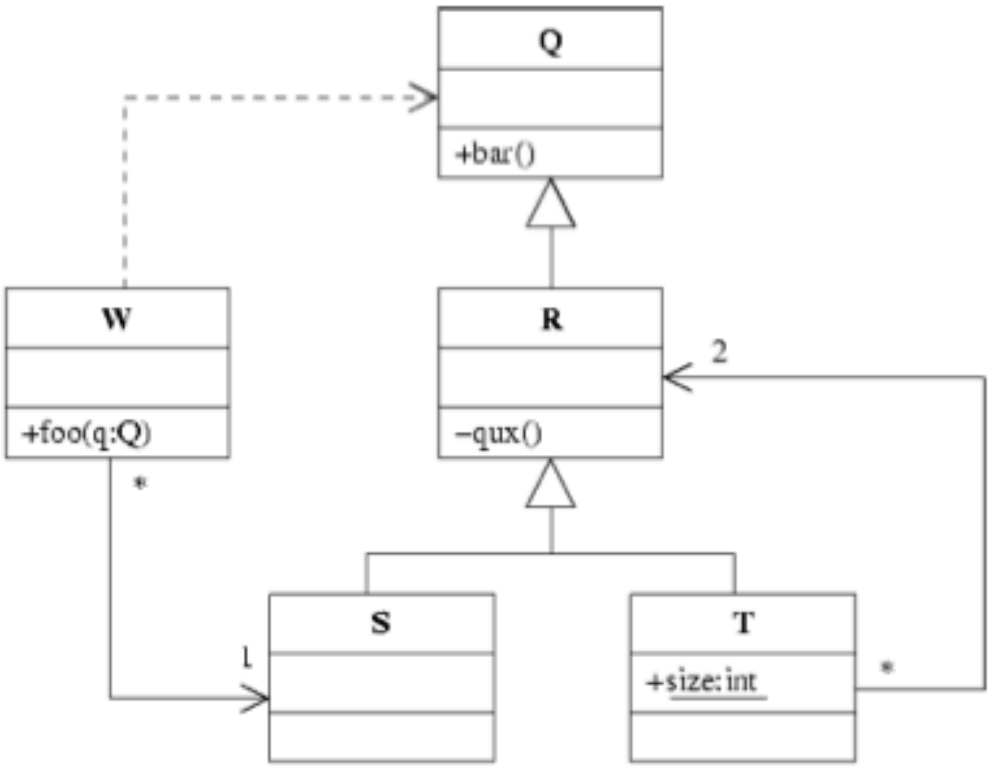
[B] ***D add(b:B)*** függvénye meghívhatja egy paraméterül kapott ***E count()*** metódusát, mert ***E*** megvalósítja az ***A*** interfészt.

[C] ***E*** meghívhatja egy ***D add(b:B)*** metódusát, mert közös az ősük.

[A] ***E*** nem hívhatja meg egy ***D bar()*** metódusát, mert a metódus *protected*.

[B] ***F*** meghívhatja egy ***D add(b:B)*** metódusát, mert ***E*** egyszerre a ***B*** és a ***C*** osztály leszármazottja.

[E] ***D add(b:B)*** metódusából nem hívhatjuk meg ***D do()*** metódusát, mert a metódus absztrakt.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[D] *Q* helyettesíthető *S*-sel, mert *S Q* leszármazottja.

[E] *T* helyettesíthető *R*-rel, mert *R T* leszármazottja.

[D] *T size* attribútuma *T* példányosítása nélkül is elérhető, mert az attribútum statikus.

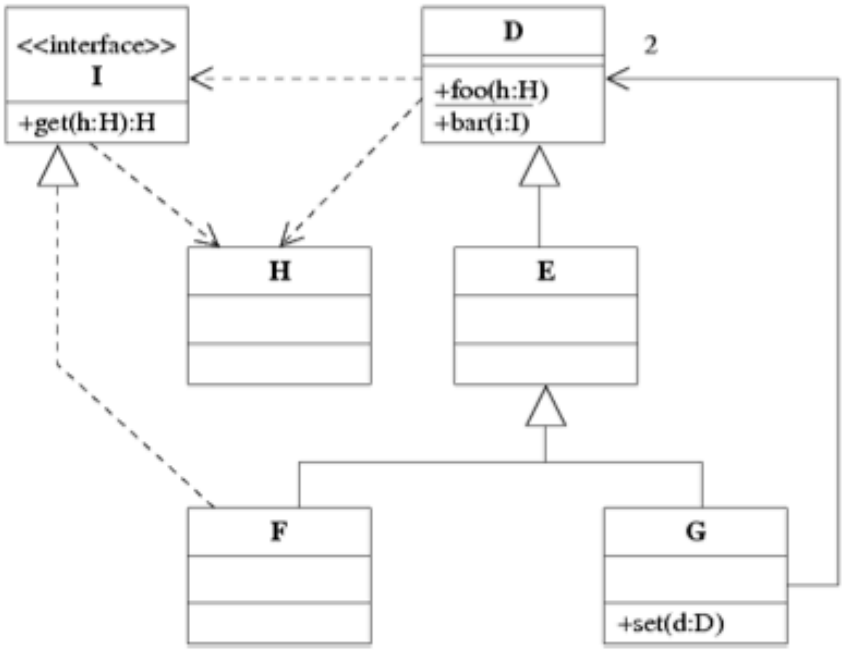
[B] *T* nem hívhatja meg *R bar* metódusát, mert nem ismeri *R* ősét.

[C] *W foo* metódusa kaphat paraméterül *T*-t, mert *T*-nek *S*-sel közös az őse.

[B] *T* legalább kettő *R*-t ismer, mert a '*' számosságban a 'sehány' is benne van.

[E] *W* meghívhatja *S qux* metódusát, mert *W* ismeri *R*-t.

[B] *S* meghívhatja *W foo* metódusát, mert *S*-t több *W* is ismerheti.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[B] `F` helyettesíthető `E`-vel, mert `F` az `E` leszármazottja.

[B] `G` helyettesíthető `F`-fel, mert `D` közös ősük.

[E] `H` meghívhatja `D` `foo` metódusát, mert a metódus absztrakt.

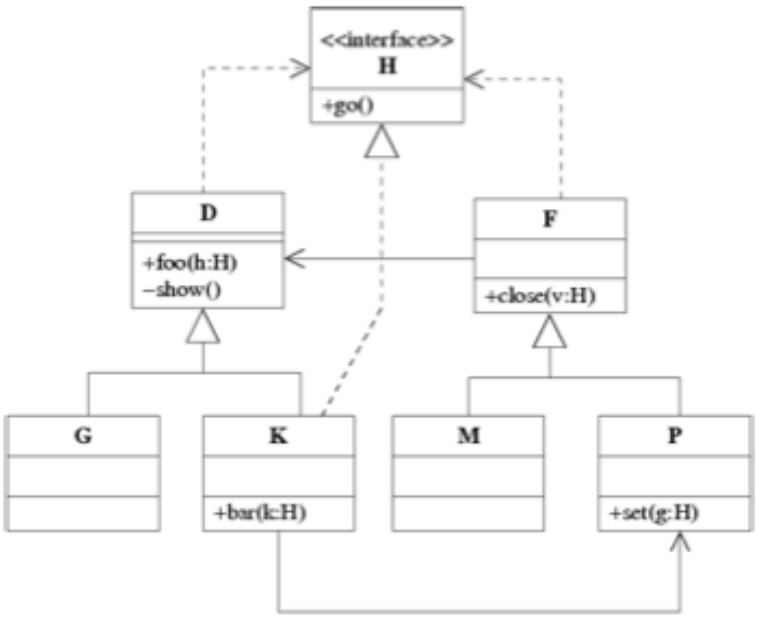
[C] `D` `bar` metódusa kaphat paraméterül `F`-et, mert `F` a `D` leszármazottja.

[E] `D` `foo` metódusa kaphat paraméterül `I`-t, mert az `I` megvalósítja `H`-t.

[B] `G` konstruktora pontosan kétszer köteles meghívni `D` `bar` metódusát, mert pontosan két `D`-t ismer.

[E] `G` `set` metódusa nem hívhatja meg a paraméterül kapott `D` objektum `bar` metódusát, mert a metódus statikus.

[B] `F` meghívhatja `G` `set` metódusát, mert mindketten függnek az `I` interfésztől.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **F** meghívhatja **D** **show()** metódusát, mert a metódus statikus.

[A] **M** meghívhatja **D** **foo(h:H)** metódusát, mert mindketten implementálják a **H** interfészt.

[B] **G** helyettesthető **D**-vel, mert **G** a **D** leszármazottja.

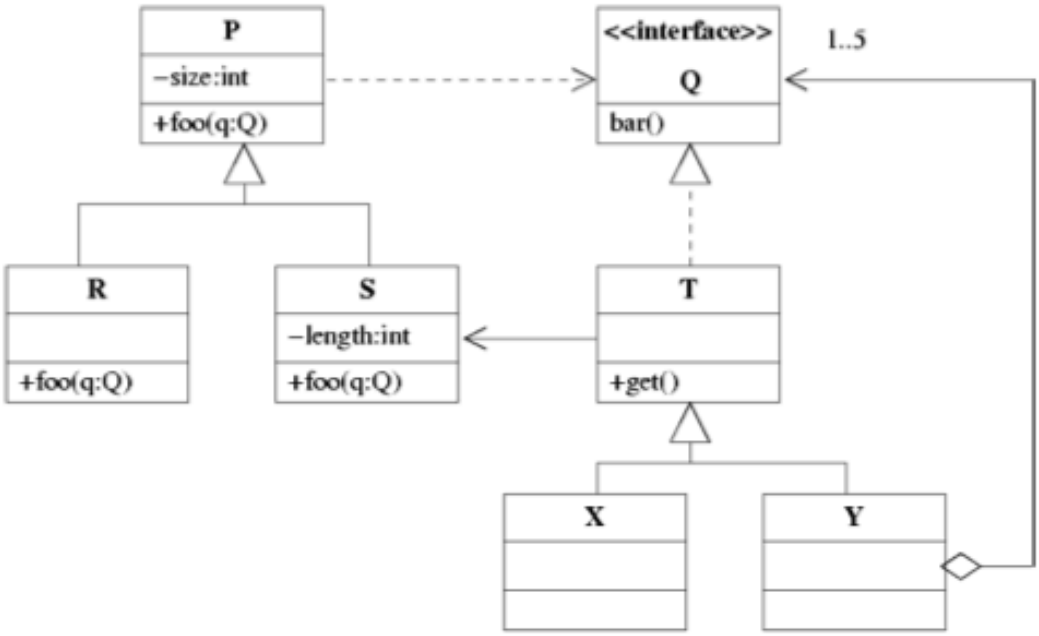
[C] **K** **bar(k:H)** metódusából átadhatjuk a **k** paramétert **P** **set(g:H)** metódusának, mert **K** implementálja **H**-t.

[C] **F** **close(v:H)** metódusa kaphat paraméterül **K**-t, mert **K** a **D** leszármazottja.

[B] **G** helyettesíthető **K**-val, mert van közös ősük.

[A] **K** meghívhatja **P** **set(g:H)** metódusát, mert van közös ősük.

[D] **K** **bar(k:H)** metódusából meghívhatjuk a **k** paraméteren a **go()** metódust, mert **K** ismeri a **H** interfészt.



A - csak az első tagmondat igaz	(+ -)
B - csak a második tagmondat igaz	(- +)
C - mindkét tagmondat igaz, de a következtetés hamis	(+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes	(+ + +)
E - egyik tagmondat sem igaz	(- -)

[B] **S** bárhol helyettesíthető **P**-vel, mert **S** a **P** leszármazottja.

[B] **T** nem hívhatja meg **S foo(q:Q)** metódusát, mert **T** nem ismeri **P**-t.

[E] **R foo(q:Q)** metódusa nem kaphat paraméterül **X**-et, mert **X** nem valósítja meg a **Q** interfészt.

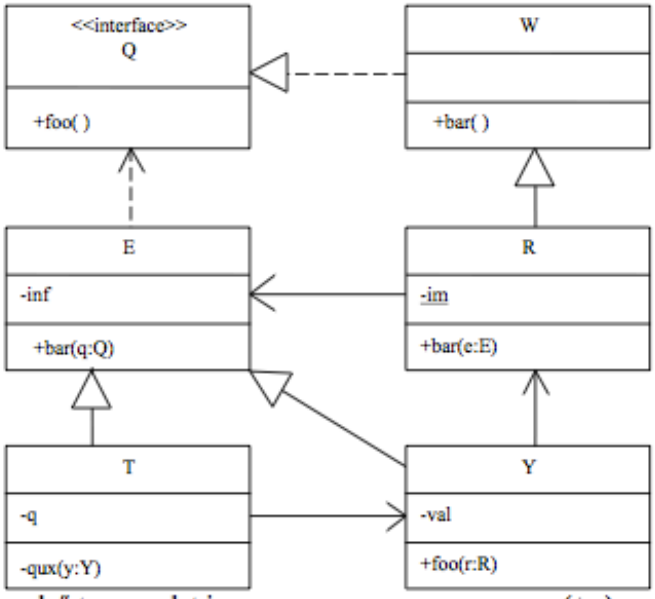
[B] **X** nem hívhatja meg **S foo(q:Q)** metódusát, mert az **S-T** asszociáció **T** felé nem navigálható.

[E] **P** tartalmaz **bar()** metódust is, mert **P** megvalósítja **Q**-t.

[E] **S foo(q:Q)** metódusában nem hívhatjuk meg a **q** paraméter **bar()** metódusát, mert **S** nem ismeri a **Q** interfészt.

[D] **Y** nem olvashatja **S length** attribútumát, mert az attribútum privát.

[B] **X** bárhol helyettesíthető **Y**-nal, mert van közös ősük.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[C] **Y bar(q:Q)** metódusa kaphat paraméterül **R** objektumot, mert **Y** függ **R**-től.

[B] **T qux(y:Y)** metódusa módosíthatja a paraméter **val** attribútumát, mert a metódus privát.

[E] **E** bárhol helyettesíthető **R**-rel, mert azonos az interfészük.

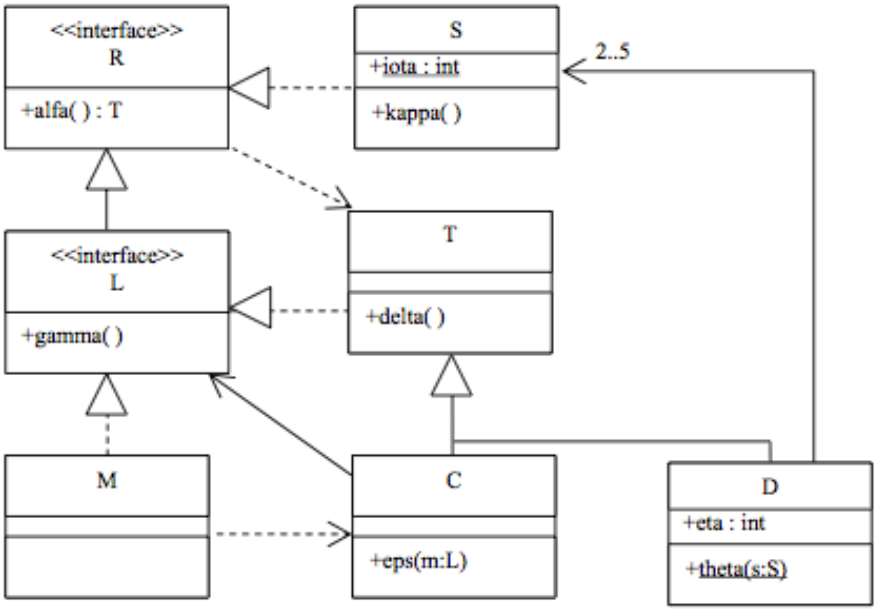
[C] **Y foo(r:R)** metódusa nem módosíthatja a paraméter **im** attribútumát, mert az attribútum statikus.

[E] **E bar(q:Q)** metódusa kaphat **E** objektumot paraméterül, mert az **E** megvalósítja a **Q** interfészt.

[E] **E bar(q:Q)** metódusa nem hívhatja meg egy paraméterül kapott **W foo()** metódusát, mert **W**-nek nincs ilyen szignatúrájú metódusa.

[B] **R bar(e:E)** metódusa nem kaphat paraméterül **Y** objektumot, mert az **Y**-**R** asszociációban csak **Y** hívhatja **R**-t.

[B] **R** nem valósítja meg a **Q** interfészt, mert van olyan szignatúrájú metódusa, ami nem szerepel a **Q** metódusai között.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[A] **M alfa():T** metódusa visszaadhat **C** objektumot, mert **C** függ **M**-től.

[C] **D theta(s:S)** metódusa nem kaphat paraméterül **C** objektumot, mert **S** és **C** is megvalósítja az **R** interfészt.

[E] **C eps(m:L)** metódusa nem hívhatja meg a paraméter **gamma()** metódusát, mert az utóbbi metódus protected láthatóságú.

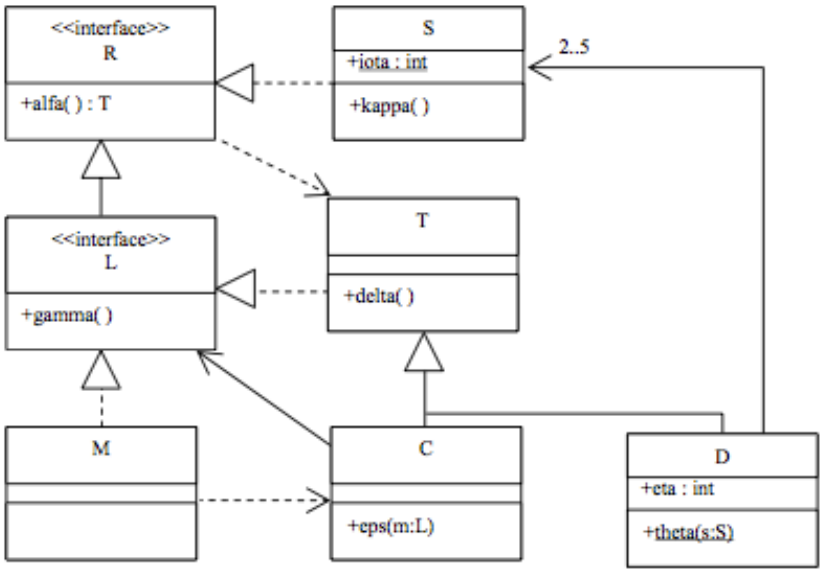
[B] **D theta(s:S)** metódusa nem módosíthatja a paraméter **iota** attribútumát, mert a **theta(s:S)** statikus.

[E] **S** nem valósítja meg az **alfa():T** szignatúrájú metódust, mert **S** nem függ **T**-től.

[B] **D theta(s:S)** metódusa legfeljebb 5-ször hívható meg, mert **D** objektum legfeljebb 5 **S**-sel állhat asszociációban.

[B] **M** bárhol helyettesíthető **C**-vel, mert mindketten megvalósítják az **R** interfészt.

[D] **T** osztálynak van **alfa():T** szignatúrájú metódusa, mert **T** megvalósítja az **R** interfészt.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] **M** bárhol helyettesíthető **C**-vel, mert mindketten megvalósítják az **R** interfészt.

[E] **C eps(m:L)** metódusa nem hívhatja meg a paraméter **gamma()** metódusát, mert az utóbbi metódus protected láthatóságú.

[C] **D theta(s:S)** metódusa módosíthatja a paraméter **iota** attribútumát, mert a **theta(s:S)** statikus.

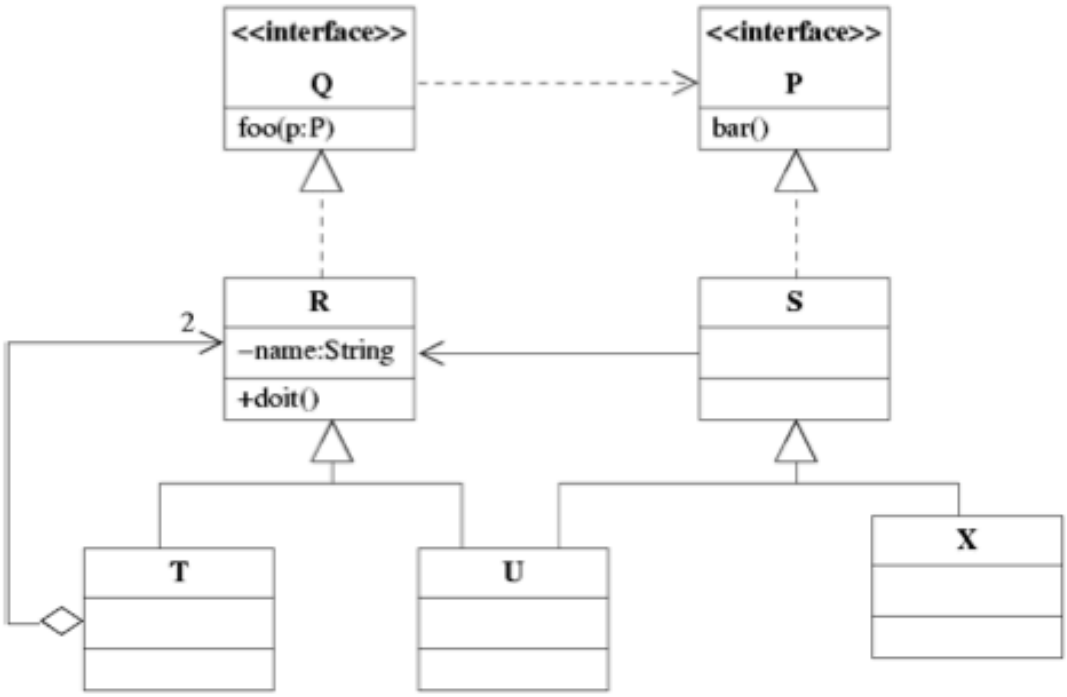
[E] **T** osztálynak nincs **alfa():T** szignatúrájú metódusa, mert **T** nem valósítja meg az **R** interfészt.

[A] **M alfa():T** metódusa visszaadhat **C** objektumot, mert **C** függ **M**-től.

[B] **D theta(s:S)** metódusa kaphat paraméterül **C** objektumot, mert **S** és **C** is megvalósítja az **R** interfészt.

[E] **S** nem valósítja meg az **alfa():T** szignatúrájú metódust, mert **S** nem függ **T**-től.

[B] **D theta(s:S)** metódusa legfeljebb 5-ször hívható meg, mert **D** objektum legfeljebb 5 **S**-sel állhat asszociációban.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **P** mindig helyettesíthető **Q**-val, mert **Q** a **P** leszármazottja.

[D] **T foo(p:P)** metódusa kaphat paraméterül **X**-et, mert **X** megvalósítja a **P** interfészt.

[B] **U foo(p:P)** metódusa nem kaphat paraméterül **U**-t, mert **R**-nek és **S**-nek nincs közös őse.

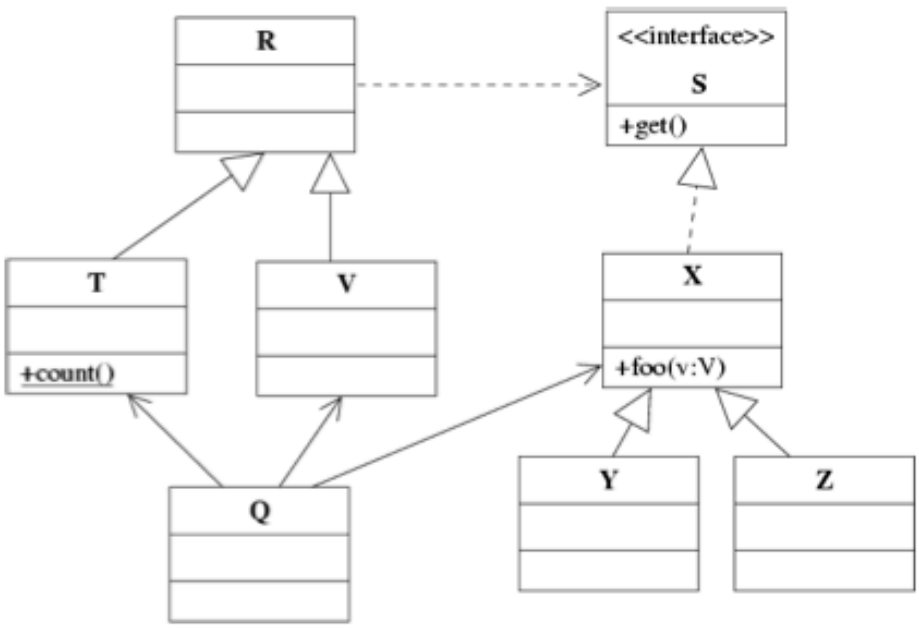
[B] **R foo(p:P)** metódusa nem hívhatja meg egy paraméterül kapott **S bar()** metódusát, mert **R** nem ismeri **S**-t.

[B] **T** nem tartalmazhat egyszerre egy **T** és egy **U** objektumot, mert az **U S** leszármazottja is.

[E] **X**-ből nem hozhatunk létre példányt, mert nincs asszociációban senkivel.

[D] **S** nem látja **R name** attribútumát, mert az attribútum privát.

[B] **X** bárhol helyettesíthető **T**-vel, mert **U**-val mindkettőnek van (külön-külön) közös őse.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **T** bárhol helyettesíthető **Q**-val, mert **Q** a **T** leszármazottja.

[E] **S** helyén bárhol állhat **R**, mert **R** megvalósítja az **S** interfészt.

[D] **S** helyén bárhol állhat **Y**, mert **Y** megvalósítja az **S** interfészt.

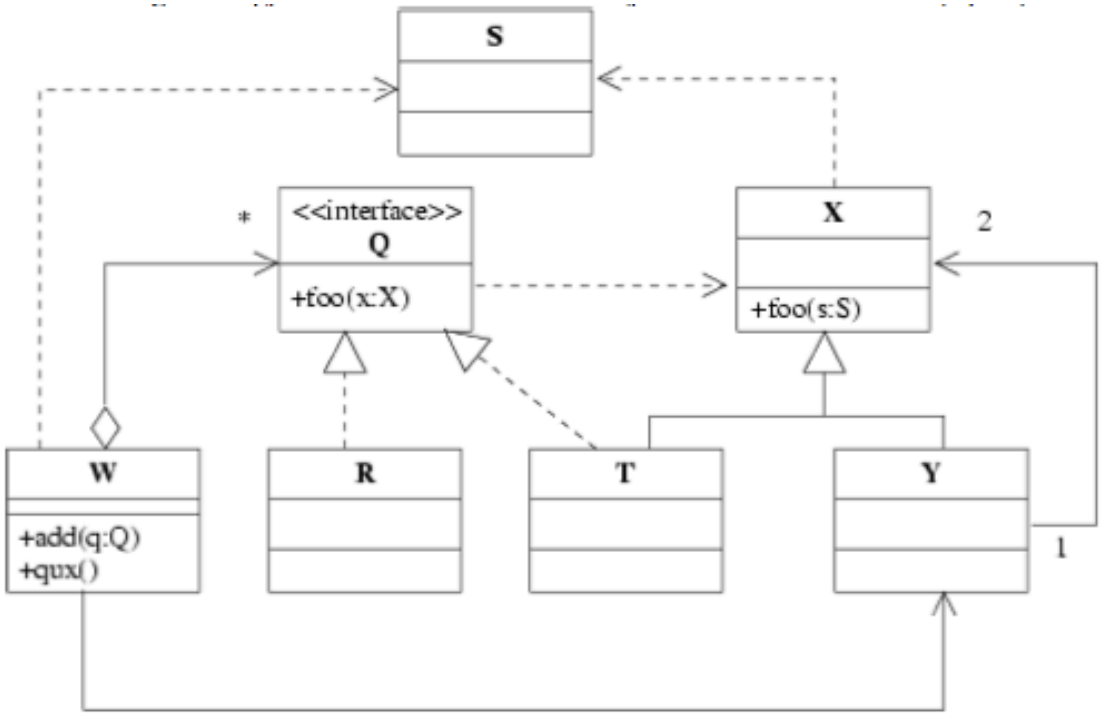
[B] **Z** helyén bárhol állhat **Y**, mert a két osztálynak van közös őse.

[B] **V** meghívhatja **X foo()** metódusát, mert a metódus publikus.

[B] **Q** meghívhatja **X foo(v:V)** metódusát **T** osztályú paraméterrel, mert **Q** ismeri mind **V**-t, mind **T**-t.

[E] **T** nem hívhatja meg egy **S**-et megvalósító objektum **get()** metódusát, mert nem ismeri az **S** interfészt.

[C] **Q** meghívhatja **T count()** metódusát, mert a metódus statikus.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[B] `T` helyettesíthető `R`-rel, mert közös interfészt implementálnak.

[E] `T` helyettesíthető `X`-szel, mert `X` a `T` leszármazottja.

[D] `W` `add` metódusa kaphat paraméterül `R`-t, mert `R` megvalósítja a `Q` interfészt.

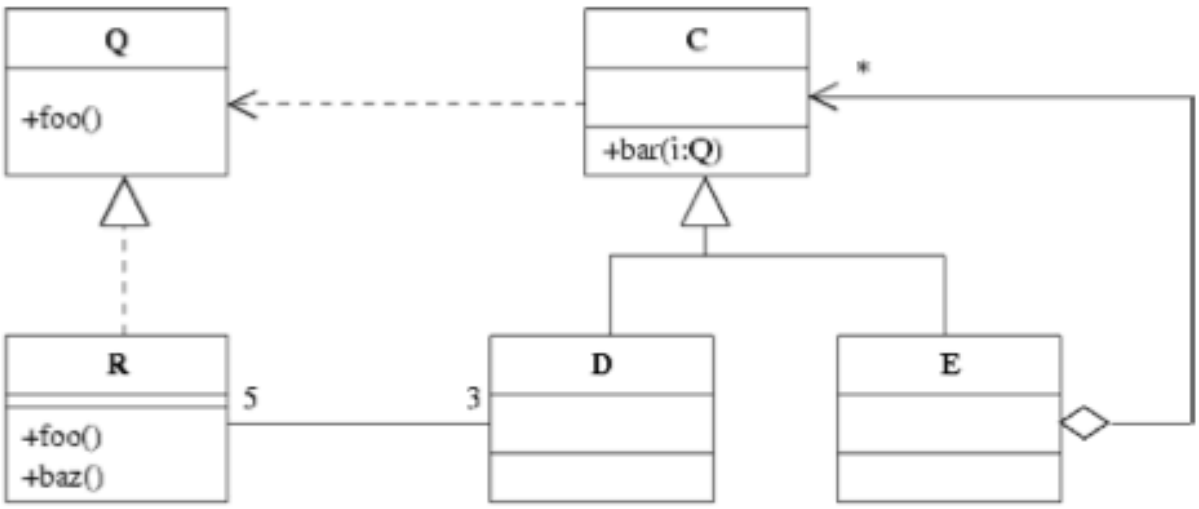
[E] `W` meghívhatja egy `T` objektum `foo(s:S)` metódusát, mert a metódus szerepel a `Q` interfészben.

[B] `W` nem hívhatja meg `Y` `foo(s:S)` metódusát, mert nem ismeri `X` osztályt.

[B] `Y` pontosan egy `X`-et ismer, mert a kapcsolatuk asszociáció.

[E] `T` `foo(x:X)` es `foo(s:S)` metódusa azonos, mert egyforma a szignatúrájuk.

[B] `Y` meghívhatja `W` `qux()` metódusát, mert mindketten függnek `S`-től.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)
(**F** - az ki van zárva, mert nem tudom)

[B] D helyettesíthető E-vel, mert közös az ősük.

[D] E bar metódusa kaphat paraméterül R-t, mert R megvalósítja a Q interfészt.

[A] Egy R pontosan 3 darab D-vel állhat kapcsolatban, mert az asszociáció irányított.

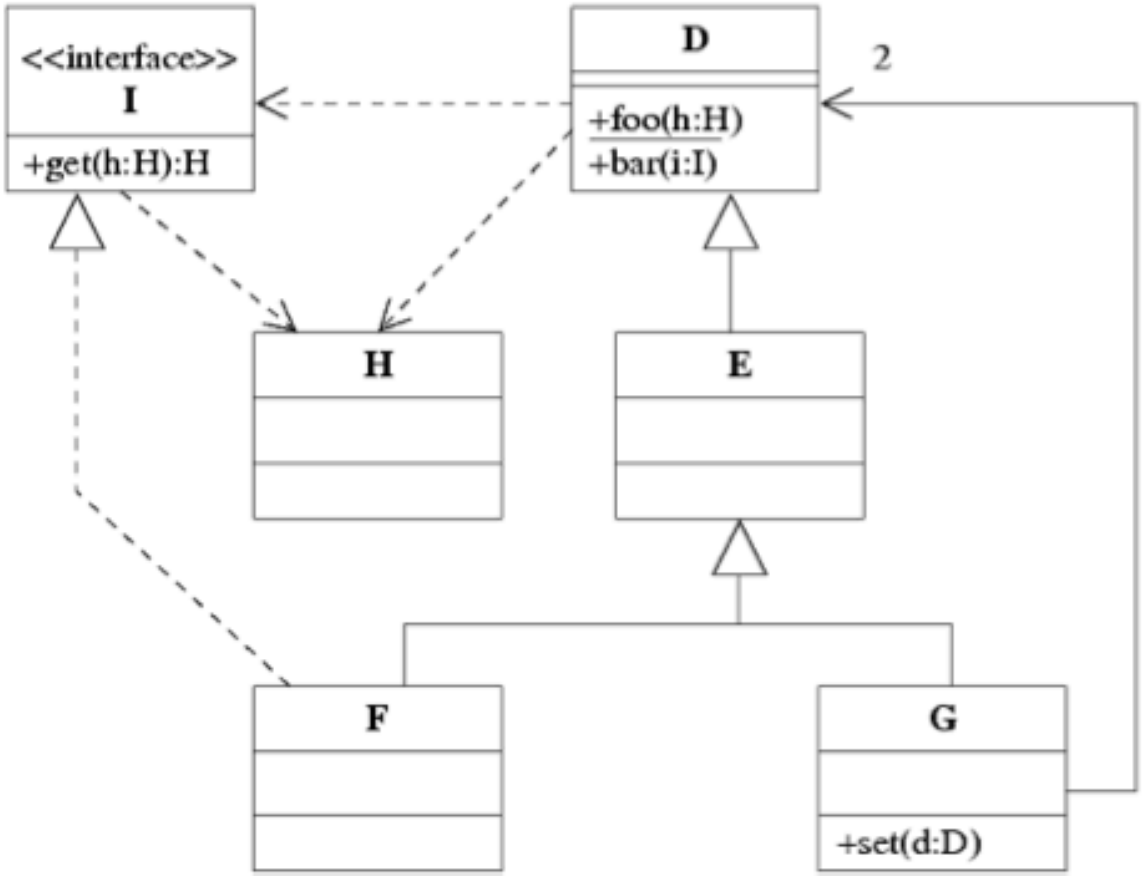
[D] E aggregálhat más E-ket, mert E C leszármazottja.

[B] R nem hívhatja meg D bar metódusát, mert nem ismeri D ősét.

[C] D létrehozhat R-t, mert ismeri a Q interfészt.

[B] E meghívhatja R baz metódusát, mert D közvetlenül ismeri R-t.

[E] C bar metódusa paraméterül kaphat D-t, mert C megvalósítja a Q interfészt.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] `F` helyettesíthető `G`-vel, mert `D` közös ősük.

[E] `F`-nek nincs mindig `get` metódusa, mert csak feltételesen örököl `I`-től.

[A] `H` nem hívhatja meg `D` `foo` metódusát, mert a metódus absztrakt.

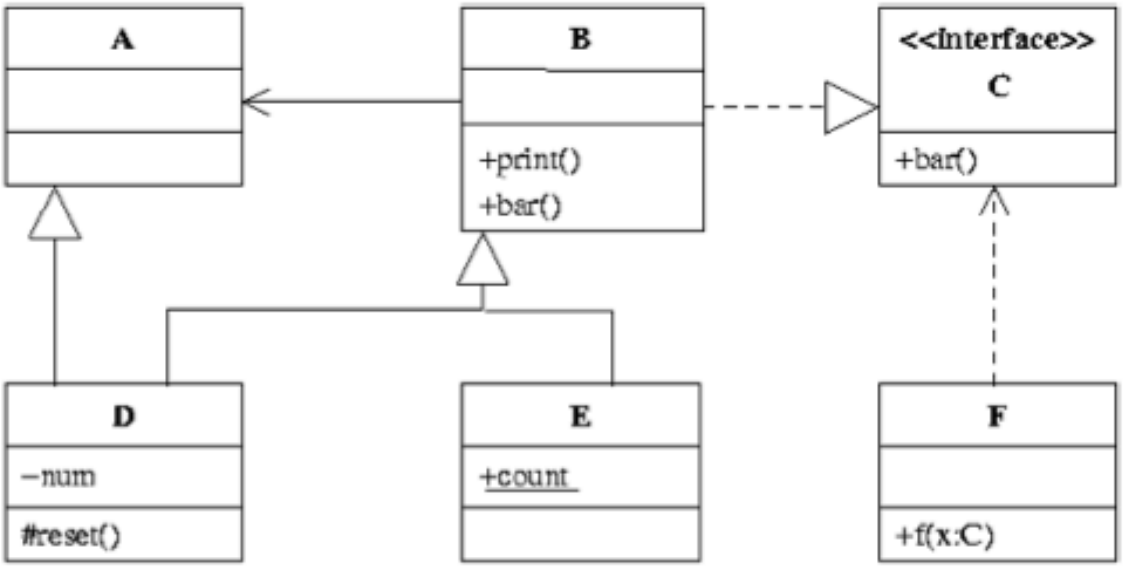
[C] `D` `bar` metódusa kaphat paraméterül `F`-et, mert `F` leszármazottja `D`-nek.

[A] `D` `foo` metódusa nem kaphat paraméterül `I`-t, mert `foo` nem osztály-metódus.

[C] `G` kétszer is meghívhatja `D` `bar` metódusát, mert `G` `D`-nek a leszármazottja.

[E] `G` nem tud `H` paramétert átadni `D` `foo` metódusának hívásakor, mert `G` nem ismeri `H`-t.

[B] `F` meghívhatja `G` `set` metódusát, mert mindketten függnek az `I` interfésztől.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] D helyettesíthető E-vel, mert van közös ősük.

[E] D ismeri E-t, mert D egyik őse (A) ismeri E ősét.

[D] F `f` metódusa kaphat paraméterül D-t, mert D megvalósítja a C interfészt.

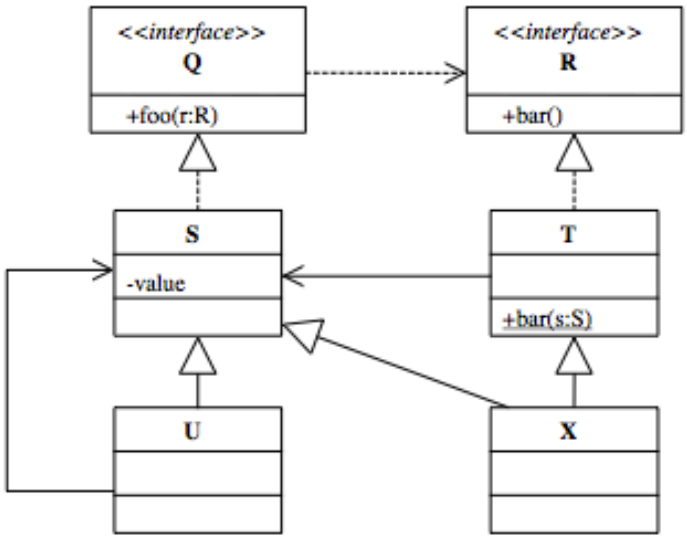
[A] B ismeri A-t, mert B komponense A.

[B] F hozzáfér E `count` attribútumához, mert E megvalósítja a C interfészt.

[B] A is megvalósítja C interfészt, mert van B-vel közös leszármazottja (D).

[A] E-nek van osztály-attribútuma, ezért E-t nem lehet példányosítani.

[B] D `reset` metódusából nem érjük el D `num` attribútumát, mert a `num` privát.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] **X** helyettesíthető **U**-val, mert mindketten megvalósítják a **Q** interfészt.

[C] **S foo(r:R)** metódusa kaphat paraméterül **X**-et, mert **X** az **S** leszármazottja.

[E] **X** módosíthatja egy **S value** attribútumát, mert az attribútum statikus.

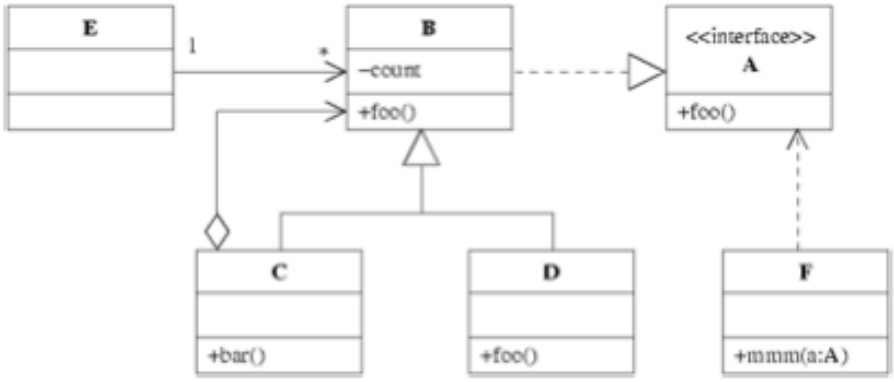
[B] **S** nem hívhatja meg egy **T bar( )** metódusát, mert az asszociáció **T**-ből **S**-be irányul.

[B] **S foo(r:R)** metódusa nem módosíthatja a **value** attribútumot, mert különböző a láthatóságuk.

[B] **U** módosíthatja a vele asszociációban levő **S** objektum **value** attribútumát, mert **S** az **U** ősosztálya.

[B] **T** nem hívhatja meg **S foo(r:R)** metódusát, mert az **R** interfész nem függ **Q**-tól.

[B] **T bar(s:S)** metódusa nem hívható meg egy **X** objektummal, mert **X**-nek nincs statikus attribútuma.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] D osztályból meghívhatjuk a C `bar()` metódusát, mert a C osztályú objektumnak lehet komponense D osztályú objektum.

[B] D osztály nem definiálhatja felül a `foo()` metódust, mert B nem absztrakt.

[B] F osztályú objektum `mmm(a:A)` metódusából meghívhatjuk a C `bar()` metódusát, mert C lehet az `mmm(a:A)` paramétere.

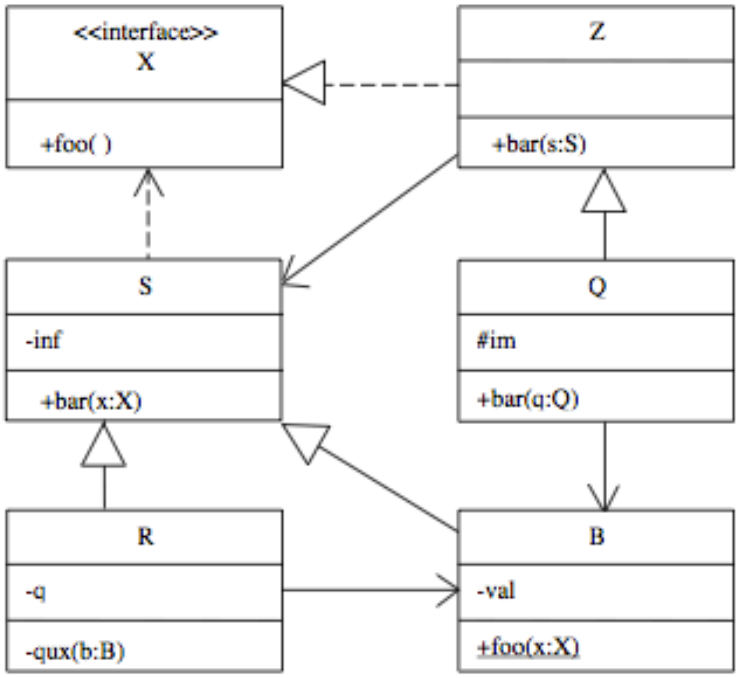
[B] C osztály helyettesíthetõ D-vel, mert mindketten megvalósítják az A interfészt.

[D] F osztály `mmm(a:A)` metódusa paraméterül kaphat D osztályú objektumot, mert utóbbi megvalósítja az A interfészt.

[A] E nem ismeri az A interfészt, ezért E nem hívhatja meg B `foo()` metódusát.

[E] E osztály ismeri az A interfészt, mert B helyettesíthető E-vel.

[D] E osztályú objektum nem hívhatja meg C osztályú objektum `bar()` metódusát, mert csak a B osztályt ismeri.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **R** helyettesíthető **S**-sel, mert **S** az **R** leszármazottja

[E] **R** helyettesíthető **B**-vel, mert mindketten megvalósítják az **X** interfészt

[A] **R** átadható paraméterül **Q** `bar(s:S)` metódusának, mert **Q** és **S** interfésze megegyezik.

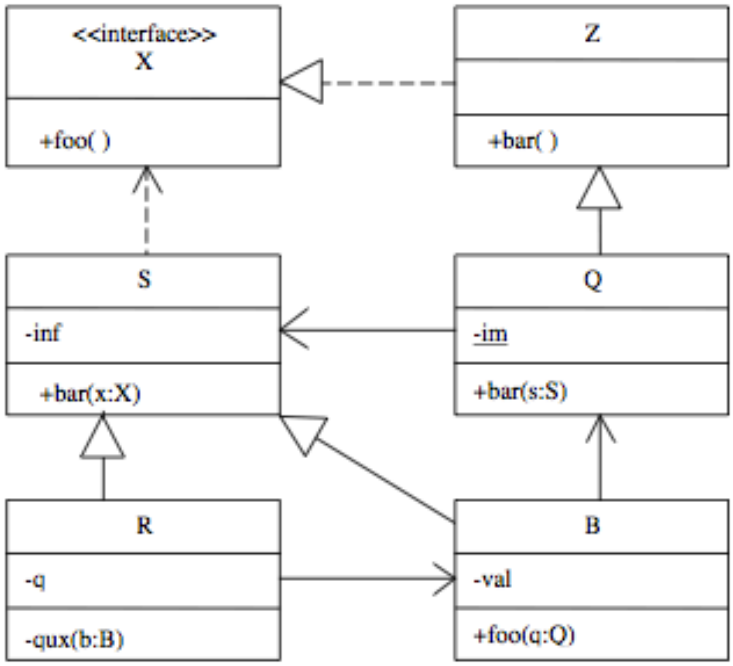
[D] **B** `foo(x:X)` metódusa nem látja a `val` attribútum értékét, mert az attribútum nem statikus.

[A] **Q** meghívhatja **S** `bar(x:X)` metódusát, mert mindketten megvalósítják az **X** interfészt.

[B] **B** interfésze tartalmaz `qux(b:B)` metódust, mert **B**-nek van **R**-rel közös őse.

[A] **Q** `bar(s:S)` metódusa nem módosíthatja az `im` attribútumot, mert az attribútum privát.

[B] **B** meghívhatja **R** `qux(b:B)` metódusát, mert a metódus paramétere **B** osztályú.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **S** helyettesíthető **Q**-val, mert **Q** az **S** leszármazottja

[A] **S** helyettesíthető **B**-vel, mert **B** megvalósítja az **X** interfészt

[A] **R** átadható paraméterül **Q** `bar(s:S)` metódusának, mert **Q** és **S** interfésze megegyezik.

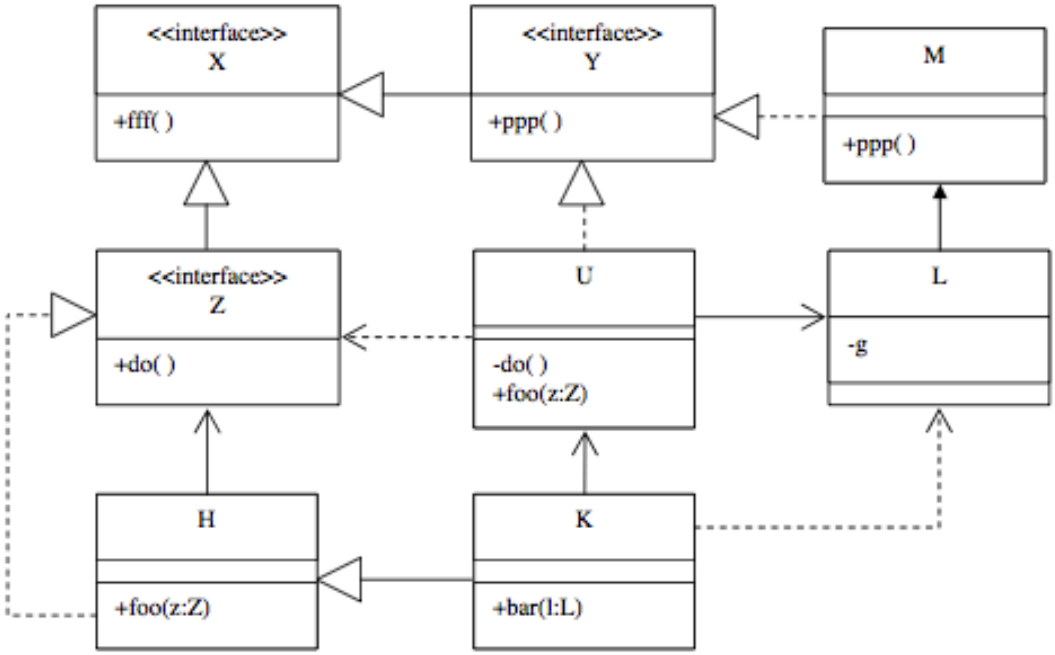
[B] **B** `foo(q:Q)` metódusa nem látja saját `val` attribútumának értékét, mert az attribútum privát.

[A] **Q** meghívhatja **S** `bar(x:X)` metódusát, mert mindketten megvalósítják az **X** interfészt.

[E] **B** interfésze tartalmazza `bar(s:S)` metódust, mert a metódus statikus.

[A] **Q** `bar()` metódusa nem módosíthatja az `im` attribútumot, ezért az attribútum konstans.

[E] **B**–nek nincs `bar(x:X)` metódusa, ezért nem függ az **X** interfésztől.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] **H** bárhol helyettesítheti **U**-t, mert mindketten megvalósítják az **X** interfészt.

[B] **H foo(z:Z)** metódusa meghívható egy **U**-val, mert **U** megvalósítja az **Y** interfészt.

[A] **U** nem hívhatja meg **K bar(l:L)** metódusát, mert **K** nem függ **L**-től.

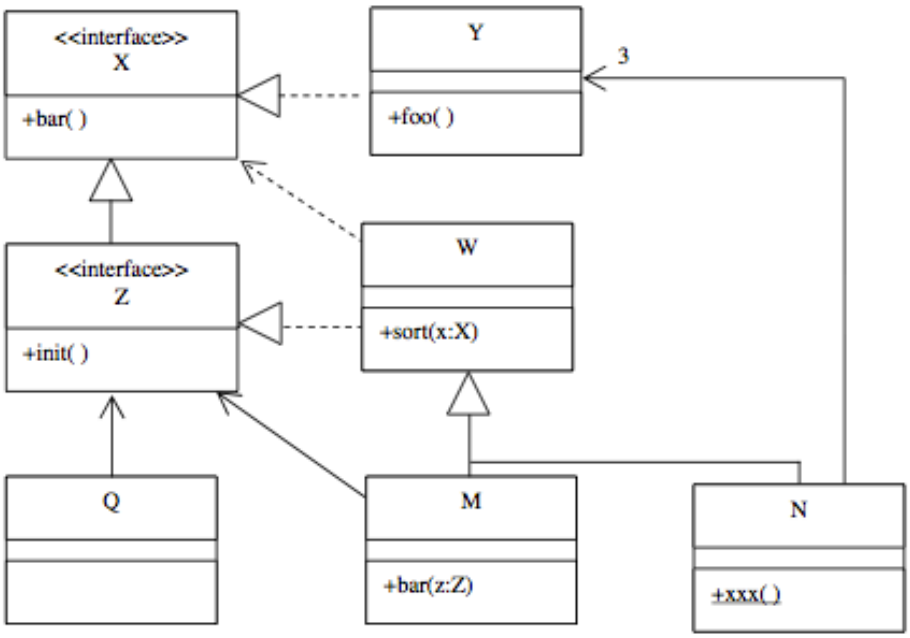
[A] **K** implementálja a **Z** interfészt, ezért **K** meghívhatja **U do( )** metódusát.

[B] **K** nem hozhat létre **L** objektumot, mert az **L g** attribútuma privát.

[C] **U foo(z:Z)** metódusa nem hívhatja meg egy paraméterül kapott **H foo(z:Z)** metódusát, mert **U** nem implementálja a **Z** interfészt.

[E] **K** bárhol helyettesítető **U**-val, mert **K** az **U** leszármazottja.

[A] **L** nem ismeri az **Y** interfészt, ezért **L** nem hívhatja meg **M ppp( )** metódusát.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] `Y` helyettesíthető `W`-vel, mert mindketten megvalósítják az `X` interfészt.

[C] `N` meghívhatja `Y` `foo( )` metódusát, mert `N` megvalósítja a `Z` interfészt.

[C] `M` `bar(z:Z)` metódusa kaphat paraméterül `N` objektumot, mert van közös ősük.

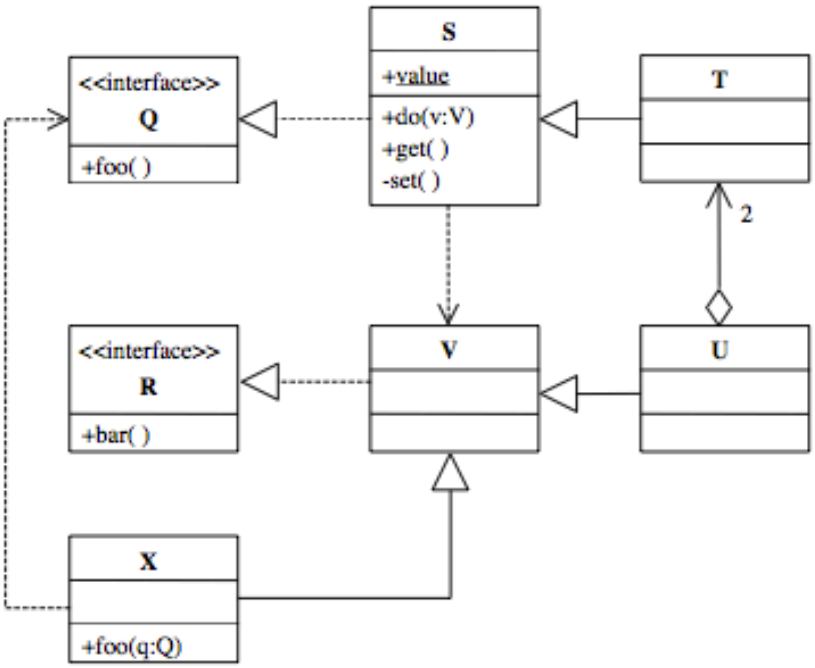
[B] `W` nem helyettesíthető `M`-mel, mert `W`-nek nincs `bar(z:Z)` szignatúrájú metódusa.

[E] `Q` helyettesíthető `M`-mel, mert mindkettő megvalósítja a `Z` interfészt.

[B] `N` `xxx( )` metódusa meghívható a `W` osztály `sort(x:X)` metódusából, mert az `N.xxx( )` statikus.

[C] `W` `sort(x:X)` metódusa meghívhatja egy paraméterül kapott `Y` objektum `bar( )` metódusát, mert `W`-nek is van ugyanilyen szignatúrájú metódusa.

[A] `M`-nek és `N`-nek különböző az interfésze, mert `N` nem valósítja meg `X`-t.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] X foo(q:Q) metódusa kaphat paraméterül U-t, mert U megvalósítja a Q interfészt.

[B] T value attribútuma nem érhető el U-ból, mert az attribútum statikus.

[E] V helyettesíthető S-sel, mert mindketten megvalósítják a Q interfészt.

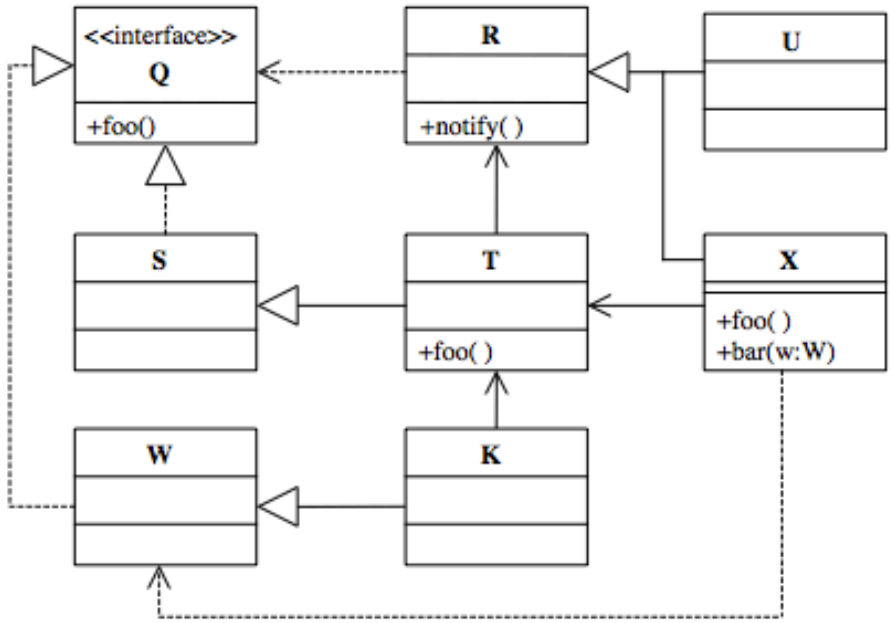
[A] T meghívhatja egy X bar( ) metódusát, mert X függ az S osztálytól.

[C] S nem hozhat létre U osztályú objektumot, mert nem ismeri a T osztályt.

[A] X meghívhatja egy Q interfészű objektum foo( ) metódusát, mert X-nek is van azonos szignatúrájú metódusa.

[E] U megszüntekor 2 T is megsemmisül, mert U kompozíciós kapcsolatban áll 2 T-vel.

[B] S set( ) metódusa nem módosíthatja a value attribútumot, mert a metódus nem publikus.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] *W* és *S* bárhol felcserélhetők, mert közös interfészt implementálnak.

[C] *T* meghívhatja egy *X* osztályú objektum *foo( )* metódusát, mert *X* függ a *Q* interfésztől.

[A] *X* létrehozhat egy *T* osztályú objektumot, mert van közös ősük.

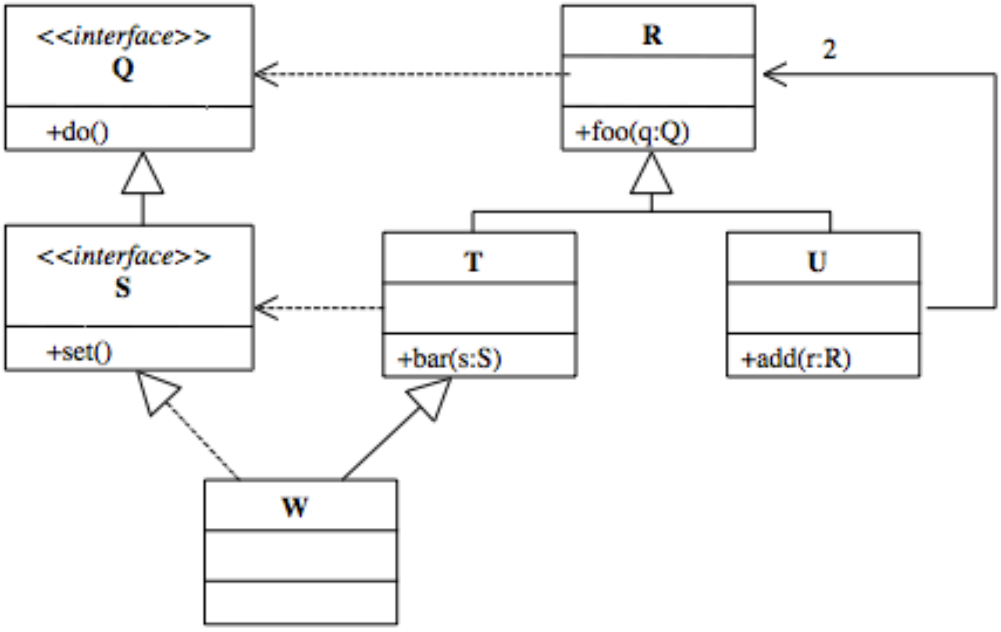
[E] *R* helyettesíthető *T*-vel, mert *T* az *R* leszármazottja.

[B] *K* meghívhatja egy *R* osztályú objektum *notify( )* metódusát, mert *K* megvalósítja a *Q* interfészt.

[B] *K* nem hívhatja meg egy *T foo( )* metódusát, mert *K*-nak is van ugyanilyen szignatúrájú metódusa.

[A] *X bar(w:W)* metódusa kaphat paraméterül *K* osztályú objektumot, mert mindketten *S* leszármazottai.

[E] *U* meghívhatja egy *X foo( )* metódusát, mert van közöttük asszociáció.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[B] **T** helyettesíthető **U**-val, mert **T** és **U** is az **R**-nek leszármazottja.

[D] **R** helyettesíthető **W**-vel, mert **W** az **R** leszármazottja.

[E] **T bar(s:S)** metódusa kaphat paraméterül **T**-t, mert **T** megvalósítja az **S** interfészt.

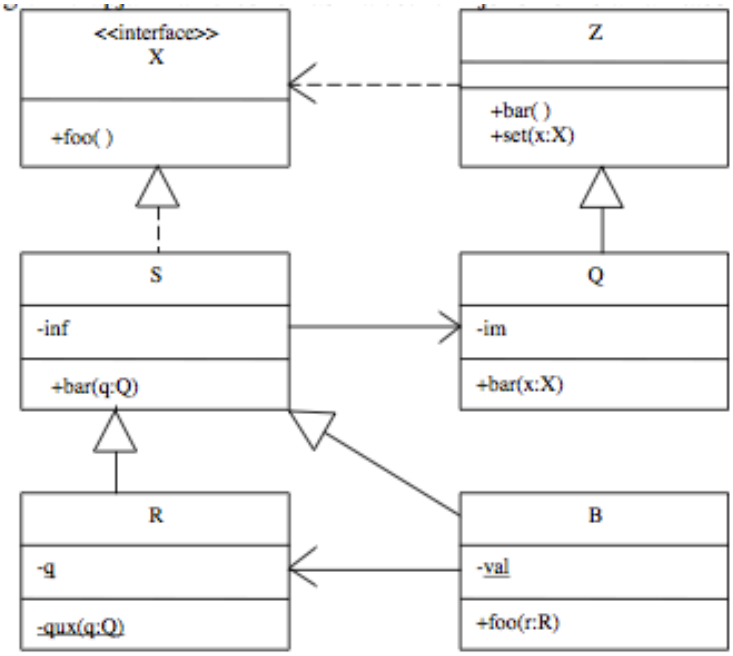
[A] **T bar(s:S)** metódusából meghívhatjuk egy paraméterül kapott **W do( )** metódusát, mert **W** nem valósítja meg a **Q** interfészt.

[C] **U** nem hívhatja meg **T bar(s:S)** metódusát, mert a metódus nem statikus.

[A] **R** meghívhatja egy **Q** interfészű objektum **do( )** metódusát, mert **R** megvalósítja a **Q** interfészt.

[C] **U** pontosan 2 **R**-t ismer, mert **R**-nek 2 közvetlen leszármazottja van.

[B] **U add(r:R)** metódusa nem kaphat **W**-t paraméterül, mert **U** nem ismeri **W**-t.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[E] **Q** bárhol helyettesíthető **S**-sel, mert **S** a **Q** leszármazottja.

[A] **R qux(q:Q)** metódusa nem kaphat paraméterül **Z** objektumot, mert a metódus absztrakt

[E] **B foo(r:R)** metódusa nem hívhat meg a paraméterül kapott objektumon **foo()** metódust, mert **R**-nek nincs ilyen metódusa

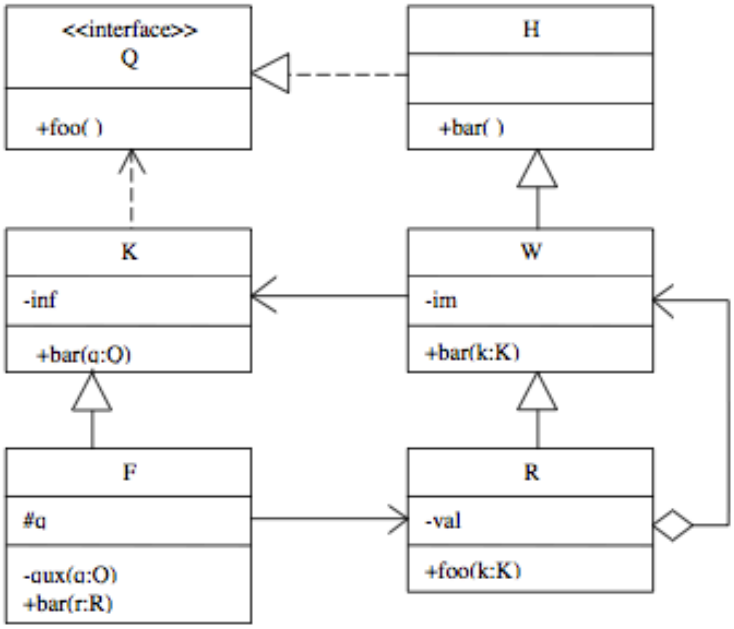
[E] **S bar(q:Q)** metódusa nem módosíthatja az **S inf** attribútumát, mert az attribútum konstans.

[E] **S bar(q:Q)** metódusa nem hívhatja meg a paraméterül kapott objektum **bar()** metódusát, mert a **Q** osztálynak nincs ilyen szignatúrájú metódusa.

[B] **Z set(x:X)** metódusa nem kaphat paraméterül **B** objektumot, mert **B** megvalósítja az **X** interfészt.

[B] **B** módosíthatja egy **Q** objektum **im** attribútumát, mert **S** függ **Q**-tól.

[B] **R** és **B** bárhol felcserélhetők, mert közös az ősük.



A - csak az első tagmondat igaz (+ -)
B - csak a második tagmondat igaz (- +)
C - mindkét tagmondat igaz, de a következtetés hamis (+ + -)
D - mindkét tagmondat igaz és a következtetés is helyes (+ + +)
E - egyik tagmondat sem igaz (- -)

[D] **H** bármikor helyettesíthető **R**-rel, mert **R** a **H** leszármazottja.

[A] **W** nem módosíthatja **K inf** attribútumát, mert az attribútum protected.

[E] **F bar(r:R)** metódusa nem hívhatja meg a **qux(q:Q)** metódust, mert az utóbbi statikus.

[B] **F qux(q:Q)** meghívhatja a **bar(r:R)** metódust a **q** paraméterrel, mert az **R** megvalósítja a **Q** interfészt.

[E] Az ábrán szereplő összes egy-paraméteres **bar** metódus kaphat **R** objektumot paraméterül, mert **R** megvalósítja az összes említett metódust.

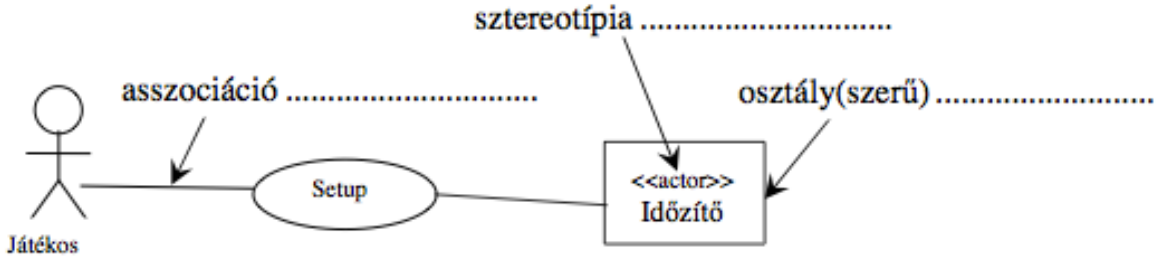
[E] **W bar(k:K)** metódusa nem módosíthatja az osztály **im** attribútumát, mert az attribútum konstans.

[C] **W** rendelkezik **foo( )** szignatúrájú metódussal, mert függ **K**-tól.

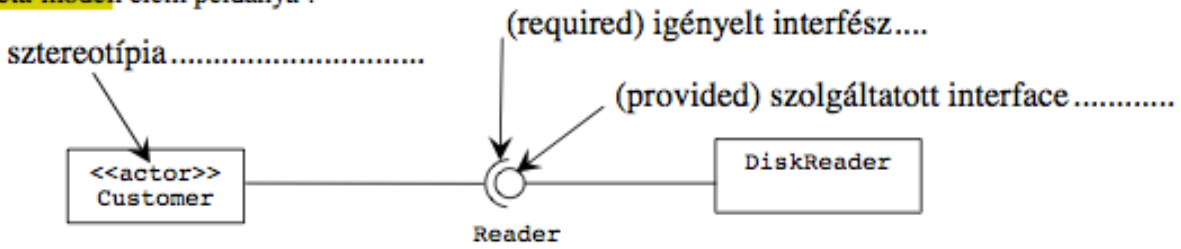
[D] Egy **F** objektum meghívhatja saját magával mint paraméterrel egy **R foo(k:K)** metódusát, mert **F** a **K** leszármazottja.

6. <u>feladat típus (ábra - megnevezés):</u>

Az alábbi ábrán három UML2 modell elemet megjelöltünk. Adja meg elemenként, hogy az melyik UML2 meta-modell elem példánya !



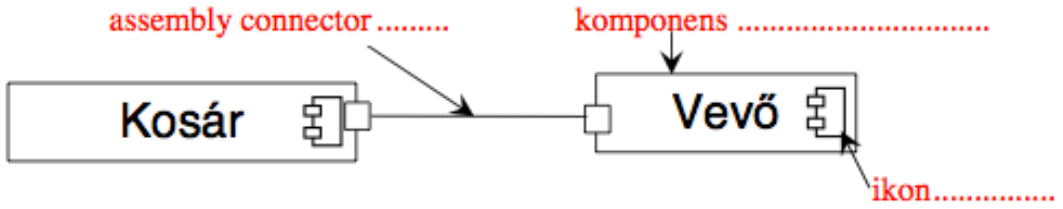
Az alábbi ábrán három UML2 modell elemet megjelöltünk. Adja meg elemenként, hogy az melyik UML2 meta-modell elem példánya !



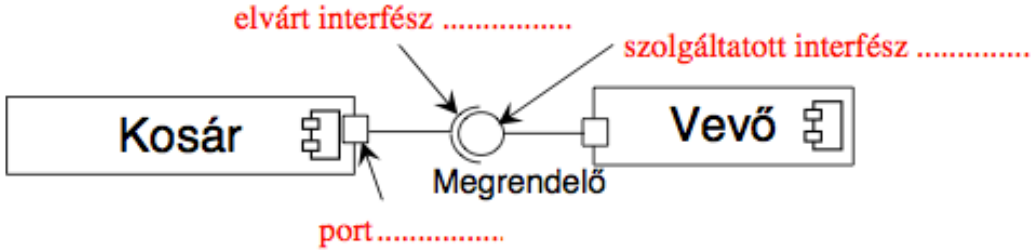
Feladatok Komponens diagram témakörből

1. feladat típus (ábra - megnevezés):

Az alábbi ábrán három UML2 modell elemet megjelöltünk. Adja meg elemenként, hogy az melyik UML2 meta-modell elem példánya !



Az alábbi ábrán három UML2 modell elemet megjelöltünk. Adja meg elemenként, hogy az melyik UML2 meta-modell elem példánya !



2. feladat típus (elmélet):

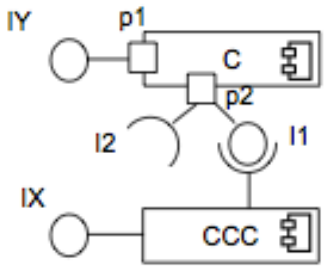
Az UML komponensek fajtái:

deployment (a sw indításához kell, pl class) ..... execution (a végrehajtás közben keletkezik, pl file)..

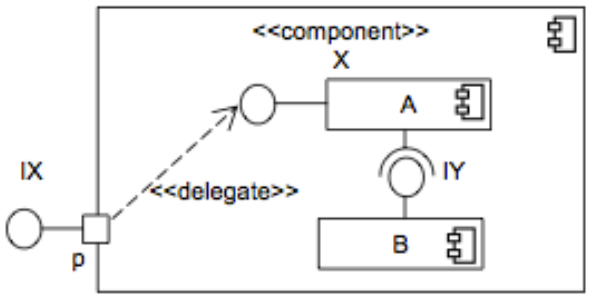
work product (a fejlesztő munka terméke, pl *.h) . .....

3. feladat típus (rajzolás):

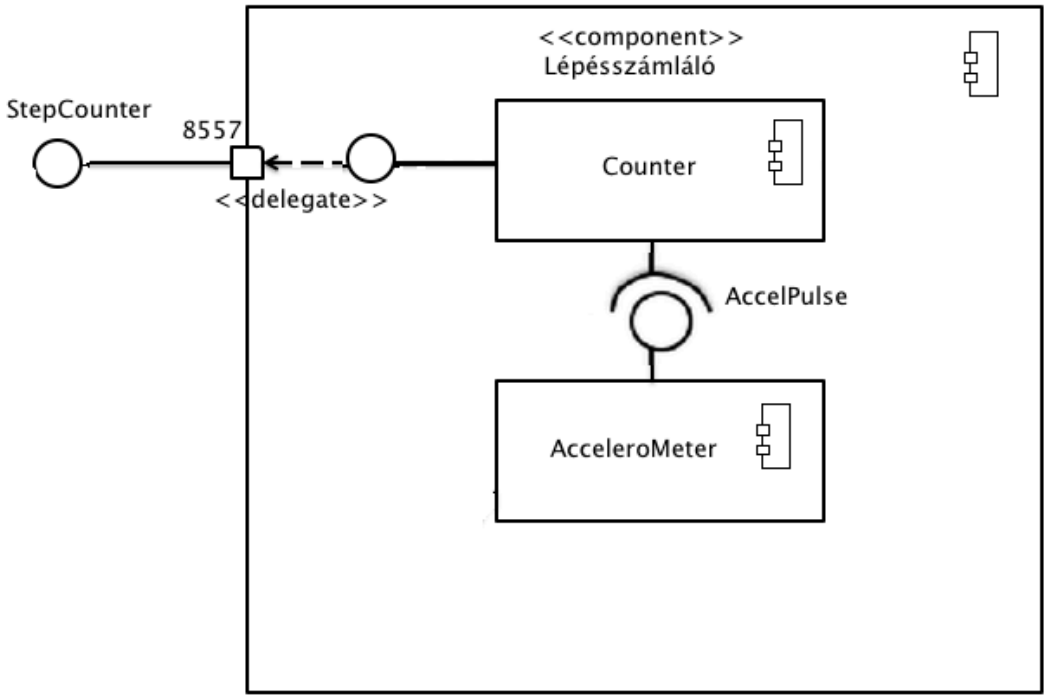
Legyen egy `C` komponensünk, amely a `p1` portján megvalósítja az `IY` interfészt, a `p2` portján megvalósítja az `I1` interfészt és várja az `I2` interfészt. Van egy `CCC` komponensünk is, amely az `I1` interfészen keresztül kapcsolódik a `C` komponenshez. A `CCC` komponens megvalósítja az `IX` interfészt is. Rajzoljon UML2 komponens diagramot !



Legyen egy X komponensünk, amely p portján megvalósítja az IX interfészt. Ennek a komponensnek a felépítéséhez felhasználjuk az A komponenst, amely realizálja az IX interfészt, de szükséges felhasználnunk egy B komponenst is, amely megvalósítja az A által elvárt IY interfészt. Rajzoljon UML2 komponens diagramot !

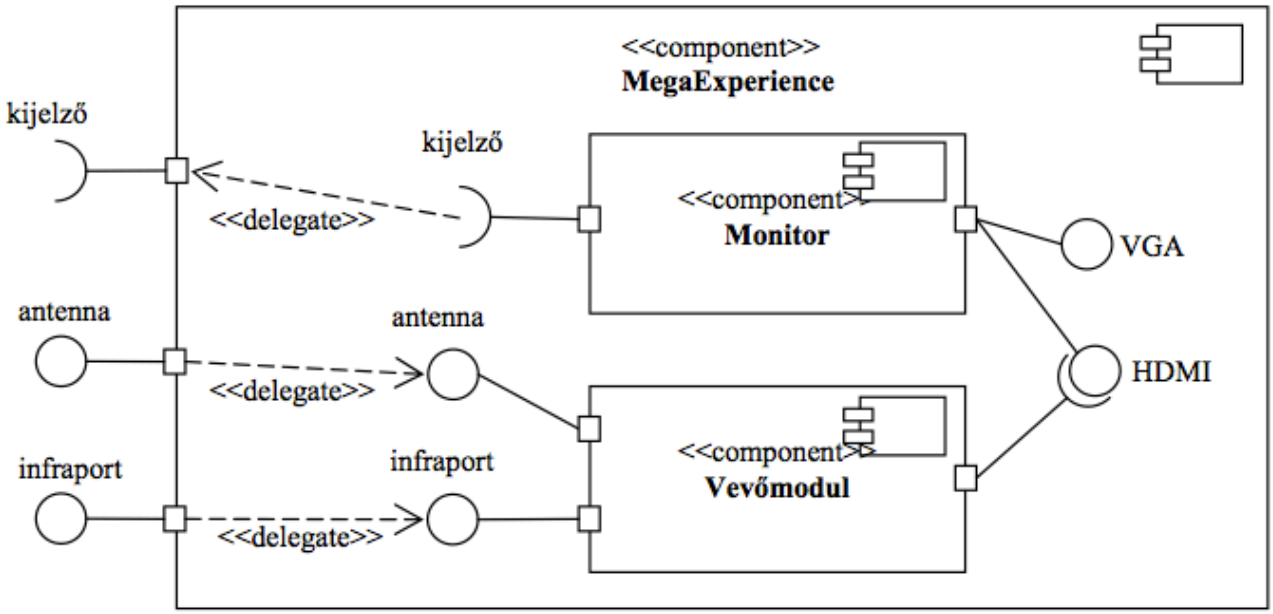


A mobiltelefonba épített lépésszámláló komponens a 8557-es porton megvalósítja a StepCounter interfészt. Ez az interfész megegyezik a lépésszámláló felépítéséhez felhasznált Counter komponens interfészével. A Counter felhasználja az AcceleroMeter komponenst, azzal az AccelPulse interfészen keresztül kapcsolódik. Rajzoljon UML2 komponens diagramot !



Készítsen UML2 komponens diagramot az alábbi leírás alapján!

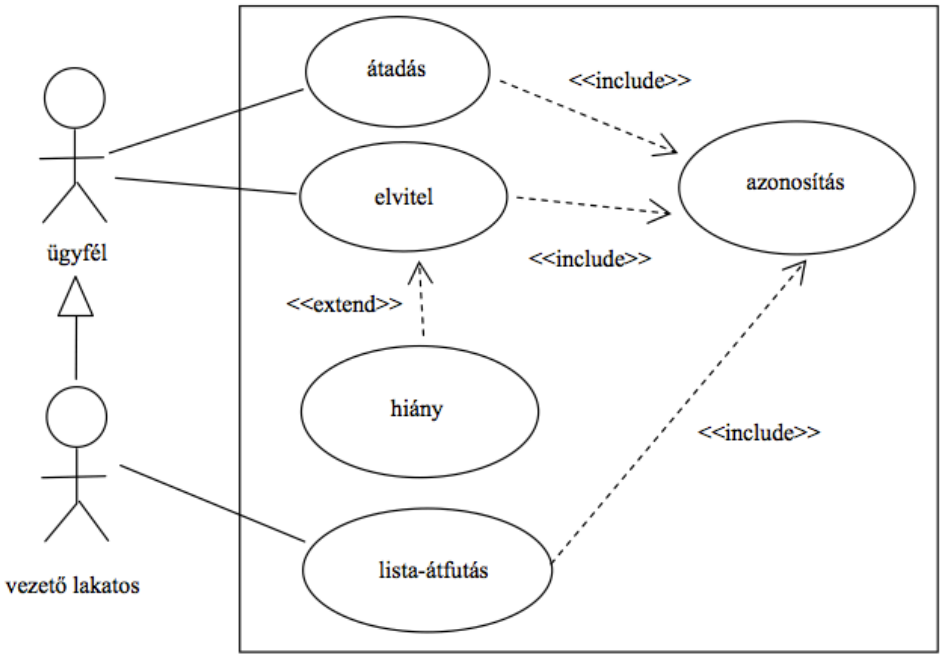
A MegaExperience televíziókészülék a szokásos felülettel rendelkezik: van kijelzője (ami elvárja, hogy valaki nézze), antennabemenete és infraportja. Ha a készüléket szétszedjük, akkor azt látjuk, hogy valójában egy monitorból és egy vevőmodulból áll. A monitor biztosítja a kijelzést, míg az antennabemenet és az infraport a vevőmodulhoz kapcsolódik. A monitor VGA és HDMI interfésszel is rendelkezik, a modul ebből a HDMI interfészt használja a kommunikációhoz.



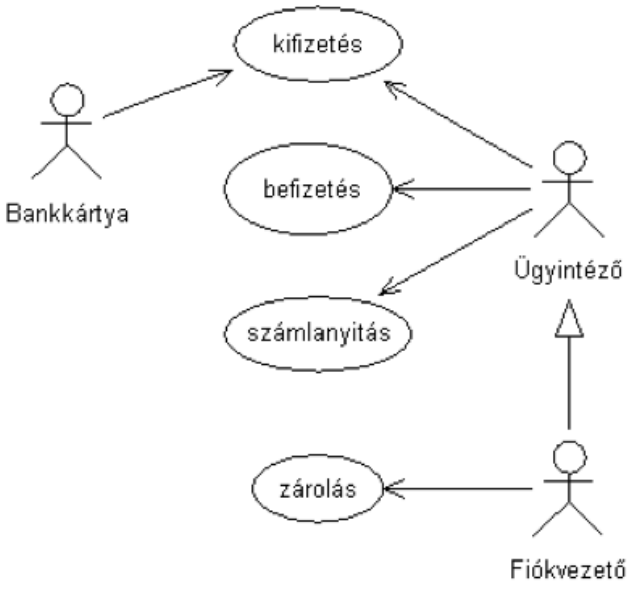
Feladatok Use-Case Diagram témakörből

1. feladat típus (rajz):

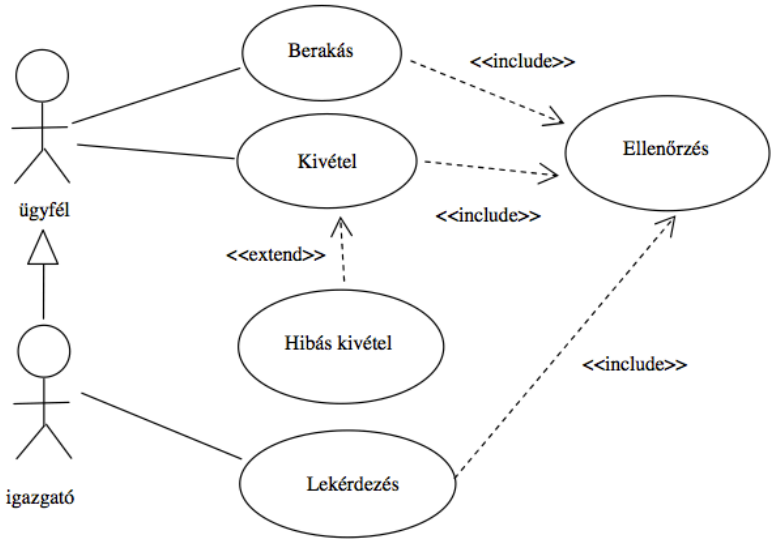
Rajzoljon UML 2.0 use-case diagramot az alábbi leírás alapján!
A Rezesbanda Kft. éjjel-nappali autóbontó telepet működtet, amit informatikai rendszerrel kíván megtámogatni. A regisztrált ügyfelek roncs autókat adnak át, és alkalmanként használt autóalkatrészt visznek el. Mindkét esetben jelszóval azonosítják magukat. Ha a kért autóalkatrész nincs raktáron, akkor a rendszer felírja a kérést a kívánságlistára. A telep vezető lakatosa időnként átfutja a kívánságlistát, hogy lássa, mire van szükség, ekkor ő is jelszóval azonosítja magát. A vezető lakatos ügyfélként is viselkedhet.



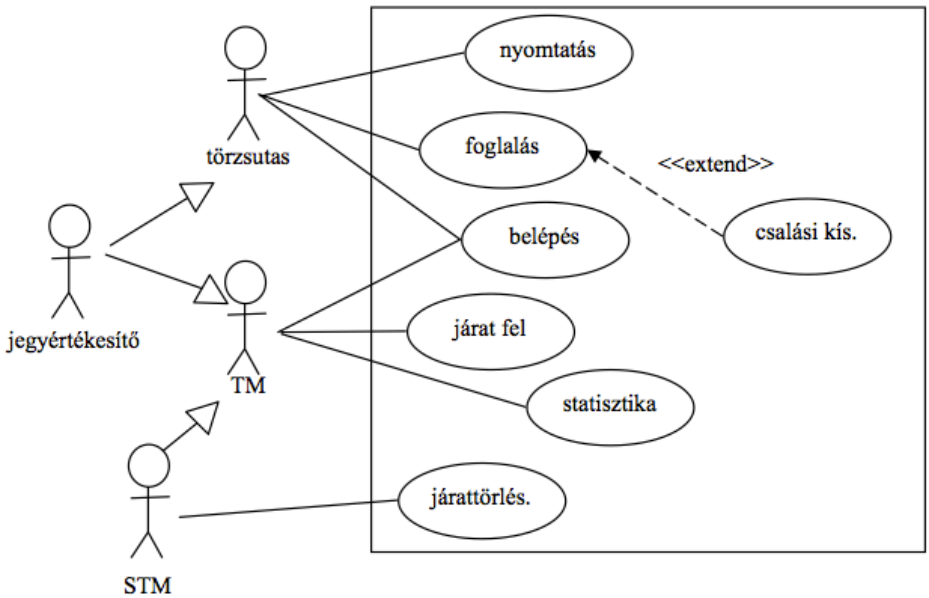
Egy banki rendszerben bankkártyával pénzt tudunk felvenni egy bankszámláról, banki ügyintéző segítségével ezen kívül pénzt lehet befizetni és átutalni. A számlát zárolni csak a fiókvezető tudja, aki természetesen rendelkezik az ügyintézési jogosultságokkal is. Rajzoljon **UML use-case diagram**ot.



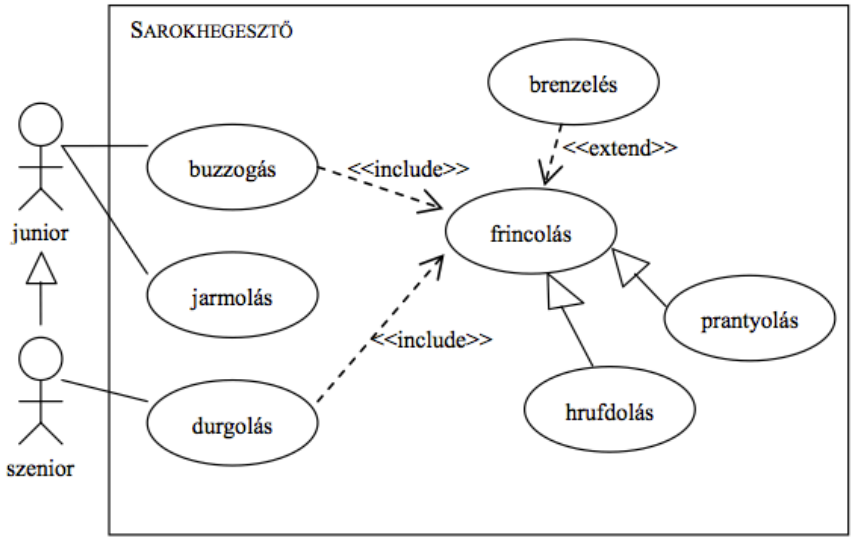
Rajzoljon UML2 use-case diagramot az alábbi történet alapján!
A PickPack Raktárszolgáltató Kft bivalypasnádi telepén hatalmas polcrendszerekben tárolják a csomagokat. Ha egy regisztrált ügyfél csomagot hoz tárolási célból, ellenőrzik az azonosítóját, majd robotok egy üres helyre teszik a csomagot. Mikor az ügyfél elvinné egyik csomagját, ismét ellenőrzik az azonosítóját, majd előhozzák a kívánt csomagot. Előfordulhat, hogy a csomag nincs a helyén, mert már korábban elvitték. Ebben az esetben a rendszer figyelmezteti az ügyfelet a hibára. A raktár igazgatója mindazt meg tudja tenni, amit egy egyszerű ügyfél, de ő lekérdezheti a telepen tárolt csomagok számát is. Ehhez természetesen az ő azonosítójót is ellenőrzik.



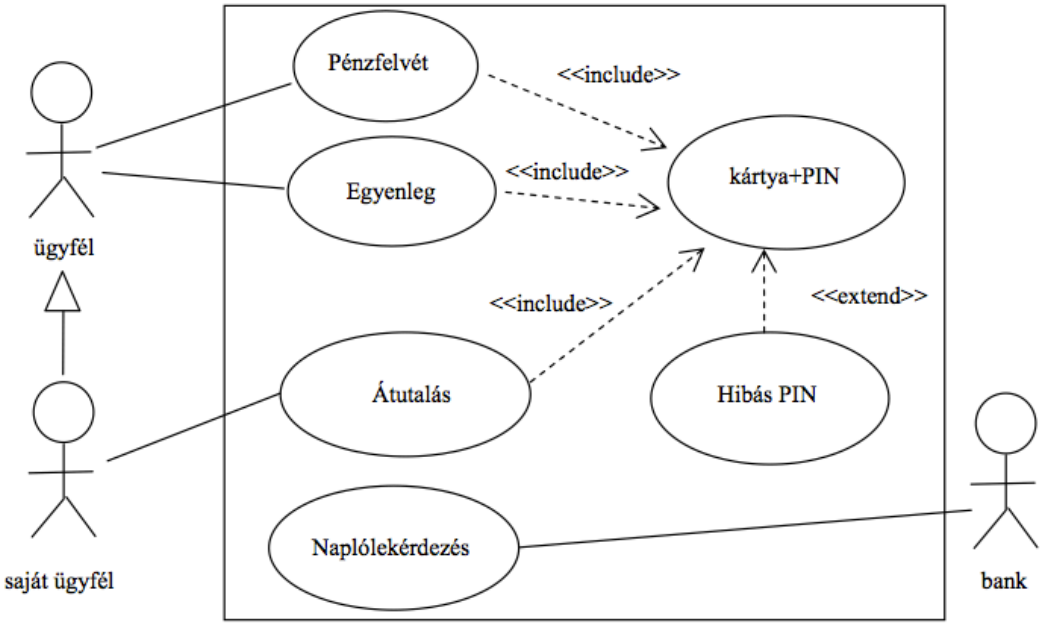
Rajzoljon UML 2.0 use-case diagramot az alábbi leírás alapján!
A Lushfanta légitársaság jegyfoglaló rendszerét törzsutasok és ticket-managerek (TM) használhatják a repülőjegyfoglalások rögzítésére. A törzsutasok bejelentkezhetnek, jegyet foglalhatnak és beszállókártyát nyomtathatnak. Ha jegyfoglalás közben kiderül, hogy módosultak a session-adatok, akkor a rendszer rögzíti a csalási kísérletet. A TM-ek is belépnek, járatok adatait vihetik fel, illetve jegyvásárlási statisztikákat kérhetnek le. A senior ticket-managerek (STM) az egyszerű TM lehetőségein túl még járatot is törölhetnek. A jegyértékesítők, mivel járatok adatait is kezelniük kell, mind TM-ként, mint törzsutasként használhatják a rendszert.



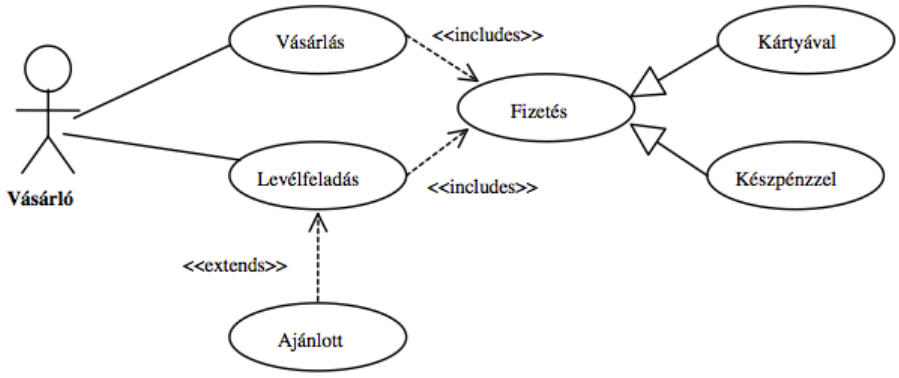
Készítsen UML2 use-case diagramot az alábbi leírás alapján!
A sarokhegesztővel a junior furga buzzogni, és jarmolni tud. A buzzogáshoz be kell kapcsolni a frincolást. Ennek két módja van: a prantyolás és a hrufdolás. A herkentyű hibás beállítások esetén frincolás közben brenzel is kicsit. A szenior furga a fentiek mellett a durgolás funkcióhoz is hozzáfér, amihez szintén be kell kapcsolni a fenti frincolás funkciót.



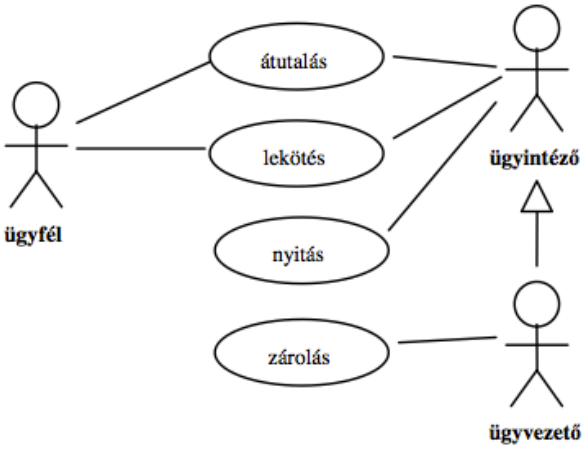
Rajzoljon UML2 use-case diagramot az alábbi történet alapján!
A Management Optimal Bonus (MOB) Bank automatáival pénzt lehet felvenni, számlaegyenleget lehet lekérdezni, és a bank saját ügyfelei pénzt utalhatnak a bank vezetőségi bónusz programja számára. Mindezen funkciók eléréséhez be kell helyezni a kártyát és meg kell adni a 4 jegyű azonosítót (PIN). Ha ez háromszor egymás után nem sikerül, az automata a kártyával elérhető teljes összeget a bank jutalomkeretére utalja. Ezen kívül a bank lekérdezheti az automata naplófájlját.



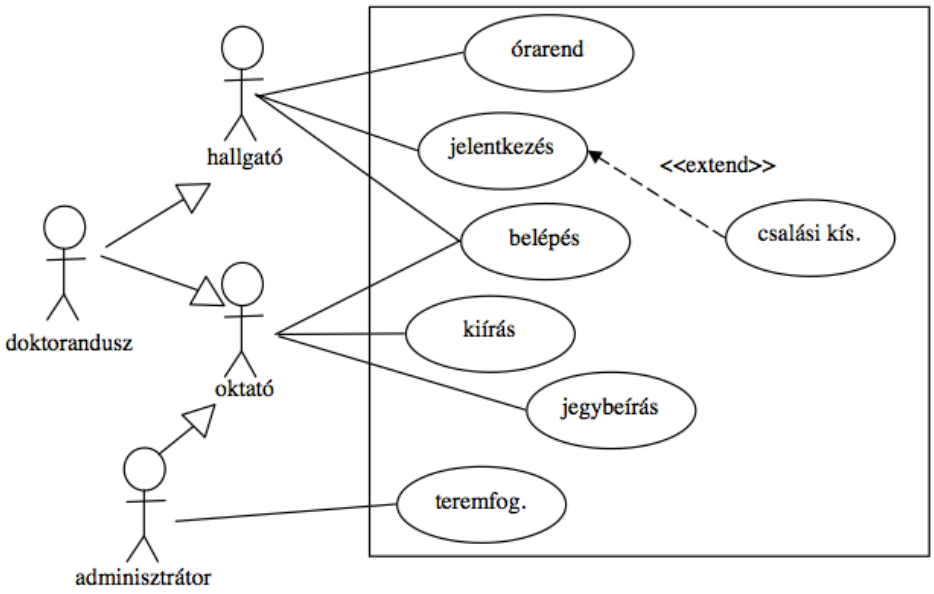
Egy jobb angol boltocskában lehet árut (tejet, újságot, stb.) vásárolni és levelet föladni. Mindkettőnek elengedhetetlen része a fizetés, ami történhet készpénzzel vagy kártyával. Egyes boltokban lehetőség van ajánlott leveleket is föladni. Rajzoljon use-case diagramot !



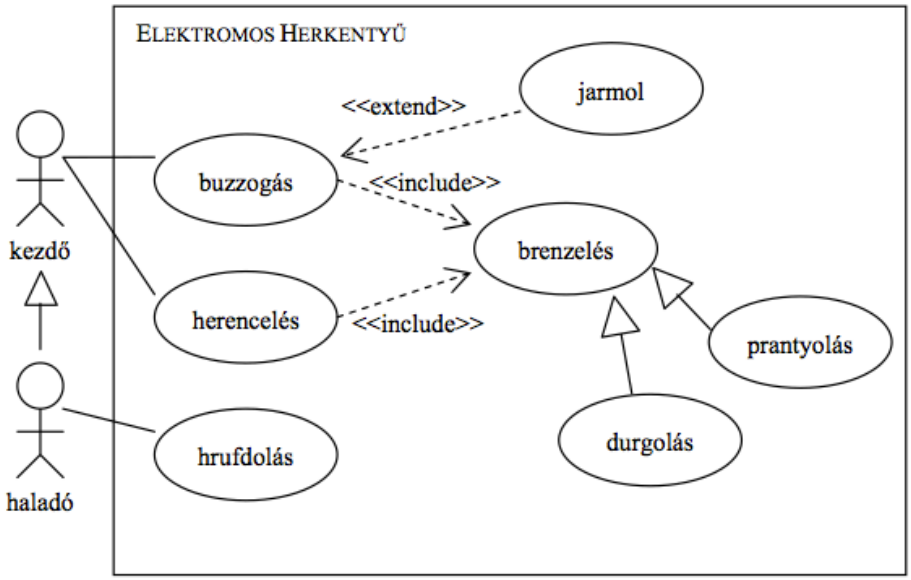
Rajzoljon UML use-case diagramot a következő leírás alapján !
A *MoneyOrLife* bankban az ügyfelek a weben átutalást és lekötést kezdemenyezhetnek. A fiókokban az ügyintézők átutalás és lekötés mellett új számlát is nyithatnak. A fiók ügyvezetője az ügyintézői jogosultságokon kívül a számlákat zárolhatja is.



Rajzoljon UML 2 use-case diagramot az alábbi leírás alapján!
A Poseidon rendszert egyetemi oktatók és hallgatók használhatják a tanulmányi adatok rögzítésére. A hallgatók bejelentkezhetnek, vizsgára jelentkezhetnek és órarendet nyomtathatnak. Ha vizsgára jelentkezés közben kiderül, hogy valamely előfeltétel nem teljesül, akkor a rendszer rögzíti a csalási kísérletet. Az oktatók is belépnek, vizsgát írhatnak ki, illetve vizsgaeredményeket írhatnak be. Az adminisztrátorok az oktatók lehetőségein túl még termet is foglalhatnak. A doktoranduszok, mivel oktatniuk és tanulniuk is kell, mind oktatóként, mind hallgatóként használhatják a rendszert.



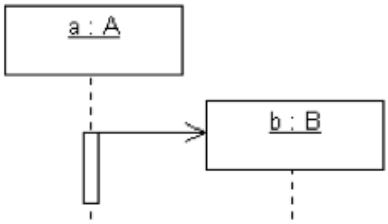
Készítsen UML2 use-case diagramot az alábbi leírás alapján!
Az elektromos herkentyűn a kezdő kukor buzzogni, és herencelnı tud. Mindkettőhöz be kell kapcsolni a brenzelést. Ennek két módja van: prantyolással vagy durgolással. A herkentyű hibás beállítások esetén buzzogás közben jarmol is kicsit. A haladó kukor a fentiek mellett a hrufdolás funkcióhoz is hozzáfér.

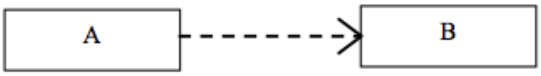


Feladatok Szekvencia diagram témakörből

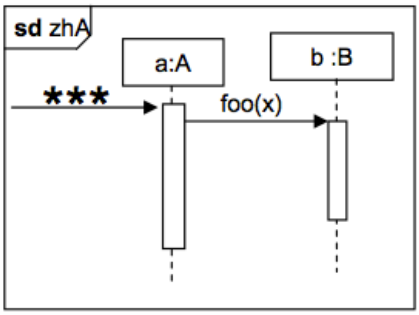
1. feladat típusok:

A szekvencia diagram alapján jelölje az A és B közötti kapcsolatot **UML osztálydiagram**on!

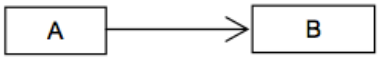




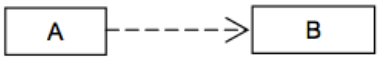
Tételezze fel, hogy az alábbi (zhA nevű) UML2 szekvenciadiagramon szereplő objektumok osztályai között nincs más egyéb – a diagramból nem kiolvasható – kapcsolat (pl. öröklés) ! Rajzolja be az A és B között kapcsolatot, ha



*** = bar(x)

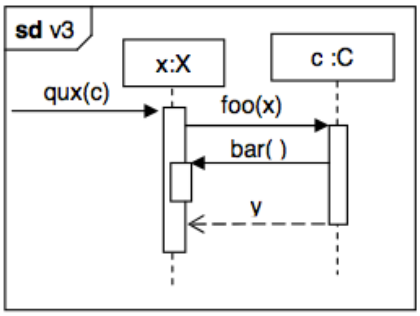


*** = qwx(b)

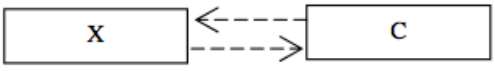


Tételezze fel, hogy az alábbi (v3 nevű) UML2 szekvenciadiagramon szereplő objektumok osztályai között nincs más egyéb – a diagramból nem kiolvasható – kapcsolat (pl. öröklés) ! Mi a kapcsolat X és C között ?

- ☐ implementálás (implementation)
- ☐ kollaboráció (collaboration)
- ☒ függőség (dependency) C függ X-től
- ☒ függőség (dependency) X függ C-től
- ☐ asszociáció (association)
- ☐ példányosítás (instantiation)
- ☐ interakció (interaction)

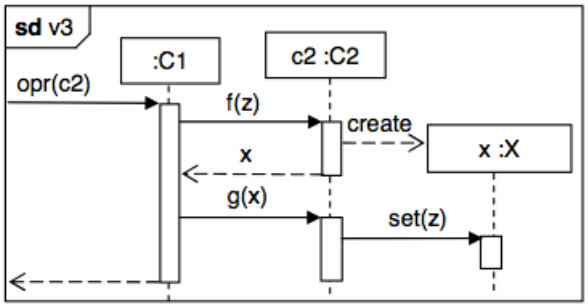


Rajzolja be választását az alábbi osztálydiagramba !



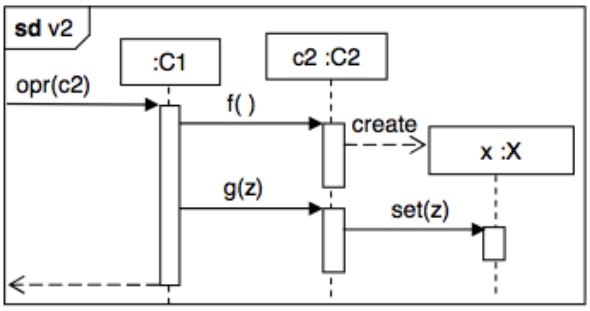
Tételezze fel, hogy az alábbi UML2 szekvenciadiagramon szereplő objektumok osztályai között nincs más egyéb – a diagramból nem kiolvasható – kapcsolat (pl. öröklés) ! Mi a kapcsolat C2 és X között ? Válaszát egy, a magyar nyelv szabályainak megfelelő, olvasható MONDATtal indokolja ! Indoklás nélkül a választása nem érvényes.

- ☐ példányosítás (instantiation)
- ☐ asszociáció (association)
- ☐ kollaboráció (collaboration)
- ☒ függőség (dependency) C2 függ X-től
- ☐ függőség (dependency) X függ C2-től
- ☐ interakció (interaction)
- ☐ implementálás (implementation)



Tételezze fel, hogy az alábbi UML2 szekvenciadiagramon szereplő objektumok osztályai között nincs más egyéb – a diagramból nem kiolvasható – kapcsolat (pl. öröklés) ! Mi a kapcsolat C2 és X között ? Válaszát egy, a magyar nyelv szabályainak megfelelő, olvasható MONDATtal indokolja ! Indoklás nélkül a választása nem érvényes.

- ☐ példányosítás (instantiation)
- ☒ asszociáció (association)
- ☐ kollaboráció (collaboration)
- ☐ függőség (dependency) C2 függ X-től
- ☐ függőség (dependency) X függ C2-től
- ☐ interakció (interaction)
- ☐ implementálás (implementation)

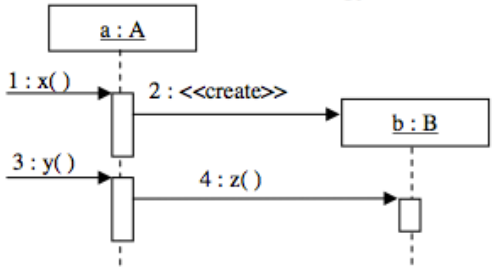


Indoklás:

C2-nak ”emlékeznie kell” X-re .....

Tételezze fel, hogy az alábbi szekvenciadiagramon szereplő objektumok osztályai között nincs más egyéb – a diagramból nem kiolvasható – kapcsolat (pl. öröklés) ! Mi a kapcsolat A és B között ? Válaszát indokolja ! Az indoklás legalább egy, a magyar nyelv szabályainak megfelelő, olvasható MONDAT legyen ! Indoklás nélkül a választása nem érvényes.

- ☐ függőség (dependency)
- ☒ asszociáció (association)
- ☐ kollaboráció (collaboration)
- ☐ példányosítás (instantiation)
- ☐ implementálás (implementation)

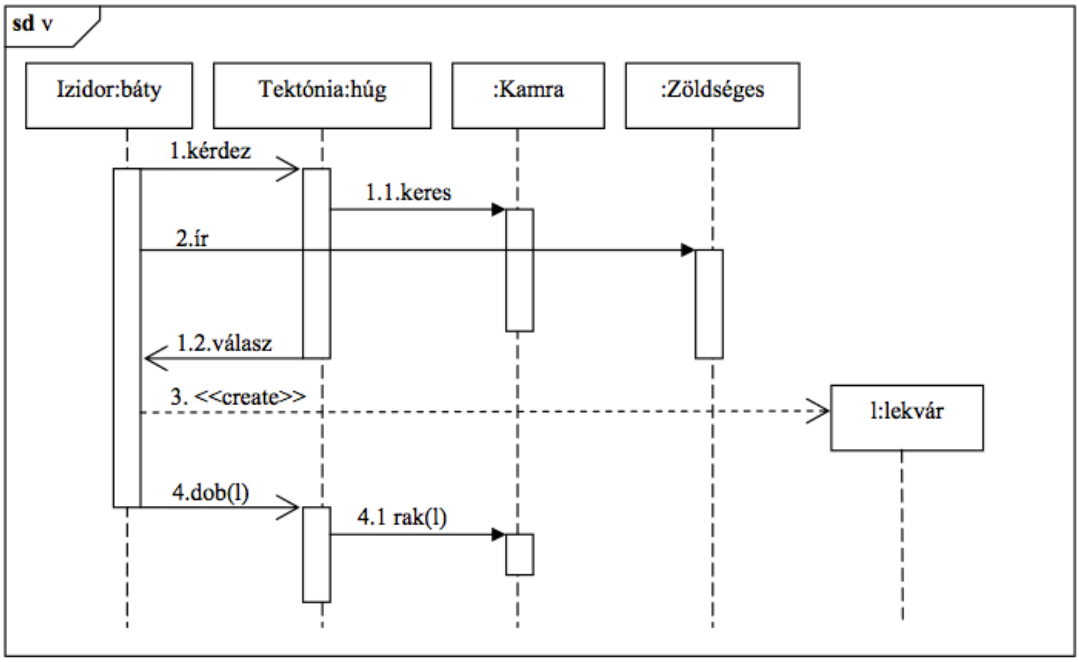


Indoklás:

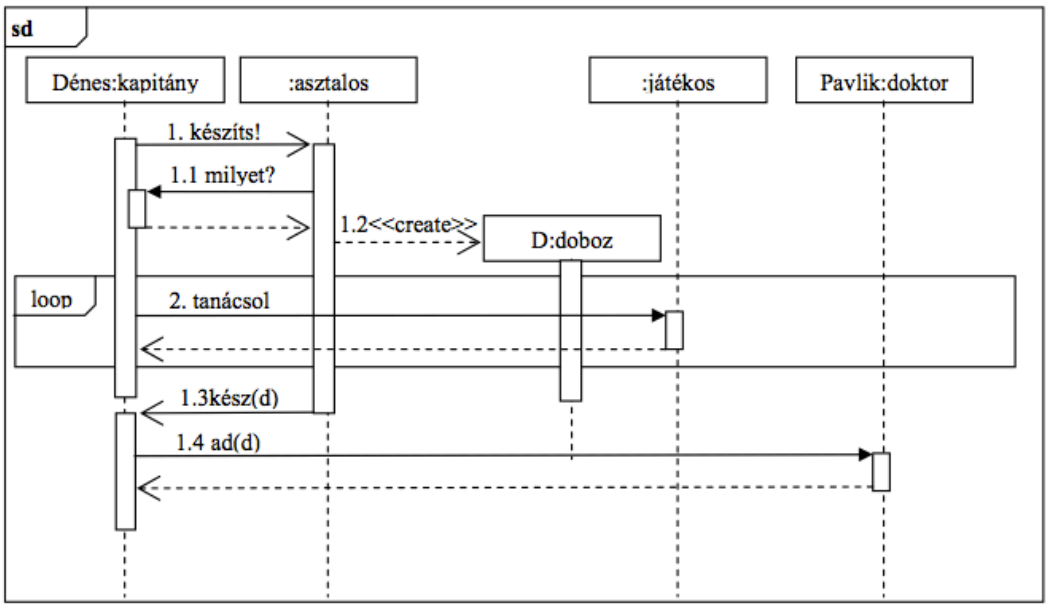
A-nak ”emlékeznie kell” B-re .....

2. feladat típus (rajz):

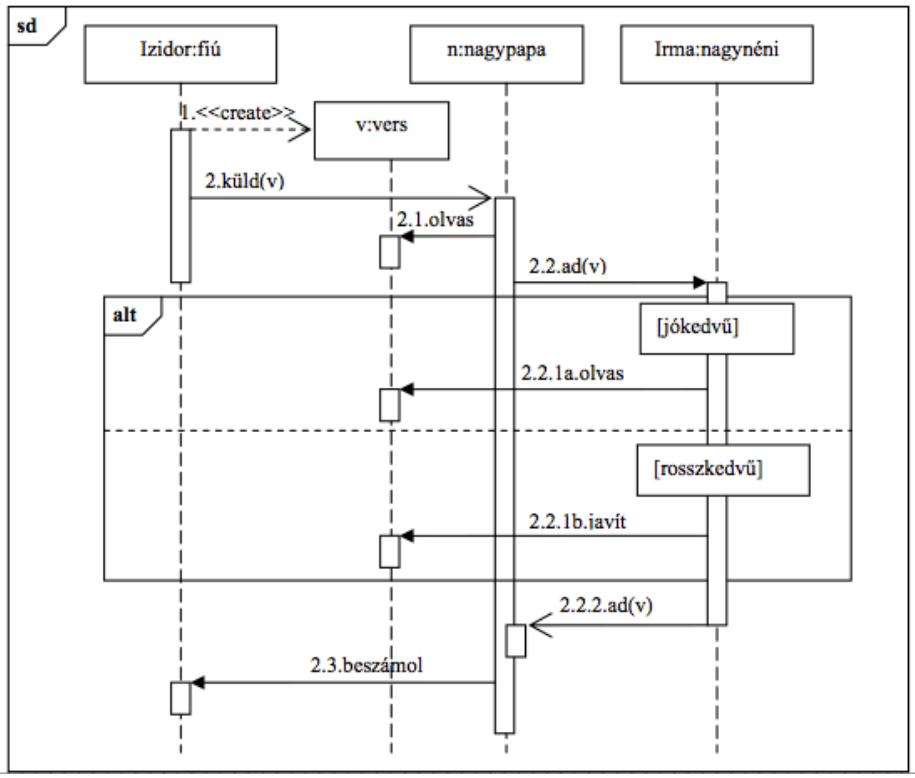
Készítsen UML2 szekvencia-diagramot az alábbi történet alapján! Ne feledkezzen el a hierarchikus számozásról sem!
Izidor a zöldségesnél rájön, hogy lekvárt akar főzni, ezért SMS-ben megkérdi húgát, Tektóniát, hogy van-e otthon befőző cukor. Tektónia átkutatja a kamrát, és talál cukrot, amiről (szintén SMS-ben) értesíti bátyját. Izidor eközben vicces szöveget ír a zöldséges hátára, amíg a választ meg nem kapja, majd hazamegy, és megfőzi a lekvárt. A kész lekvárt választ sem várva odadobja húgának, és elsiet. Tektónia a lekvárt beteszi a kamrába.



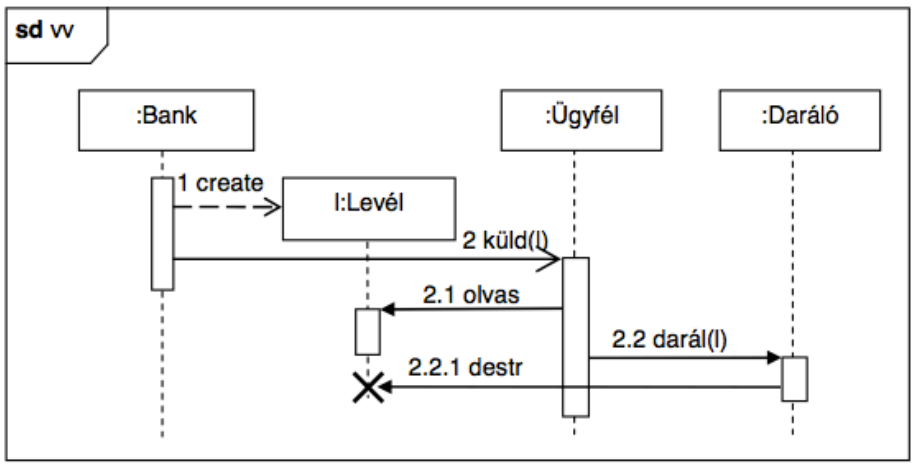
Rajzoljon UML 2.0 szekvenciadiagramot (sequence diagram) az alábbi leírás alapján!
Kemény Dénes (a vízilabda-válogatott szövetségi kapitánya), mivel tudja, hogy Pavlik doktornak nincs miben tartania az olimpiai aranyait, ezért egy intarziás fadobozt csináltat kedvenc asztalosával. Az asztalos, mielőtt elkezdené a munkát, megkérdi Dénest, hogy pontosan milyen minta legyen a dobozon, majd elkezdi legyártani a remekművet. Közben a kapitány minden egyes játékosát egyenként szakmai tanácsokkal látja el. Mikor a doboz elkészül, az asztalos elküldi Dénesnek, aki fogja, és azon nyomban átadja a doktornak.



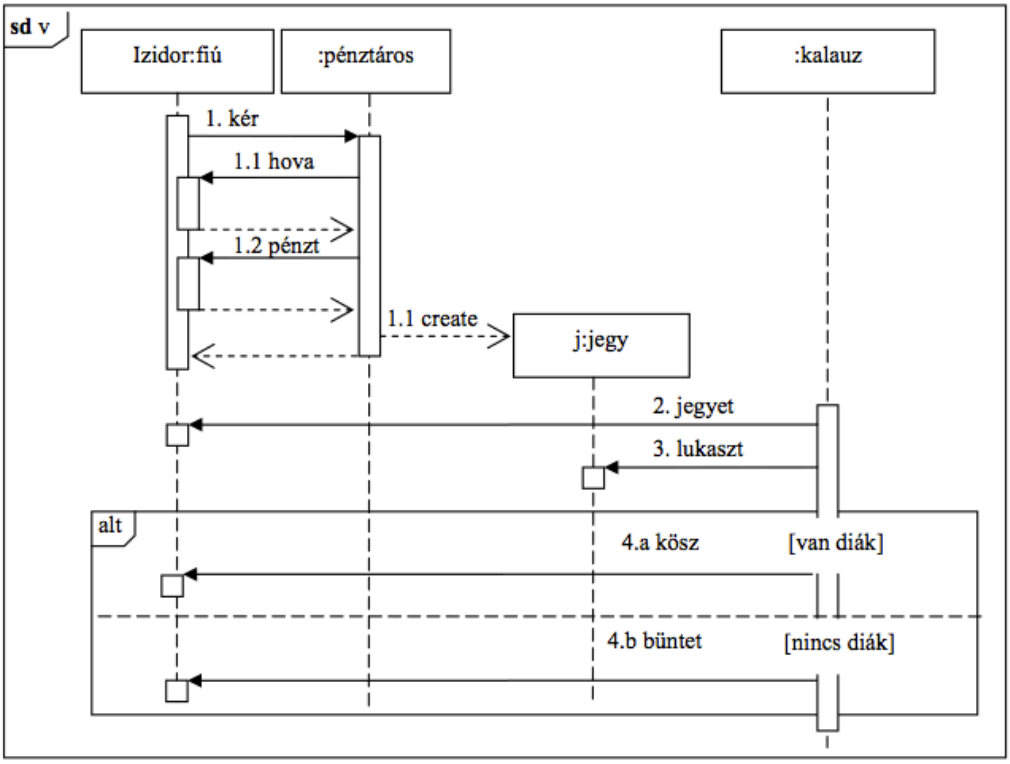
Készítsen UML2 szekvencia-diagramot az alábbi történet alapján! Ne feledkezzen el a hierarchikus számozásról sem!
Izidor verset ír ajándékként vidéken élő nagynénjének, Irmának. Az ajándékot a nagypapának küldi azzal, hogy adja át a nagynéninek. A nagypapa kíváncsi, és elolvassa a verset. Ezután odaadja Irmának, és várja a hatást. Ha a nagynéni jókedvű, akkor a verset felolvassa, ha rosszkedvű, akkor aláhúzza benne a nyelvtani hibákat. Mindezek után a verset visszaadja a nagypapának, és elsiet. A nagypapa ezután felutazik Pestre, és a történtekről beszámol Izidornak.



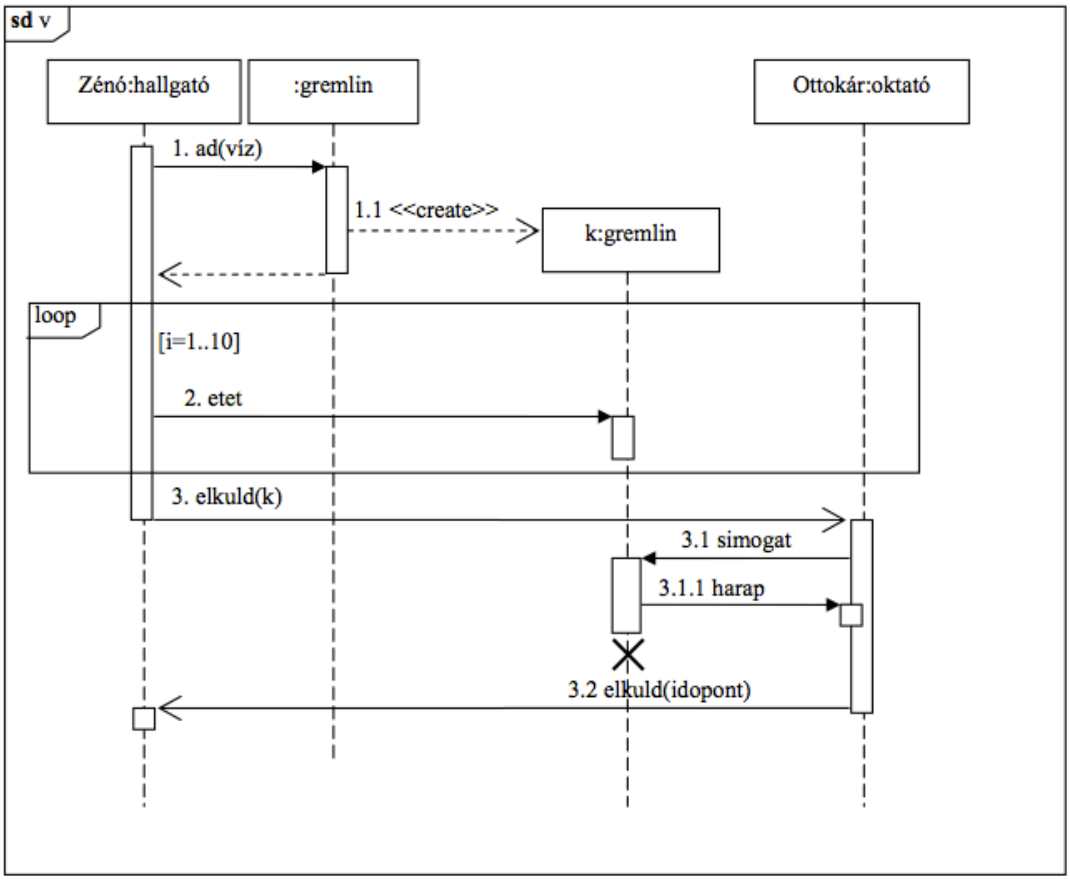
Rajzoljon UML2 szekvenciadiagramot ! Az üzeneteket hierarchikus számozással lássa el !
Az InterCredit Bank felszólító levelet ír, amelyet elküld egyik ügyfelének, Gerzsonnak, akinek hiteltartozása van. Gerzson a levelet elolvassa, majd a darálón ledarálja.



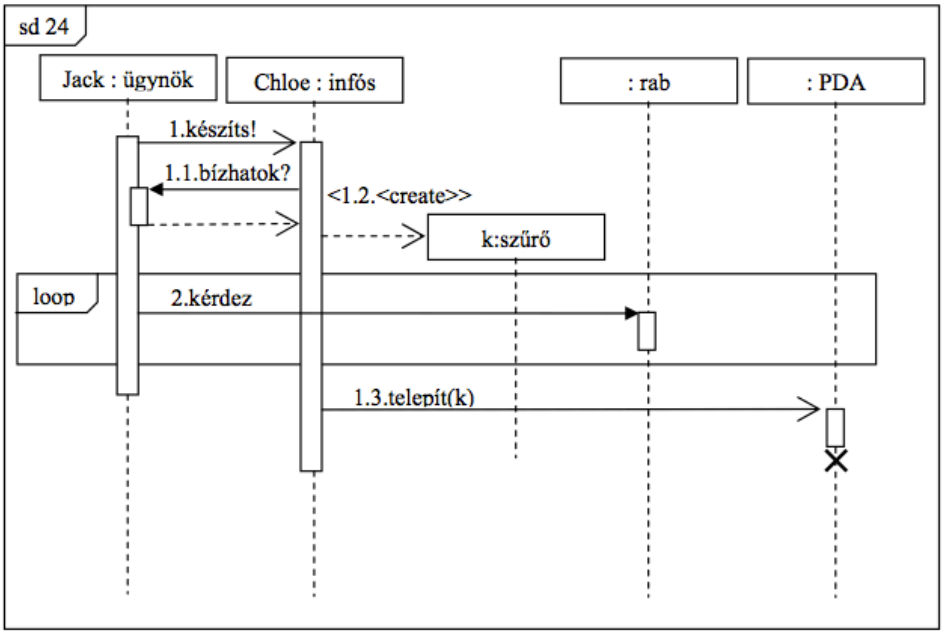
Rajzoljon UML 2.0 szekvenciadiagramot (sequence diagram) az alábbi leírás alapján!
Izidor vonatjegyet szeretne venni, hogy elutazzon nagymamájához. A jegypénztárnál kér egy retúrjegyet. A pénztáros megkérdezi, hogy hova. A pénztáros elkéri a pénzt, majd kinyomtatja a jegyet, és a visszajáróval együtt Izidornak adja. Később (már a vonaton) a kalauz elkéri a jegyet és kilukasztja. Ha Izidornál nincs nála a diákigazolványa, akkor (a kalauz) megbünteti, ha nála van, akkor (a kalauz) megköszöni.



Készítsen UML2 szekvencia-diagramot az alábbi történet alapján! Ne feledkezzen el a hierarchikus számozásról sem!
Zénó a karácsonyra kapott gremlinjének véletlenül vizet ad, mire az egy kisgremlinnek ad életet. Zénó nagyon megörül, és 10-szer megeteti a kisgremlint, majd elküldi oktatójának, Gyíkarcú Ottokárnak házi feladat helyett. Ottokár megsimogatja a kisgremlint, aki simogatás közben megharapja, de Ottokár vitriolos vérétől el is pusztul. Ottokár elküldi Zénónak a pótleadás időpontját.

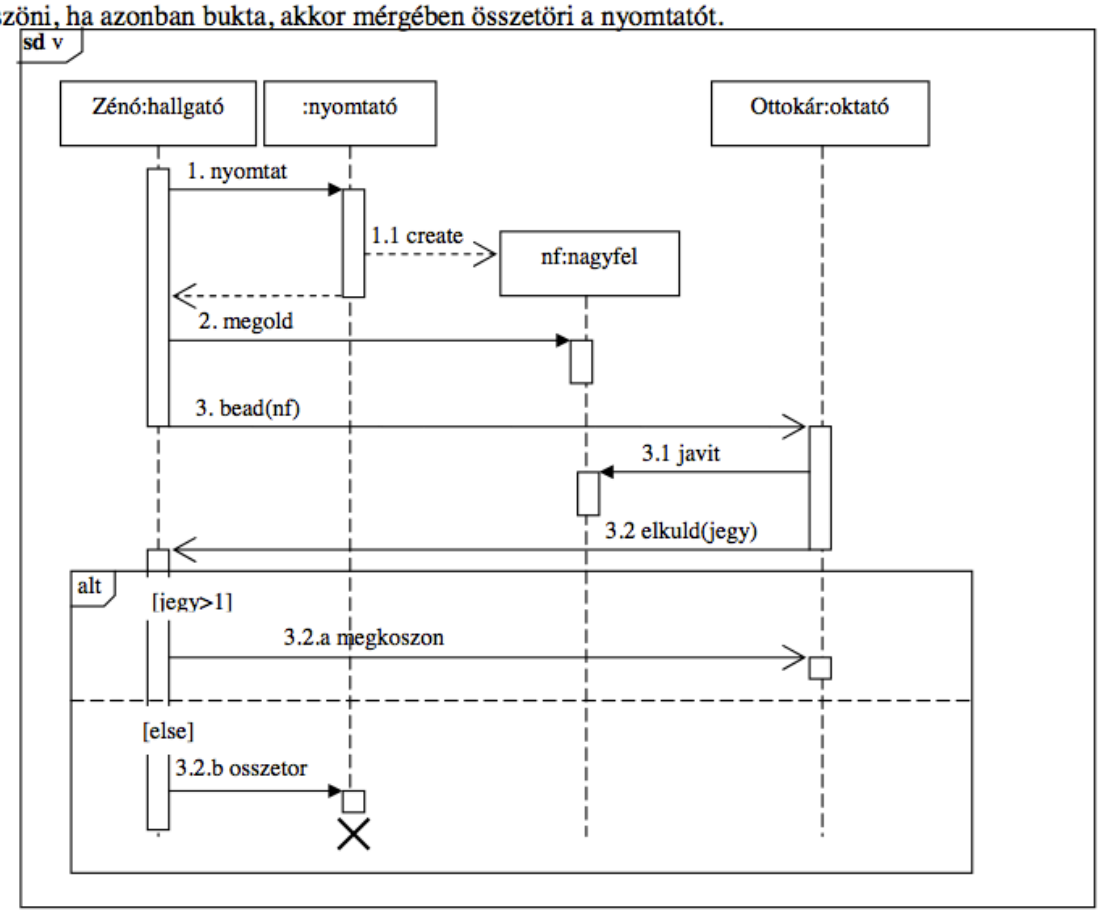


Készítsen UML2 szekvencia-diagramot az alábbi történet alapján! Ne feledkezzen el a hierarchikus számozásról sem!
Jack Bauer, a terroristák veszedelme visszatér. Információra van szüksége, ezért készíttet egy Kalman-szűrőt Chloe-val. Chloe, mielőtt nekilátna, megkérdi Jack-et, hogy megbízhat-e benne, majd elkezdi legyártani a szoftvert. Közben Jack az egyik rabot kihallgatja, és többször is megkérdezi, hogy hol vannak a fegyverek. A kihallgatás végére készül el a szűrő, amit Chloe elkezd feltelepíteni Jack PDA-jára, de közben másra is figyel. A PDA a telepítés hatására tönkremegy.

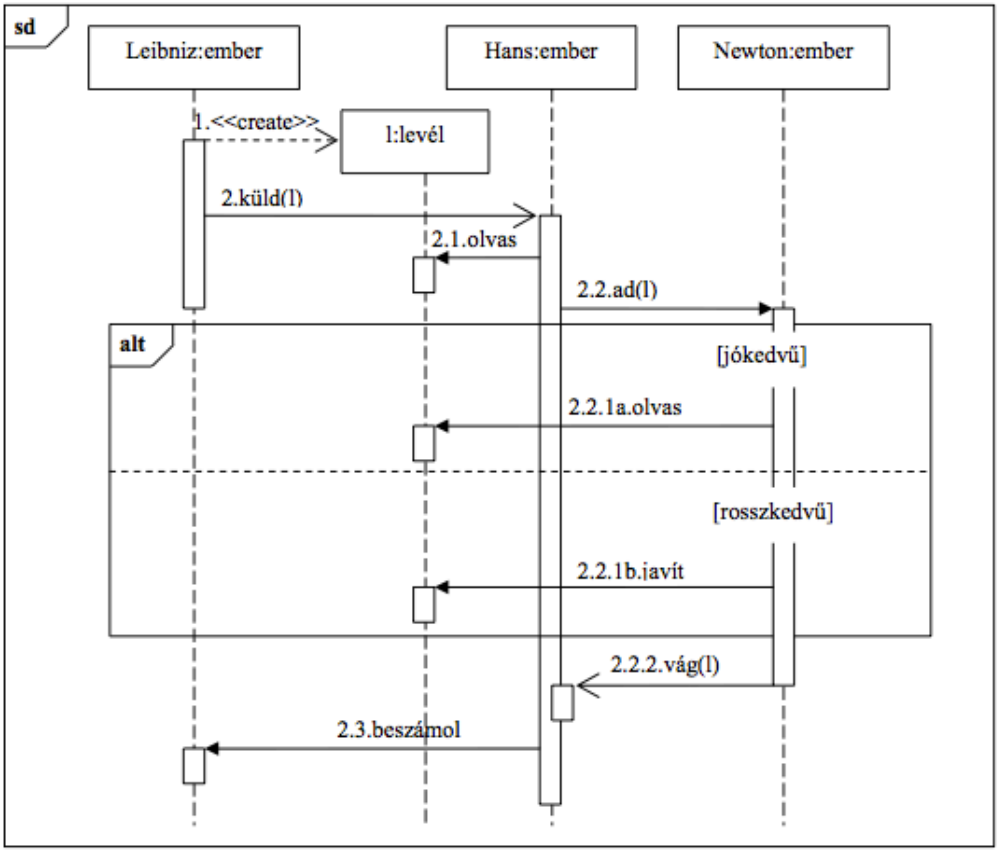


A történet alapján rajzoljon UML 2.0 szekvenciadiagramot (sequence diagram). Az üzeneteket hierarchikus számozással lássa el !

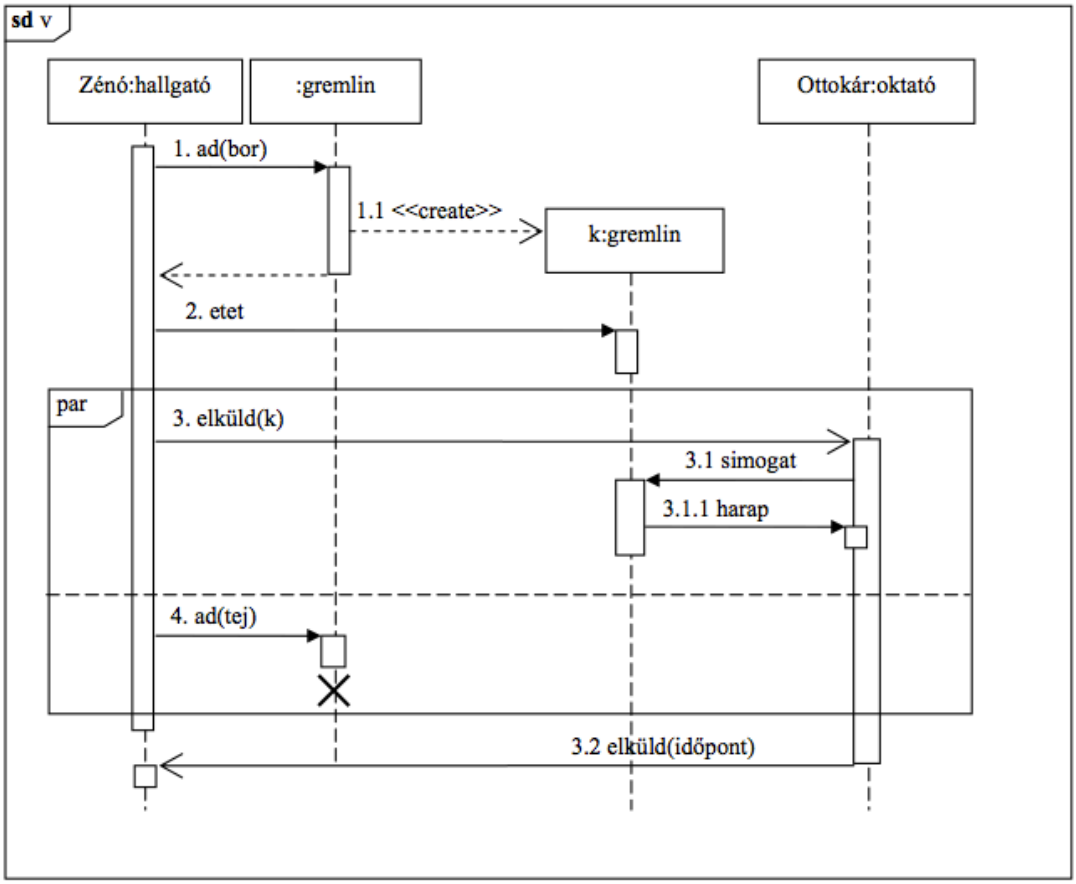
Zénó (Izidor bátyja) otthoni printerén kinyomtatja Bitgörbítés nagyfeladatát, a papíron megoldja, és elküldi Gyíkarcú Ottokárnak, a tárgy oktatójának javításra. Ottokár egyből nekilát és kijavítja a feladatot, majd ezzel a lendülettel vissza is küldi a jegyet Zénónak. Zénó, ha jobb jegyet kap, mint elégtelen, akkor választ sem várva megköszöni, ha azonban bukta, akkor mérgében összetöri a nyomtatót.



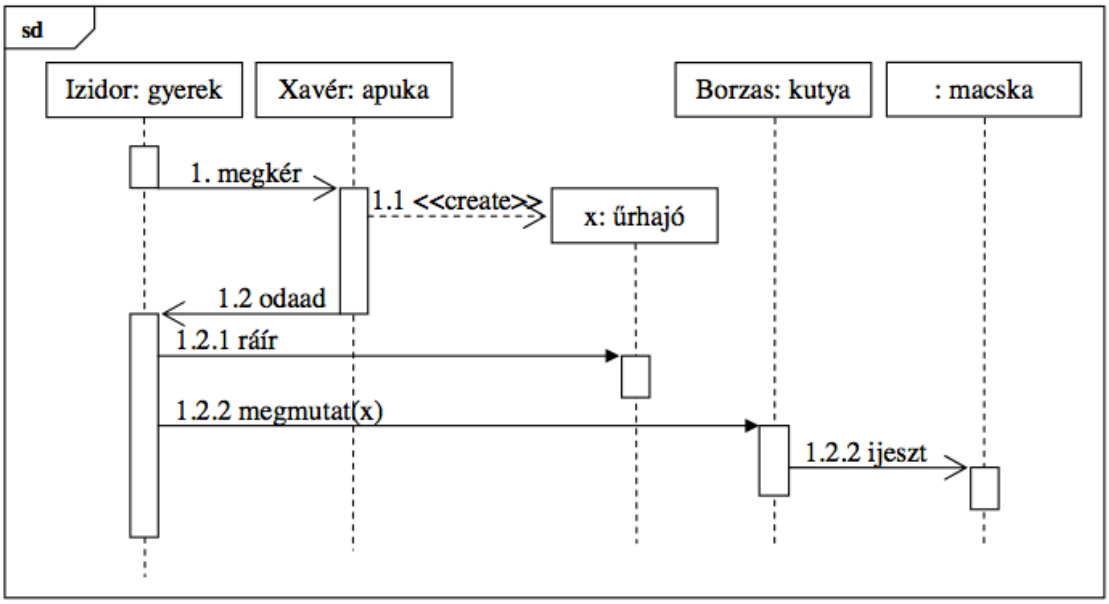
Készítsen UML2 szekvencia-diagramot az alábbi történet alapján! Ne feledkezzen el a hierarchikus számozásról sem!
Gottfried Wilhelm Leibniz szeretne tálkozni Sir Isaac Newtonnal. Ír egy latin nyelvű levelet, amelyben differenciálszámítással kapcsolatos eredményeit ecseteli. A levelet hű barátjának, az éppen Angliában tartózkodó Hans Georg von Hirscheissenfeldnek küldi azzal, hogy adja át Newtonnak. Hans kíváncsi, és elolvassa a levelet. Ezután talalálkozik Newtonnal, és odaadja neki a levelet. Ha Newton jókedvű, akkor a levelet elolvassa, ha rosszkedvű, akkor a levélben aláhúzza a nyelvtani hibákat. Mindezek után a levelet hozzávágja Hanshoz, és elsiet. Hans hazautazik, és az eseményről beszámol Leibniznek.



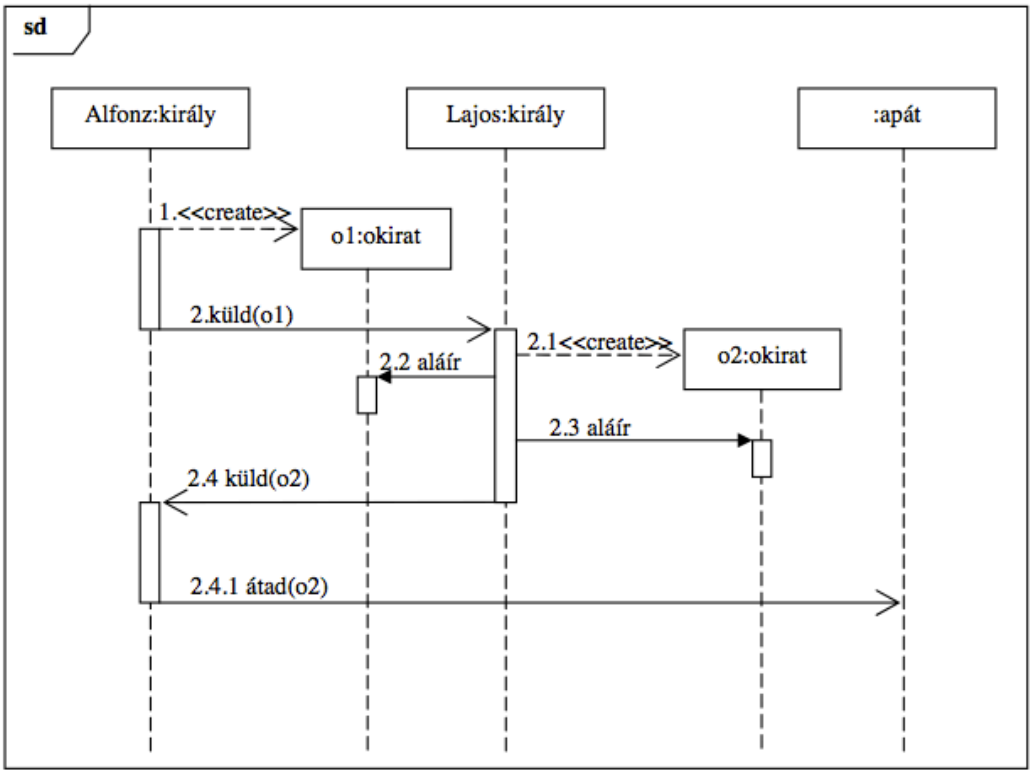
Készítsen UML2 szekvencia-diagramot az alábbi történet alapján! Ne feledkezzen el a hierarchikus számozásról sem!
Zénó a karácsonyra kapott gremlinjének véletlenül bort ad, mire az egy kisgremlinnek ad életet. Zénó nagyon megörül, és egyszer megeteti a kisgremlint, majd elküldi oktatójának, Gyíkarcú Ottokárnak házi feladat helyett. Ottokár megsimogatja a kisgremlint, aki simogatás közben megharapja. Közben Zénó az eredeti gremlinnek tejet ad, aki ebbe belepusztul. Végül Ottokár elküldi Zénónak a pótleadás időpontját..



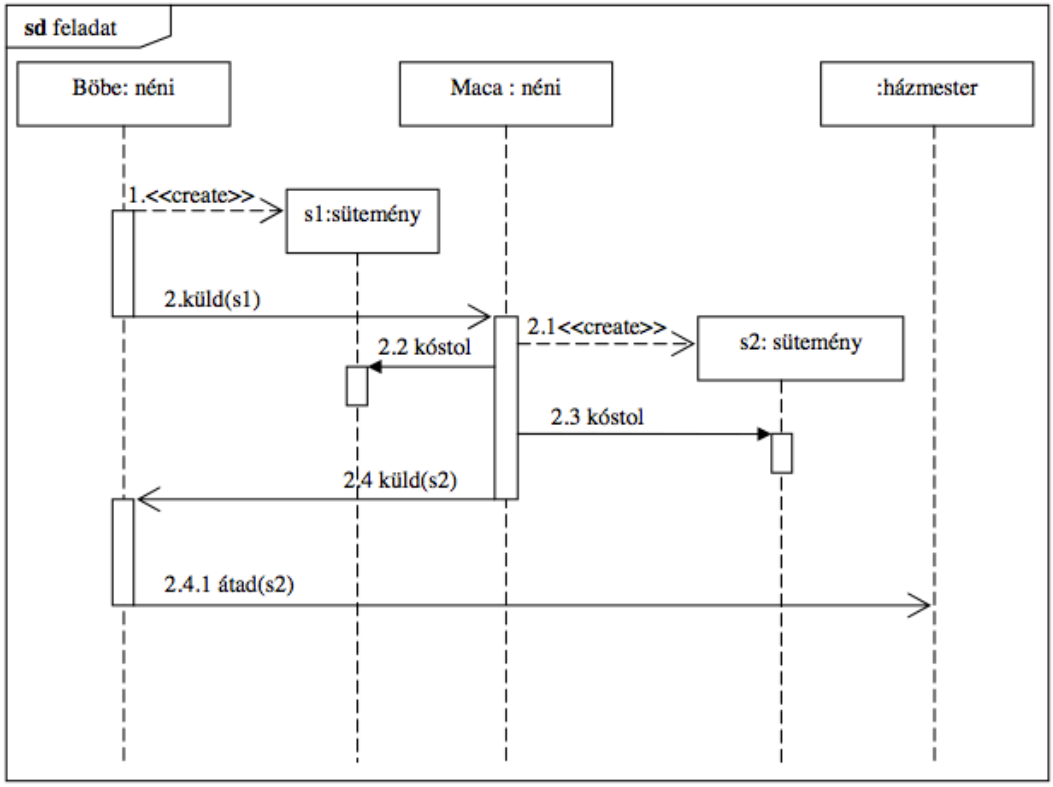
Rajzoljon UML2 **szekvenciadiagramot** ! Az üzeneteket hierarchikus számozással lássa el !
Izidor megkéri apukáját, Xavért, hogy készítsen neki egy űrhajót. Mikor Xavér kész van, az űrhajót odaadja Izidornak, majd elsiet, mert sok a dolga. Izidor az űrhajóra ráírja a nevét, majd megmutatja a legjobb barátjának, Borzas kutyának, aki a mutogatás közben megijeszt egy macskát.



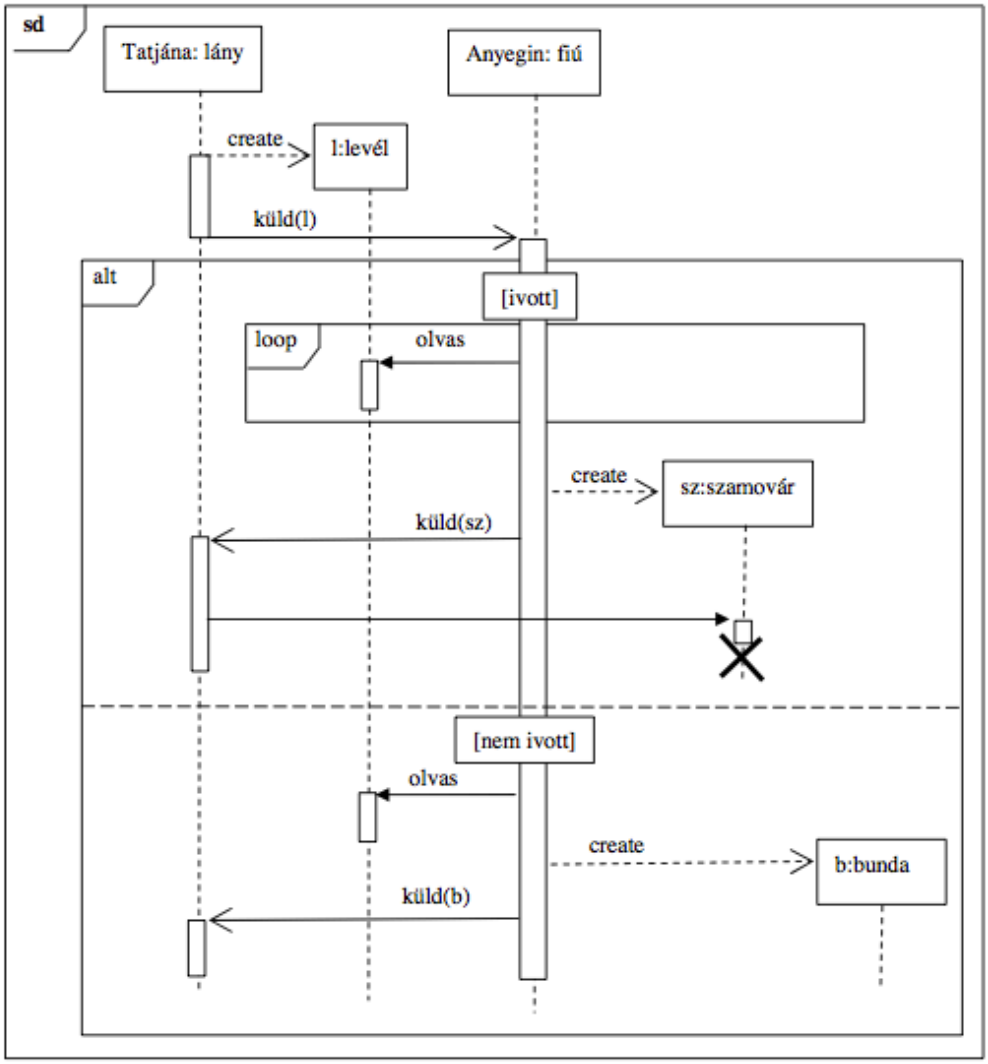
Rajzoljon UML 2 szekvenciadiagramot !
X. (Bölcs) Alfonz, Kasztília királya szövetséget kíván kötni IX. (Szent) Lajossal, Franciaország királyával. Ezért elkészít egy okiratot, amit el is küld a francia uralkodónak. Lajos, amint megkapja a dokumentumot, készíttet egy másolatot, majd mindkettőt kézjegyével látja el, és a másolatot visszaküldi Alfonznak, aki az okiratot a toledói apátnak adja át megőrzésre.



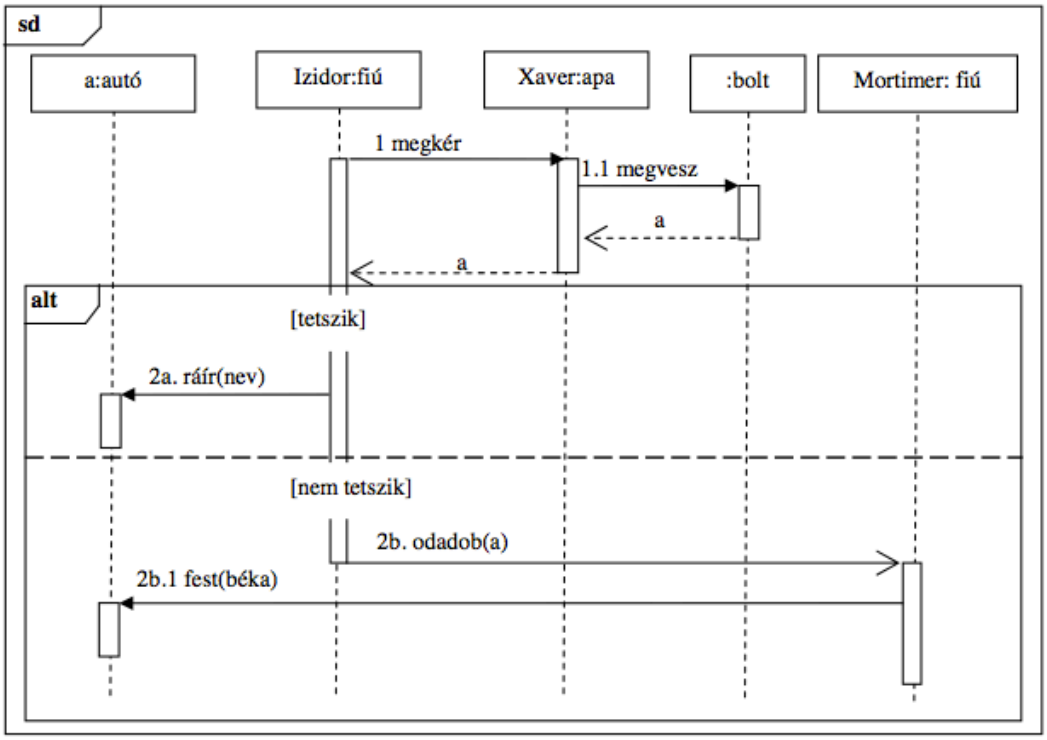
Rajzoljon UML2 **szekvenciadiagramot** ! Az üzeneteket hierarchikus számozással lássa el !
Böbe néni süt egy bejglit, és elküldi Maca néninek. Maca erre gyorsan süt egy másikat, mindkettőt megkóstolja, és a sajátját visszaküldi Böbének. Böbe a süteményt, választ sem várva, átküldi a házmesternek.



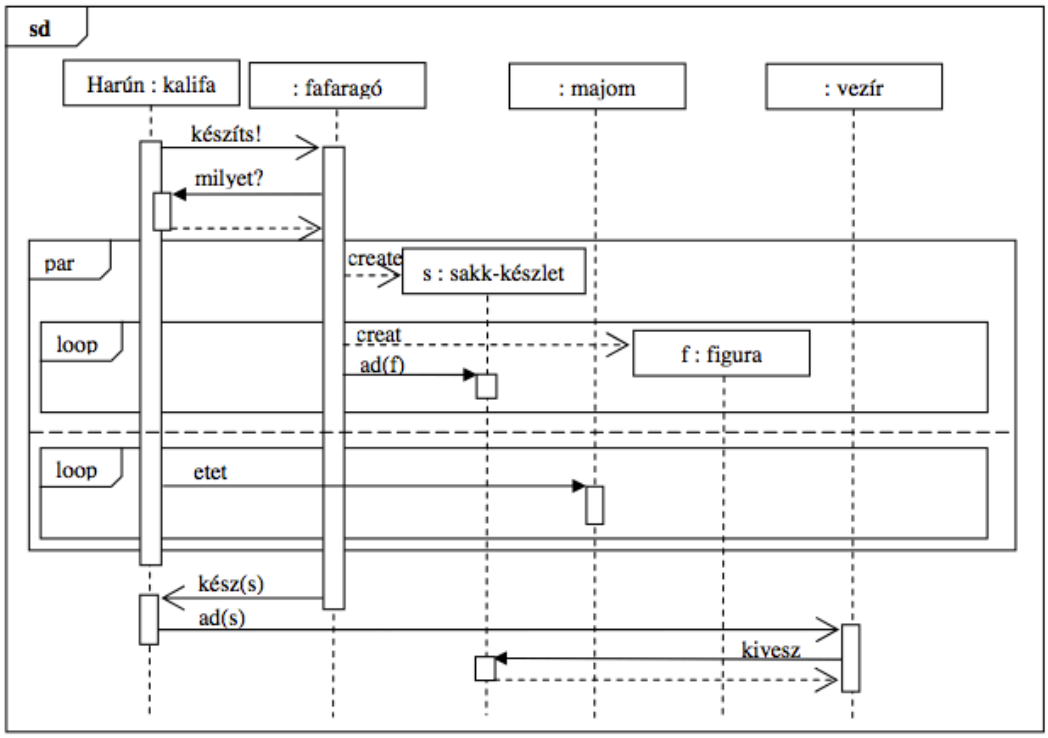
Rajzoljon UML 2 szekvenciadiagramot !
Tatjána levelet ír, majd elküldi Anyeginnek. Ha Anyegin ivott vodkát, akkor többször is elolvassa a levelet, majd készít egy gyönyörű, aranyozott szamovárt, és elküldi Tatjánának, aki (mivel nem erre számított) megsemmisíti a szamovárt. Ha azonban Anyegin nem ivott, akkor csak egyszer olvassa el a levelet, majd bundát csinál, és ezt a bundát küldi Tatjánának.



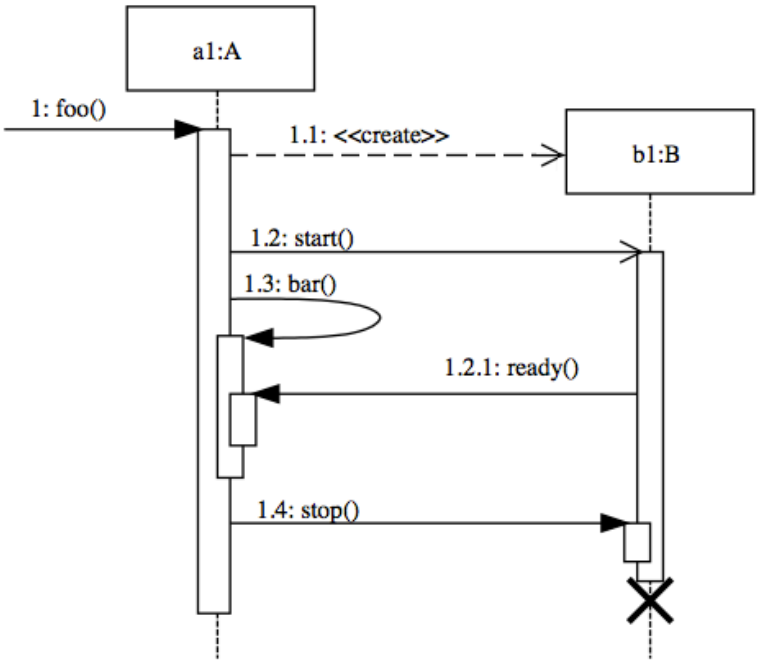
Rajzoljon UML2 **szekvenciadiagramot** ! Az üzeneteket hierarchikus számozással lássa el !
Izidor megkéri apukáját, Xavért, hogy vegyen neki egy versenyautót, mire együtt elmennek a boltba, ahol Xavér megveszi a fiának a versenyautót, majd elsiet a dolgára. Izidornak ha tetszik az autó, akkor ráírja a nevét, ha nem, akkor odadobja az öccsének, Mortimernek és elrohan. Utóbbi esetben Mortimer az autóra ráfest egy békát.



Rajzoljon UML2 szekvencia-diagramot az alábbi történet alapján !
Harún ar-Rasíd, a bölcs kalifa meg kívánja jutalmazni vezírjét, ezért készíttet egy sakk-készletet udvari fafaragójával. A fafaragó, mielőtt nekilátna, megkérdi a kalifát, milyen fából készüljön, majd elkezdi legyártani a remekművet. A készlethez egyenként faragja figurákat. Eközben a kalifa a kedvenc majmával szép sorban egyenként megeteti a kezében levő összes fügét. Mikor a sakk-készlet megvan, a faragó átadja a kalifának, aki továbbadja vezírjének. A vezír később, otthon kivesz a készletből egy fehér lovat.



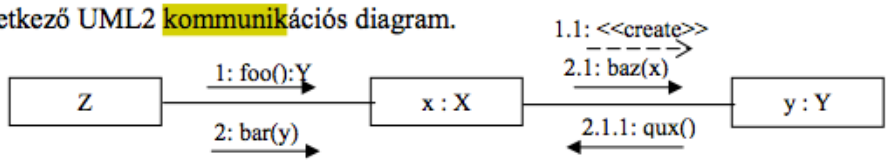
Rajzoljon UML szekvencia diagramot !
Az **A** osztály egy **a1** példánya a **foo()** metódus hatására létrehoz egy B osztályba tartozó **b1** objektumot, majd aszinkron módon meghívja annak **start()** metódusát. Ezután **a1** meghívja saját **bar()** metódusát.
A **bar()** indulása után **b1 a1**-nek szinkron módon a **ready()** üzenetet küldi. Ekkor a **bar()** véget ér, és **a1** meghívja **b1 stop()** metódusát, mire **b1** megsemmisül.



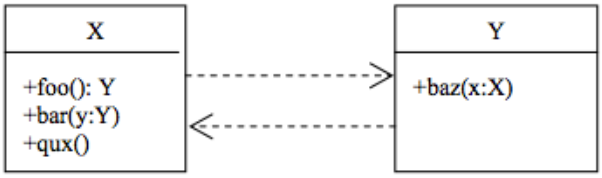
Feladatok Kommunikációs diagram témakörből

1. feladat típus (kommunikációs diagramból osztály diagram):

Adott a következő UML2 kommunikációs diagram.

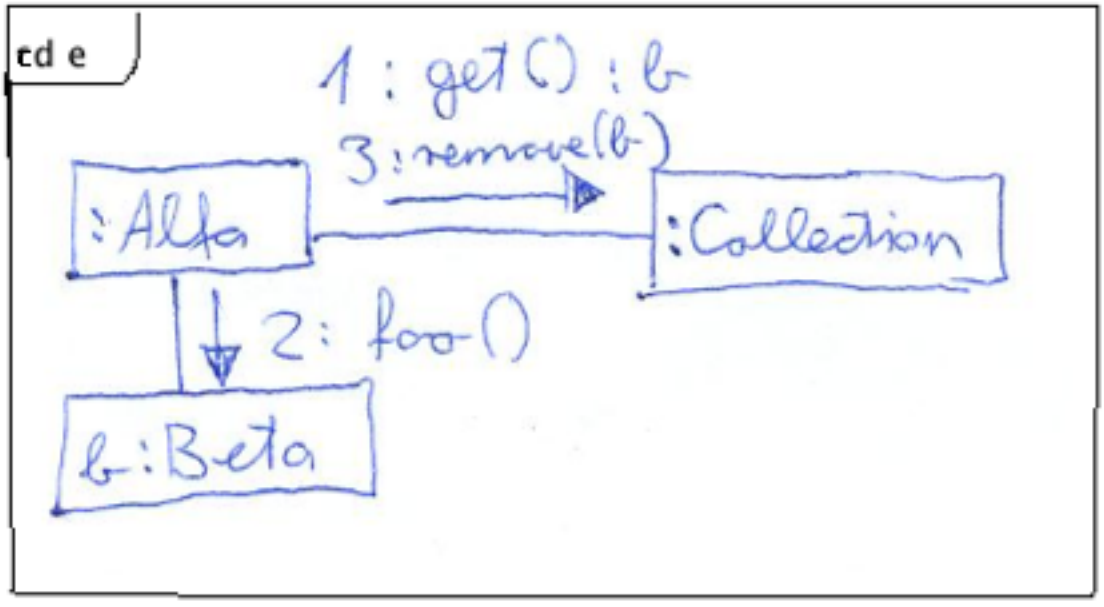


Feltételezve, hogy a fenti diagramon szereplő objektumok osztályainak nincsenek – a diagramból nem kiolvasható – további metódusai, közöttük nincs más egyéb kapcsolat (pl. öröklés), az alábbi ábrát korrekt UML2 osztálydiagrammá alakítva ábrázolja az osztályokat a metódusok szignatúráival együtt, valamint a két osztály közötti kapcsolatot !

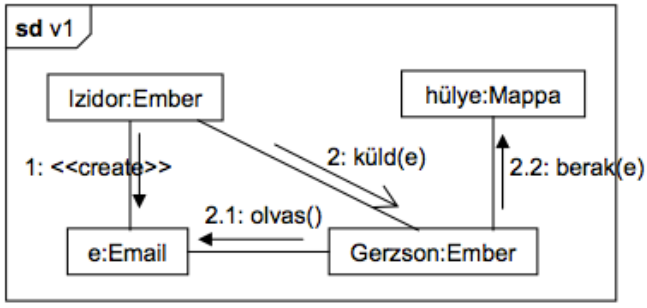


2. feladat típus (rajzolás):

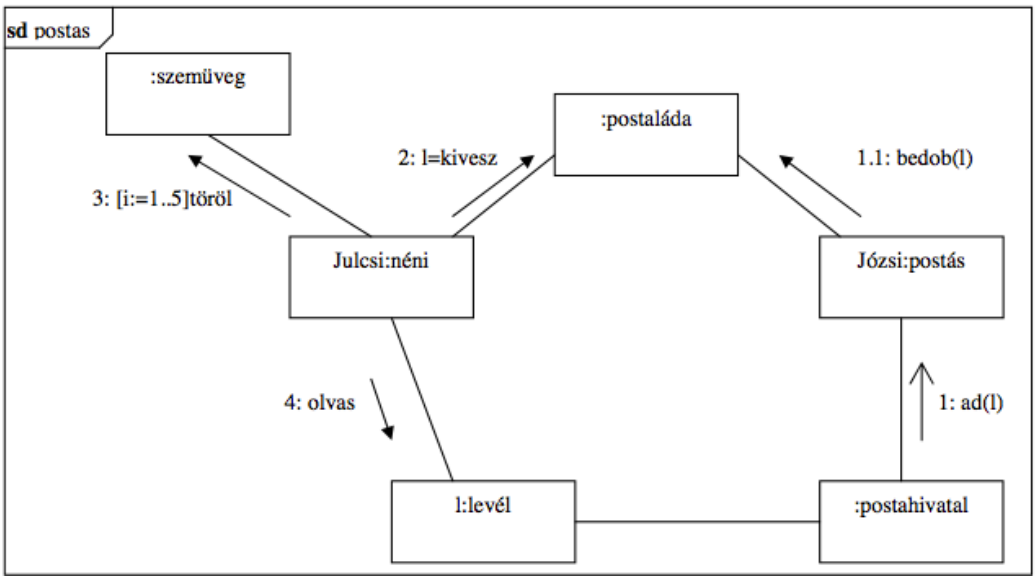
Egy Alfa osztálybeli objektumnak van egy Beta osztályú elemekből álló kollekciója. Ebből a get() metódussal kiveszi az egyik elemet, amin végrehajtja a foo() műveletet. Végül a remove() metódussal törli az elemet a gyűjteményből. Rajzoljon UML kommunikációs diagramot !



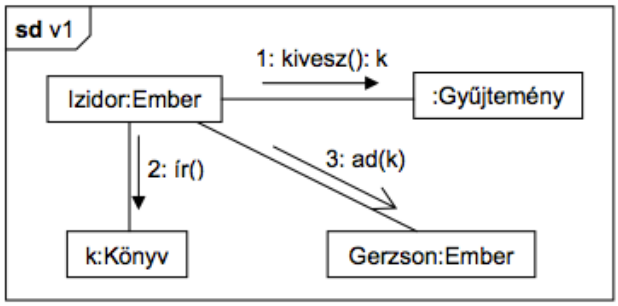
Izidor ír egy e-mailt, amit elküld Gerzsonnak. Gerzson a levelet elolvassa, majd beteszi a „hülye” mappába. Rajzoljon UML2 kommunikációs diagramot ! Alkalmazzon hierarchikus számozást !



A postahivatal Józsinak, az énekes postásnak adja a Julcsi néninek szóló levelet. A postás a levelet bedobja a címzett postaládájába. Julcsi néni este, hazamenet, kiveszi a postaládából a levelet, majd ötször megtörli a szemüvegét (az utcán esett az eső), és elolvassa levelet. Rajzoljon UML2 kommunikációs diagramot !

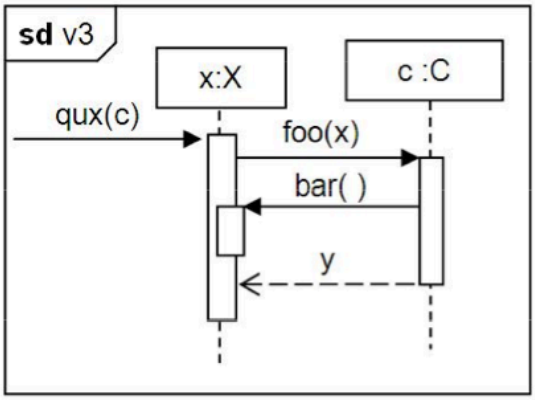


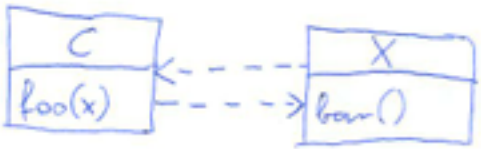
Izidor kiveszi a könyvgyűjteményéből kedvenc könyvét, ajánlást ír bele, majd odaadja Gerzsonnak. Rajzoljon UML2 kommunikációs diagramot !

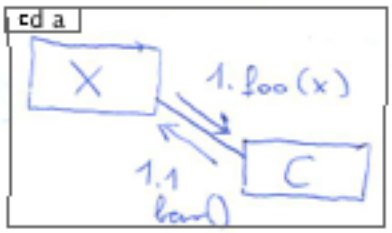


3. feladat típus (szekvencia d -> osztály és komm. diagram rajzolás):

Tételezze fel, hogy az alábbi UML2 szekvencia-diagramon szereplő objektumok osztályai között nincs más egyéb – a diagramból nem kiolvasható – kapcsolat (pl. öröklés) ! Rajzoljon osztálydiagramot ! Rajzoljon kommunikációs diagramot !

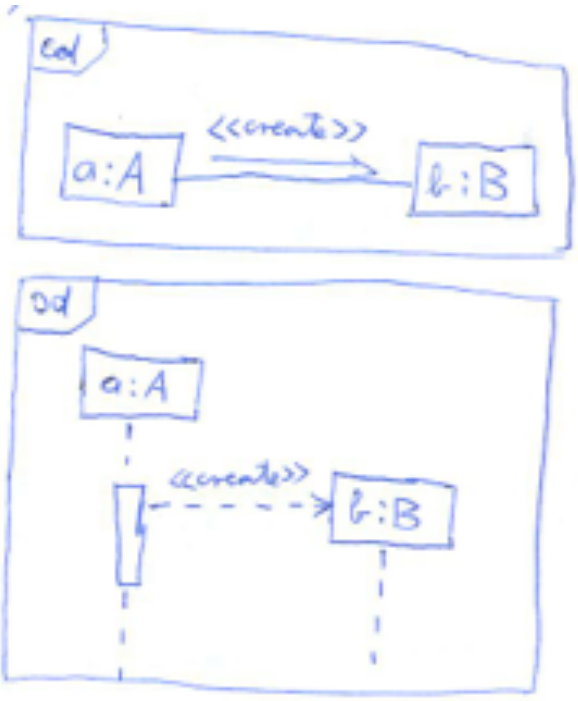






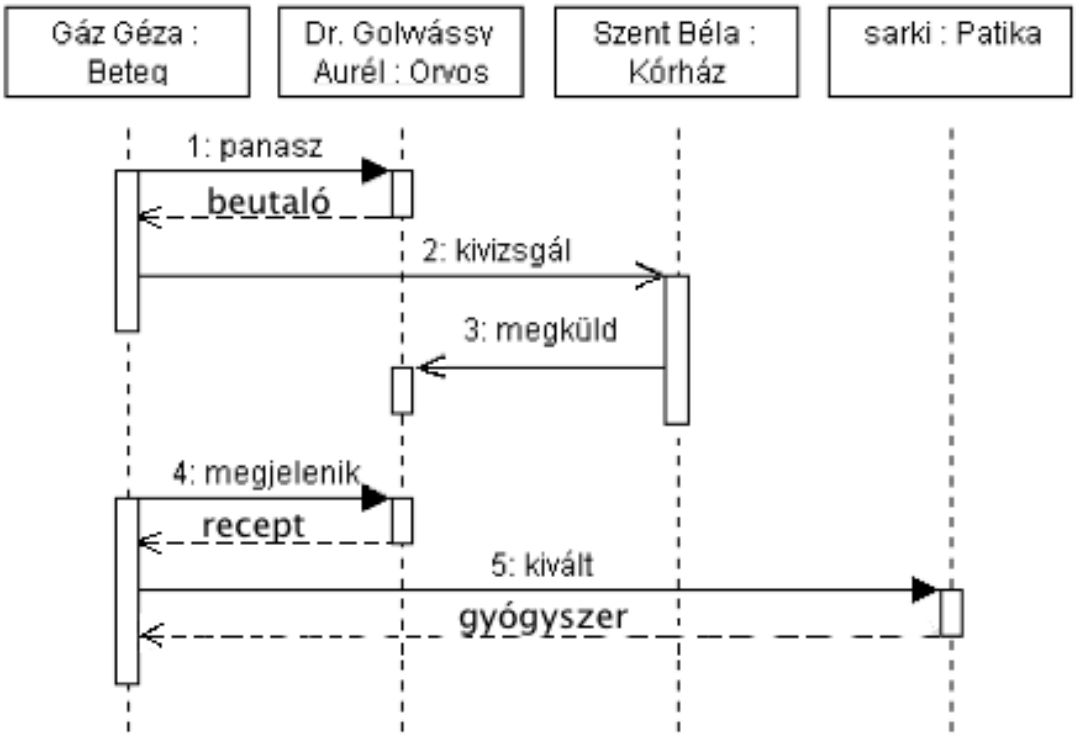
4. feladat típus (komm. és szekvencia rajz):

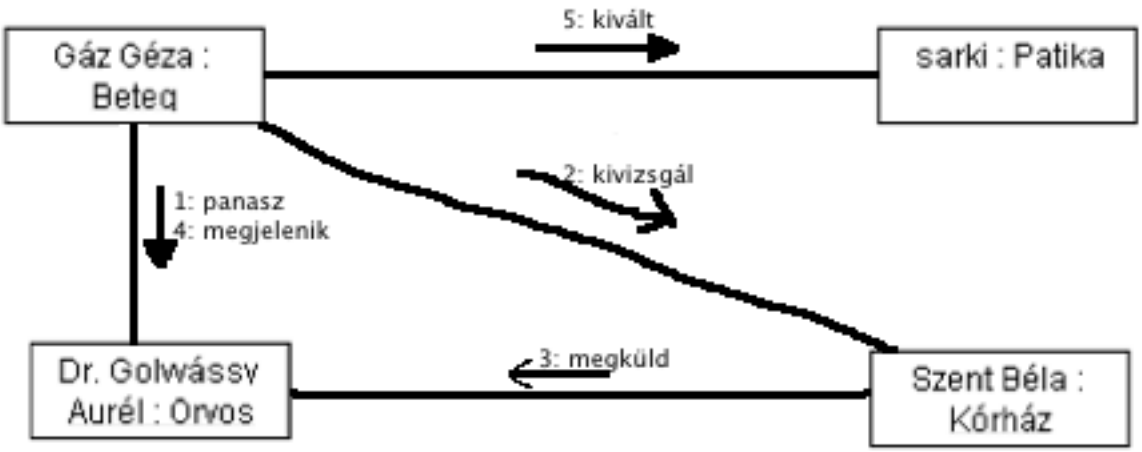
Az A osztály „a” példánya készít egy B osztály „b” példányát. Rajzoljon kommunikációs és szekvencia diagramot !



-

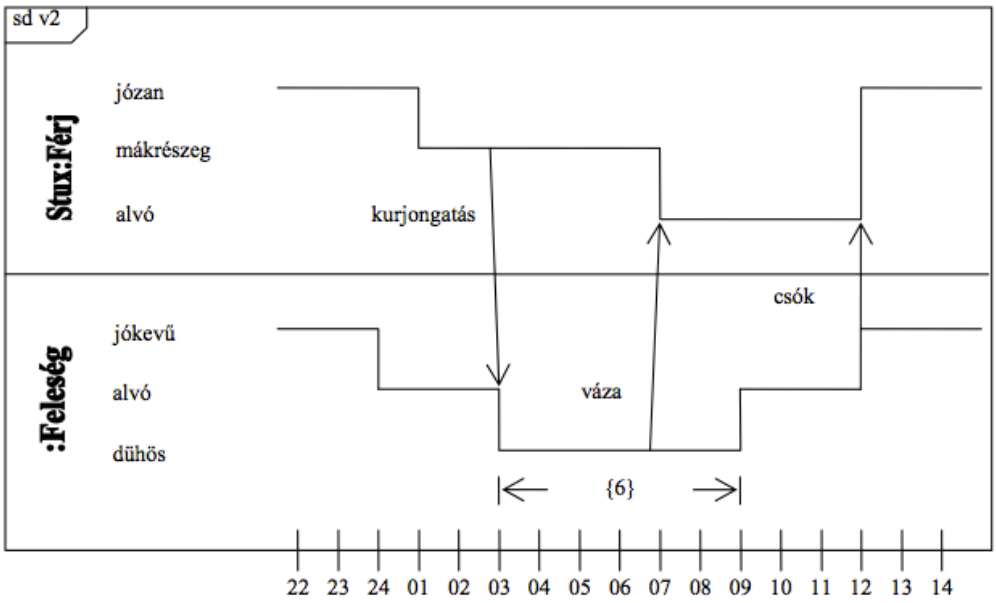
Gáz Géza rosszul érzi magát, ezért elpanaszolja a háziorvosának, hogy már két hete lázas, és a jobb lábfeje a kétszeresére dagadt. Dr. Golyvássy Aurél, az orvos beutalót ad Gáz úrnak a Szent Béla kórházba, ahol vért vesznek, és a levett vért vizsgálat alá vetik. Pár nap múlva, a vizsgálat befejeződésekor az eredményt e-mailen megküldik Dr. Golyvássynak. Amikor Gáz úr ismét megjelenik a rendelésen, a doktor receptre felír Gáz úrnak köptetőt, amelyet ő kivált a sarki gyógyszertárban. Rajzoljon az eseményekről UML szekvencia és kommunikációs diagramot !



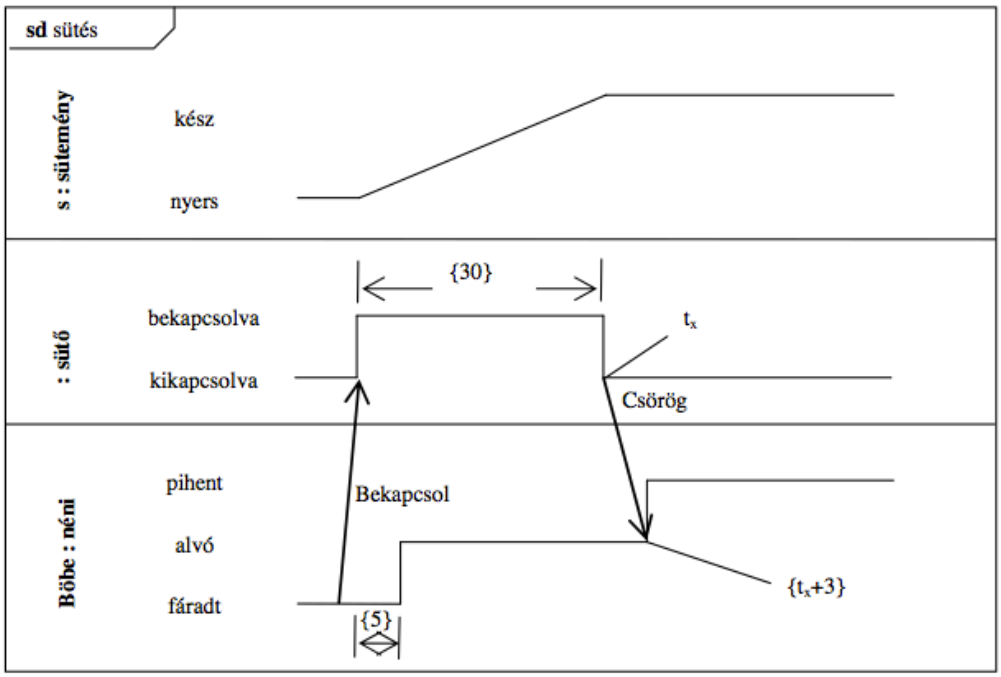


Feladatok Timing Diagram témakörből

Az alábbi történet alapján rajzoljon UML 2.0 időzítési diagramot (timing diagram)!
Stux este 10-kor józanul ment el otthonról, egyedül hagyva jókedvű feleségét, aki 12-kor elaludt. Stux hajnali 1-re lett mákrészeg, és hajnali 3-ra ért haza. Ekkor a kurjongatásra felesége dühösen ébred, és hat órán át dühös is marad. 7-kor felesége egy vázát vág hozzá, amitől Stux elalszik. Miután az asszony kidühöngte magát, elalszik. Mikor délben jókedvűen felébred, megcsókolja férjét, aki józanul ébred.



Az alábbi történet alapján rajzoljon UML 2 időzítési diagramot (timing diagram)!
Böbe néni fáradt, de másnapra süteményt kell sütnie. A nyers süteményt beteszi a sütőbe, amit aztán bekapcsol. Öt perc múlva Böbe néni elalszik. A sütő fél óra elteltével kikapcsol és csörög, amire Böbe néni 3 perc múlva kipihenten felébred, és kiveszi a kész süteményt.
Böbe néni diszkrét állapotai: fáradt, alvó, pihent. A sütő diszkrét állapotai: ki, be. A sütemény folytonosan változik a nyers és a kész között.



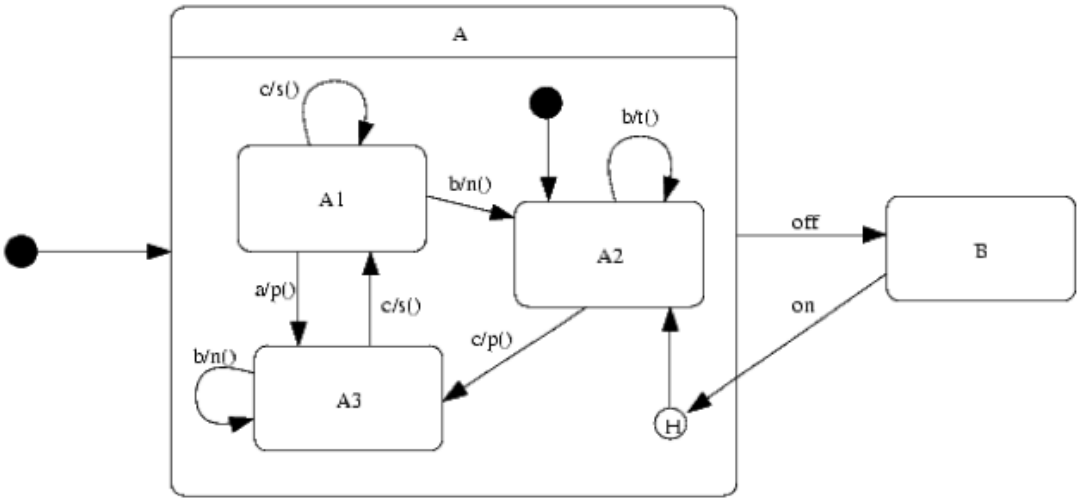
Feladatok Állapot Diagram témakörböl

1. feladat típus (állapottábla):

Legyen egy objektumunk, amely két fő állapottal (A, B) rendelkezik. Az A állapotban a következő állapotgép működik:

	a	b	c
A1	A3/p()	A2/n()	A1/s()
A2	-	A2/t	A3/p()
A3	-	A3/n()	A1/s()

Az A állapotból B-be az off esemény hatására kerül. Visszatérni az on-ra fog, és ekkor ott folytatja, ahol a kilépéskor abbahagyta. Kezdetben az A állapot aktivizálódik, az A-n belül pedig az A2. Ugyancsak A2 az on-ra vonatkozó predefinit állapot.
Rajzolja meg a objektum **UML state-chart**-ját !



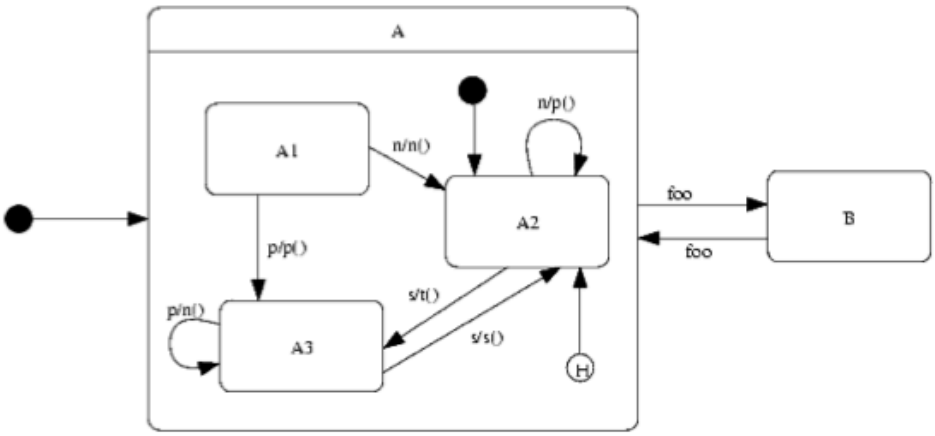
Adva van egy objektumunk, amely két fő állapottal (A, B) rendelkezik. Az A állapotban a következő állapotgép működik:

	p	n	s
A1	A3/p()	A2/n()	-
A2	-	A2/p()	A3/t()
A3	A3/n()	-	A2/s()

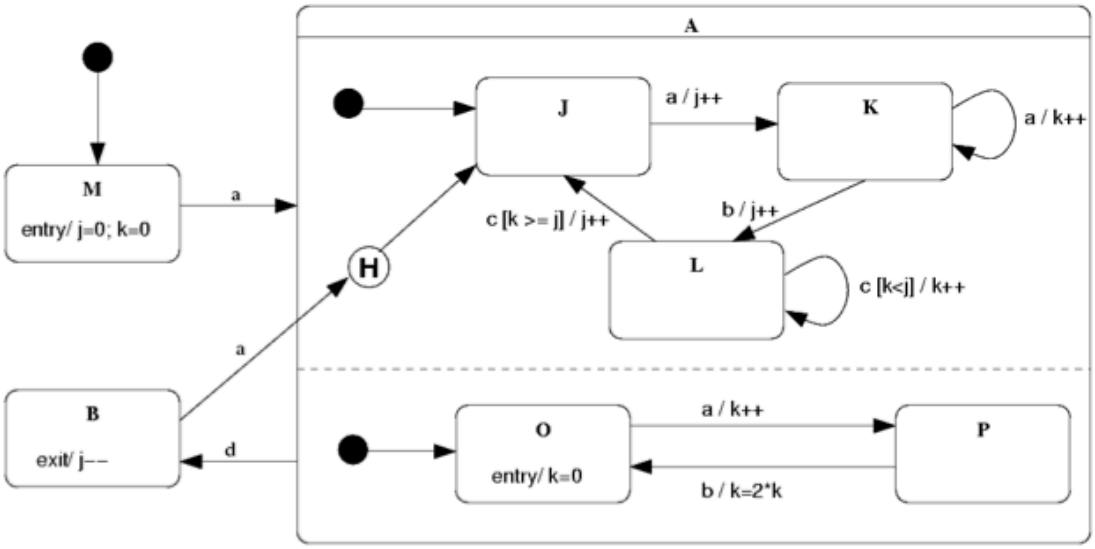
Az A állapotból B-be a foo esemény hatására kerül. Visszatérni ismét a foo-ra fog, és ekkor ott folytatja, ahol a kilépéskor abbahagyta. A predefinit kezdő állapot az A, azon belül A2.

Rajzolja meg a objektum **UML state-chart**-ját

A és B állapot 1 pont
A belső állapotai 1 pont
Induló állapotok 1 pont
Össz átmenet (8) 2 pont
History indikátor 1 pont
HI inicializálása 1 pont



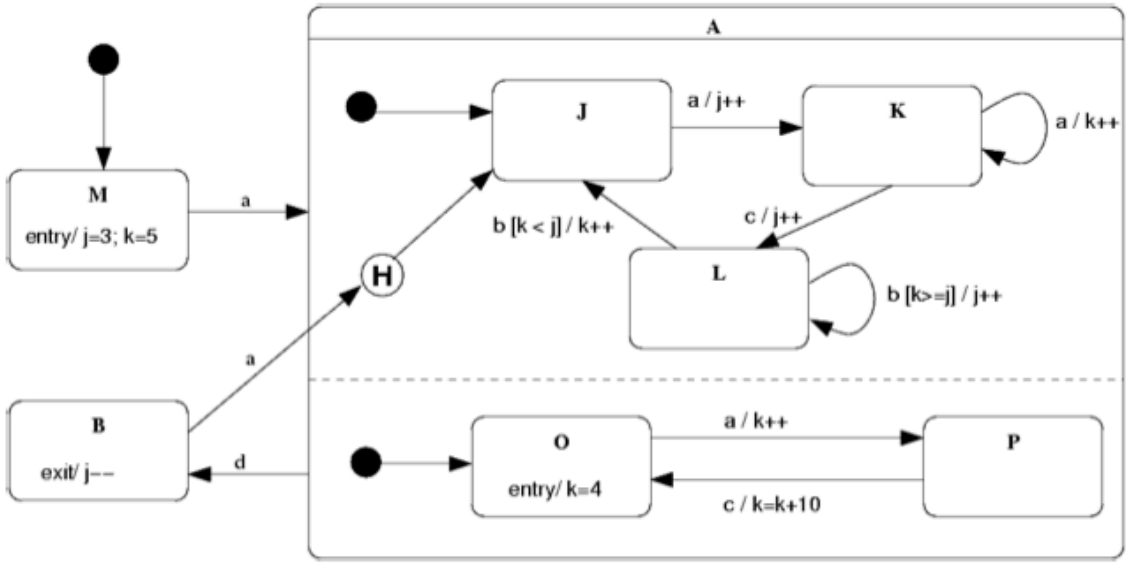
Az alábbi UML statechart egy objektum viselkedését írja le.



Az alábbi táblázat kitöltésével adja meg, hogy a táblázat első oszlopában álló események hatására mennyi lesz a `j` és `k` változó értéke, valamint mi lesz a következő állapot ! A táblázat első oszlopa az objektumot egymás után ért eseményeket mutatja. Az első sorban szereplő esemény előtt az objektum még nem kapott eseményt. A feladat a táblázat üres helyeinek kitöltése az adott sorban szereplő esemény teljes feldolgozása után.

esemény	j	k	állapot
a	0	0	A(J,O)
a	1	1	A(K,P)
a	1	2	A(K,P)
b	2	0	A(L,O)
c	2	1	A(L,O)
a	2	2	A(L,P)
d	2	2	B
a	1	0	A(L,O)
c	1	1	A(L,O)
c	2	1	A(J,O)

Az alábbi UML statechart egy objektum viselkedését írja le.

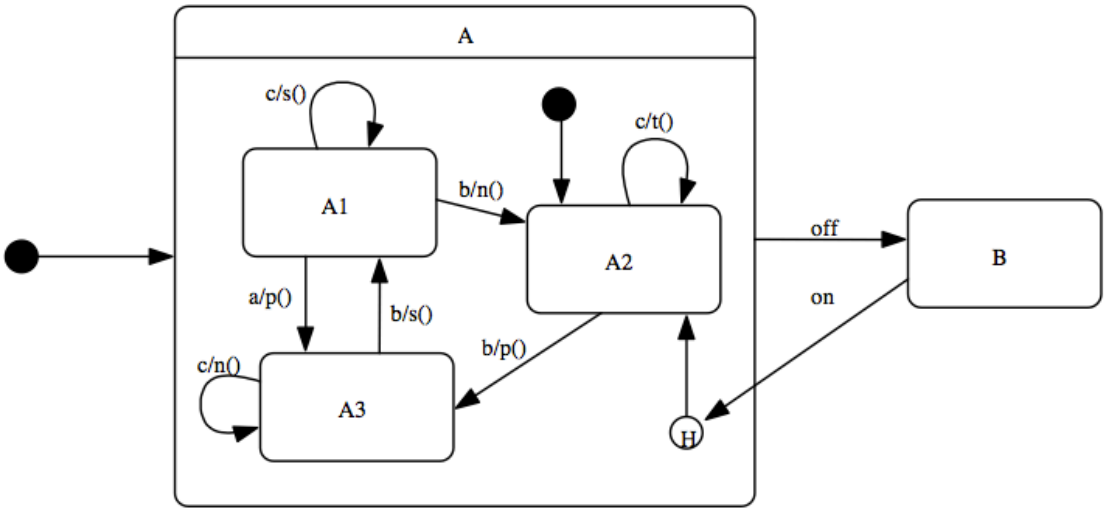


Az alábbi táblázat kitöltésével adja meg, hogy a táblázat első oszlopában álló események hatására mennyi lesz a `j` és `k` változó értéke, valamint mi lesz a következő állapot ! A táblázat első oszlopa az objektumot egymás után ért eseményeket mutatja. Az első sorban szereplő esemény előtt az objektum még nem kapott eseményt. A feladat a táblázat üres helyeinek kitöltése az adott sorban szereplő esemény teljes feldolgozása után.

esemény	j	k	állapot
a	3	4	A(J,O)
a	4	5	A(K,P)
c	5	4	A(L,O)
a	5	5	A(L,P)
b	6	5	A(L,P)
d	6	5	B
a	5	4	A(L,O)
a	5	5	A(L,P)
b	6	5	A(L,P)
b	6	6	A(J,P)

Legyen egy objektumunk, amelynek két fő állapota (**A**,**B**) van. Az **A** állapotban a táblázatban megadott állapotgép működik. Az **A** állapotból **B**-be az 'off' esemény hatására kerül. Visszatérni az 'on'-ra fog, és ekkor ott folytatja, ahol a kilépéskor abbahagyta. Kezdetben az **A** állapot aktivizálódik, az **A**-n belül pedig az **A2**. Rajzolja meg az objektum UML state-chart-ját!

	a	**b**	**c**
A1	A3/p()	A2/n()	A1/s()
A2	-	A3/p()	A2/t()
A3	-	A1/s()	A3/n()

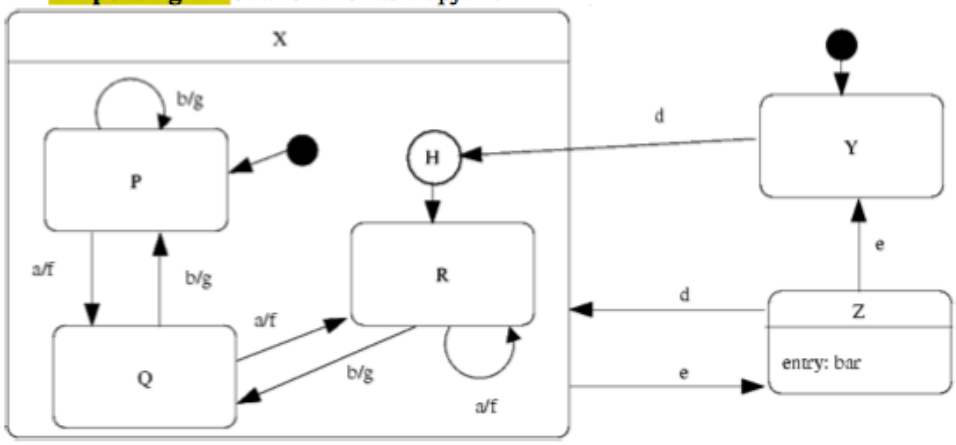


Az Antipheca WC-tisztító robot szenzorai a beérkező jeleket `a`, `b`, `d`, `e` eseményként továbbítják a robot agyába. Ez utóbbiban egy állapotgép működik, melynek `X`, `Y` és `Z` főállapotai vannak. Az `X` főállapot belső működését a következő állapottábla írja le:

	a	b
P	Q/f	P/g
Q	R/f	P/g
R	R/f	Q/g

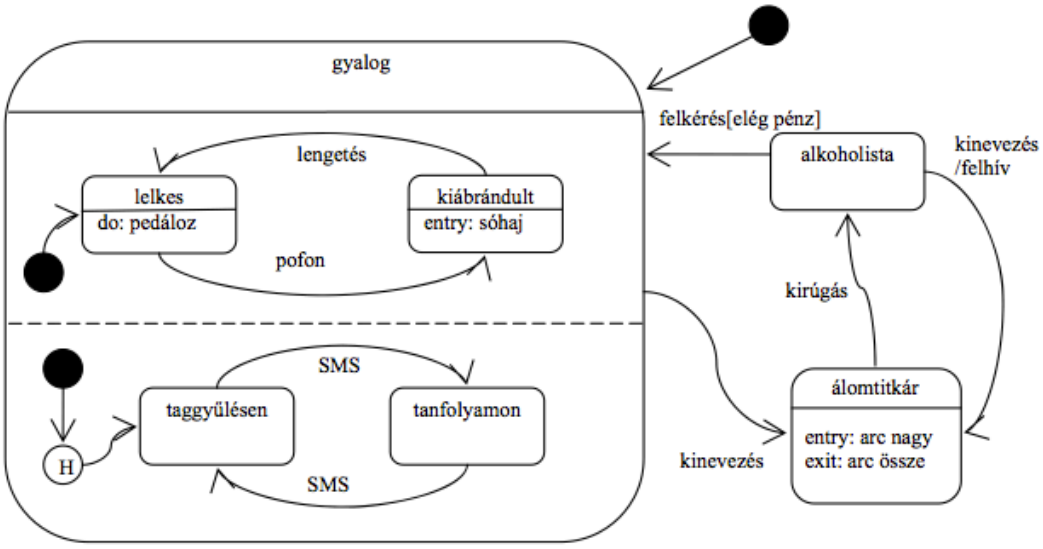
`X`-ben a kezdőállapot a `P`. Ha `X`-ben az `e` esemény történik, akkor a `Z` állapotba jutunk. Az `Y` állapotból `d` esemény bekövetkeztére az `X` legutóbbi állapotába jutunk. Ha még nem jártunk `X`-ben, akkor `R`-be. `Z`-ből `d` eseményre `X`-be, `e` eseményre `Y`-ba jutunk. `Z`-be való belépéskor mindig lefut a `bar` akció. A főállapotok közül a kezdőállapot az `Y`.

Rajzoljon **UML állapotdiagramot** a fenti leírás alapján !

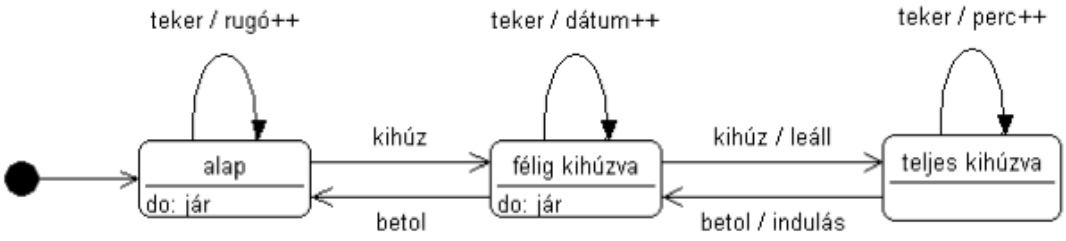


2. feladat típus (rajz):

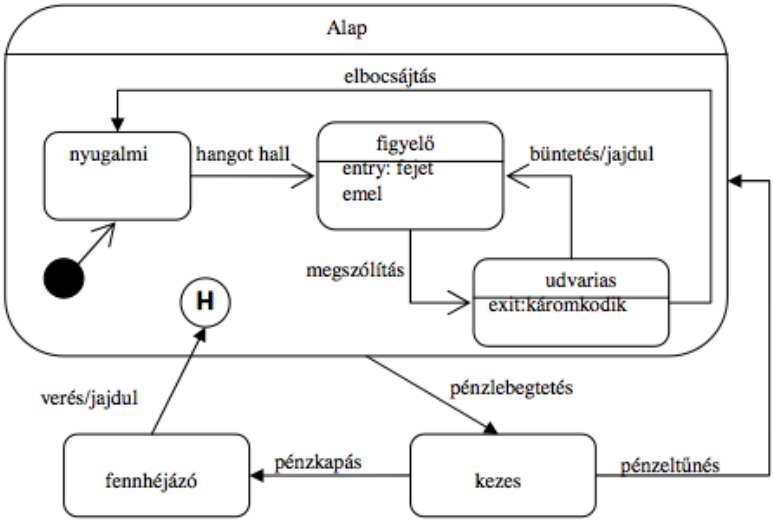
Rajzoljon UML 2.0 állapotábrát (state chart) az alábbi történet alapján!
A Stupiditas nevű szervezet tagja *gyalog*ként (másként paraszt) kezdi pályafutását. Először nagyon *lelkes*, ilyenkor folyton pedálozik. Ha nagy pofont kap, *kiábrándul* (és elsóhajtja magát). Némi állami támogatás belengetésével ismét lelkes lesz (és pedálozik). Mindeközben (vagyis hogy éppen lelkes vagy kiábrándult), csak két összejövetelen lehet megtalálni: *taggyűlésen* és *tanfolyamon* (a taggyűlés az első). Ha az egyiken SMS-t kap, átmegy a másikra. Mikor kinevezik *álomtitkár*nak, akkor maga mögött hagyja a gyalogos életet (hiszen nagy fekete autót is kap). Álomtitkárként először az arca lesz nagy. Amikor kirúgják „állásából", az arca összemegy és *alkoholista* lesz. Ekkor felkérésre, ha elég pénzt kap (az alkoholizmust levetkőzve) ismét gyalog lesz. Itt mindenképpen lelkesen azon az összejövetelen folytatja, ahol utoljára gyalogként megfordult. Az alkoholizmusból egy újabb álomtitkári kinevezés is kigyógyítja. Ilyenkor felhívja anyukáját.



A Schmalling óragyár gyártja a világhírű NeverStops karórát. Ennek jobb oldalán található az ún. korona. Ezt kihúzni, betolni, illetve tekerni lehet. Kezdetben a korona teljesen be van tolva. Ha ilyenkor megtekerjük a koronát, akkor az órát felhúzzuk. Ha a koronát egy kicsit kihúzzuk, akkor a tekerésre a dátumot jelző lapka egyet-egyet ugrik. Ha a koronát még jobban kihúzzuk (ennél jobban nem is lehet), a mutatókat tudjuk a tekeréssel állítani. Az óra folyamatosan jár, kivéve azt az esetet, amikor a koronát teljesen kihúztuk; ilyenkor addig áll, míg a koronát beljebb nem toljuk.
Rajzoljon **UML state-chart**-ot !

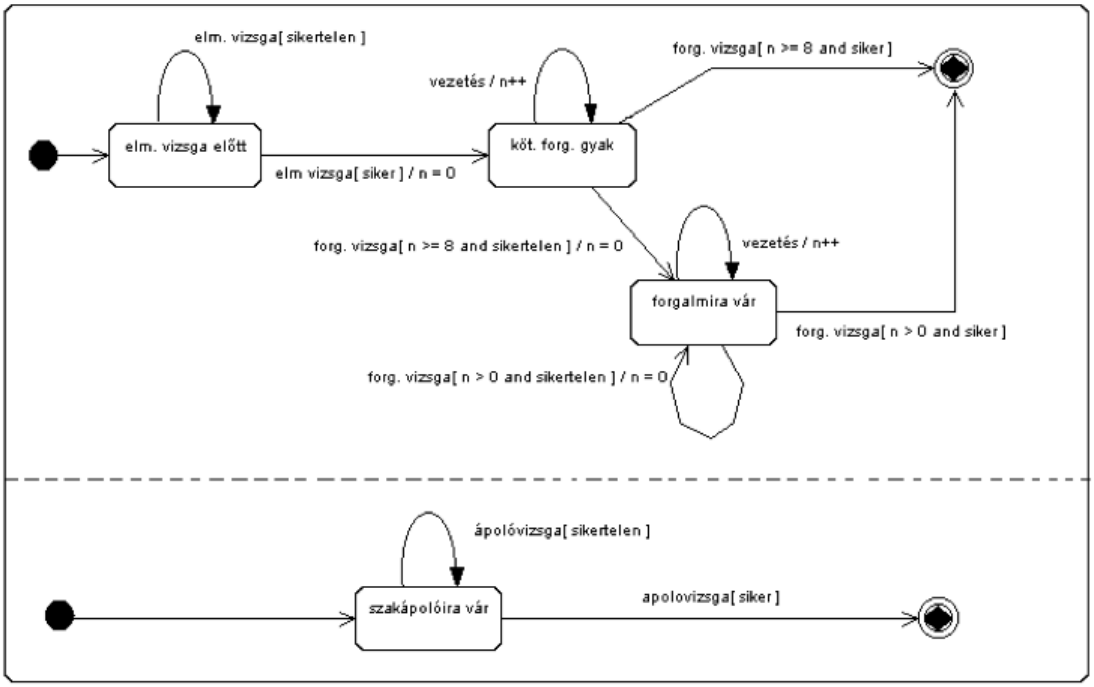


Az alábbi történet alapján rajzoljon a diagramba UML 2 állapotábrát (state chart)!
A Titanon létezik egy Lowyir-nek keresztelt parazita életforma. A Lowyir életét nyugalmi állapotban kezdi. Ha hangot hall, figyelő állásba lép (figyelő állás kezdetén mindig felemeli a fejét). Ha ekkor megszólítják, akkor udvarias lesz. Udvariasságából két módon lehet kimozdítani. Elbocsájtással, amire ismét nyugalmi állapotba kerül, vagy büntetéssel, ekkor megint figyelő állásba lép, de előtte feljajdul. Az udvarias állapotból való kizökkentéskor mindig elkáromkodja magát. Bármely fenti állapotban is volt, ha pénzt lebegtetnek meg előtte, akkor kezessé válik, ha a pénz eltűnik, a nyugalmi helyzetét veszi fel. Ha kezes és a pénzből kap, akkor fennhéjázó lesz. Fennhéjázása csak akkor szűnik meg, ha megverik, ekkor feljajdul, de aztán ott folytatja, ahol a pénz meglebegtetése előtt abbahagyta.

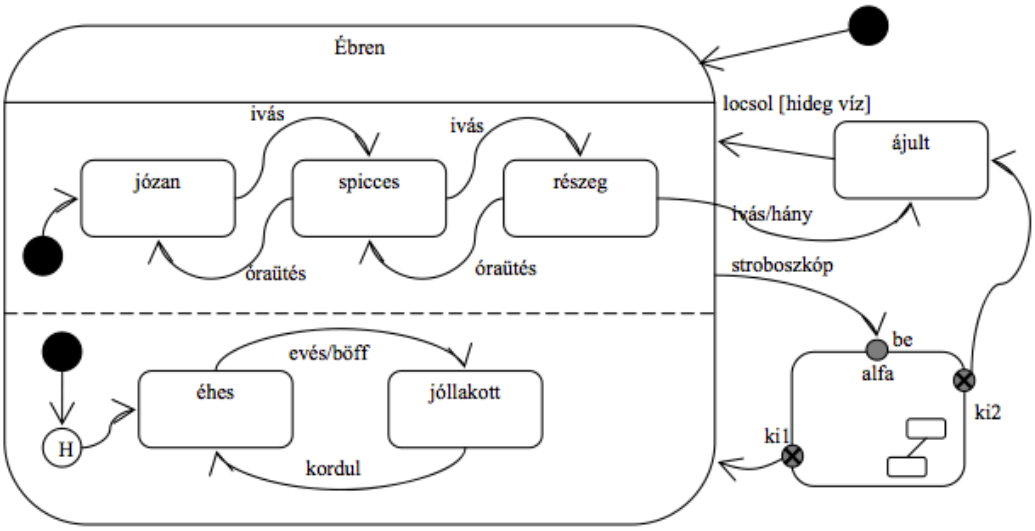


Rajzolja meg a hallgató viselkedését leíró UML state-chartot ! Az állapot-átmeneteket pontosan definiálja.

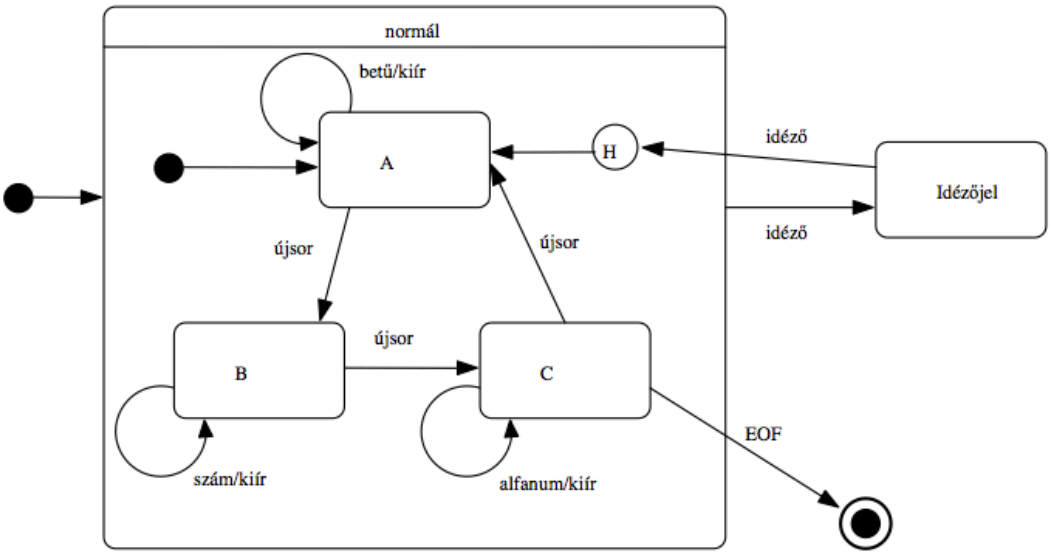
A zöldlámpás autók vezetésére jogosító képzésben résztvevő hallgatónak a zöldlámpa használatára vonatkozó elméleti vizsga letétele után legalább 8 alkalommal kell a forgalomban felügyelet mellett vezetési gyakorlatot végezni, majd ezt követi a forgalmi vizsga. A jogosítvány megszerzéséhez ugyancsak szükséges a szakápolói vizsga, amelyet a hallgató a felvételt követően bármikor letehet. Zöldlámpás jogosítványt akkor kap, ha a forgalmi és ápolói vizsgát egyaránt letette. Valamennyi sikertelen vizsga korlátlan számban ismételhető. Az ismételt forgalmi vizsgának előfeltétele, hogy legalább 1 alkalommal vezetési gyakorlatot kell végezni.



Rajzoljon UML 2.0 állapotábrát (state chart) az alábbi történet alapján!
Stux ébren háromféle hangulatban lehet: józan, spicces, részeg. Ezen kívül (szintén ébren) lehet éhes vagy jóllakott. Értelemszerűen, ha józan és iszik, akkor spicces, ha megint iszik, részeg lesz. Óraütésre visszafele változik. Ha éhes és eszik, akkor elböffenti magát és jóllakott lesz. Akkor éhezik meg, mikor megkordul a gyomra. Ha részeg, és még iszik, akkor elhányja magát és elájul, amely állapotban sem az éhséget, sem a jóllakottságot nem érzi. Ha ájult, akkor addig marad így, amíg le nem locsolják, de csak hideg vízre reagál. Ájultából kelve mindig józan lesz, de az éhsége vagy jóllakottsága nem változik. Mindezeken kívül le tud menni alfába. Ennek az állapotnak a belsejéről annyit tudunk, hogy összetett, de többet nem. Egy entry (be) és két exit pontja (ki1, ki2) van. Éber állapotból kerülhet ide, ha stroboszkópba néz. Hogy hogyan jön ki belőle, arról csak annyi bizonyos, hogy a ki1 pontból józanul és éhesen jön elő, a ki2-n keresztül pedig elájul. Az életét állítólag józanul és éhesen kezdte.

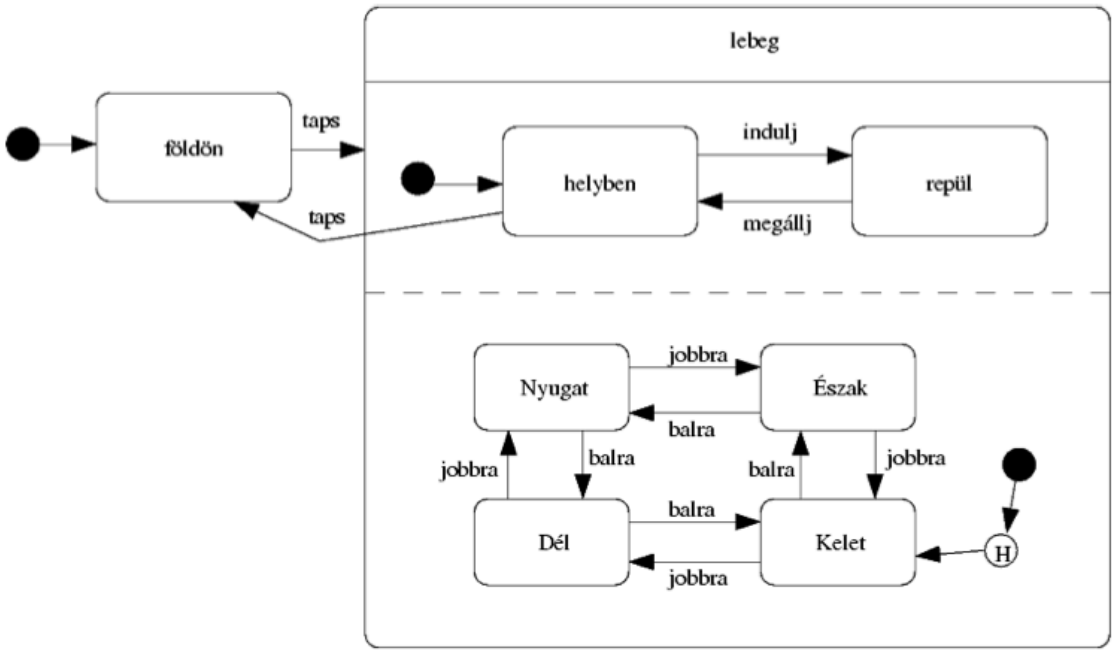


Készítsen UML állapotdiagramot (state chart) az alábbi leírás alapján !
A parser normál állapotban három (A,B,C) műveletet végez. Kezdetben A-ban van. Ha ilyenkor betűt kap, kiírja. Sorvégjelkor átvált B-re. Ekkor csak a számjegyeket írja ki. Sorvégjelkor C-re vált: ilyenkor minden betűt és számot kiír. Újabb sorvégre ismét A-ba kerül. C-ben fájlvégjelre befejezi a működést. Ha bármely állapotában idézőjel jön, onnantól nem ír ki semmit, csak ha újabb idézőjel jön, ekkor az utoljára otthagyott módot folytatja.



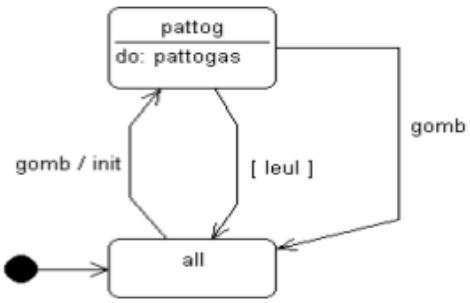
Rajzoljon az alábbi történetnek megfelelően UML state-chartot!

Harún ar-Rasíd repülő szőnyeget kap ajándékba tengerjáró Szindbádtól. A szőnyeg alapesetben a földön pihen. Tapsra felemelkedik, lebegni kezd. Ekkor az 'indulj' parancsra elindul, a 'megállj'-ra megáll, és ismét csak lebeg. Ha egyhelyben lebeg, tapsra a földre ereszkedik. Mikor lebeg vagy repül, az iránya módosítható: a 'jobbra' illetve 'balra' szavakra a megfelelő égtáj fele (pl. 'jobbra' esetén északról keletre) fordul. Mikor a földről felemelkedik, abba az irányba áll, amiben utolsó lebegésekor volt. Szindbád elmondta, hogy a szőnyeg legelőször kelet fele repült.

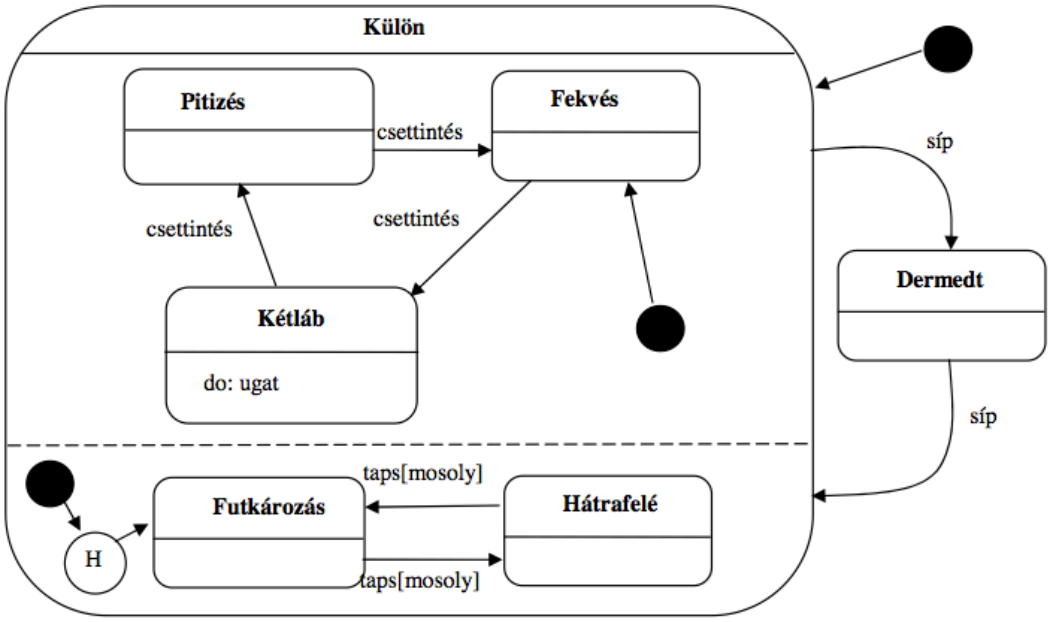


A leejtett labda pattogását utánzó program vezérlésére egy gomb szolgál. A gombot megnyomva a labdát az induló helyzetből elengedjük, mire a labda leesik és energiát vesztve a földről visszapattan. A pattogást addig folytatja, amíg az energiája el nem fogy (leül). Ha a labda leült, a gombot megnyomva a pattogás az induló helyzetből ismét elkezdhető. Ha a gombot pattogás közben nyomjuk meg, a labda mozgása megáll és csak a gomb ismételt megnyomásával indítható el az induló helyzetből.

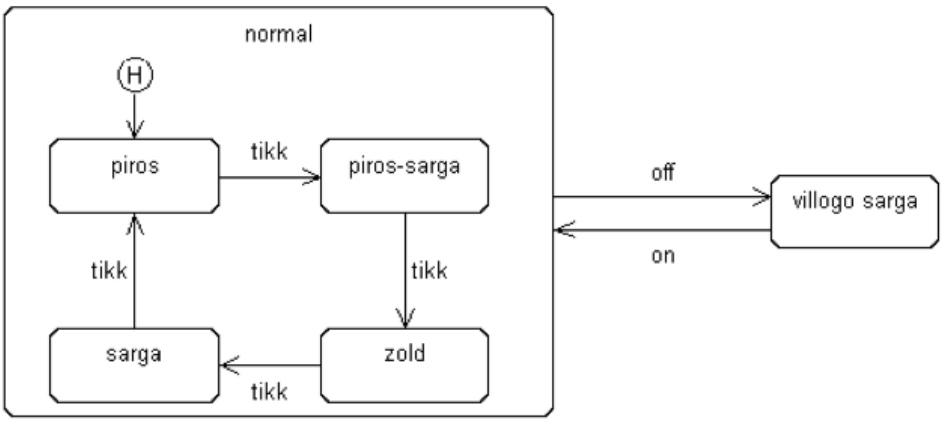
Rajzolja meg a labda viselkedését leíró UML state-chartot !



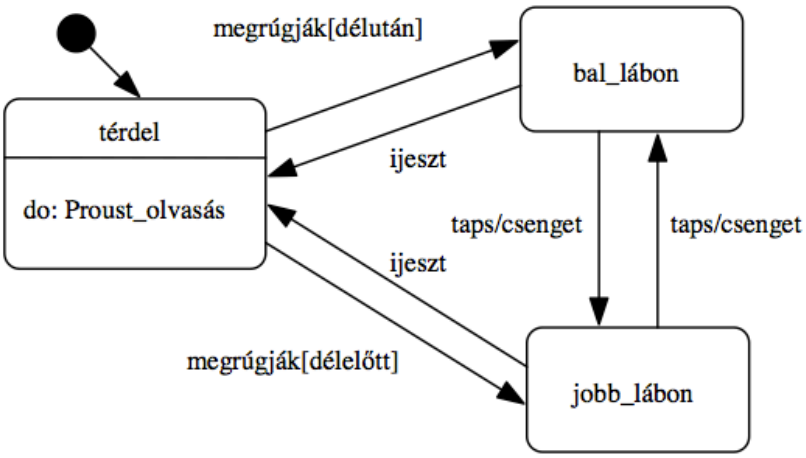
Rajzoljon UML 2 **állapotdiagramot** (state chart) az a következő leírás alapján a kutyapárról!
A Csinnbumm Cirkusz szerződtette Lali Bohócot. A bohócnak van egy idomított kutyapárja (Csahos és Rühes). Mindkét kutya külön-külön mutatványt tud, de néha ugyanazt csinálják. Csahos csettintésre pitizik, újabb csettintésre fekszik, újabb csettintésre két lábon járva ugat, végül újabb csettintésre ismét pitizik. A műsort a fekvéssel kezdi. Rühös tapsra vált ha, közben a bohóc mosolyog: először futkározik (mindig ezzel kezd), aztán hátrafele megy, majd megint futkározik. Amikor a bohóc megfújja a sípját, a két kutya megdermed, és addig így maradnak, amíg újabb sípszó nem érkezik. Rühös az okosabb, ő ott folytatja, ahol abbahagyta, Csahos mindig fekvéssel kezd.



Egy közlekedési lámpa a beépített óra jeleinek hatására a szokásos módon rendre pirosra, piros-sárgára, zöldre, sárgára majd ismét pirosra vált. A lámpa szekrényében elhelyezett kapcsolóval a lámpa átállítható felfüggesztett módra, amikor sárgán villog. Ugyanezen kapcsolóval visszaállítható a normál működés. A lámpa, mikor visszakapcsolják, a felfüggesztés előtti színnel kezdi a működést.
Rajzoljon **UML state-chart**ot a probléma leírására !

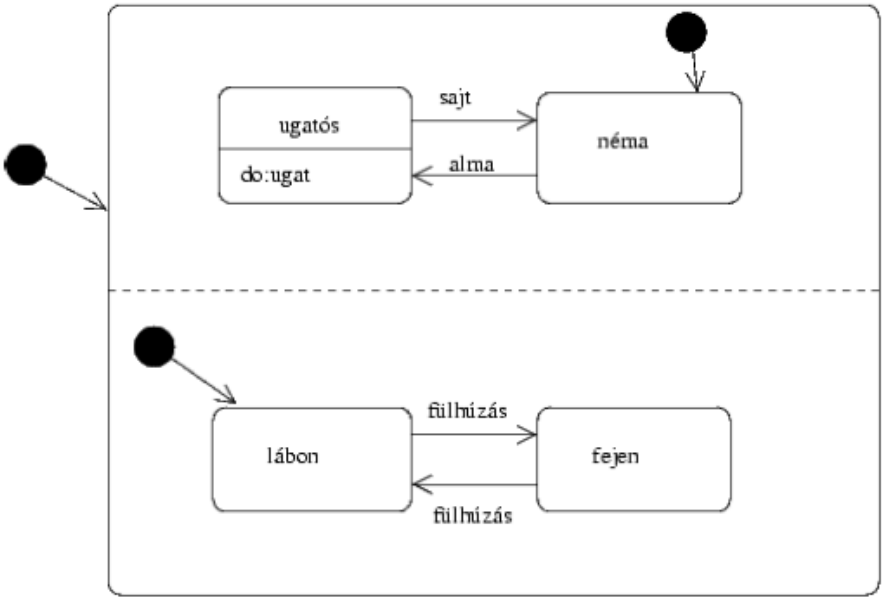


Készítsen UML állapotdiagramot (state chart) az alábbi történet alapján!
Harún ar-Rasíd kap Nagy Károlytól egy Heribert Illig nevű, 300 éves koboldot. A kobold alapból térdel. Ha ekkor megrúgják, attól függően, hogy délelőtt vagy délután van, a jobb illetve a bal lábára áll. Ha bármelyik lábán áll, és tapsot hall, akkor csenget egyet, és átáll a másik lábára.
Ha bármelyik lábán áll, és egy nullával ráijesztenek, térdre kényszerül. Mikor térdel, folyamatosan Proustot olvas.



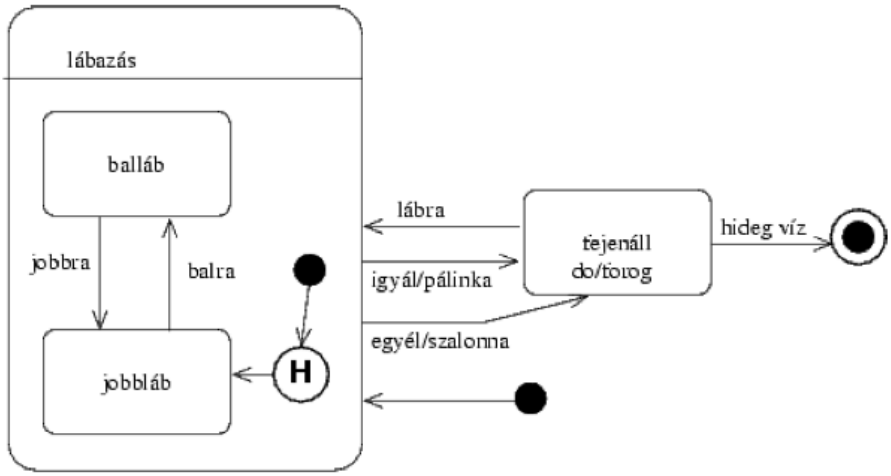
Készítsen az alábbi leírás alapján UML állapotábrát (statechart)!

II. Frigyes udvari koboldja alapesetben néma. Ha almát kap, elkezd ugatni, ha ekkor sajtot kap, ismét megnémul. Ha bármikor meghúzzák a fülét, akkor fejre áll, újabb fülhúzásra ismét lábra áll. Mikor megszületett, lábon állt.

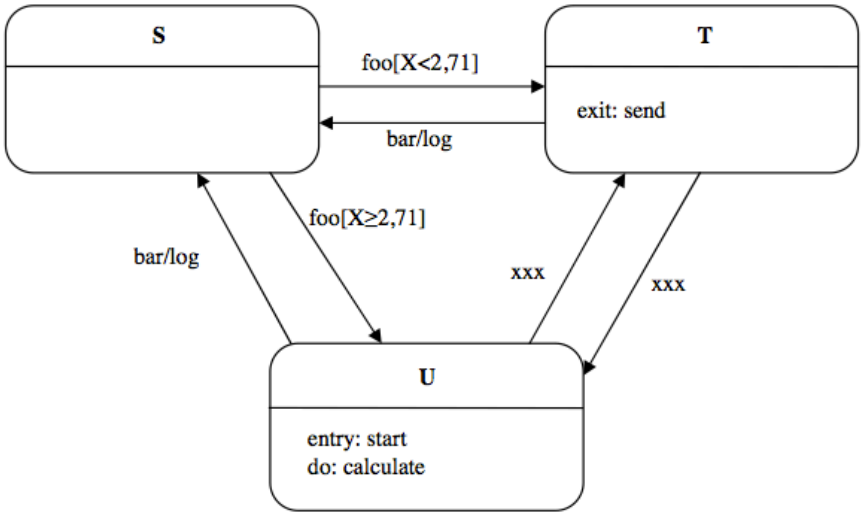


Rajzoljon az alábbi történetben szereplő tánchoz UML state-chartot!

Aszályfürdőben évente megrendezik a forrástisztító ünnepélyt. Ekkor a falu legöregebb lakója a forráscsurgató táncot járja. A tánc a lábazással kezdődik: a táncos a jobb lábán áll, majd a javasasszonyok 'balra' kiáltására balra vált, a 'jobbra' kiáltásra pedig jobbra. Ez így megy egészen addig, amíg vagy a javasurak 'igyál' kiáltására megiszik egy kupica hagymapálinkát, vagy a javasgyerekek 'egyél' kiáltására megeszik egy falat szalonnabőrt, majd mindkét esetben fejre áll, és így forogni kezd. Ekkor a javasasszonyok 'lábra' kiáltására ismét féllábra áll, pont arra, amelyiken a fejreállás előtt állt. A tánc akkor véget, amikor a táncos fejen áll, és a felesége hideg vízzel leönti.



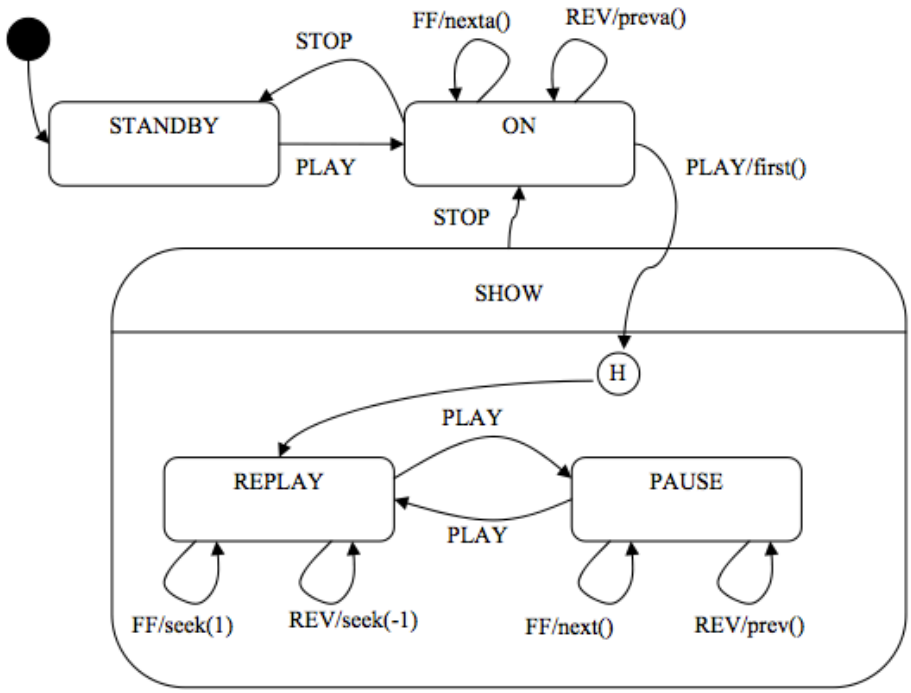
Egészítse ki az alábbi UML 2 állapotdiagramot (state chart) a következő leírás alapján!
Egy objektum három fő állapottal (**S**, **T**, **U**) rendelkezik. A kezdőállapot az **S**. Ha **S**-ben **foo** esemény éri, akkor attól függően, hogy **X** értéke kisebb, mint 2,71 vagy sem, rendre a **T** vagy az **U** állapotba kerül. Mindkét állapot a **bar** és az **xxx** események hatására hagyható el. Előbbi esemény esetén visszatér **S**-be, utóbbinál pedig **T**-ből **U**-ba, **U**-ból **T**-be kerül. **U**-ba lépéskor mindig lefut a **start** metódus, míg **T**-t elhagyva a **send**. A **bar** eseményre történő állapotváltás során a **log** metódus hívódik meg. Az **U** állapotban tartózkodás közben a **calculate** metódus fut.



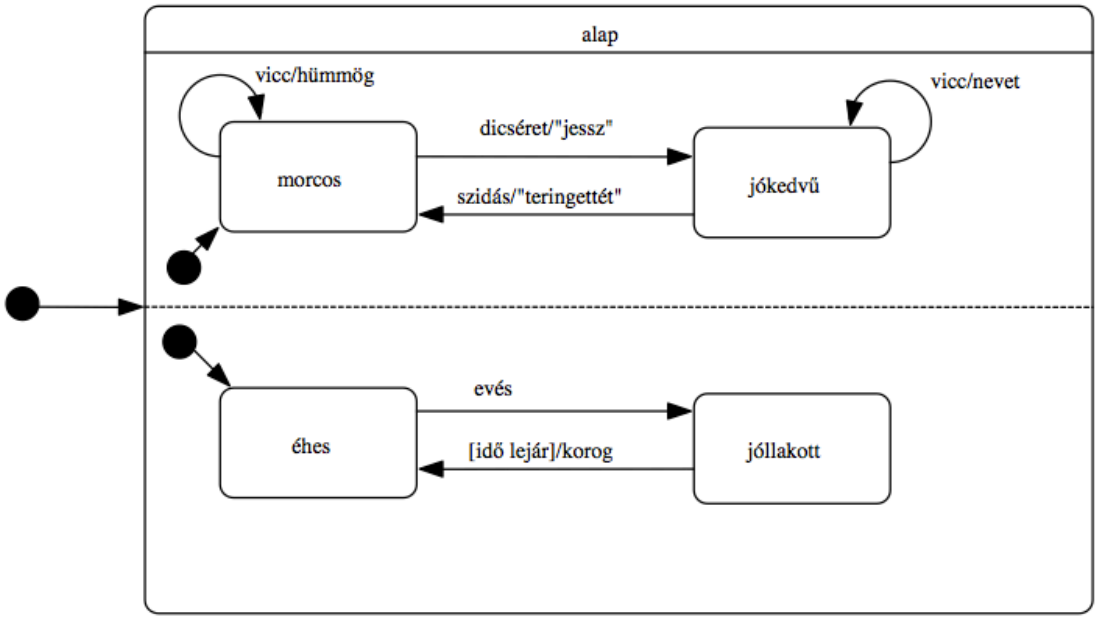
Rajzoljon UML2 state-chartot (állapot-diagram) az alábbi történet alapján !

A Dárembéz MP3 lejátszón 4 gomb van: PLAY, STOP, FF, REV. Amikor elemet teszünk bele, akkor STANDBY állapotba kerül. PLAY hatására kapcsol be (ON). Ekkor az FF és a REV gombokkal lehet előre- és hátralépni az albumok között. PLAY megnyomására lejátszó (SHOW) módba kerül, amikor vagy az aktuális album első számát kezdi lejátszani (REPLAY), vagy szünetelteti a lejátszást (PAUSE). Hogy melyiket csinálja, az attól függ, hogy utoljára melyiket végezte SHOW módban (ha még egyiket sem, akkor REPLAY az alap). Ha REPLAY alatt nyomkodjuk az FF és a REV gombokat, a számon belül tekerünk előre vagy hátra 1 mp-nyit. Ha PAUSE módban nyomkodjuk őket, akkor a számok között ugrálhatunk. STOP hatására ismét ON-ba kerülünk, újabb STOP-ra STANDBY-ba. SHOW állapotban a PLAY gombbal lehet megállítani (PAUSE) és újraindítani (REPLAY) az aktuálisan játszott számot.

A lejátszó mp3-API-ja a következő függvényeket ismeri: *seek(int x)*: x mp-et előre megy; *next(), prev()*: következő, előző számot választja; *nexta(), preva()*: következő/előző album; *first()*: album első számára áll.

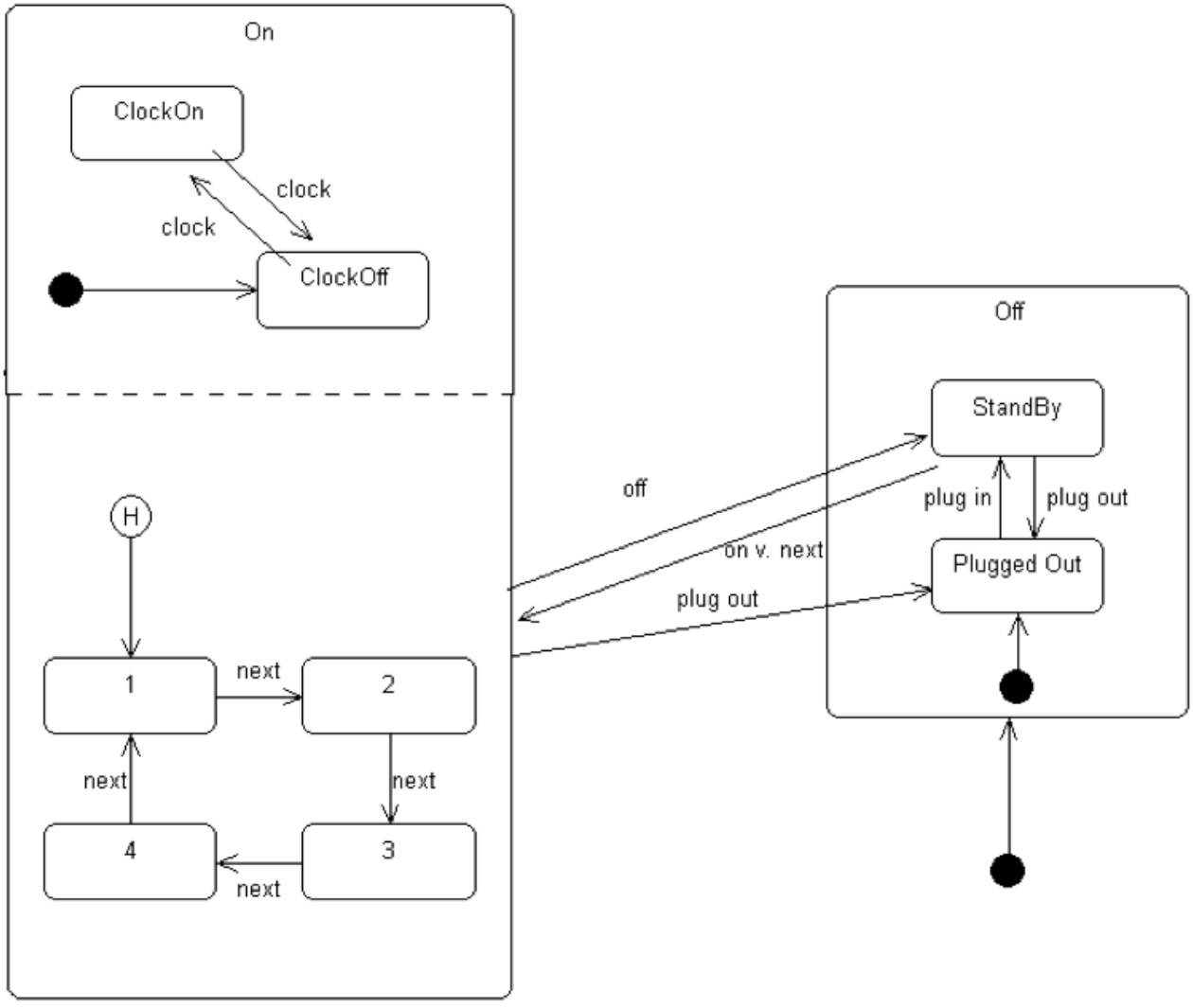


Készítsen UML állapotdiagramot (state chart) az alábbi leírás alapján !
Izidor alapból morcos kisfiú. Ha ekkor viccet hall, csak hümmög. Ha megdicsérik, azt mondja, „jessz”, és jókedvű lesz. Ekkor a vicceken nevet, de. ha megdorgálják, azt mondja, hogy “teringettét”, és ismét morcos lesz. Mindeközben többnyire éhes (már éhesen született). Ha kap enni, akkor jóllakik. Rövid idő elteltével azonban újra éhes lesz, és ekkor kordul egyet a gyomra.



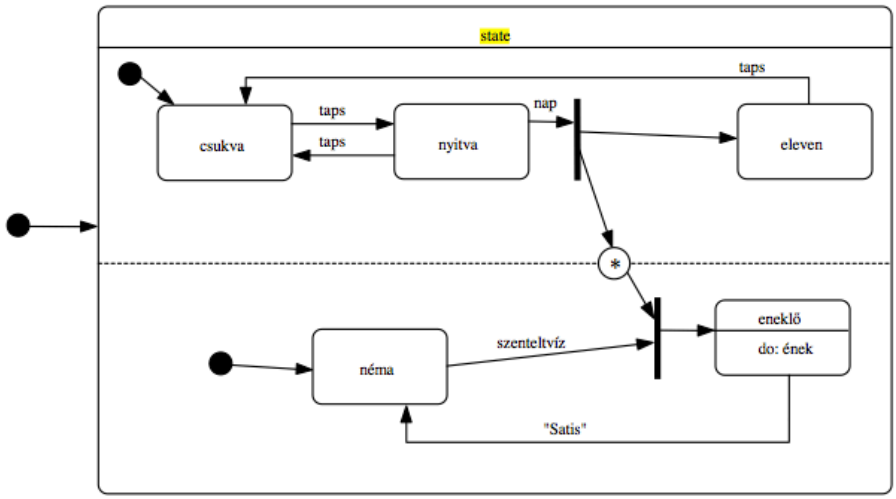
Rajzolja meg a HályTek® televíziókészülék viselkedését leíró UML state-chartot !
A kicsomagolt televíziókészülék csatlakozóját a dugaszoló aljzatba kell helyezni, aminek hatására a készülék feszültség alá kerül, és készenléti állapotban vár. A készüléket a távirányítón található ON vagy NEXT gombokkal lehet bekapcsolni. A kikapcsoláshoz a távirányító OFF gombját kell megnyomni, ennek hatására a készülék ismét készenléti állapotba kerül. Ismételt bekapcsoláskor a legutoljára beállított csatorna lesz látható. A csatornák között a NEXT gombbal tudunk váltani, a gomb megnyomásakor mindig a soron következő csatorna lesz látható. A készüléknek 4 különböző csatornája van, a negyedik után a NEXT gomb az első csatornára kapcsol. A készülék tápkábelét bármikor ki lehet húzni, de ekkor a készülék nem fog működni ! A feszültség alá helyezés utáni első bekapcsoláskor az első csatorna lesz kiválasztva.

A készülék képes az aktuális idő megjelenítésére. Az óra azonban bekapcsoláskor nem látszik. Ha a képernyőn látni kívánjuk az órát, akkor a CLOCK gombot kell megnyomnunk. Az órát a CLOCK gomb ismételt megnyomásával tudjuk eltüntetni.



3. feladat típus (szopatós rajz):

Készítsen UML állapotdiagramot (state chart) az alábbi leírás alapján !
II Frigyes, német-római császár érdekes kódexet kapott a monte cassinoi apátságból. A kódex kezdetben csukva volt. Tapsra kinyílt, újabb tapsra becsukódott. Ha nyitott állapotában ráesett a napsugár, megelevenedtek a képei. Tapsra azonban ilyenkor is becsukódott, és a képek is visszamerevedtek. Mindezek közben a könyv néma volt. Ha ugyanezen napsugár ráesése után valamikor szenteltvízzel meghintették, beneventán antifónákat énekelt, függetlenül attól, hogy éppen nyitva vagy csukva volt. Egészen addig énekelt, amíg azt nem mondták neki, hogy „Satis!", mikor is ismét néma lett.



4. feladat típus (hiba keresés):

Milyen hibá(ka)t talál az alábbi **UML statechart**-on ?

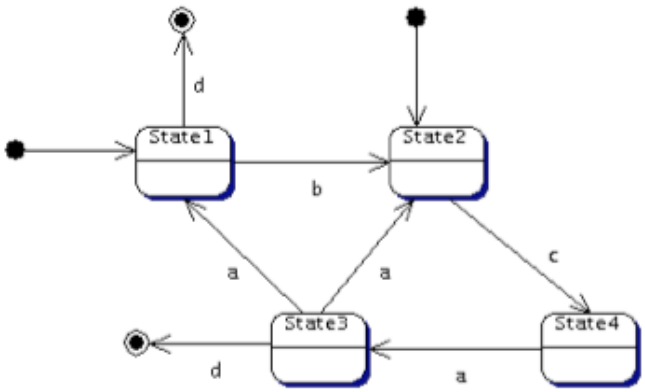
Két kezdő állapot .....

State3-ban a hatása nem egyértelmű.....

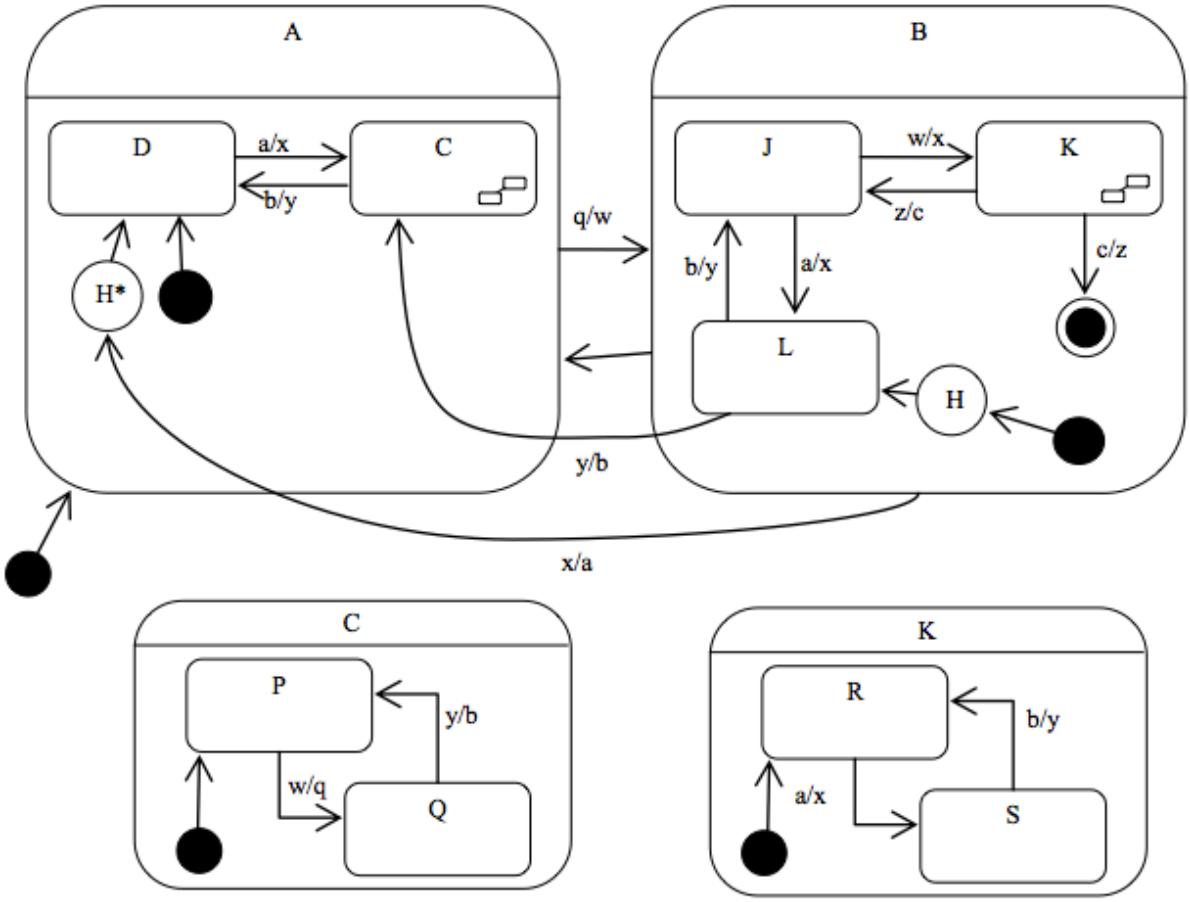
.....

.....

.....



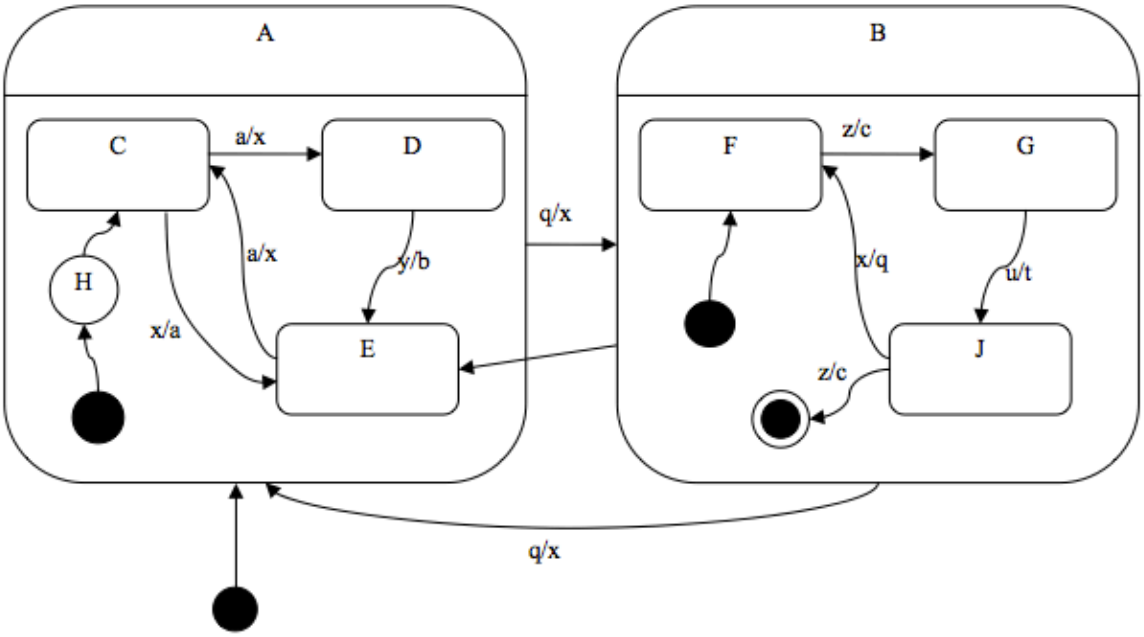
5. feladat típus (állítások):



Igaz	Hamis	Állítás
☑	☐	L állapot után közvetlenül következhet Q állapot
☐	☑	C állapotból elérhető egy lépésben S
☑	☐	K állapotból csak „c” és „x” esemény hatására léphetünk át A állapotba
☐	☑	Q állapotból „y” esemény hatására átlépünk D-be
☐	☑	L állapot után csak C és J következhet egy lépésben

A kezdés után az **a, w, q, b, x** esemény-szekvencia hatására

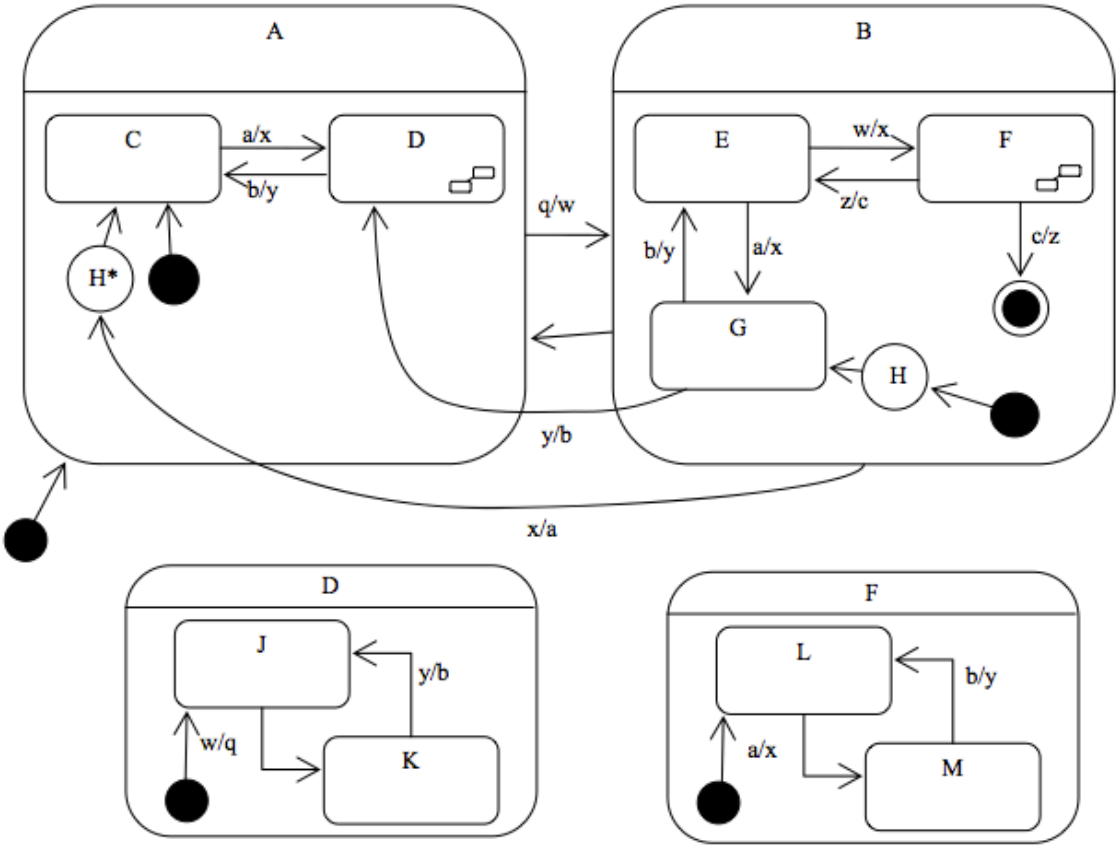
Igaz	Hamis	Állítás
☐	☑	pontosan kétszer fut le a „q” tevékenység
☑	☐	Q állapotba kerülünk
☐	☑	érintettük az K állapotot



Igaz	Hamis	Állítás
☑	☐	D állapotból 2 lépésben visszaérhet D-be
☐	☑	F állapotból „q” esemény hatására H állapotba kerül
☑	☐	B-ből A-ba való váltáskor végrehajtódhat a „c” tevékenység
☐	☑	E állapotból egyetlen esemény hatására csak a C állapot következhet
☑	☐	J állapotból egyetlen esemény hatására E állapot következhet

A kezdés után az **x, q, z, q** esemény-szekvencia hatására

Igaz	Hamis	Állítás
☐	☑	C állapotba kerülünk
☑	☐	Kétszer lefut az „x” tevékenység
☐	☑	Érintjük a J állapotot

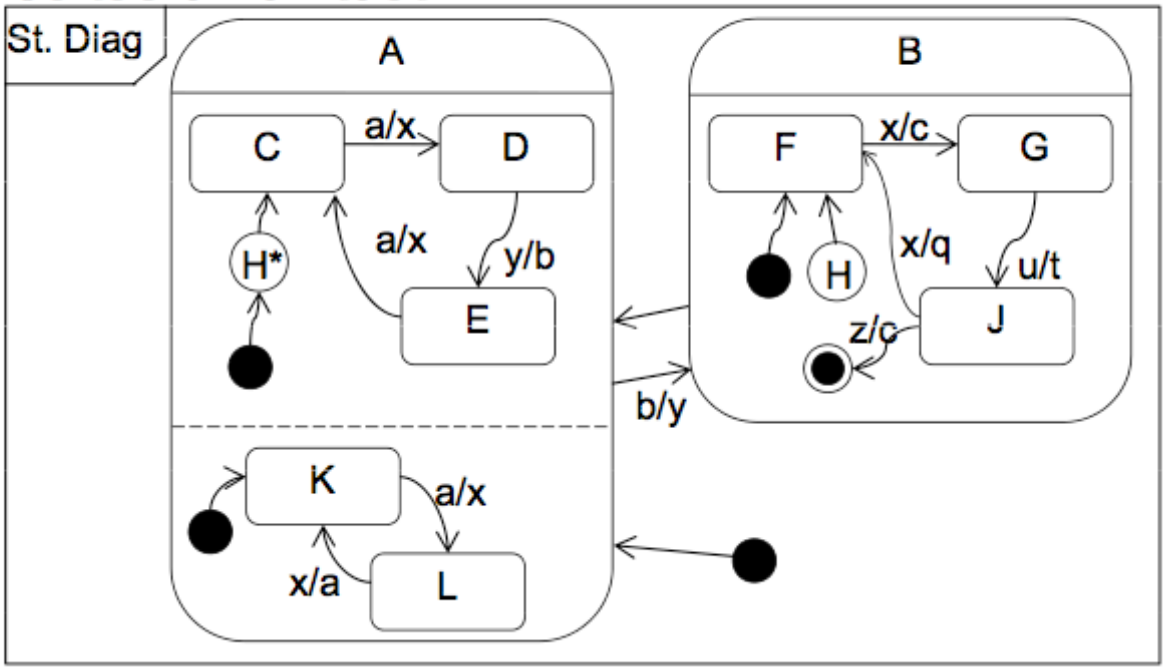


Igaz	Hamis	Állítás
☐	☑	G állapot után csak D és E következhet egy lépésben
☐	☑	D állapotból elérhető egy lépésben M
☑	☐	G állapot után közvetlenül következhet K állapot
☐	☑	K állapotból „y” esemény hatására átlépünk C-be
☑	☐	F állapotból csak „c” és „x” esemény hatására léphetünk át A állapotba

A kezdés után az **a, w, q, b, x** esemény-szekvencia hatására

Igaz	Hamis	Állítás
☑	☐	K állapotba kerülünk
☐	☑	érintettük az F állapotot
☐	☑	pontosan kétszer fut le a „q” tevékenység

A következő UML2 állapotdiagram alapján minősítse az állítást!



[**I**] D állapotból 2 lépésben visszaérhet D-be

[**I**] Az E állapotból egy lépésben elérhető állapotok: C, F, G, J.

[**I**] J-ből egy lépésben nem juthatunk el L-be.

[**H**] K-ból L-be csak akkor léphetünk, ha közvetlenül előtte C-ből D-be léptünk.

[**H**] Az F állapottal egyidőben lehetünk L-ben is.

A kezdés után az **a, b, x, u, z, b** szekvencia hatására

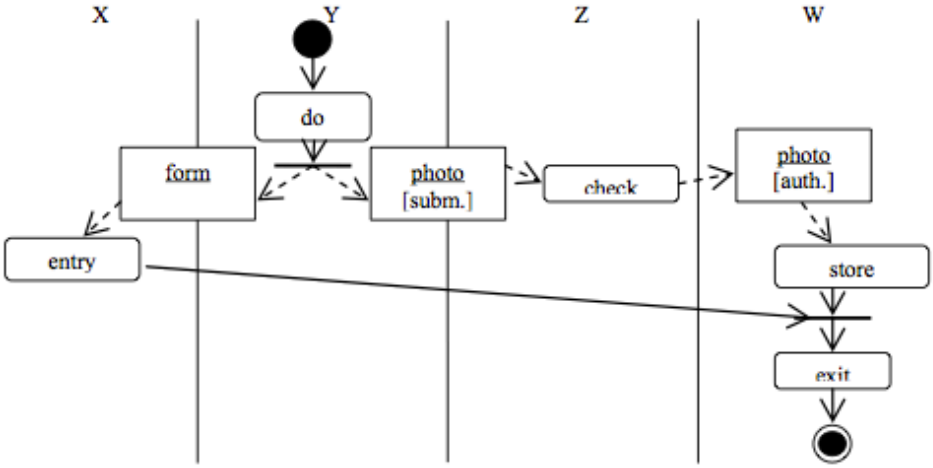
[**I**] pontosan kétszer fut le az 'x' tevékenység.

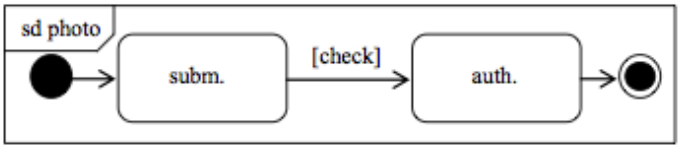
[**H**] pontosan kétszer érintettük az L állapotot.

[**I**] végül J állapotba kerülünk.

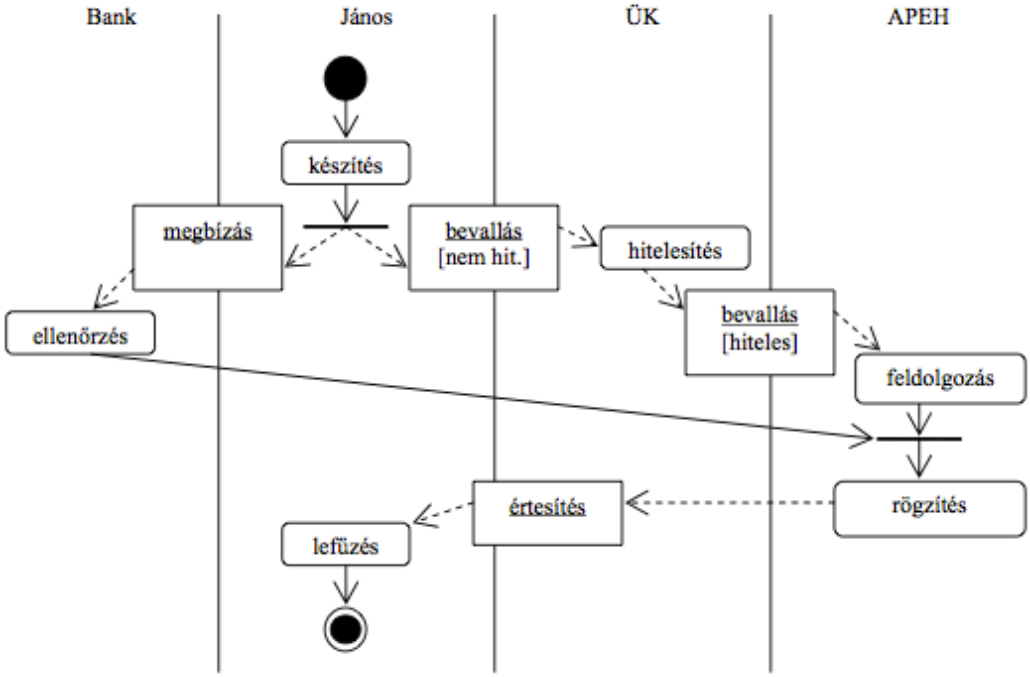
Feladatok Activity Diagram témakörböl

Legyen adott az alábbi – object flow-val kiegészített – aktivitás-diagram (activity diagram) ! Rajzolja meg azon objektumok UML2 state-chartját, amelyeknek az ábra alapján több állapota is van!





Készítsen UML 2 aktivitás-diagramot (activity diagram) az alábbi leírás alapján! Jelölje az action-object flow-t is! Használja a kövéren szedett kifejezéseket!
Regenkurt **János** vállalkozó adóbevallást **készít**. A **bevallást** feltölti az ügyfélkapura (**ÜK**), és ezzel párhuzamosan a **bank**jánál átutalási **megbízást** ad az adóhátralék befizetésére. Az ügyfélkapu **hitelesíti** az adóbevallást, és továbbküldi az **APEH**-nek, ahol **feldogozzák**. A bank **ellenőrzi** az átutalási megbízást, majd a pénzt átutalja az APEH-nek. Amikor az APEH feldolgozta a bevallást és megkapta az átutalást, akkor **rögzíti** az állapotot, és **értesítést** küld Jánosnak, hogy minden rendben. Ezt az értesítést János **lefűzi**. Ezzel az adóbevallás véget ér.



Feladatok Verifikáció és Validáció témakörből

1. feladat típus (bug/error/failure/fault):

Felsoroltunk szoftverrel kapcsolatos "hibákat". A hibák mellett jelölje be, hogy az melyik kategóriába tartozik!

	bug	failure	error	fault
Hiányzik a "synchronized" kulcsszó	☒	☐	☐	☒
A ciklusfeltétel hibás	☒	☐	☐	☒
Elmaradt a kritikus kódok felülvizsgálata	☐	☐	☒	☐
Alábecsültük a példányosítás erőforrás-igényét	☐	☐	☒	☐
Sok objektumnál nagyon lassan kapjuk az eredményt	☐	☒	☐	☐
Az első futásnál hibás eredményt kapunk	☐	☒	☐	☐

Felsoroltunk szofverrel kapcsolatos "hibákat". A hibák mellett jelölje be, hogy az melyik kategóriába tartozik!

	bug	error	failure	fault
Az "==" helyett "=" áll a kifejezésben	☒	☐	☐	☒
A weboldal nem jön le	☐	☐	☒	☐
A szerver memóriaigényét alulbecsülték	☐	☒	☐	☐
A program lefagy, ha hibás bemenetet adunk meg	☐	☐	☒	☐
Elmaradt a javított metódus tesztelése	☐	☒	☐	☐
A function két paramétere fel van cserélve a kódban	☒	☐	☐	☒

Felsoroltunk szofverrel kapcsolatos "hibákat". A hibák mellett jelölje be, hogy az melyik kategóriába tartozik!

	bug	fault	failure	error
A relációjel fordított	☒	☒	☐	☐
Hibás bemenetekre nem teszteltünk	☐	☐	☐	☒
Figyelmen kívül hagytuk az objektum konstruálás időigényét	☐	☐	☐	☒
A pointer értéke null	☒	☒	☐	☐
A program lefut, de rossz eredményt kapunk	☐	☐	☒	☐
A reggeli órákban nem sikerül a szolgáltatóhoz kapcsolódni	☐	☐	☒	☐

Definiálja a szoftver hibával kapcsolatos alábbi fogalmakat !

Bug Kódbeli hiba, ami el van rontva, valami ki van hagyva a kódból. A bug failure-t okoz.

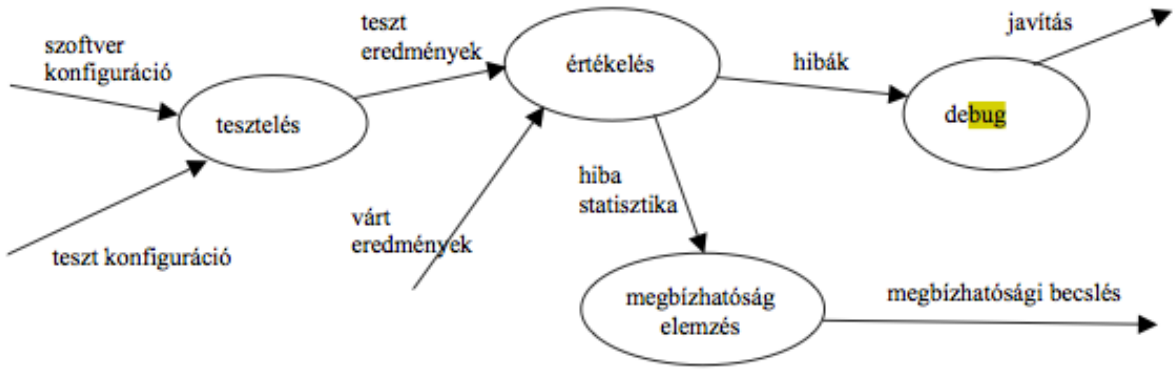
Error Emberi tevékenység (tévedés, elhanyagolás) hibája, ezért keletkezik a program kódjában valami hiba (bug).

Failure Kívülről látható hiba.

Fault = bug

2. feladat típus (elmélet rajz):

Rajzolja fel a tesztelés információs folyamatának adatfolyamábráját !



3. feladat típus (elmélet):

A "V model" szoftver életciklus modell alapján milyen tesztelési szinteket azonosíthatunk

unit test..... integration test.....

system test..... acceptance test.....

Adja meg a fejlesztési folyamatban előforduló tesztelési fokozatokat (stages)

unit (egység) test, integration (integrációs) test, system (rendszer) test, acceptance (átadás, elfogadás) test

Milyen integrációs (vagy tesztelési) stratégia esetében használunk teszt ágyakat (test bed) ? Mi a funkciója a teszt ágynak ?

- bottom-up
- környezet, amelyben az integrált unitok együttes működése tesztelhető.

A bottom-up tesztelési (integrációs) stratégia esetében milyen kiegészítő eszközre van szükség ?

- tesztágy (test bed)

Mi a funkciója az eszköznek ?

- a tesztelendő alrendszerek kezeléséhez keretet ad

Melyek a legfontosabb integrációs stratégiák és azok milyen következménnyel járnak a tesztelésre nézve ?

top-down: fönttöl raknk össze egy programot, teszt betéteket (test stub) alkalmazunk ott, amit még nem implementáltunk
bottom-up: alulról készülünk el az elemekkel, alacsony szintü metódusokat rakunk össze, majd építkezünk, teszt ágyat (test bed) kell készíteni és ebbe tudjuk a kész elemeket beletenni

A tesztelés információs folyamatában álló „értékelés” (evaluation) processznek mi(k) a be- és kimenete(i) ?

Bemenet	**Kimenet**
teszt eredmények (test results)	hibák (errors) .....
várt eredmények (expected results)	hiba statisztika (error statistics).....

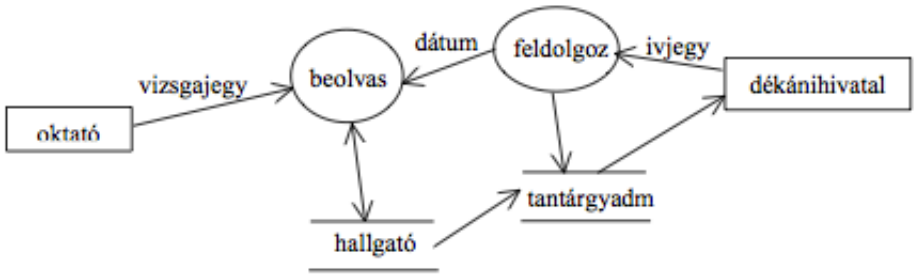
Milyen minőségi jellemzőket tesztelünk a különböző FURPS típusú tesztekkel ?

Functionality, funkcionalitás	Usability, használhatóság.....
Reliability, megbízhatóság	Performance, teljesítőképesség .....
Supportability, támogatottság.....	.....

Feladatok Funkcionális Specifikáció témakörből

1. feladat típus (hiba keresés):

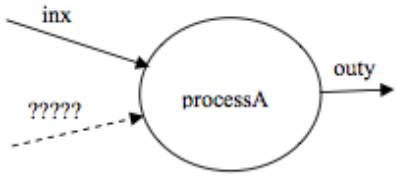
Milyen szintaktikai hibákat talál az alábbi adatfolyam ábrán ?

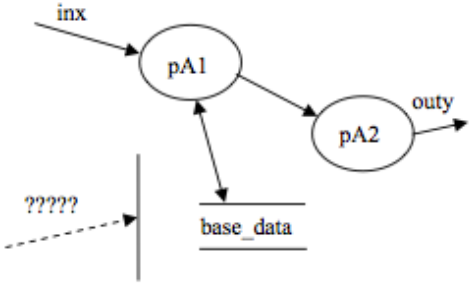


- *ha ez DFD, akkor nincs terminátor, ha ez ContextDiag, akkor nincs 2 processz és adattár*
- ***beolvas**ból nincs kimenő adat, ezért a **hallgató**ból **beolvas**ba mutató nyílhegy hiba.*
- ***tantárgyadm** (adattár) nem köthető a **dékánihivatal**hoz (terminátor)*
- ***hallgató** és **tantárgyadm** (két adattár) közvetlenül nem köthető össze.*

2. feladat típus (vezérlö folyam):

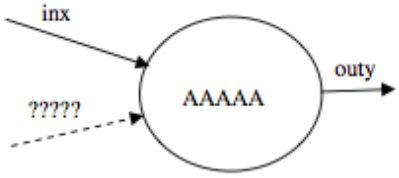
A baloldalon álló adatfolyamábra részleten mi a `????`-el jelölt ábraelem ?
Tételezzük fel, hogy a `processA` folyamatot DFD-ként akarjuk megjeleníteni a jobboldali ábrarészlet szerint.
Kapcsolja a kérdezett (`????`) ábraelemet korrekt módon az ábrához !

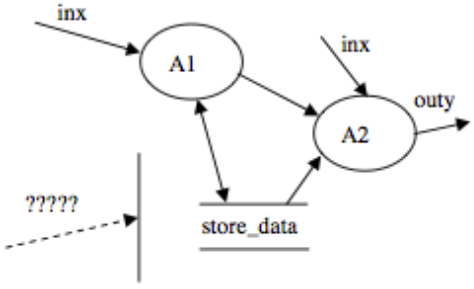




Vezérlő folyam, vezérlés .....

Tételezzük fel, hogy az **`AAAAA`** folyamatot DFD-ként akarjuk megjeleníteni a jobboldali ábrarészlet szerint. Mi a **`????`**-el jelölt ábraelem ? Kapcsolja a kérdezett ábraelemet korrekt módon a jobboldali ábrához ! Van-e szintaktikai hiba a jobboldali ábrán ?

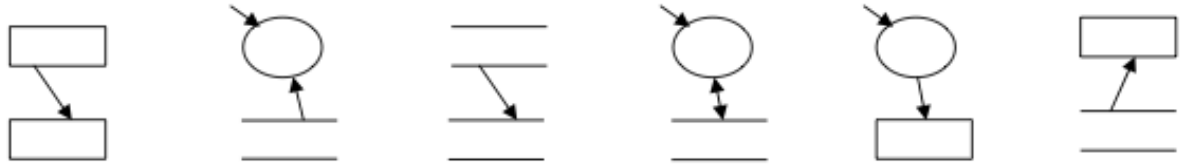




Vezérlő folyam, vezérlés, NINCS.....

3. feladat típus (melyik ábra helyes):

Adja meg, hogy helyesek-e a következő adatfolyamábra illetve context diagram részletek !



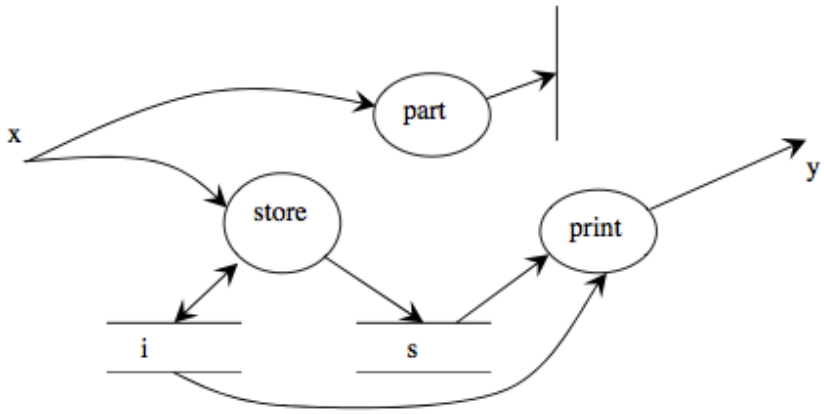
☐ igen ☒ nem	☐ igen ☒ nem	☐ igen ☒ nem	☐ igen ☒ nem	☒ igen ☐ nem	☐ igen ☒ nem

4. feladat típus (állapottáblából rajz):

Egy program (amelynek bemenete `x`, kimenete `y`) működését az alábbi állapottábla írja le:

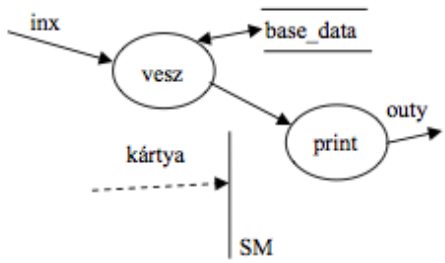
	e1	e2	e3
A1	A1/-	A1/-	A2/s[i++]=x
A2	A3/-	A3/-	A2/s[i++]=x
A3	A3/s[i++]=x	A3/-	A1/y=print(s,i)

ahol `e1`, `e2`, `e3` események, amelyek a `part(x)` függvény lehetséges értékei, `s` egy 1000 elemű `x` típusú értékek tárolására szolgáló tömb, `i` integer index.
Rajzolja fel a program **adatfolyamábráját** !



5. feladat típus (rajzból állapottábla):

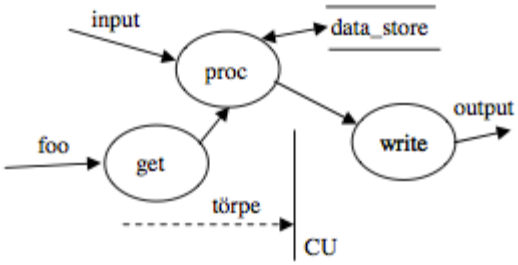
Definiálja az adatfolyam-ábrán szereplő SM vezérlőegység egy lehetséges állapottábláját úgy, hogy az konzisztens legyen (ne legyen ellentmondásban) az adatfolyam és adatspecifikációval



	treff	káró	makk
a	b/vesz	a/	a/
b	b/print	a/	b/

kártya = [treff | káró | makk]

Definiálja az adatfolyam-ábrán szereplő CU vezérlőegység egy lehetséges állapottábláját úgy, hogy az konzisztens legyen (ne legyen ellentmondásban) az adatfolyam és adatspecifikációval !



	tudor	vidor	kuka	
x	x/get	y/proc	y/	
y	y/write	x/	y/	

törpe = [tudor | vidor | kuka]

6. feladat típus (jellemzök felsorolása):

Egy adatfolyam-ábrán szereplő vezérlőegységet az alábbi állapotgéppel írunk le:

	X1	X2	X3
S1	S1/alfa	S2/	S1/beta
S2	S2/	S2/xxx	S1/alfa

Feltéve, hogy a DFD-n csak az adott állapotmodellel kapcsolatos elemek állnak

Specifikálja a vezérlő egység bemenetét ! : [X1|X2|X3] .....

Sorolja fel a processzeket ! : alfa, beta, xxx .....

Adja meg az adattárakat ! : nem tudjuk .....

Egy adatfolyam-ábrán szereplő vezérlőegységet az alábbi állapotgéppel írunk le:

	beta	X	c3
A1	A1/alfa	S2/	A1/ubul
S2	S2/	S2/x2	A1/alfa

Feltéve, hogy a DFD-n csak az adott állapotmodellel kapcsolatos elemek állnak

Specifikálja a vezérlő egység bemenetét ! : [beta|X|c3] .....

Sorolja fel a processzeket ! : alfa, x2, ubul .....

Adja meg az adattárakat ! : nem tudjuk .....

Egy adatfolyam-ábrán szereplő vezérlőegységet az alábbi állapotgéppel írunk le:

	F5	xxx	LO
X1	X1/store	X2/	X1/yyy
X2	X2/	X2/tar2	X1/tar2

Feltéve, hogy a DFD-n csak az adott állapotmodellel kapcsolatos elemek állnak

Specifikálja a vezérlő egység bemenetét ! : [F5|xxx|LO] .....

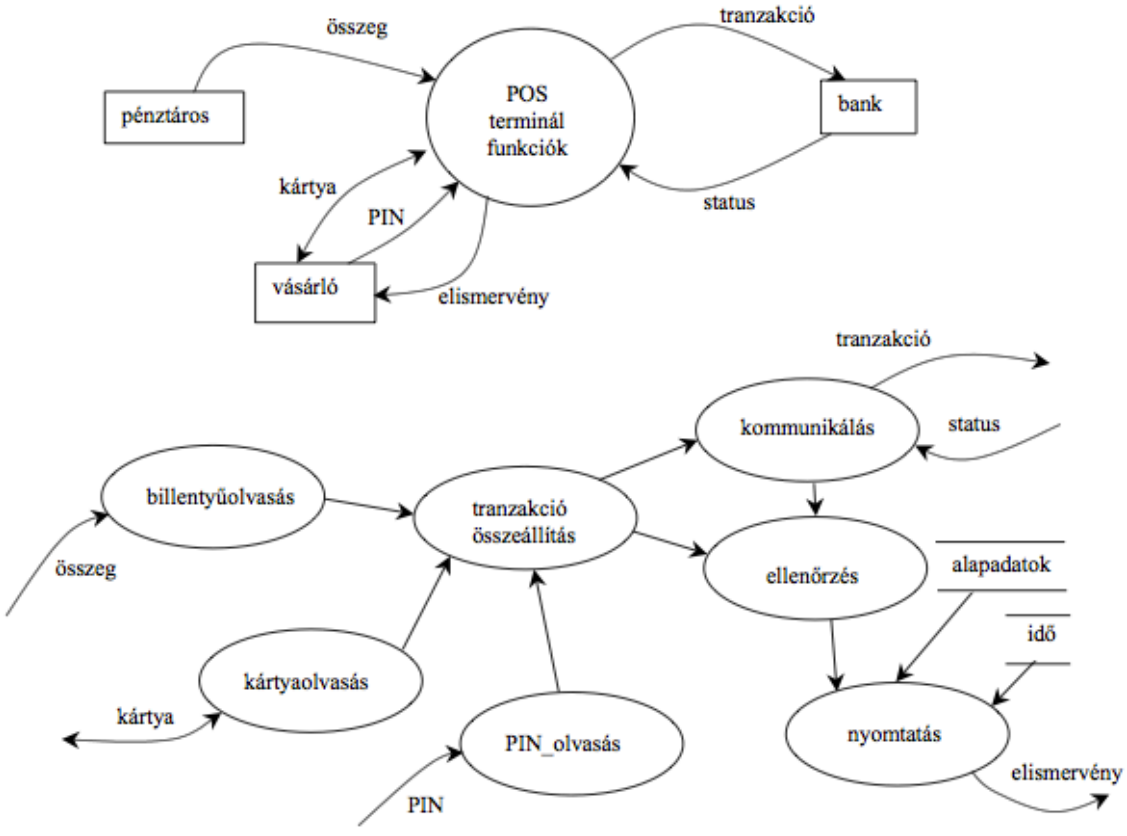
Sorolja fel a processzeket ! : store, tar2, yyy .....

Adja meg az adattárakat ! : nem tudjuk .....

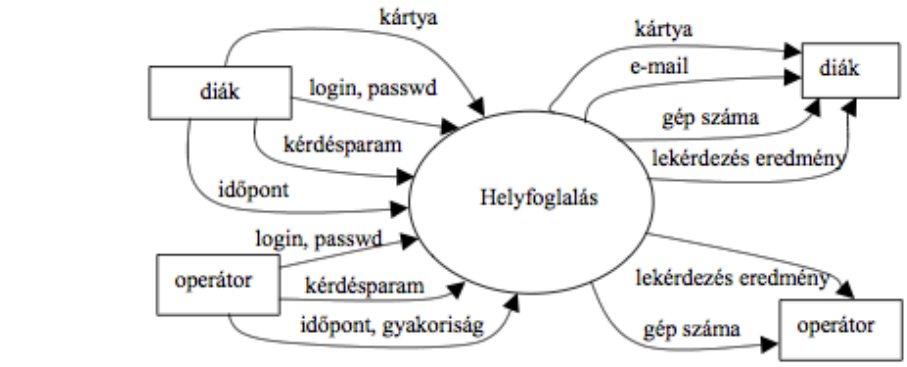
7. feladat típus (DFD, CD rajz):

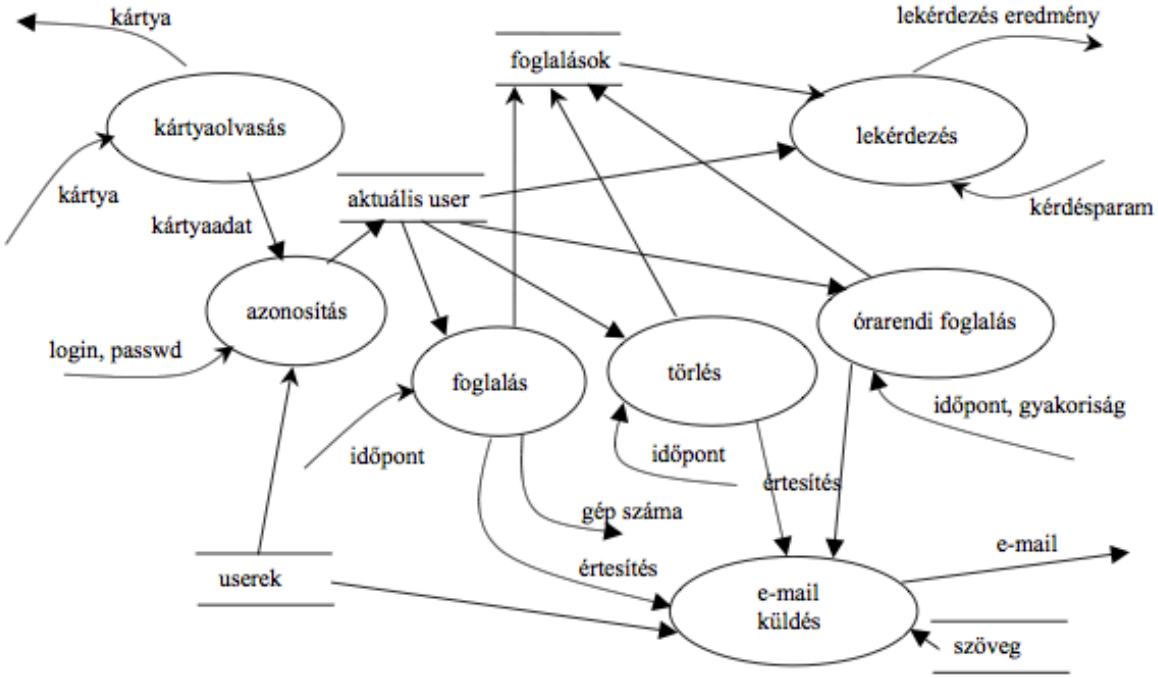
Készítse el egy POS terminál adatfolyamábráját ! Adja meg a context diagramot és annak 0 szintű felbontását !

A POS (Point of Sale) terminált a bankkártya elfogadó helyeken használják. A pénztáros a billentyűzeten keresztül megadja a vásárlás összegét (a kártya terhelését), majd a kártyát az olvasón végighúzva leolvassa a kártya információkat. Bizonyos esetekben a vásárlónak a külön billentyűzeten meg kell adni a PIN kódját. A terminál telefonvonalon keresztül rákapcsolódik a bank számítógépére, ahonnan a tranzakciót elfogadják vagy elutasítják. Végezetül a terminál elismervényt nyomtat, amin szerepel a vásárlás helye, ideje, a kártya száma, a vásárlás összege, a terminál azonosítója, a banki jóváhagyás kódja.



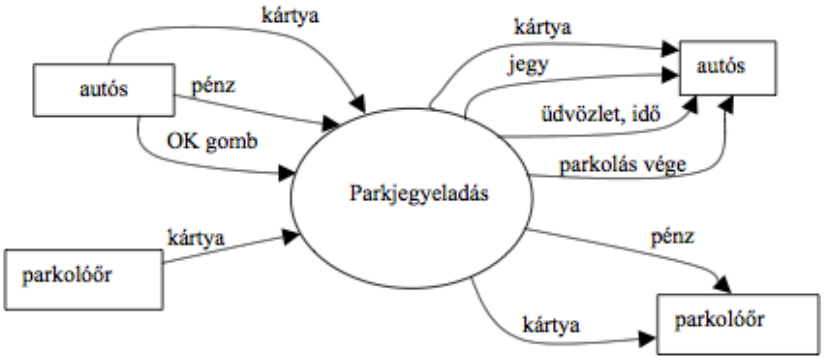
Készítse el egy számítógépes labor helyfoglaló rendszerének adatfolyamábráját ! Adja meg a context diagramot és annak 0 szintű felbontását !
A helyfoglaló rendszer segítségével a hallgatók gépidőt (kvóta = 3 óra/hét) foglalhatnak maguknak. A rendszer használatához szükséges a hallgató kártyája, amit az olvasó leolvas. A bejelentkezőnek az azonosításhoz még meg kell adni a login nevét és a passwordjét. A hallgató különböző lekérdezéseket (az általa foglalt idők, szabad gépidő) kezdeményezhet. Ha van még kvótája, annak terhére foglalhat. A sikeres foglalás eredménye a lefoglalt gép száma. Lehetséges a foglalás törlése is. A rendszert kezelő operátoroknak nem kell kártya, őket a login név és a password azonosítja. Ők bárki nevében tudnak foglalni (akár a kvótán felül is) és törölni. Az operátorok feladata az egész labor lefoglalása a tanórák idejére. Ehhez meg kell adni az óra időpontját és a gyakoriságát (pl. hetente, 6 héten keresztül). Ha az operátori foglalások vagy törlések következtében már bejegyzett hallgatói foglaltság elvész, akkor a rendszer e-mailt küld az érintett hallgatónak. Az operátorok is tudnak lekérdezéseket indítani.

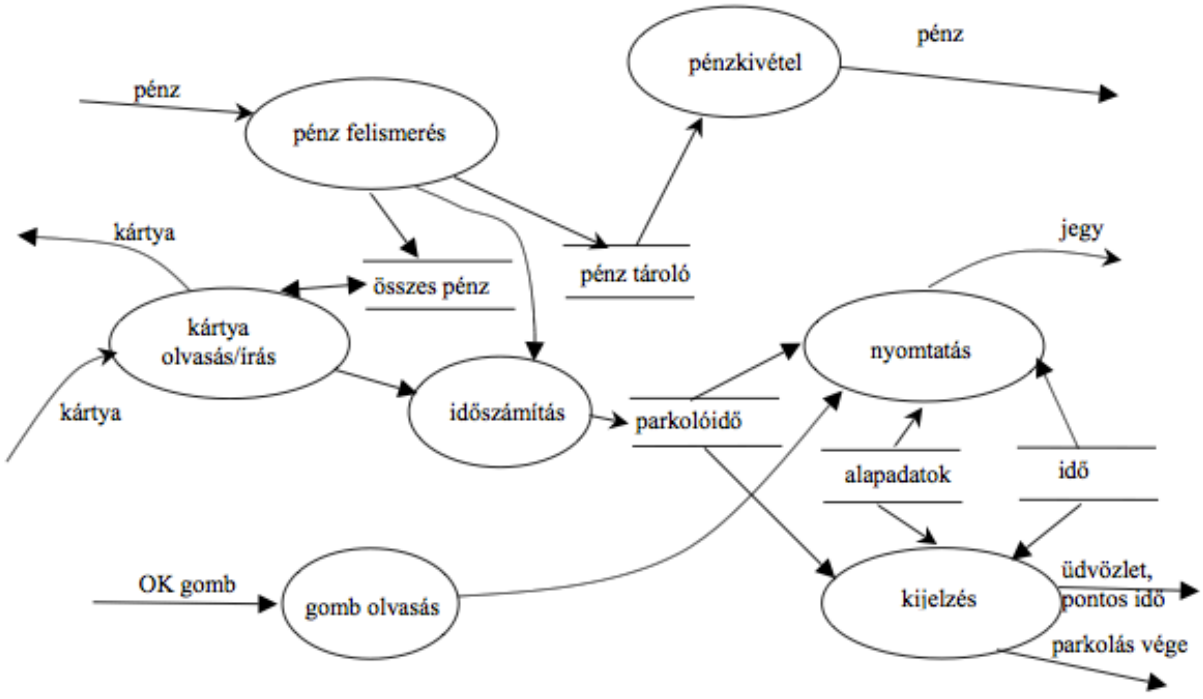




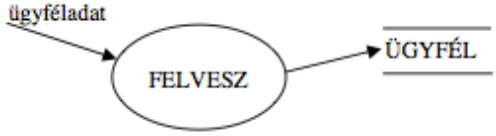
Készítse el egy parkolójegyet kiadó automata adatfolyamábráját ! Adja meg a context diagramot és annak 0 szintű felbontását !

Az automatától pénzért parkolási idő vásárolható. Alaphelyzetben az automata kijelzőjén a pontos idő és üdvözlőszöveg látható. Pénz bedobásakor a kijelzőről leolvashatjuk, hogy a bedobott pénz ellenében meddig parkolhatunk. Az OK gombot megnyomva az automata parkolójegyet nyomtat. Ha a pénz bedobása előtt a parkolókártyánkat az olvasóba helyezzük, akkor kedvezményes tarifáért vásárolhatunk parkolóidőt, amíg a kártyán levő kedvezmény el nem fogy. A parkolóőr speciális kártyájára felíródik a legutóbbi ürítés óta beszedett pénz mennyisége és az automatában összegyűlt pénz kivehető.

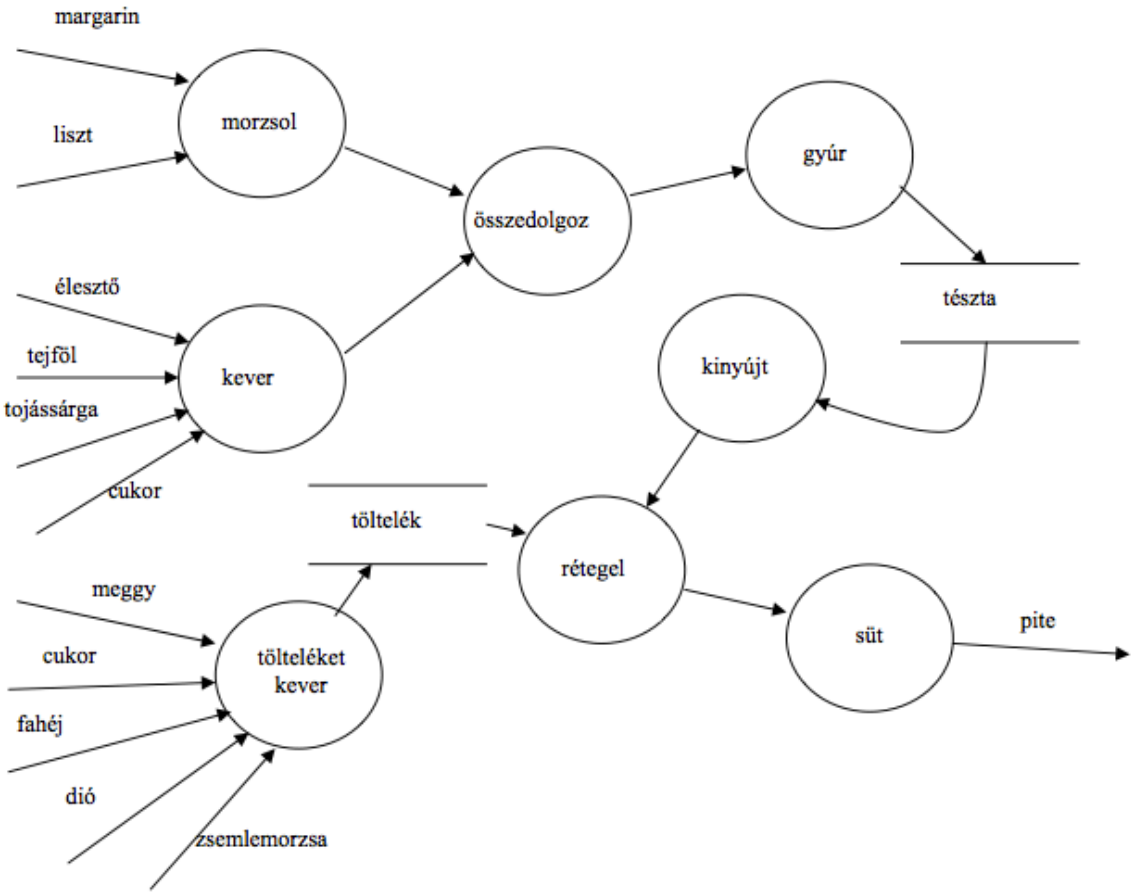




Legyen egy `ügyfél` adattár, amelybe a `felvesz` processz vesz fel ügyfelet. A processz bemenete egy ügyfél adata (`ügyféladat`). Ha az ügyfél már szerepel az adattárban, akkor nincs teendő. Ha az ügyfél még nincs az adattárban, a processz azt beleírja. Rajzoljon DFD részletet a fentiek specifikálására !



Izidor meggyes pitét süt karácsonyra. Rajzoljon adatfolyam ábrát (context diagram NEM KELL!!), amely specifikálja a meggyes pite elkészítésének folyamatát !
A margarint a liszttel elmorzsoljuk. Az élesztőt a tejfölben tojássárgájával és cukorral jól elkeverjük, majd a lisztes margarinnal összedolgozzuk. Meggyúrjuk a tésztát, amit fél óráig hűvös helyen pihentetünk. A meggyet cukorral, fahéjjal, dióval és a zsemlemorzsával keverve elkészítjük a tölteléket. A tésztát kinyújtjuk, majd a töltelékkel rétegelve (tészta, töltelék, tészta) 180 fokon 20-25 percig sütjük.

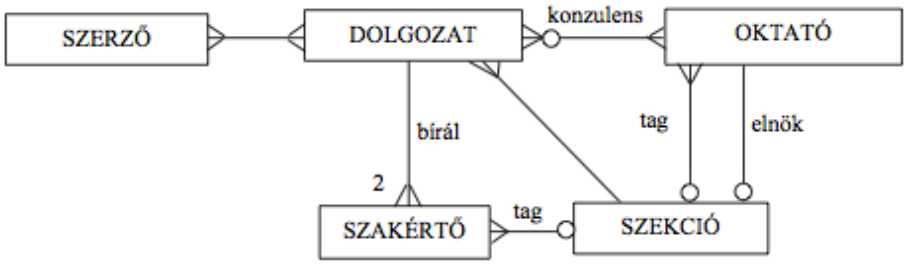


Feladatok Entitás-Reláció Diagram témakörből

Készítsen **entitás-relációs modellt** az alábbi problémára!

A kari TDK konferenciára száznál több dolgozatot nyújtottak be a szerzők. A dolgozatot jellemzi a címe, oldalszáma, és a konferencián szerzett díj. Egy dolgozatnak több szerzője lehet, egy szerző több dolgozatban is érdekelt lehet. A szerzőre jellemző a neve, neptun-kódja és az évfolyama. A dolgozatok oktatók irányításával készültek. Egy oktató több dolgozatot is konzultálhatott, de nem mindegyik oktatónak ért el olyan eredményt a hallgatója, amely eredmény alapján dolgozat született. Van olyan dolgozat is, aminek több konzulense is van. Az oktató jellemzője a neve, beosztása és tanszéke. A bírálat objektivitása érdekében minden dolgozatot pontosan két egyetemen kívüli szakértővel bíráltatnak el. Egy szakértő (név, cég, telefon) csak egyetlen dolgozatot bírál. Minden dolgozatot a témája alapján egy névvel ellátott szekcióba sorolják; egy szekcióba 7-10 dolgozat kerül. A szekciók ülésére különböző termekben, de egyidőben kerül sor. Egy szekció munkáját az elnök irányítja, aki egyetemi oktató. A szekció munkájában szekciónként részt vesz még legalább egy oktató és egy a bírálásban is érintett szakértő. A szekció ülésén tartott előadást is figyelembe véve a dolgozatoknak a szekció díjat adományoz.

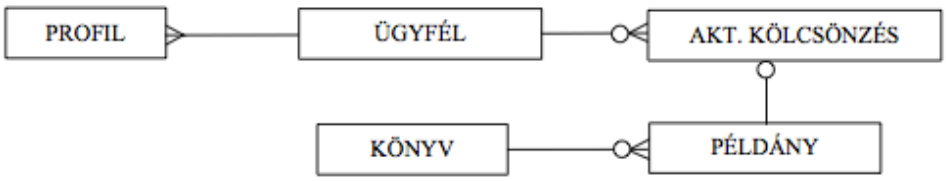
SZERZŐ (név, neptun-kód, évfolyam)
DOLGOZAT (cím, oldalszám, díj)
OKTATÓ (név, beosztás, tanszék)
SZAKÉRTŐ (név, cég, telefon)
SZEKCIÓ (név, terem)



Készítsen **entitás-relációs diagramot** az alábbi problémára !

Egy könyvtárban könyveket lehet kölcsönözni. A könyvekről nyilvántartják a címét, a kiadóját, a kiadás évét valamint kategóriáját (pl.: klasszikus, krimi, gyerek, tudományos stb.). Egy könyvből több példány is lehet, de olyan könyvek adatait is tárolják, amelyekből még vagy már nincs példány. Példányonként tárolják a vásárlás idejét, az árat és az adott példány kölcsönzéseinek számát. Az ügyfeleket első kölcsönzésük alkalmával veszik nyilvántartásba. Az ügyfelet azonosító jellemzi, de nyilvántartják a nevet, e-mail címet, telefonszámot és az éves tagdíj lejárati dátumát is. Egy kölcsönzésben egy ügyfél egy könyv példányt megadott határidőig kölcsönbe vesz. A kölcsönzés megszűnik, ha az ügyfél a könyvet viszahozta. Egy időben egy ügyfél több könyvet is kölcsönözhet. Az ügyfelekről profilt készítenek, amely tartalmazza, hogy az ügyfél az élete során a különböző kategóriájú könyvekből hányszor kölcsönzött (Ennek alapján lehetséges az ügyfeleknek e-mailben személyre szóló reklámokat küldeni).

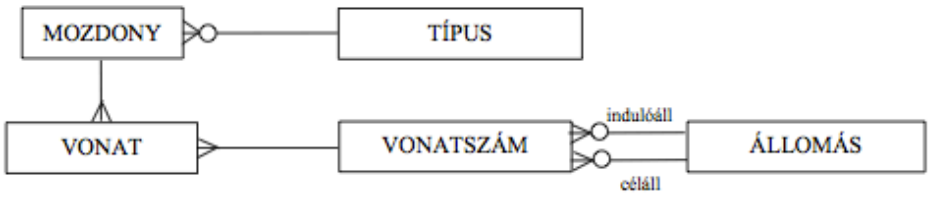
KÖNYV (<u>cím</u>, <u>kiadó</u>, <u>kiadási év</u>, kategória)
PÉLDÁNY (*<u>könyv</u>*, <u>azonosító</u>, vásárlás ideje, ára, kölcsönzések száma)
ÜGYFÉL (<u>azonosító</u>, név, e-mail, telefon, tagdíj lejárat)
AKT. KÖLCSÖNZÉS (*<u>példány</u>*, *<u>ügyfél</u>*, dátum, határidő)
PROFIL (*<u>ügyfél</u>*, <u>kategória</u>, kérések száma)



Készítsen **entitás-relációs diagramot** az alábbi problémára !

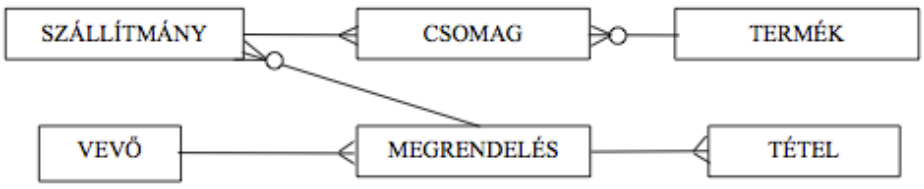
Egy vasúttársaság több mozdonytípust is használ. A típusokra jellemző a fajta (gőz, diesel, villamos), a tengelynyomás és a vonóerő. A típus kódja egy háromjegyű szám vagy egy betű és egy kétjegyű szám. Döntéselőkészítési és összehasonlítási célból olyan típusok adatait is tárolják, amilyen mozdonnyal nem rendelkeznek. A mozdonyokat a típus és egy három vagy négyjegyű sorszám azonosítja, jellemzője még a gyártás éve. A mozdonyok vonatokat továbbítanak, minden vonatot pontosan egy mozdony. A vonat jellemzője az össztömege, a dátum, a késés és a vonatszám. Ugyanazzal a vonatszámmal nem indulhat két vonat ugyanazon a napon. A vonatszámtól függ a vonat fajtája (személy, teher), az induló- és célállomás. Az állomások jellemzője a név, a vágányok száma és a szolgálat létszáma.

TÍPUS (kód, fajta, tengelynyomás, vonóerő,)
MOZDONY (*típus*, sorszám, gyártási év)
VONAT (*vonatszám*, össztömeg, dátum, késés)
VONATSZÁM (szám, fajta, *indulóállomás*, *célállomás*)
ÁLLOMÁS (*név*, vágányok száma, létszám)



Készítsen **entitás-relációs diagramot** az alábbi problémára !

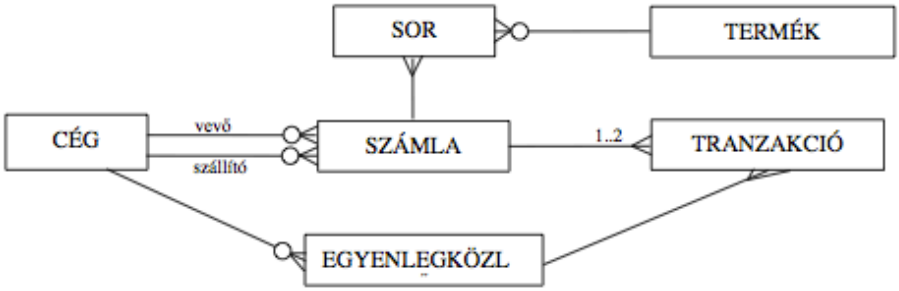
A nyúlzavarógépgyár az általa gyártott termékekről nyilvántartja azok megnevezését, a raktáron levő mennyiséget, az egységárat és az ÁFA kulcsot. A gyárhoz folyamatosan érkeznek a megrendelések (szám, vevő, a szállítás várható ideje, helye). Egy megrendelés legkevesebb egy tételből áll. A tétel megmondja, hogy egy termékből hány darabot akar venni a vásárló. A megrendelés vonatkozhat olyasmire is, amit nem gyártanak. Ilyet persze szállítani sem tud a gyár. Az egy megrendelésen szereplő termékeket a gyár egy vagy több – a mennyiségtől függően – szállítmányban (száma, szállítás dátuma, szállítást bonyolító személy/cég) küldi el a megrendelőnek. Egy szállítmány mindig egy megrendeléshez tartozik. A szállítmány csomagokból áll. Egy csomagban (csomag száma, méret, súly) azonos termékek egy vagy több darabja van. Nyilvántartják még a vevők alapadatait (név, cím, adószám, kapcsolattartó személy).



Készítsen **entitás-relációs diagramot** az alábbi problémára !

A MONEY iroda kisebb cégek pénzügyeit bonyolítja egy olyan világban, ahol nincs készpénz és minden fizetés bankszámlák közötti átutalással történik. A céget jellemző adat a név, a cím, az adószám és a banki folyószámla (egy cégnek csak egy lehet) száma. Az iroda állítja ki a cég – mint szállító – számláit és küldi meg a vevő cégnek. A kiállított számlán szerepel a szállító és a vevő cége, a számla sorszáma, dátuma, az összeg és az ÁFA. A számla szállítója mindig az iroda ügyfele (hiszen csak ügyfele nevében állíthat ki számlát az iroda), a vevő cég általában nem ügyfél, de lehet az is. Az iroda a vevő cégek adatait is nyilvántartja, még akkor is, ha azok nem az iroda ügyfelei. A számla egy sorában a szállított termék, annak mennyisége, nettó ára és ÁFA-ja áll. Egy számla több sorból állhat. A számlák kiállítását egyszerűsíti a termékjegyzék, amelyben megtalálható minden termék kódja, megnevezése és ÁFA kulcsa. Természetesen az iroda fogadja az ügyfeleinek érkező számlákat és intézkedik a kifizetésről. Az iroda megkapja ügyfeleinek bankjától az egyenlegértesítést. Ezen szerepelnek: az értesítő sorszáma, a dátuma, az egyenleg, valamint a legutóbbi értesítést követően lezajlott tranzakciók (legalább egy). Egy tranzakció egy pénzmozgást rögzít. Megadja a mozgás összegét, irányát (- kiadás , + bevétel) és hivatkozik a pénzmozgást kiváltó számlára.

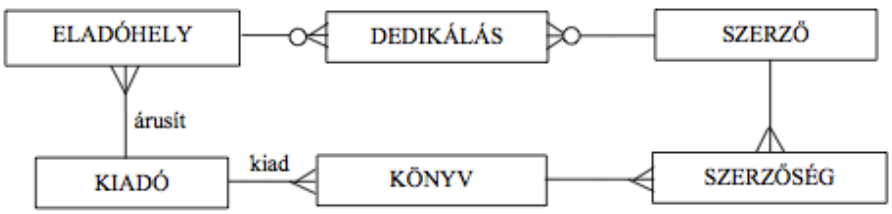
CÉG (név, cím, adószám, folyószámla, ügyfél-e)
SZÁMLA (*cég(szállító)*, számlaszám, *cég(vevő)*, összeg, ÁFA, dátum)
SOR (*számla*, sorszám, *termék*, mennyiség, nettóár, ÁFA)
TERMÉK (termékkód, terméknév, ÁFA-kulcs)
EGYENLEGKÖZLŐ (*cég*, sorszám, dátum, egyenleg)
TRANZAKCIÓ (*egyenlegközlő*, sorszám, összeg, *számla*)



Készítsen **entitás-relációs diagramot** az alábbi problémára !

A könyvhétre minden kiadó (neve, címe azonosítja) több új könyvet jelentetett meg. A könyvre jellemző a címe, az ára és a példányszáma; ugyanazt a könyvet nem adta ki több kiadó. Egy könyvnek több szerzője (név, cím, életkor) is lehet, egy szerző több könyvnek is lehet a (társ)szerzője. Nyilvántartjuk azt is, hogy egy szerző egy könyv írásában való közreműködésért mennyi honoráriumot kapott. A kiadók az új könyveket csak a saját eladóhelyeiken (bolt vagy sátor) árulták. A szerzők az eladóhelyeken dedikáltak. Egy eladóhelyen egy időben mindig csak egyetlen szerző dedikált. Előfordult olyan, hogy ugyanazon a napon ugyanazon a helyen egy szerző két különböző időben is dedikált, de voltak olyan eladóhelyek ahol egyáltalán nem volt dedikálás, és szép számmal voltak írók, akik nem dedikáltak egyszer sem. A dedikálások jellemzője a megjelent olvasók száma. Az eladóhelyet jellemzi még, hogy mennyi bevételt, forgalmat produkált a könyvhéten.

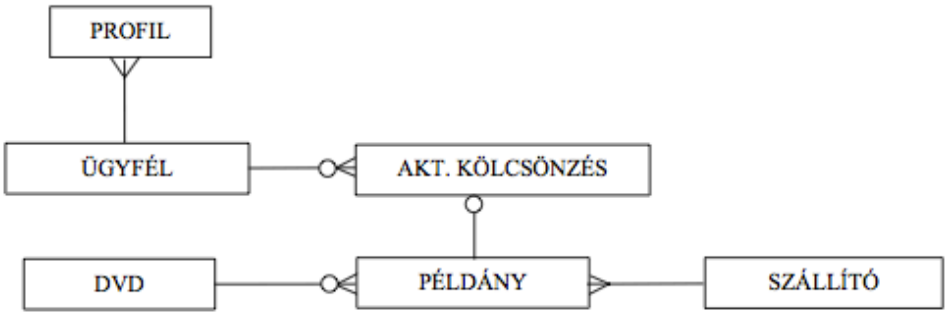
KIADÓ (neve, címe)
KÖNYV (cím, *kiadó*, ár, példányszám)
SZERZŐSÉG (*könyv*, *szerző*, honorárium)
SZERZŐ (név, cím, kor)
DEDIKÁLÁS (*szerző*, *eladóhely*, dátum, idő, megjelentek száma)
ELADÓHELY (cím, *kiadó*, fix/ideigl, forgalom)



Készítsen **entitás-relációs diagramot** az alábbi problémára !

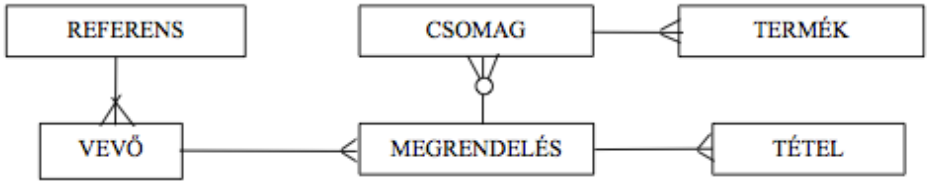
Egy videotékában DVD-ket lehet kölcsönözni. A DVD-kről nyilvántartják a címét, a kiadóját, a kiadás évét, a műsor hosszát, valamint kategóriáját (pl.: zene, akció, romantikus, vígjáték, krimi, thriller, sex, stb.). Egy DVD-ből több példány is lehet, de olyan DVD-k adatait is tárolják, amelyekből még vagy már nincs példány. Példányonként tárolják a vásárlás idejét, az árat, a szállítót és az adott példány kölcsönzéseinek számát. A szállítóról ismerik a nevét, címét és bankszámla számát. Az ügyfeleket első kölcsönzésük alkalmával veszik nyilvántartásba. Az ügyfelet azonosító jellemzi, de nyilvántartják a nevet, címet, telefonszámot és az éves tagdíj lejárati dátumát is. Egy kölcsönzés alkalmával egy ügyfél egy DVD példányt megadott határidőig kölcsönbe vesz. A kölcsönzés megszűnik, ha az ügyfél a DVD-t viszahozta. Az ügyfelekről profilt készítenek, amely tartalmazza, hogy az ügyfél az élete során a különböző kategóriájú DVD-kből hányszor kölcsönzött (Ennek alapján lehetséges az ügyfeleknek személyre szóló reklámokat küldeni).

DVD (cím, kiadó, kiadási év, hossz, kategória)
PÉLDÁNY (*dvd*, azonosító, vásárlás ideje, ára, kölcsönzések száma, *szállító*)
SZÁLLÍTÓ (név, cím, bankszámlaszám)
ÜGYFÉL (azonosító, név, cím, telefon, tagdíj lejárat)
AKT. KÖLCSÖNZÉS (*példány*, *ügyfél*, dátum, határidő)
PROFIL (*ügyfél*, kategória, kérések száma)



Készítsen **entitás-relációs modellt** az alábbi problémára!

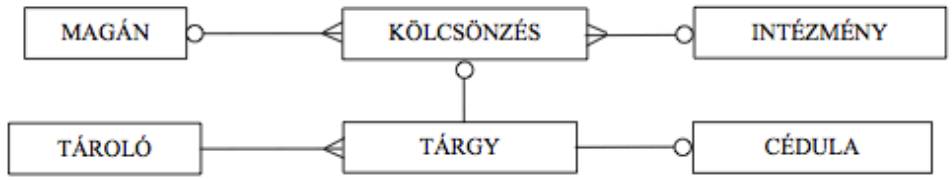
Egy csomagküldő szolgálatnál referensek (név, beosztás, gyakorlat) intézik a vevőktől (név, cím, adószám) beérkező megrendeléseket. Minden új vevőhöz hozzárendelnek egy referenst, ő felel a továbbiakban a vevőért, egy referens sok vevővel tart kapcsolatot. A megrendelés (dátum, szállítási cím) legalább egy tételt tartalmaz. Egy tétel áll a megrendelt dolog megnevezéséből, cikkszámából és a kívánt darabszámból. A tételeknek megfelelő termékeket (cikkszám, egyedi gyári szám, gyártó) a csomagküldő hosszabb-rövidebb idő alatt beszerzi, majd amikor minden termék beérkezett, azokat csomagokba rendezi és elküldi a vevőnek. Egy megrendeléshez több csomag is tartozhat, de egy csomag mindig egyetlen megrendeléshez kapcsolódik. Egy csomagban egy vagy több megrendelt termék kerül leszállításra.



Készítsen **entitás-relációs diagramot** az alábbi problémára !

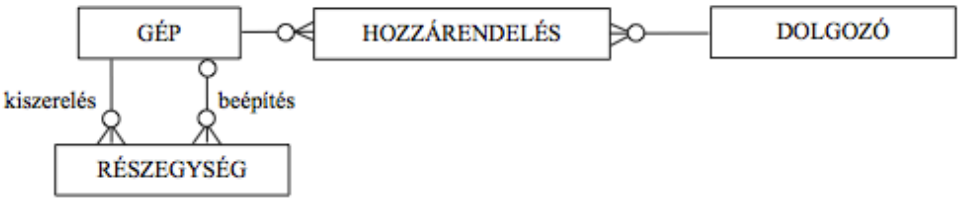
A múzeumban tárgyakat állítanak ki. Minden tárgynak van egy egyedi azonosítója, de ismert a megnevezése, értéke és kategóriája (dísztárgy, használati tárgy, fegyver, könyv, stb.) is. A tárgyakat tárolókban helyezik el. A tárolónak van egy száma, tudják róla, hogy melyik teremben van és milyen a fajtája (asztal, üveges vitrin, polc, stb.). Egy tárolóban legalább egy tárgy található. A kiállított tárgyak között vannak olyanok is, amelyeknek nem a múzeum a tulajdonosa, hanem kölcsönbe kapták. Ezekről ismert, hogy mettől meddig tart a kölcsönzés, melyik biztosító biztosítja a tárgyat és kitől kapták kölcsönbe. A kölcsönadó lehet magánszemély vagy intézmény. Magánszemély esetében nyilvántartják a személy nevét, címét és telefonszámát. Intézmény esetén annak nevét és a főhatóság elnevezését. Ugyanazon személytől vagy intézménytől több tárgy is lehet kölcsönben. Csak azon személyek és intézmények adatait tárolják, akiktől aktuálisan kölcsönöz a múzeum. A múzeum a tulajdonát képező tárgyakat alkalmanként átadja más kiállításokon történő bemutatásra. Ez esetben a tárgyat egy cédulával helyettesítik a tárolóban. A cédulán szerepel annak sorszáma, a kiállítás dátuma, a lejárat időpontja és az engedélyező személy neve.

TÁRGY (*tároló*, megnevezés, kategória, érték, azonosító)
TÁROLÓ (szám, terem, fajta)
CÉDULA (*tárgy*, szám, dátum, határidő, engedélyező)
KÖLCSÖNZÉS (*tárgy*, mettől, meddig, biztosító, *magán*, *intézmény*)
MAGÁN (név, cím, telefon)
INTÉZMÉNY (név, főhatóság)



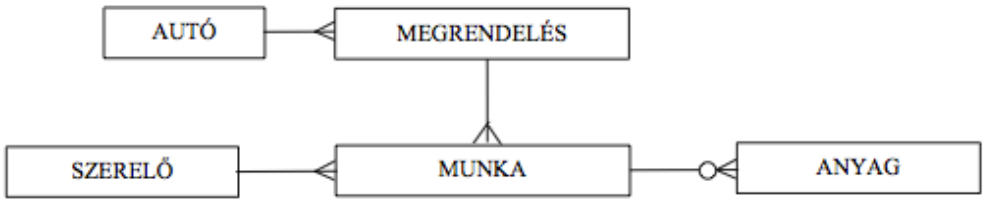
Az X cégnél nyilvántartják, hogy a dolgozóknál (név, adószám) milyen számítógép van. A gépet jellemzi típusa, gyári száma és ára. Aki új gépet kap, annak korábbi gépét általában más dolgozónak adják tovább, vagyis a gépek vándorolnak. A nyilvántartásból vissza kell tudni keresni, hogy egy gép mely időszakban melyik dolgozónál volt és a gépet ki installálta ott. Vannak olyan dolgozók, akiknek nincs gépük, és vannak tartalék gépek is. Bármely gépből (különböző okokból) részegységeket (tápegység, alaplap, diszk stb.) szerelhetnek ki. A részegységet jellemzi a megnevezése és száma, valamint az, hogy melyik gépben volt eredetileg. A működőképes részegységek más gépekbe beépíthetők. Egy gépből több részegységet is kiszerelhetnek, egy gépben több beépített részegység is lehet. A részegységről azt is tudni kell, hogy aktuálisan melyik gépben van, mikor és ki installálta ott. Rajzoljon entitás-relációs diagramot !

GÉP (típus, gyári szám, ár)
DOLGOZÓ (név, adószám)
HOZZÁRENDELÉS (*gép*, *dolgozó*, időszak, installátor)
RÉSZEGYSÉG (megnevezés, szám, *eredeti_gép*, *aktuális_gép*, beépítési_idő, installátor)



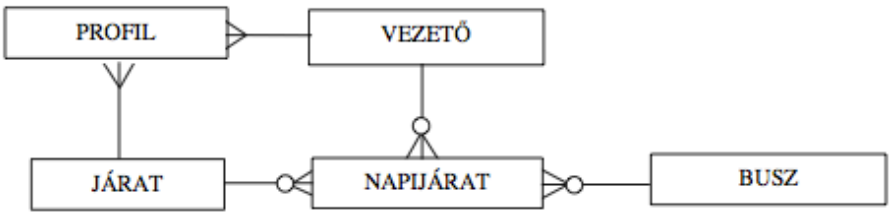
Egy autószervízben az autót azonosítja az alvázszám, de rögzítik a rendszámot, a gyártót és a típust is. A szerelőkről tudjuk nevüket, végzettségüket, korukat és a munkaviszonyuk kezdetét. Az ügyfél megrendelése tartalmazza a kérést (pl. motor beállítás, olajcsere), a megrendelés felvételének időpontját, a várható költséget. A megrendelést annak száma azonosítja. A megrendelés teljesítése munkával történik. A munkavégzés kapcsán nyilvántartjuk az elvégzésre ténylegesen ráfordított időt és azt, hogy a munkát ki végezte. A munka közben felhasznált anyagokról tudjuk azok megnevezését és mennyiségét. Egy munkához nem mindig használunk anyagot, viszont az anyagfelhasználás mindig munkához kapcsolódik. Egy megrendelés teljesítéséhez legalább egy munkát el kell végezni. Egy autón egy munkát egy szerelő végez. Munkavégzés csak megrendelés alapján történhet. Egy megrendelés mindig egy autóhoz kapcsolódik, egy autóhoz az idők során több megrendelés tartozhat.
Rajzoljon entitás-relációs diagramot !

AUTÓ (rendszám, alvázszám, gyártó, típus)
SZERELŐ (név, végzettség, életkor, munkaviszony kezdete)
MEGRENDELÉS (*autó*, kérés, felvétel ideje, szám, várható költség)
MUNKA (*megrendelés*, tevékenység, tényleges idő, *szerelő*)
ANYAG (*munka*, megnevezése, mennyisége)



A Grimbusz társaság aktuális és tervezett buszjáratait megtaláljuk a menetrendben. A járat jellemzője a járatszám, az induló és célállomás. Egy adott napon a járatra vonatkozóan nyilvántartjuk a késést. Egy napon nem indul többször ugyanazon számú járat. A konkrét napi járathoz a buszt és a vezetőt az éppen rendelkezésre állók közül választják. Egy konkrét járattal csak egy vezető megy. Minden vezetőről tudjuk, hogy melyik járaton hány alkalommal vezetett. A napi járatokat 60 nap elteltével töröljük. Rajzoljon entitás-relációs diagramot !

JÁRAT (szám, indulás, cél)
NAPIJÁRAT (*járat*, dátum, késés, *busz*, *vezető*)
BUSZ (rendszám, megtett km)
VEZETŐ (név, jogsi száma)
PROFIL (*járat*, *vezető*, darab)



Feladatok XML témakörből

Az alábbi XML leírás jól formált ? Ha nem, akkor mi a baja ?

```
<?xml version="1.0"?>
<uzenet>
<cim iranyitoszam=1117>Budapest
<utca>Magyar tudosok korutja</utca></cim>
<felado>LZ</felado>
</uzenet>
```

☐ IGEN
☒ NEM

attributum nincs idézőjelek között.....

Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  ((a|(b+,c?)+), d)>
  <!ATTLIST b d CDATA #REQUIRED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
  <!ELEMENT d   (#PCDATA)>
]>
<x>
    <b d="nem d">bbbb</b>
    <d><![CDATA[</d>
    <b> bbbb </b>
    <d>]]></d>
</x>
```

☒ IGEN
☐ NEM

Hiba ?: .....

.....

Az alábbi XML leírás jól formált ? Ha nem, akkor mi a baja ?

```
<?xml version="1.0"?>
<targy>Programozas technologiaja</targy>
<neptun>VIFO2228</neptun>
<vizsga>elso</vizsga>
<torzs>nem is nehez</torzs>
```

☐ IGEN
☒ NEM

nincs gyökere.....

Az alábbi XML leírás jól formált ? Ha nem, akkor mi a baja ?

```
<?xml version="1.0"?>
<uzenet>
<cim varos="Budapest">"Budapest"
<utca>Hintalo</utca>777</cim>
<felado>LZ</felado>
</uzenet>
```

☒ IGEN
☐ NEM

Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  ((a,c*)|b)>
  <!ATTLIST a d CDATA #REQUIRED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
]>
<x>
    <a d></a>
    <c>zh</c>
</x>
```

☐ IGEN
☒ NEM

Hiba ?: A paraméterből hiányzik az egyenlőségjel és az érték .....

Az alábbi XML leírás jól formált? Ha nem, akkor mi a baja?

```
<?xml version="1.0"?>
<dolgozat>
<egyebkent nyul="Na mi lesz ?"> Marha-sag
<tilos>Loch Ness-i szorny</tilos></egyebkent>
<szemely>MZ/X</szemely>
<szemely>MZ</szemely>
</dolgozat>
```

☒ IGEN
☐ NEM

Az alábbi XML leírás jól formált? Ha nem, akkor mi a baja?

```
<?xml version="1.0"?>
<pt-vizsga>
<pelda_xml datum="tegnap"> 2006 majus 23
</pelda_xml>
<xxx/>kapufa
</pt-vizsga>
```

☒ IGEN
☐ NEM

Az alábbi XML leírás jól formált ? Ha nem, akkor mi a baja ?

```
<?xml version="1.0"?>
<zh>
<most>Szoftvertechnologia</most>
<most/>viiia217<most/>viiia217
<most>december elsejen
<torzs>penteken tartjuk</torzs>
</most></zh>
```

☒ IGEN
☐ NEM

Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE a [
  <!ELEMENT a  ((b|(c?, d+)), c)>
  <!ATTLIST d d CDATA #REQUIRED>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
  <!ELEMENT d   (#PCDATA)>
]>
<a><b/><c/></a>
```

☒ IGEN
☐ NEM

Hiba ?: .....

Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszer-kezetet, amelyben van d elem ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<a>
  <d d="xxx">dddd</d>
  <c/>
</a>
```

Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  ((b|(a+,c+)), a)>
  <!ATTLIST b d CDATA #REQUIRED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
]>
<x>
    <b>d="f"</b>
    <a/>
</x>
```

☐ IGEN
☒ NEM

Hiba ?: -nek nincs paramétere .....

Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van c elem ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
    <a>aaa</a>
    <c>aaa</c>
    <a>aaa</a>
</x>
```

Adott az alábbi dekorált XML leírás.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE a [
  <!ELEMENT a   ((b,(c?, d*))|c)>
  <!ELEMENT b   ANY>
  <!ELEMENT c   (#PCDATA)>
  <!ELEMENT d   (#PCDATA)>
]>
<a><b><a>
<b><c>b</c>
</b><c>d</c>①
<c>a</c></a>
```

Mi állhat ① helyében, hogy az XML érvényes legyen ?

- ☐
- ☐
- ☐ </c><a/>
- ☒
- ☐ <c/>
- ☐ értelmetlen a kérdés, mert a DTD nem jól formált

Adott az alábbi dekorált XML leírás.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x    (#PCDATA|b|c) ①>
  <!ELEMENT b    ANY>
  <!ELEMENT c    (#PCDATA)>
]>
<x>
      ②
      <x>januar<c>26</c>
      </x>vizsga</b><b><b/>/x</b>
</x>
```

Mit írna ① helyére, hogy a DTD jól formált legyen ?

- ☐ semmit
- ☒ *
- ☐ +
- ☐ + vagy *
- ☐ egyéb:

Feltételezve, hogy a DTD jól formált, mi állhat ② helyében, hogy az XML érvényes legyen ?

- ☐ semmi
- ☒ <b>/x
- ☒ /x<b>
- ☐ <b/>
- ☐ <b>x<b>

Készítsen kettő különböző, az alábbi DTD szerint érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<!DOCTYPE x [
  <!ELEMENT x   (a?)>
  <!ELEMENT a   (b*,(c|d))>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
  <!ELEMENT d   (#PCDATA)>
]>
```

```
<x> </x>
```

```
<x>
  <a>
    <c>akarmi</c>
  </a>
</x>
```

Készítsen az alábbi DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben pontosan egy darab j elem van ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE f [
  <!ELEMENT j   (#PCDATA)>
  <!ELEMENT h   (#PCDATA)>
  <!ELEMENT g   (f, j)>
  <!ELEMENT f   (g | h)>
]>
```

```
<f>
   <g>
      <f>
         <h>hhh</h>
      </f>
      <j>jjj</j>
   </g>
</f>
```

Készítsen kettő különböző struktúrájú, az alábbi DTD szerint érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<!DOCTYPE x [
  <!ELEMENT x   (d|b)>
  <!ELEMENT d   (a,c*)>
  <!ELEMENT a    (#PCDATA)>
  <!ELEMENT b    (#PCDATA)>
  <!ELEMENT c    (#PCDATA)>
]>
```

```
<x>
  <b>barmi</b>
</x>
```

```
<x>
  <d>
    <a>akarmi</a>
  </d>
</x>
```

Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

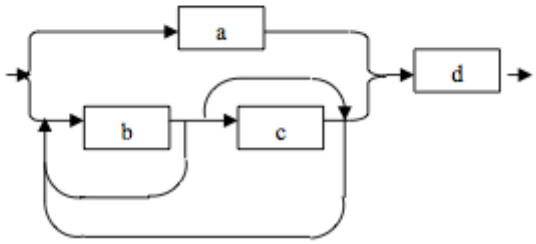
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  ((a|(b+,c?)+), d)>
  <!ATTLIST b d CDATA #REQUIRED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
  <!ELEMENT d   (#PCDATA)>
]>
<x>
    <b d="nem d">bbbb</b>
    <d><![CDATA[</d>
    <b> bbbb </b>
    <d>]]></d>
</x>
```

☒ IGEN
☐ NEM

Hiba ?: .....

.....

Rajzolja fel a DTD-vel definiált adatszerkezet szintaxis gráfját. A DTD esetleges hibáját hagyja figyelmen kívül !



Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

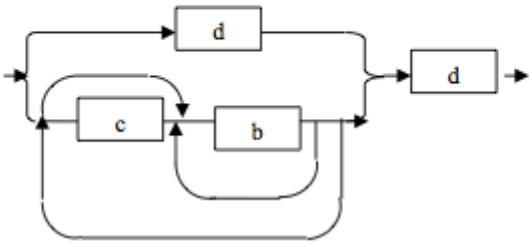
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  ((d|(c?,b+)+), d)>
  <!ATTLIST b d CDATA #IMPLIED>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
  <!ELEMENT d   (#PCDATA)>
]>
<x>
    <d>b="Vizsga"</d>
    <d><![CDATA[</d>
    <x>vizsga</x>
    <d>]]></d>
</x>
```

☒ IGEN
☐ NEM

Hiba ?: .....

.....

Rajzolja fel a DTD-vel definiált adatszerkezet szintaxis gráfját. A DTD esetleges hibáját hagyja figyelmen kívül !



Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  ((a,c*)|b)>
  <!ATTLIST a d CDATA #IMPLIED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
]>
<x>
    <a></a>
    <c>zh</c>
</x>
```

☒ IGEN
☐ NEM

Hiba: .....

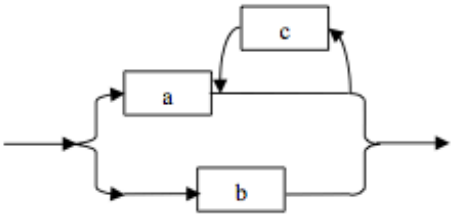
Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van attribútum is ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
    <a d="x">aa</a>
</x>
```

Írja fel az adatszerkezet algebrai definícióját !

x = [(a + {c})|b]

Rajzolja fel az adatszerkezet szintaxisgráfját !



Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  (((a+,c)|b), a)>
  <!ATTLIST a d CDATA #IMPLIED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
]>
<x>
    <a/><c></c>
    <a d="f">b a d=f</a>
</x>
```

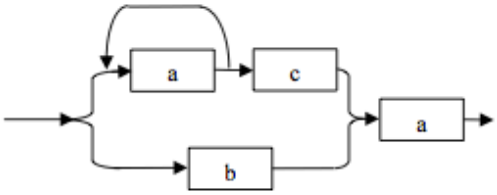
☒ IGEN
☐ NEM

Hiba ?: .....

Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van b elem ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
    <b></b>
    <a></a>
</x>
```

Rajzolja fel az adatszerkezet szintaxisgráfját !



Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  ((a,c*)|b)>
  <!ATTLIST a d CDATA #REQUIRED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
]>
<x>
    <a d></a>
    <c>zh</c>
</x>
```

☐ IGEN
☒ NEM

Hiba: A paraméterből hiányzik az egyenlőségjel és az érték. .....

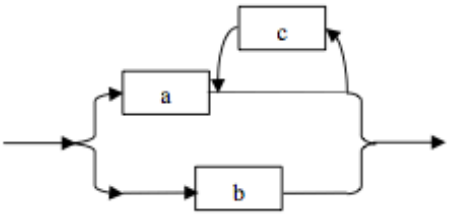
Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van c elem ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
    <a d="x">aaa</a>
    <c>ccc</c>
</x>
```

Írja fel az adatszerkezet algebrai definícióját !

$$x = [(a + \{c\})|b]$$

Rajzolja fel az adatszerkezet szintaxisgráfját !



Feladatok Algebrai axiómák témakörből

Jellemezzünk egy (x,y) párokból álló listát – ahol x egy kulcs, y egy 0-nál nagyobb egész érték – az alábbi műveletekkel! Adja meg az algebrai axiómákat! Az axiómák felírásakor használhatja a két egész összehasonlítására szolgáló műveleteket.

NEW( )	új (üres) listát hoz létre.
ADD(l,(x,y))	az **l** listához kapcsolja az **x,y** párt, ha **x** nem szerepelt a listán. Ha a listán már van **x**, akkor a hozzá tartozó értéket az új **y**-nal helyettesíti, ha az nagyobb a listán szereplőnél.
VALUE(l,x)	megadja az **l** listán az **x** kulcshoz tartozó **y**-t. Ha a listán a megadott **x** nem szerepel, akkor az eredmény 0.
MAX(l)	a listában szereplő legnagyobb **y**. (üres lista esetén nulla)

VALUE(NEW()) = 0	0.5 pont
MAX(NEW()) = 0	0.5 pont
VALUE(ADD(l, (x,y),z)) = if (x == z && VALUE(l,x) < y) y else VALUE(l, z)	2 pont
MAX(ADD(l, (x,y))) = if (MAX(l) < y)) y else MAX(l)	1 pont

Jellemezzünk egy stringet az alábbi műveletekkel ! Adja meg a CUT műveletre vonatkozó algebrai axiómákat (a többi axióma nem kell!) ! Az axiómák felírásakor használhatja a két egész összehasonlítására szolgáló műveleteket.

CRT( )	új (üres) stringet hoz létre.
SET(s,x)	az **s** string elejére rakja az **x** karaktert.
LGTH(s)	az **s** string karaktereinek számát adja.
CUT(s, n)	az **s** string legrégebbi **n** darab karakterének levágása után maradó stringet adja Ha **n** nem kisebb, mint **s** string hossza, üres stringet kapunk. Tételezze fel, hogy **n** > 0 !

CUT(CRT()) = CRT()		-0.5 pont, ha hiányzik
CUT(SET(s, x), n) ==	if (n > LGTH(s)) CRT()	1 pont
	if (n == LGTH(s)) SET(CRT(), x)	1 pont
	if (n < LGTH(s)) SET(CUT(s, n), x)	2 pont

Jellemezzünk egy stringet az alábbi műveletekkel ! Adja meg a PALIN műveletre vonatkozó algebrai axiómákat (a többi axióma nem kell!) ! Az axiómák felírásakor használhatja az egészek és a karakterek összehasonlítására szolgáló műveleteket.

CRT( )	új (üres) stringet hoz létre
LGTH(s)	az **s** string karaktereinek számát adja
TAIL(s)	az **s** string első karakterének levágása után maradó stringet adja
APPEND(s,x)	az **s** string végére rakja az **x** karaktert
HEAD(s)	az **s** string első karakterét mutatja meg
PALIN(s)	igaz, ha az **s** string palindróma

Egy string palindróma, ha az elejéről olvasva ugyanaz, mint visszafelé. Pl.: "görög", "abba".

PALIN(CRT( )) = true
PALIN (APPEND(s, x)) = (LGTH(s) == 0) or (PALIN(TAIL(s)) and (HEAD(s) == x))

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett stringet ! Az axiómák felírásakor megengedett a két karaktert összehasonlító művelet használata.

CRT( )	új (üres) stringet hoz létre.
ADD(s, x)	az **s** string végére rakja az **x** karaktert.
LAST(s)	az **s** string végén álló karaktert adja.
END(s1, s2)	igaz, ha az **s2** string az **s1** string végén áll.
DUPLO(s)	igaz, ha az **s** stringben legalább egyszer dupla karakter fordul elő. Például: a**gg**ódik

LAST(CRT( )) = undefined	0,5 pont
LAST(ADD(s, x)) = x	0,5 pont
END(s, CRT( )) = true	1 pont
END(CRT( ), ADD(s, x)) = false	1 pont
END(ADD(s1, x1), ADD(s2, x2)) = (x1 == x2) and END(s1, s2)	1 pont
DUPLO(CRT( )) = false	1 pont
DUPLO(ADD(s,x)) = (x == LAST(s)) or DUPLO(s)	1 pont

Jellemezzünk egy stringet az alábbi műveletekkel ! A string karakterei előről 1-gyel kezdődően számozottak. Adja meg a PAR műveletre vonatkozó algebrai axiómákat ! Az axiómák felírásakor használhatja a két egész összehasonlítására és két karakter összehasonlítására szolgáló műveleteket.

CRT( )	új (üres) stringet hoz létre.
SET(s,x)	az **s** string elejére – az 1. számú helyre – rakja az **x** karaktert.
PAR(s)	igaz, ha bárhol a stringben egymás mellett legalább két egyforma karakter áll.
IN(s,i)	eredményül adja az **s** string **i**-ik karakterét. Ha **i** nagyobb mint a string hossza, akkor az eredmény értelmetlen (nem definiált).
LGTH(s)	az **s** string karaktereinek számát adja.

PAR(CRT()) = false	-0.5 pont, ha hiányzik
PAR(SET(s, x)) == (LGTH(s) != 0) &&	1 pont
(x == IN(s, 1) ||	1 pont
PAR(s))	1 pont

ez utóbbival egyenértékű:

if (LGTH(s) == 0) false else if (x == IN(s,1) || PAR(s)) true

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett stringet !

CRT( )	új (üres) stringet hoz létre.
LGTH(s)	az **s** string karaktereinek számát adja.
ALL(s)	hamis, ha az **s** string üres vagy benne nem mindegyik karakter egyforma.
ADD(s,x)	az **s** string végére rakja az **x** karaktert.
HEAD(s)	az **s** string első karakterét mutatja meg.

LGTH(CRT()) = 0	-0.5 pont, ha hiányzik
LGTH(ADD(s, c)) = LGTH(s) + 1	0.5 pont
ALL(CRT()) = false	0.5 pont
ALL(ADD(s, c)) = LGTH(s) == 0 || ALL(s) && HEAD(s) == c	1.5 pont
HEAD(CRT()) = undefined	-0.5 pont, ha hiányzik
HEAD(ADD(s, c)) == if (LGTH(s) == 0) c else HEAD(s)	0.5 pont

Algebrai axiómák segítségével specifikálja az egész számoknak az alábbi műveletekkel jellemzett halmazát ! Az axiómákban alkalmazhat egy `odd(x)` →`boolean` függvényt, amely igaz, ha az `x` egész páratlan.

NEW( )	új (üres) halmazt hoz létre.
LGTH(h)	a **h** halmaz elemeinek számát adja.
EVEN(h)	a **h** halmaznak azon részhalmaza, amely csak a páros elemeket tartalmazza.
ADD(h,i)	a **h** halmazhoz hozzáveszi az **i** egészet.
ISIN(h, i)	igaz, ha **i** a **h** halmaz eleme.

LGTH(NEW()) = 0	- 0.5 pont, ha hiányzik
ISIN(NEW(),x) = false	- 0.5 pont, ha hiányzik
EVEN(NEW()) = NEW()	- 0.5 pont, ha hiányzik
LGTH(ADD(h, x)) = if (ISIN(h,x)) LGTH(h) else LGTH(h) + 1	1 pont
ISIN(ADD(h, x), y) = (x==y) || ISIN(h, y)	1 pont
EVEN(ADD(h, x)) = if (odd(x)) EVEN(h) else ADD(EVEN(h), x)	2 pont

Jellemezzünk egy stringet az alábbi műveletekkel ! Adja meg a TAIL műveletre vonatkozó algebrai axiómákat (a többi axióma nem kell!) ! Az axiómák felírásakor használhatja a két egész összehasonlítására szolgáló műveleteket.

NEW( )	új (üres) stringet hoz létre.
ADD(s,x)	az **s** string végére rakja az **x** karaktert.
LGTH(s)	az **s** string karaktereinek számát adja.
TAIL(s, n)	az **s** string legrégebbi **n** darab karakterének levágása után maradó stringet adja Ha **n** nem kisebb, mint **s** string hossza, üres stringet kapunk. Tételezze fel, hogy **n** > 0 !

TAIL(NEW()) = NEW()		-0.5 pont, ha hiányzik
TAIL(ADD(s, x), n) ==	if (n > LGTH(s)) NEW()	1 pont
	if (n == LGTH(s)) ADD(NEW(), x)	1 pont
	if (n < LGTH(s)) ADD(TAIL(s, n), x)	2 pont

Jellemezzünk egy listát az alábbi műveletekkel ! Adja meg a CUT műveletre vonatkozó algebrai axiómákat (a többi axióma nem kell!) ! Az axiómák felírásakor használhatja a két egész összehasonlítására szolgáló műveleteket.

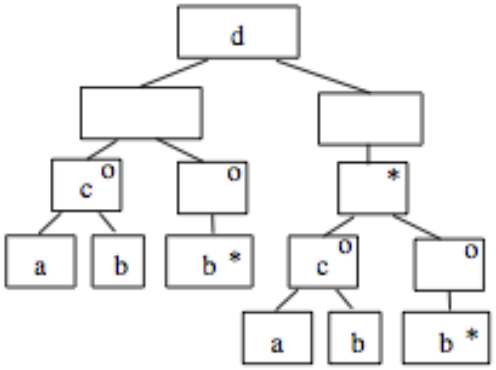
CRT( )	új (üres) listát hoz létre.
SET(l,x)	az **l** lista elejére rakja az **x** elemet.
LGTH(l)	az **l** lista elemeinek számát adja.
CUT(l, n)	az **l** lista legrégebbi **n** darab elemének levágása után maradó listát adja Ha **n** nem kisebb, mint **l** lista hossza, üres listát kapunk. Tételezze fel, hogy **n** > 0 !

CUT(CRT()) = CRT()		-0.5 pont, ha hiányzik
CUT(SET(l, x), n) ==	if (n > LGTH(l)) CRT()	1 pont
	if (n == LGTH(l)) SET(CRT(), x)	1 pont
	if (n < LGTH(l)) SET(CUT(l, n), x)	2 pont

Feladatok Jackson-ábra témakörből

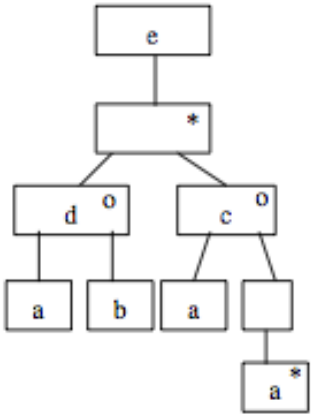
Rajzolja fel az alábbi DTD-vel specifikált adatszerkezetet Jackson ábrával !

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE d [
  <!ELEMENT a (#PCDATA)>
  <!ELEMENT b (#PCDATA)>
  <!ELEMENT c (a, b)>
  <!ELEMENT d (c|b*)+>
]>
```

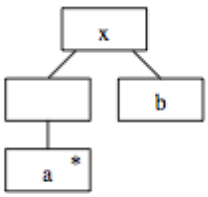


Rajzolja fel az alábbi DTD-vel specifikált adatszerkezetet Jackson ábrával !

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE e [
  <!ELEMENT a (#PCDATA)>
  <!ELEMENT b (#PCDATA)>
  <!ELEMENT c (a+)>
  <!ELEMENT d (a, b)>
  <!ELEMENT e (d|c)*>
]>
```



Egészítse ki az alábbi DTD vázat úgy, hogy az a mellékelt JSP struktúra szerinti adatszerkezetet definiálja ! Az a elemnek legyen egy opcionális c attribútuma is !

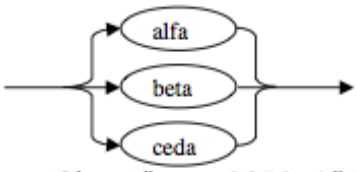


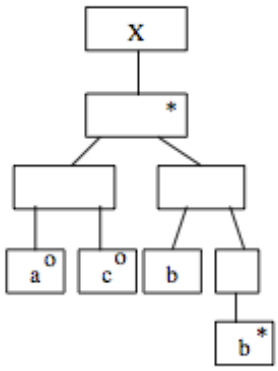

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
   <!ELEMENT x (a*,b)>
   <!ATTLIST a c CDATA #IMPLIED>
  <!ELEMENT a    (#PCDATA)>
  <!ELEMENT b    (#PCDATA)>
]>
```

Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van c attribútumot tartalmazó a elem is ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
    <a c="CCC"></a>
    <b></b>
</x>
```

Egészítse ki az alábbi DTD vázat úgy, hogy az a Jackson-ábra szerinti adatszerkezetet definiálja ! A c elemnek legyen az alábbiak szerint definiált y nevű, predefinit alfa értékű attribútuma is !



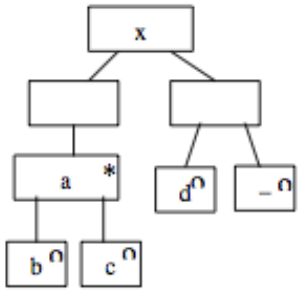



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
   <!ELEMENT x ((a|c), b+)*>
   <!ATTLIST c y (alfa|beta|ceda) "alfa">
  <!ELEMENT a    (#PCDATA)>
  <!ELEMENT b    (#PCDATA)>
  <!ELEMENT c    (#PCDATA)>
]>
```

Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van a, b és c elem, ez utóbbi beta attribútummal ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
    <c y="beta"></c>
    <b></b>
    <a></a>
    <b></b>
</x>
```

Egészítse ki az alábbi DTD vázat úgy, hogy az a mellékelt JSP struktúra szerinti adatszerkezetet definiálja ! A **c** elemnek legyen egy opcionális **y** attribútuma is !




```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
   <!ELEMENT x (a*,d?)>
   <!ELEMENT a (b|c)>
   <!ATTLIST c y CDATA #IMPLIED>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
  <!ELEMENT d   (#PCDATA)>
]>
```

Készítsen az ábrának megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van két **c** elem és egy **d** elem is ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
  <a> <c ></c> </a>
  <a> <c y="kedd"></c> </a>
  <d></d>
</x>
```

Az alábbi XML leírás jól formált és érvényes-e ? Ha nem, akkor mi a baja?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x  (((a,c+)|b), a)>
  <!ATTLIST a d CDATA #REQUIRED>
  <!ELEMENT a   (#PCDATA)>
  <!ELEMENT b   (#PCDATA)>
  <!ELEMENT c   (#PCDATA)>
]>
<x>
    <a d="|b"></a>
    <c>|b</c>
    <c>"|b"</c>
    <a d=|b></a>
</x>
```

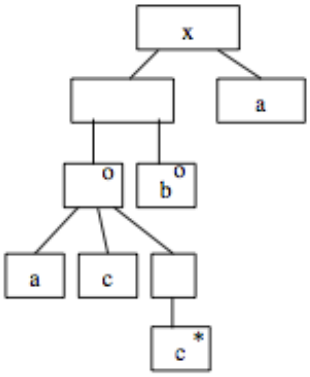
☐ IGEN
☒ NEM

Hiba ?: a második “a” attribútum paramétere nincs idézve.....

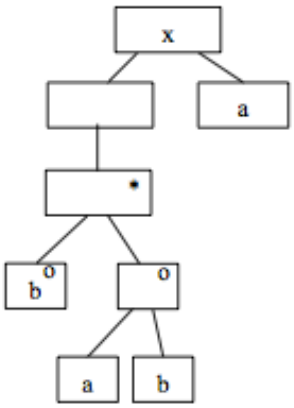
Készítsen a DTD-nek megfelelő érvényes (valid) és szintaktikailag helyes (jól formált) XML adatszerkezetet, amelyben van **b** elem ! Az XML deklaráció (<?xml version="1.0" encoding="ISO-8859-1"?>) nem kell.

```
<x>
    <b>dddd</b>
    <a d="xxx"></a>
</x>
```

Rajzolja fel a DTD-nek megfelelő adatszerkezet Jackson-ábráját !



Adott az alábbi Jackson-ábra.



Jelölje meg, hogy melyik elv jelenik meg a szerkezetben egynél többször !

- [x] szekvencia (sorrend)
- [ ] szelekció (választás)
- [ ] iteráció (ismétlődés)
- [ ] egyik sem

Az alábbiak közül ① helyén melyik válasz írja le helyesen a Jackson-ábrát ?

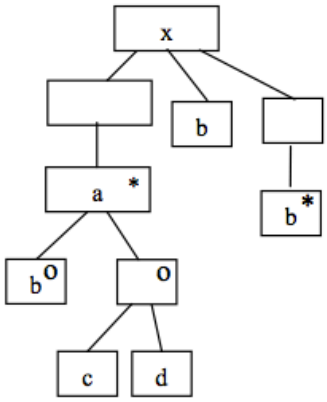
- [ ] `(a*,b|(a,b))`
- [x] `(((a,b)|b)*,a)`
- [ ] `(b*,(a,b)|a)`
- [ ] `((b,(a,b)*),a)`
- [x] `((b|(a,b))*,a)`
- [ ] `((b|(a,b)*),a)`
- [ ] `(((a,b)|b*),a)`
- [ ] egyik sem

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x      ①    >
  <!ELEMENT a    (#PCDATA)>
  <!ELEMENT b    (#PCDATA)>
]>
<x>
    <a>a</a>
    <b>b</b>
    <b>-x</b>
    <b>x</b>
    <a>-x</a>
</x>
```

A baloldalon álló XML adatszerkezet

- [ ] mint XML szerkezet szintaktikai hibás
- [x] megfelel a Jackson-ábrának (érvényes)
- [ ] nem felel meg a Jackson-ábrának (érvénytelen)
- [ ] érvényessége nem dönthető el

Adott az alábbi ELH-ábra.




```
<?xml version="1.0"
encoding="ISO-8859-1"?>
<!DOCTYPE x [
  <!ELEMENT x    ①>
  <!ELEMENT a    ②)>
  <!ELEMENT b    (#PCDATA)>
  <!ELEMENT c    (#PCDATA)>
  <!ELEMENT d    (#PCDATA)>
]>
<x>
    <b>a</b>
    <b>cd</b><b>*</b>
    <b>b,b*</b>
</x>
```

Az alábbiak közül ① helyén melyik válasz írja le helyesen az ELH-ábrát ?

- ☐ (-,b,b*)
- ☐ (*,b?,b*)
- ☒ (a*,b+)
- ☐ ((a,b,b)?)
- ☐ (a,b((c|d),b,b*)
- ☒ (a*,b,b*)
- ☐ (a,b|(c,d),b*)
- ☐ (,b+b)
- ☐ egyik sem

Az alábbiak közül ② helyén melyik válasz írja le helyesen az ELH-ábrát ?

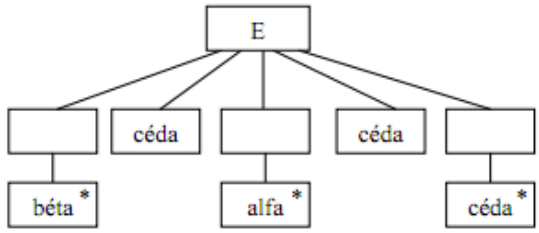
- ☐ (b+(c,d))
- ☐ ((b+(c,d)*),b?)
- ☐ (b*(c|d))
- ☐ (b,(c|d))
- ☒ (b|(c,d))
- ☐ ((b,c)|(b,d)?)
- ☐ ((b|(c,d)*),b+)
- ☐ ((b|(c,d)),b, b*)
- ☐ egyik sem

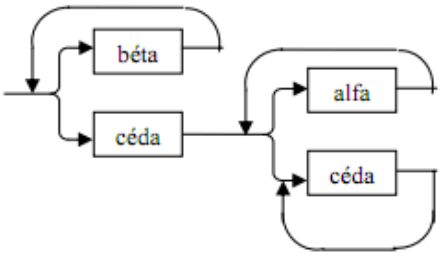
A baloldalon álló XML adatszerkezet

- ☐ mint XML szerkezet szintaktikai hibás
- ☒ megfelel az ELH-ábrának (érvényes)
- ☐ nem felel meg az ELH-ábrának (érvénytelen)
- ☐ érvényessége nem dönthető el

Az E entitás élettörténetét az alábbi állapottáblával írhatjuk le (`betű` a kezdőállapot). Rajzolja fel az élettörténetet JSD ábrán és szintaxis gráffal !

	alfa	béta	céda
betű	--	betű	szám
szám	szám	--	egyéb
egyéb	--	--	egyéb





Megoldás menete:
Játékszabályok:
- A táblázat oszlopában vertikális tagolásban a lehetséges állapotokat látod - betû, szám, egyéb.
- A legfelsõ sor (horizontálisan) a lehetséges, értelmezett bemeneteket reprezentálja - alfa, béta, céda.
- A táblázat további soraiban, oszlopaiban pedig azt látod, hogy ezekre a bemenettípusokra milyen állapotátmenetek figyelhetõk meg a rendszerben. A kihúzott állapotátmenetekkel nem foglalkozunk (don't care).

Betû a kezdõállapotunk. Ha kap egy bétát a rendszer, betû állapotban marad - ha ismét azt kap, akkor is itt marad, így ebbõl látható, hogy tetszõleges mennyiségû bétát beleönthetünk a rendszerünkbe, nem fog állapotot váltani.
JSD-reprezentáció: Ez egy iteráció, mert megtehetjük, hogy soha egy bétát nem küldünk a rendszernek, de ugyanakkor tetszõlegesen sok ilyen bemenet is engedélyezett - *-os doboz (lásd lentebb a JSD ábra elemeinek leírását).

Ha betû állapotban kapunk egy cédát, rögvest szám állapotba kerül a rendszerünk - ez pontosan egyszer történhet meg, utána már más állapotban leszünk.
JSD-reprezencáió: Pontosan egyszer érkezhet meg a bemenet. Szekvencia - sima doboz.

Ha szám állapotban alfa érkezik, akkor nem történik állapotváltás. Iteráció.
Ha jön egy céda, megyünk az egyéb állapotba. Pontosan egyszer - szekvencia.
Az egyéb állapotban az alfa és a béta bemenetekkel nem foglalkozunk, ha céda érkezik, akkor viszont maradunk. Annyi cédát küldhetünk neki, amennyit csak úri kedvünk tartja, nem történik majd állapotátmenet. Iteráció.

A JSD-ábra megkreálásakor itt is figyelni kell arra, hogy azonos szinten csak azonos típusjelzésû objektumaink legyenek. A mintamegoldásban az elsõ emeletet elfoglalták a szekvenciák, az iterációk a másodikra kényszerültek. Hatalmi harc van az SCH-ban. ;)

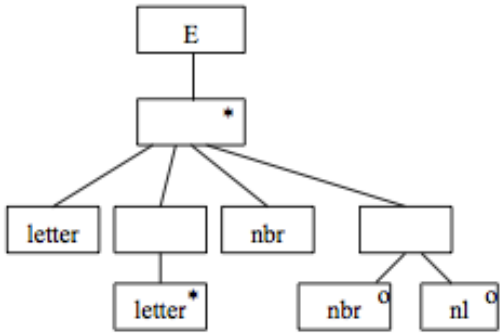
Szintaxisgráf: ez egy olyan gráf, ami a lehetséges bemenetek alapján az egyes állapotok között való lépkedés lezajlásának mikéntjét mutatja meg számunkra.

Elindulunk a kezdõállapotból, és mindig annyi felé ágazunk el, ahányféle esetet megkülönböztetünk a bemenetek alapján. Kezdjük!

- Betû állapotból indulunk, kétféle bemenettel foglalkozunk, így kétfelé ágazunk el: ha bétát kapunk, visszajövünk és kezdjük az egészet újra. Amennyiben céda érkezik, gyaloglunk tovább a következõ állapotba.
- Szám állapotban megint kétféle olyan bemenet van, amivel foglalkozni kívánunk, így ismét kétfelé ágazunk el: alfa esetén bécsi keringõzünk egyet, majd visszaérkezünk, céda esetben következik a következõ állapot.
- Egyéb állapotban csak egy esetet vizsgálunk, és az is visszavezet ugyanebbe az állapotba, így már csak egy tollvonás, és szenvedéseink véget értek.

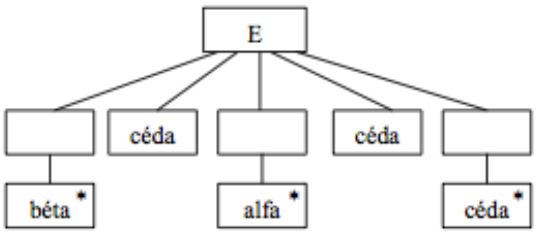
Az E entitás élettörténetét az alábbi állapottáblával írhatjuk le (**ax** a kezdőállapot). Rajzolja fel az élettörténetet JSD ábrán !

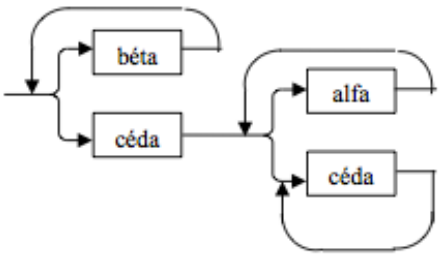
	letter	nbr	nl
ax	by	--	--
by	by	cz	--
cz	--	ax	ax



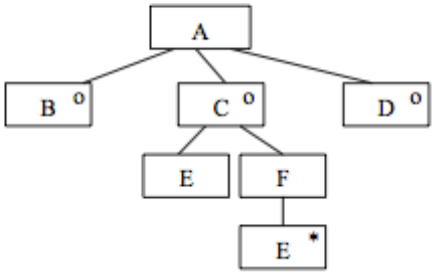
Az E entitás élettörténetét az alábbi állapottáblával írhatjuk le (`betű` a kezdőállapot). Rajzolja fel az élettörténetet JSD ábrán és szintaxis gráffal !

	alfa	béta	céda
betű	--	betű	szám
szám	szám	--	egyéb
egyéb	--	--	egyéb



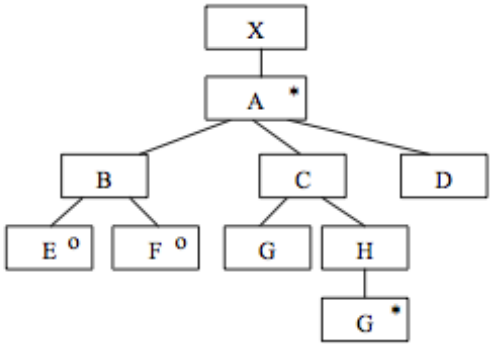


Az alábbi JSD ELH diagram alapján készítsen állapottáblát ! Az állapotokat a bevezetett jelölésrendszer szerint, számokkal jelölje ! Az ① legyen az induló állapot !



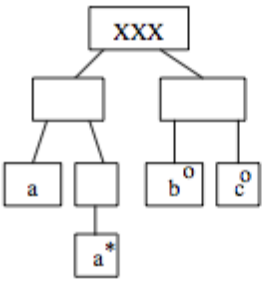
	B	D	E		
①	③	③	②		
②			②		
③					

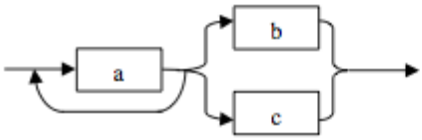
Az alábbi JSD ELH diagram alapján készítsen állapottáblát ! Az állapotokat a bevezetett jelölésrendszer szerint, - az előfordulás ideje szerint növekvő - számokkal jelölje ! Az ① legyen az induló állapot !



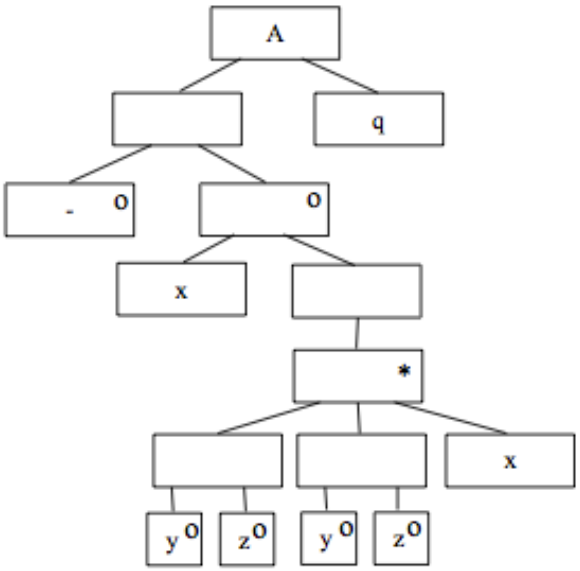
	D	E	F	G	
①		②	②		
②				③	
③	①			③	

Az XXX entitás élettörténetét az alábbi JSD ábra definiálja. Írja le ugyanezen élettörténetet szintaxis gráffal ! (Az élettörténetben szereplő eseménynek feleljen meg az azonos nevű szimbólum !)

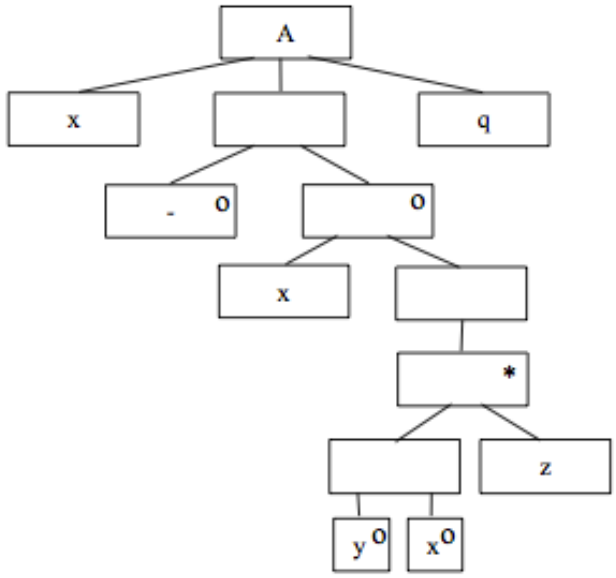




Legyen egy A entitás, amelyen az x, y, z és q események fordulhatnak elő. Rajzolja fel az A entitás élettörténetét a JSD szerint, ha az események sorrendjét az alábbi BNF leírással definiáltuk !
[x{ [y | z] [y | z] x}] q

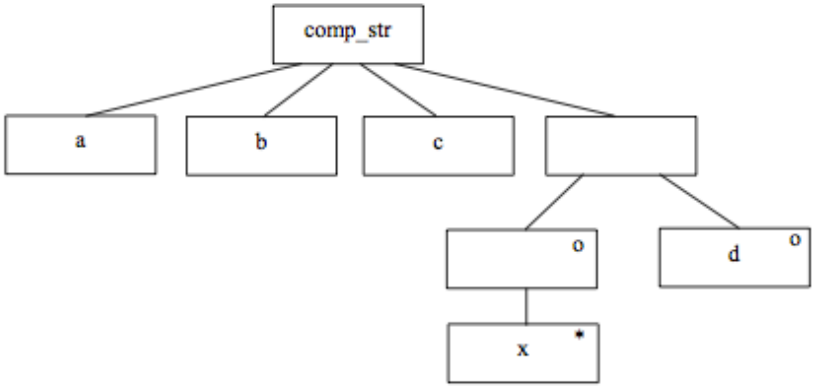


Legyen egy A entitás, amelyen az x, y, z és q események fordulhatnak elő. Rajzolja fel az A entitás élettörténetét a JSD szerint, ha az események sorrendjét az alábbi BNF leírással definiáltuk !
x[x{ [y | x] z}] q



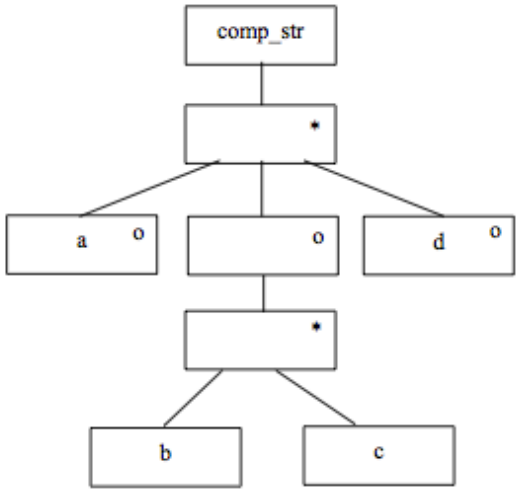
Írja fel az alább algebrailag leírt `comp_str` nevű összetett adatstruktúrát a JSD-ben alkalmazott jelöléstechnikával ! Egy z attribútum egyszeri előfordulását jelölje a z esemény !

```
comp_str = a + b + c + [ { x } | d ]
```



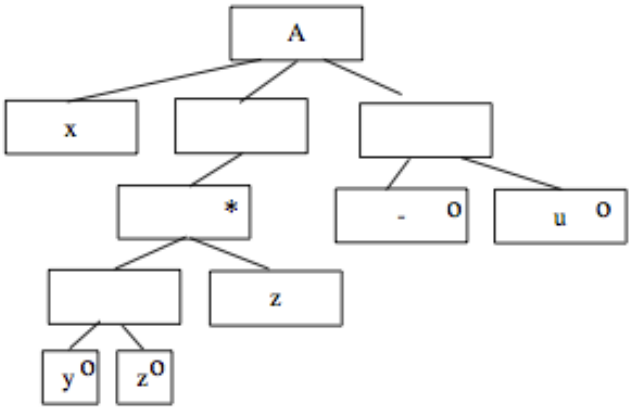
Írja fel az alább algebrailag leírt `comp_str` nevű összetett adatstruktúrát a JSD-ben alkalmazott jelöléstechnikával ! Egy z attribútum egyszeri előfordulását jelölje a z esemény !

```
comp_str = {[ a | { b + c } | d ]}
```



Legyen egy **A** entitás, amelyen az **x**, **y**, **z** és **u** események fordulhatnak elő. Rajzolja fel az **A** entitás élettörténetét a JSD szerint, ha az események sorrendjét az alábbi BNF leírással definiáltuk !

```
x{[y|z]z}[u]
```



Legyen egy **A** entitás, amelyen az **x**, **y**, **z** és u események fordulhatnak elő. Rajzolja fel az **A** entitás élettörténetét a JSD szerint, ha az események sorrendjét az alábbi BNF leírással definiáltuk !

x[u]{[y|z]z}