5. rész

OO
Interfész
osztály
Liskov-féle helyettesítési törvény
Demeter törvénye

OO

OO paradigmák

Paradigmának nevezzük azt a világszemléletet, látás- és gondolkodásmódot, amelyet az adott fogalomkörben használt elméletek, modellek és módszerek összessége jellemez.

- valódi világbeli szervezésnek megfelelően szeretnénk gondolkodni a problémákról
- alapkonstrukció az objektum, amely egy entitásban kombinálja az adatot és viselkedést
- modellünket, szoftverünket kooperatív, diszkrét és együttműködő objektumok kollekciójaként szervezzük
- objektum orientált metódusokat használunk a szoftver fejlesztéséhez, és objektum orientált eszközöket

Motivációk

- újrafelhasználhatóság, könnyű változtathatóság
- a problématér és az implementáció szerkezete minél inkább hasonlítson egymásra

Objektum szemantika

- egy objektumrendszer szolgáltatásokat nyújt a kliensnek
- egy szolgáltatás kliense bármely entitás, amely szolgáltatást képes kérni

Objektumok

- egy objektumrendszer objektumokként ismert entitásokat tartalmaz
- az objektum identifikálható egységbezárt entitások, amely egy vagy több szolgáltatást képe nyújtani (felelőség)

Objektum

- az alapszabálya a rendszer dekompozíciójának a felelőségek (responsibility; pl. szerep, képesség) komponensekhez (objektum) rendelése

Kérések

- kliensek kéréseket nyújtanak be
- kérés = tevékenyég sorozat (kauzális) a kliens inicializációjától az utolsó eseményig, amely kauzális kapcsolatban van az inicializációval
- kérésforma (**request form**) = egy leírás vagy minta, amely kiértékelődhet/végrehajtódhat többszörösen, hogy előidézze a kibocsátását igényét.

- érték (**value**) = minden, ami egy kérés legitim paramétere lehet
- objektum referencia (**object reference**) egy érték, amely megbízhatóan rámutat/megjelöl egy egyéni objektumot. Egy objektumot megjelölhet több különböző objektum referencia.
- kérés benyújtása a szolgáltatás végrehajtódását eredményezi. A végrehajtás egyik lehetséges eredménye az eredmény visszajuttatása a klienshez.
- Ha abnormális jelenség következik be a végrehajtás során, akkor kivétel (**exception**) kap vissza a kliens.

Objektum létrehozás és megsemmisítés

Objektumot létre lehet hozni és megsemmisíteni. A kliens szemszögéből nincs speciális mechanizmusa az objektum létrehozásának és megsemmisítésének. Objektumok kérésbenyújtás eredményeképp jönnek létre és semmisülnek meg. Az objektum létrehozásának eredménye nyilvánosságra kerül a kliens előtt objektum referencia formájában, amely rámutat az objektumra.

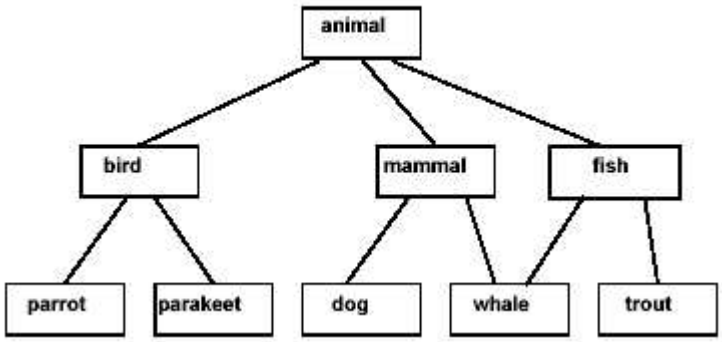
Típusok

- a típus egy azonosítható entitás, amelyhez az entitások fölött értelmezett predikátum van rendelve (egyváltozós matematikai függvény igaz-hamis eredménnyel). Egy entitás kielégíti a típust, ha a predikátum igaz rá. Az entitás, amely kielégíti a típust, a típus tagjának hívjuk (**member of type**).
- Típusokat aláírásokban használatosak, hogy korlátozzák a lehetséges paramétereket vagy hogy karakterizálják a lehetséges eredményt
- egy típus kibővítése (**extension of type**) azon entitások halmaza, amelyek kielégítik a típust minden konkrét időpontban.
- az objektum típus egy típus, amely tagjai objektum referenciák. Más szavakkal, objektum típust csak az objektum referenciák elégítik ki.

Interfész (Interface)

- interfész a leírása azoknak a lehetséges operációknak, amelyeket a kliens kérhet az objektumtól, azon az interfészen keresztül.
- Egy objektum kielégíti az interfészt, ha nyújtani tudja annak szolgáltatásait az interfész operációin keresztül, amelyek a szolgáltatások specifikációjához tartoznak
- Az interfész típusa (**interface type**) egy adott interfésznek egy objektum típus, úgy, hogy az objektum referencia kielégíti a típust, ha a megjelölt objektum kielégíti az interfészt
- Az interfészek kielégítik a **Liskov-féle helyettesítési törvényt** (interfész típusok kompatibilitása):
    - ***Ha az A interfész a B interfészből van származtatva, akkor egy objektum referencia, amely nyújtani tudja az A interfészt, használható ott, ahol a formális típusa a paraméternek B-ként van deklarálva.***

- A – **subtype**; B – **supertype**
- A *kompatibilis* B-*vel reláció*: **A is_a B**
- Az **is_a** reláció **reflexív**, **nem szimmetrikus** és **tranzitív**



Operációk

- az operáció egy identifikálható entitás, amely megjelöl egy oszthatatlan primitívjét a szolgáltatásnak, amelyet lehet kérni. Az operációkérés aktusa az operáció lehívása (**invoking the operation**). Az operációt az operáció azonosító (**operation identifier**) azonosítja.
- Az operáció szignatúra alapformája:
  `[oneway] <op_type_spec> <identifier> (param1, ...,paramL) [raises(except1,...,exceptN)] [context(name1,..., nameM)]`

Paraméterek

- a paramétert a típusa és módja karakterizálja. A mód azt jelzi, hogy a paramétert a kliens küldi a szervernek (**in**), a szerver a kliensnek (**out**), vagy mindkettő (**inout**). A paraméter típusa megszabja a lehetséges értéket, amely átadható a módban meghatározott irányban.

Visszatérés eredménye (return result)

- a visszatérés eredménye egy kitüntetett out paraméter

Futtatási szemantika (execution semantics)

- operációhoz társul
  - **At-most-once** – ha egy operációkérés sikeresen tér vissza, akkor pontosan egyszer volt végrehajtva. Ha egy kivétel figyelmeztetéssel tér vissza, akkor at-most-once-szor volt végrehajtva.
  - **Best-effort** – best-effort operáció egy csak-kérés operáció (nem ad vissza eredményt, a kérést sosem lehet szinkronizálni a kérés végrehajtásának befejezésével)

Objektum megvalósítás/implementáció

- Egy objektumrendszer inplementációja eleget tesz a számítási tevékenységeknek, amelyek ahhoz kellenek, hogy kívánt szolgáltatások működését eredményezze
- Az implementációs modell két részből áll:
  - A futtatási modell leírja a szolgáltatás végrehajtásának hogyanját
  - A konstrukciós modell leírja, hogyan van a szolgáltatás definiálva

Futtatási modell (execution model)

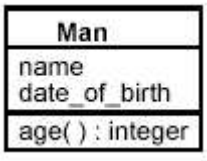
- Az igényelt szolgáltatás, amely végrehajtódik egy számítógépes rendszeren egy kód futtatásával, amely az adatokkal operál. Az adat a számítógépes rendszer egy komponensének az állapotát reprezentálja. A kód végrehajtja az igényelt szolgáltatást, amely meg tudja változtatni a rendszer állapotát.
- a kódot, amely teljesít egy szolgáltatás **metódus**nak (***method***) hívjuk. A metódus egy állandó (megváltozhatatlan) leírása egy számításnak, amely interpretálható egy futtató gép által
- egy metódus futtatását metódus aktiválásának (***method activation***) nevezzük

Konstrukciós modell

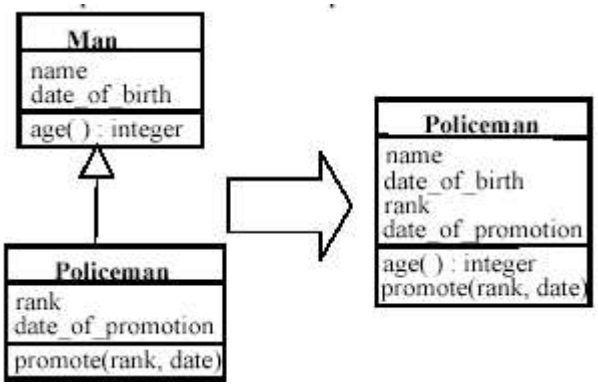
- egy számítógépes objektum-rendszernek mechanizmusokat kell nyújtania a igény viselkedésének realizációjára. Ezek a mechanizmusok tartalmaznak: objektum állapot definíciókat, metódusok definícióját és az objektum felépítésének definícióját. Ezek megjelölik a metódusokat amelyek végrehajtódnak és megjelölik az objektum állapotának fontos részeit, amelyek elérhetőek lesznek a metódusok számára.
- mechanizmusnak kell nyújtva lennie azon konkrét akciók leírásához, amelyek az objektum létrehozásával vannak kapcsolatban.
- Egy objektum implementáció egy definíció, mely magában foglalja az információt, amely szükséges az objektum létrehozásához és lehetővé teszi az objektumnak megfelelő szolgáltatások halmazának nyújtásához való hozzájárulását.
- **Viselkedés**: Az objektum egység definiált viselkedéssel. Minden objektum felelős saját magáért. Egy objektum az, amit csinál. Egy objektum az általa végrehajtani tudott operációk által definiálja magát. Az operációk metódusokként vannak megvalósítva.
- **Struktúra**: Minden objektumnak van belső struktúrája. Az objektumok elrejtik belsejüket, láthatatlan az a kliensei számára.
- **Állapotok**: Az objektum állapota a változói aktuális értékeitől függ.
- **Identitás**: Minden objektum egyedileg azonosítható.

Osztály (Class)

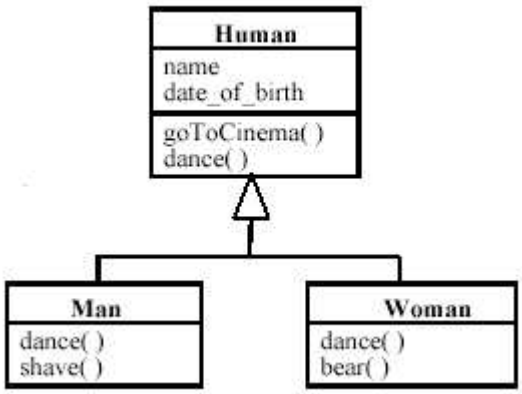
- Az osztály az objektumok egy csoportja, amelyek azonos tulajdonságokkal rendelkeznek.



- Új osztály definiálható már létező osztályok szempontjából az **öröklődés** (**inheritance**) segítségével. Az új osztály tartalmazza mindazon adatok és metódusok definícióit, amelyeket a szülő osztály definiált.

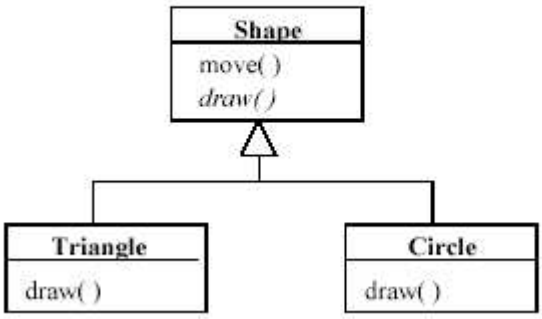


- Az **származtatott osztály**ok (**subclasses**) finomíthatják (**refine**) vagy felüldefiniálhatják (**redefine**) a szülő osztály viselkedését. Az osztály felülírhatják a szülőjük operációit. A felülírás lehetőséget nyújt arra, hogy a kérelmet ők kezeljék, ne pedig szülőjük.



- **Absztrakt operáció** (**abstract operation**):

```
move: draw (background_color, old_position)
      draw (ink_color, new_position)
```



Tervezési szabályok

Tervezés szerződés (contract) segítségével

- Bertrand Meyer
- Objektumok együttműködésének precíz specifikációkon – szerződéseken -- kellene nyugodnia, amelyek leírják minden oldal kilátásait és biztosítékait

provide_service	**Kötelezettség**	**Előny (benefit)**
(Kliens) Háztulajdonos	(teljesíti a prekondíciót) számla befizetése	(a posztkondícióból) telefonálhat
(Szollgáltató) Cég	(teljesíti a posztkondíciót) telefonszolgáltatást nyújt	(a prekondícióból) nem kell semmit sem szolgáltatnia, ha a számla nincs megfizetve

- **prekondíció** (**precondition**) – a kliens kötelessége, amely a szolgáltatót védi. Azt állapítja meg, hogy mit kell a kliensnek teljesítenie mielőtt kérne egy meghatározott szolgáltatást
- **posztkondíció** (**postcondition**) – a kliens haszna, ami leírja, hogy mit kell a szolgáltatónak tennie (feltéve, hogy a prekondíció teljesítve van)
- **osztály-invariánsok** (**class invariants**) – teljesülnie kell a szolgáltatás lezajlása alatt
- prekondíció megszegése a kliens hibáját jelzi, aki nem tartotta meg az ő részét az alkunak
- a posztkondíció (vagy invariáns) megszegése a szolgáltató hibáját jelzi (nem csinálta meg a munkáját)

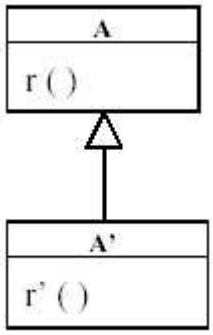
Öröklődés

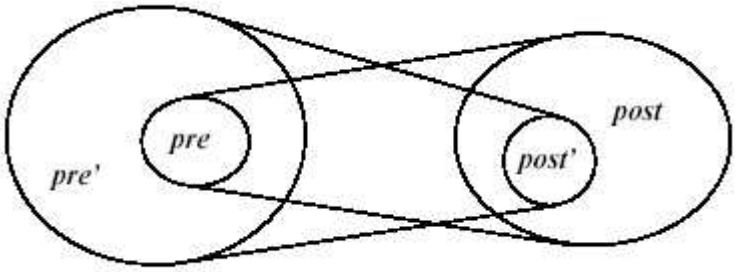
- Egy objektumnak többfajta típusa lehet, és különböző osztályokba tartozó objektumoknak is lehet azonos típusa.
- **Osztályöröklődés** (**class inheritance**)
- **Interfész-öröklődés** (**interface inheritance**) leírja, hogy mikor lehet használni egy objektumot egy másik objektum helyén.
- **Liskov-féle helyettesítési szabály**
    - **Barbara Liskov:** ***Ha minden egyes o1 S típusú objektumhoz létezik olyan o2 T típusú objektum, minden P programban T hatókörében P viselkedése változatla, ha o1-et helyettesítjük o2 helyére, akkor S származtatott típusa (subtype) T-nek.***
- **A nyitott-zárt szabály (open-close principle)**
    - **Bertrand Meyer:** ***Az osztálynak nyitottnak kell lennie a kiterjesztésre, de zártnak a változtatásra nézve.***

Öröklődés és szerződés

- Def: A p állítás erősebb vagy egyenlő q állítással, ha p-ből következik q.

- **Invariáns felhalmozódási szabály** (**invariant accumulation rule**): A szülők invariánsai vonatkoznak az osztályra magára. Inv' erősebb vagy egyenlő Inv-vel.
- **Állítás újradeklarációs szabály** (**assertion redeclaration rule**): Vezessünk be új kondíciókat, &-sel az eredeti prekondícióból, és |-gyal az eredeti posztkondícióból.
    - pre'-nek gyengébbnek vagy egyenlőnek kell lennie pre-vel
    - post'-nak erősebbnek vagy egyenlőnek kell lennie post-tal





Objektumváltozó

- **változó típusa**: Milyen fajta objektum tárolható a változóban?
    - **Statikusan tipizált** (**static typing**) – az akceptálható objektumok osztálya már fordításkor ismert. A statikusan tipizált változók használatának előnye, hogy a programozási nyelvekben egyszerűen és hatékonyan implementálhatók a változókat kezelő műveletek.
    - **Dinamikusan tipizált** (**dynamic typing**) – a változóknak nincs előre meghatározott típusa, futás közben tetszőleges értéket felvehet. Minden osztály objektumait akceptálja.
- **kötés**: Mi határozza meg a változón végrehajtható műveleteket?
    - **Statikus kötés** (**static binding**) – a műveletek a változóhoz vannak kötve. A változón értelmezett műveleteket a változó típusa határozza meg.
    - **Dinamikus kötés** (**dynamic binding**) – a műveletet meghatározó tényező a változó által hordozott érték.

Statikus tipizálás - statikus kötés

Megfelel a hagyományos programnyelvekben implementált megoldásnak. A változónak típusa van és ez a típus meghatározza mind a változó értékkészletét, mind pedig a változón végrehajtható műveletet.

Dinamikus tipizálás - statikus kötés

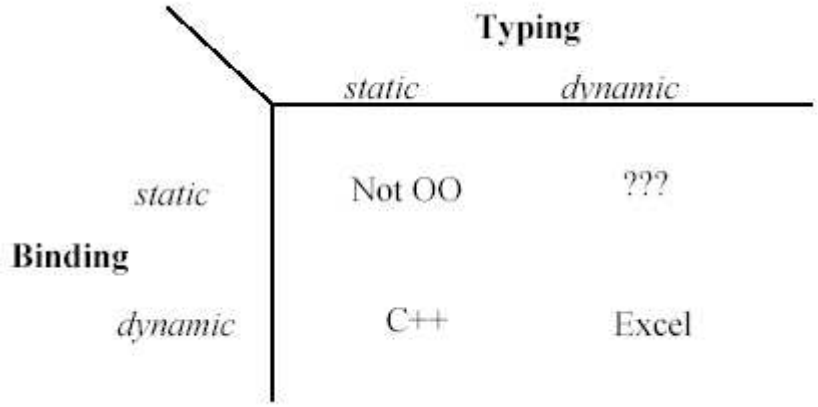
Azzal a meglehetősen furcsa helyzettel kerülünk szembe, hogy a változóba tetszőleges objektumot tehetünk, ugyanakkor a végrehajtható műveleteket a változó nem létező típusa határozza meg.

Dinamikus tipizálás - dinamikus kötés

A változó tetszőleges értéket felvehet és a végrehajtható műveleteket az érték határozza meg.

Statikus tipizálás - dinamikus kötés

Látszólag nincs sok értelme annak, hogy egy változóba elhelyezhető értékek típusát megkössük, ugyanakkor a végrehajtható műveleteket az értékekhez kössük. Ha szigorúan betartjuk a tipizálást, akkor ez így van. Ha azonban felhasználjuk a típusok kompatibilitásával kapcsolatos korábbi eredményeinket és a kompatibilitási reláció mentén felpuhítjuk a kemény típuskorlátozást, kijelentve, hogy egy adott típusú változó a vele kompatíbilis típusok értékeit is felveheti, akkor már nagy jelentősége lehet ennek az esetnek.



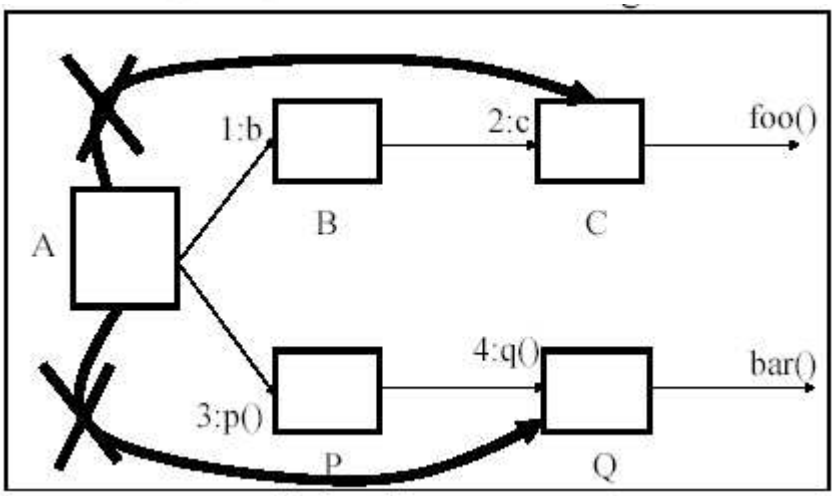
Demeter törvénye (**the law of Demeter, LoD**)

- a Demeter Research Group javasolta 1987-ben
  - Minden egységnek más egységek csak egy véges halmazát kellene használnia: csak olyan egységeket, amelyekkel közeli kapcsolatban áll (közeli rokonok, closely related)
  - Minden egységnek csak a barátaival szabadna beszélnie. Ne beszélj idegenekkel!
- alkalmazása OO-ban
  - egység = M metódusa a C osztálynak
  - közeli rokonok:
    - az M metódus paraméterei
    - saját osztály, ősei
    - a C osztály attribútumai
    - M által kreált átmeneti változók
    - globális változók
    - M által kreált objektumok

Demeter-törvény megsértése

```
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();}
```

A eléri C-t



Kohézió

Általánosságban az OOP nagy kohézióhoz vezet, de

- **véletlenszerű** (**coincidental**) – rossz eset, mely akkor történik meg ez, ha osztályok valódi attribútumok nélkül keletkeznek – ritka
- **logikai** (**logical**) – olyan metódusok vannak, amelyek egy feltételbit szerint ezt vagy azt hajtanak végre, pl. ScaleOrRotate(double sf, double ra, bool flag); - ritka
- **időbeli** (**temporal**) – konstruktor – ritka
- **eljárásos** (**procedural**) –
- **kommunikációs** (**communicational**) – elfogadható
- **szekvenciális** (**sequential**) – elfogadható
- **funkcionális** (**functional**) – ezt szeretnénk elérni

Operációk kohéziójának megállapítása

- operáció neve alapján:
  - "**and**"-del a névben – többfajta viselkedés – valószínűleg szekvenciális vagy kommunikációs kohézió (ScaleAndRotate)
  - "**or**"-ral a névben – logikai kohézió (ScaleOrRotate)

- egyébbként, “szilárd” névvel – funkcionális kohézió (doSomeOperation)

- operáció kohéziójának becslése a forráskód alapján nem nyilvánvaló

Osztályok kohéziójának mérése

- az osztályban lévő összes operációjának lehet jó a kohéziója, de ez nem garantálja, hogy az osztály kohéziója jó lesz
- Alapötlet: elemek közötti rokonsági kapcsolat tanulmányozása. Opráción belül meg tudjuk becsülni a kontrollmechanizmust, időbeli megszorítást, folyamatlogikát vagy adatot. DE, különböző operációk között amit mi könnyen ki tudunk számolni, az az operációk közötti hasonlóság az attribútumaik szemszögéből.
- Minél nagyobb a hasonlóság, annál nagyobb az osztály kohéziója

Kohézióhiány metrika (**lack of cohesion metric, LCOM**)

- Legyen p(X) az X tulajdonsága , ekkor $\sigma(X,Y)=p(x)\cap p(Y)$ tekinthetjük az X és Y hasonlóságának a p() tulajdonsága szempontjából
- Az osztály kohéziójához legyenek az O operáció attribútumai az O operáció tulajdonsága
- A C osztálynak n operációja van ($O_1,O_2,\ldots,O_n$), és attribútumai $a_1,\ldots$
- Minden O_j operáció az A_j halmazban attribútumokat használja
- Legyen $\mathbf{P=\{(A_i,A_j) \mid Ai\cap Aj=0\}}$ és Legyen $\mathbf{Q=\{(A_i,A_j) \mid A_i\cap A_j\neq 0\}}$
- $$LCOM = \begin{matrix} |P|-|Q| & ha & |P|>|Q| \\ 0 & egyebbkent & \end{matrix}$$
- például:
  - $A_1=\{a,b,c,d,e\}$; $A_2=\{a,b,e\}$; $A_3=\{x,y,z\}$
  - $A1\cap A2\neq 0$; $A_1\cap A_3=A_2\cap A_3=0$
  - $|P|=2$; $|Q|=1 \Rightarrow$ LCOM=2-1=1

Objektumorientált metrikák

Source	Metric	Const
Trad	Cycl. comp. (CC)	method
Trad	Lines of code (LOC)	method
Trad	Comment perc. (CP)	method
OO	Weighted methods per class (WMC)	method
OO	Response for a class(RFC)	class/msg
OO	Lack of cohesion of methods (LCOM)	class/coh
OO	Coupling between objects (CBO)	coupling
OO	Depth of inheritance tree (DIT)	inherit
OO	Number of children (NOC)	inherit

Ciklomatikus komplexitás (**cyclomatic complexity, CC**)
- a metódusban használt algoritmus komplexitásának megbecsülésére szolgál
- a metódus átfogó teszteléséhez szükséges tesztelések száma
- **CC = #él - #pont + 2**
- Kisebb CC általában jobb

Méret
- LOC – minden sor száma
- NCNB – sorok száma az üres sorok és kommentek nélkül
- EXEC – futtatható utasítások

Komment-százalék (**comment percentage, CP**)
- CP = a kommentek száma (LOC – üres sorok(BL))

Súlyozott metódusok osztályonként (**weighted methods per class, WMC**)
- osztályban implementált metódusok száma
- metódusok komplexitásánakaz összege
  - idő és erőfeszítés becsült értéke, amely az osztály fejlesztéséhez szükséges
  - nagy #metódus → nagyobb applikáció specifikáció

Feleletek száma az osztálynak (**response for a class, RFC**)
- minél nagyobb az RFC, annál nagyobb az osztály komplexitása
- metódusok száma, melyek elérhetők az osztályhierarchián belülről

Kohézióhiány metrika (**lack of cohesion metric, LCOM**)
- metódusok eltérését méri az attribűtumok függvényében
- nagy LCOM nagy komplexitást jelent, a fejlesztés során valószínűleg hibát követtünk el

Csatolás az objektum-osztályok között (**coupling between object classes**)

- azon osztályok száma, amelyektől az osztály függ, de nem öröklődés szintjén

Öröklődési fa mélysége (**depth of inheritance tree**)

Gyermekek száma (**number of children**)

- közvetlen származtatott osztályok száma

6, rész
UML

UML áttekintése

A UML (Unified Modeling Language) magyarul annyit jelent, mint egységesített modellező nyelv. Azért *nyelv*, mert egy nyelvezetet biztosít vizuális dokumentáció készítéséhez. Az általa definiált elemek, kapcsolatok, diagramok, stb. bizonyos szintaktikai és szemantikai szabályok összességének felelnek meg, mint minden más nyelv esetében is. Egy vizuális nyelv esetében a szintaktika a grafikus kinézetet jelenti, a szemantika meg a megjelenített elemeknek értelmezést ad és további szemantikai szabályokat definiál. Az UML egységesített, mert számos korábban meglévő nyelv ötvözésével keletkezett, azokból kiragadva a legjobb tulajdonságokat. A UML egyik legfontosabb jó tulajdonsága éppen ez, ugyanis hatalmas tapasztalati tudásra épít. Ezáltal nyugodtak lehetünk afelől, hogy a nyelvalkalmazásával csakis nyerhetünk. Tehát a UML nem egy cég által kifejlesztett és jól marketingelt termék, hanem egy megbízható, és jól bevált módszer.

UML története. Számos módszertan és jelölés járult hozzá a UML létrejöttéhez, de alapvetően az OMT, Booch és OOSE módszerek egyesítésével jött létre 1995 októberében. Ezekből elsődlegesen a szemantikus modellek (mi az amit ábrázolni kell), a szintaktikus jelölések (grafikus megjelenés) és nyilván a diagramok lettek „unifikálva". Az első szabványosított specifikációt (1.0) 1997 júliusában adta ki az OMG (Object Management Group

UML metamodell architektúra. A specifikáció egy négyszintű metamodell szerkezet segítségével definiálja a nyelvet. E rendszer lehetővé teszi a lehető legáltalánosabb leírást és a bővíthet_séget is precízen definiálja. Azért előnyös a használata, mert mindegyik szinten maga a UML van használva a leírásra. A legfelső szint a *meta-metamodell*, amely azt írja le, hogyan kell a metamodell szinten definiálni a UML nyelvet. A második szint a *metamodell*, amely magát a UML nyelv szintaxisát írja le. Ez alatt helyezkedik el a felhasználó analízis szintű modellje a konkrét modellezési elemekkel. Ez alatt még elhelyezkedhet az objektumok rétege, ahol az analízis szintű modell osztályainak példányai szerepelnek. A specifikáció szempontjából a legfontosabb a metamodell, amely megfelelően csoportosítva részletesen leírja, hogy hogyan kell hogy kinézzenek az egyes UML diagramok.

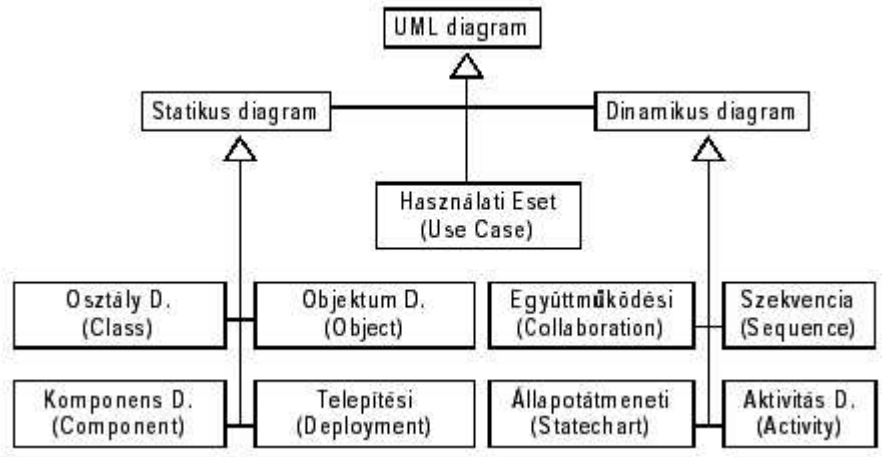
A metamodell az alábbi építőkövek segítségével írja le a nyelvet. Minden használatos diagramfajtánál ezeknek valamilyen kombinációját alkalmazzák:

- **Modellezési elemek.** A modellezési elemek azok az entitások, amelyek valamilyen részét modellezik a rendszernek. A következő alapvető modellezési elemek léteznek:
    - *szerkezeti elemek:* class, interface, collaboration, use case, component, node
    - *viselkedési elemek:* interaction, state machine
    - *csoportosítási elemek:* package, subsystem
    - *egyéb:* note

- **Kapcsolatok (relationships).** A modellezési elemeket, mint csomópontokat összekötő élek. A kapcsolatok fajtái:
  - függőségi kapcsolat (dependency)
  - asszociációs kapcsolat (association)
  - általánosítás, öröklődés (generalization)
  - megvalósítás (realization)

- **Bővíthetőség (Extensibility Mechanisms).** A UML nyelv bővítésére nyújtott támogatások:
  - sztereótípus (stereotype)
  - címke-érték (tagged value)
  - megszorítás (constraint)

- **Diagramok.** 9 diagramfajta van definiálva:
  - **Használati eset** (Use Case D.). Funkcionalitás felhasználó szemszögéből.
  - **Osztálydiagram** (Class D.). A rendszer „szótára”: osztályok és kapcsolataik.
  - **Objektum diagram** (Object D.). Osztály példányok és kapcsolataik.
  - **Komponensdiagram** (Component D.). Implementáció fizikai szerkezete.
  - **Telepítési diagram** (Deployment D.). A rendszer hardver topológiája.
  - **Szekvenciadiagram** (Sequence D.). Dinamikus viselkedés (idő-orientált).
  - **Együttműködési diagram** (Collaboration D.). Dinamikus viselkedés (üzenet-orientált).
  - **Állapotátmeneti diagram** (Statechart). Dinamikus viselkedés egy objektumra (esemény-orientált).
  - **Aktivitás diagram** (Activity D.). Dinamikus viselkedés egy objektumra (aktivitás-orientált).

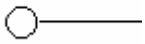
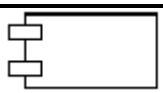
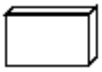
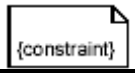


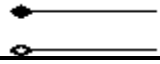
A fenti építőkövek segítségével nagyon sok szempontból leírható a modellezni kívánt rendszer. Általában három csoportra osztják a modellezés mibenlétét (és ezáltal a használt elemeket és diagramokat):

- **Szerkezeti modellezés**. Ezt a fajta modellezést a rendszer szerkezetének leírásánál használják. Jellemző diagramok az osztály-, objektum-, komponens- és telepítési diagram.
- **Használati eset modellezés**. A rendszer viselkedésének leírására alkalmas ahogyan az kívülről látszik. Használati diagram használatos.
- **Viselkedési modellezés**. A rendszer (futás közbeni) viselkedésének leírásakor modellezünk ily módon.

Szerkezeti modellezés

A szerkezeti modellek alapvető elemei a következők:

Elem	***Leírás***	***Szintaxis***
Osztály (*class*)	azonos tulajdonságú objektumok halmazának leírása	
Interfész (*interface*)	operációk nevesített halmaza,amely egy viselkedést ír le «interface»	 vagy 
Komponens (*component*)	moduláris és cserélhető egysége a rendszernek, amely interfészeket implementál	
Csomópont (*node*)	futás közbeni fizikai feldolgozó egység	
Megszorítás (*constraints*)	szemantikus feltétel vagy megszorítás	

Elem	Leírás	Szintaxis
Asszociáció (association)	kettő vagy több osztályozási elem közötti szemantikus kapcsolat, amely a létrejövő objektumok között fogösszeköttetést eredményezni	
Aggregáció (aggregation)	speciális asszociáció, amely rész-egész kapcsolatot ír le	
Általánosítás (generalization)	osztályozó kapcsolat egy általánosabb és egy speciálisabb elem között	
Függőség (dependency)	két elem közötti kapcsolat, amelyeknél az egyik változása befolyásolja a másikat	
Megvalósítás (realization)	egy specifikáció és annak megvalósítása közötti kapcsolat (pl. osztály és interface)	

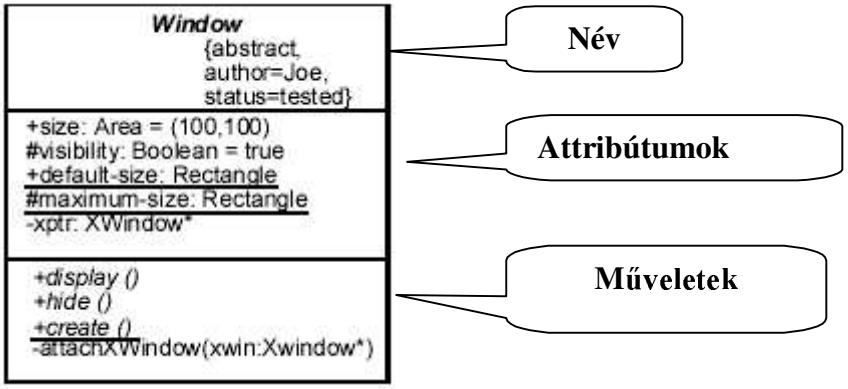
A fenti elemek és kapcsolatok felhasználásával a következő szerkezeti diagramokat definiálja a UML:

- **Statikus szerkezeti diagramok** (logikai, elvi): ***osztálydiagram*** (class diagram) és ***objektum diagram*** (object diagram).

- **Implementációs diagramok** (fizikai): ***komponens diagram*** (component diagram) és ***telepítési diagram*** (deployment diagram).

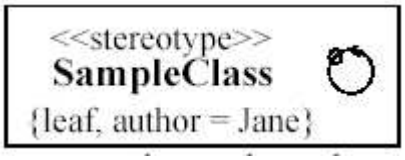
Osztály (CLASS)

Egyforma struktúrájú, viselkedésű és azonos kapcsolatokkal rendelkező objektumok halmazának leírója. Azon a csomagon (package) belül van a nevének a hatóköre, amelyen belül deklaráltuk.



Névrekesz:

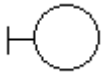
- az absztrakt osztályok neveit dőlt betűkkel írjuk



- osztályok esetén használt sztereotípiák:

«Entity». Entitás osztályok modellezésére, általában hosszú életű információt, viselkedést tartalmaz. Ezek az osztályok a valós világ entitásai, amelyek kevésbé érzékenyek a környezetük változásaira.

«Boundary» (határ). Ezen osztályok a rendszer környezete és belseje közötti kommunikációt valósítják meg és valójában interfészt képeznek a felhasználó vagy más rendszer (szereplő) felé.



«Control». A *control* (vezérlő) osztályok a használati esetek szekvenciális viselkedését valósítják meg, azaz a használati eset „végrehajtását”.

«Utility».Olyan osztályt specifikál, amely szolgáltatásai minden osztályra kiterjednek. (pl matematikai csomagok)

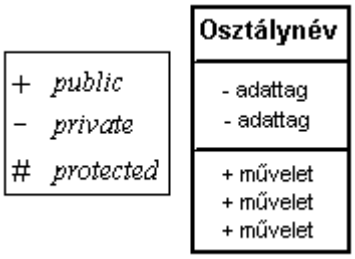
Attribútumok:

A középső rekeszben kell megadni a tulajdonságok nevét. A tulajdonságokhoz a néven kívül típus, kezdőérték és a láthatóságot jelző információ rendelhető a következő szintaxis szerint:

<láthatóság> [multiplicitás] <név> : <típus> = <kezdőérték> {tulajdonságsztring}

A láthatóságnak (**visibility**) három fajtája van:

- Csak class-on belül (**private**)
- Bárhonnan látható (**public**)
- Public örökléssel is private marad (**protected**)



Az osztály szintű attribútumokat aláhúzás jelzi.

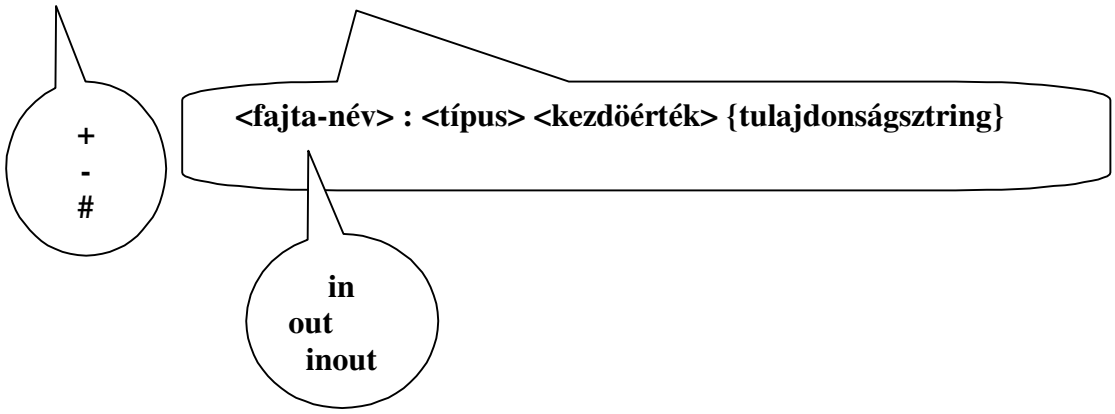
Tulajdonságsztring lehet:

- **changeable** – nincs megkötés
- **addOnly** – ha a multiplicitás >1 egyszel keletkezik, majd az értéke nem változhat meg
- **frozen** – az objektum inicializálása után nem változhat

Műveletek:

Egy szolgáltatás, mely az osztály egy példányától kérhető a végrehajtása végett. Szintaxisa:

<láthatóság> <név>(<paraméterlista>):<vissza-érték-típusa> {tulajdonságsztring}



Ostályszintű műveleteket szintén aláhúzás jelzi.

Tulajdonságsztring (**property string**) lehet:

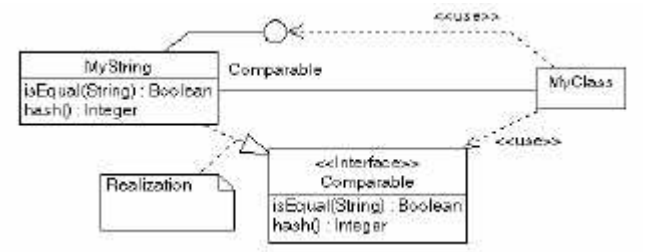
- **isQuery** – nem változtatja meg az állapotot
- **sequential** – a hívónak kell kintről úgy koordinálnia az objektumokat, hogy egyszerre csak egy folyamat van az objektumban
- **guarded** – többszörös hívás konkurens szálakból lehetnek, de csak egy kezdődhet el
- **concurrent** – többszörös hívás konkurens szálakból lehet ilyen műveletekhez, és azok konkurens módón fognak végrehajtódni korrekt szemantikával

Osztályszerű elem (CLASSIFIER)

- egy elem, mely viselkedési és strukturális sajátságokat ír le
- classifierek: osztály, interfész, szignál, adattípus, komponens, use case, alrendszer, DE asszociáció és üzenet NEM

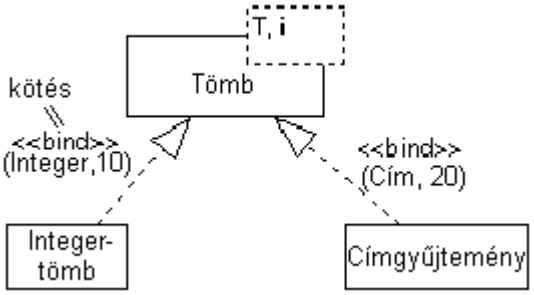
Interfész (INTEFACE)

Meghatározza a kívülről látható operációit az osztálynak. Komponens belső struktúra nélkül.



Paraméterezett osztály (PARAMETERIZED CLASS, TEMPLATE)

Egy olyan osztály leírója, mely tartalmaz egy vagy több nem meghatározott formális paramétert. Osztályok egy családját definiálja.

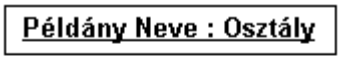


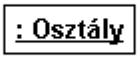
Az ábrán a Tömb egy sablon (Template), ami több különbözõ osztályra is alkalmazható, és olyan funkciókat valósít meg, amely a hozzá kapcsolódó osztályokat általánosságban kezelni tudja.

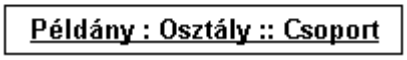
A szaggatott vonalas téglalap azt jelzi, hogy az adott sablon vagy osztály milyen adatszerkezeteket kezel. (pl. Integerekbõl álló verem: osztály = verem, kezelt adatszerkezet = integer).

Objektum (OBJECT)

Osztály konkrét példánya.

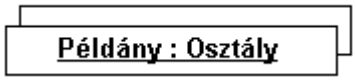






Létrehozhatunk név nélkül objektumokat, elláthatjuk névvel, sõt, elõfordulhat, hogy az osztály egy csoport tagja. Fontos, hogy a objektum nevét alá kell húzni.

A példányoknál is lehetõség van az adattagok felsorolására, de ellentétben osztályokkal itt konkrét értéket kell nekik adni.



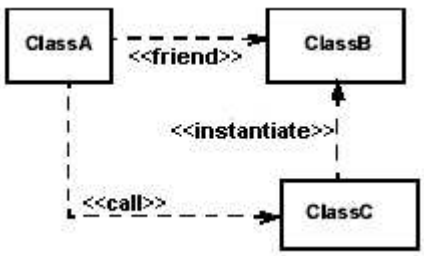
Több példányt is jelölhetünk egyszerre, ha az adattagok nem különbözõek.

Kapcsolatok (RELATIONSHIP)

Összeköttetés az elemek között.

❖ Függőség (dependency)

Használati kapcsolat, amely azt fejezi ki, hogy az egyik elem specifikációjának változása hatással a másik, őt használó elemre, de fordítva nem feltétlenül.



Standard sztereotípőiák:

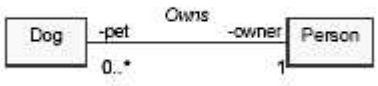
- **<<bind>>**. A forrás egy példánya lesz a cél templatenek, az adott aktuális paramétert felhasználva.
- **<<derive>>**. A forrás meghatározható a célból.
- **<<friend>>**. A forrás (**source**) speciális láthatóságot kap a célba (into **target**).
- **<<instanceOf>>**. A forrás a cél classifier egy konkrét példánya.
- **<<instantiate>>**. Példányosítás. A forrás generálja a cél példányait.
- **<<refine>>**. A forrás az absztrakció egy finomabb szintjén van.
- **<<use>>**. Forrás használja a célt.

❖ Asszociáció (association)

Az általános asszociáció kapcsolat egy classifierek közötti kétirányú összeköttetés, amelynél navigálási irány megadható az üzenet irányának jelzésére. Általában valamilyen használati kapcsolatot jelent a két classifier között, amelyek léte egymástól általában független, de legalább az egyik ismeri és/vagy használja a másikat (az ismeretség irányát megadhatjuk a navigálási irány segítségével). A kapcsolat

szemantikus összefüggést ábrázol és nem adatfolyamot (mindkét irányba lehet információ/adat továbbítás). A valódi összefüggés az osztályokból létrejövő objektumok között jön létre, amely különböző kölcsönhatásokat jelent. UML jelölés szerint az asszociációs kapcsolat egyszerű tömör vonal a két classifier között, amelyekre különböző tulajdonságok is elhelyezhetők.

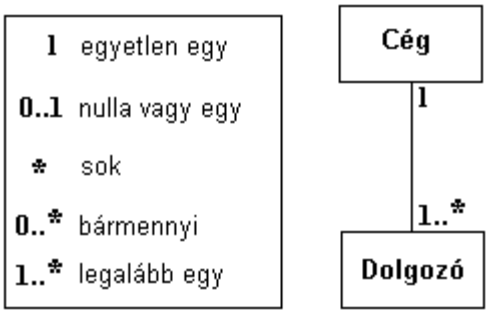
Az asszociáció kapcsolatnak opcionálisan lehet neve, amely általában egy igés kifejezés (pl. *Owns*, *Carries*, *HasHas*). A név olvasásának iránya is fontos (melyik osztálytól melyik felé), ha nem egyértelmű, akkor egy kis háromszög a név mellett mutathatja meg. A nevet szenvedő módba is lehet rakni, ha a fordított irány kifejezése fontosabb (pl. *Is owned by*).



A kapcsolatok további fontos tulajdonságait a *végpontokhoz* rendelhetjük hozzá. Így ezen tulajdonságok az adott végpontoknál szereplő osztállyal való kapcsolatra vonatkoznak, míg a kapcsolat név az egész kapcsolatra.

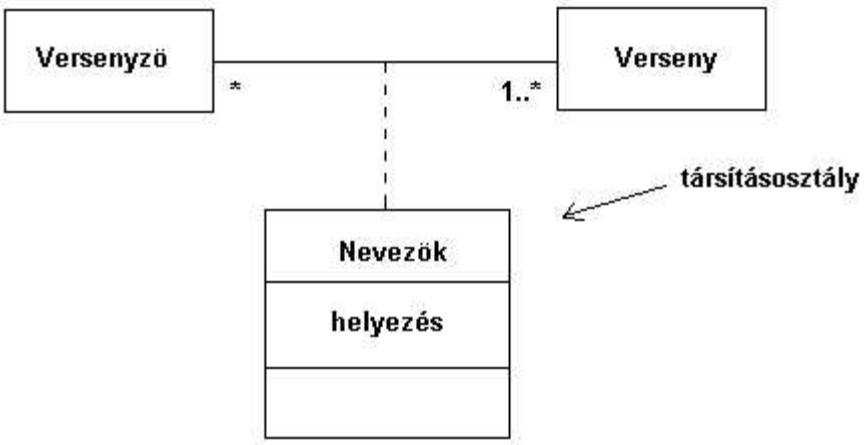
Az egyik végpont tulajdonság a **szerepnév** (***role names***). A szerepnév általában egy főnév, amely a kapcsolat adott végpontján levő osztály szerepét írja le az adott kapcsolatban. A szerepnek van láthatósága (private -, protected #, public +).

Multiplicitást a kapcsolatok végpontjainál definiálhatunk a hozzá tartozó osztályokhoz annak érdekében, hogy a kapcsolat megvalósulásánál résztvevő objektumoknak számát pontosítsuk. Például a cégnek több alkalmazottja is lehet, de az adott dolgozó csak egy cégnél dolgozik.

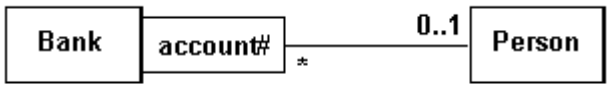


Kapcsolat természetesen létesíthetünk egyazon osztályból kiindulva saját magával is, hiszen az azonos osztályhoz tartozó különböző objektum példányok egymással is kommunikálhatnak. A reflexív kapcsolatoknál általában a szerepek vannak megnevezve és nem maga a kapcsolat.

Társításosztály (association class). A társításosztályok segítségével az asszociációkhoz (társításokhoz) tulajdonságokat és műveleteket rendelhetünk. Erre például akkor van szükség, ha két osztály egyedei között több-több értelmű leképezést akarunk megvalósítani, de az egymáshoz rendelt párokhoz még olyan további információkat is rendelnénk, amelzek igazából egyik osztályhoz sem rendelhetők kizárólagos módon.

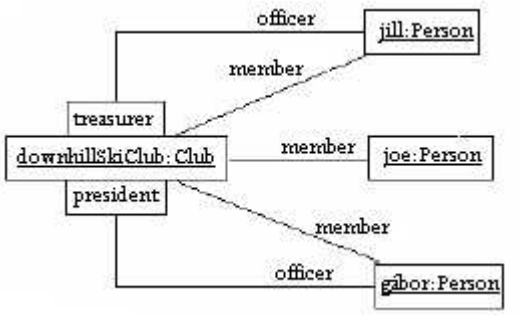


Minősítők (qualifiers). A kvalifikált társítás egy tulajdonságot használ föl kulcs gyanánt, melynek segítségével a társítás megfelelő végéhez tartozó osztályok csoportosíthatók, és a megfelelő kulcsérték alapján az adott csoport elemei lekérdezhetők.



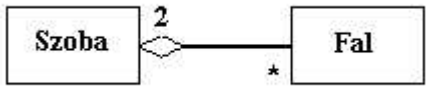
- **Link (link)**

Asszociáció (társítás) egy példánya. Objektumok közötti kapcsolat. Multiplicitást nem jelöljük, mert a link az példány. Minősítő, vagy annak értéke jelölhető.



- **Aggregáció (aggregation)**

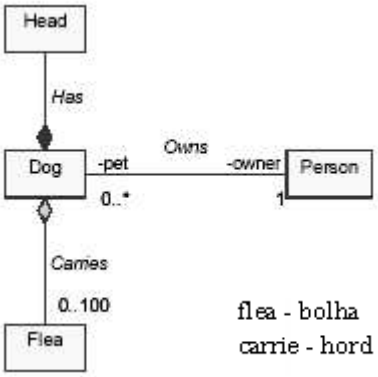
Ez a kapcsolat az asszociáció egy speciális formája, amely „rész-egész” viszonyt fejez ki a résztvevő osztályok között (erősebb mint az asszociáció), vagyis azt, hogy az egyik objektum fizikailag tartalmazza vagy birtokolja a másikat. Ezt a kapcsolatot sok esetben nagyon nehéz precízen meghatározni, hiszen nem mindig egyértelmű, hogy a kapcsolat teljes vagy részleges tartalmazást jelent, vagy csak hivatkozást a másik objektumra. Egy asszociációból általában akkor célszerű aggregációt csinálni, amikor a kapcsolat leírásánál a „része” kifejezés egyértelműen használható vagy amikor az egészobjektum bizonyos műveletei automatikusan a rész-objektumokra is vonatkoznak (pl. megszűnés). UML jelölés szerint a rombusz a tartalmazó oldalán jelzi az aggregációt.

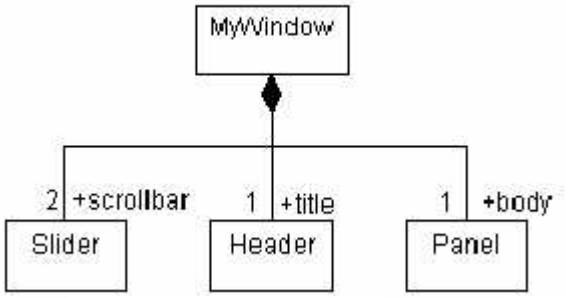


Alapvetően kétféle aggregációt különböztetünk meg:

- összetétel,gyenge tartalmazás – általános aggregáció (üres rombusszal jelölve)
- erős tartalmazás, vagy más néven *kompozíció* – akkor használjuk amikor rész(ek) élettartama szigorúan megegyezik az egészével (teli rombusz)

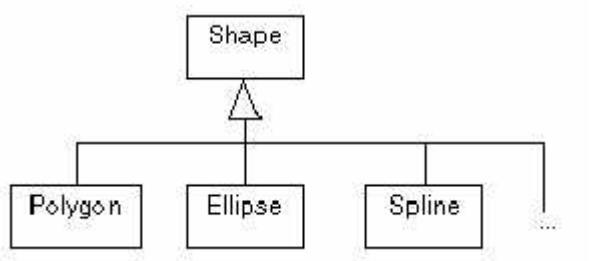
A kompozíciót általában könnyű felismerni, pl. arról, hogy ha a tartalmazó megszűnik, akkor biztos a tartalmazottnak is meg kell szűnnie.





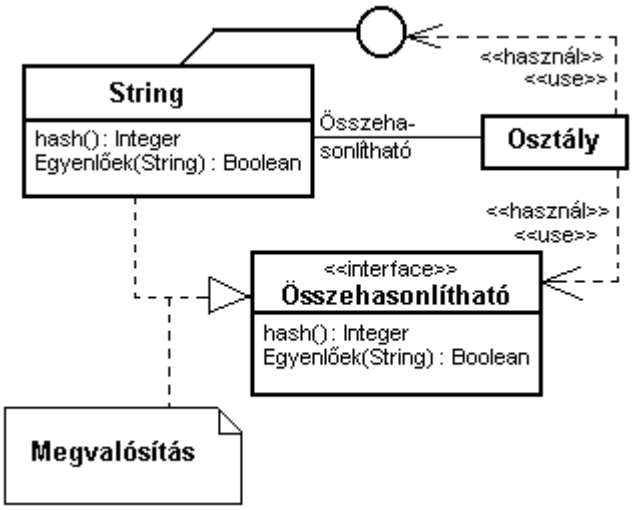
❖ **Generalizáció (generalization)**

Rendszertani kapcsolat egy általánosabb és egy specifikusabb elem között. A specifikusabb elem teljesen konzisztens az általánosabb elemmel (tartalmazza annak minden tulajdonságát, kapcsolatait), de tartalmazhat még kiegészítő információt.
Az általánosítás kapcsolat az UML-ben egy nagyon sokrétűen alkalmazható kapcsolat. Osztály diagramok esetében az osztályok közötti öröklődés (*inheritance*) ábrázolására használatos.



❖ **Realizáció (realization)**

Egy classifier specifikál egy szerződést, melyet egy másik classifier kivitelez.



➔ ***osztály-diagram***

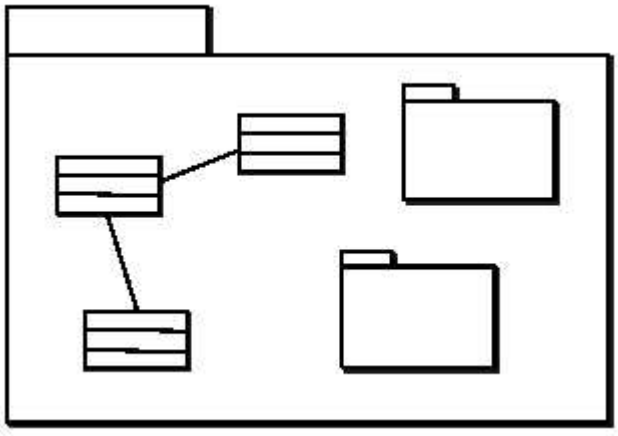
- Az osztályok és az interfészek kapcsolatrendszerét kifejező struktúra-ábra
- Egyszerű kapcsolatokat mutat
- Tartalmaz: osztályokat, interfészeket, kapcsolatokat

➔ ***objektum-diagram***

- az objektumok kapcsolatrendszerét, a hierarchikus viszonyt kifejező struktúra-ábra
- pillanatfelvételt ad a rendszer állapotáról
- tartalmaz: objektumokat, linkeket

Csomagok

- a csomagok logikailag összetartozó elemek osztálybasorolására, egységes kezelésére és megjelenítésére szolgálnak
- A csomag modellezési elemek csoportosítására szolgál és általánosan használható, de legnagyobb jelentősége a szerkezeti modellezésnél van.
- Különböző elemeket is tartalmazhat, további csomagokat is, így egy hierarchikus szerkezetet képezve.
- Egy csomag elemeinek láthatósági attribútumokat lehet megadni, amelyek más csomag elemei számára korlátozzák az elérhetőséget.(A csoportokra értelmezett láthatóság az osztályoknál megismertek.)
- Csomagok között függőségi kapcsolatok lehetnek.
- A csoportok közötti függőséget egy szaggatott vonalas, teli nyíllal jelöljük. Az **<<import>>** a függőség speciális esete, amikor is garantáljuk egyik csoport számára, hogy az láthassa a másik tartalmát. **<<acces>>** esetében a teljes elérési útvonalat át kell adni.



❖ **Generalizáció**

Szemantikája megegyezik az osztályoknál bemutatottakkal.

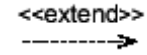
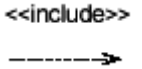
Sztereotípiák: **<<facade>>** –
<<framework>> – minta van deffelve
<<stub>> –a modell nincs teljesen kidolgozva; távoli rendszer szimulálása
<<subsystem>> – önállóan működőképes rész
<<system>> – teljes rendszert reprezentálja

Use Case

Használati eset modellezéskor a rendszer viselkedését írjuk le, ahogyan az egy külső szemlélő szemszögéből látszik. Ez azt jelenti, hogy a rendszer belső szerkezetéről vagy viselkedéséről nem tesz említést. Leírja a rendszer funkcionalitását, a főbb tevékenységeit és kapcsolatát a külvilággal.

Használati eset modellezésnek fő kellékei a használati esetek és a szereplők, amelyekből maguk a használati eset diagramok épülnek fel.

Elem	Leírás	Szintaxis
Használati eset (use case)	tevékenységek sorozata, amelyet a rendszer végre tud hajtani a szereplőkkel kommunikálva	
Szereplő (actor)	szerepek összefüggő halmaza, amelyeket a használati eseteket használók játszhatnak	
Rendszerhatár	fizikai rendszer és a szereplők közötti határvonal	

Elem	Leírás	Szintaxis
Asszociáció (association)	jelzi, hogy a szereplő részt vesz a használati esetben	
Általánosítás (generalization)	osztályozó kapcsolat egy általánosabb és egy speciálisabb elem között (itt: használati eset)	
Kibővítés (extend)	kibővített használati esettől alap használati eset felé (bővített tevékenység)	
Magábanfoglalás (include)	alap használati esettől magában foglalt használati eset felé (résztevékenység)	

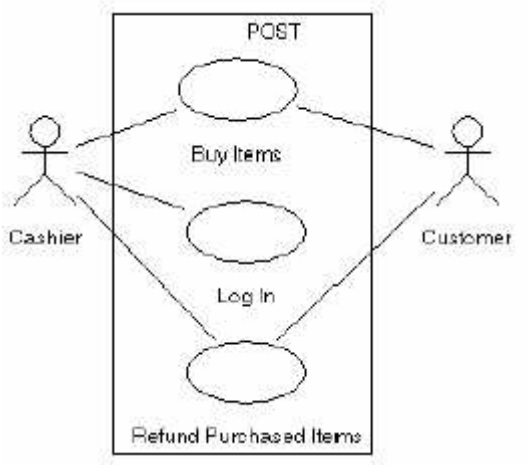
Szereplők (aktorok). Az szerplők nem részei a rendszernek, akárki vagy akármi lehet, ami a rendszerrel kapcsolatban van. A szereplő továbbít és/vagy fogad információt a rendszer felé, -től.

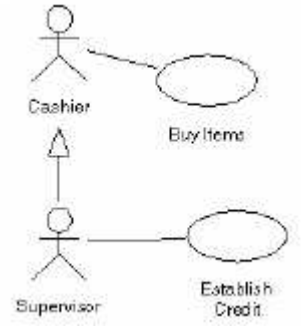
Használati Esetek. A használati esetek képezik a használati eset modellezés azon elemeit, amellyel a rendszer funkcionalitását írjuk le. Nagyon fontos, hogy ez a funkcionalitás úgy legyen megfogalmazva, ahogy az a szereplő számára, azaz kívülről

Use Case: **Buy Items**
Actors: Customer (initiator), Cashier
Purpose: Capture a sale and its payment
Overview: A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collects payment. On completion, the Customer leaves with the items.
Cross Refs: Refs. to requirement docs.

Actor Action	System Response
1. Customer arrives at a POST checkout with items to purchase	
2. The Cashier records the identifier from each item If there is more than one item of the same item, the Cashier can enter the quantity as well	3. Determines the item price and adds the item information to the running sales transaction The description and price of the current item are presented

látható, (tehát nem a működés megvalósulását írjuk le). Valójában a használati eset nem más, mint a szereplő és rendszer közötti „párbeszéd”. A használati esetek mindegyike megadható egy egyszerű, természetes nyelvi leírással.





Generalizáció:

- ***use casek között*** → gyermek örökli a szülő viselkedését és jelentését
- **aktorok között** → a leszármaztatott tud kommunikálni azokkal a fajta use caseszekkel, amivel a szülő

Amikor két használati eset közé adunk meg kapcsolatot, akkor az kétféle specialitás függőségi (dependency) kapcsolat lehet, sztereótípusokkal megadva: „magábanfoglalja” és „bővíti” (**<<include>>** és **<<extend>>**). A „magábanfoglalja” esetében részfunk-cionalitást ábrázolunk, amikor az egyik (rész)funkció többször is szerepel. Ez azt jelenti, hogy a részfunkcionalitás minden esetben végre hajtandó. A „bővíti” esetében viszont, valamilyen speciális funkcionalitással történő kibővítést írunk le (pl. kivételes viselkedés) és esetleg opcionális végrehajtást is jelenthet.

Viselkedési modellezés

Szekvencia diagram

A szekvencia diagram objektum-kölcsönhatásokat mutat be az idő függvényében, a szcenárióban szereplő objektumokat és osztályokat ábrázolja a közöttük küldött üzenetekkel. A legfontosabb ismérve, hogy időorientáltan mutatja be az üzeneteket, tehát az időbeli változások jól követhetők rajta. Egy-egy szekvencia diagram mindig egy konkrét végrehajtást ábrázol.

A szekvencia diagram az egyes objektumok közötti üzenetváltásokat, és azok időbeli lefolyásának sorrendjét mutatja. Vízszintes tengelyéhez a tevékenységek végrehajtásában részt vevő objektumok, függőleges tengelyéhez pedig az idő tartozik.

A szekvencia diagramon az objektumokat az objektumdiagramokon szokásos módon az **<objektum>:<osztály>** szintaxissal jelöljük. Az osztály vagy objektum neve elhagyható, a kettőspont és az aláhúzás azonban mindig kötelező.

Az egyes objektumok időbeli viselkedését (életpályáját, lifeline) az objektumot jelölő dobozból kiinduló szaggatott függőleges vonal jelöli. Az objektumok mindig a hozzájuk tartozó életpálya legfelső pontján helyezkednek el.

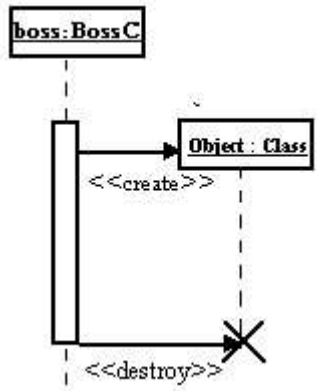
Az objektumok egymással üzeneteket váltanak. Az üzenetek vízszintes nyilak jelölik, melyekre címkeként rá kell írni az üzenet nevét. A nyíl iránya értelemszerűen megadja az üzenet küldőjét és fogadóját. Az üzenetek különböző fajtáinak jelölése:

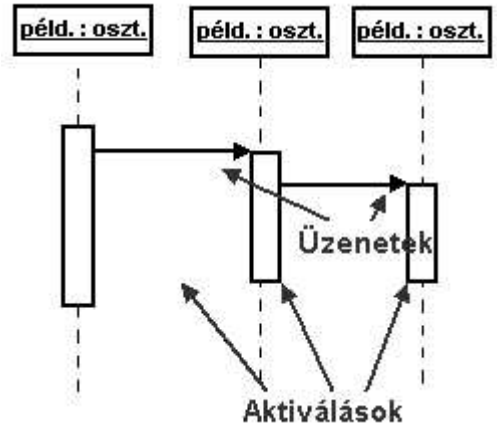
Nyílak	Jelentésük
→	Egyszerű
→	Szinkron
→	Aszinkron
↩	Kikerül
◷→	Időt kér
←- - - -	Visszatérés

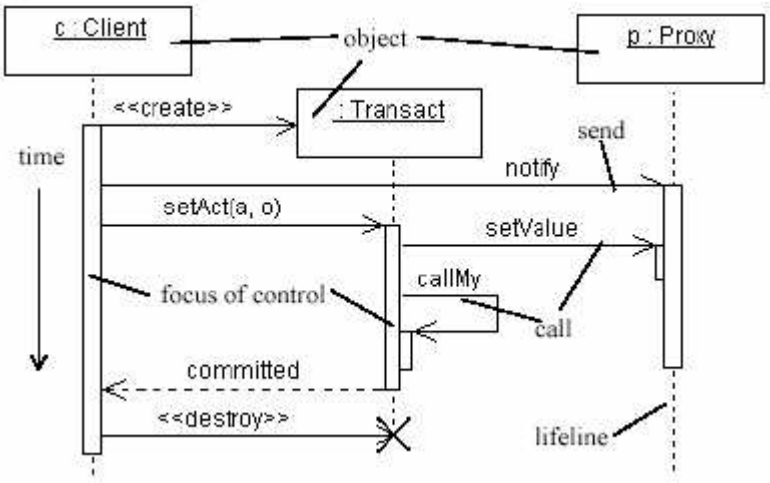
- ***Hívás-visszatérés***: a híváshoz tartozó nyíl folytonos; visszatéréshez tartozó nyíl szára szaggatott
- Konkurrens rendszerben a tömör nyílfej ***szinkron*** üzenetváltást, a fél nyílfej pedig ***aszinkron*** üzenetváltást jelöl, vagyis az első esetben az üzenet küldője köteles kivárni a választ, a második esetben viszont nem.

Az egyes objektumok életpályájuk során lehetnek aktív vagy passzív állapotban. Az aktív objektum éppen tevékenységet hajt végre, a passzív objektum pedig nem. Az aktív állapotot a szaggatott vonalat eltakaró téglalappal jelöljük.

Objektumok a szekvencia diagramban jelölt események hatására létrejöhetnek (ekkor az illető objektum nem a diagram felső szélén, hanem az őt létrehozó üzenettel egy magasságban jelenik meg), illetve meg is szűnhetnek (ekkor az objektum életpályáját egy nagy X zárja le, és onnantól az életpálya nem folytatódik tovább).



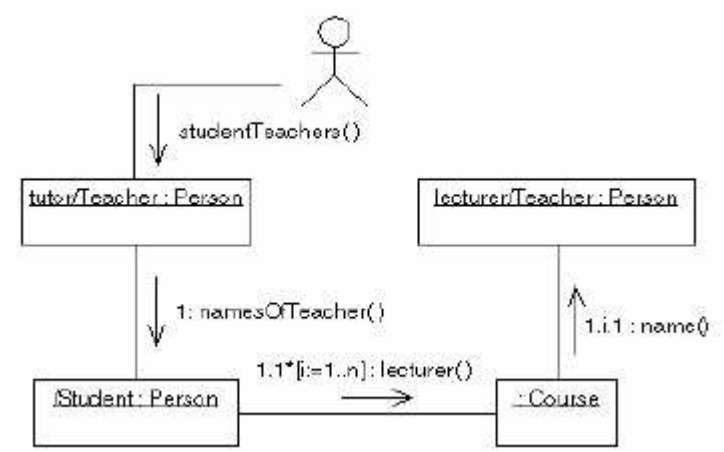




Együttműködési diagram (collaboration diagram)

A kollaborációs diagram - ellentétben a szekvenciális diagrammal - nem időt kezel, hanem üzenetsorrendet. Ezt számozással érjük el, pl.: az első üzenetcsoportbeli elemek számozása 1.1 1.2 1.3 ... Az üzenetek kiadása lehet feltételtől függő vagy ciklikus. A megfelelő kifejezést szögletes zárójelbe tesszük. A ciklust az különbözteti meg a feltételtől, hogy a számozás után csillag szerepel.

2: display(x,y) egyszerû üzenet
1.3.1: P:= find(spec) egymásbaágyazott hívás visszatérési értékkel
[x < 4] 4: invert(x, color) feltételes üzenet
1.1 *[i :=1..n]: drawShape(i) iteráció
3a, 4b/ 4.1 * : update( ) előfeltétel (4,1-hez kell a 3a és 4b)



Objektumnevek szintaxosa:

:C – nemmegnevezett objektum a C osztályból

/R – nemmegnevezett objektum az R szerepben

/R:C – nemmegnevezett objektum a C osztályból R szerepben

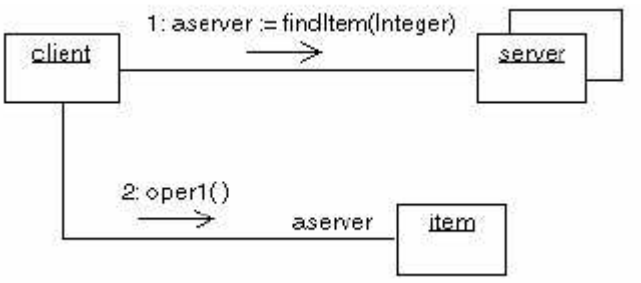
O/R – az O nevű objektum az R szerepet játsza

O:C – O nevű objektum a C osztályból

O/R:C – O nevű objektum a C osztályból R szerepben

O – O nevű objektum

Együttműködési diagramban multiobjektum is szerepelhet → sokaságból keresünk egy példányt.



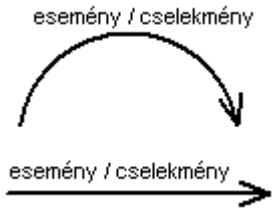
Állapotátmeneti diagram (statechart diagram)

Az állapotátmenet diagram egy konkrét objektum állapotait ábrázolja, az eseményeket és üzeneteket, amik az állapotváltásokat kiváltják, és az akciókat, amik végrehajtódnak állapotváltáskor. Az állapotátmeneti diagram magába foglalja az összes üzenetet, amit egy objektum küldhet, illetve fogadhat.

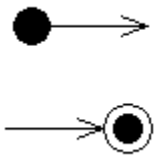
Az **állapot** a példány életében egy szituációt jelent. Az állapotoknak megadhatunk több jellemzõt is:

- Állapotba érkezéskor végrehajtódó funkciók (**entry**)
- Állapotban tartózkodáskor folyamatosan végrehajtódó funkciók (**do**)
- Állapot elhagyásakor végrehajtandó funkciók (**exit**)
- Ségítség-információ

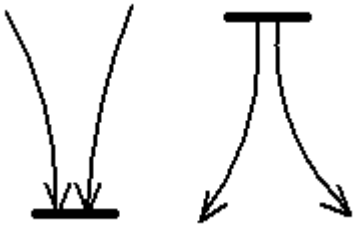
Az **állapotátmenet** egy objektum állapotának megváltozását jelenti (maradhat ugyanaz is), amit egy akció kísérhet. Az állapotátmenet lehet *automatikus*, amikor befejeződött az egyik aktivitása, vagy *nem automatikus*, ha egy névvel ellátott üzenet váltja ki (másik objektumtól vagy külső). Mindkét esetben azonban az állapotátmenet időtartama nullának tekintendő és nem lehet azt megszakítani (az állapot megszakítható). UML jelölése egy nyíl az eredendő állapotból az újba.



Két speciális állapot van:



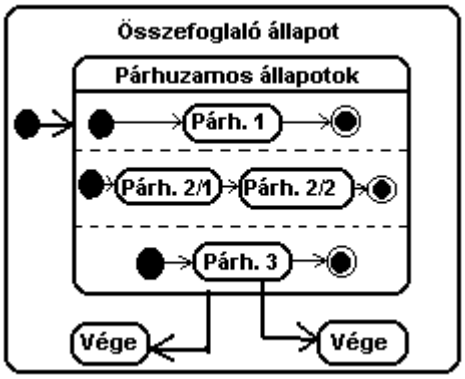
- Kezdő (pszeudo-) állapot (*Start State*): minden diagramnak kell lennie pontosan egy kezdő állapotnak.
- Végállapot (*End State*): több is lehet egy diagramon belül.



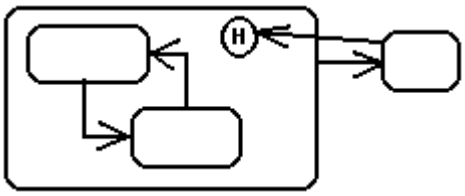
A vezérlést szinkronizálni, valamint állapotok közt megosztani tudjuk. A szinkronizálás azt jelenti, hogy a következő állapotba lépés feltétele mindkét vezérlőjel megérkezése.

Egyes állapotokat kinagyíthatunk (az Összefoglaló állapot), valamint párhuzamosíthatjuk õket. A párhuzamosan folyó feladatok állapotából két féleképpen lehet kilépni:

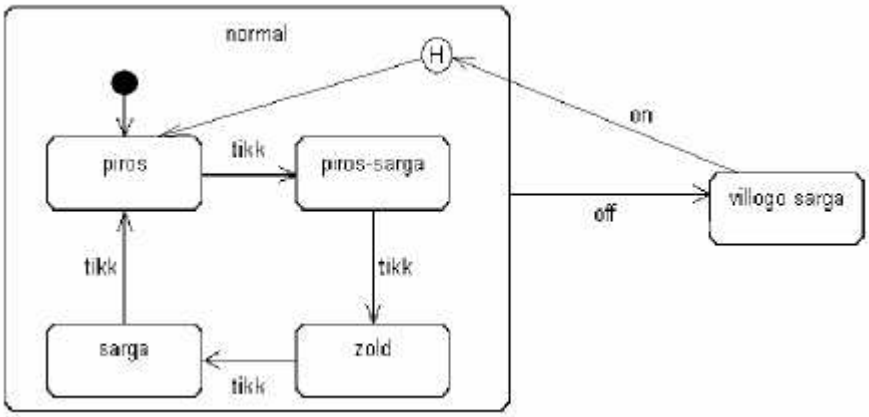
- Egy belsõ állapotból kilépünk egy külsõbe.
- Az összes párhuzamos folyamat végállapotba került.



A **History Indicator** (jelölése: H betû egy körben) megjegyzi, az adott állapotból milyen alállapot lévén kerültünk ki. Ha visszalépéskor a History Indicatorba lépünk, ebbe az alállpotba kerülünk.



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.



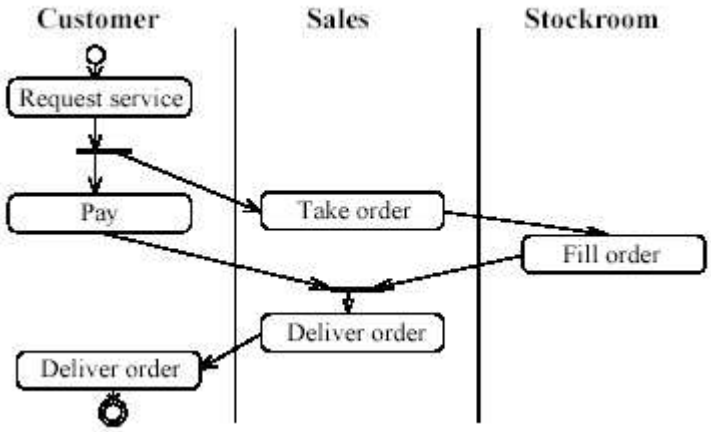
Aktivitás diagram (activity diagram)

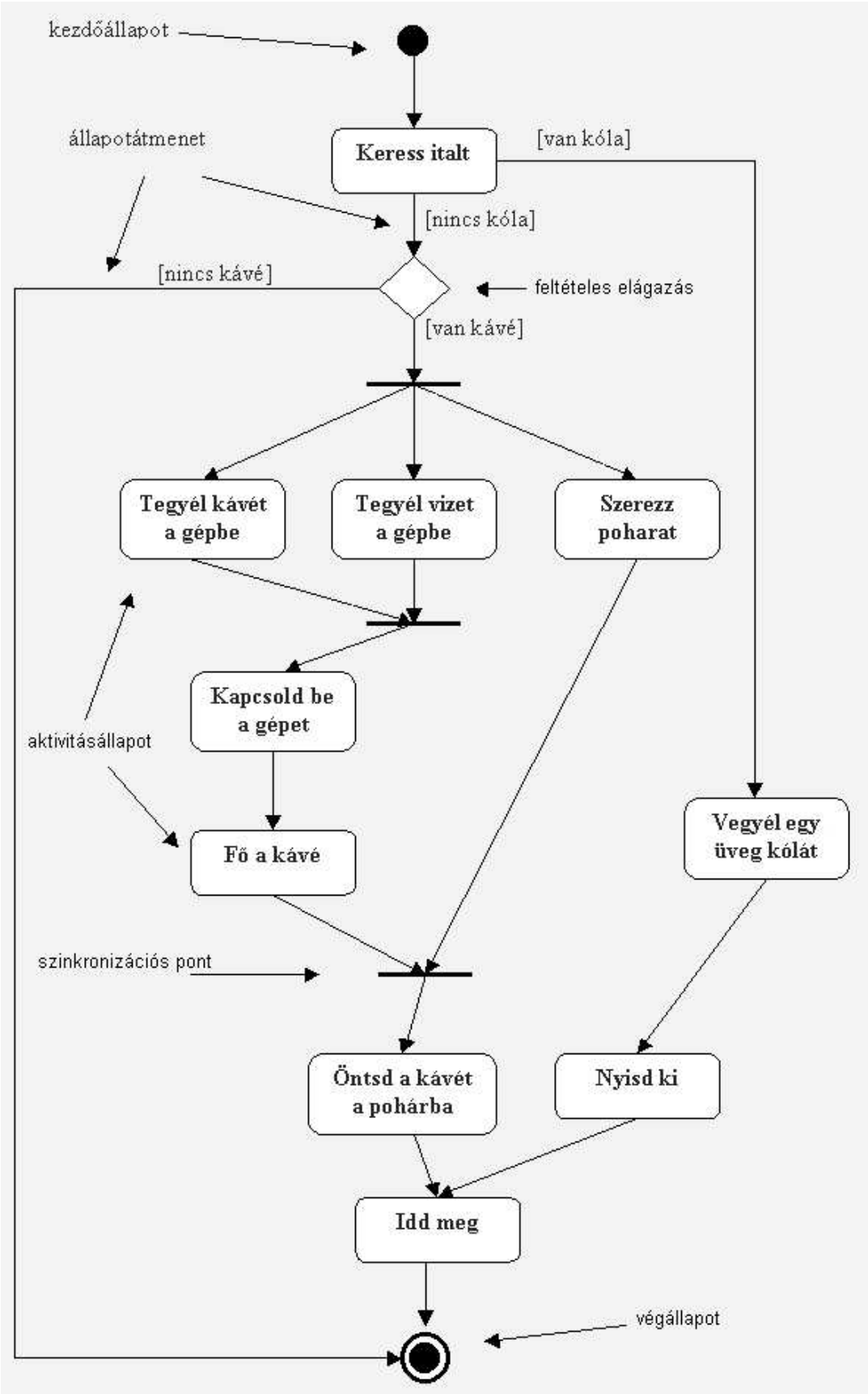
Az aktivitásdiagramok aktivitásállapotokat (félkör oldalú téglalapok), állapotátmeneteket (nyilak), feltételes elágozásokat (rombuszok) illetve szinkronizációs pontokat tartalmaznak (vastag, vízszintes vonalak). Az aktivitásdiagramok is rendelkeznek kezdőállapottal, és tartalmazhatnak egy vagy több végállapotot is.

Az **aktivitásállapot** (***action state***) belső tevékenységet magába foglaló állapotot jelöl, melyhez legalább egy bemeneti, és legalább egy kimeneti állapotátmenet kapcsolódik.

Párhuzamos elágozások és csatlakozások szinkronizációs pontok formájában írhatók le. A szinkronizációs pont az aktivitásdiagram olyan pontja, melyhez bemenő és kimenő állapotátmenetek kapcsolódnak. A kimenő állapotátmenetek mindegyike párhuzamosan végrehajtódik, ha az utolsó bemenő állapotátmenet is végrehajtódott.

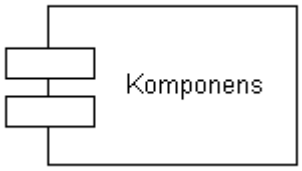
Az aktivitásdiagramot feloszthatjuk függőleges életpályákra. Egy-egy életpálya a szekvenciadiagramhoz hasonlóan egy-egy objektumhoz tartozik, ily módon megmondhatjuk, hogy az aktivitásdiagram egyes aktivitásállapotait melyik objektumok hajtják végre.




kezdőállapot
állapotátmenet
Keress italt
[van kóla]
[nincs kóla]
[nincs kávé]
feltételes elágazás
[van kávé]
Tegyél kávét a gépbe
Tegyél vizet a gépbe
Szerezz poharat
Kapcsold be a gépet
aktivitásállapot
Fő a kávé
Vegyél egy üveg kólát
szinkronizációs pont
Öntsd a kávét a pohárba
Nyisd ki
Idd meg
végállapot

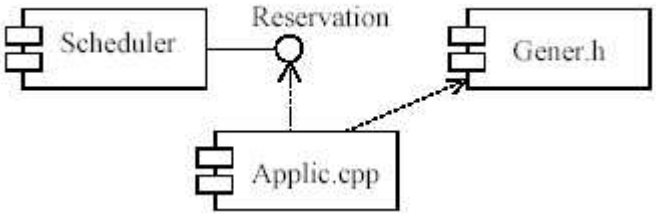
Komponensdiagram

Leírja a rendszer fizikai komponenseinek szervezését. A komponens a rendszer fizikai építőeleme.

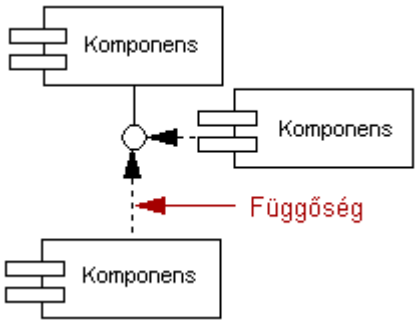


A komponensdiagram tartalmaz:

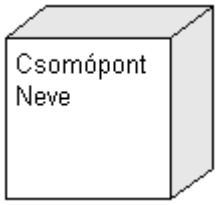
- Komponenseket (amelyek a node-okon élnek)
- Interfészeket
- Csomagokat
- Függőséget, asszociációt, realizációt, generalizációt



Komponens csak az interfészén keresztül érhető el.



Telepítési (feladatkiosztási diagram, deployment diagram)

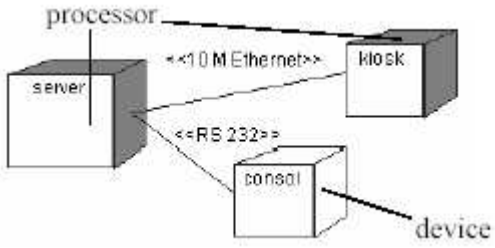


Leírást ad a rendszerről csomópontok, összeköttetések és komponensek formájában.

A csomópontok (node) a számítógépes rendszerünk fizikai erőforrásait reprezentálják.

Csomópont : processzor → feldolgozóképességgel is rendelkezik

device → számítási kapacitása van, de feldolgozóképessége nincs

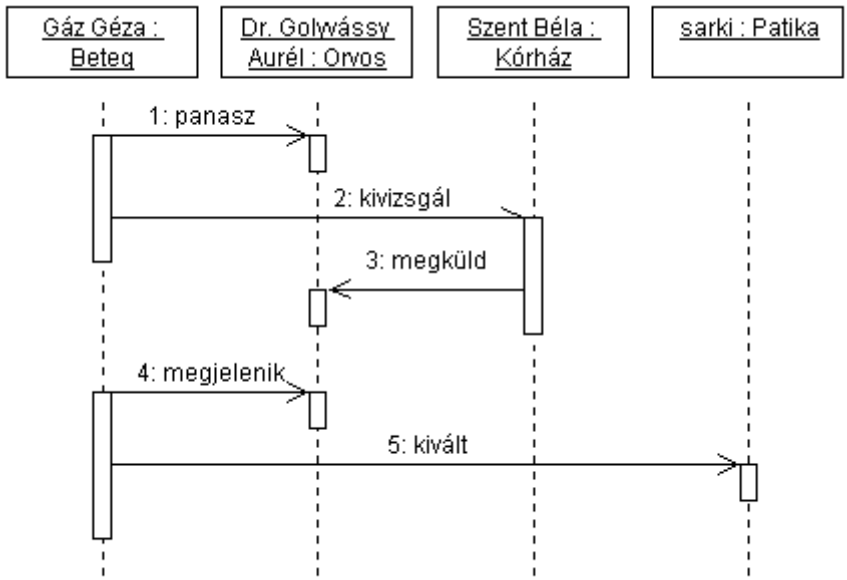


Feladatok:

1. Rajzoljon az eseményekről UML szekvencia-diagramot !

Gáz Géza rosszul érzi magát, és elmegy a háziorvosához. Elpanaszolja neki, hogy már két hete lázas, és a jobb lábfeje a kétszeresére dagadt. Dr. Golyvássy Aurél, az orvos elküldi beutalóval Gáz urat kivizsgálásra a Szent Béla kórházba, ahol különböző vizsgálatokat ejtenek meg rajta. Az adatokat pár nap alatt kiértékelik és postán megküldik Dr. Golyvássynak. Amikor Gáz úr ismét megjelenik a rendelésen, a doktor receptre felír Gáz úrnak köptetőt, amelyet ő kivált a sarki gyógyszertárban.

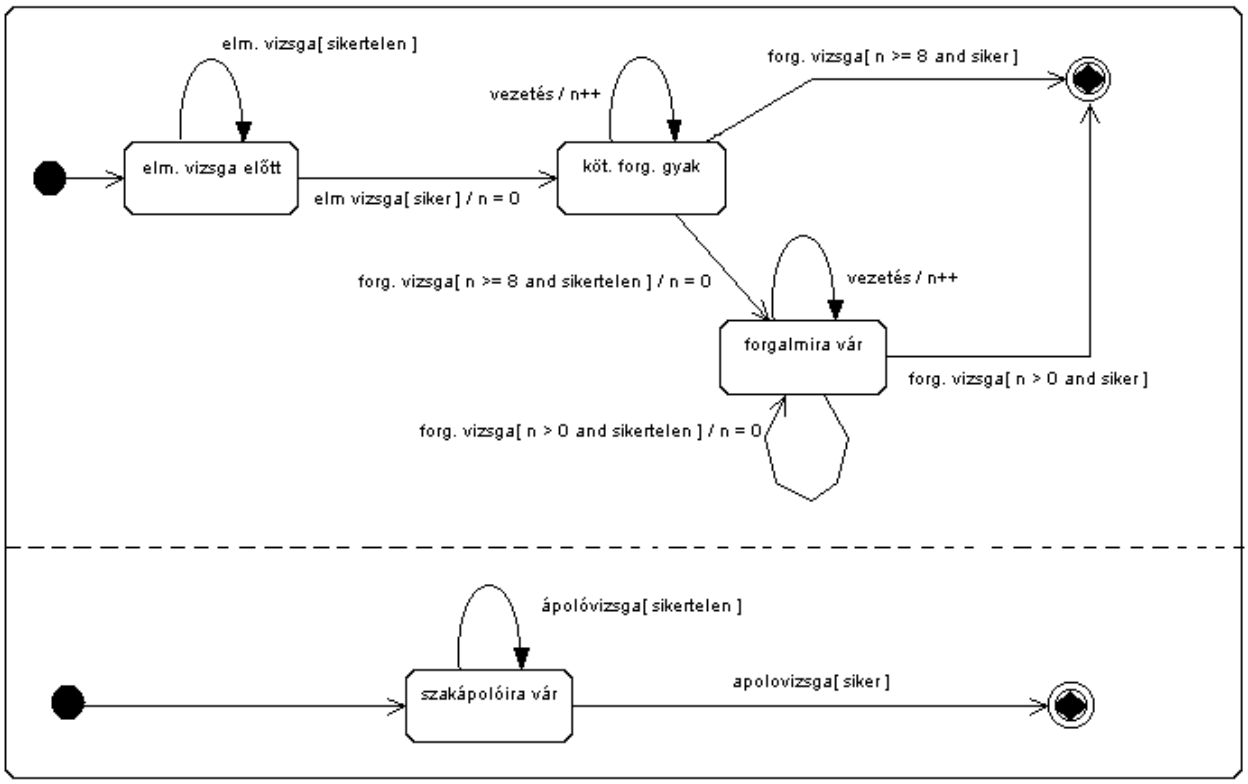
megoldás:



2. Rajzolja meg a hallgató viselkedését leíró UML state-chartot! Az állapot-átmeneteket pontosan definiálja.

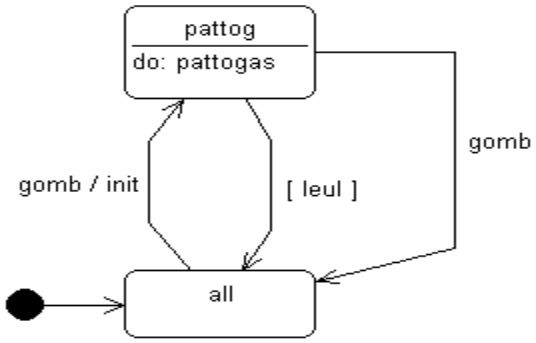
A zöldlámpás autók vezetésére jogosító képzésben résztvevő hallgatónak a zöldlámpa használatára vonatkozó elméleti vizsga letétele után legalább 8 alkalommal kell a forgalomban felügyelet mellett vezetési gyakorlatot végezni, majd ezt követi a forgalmi vizsga. A jogosítvány megszerzéséhez ugyancsak szükséges 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.

megoldás:

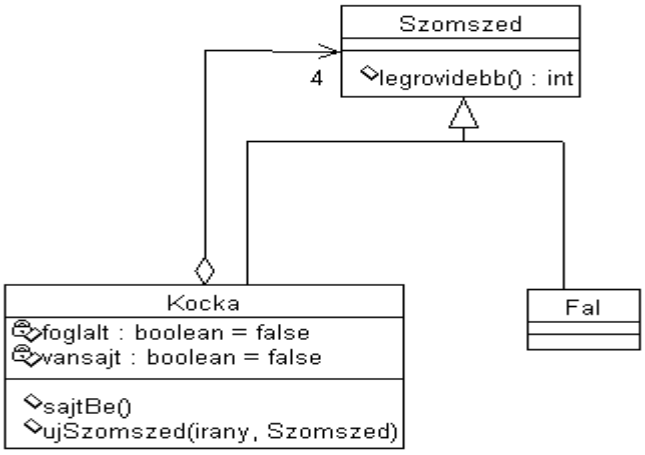


3. A leejtett labda pattogását utánzó program vezérlésére egy gomb szolgál. A gombot megnyomva a labdát az induló helyzetből elengedjük, mire a labda leesik és energiát vesztve a földről visszapattan. A pattogást addig folytatja, amíg az energiája el nem fogy (leül). Ha a labda leült, a gombot megnyomva 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.

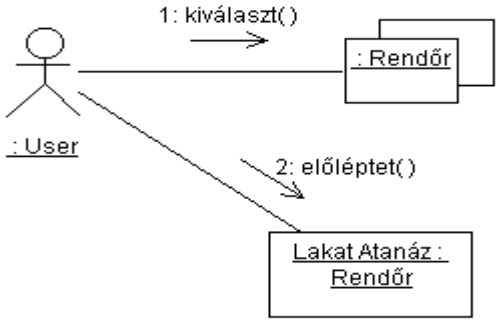
Rajzolja meg a labda viselkedését leíró UML state-chartot !



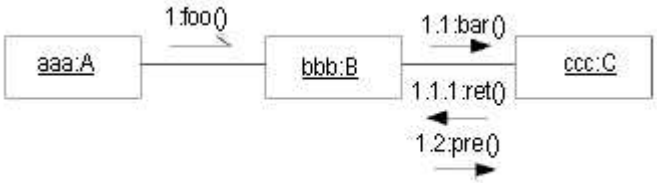
4. Egy labirintus olyan kockákból áll, amelyeknek négy szomszédja van. A szomszéd vagy fal, vagy egy másik kocka. Egy kockához hozzárendelhetjük a szomszédait. A kockába tehetünk sajtot, és minden kockától megkérdezhetjük, hogy milyen hosszú (hány kockából áll) a legrövidebb út a sajtig.
Készítsen UML osztálydiagramot és azon adja meg az osztályok metódusainak szignatúráit és az attribútumokat is !



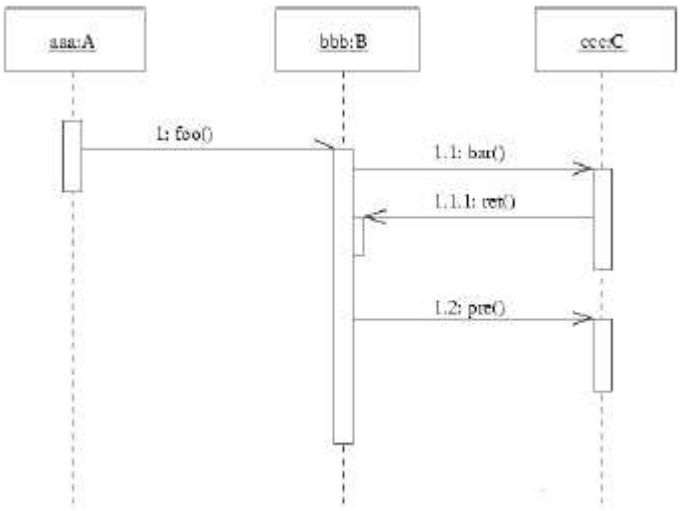
5. Rajzoljon kollaborációs diagramot, arra az esetre, amikor egy user a Rendőr osztályú objektumok egy kollekciójából egy konkrét rendőrt előléptet!



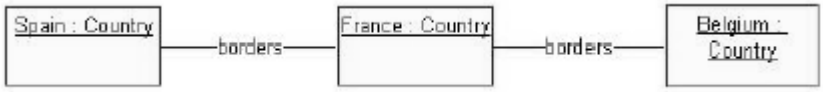
6. Az alábbi UML kollaborációs diagram alapján rajzoljon UML szekvencia diagramot! Ügyeljen az activation bar-ok helyes ábrázolására is!



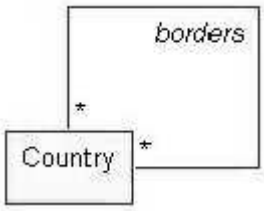
megoldás:



7. Rajzoljon UML osztálydiagramot az alábbi objektumdiagram alapján !

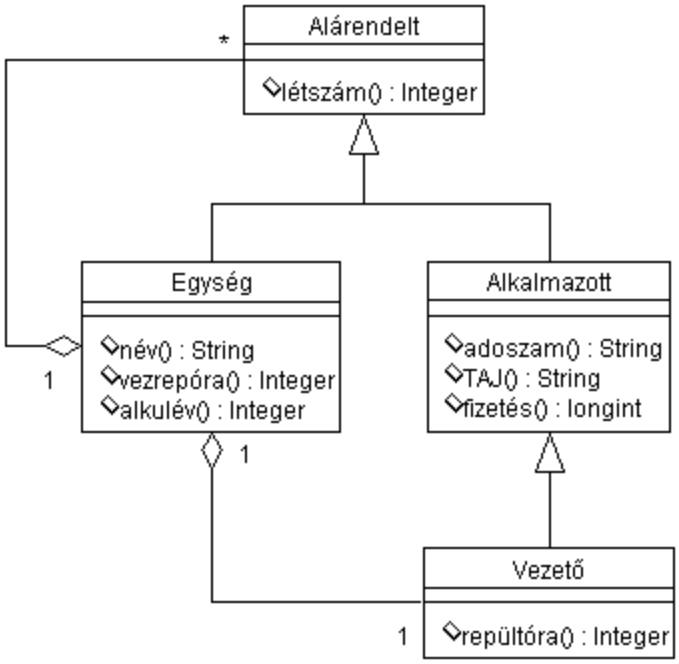


megoldás:



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

A Vidékelőremozdító Minisztérium felépítése sok rétegű hierarchiát alkot. A minisztérium államtitkárságok-ból, 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.

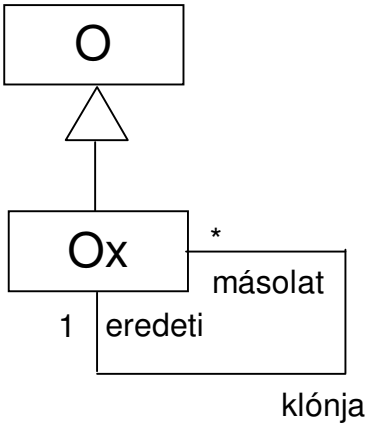


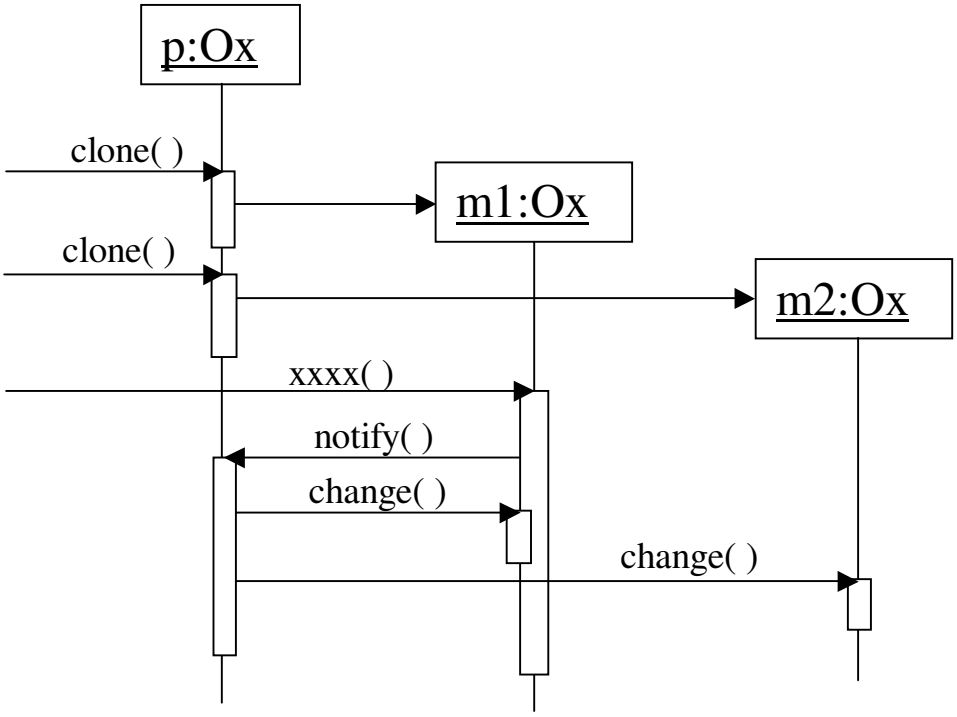
9. Legyen egy tetszőleges O osztály. Ennek egy olyan Ox változatát kell elkészíteni, amelyik rendelkezik O valamennyi tulajdonságával + egy p példánya a clone( ) metódusának meghívásakor készít magáról egy m másolatot, amely folyamatosan követi p állapotának minden változását. p-ről több másolat is készíthető, de m már nem másolható tovább. Egy m másolaton egy kliens részéről kezdeményezett olyan műveletet, amely m állapotának megváltozását okozná, m továbbítja p-nek, aki a műveletet saját magán hajtja végre, hogy valamennyi másolata kövesse a változtatást.

Adja meg Ox attribútumait. Rajzoljon szekvencia diagramot arra az esetre, amikor p-ről egy kliens készít két másolatot (m1, m2), majd az m1-nek meghívja egy olyan metódusát (xxxx()), amely m1-nek módosulását okozná.

megoldás:

Ox attribútumai: klónozható

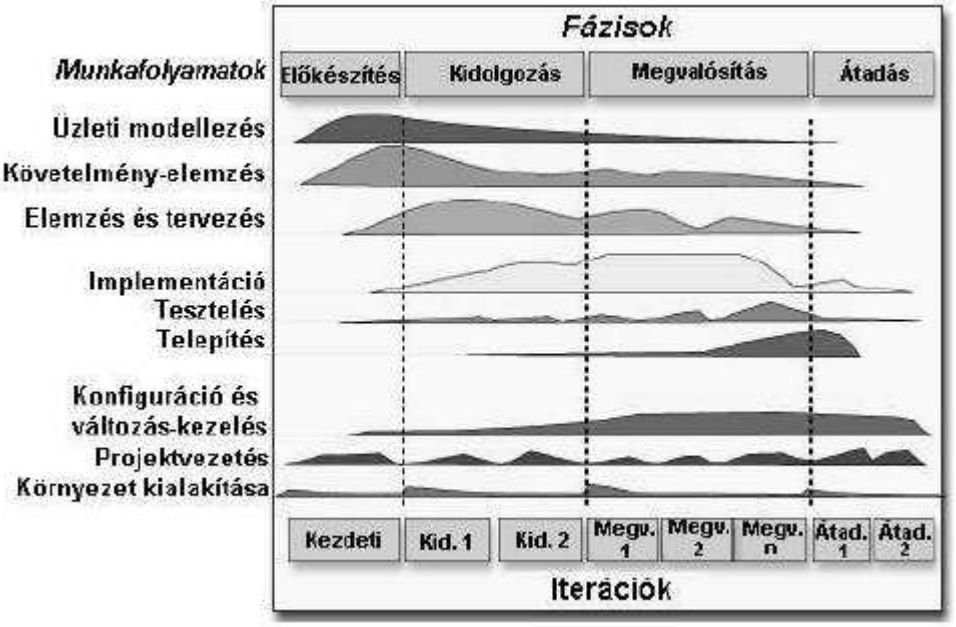




7, rész
RUP

RUP (Rational Unified Process)

Az UML tervezési nyelv megalkotói által létrehozott iteratív szoftverfejlesztési modell. A modellt az alábbi fázisok és munkafolyamatok diagram illusztrálja. A fejlesztés négy fázisból áll és minden fázis egy vagy több iterációból. Az egyes iterációkban az ábra szerinti munkafolyamatok felülről lefelé kerülnek végrehajtásra. Egy átlag projekt összes elvégzendő munkájának az egyes fázisokra és iterációkra eső hányadát a munkafolyamatok soraiban lévő munkamennyiség alakzatok területei szemléltetik.



RUP munkafolyamatok-fázisok diagram

A RUP használati (felhasználói) **esetek által hajtott, architektúra centrikus és inkrementális** (fokozatos) **fejlesztési modell**.

Az architektúra centrikusság úgy értendő, hogy a kidolgozás fázis végére az architektúrának stabilnak kell lennie (lásd alább).

A használati esetek által hajtott modell úgy értendő, hogy az architektúrát a megfelelően kiválasztott használati esetek tervezése, implementálása és tesztelése jellemzi (lásd alább), majd a megvalósítás fázis iterációiban is, a megfelel_ sorrendben választott további használati esetek megvalósítása történik.

Az inkrementális modell úgy értendő, hogy a szoftver fokozatosan épül; kiindulva az architektúrális alapverzióból (lásd alább), minden iterációban teljes integrálásra kerülnek az abban megvalósított használati esetek.

RUP-fázisok

A RUP előkészítés (kezdet) - fázisa

A RUP kezdet-fázisa munkafolyamatainak súlypontját a RUP munkafolyamatok-fázisok diagram illusztrálja . Itt, a következő kérdések megválaszolása van a fókuszban:

- A rendszer milyen szolgáltatást nyújtson a legfontosabb felhasználók számára?
- Hogy nézhet ki a rendszer architektúrája?
- Mi a terve és költsége a rendszer fejlesztésének?

Az első kérdésre aktivációs és felhasználói esetek diagramjai, valamint a funkcionális és nemfunkcionális követelmények meghatározása adja a választ. A rendszer architektúrája még próbálkozó ebben a fázisban és csak a fontosabb modulok/alrendszerek vázlatát tartalmazza. A legfontosabb kockázatok azonosítása és fontossági sorrendbe állítása, a következő fázis (kidolgozás) részletes megtervezése és az egész projekt előzetes terve és költségfelbecslése ad választ a harmadik kérdésre.

A RUP kidolgozás - fázisa

A RUP kidolgozás-fázis munkafolyamatainak súlypontját a RUP munkafolyamatok-fázisok diagram illusztrálja. Itt, a következő kérdés megválaszolása van a fókuszban:

- Kellően stabilok és kezelhetőek a felhasználói esetek, az architektúra és a tervek ahhoz, hogy a teljes fejlesztési munkára aláírjuk a szerződést?

Ez alatt a fázis alatt elkészül a legtöbb felhasználói eset részletes specifikációja és a rendszer architektúrájának terve. A rendszer és annak architektúrája közötti összefüggés dominálja ezt a fázist. Az architektúrát, a RUP modell megalkotói által használt metafora szerint, egy bőrrel fedett csontváznak lehet tekinteni, amelyen ebben a fázisban csak minimális mennyiségő izom (programkód) van, csak annyira elegendő, hogy a főbb mozgásokat elvégezhesse a test. A rendszer az egész test, csontvázzal, izmokkal és bőrrel.

A rendszer architektúráját, mint a rendszer összes modelljének nézeteit foghatjuk fel. Ez azt jelenti, hogy van architektúrális nézete a felhasználói modellnek, az analízis modellnek, a dizájn modellnek, az implementációs modellnek és a futtatási környezeti modellnek. Az implementációs modell architektúrális nézetéhez tartoznak olyan komponensek melyek kimutatják, hogy az architektúra működőképes (futtatható). Ezért, ez alatt a fázis alatt a legkritikusabb felhasználói esetek azonosítása után, azok implementálása és tesztelése is megtörténik. Ez adja a fenti metaforának megfelelő működő vázát a szoftvernek.

A kidolgozás fázis végén a projektvezető olyan helyzetbe kerül, hogy képes az egész hátralévő fejlesztési munka megtervezésére és az ahhoz szükséges er_források felbecsülésére.

A fázison történő munkát mindaddig kell folytatni, míg a kellő stabilitás és a főbb kockázatok kezelhetősége nem biztosított.

Ennek a fázisnak az összeredménye a rendszer **architektúrális alapverziója**.

A RUP megvalósítás (konstrukció) - fázisa

A megvalósítás fázis alatt elkészül a termék – az izmok (a teljes programkód) ráépülnek a csontvázra (az architektúrára) és az architektúrális alapverzió kész rendszerré fejlődik. A kezdeti elképzelés egy termékké válik, amely hamarosan a teljes felhasználói társadalom elé kerül. Ez alatt a fázis alatt a fejlesztéshez szükséges erőforrások nagy részét felhasználja a projekt. A rendszer architektúrája stabil, de előfordulhat, hogy a fejlesztők felfedeznek jobb lehetőségeket a rendszer szerkezetének kialakítására, és így javasolhatnak kisebb architektúrális változtatásokat. A fázis végén az összes felhasználói eset, amelynek ebben a kiadásban történő implementálásáról megegyezett a megrendelő és a projektvezetés, implementálva van. A rendszer nem kell, hogy teljesen hibátlan legyen. Az átadás fázis alatt további hibák felfedezésre és kijavításra kerülhetnek.

Itt, a következő kérdés megválaszolása van a fókuszban:

- Megfelel a termék a felhasználói igényeknek olyannyira, hogy néhány kiválasztott felhasználó megkaphat egy korai kiadást (bétakiadást, „beta release”-t)?

A RUP átadás (átmenet) - fázisa

Az átadás-fázis ideje alatt a termék először egy bétakiadás formájába kerül. A bétakiadásban kisszámú gyakorlott felhasználó kipróbálja a terméket és jelenti az észlelt hibákat és hiányosságokat. A fejlesztők kijavítják a jelentett hibákat és beépítenek néhányat a javasolt javításokból az általános kiadásba, amely ezután a teljes felhasználói társadalom elé kerül. A fázis magába foglal olyan tevékenységeket, mint kézikönyvek és CD-lemezek gyártása, felhasználói személyzet képzése, segélyvonalon keresztüli támogatás létrehozása és fenntartása, és a kiadás után észlelt hibák kijavítása. A karbantartó csapat általában két részre osztja ezeket, a kiadás után észlelt hibákat: azokra, amelyek olyannyira befolyásolják a rendszer működését, hogy egy azonnali deltakiadás indokolt, és azokra, amelyek kijavítására a következő rendesen tervezett kiadásban kerül majd sor.

Néhány szakkifejezés:

- *Bétakiadás:* Egy korai kiadás, melyben kisszámú gyakorlott felhasználó kipróbálja a terméket és jelenti az észlelt hibákat és hiányosságokat.
- *Alfakiadás:* Egy felhasználó számára mértékre szabott szoftver els_ kiadása.
- *Delta:* Változtatás a felhasználói esetekben (követelményekben).
- *Deltakiadás:* Egy kiadás melyben egy delta implementálásra került.