UML

Unified Modelling Language. Ez egy grafikus módszer arra, hogy leírjunk egy szoftver rendszert. Ezt kétféleképpen valósíthatjuk meg vele. Vagy vázlatokat csinálunk, vagy pedig tervrajzokat.

Vázlatok esetén a fő cél az, hogy az ötleteinket kommunikálni tudjuk, akár olyan stakeholdereknek, akik nem értenek a szoftver rendszerekhez. Nagy vonalakban tudunk megjeleníteni megközelítéseket és több megoldást is meg tudunk mutatni nekik. Nem akarunk specifikusak lenni.

Tervrajzok viszont kiemelkedően részletesek és tartalmaznak mindenféle tervezési döntést. Ezeket a fejlesztők sokszor direktben át tudják fordítani kóddá. Ezt kétféleképpen tudjuk megtenni, vagy platformtól (technológia, oprendszer, programozási nyelv, vendor stb) függetlenül vagy platformfüggően.

Tervezésben két utat követhetünk:

* Waterfall: analízis, design, kódolás, tesztelés
* Iteratív: darabokra szedjük a rendszert az elejétől fogva és ezekre egyesével a fenti szekvenciát végigcsináljuk

A döntő tényező az, hogy mekkora biztonsággal tudjuk kijelenteni, hogy nem fognak változni a követelmények, körülmények és minden fejlesztéshez szükséges ismeretünk megvan a project elején.

Az UML (a termék tervezése) szempontjából ez azt eredményezi, hogy két megközelítés közül választhatunk:

* Predictive planning
* Adaptive planning (~agilis)

# Típusok

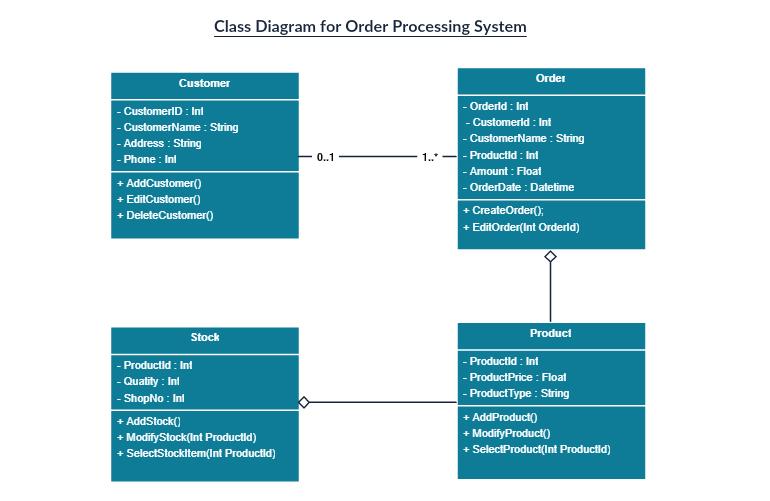
# C:\Users\ajonas\Desktop\Oktatás\2023_JSZ07\19_nap\UML-Diagram-types-1-1024x658.png

A **struktúra** **diagramok** azt mutatják, hogyan néz ki a modellezett rendszer. Műszaki szempontból különböző objektumokat mutatnak be a rendszerben. A **viselkedési** **diagramok** azt mutatják, hogy mi történik a rendszerben. Leírják, hogyan vannak kölcsönhatásban egymással az objektumok, hogy egy működő rendszert hozzanak létre.

## Class diagram

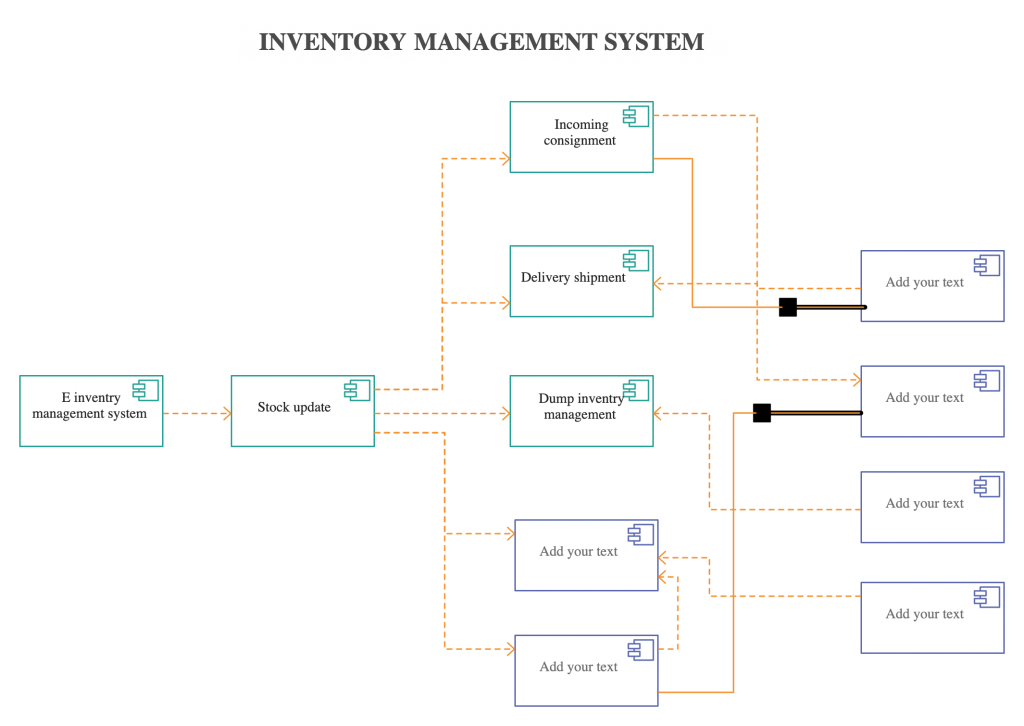
Az osztály diagramok az objektum orientált tervek fő építőkövei. Megmutatják a rendszerben lévő osztályokat, az osztályok attribútumait és műveleteit, valamint az osztályok közötti kapcsolatokat.

A legtöbb modellező rendszerben egy osztálynak három része van. A név a tetején, az attribútumok a közepén, és a műveletek vagy metódusok az alján helyezkednek el. Egy nagy rendszerben, ahol sok kapcsolódó osztály van, az osztályokat csoportosítják osztálydiagramok létrehozásához. A különböző osztályok közötti kapcsolatok különböző típusú nyilak segítségével vannak ábrázolva.



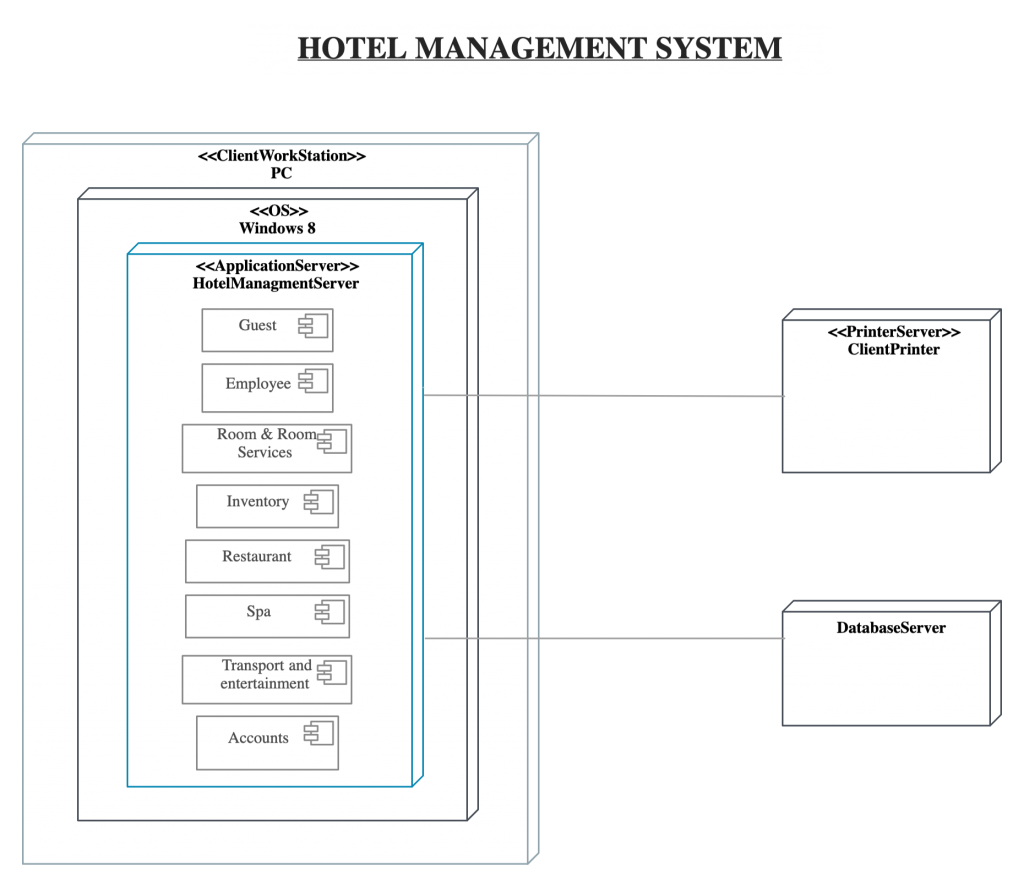
## Component diagram

A komponens diagram megjeleníti a szoftverrendszer komponenseit és egymással való kapcsolatát. Ezeket általában akkor használják, amikor komplex rendszerekkel dolgoznak, amelyekben sok komponens található. A komponensek interfészek segítségével kommunikálnak egymással. Az interfészeket csatlakozók segítségével kötik össze.



## Deployment diagram

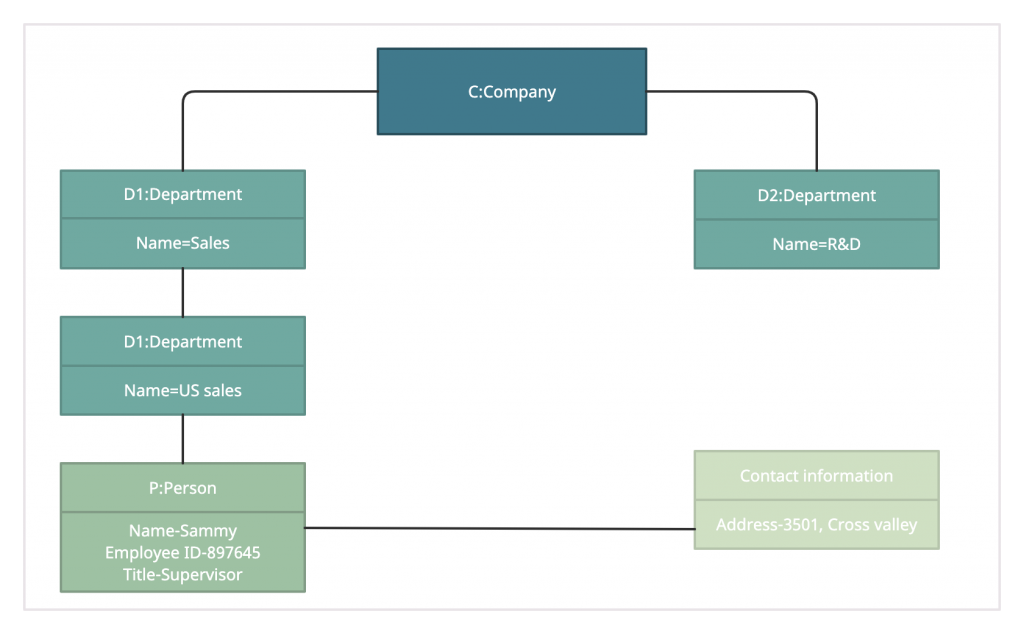
A deployment diagram a rendszer hardverét és a hardveren lévő szoftvert mutatja. A telepítési diagramok hasznosak, amikor a szoftvermegoldás több gépen van telepítve, mindegyik egyedi konfigurációval rendelkezik.



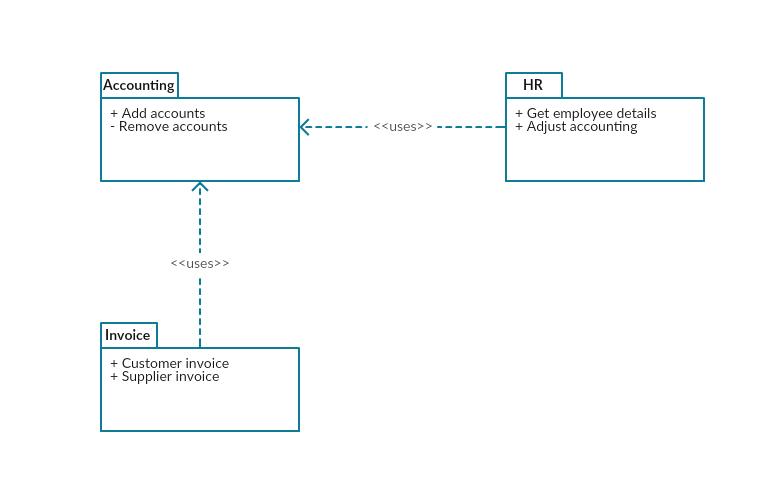
## Object diagram

Az objektumdiagramok, néha példánydiagramoknak is nevezve, nagyon hasonlóak az osztálydiagramokhoz. Mint az osztálydiagramok, az objektumdiagramok is mutatják az objektumok közötti kapcsolatot, de valós világbeli példákat használnak.

Megmutatják, hogy egy adott időpontban hogyan fog kinézni egy rendszer. Mivel az objektumokban adatok állnak rendelkezésre, ezeket az ábrákat használják objektumok közötti bonyolult kapcsolatok magyarázatára.

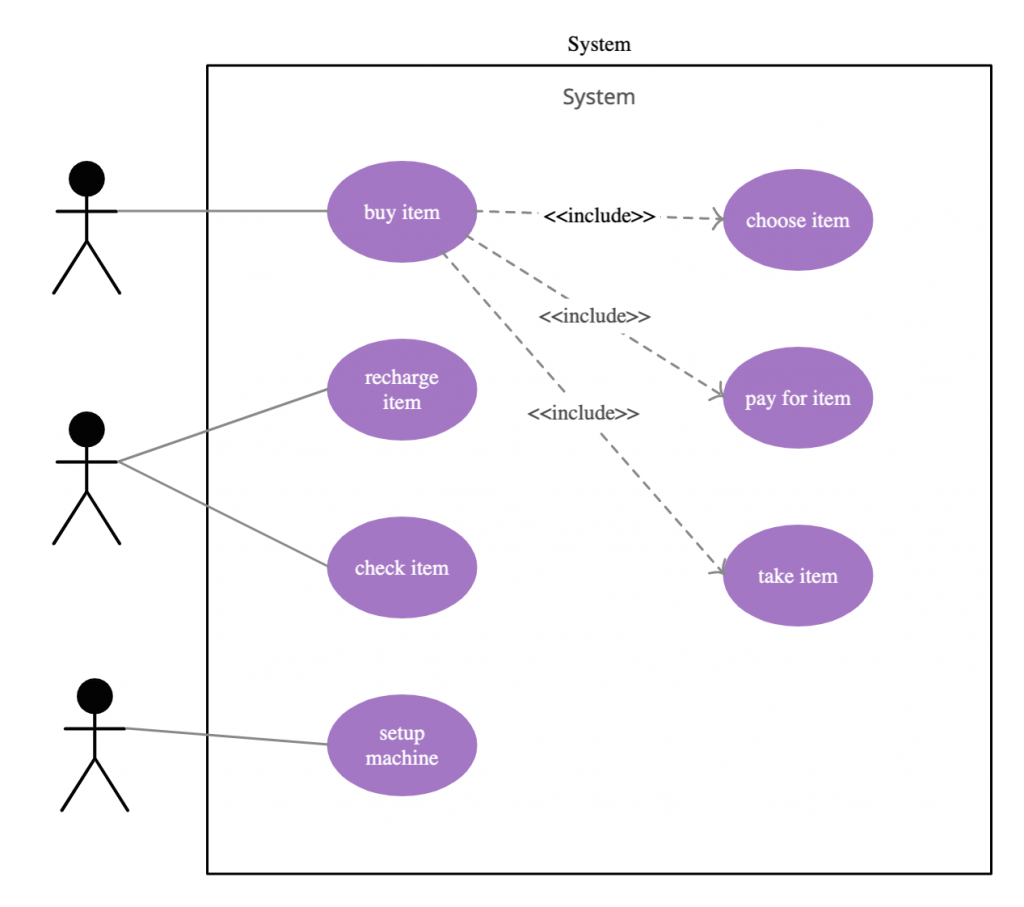


## Package diagram

Mint a neve is sugallja, a csomag diagramok bemutatják a különböző csomagok közötti függőségeket egy rendszerben. 

## Use case diagram

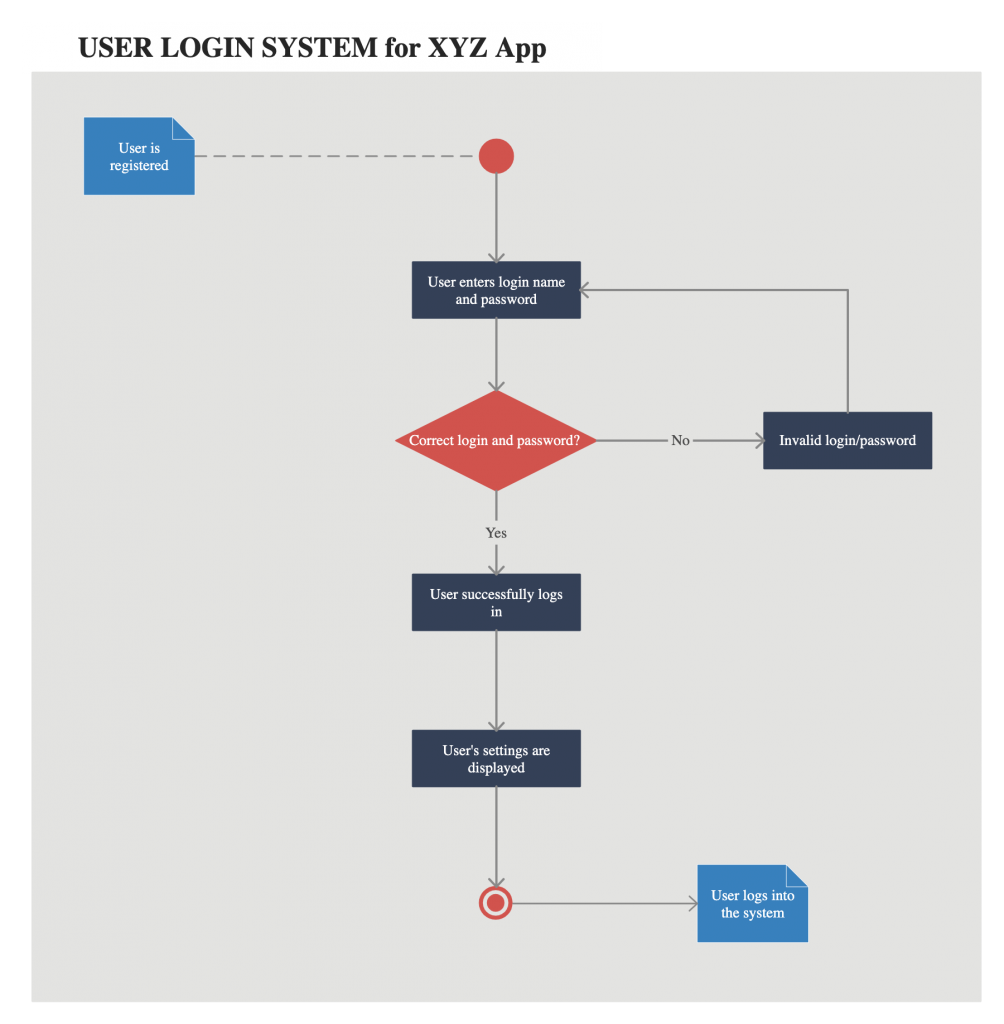
A viselkedési UML típusok közül a legismertebb diagramtípus, a használati eset diagramok grafikus áttekintést adnak a rendszerben résztvevő szereplőkről, a szereplők működéséhez szükséges különböző funkciókról és a különböző funkciók közötti interakciókról.

Ez kiváló kiindulópont a projekt megbeszéléséhez, mert könnyen azonosíthatók a fő szereplők és a rendszer fő folyamatai. 

## Activity diagrams

Az aktivitás diagramok grafikus módon ábrázolják a munkafolyamatokat. Használhatók a vállalati folyamat vagy a rendszer egyes komponenseinek működési folyamatainak (workflow) leírására. Az aktivitás diagramok néha alternatívájaként használatosak az állapotgép diagramoknak is.

Az aktivitás egy paraméterezhető viselkedés, amit koordinált cselekvéssorozatként reprezentálunk.



### Activitás

A végrehajtás folyamatát aktivitás csomópontok és aktivitás élek által modellezzük. Egy csomópont lehet egy alárendelt viselkedés végrehajtása, például egy számítási művelet, egy művelet hívása vagy objektum manipulációja. Az aktivitás csomópontok tartalmazzák a vezérlési konstrukciók folyamatát is, például a szinkronizációt, a döntést és a konkurens irányítást. Az aktivitások hívási hierarchiákat képezhetnek, más aktivitásokat hívva, végül eljutva az egyedi cselekvésekhez. Egy objektumorientált modellben az aktivitások általában közvetett módon vannak meghívva olyan metódusokként, amelyek a közvetlenül meghívott műveletekhez vannak kötve.

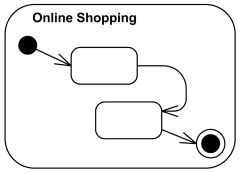
Az aktivitás tartalmazhat aktivitás csomópontokat, amelyek lehetnek:

* akciók
* objektumok
* vezérlési elemek

Az aktivitások különböző fajtájú cselekvéseket is tartalmazhatnak:

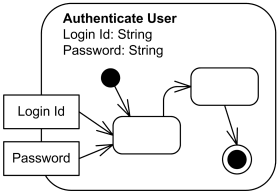
* Primitív funkciók előfordulásai, például számítási funkciók.
* Viselkedés invokációi, például aktivitások.
* Kommunikációs cselekvések, például jelzések küldése.
* Objektumok manipulációi, például attribútumok vagy kapcsolatok olvasása vagy írása.

Az aktivitást kerek sarkú téglalapként ábrázolhatjuk, az aktivitás nevével a bal felső sarokban, és az aktivitás csomópontjaival és éleivel a keret belsejében. Az UML 2.4 specifikáció példái az aktivitás nevét vastagon mutatják be.



Az aktivitás paraméterei a kereten jelennek meg, és az aktivitás neve alatt vannak felsorolva, így:

paraméter-név: paraméter-típus.



Mint viselkedés, az aktivitásnak lehet elő- és utófeltétel megszorításai. Ha jelen vannak, ezeket a "precondition" és "postcondition" kulcsszavakkal mutatják be.

A kerek sarkú aktivitási keretet keret jelölés is helyettesítheti diagramokban. A keret típusa ilyenkor "activity" vagy röviden "act" lehet. Az aktivitás paramétereket, ha vannak, a kereten tüntetjük fel.



Az osztályokhoz (kódbeli) tartozó "activity" kulcsszóval a reflektív aktivitások jellemzőit is mutathatjuk, jelezve, hogy az egy akcióval rendelkező osztály. Az asszociáció és állapotgép jelölés is használható szükség esetén.

### Aktivitás partíció

Az aktivitás partíció olyan aktivitás csoport, amelyeknek van valamilyen közös jellemzője.

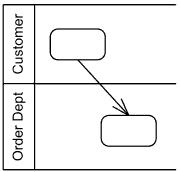
A partíciók szűkített nézetet biztosítanak az aktivitásokban hívott viselkedésekre. A megszorításokat az elem típusának megfelelően választhatják ki, amelyet a partíció képvisel. Az alábbi megszorítások normatívak (szabványosak) az UML 2.4-ben:

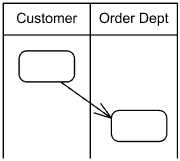
* classifier
* instance
* part
* attribute and value

Például a partíciók konkrét osztályokat is képviselhetnek. Ebben az esetben a partíciókban lévő műveleteknek olyan műveleteknek kell lenniük, amelyek objektumokat céloznak meg, és ezek az objektumok a megfelelő osztály példányai.

Egy partíció reprezentálhat egy attribútumot és a részpartíciókat - az attribútum konkrét értékeit. Például egy partíció reprezentálhatja azt a helyet, ahol egy viselkedés végrehajtódik, és a részpartíciók a hely konkrét értékeit mutathatják be, például New York-ot.

Az aktivitás partíciót úszóvonal jelöléssel is ábrázolhatják - két, általában párhuzamos vonal, legyen az vízszintes vagy függőleges, és egy név, ami jelöli a partíciót egy dobozban az egyik végén. Az összes aktivitás csomópontot, például műveleteket és éleket, amelyek ezek között a vonalak között vannak, úgy tekintjük, hogy a partícióban találhatók.



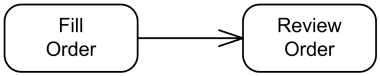


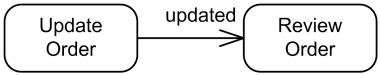
### Aktivitás élek

Az aktivitás él egy eszköz az irányított kapcsolatok megjelenítésére. Rajtuk keresztül tokenek vagy adatobjektumok áramlanak az aktivitási csomópontok között. Ezek példányai a vezérlési élek és az objektumáramlási élek is. Az él forrása és célja ugyanabban az aktivitásban kell, hogy legyen.

Az aktivitás élet egy nyitott nyílvégű vonallal jelöljük, amely két aktivitási csomópontot köt össze.

Az élek elnevezhetők, azonban az éleknek nem kell egyedülálló nevűnek lenniük egy aktivitásban.





Az aktivitás élnek lehet guard-ja futásidőben kiértékelhető megszorítása, ami meghatározza, hogy az él átjárható-e. Az guardnak minden tokennél igaznak kell lennie, amelyet az él mentén kínálnak.

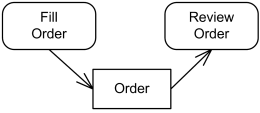
Az aktivitás él őrzését szögletes zárójelben mutatjuk be.



#### Objektum élek

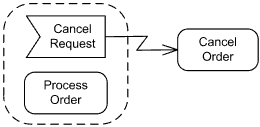
Az objektumáramlási élek olyan aktivitás élek, amelyek az objektum- és adattokenek adatáramlását mutatják be az aktivitási csomópontok között.

Az objektumáramlást nyilas vonallal jelöljük.



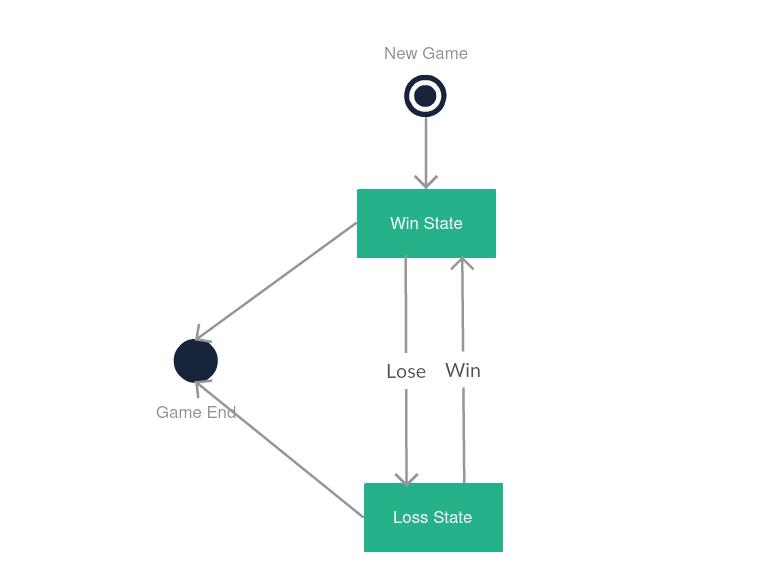
#### Megszakító élek

Az "interrupting edge" olyan aktivitás él, amely megszakítást kifejező aktivitás él, és villámként ábrázoljuk.



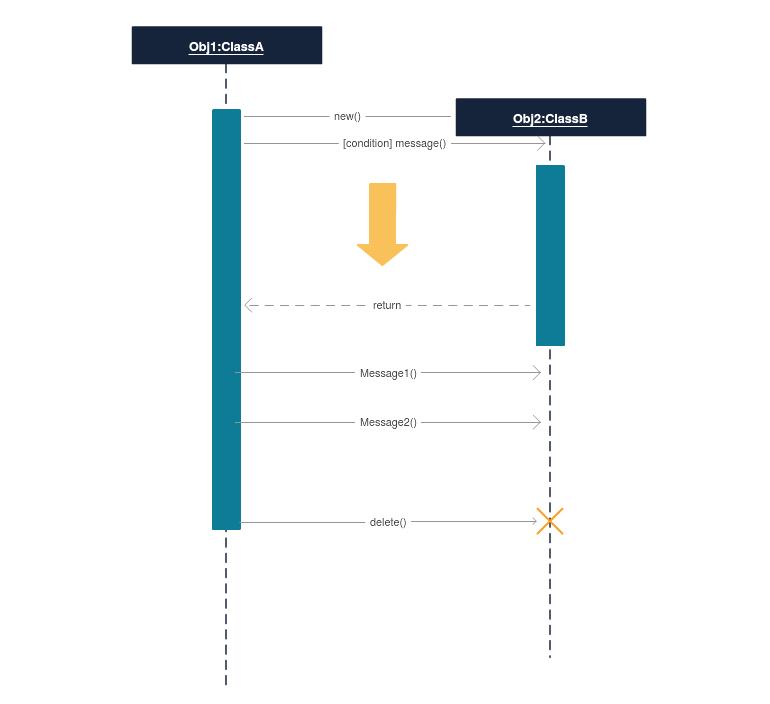
## State machine diagrams

Az állapotgép diagramok hasonlóak az aktivitás diagramokhoz, bár a jelölések és a használat kissé eltérő. Ezek néha állapot diagramokként vagy állapotgép diagramokként is ismertek. Nagyon hasznosak azoknak az objektumoknak a viselkedésének leírására, amelyek különböző állapotokban különbözően viselkednek.



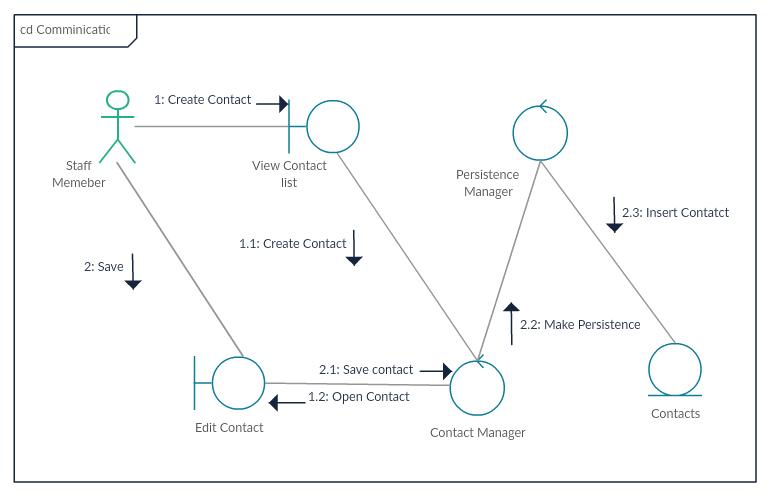
## Sequence diagrams

A szekvencia diagramok az UML-ben bemutatják, hogy az objektumok hogyan lépnek kölcsönhatásba egymással és az interakciók milyen sorrendben következnek be. Fontos megjegyezni, hogy ezek az interakciók egy adott forgatókönyvre vonatkoznak. A folyamatok függőlegesen vannak ábrázolva, és az interakciók nyilakként jelennek meg.



## Communication diagram

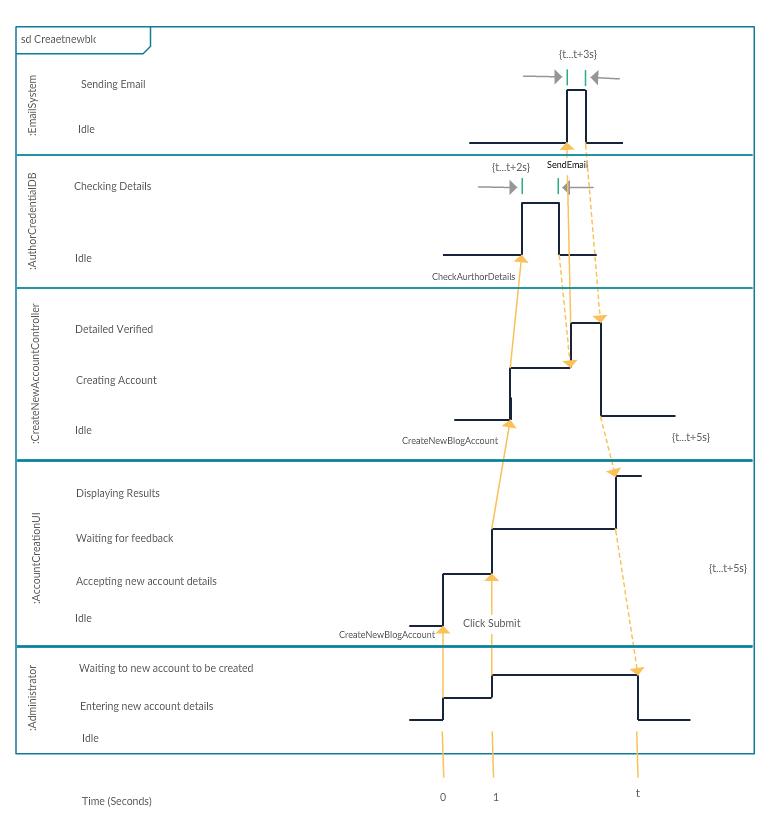
A kommunikációs diagramok hasonlóak a szekvencia diagramokhoz, de az üzenetekre helyezik a hangsúlyt, amelyeket az objektumok közvetítenek egymás között. Ugyanazt az információt szekvenciadiagram segítségével is lehet ábrázolni különböző eszközökkel. Az UML 1-ben ezeket együttműködési diagramoknak nevezték.



Az interakció áttekintő diagramok nagyon hasonlóak az aktivitás diagramokhoz. Míg az aktivitás diagramok egy sor folyamatot mutatnak, az interakció áttekintő diagramok egy sor kommunikációs diagramot mutatnak be.

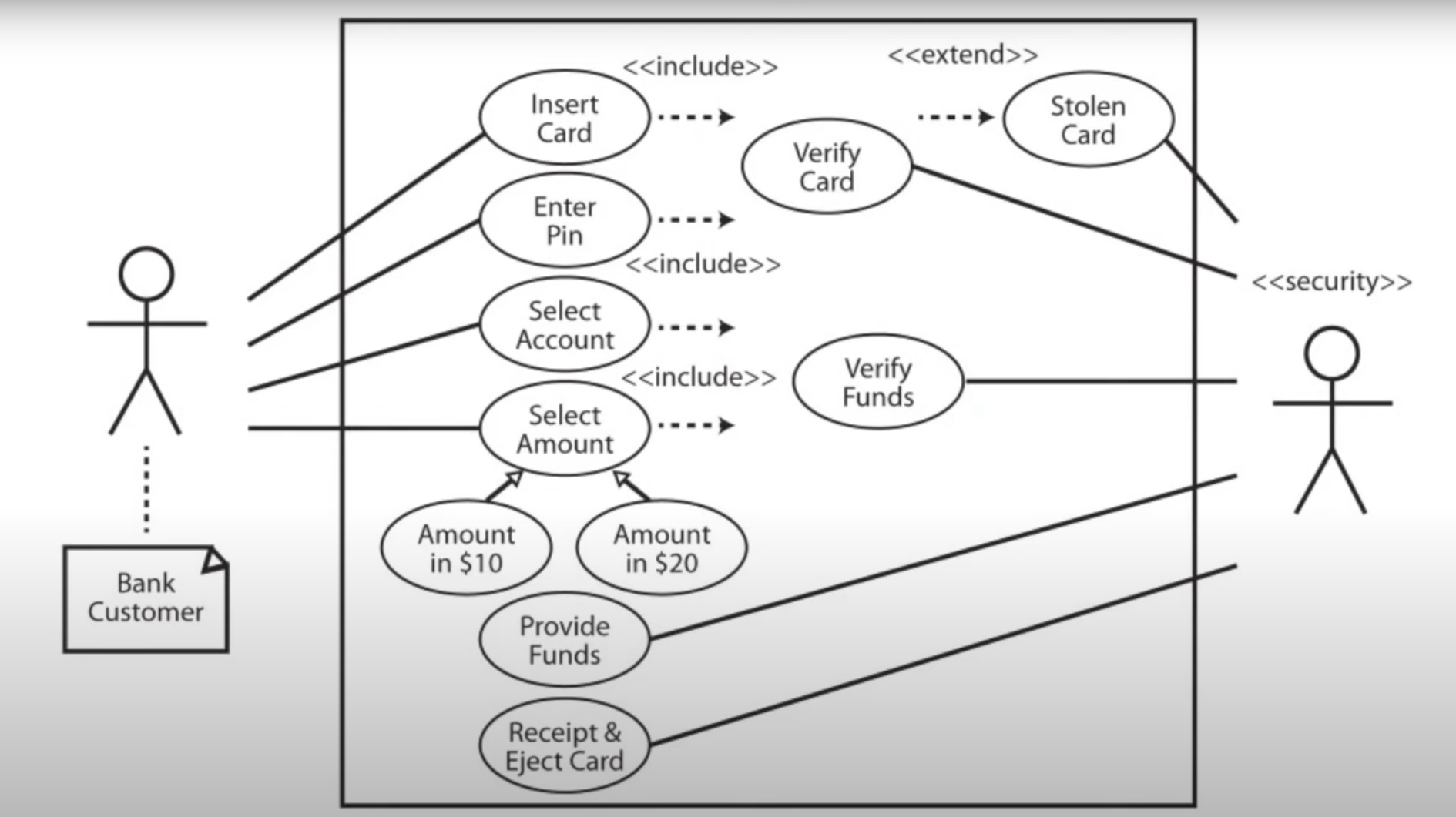
## Timing diagram

Az időzítési diagramok nagyon hasonlítanak a szekvenciadiagramokhoz. Az objektumok viselkedését ábrázolják egy adott időkeretben. Ha csak egy objektum van, a diagram egyszerű. De ha több objektum is részt vesz, az időzítési diagramot használják az objektumok közötti interakciók bemutatására ebben az időkeretben.



# Példa – use case

Ezekkel írjuk le, hogy a termékünk hogy fog valami problémát megoldani, vagy szolgáltatást nyújtani. Ezért írjuk le, hogy mik az elvárásaink a termékkel kapcsolatban és hogy ezeket hogy fogjuk megvalósítani. Itt, mint sok másik diagramban sem kódolunk. Vannak „shall” és vannak „should” requirement-jeink. Konyhanyelven a shall követelményeinket meg kell valósítanunk, a should követelményeket pedig megvalósíthatjuk.



Elemek:

* stereotype: leírja egy elem szerepét (pl a pálcikaemberét)
* actor:
* guillemet: ha meg akarunk nevezni valamit, de kevés a hely
* note: ide lehet kommentelni
* task: use case-ek, ezek vannak bekarikázva
* communication line: ez jelzi, ha agy actor használhat egy use case-t
* system boundary line: ez választja el a rendszert az actoroktól
* include: 2 vagy több elem használ egy use case-t
* extend: amikor egy use case opcionális
* nyíl: általános use case specializációját jelenti (érdemes figyelni ezek, meg a szaggatott nyilak irányát)

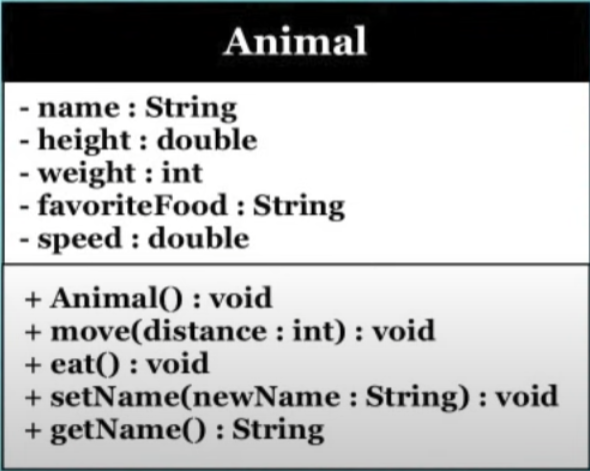
További elemei a leírásnak:

* description
* trigger
* actors
* preconditions
* goals (successful conclusions)
* failed conclusions
  + Invalid PIN
  + Napi limit feletti pénzfelvétel kísérlet
* extensions (alternatives)
  + 3 hibás PIN
  + Lopott kártya
* steps of execution

# Class diagram

Lényeges elemek:

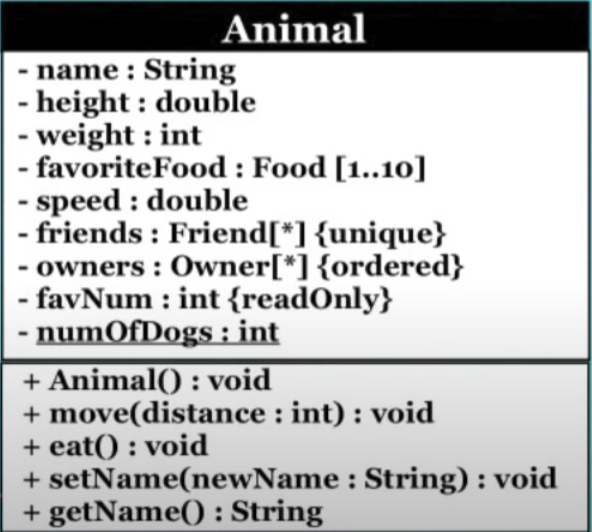
* Class
* Abstraction
* Encapsulation
* Class diagrams



A leírásban lényeges elemek:

* visibility: a + vagy – jelek (public: +, protected: #, package default: ~, private: -)
* name: pl Animal()
* parameters: pl distance: int
* return type
* type: pl String

### Számosságok

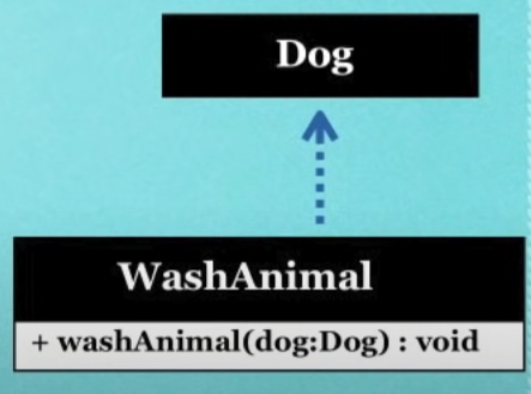


A leírásban lényeges elemek:

* Fix számosság: [1…10]
* Ismeretlen szám: [\*]
* {unique}, {notUnique}, {ordered}, {readOnly}
* Static attribute: ami alá van húzva

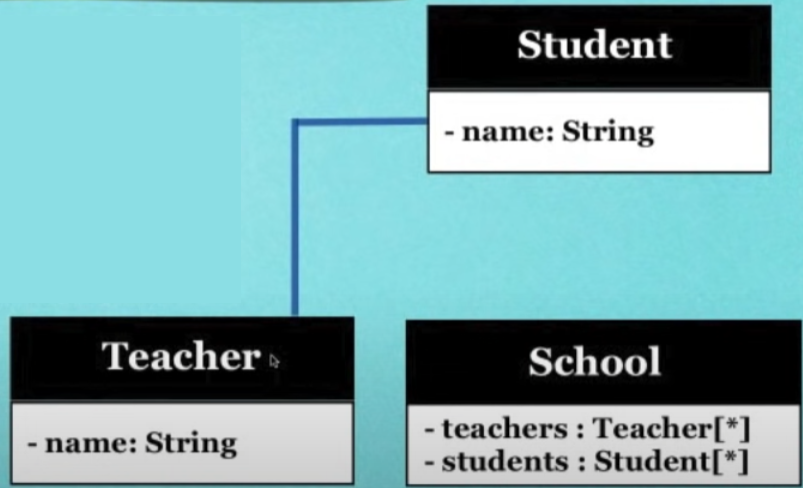
### Függőségek (coupling)

Dependency:



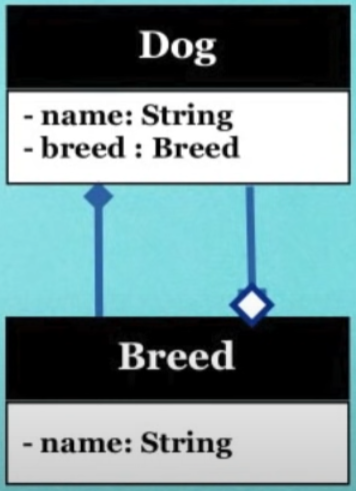
Itt azt jelöljük, hogy a WashAnimal class használja a Dog osztályt. A nyíl jelöli a függőséget. Miután sok függőség szokott lenni osztályok között, ezeket gyakran nem jelöljük az ábráinkon.

Association:



Itt azt jelöljük, hogy a Student és a Teacher osztályoknak közül van egymáshoz. Együtt tagjai a School osztálynak és logikailag is kapcsolatban vannak egymással. Miután sok asszociáció szokott lenni osztályok között, ezeket gyakran nem jelöljük az ábráinkon.

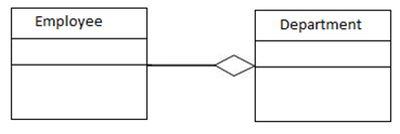
Aggregáció és kompozíció:



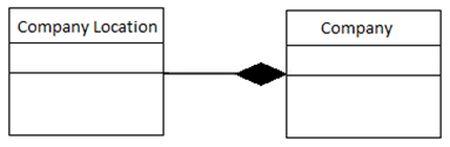
A Dog és Breed aggregációs kapcsolatban állnak egymással (tele rombuszos nyíl). Ilyenkor a Dog osztálynak van Breed része. Több különböző osztály alkothat egy Dog osztályt például.

A Dog és Breed kompozíciós kapcsolatban állnak egymással (üres rombuszos nyíl). Ilyenkor a Dog egy része a Breed osztály.

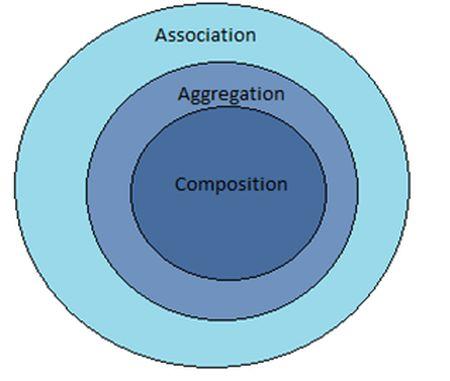
A különbség az életciklusokban van:



Itt a két objektum élete független egymástól. Ez azt jelenti, hogy ebben az egyesítésben (összetett objektum) az objektumoknak saját életciklusuk van. Az alkalmazottak létezhetnek osztály nélkül.

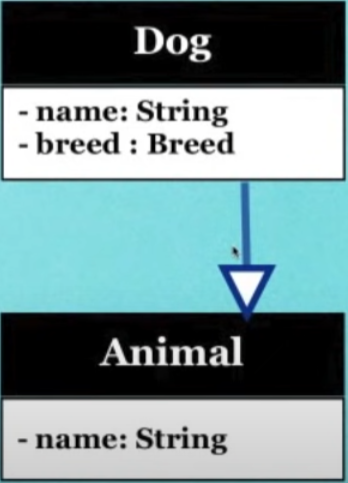


Itt viszont a két objektum élete nem független egymástól. A vállalati helyszín objektum élete a vállalat objektum életétől függ. A vállalat objektum felelős a vállalati helyszín objektumok létrehozásáért és megsemmisítéséért.

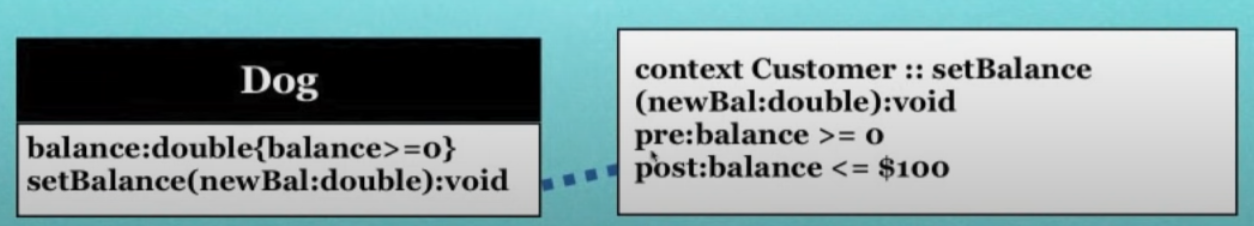


Ezek határozzák meg (többek között) azt, hogy egy osztálynak a változtatása mennyire kényszeríti ki egy másik osztály változtatását.

### Inheritence:



### Constraints



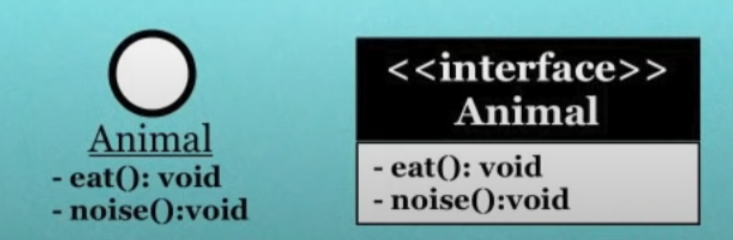
* pre: preconditions, ezeknek igaznak kell lennie, mielőtt a függvényt meghívjuk
* post: postconditions, ezeknek igaznak kell lennie, miután a függvényt meghívjuk
* Data Types: Boolean, Integer, Real, String
* Arithmetic: +, -, \*, /, a->mod(b), abs(), min(), max()
* Comparison: <, >, <=, >=, =, <>
* Boolean: and, or, xor, not

### Abstract classes and interfaces

(valószínűleg rossz irányba mutat a nyíl)

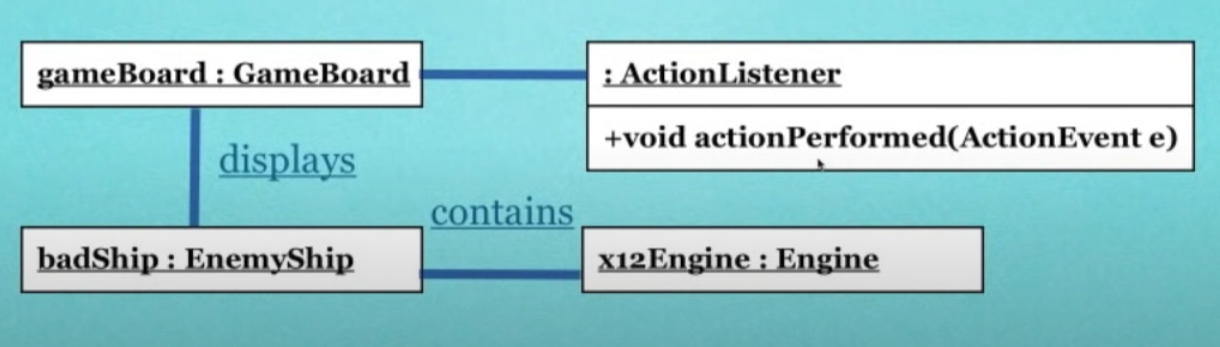


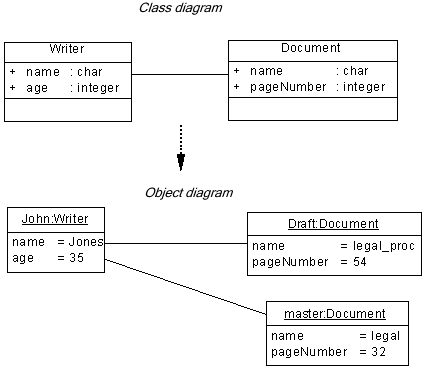
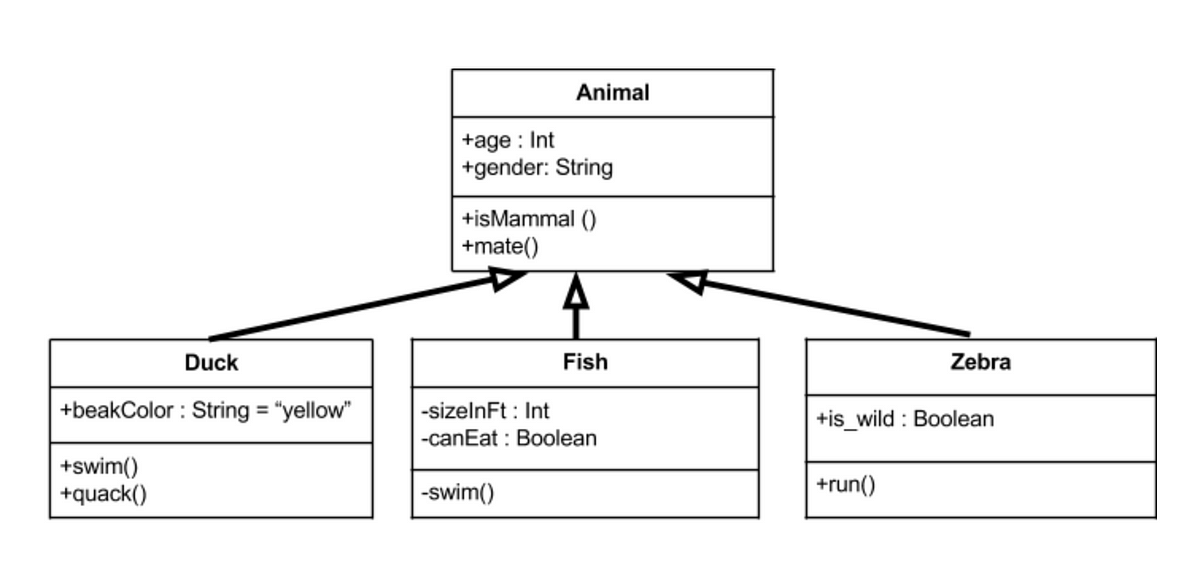
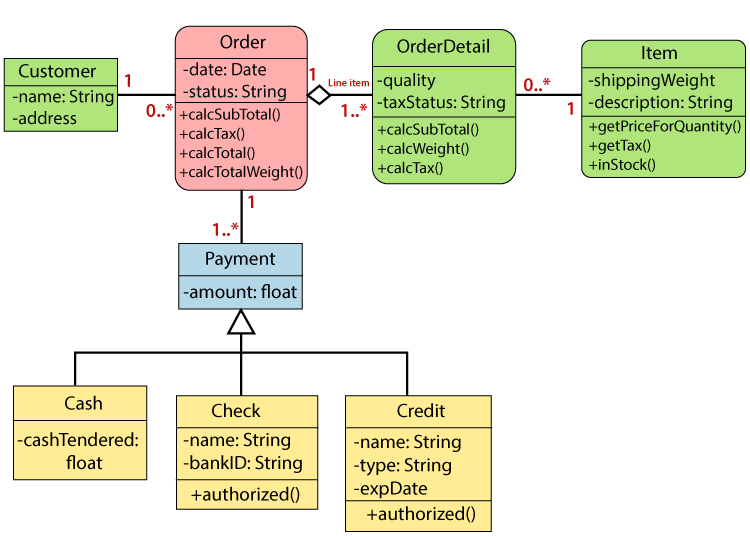
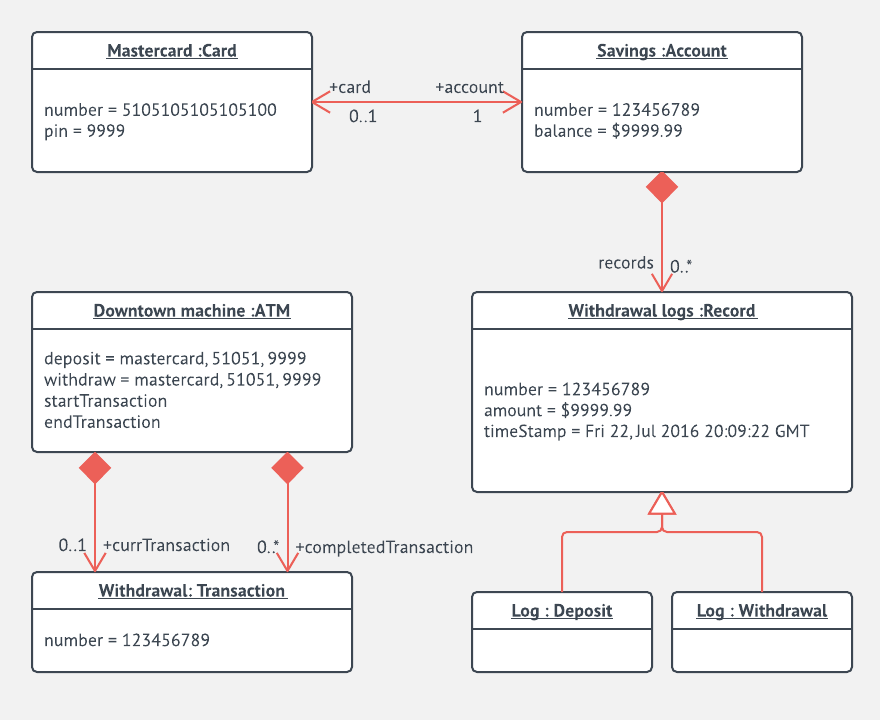
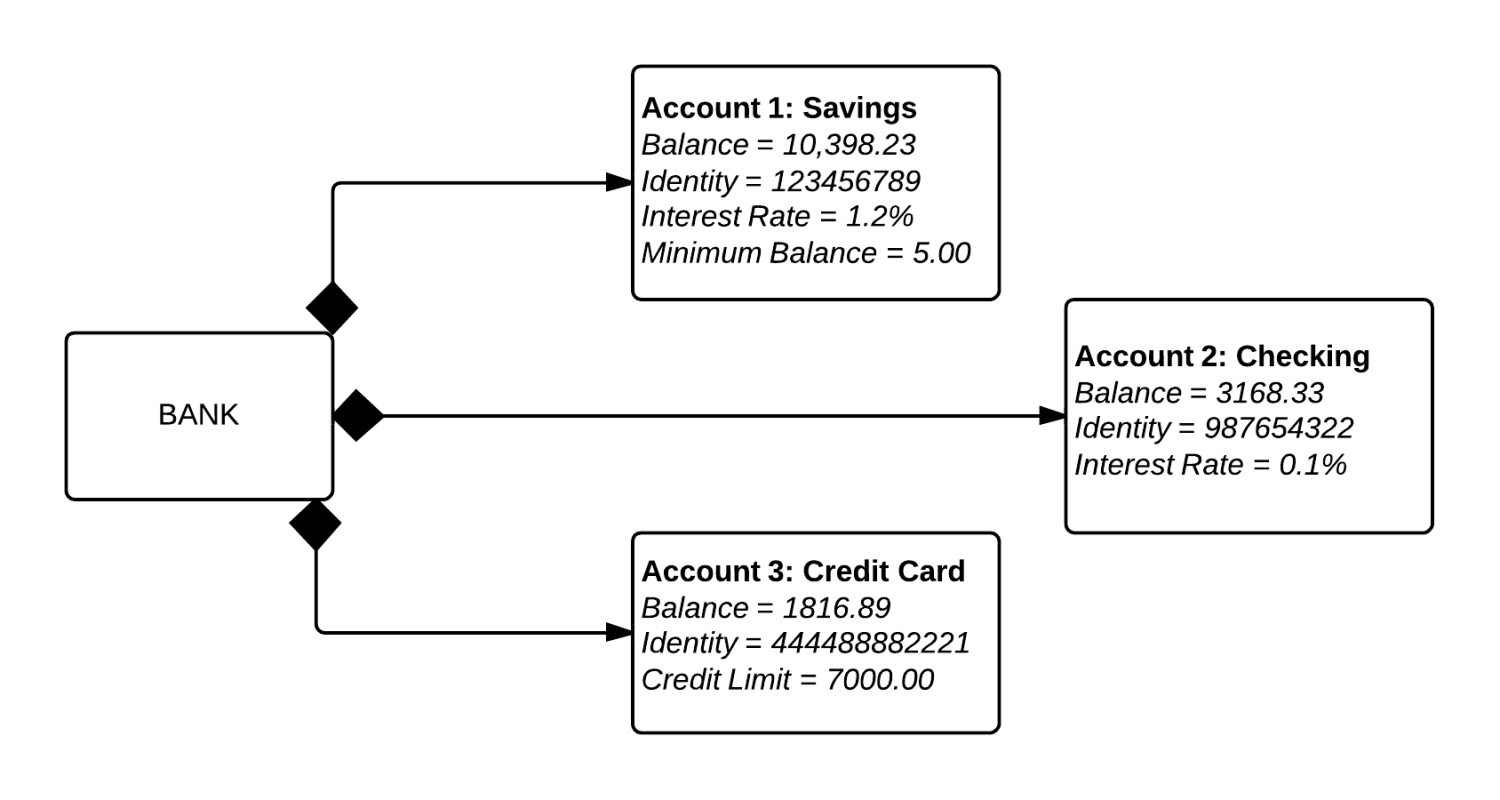
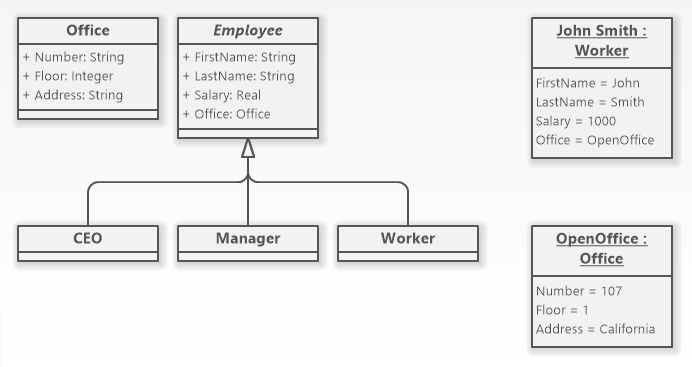
Emlékeztető, absztrakt osztályoknak vannak absztrakt függvényei. Ezeket jelöljük dőlt betűvel. Az implementált függvényeket normál betűvel jelöljük.



Emlékeztető, interface osztályoknak minden függvénye absztrakt. A tagok vagy statikusak, vagy konstansok.

# Object diagram





# Sequence diagram



Ez a típus modellezi, hogy milyen interakciók történnek a termékben és milyen sorrendben. Azt írjuk le, hogy milyen interakciók történnek és mikor, különös figyelemmel ezek sorrendjére.

Fogalmak:

* Participant:
  + egy sorban vagy amikor létre vannak hozva
  + lehet objektum vagy más része a rendszernek
  + név:osztály formában jelöljök
  + lehet névtelen is, pl egy action listener
  + Create participant – ezzel folyamat közben létre lehet hozni
  + Destroy participant – ezzel folyamat közben el lehet pusztítani
* Lifeline
  + ezek lefelé mennek
  + az interakciók az ezeket összekötő elemek
  + a sorrend a fontos és nem az eltelt idő (de lehet időt is jelölni)
* Message
  + event, ami egy callertől a reciever felé megy
  + a nyílnak lehet description-je vagy message signature-je
  + ebben lehet paraméter (mint egy hívás)
  + synchronous, ha a sender vár a válaszra – tele nyíl
  + asynchronous, ha a sender nem vár a válaszra – üres nyíl, ilyenkor X-el jelöljük az addigi helyét
  + message to self akár
  + return message
* Sequence fragment
  + Dobozok, amik interakciókat foglalnak magukban
  + Bal felső sorokban tudjuk megjelölni a típust:
    - Optional: csak akkor, ha a guard true-ra lett kiértékelve
    - Negative: ezeket (jelenleg) soha nem akarjuk végrehajtani
    - Alternative: elágazás
    - Assert: sikerül, vagy exception
    - Loop: pl guard condition-nal, vagy előre meghatározott mennyiséget
      * Break
    - Referenced: ezzel lehet beemelni egy másik részletet (ha nincs hely pl, vagy többször akarunk valamit felhasználni)
    - Parallel: vagy sorrendben, vagy nem
* Defining time constraints: ezeket a nyilakat nem vízszintesen húzzuk
* Defines lost message: mi történik, ha nem érkezik meg egy üzenet
* Defines found message: mi történik, ha egy korábban küldött üzenet érkezik meg
* Nested messages: amikor az üzenet a fogadó féltől egy vagy több üzenetet eredményez
* Critical element protected from use: ami benne van, más nem férhet hozzá addig, amíg a definiált események végbe nem mennek