**Szoftver tervezés - licensz vizsga 2021**

**Témák:**

1. Szoftver projekt fejlesztés lépései.
2. Követelmény specifikáció.
3. UML diagramok. Használati eset diagram (dinamikus kép). Osztály diagram (statikus kép).
4. Architekturális minták. Model-View-Controller architektúra. Előnyök és hátrányok.
5. Tervezési minták. Összetétel (Composite), Egyke (Singleton), Megfigyelő (Observer) minták.

**Kidolgozás:**

**1. Szoftver projekt fejlesztés lépései.**

Ezen lépések azért lettek kitalálva és feljegyezve, hogy megkönnyítsék a programozók, fejlesztők dolgát, olyan fontos fázisokat foglal magába, amelyek elengedhetetlenek számukra. Az idő elteltével több ilyen modell jött létre, elég csak a spirál, szökőkút, vagy a vízesés modellre gondolni, mindközül az utóbbi a legelterjettebb és legfontosabb, ez abból áll, hogy az egyik fázis végkimenetele a másik kezdete.

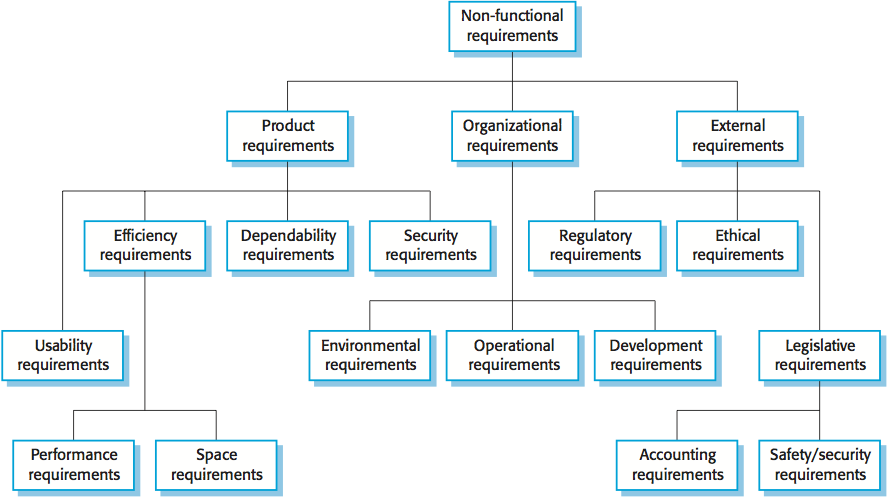
**Fázisok:**

1. **Előzetes tervezés:** Felfedezni a szervezet céljait, problémáit. Ezt követően számos megoldás jöhet létre a problémák orvoslására. Költségek kielemzése és kiírása, amelyek a céget súlythatják a fejlesztés során.
2. **Rendszerezés, követelmények meghatározása:** Itt konkrét tervekkel áll elő a programozói cég, ajánlást tesz a problémák megoldására. Tényket gyűjt (*felhasználói követelmények beszerzése* dokumentációval, ügyfélinterjúk stb.), a meglévő rendszert vizsgálja (előnyök hátrányok), javasolt rendszert elemez (megoldást talál a hátrányokra, hiányosságokra).
3. **Tervezés:** Ebben a lépésben a kívánt funkciókat és műveleteket részletesen ismertetjük, beleértve a frontend és a backend részt. (UML, pénzügyi tervek stb) Kik vannak benne? A projektvezető. A fejlesztő nincs benne általában.
4. **Implementáció**: Itt történik a kódolás. Követelmény specifikáció alapján kódolni. Tesztelés (egyesek külön fázisként teszik). Unit tesztekre gondolunk, nem adjuk ki a kezünkből a szoftvert és akkor teszteljük. Még lehet javítani rajta.
5. **Karbantartás:** Karbantartjuk a meglévő rendszert, hogy az ne tudjon elavulni a konkurencia mellett. Tesztelés itt is lehet, a kiadott programot teszteljük és javítjuk a bugokat.

2.  **Követelmény specifikáció**

Ennek tudatában tervezzük meg, majd építjük fel a rendszert. A szoftverek tervezése során nehéz megszabni azt, hogy pontosan hogyan kellene működnie a rendszernek, meg hogyan kellene kinéznie a UI-nak, ehhez jönnek segítségül a követelmények. A követelmények tervezése az a folyamat, amikor meghatározzuk és elemezzük a riválisokat, hasonló rendszereket, valamint a megszorításokat, azt követően ellenőrizzük és ledokumentáljuk azokat. A követelmény, mint fogalom nagyon tág jelentéssel bír az IT-ban, a tárgyalását általában két részre bontják: felhasználói, illetve rendszer követelményekre.

1. **Felhasználói követelmények:** A felhasználói követelmények technikai értelemben magas szintű dokumentálás diagrammokkal kiegészítve, amely elsősorban azoknak a személyeknek készül, akik nincsenek tisztában a rendszer kivitelezésével, fekete dobozként tekintenek a rendszerre és nem érdekli őket a beljeseje csak a működése, nem rendelkeznek részletes háttérinformációval, technikai ismeretekkel a rendszerről. Következtetésképpen elsősorban a felhasználóknak, az ügyfeleknek, illetve ezeket képviselő menedzsereknek készül.
2. **Rendszer követelmények:** A rendszerkövetelmények tartalmazzák a szoftver funkciók és megszorítások leírását és azt, hogy mit kell leprogramozni a rendszerbe. A rendszerkövetelményeknek két típusát különböztetjük meg: funkcionális és nem-funkcionális követelmények.
   1. **Funkcionális követelmények:** A funkcionális követelmények tartalmazzák a leírást arról, hogy a rendszernek milyen funkciókkal kell rendelkeznie és ezek a funkciók hogyan kellene működjenek. Milyen inputra milyen output érkezik, technikai oldalról vizsgálódva.
   2. **Nem funkcionális követelmények:** A nem-funkcionális követelmények tartalmazzák a rendszer megszorításokat és a működési feltételeket. Ezen követelményekben a fejlesztésre vonatkozó leírásokat is megtaláljuk. Ez a rész drasztikusabb az előzőnél, ugyanis egy rosszul, vagy egyáltalán nem működő rendszert kapunk abban az esetben, amennyiben a feltételek nem teljesülnek. Ezen rész három főbb komponensre osztható: termék, szervezetei és külső követelményekre.
      1. **Termékhez kötött követelmények:** olyan dolgok tartoznak ide, mint például a teljesítmény, security, használhatóság stb.
      2. **Szervezethez kötött:** Környezeti, szervezeti, harcképességi stb.
      3. **Külső körülmények:** Etikai, biztonsági stb.



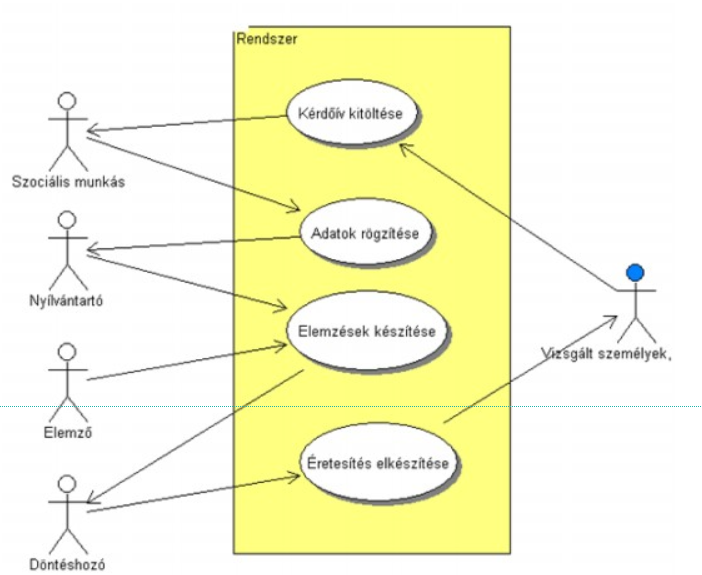
3. **UML diagramok. Használati eset diagram (dinamikus kép). Osztály diagram (statikus kép).**

Az UML diagramok szoftver modellezéshez vannak kötve, hogy azok könnyebben implementálhatóak legyenek. Két nagy csoportra osztjuk ezeket: strukturális, illetve viselkedés diagrammokra. Az előző csoportba tartozik például az osztály diagram, a komponens diagram, az objektum diagram, a másodikba a Use-Case diagram, az aktivitási diagram, a szekvencia diagram, az idődiagram.

Ezek közül az osztály, illetve a Use-Case diagram a leggyakoribb.

**A Használati eset (Use-Case) diagram:** Ezen megközelítés az egyik leggyakoribb és leghasználtabb az UML diagram típusok közül, bemutatja a rendszert használó alanyokat, felhasználókat és azok szemszögéből mutatja be a rendszert. A különféle státusszal rendelkező szereplők számára elérhető funkcionalitásokat, illetve a kölcsönhatásokat is megmutatja. Remek kiindulópont lehet minden projekt esetében, ugyanis könnyen be lehet azonosítani általa a fontosabb szereplőket és a rendszer fő folyamatait. Alany állítmány jellege van.

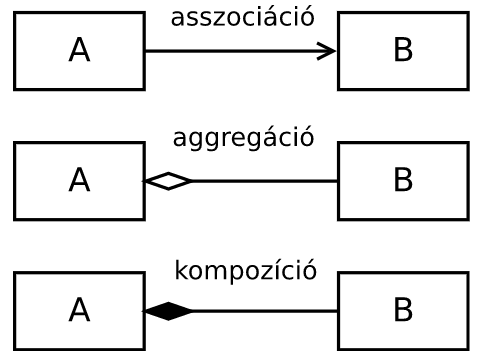
Use case dokumentálása is legalább annyira fontos, mint maga a diagram, két féle képpen lehet dokumentálni: természetes írott módon, mondatokkal, illetve táblázat formátumban, sablonokat használva.

Pl. 

**Osztály diagram:** Az objektumorientált programok tervezése jó esetben UML osztálydiagram rajzolásával, tehát papírral, ceruzával és radírral kezdődik, a legritkább esetben sikerül elsőre a tervezés, sokszor kell javítani, módosítani. Statikus, ki kivel van kapcsolatban.

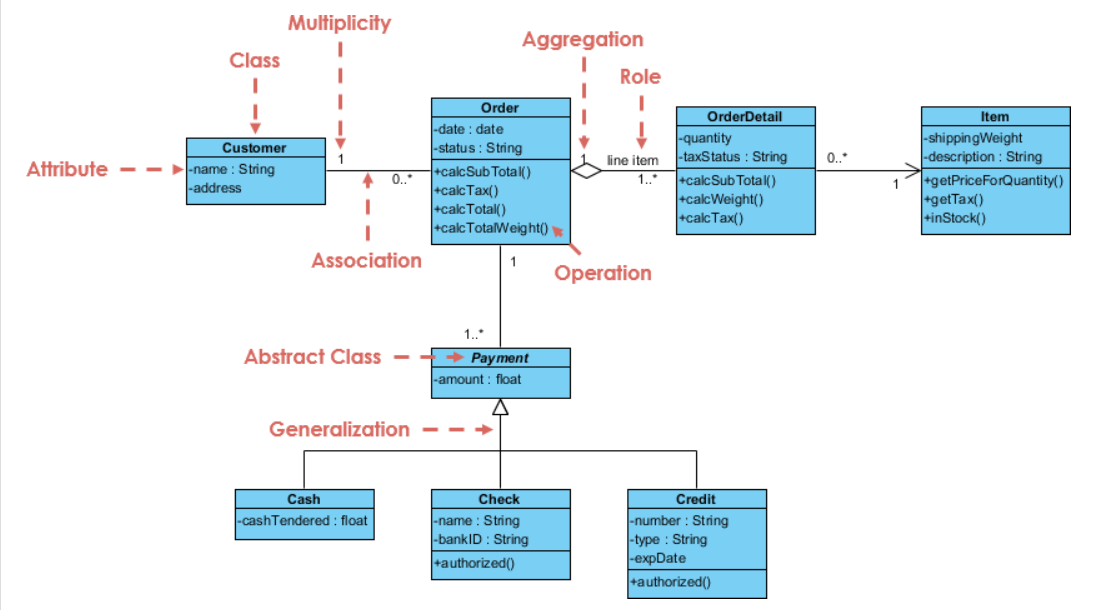
Deffiniciók, mik az osztály objektum stb itt is elmondani (OOP részből tudniillik)

**Jelölések:**

****

* Asszociáció: Gyenge ismeretség, azaz az objektum ismeri a másik létezését.
* Aggregáció: Bármi, ami a kettő között van, nem biztos, hogy annyira fontos, mint a kompozíció. Rész-egész. A rész túléli az egészet.
* Kompozíció: rész-egész viszonyt fejez ki, A objektumok tartalmaznak B-t. A rész nem éli túl az egészet.

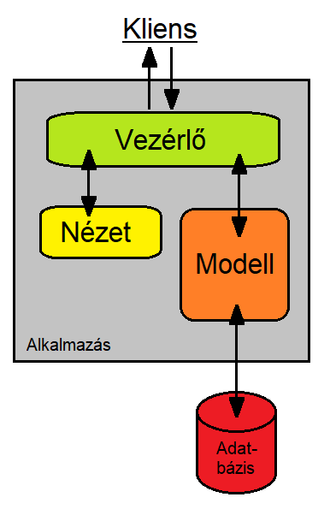
Az osztálydiagramm több osztályból áll, amelyek háromba osztott téglalapként vannak definiálva. A téglalap legfelső részében található az osztály neve, középütt az attribútumokkal, amelyek a private(-), public(+) és protected(#) kategóriába sorolhatók, úgyanúgy, mint a legalsó sorban található osztály metódusok. A függvények és attribútumok neve után kettősponttal van jelezve azok típusa.

Pl. 

**4. Architekturális minták. Model-View-Controller architektúra. Előnyök és hátrányok.**

Az **architekturális**  minták nagyban hasonlítanak a design patternekhez, annyi különbséggel, hogy a rendszert ha tározzák, míg a másik egy bizonyos problémára ad megoldást. Ez magasabb szintű a design patterneknél, jól bevált architekturális minták léteznek, amelyek megoldást nyújtanak a problémákra, az egyik ilyen a MVC.

A **Model-View-Controller** az egy programtervezési minta. A lényege az, hogy az összetett, sok adatot a felhasználó elé táró alkalmazásban az adathoz (model) és a felhasnzálói felülethez (view) tartozó dolgokat szétválassza, hogy a felhasználói felület ne befolyásolja az adatkezelést, a UI felbolygatása nélkül történjen mindez. Az MVC ezt úgy éri el, hogy elkülöníti az adatok elérését és a logikát az adatok megjelenítésétől és a felhasználói interakciótól egy közbülső összetevő, a vezérlő bevezetésével.



**Model:** Az információk. Sok alkalmazás használ állandó tároló eljárást (mint pl adatbázis), azonban az MVC nem említi külön az adatelérést, ezt beleérti a modelbe.

**View:** A felhasználói felület.

**Controller:** Az eseményeket, jellemzően felhasználói műveleteket dolgozza fel és válaszol rájuk, illetve a modellben történő változásokat is kiválthat

1. A felhasználó valamilyen hatást gyakorol a felhasználói felületre (pl. megnyom egy gombot).
2. A vezérlő átveszi a bejövő eseményt a felhasználói felülettől
3. A vezérlő kapcsolatot teremt a modellel, esetleg frissíti azt a felhasználó tevékenységének megfelelő módon (pl. a vezérlő frissíti a felhasználó kosarát).
4. A nézet (közvetve) a modell alapján megfelelő felhasználói felületet hoz létre (pl. a nézet hozza létre a kosár tartalmát felsoroló képernyőt). A nézet a modellből nyeri az adatait. A modellnek nincs közvetlen tudomása a nézetről.
5. A felhasználói felület újabb eseményre vár, mely az elejéről kezdi a kört.

**Előnyei:**

* ***Egyidejű fejlesztés*** – Több fejlesztő tud egyszerre külön a modellen, vezérlőn és a nézeteken dolgozni.
* ***Magas szintű összetartás*** – MVC segítségével az összetartozó funkciók egy vezérlőben csoportosíthatóak. Egy bizonyos modell nézetei is csoportosíthatóak.
* ***Függetlenség*** – MVC mintában az elemek alapvetően nagy részben függetlenek egymástól
* ***Könnyen változtatható*** – Mivel a felelősségek szét vannak választva a jövőbeli fejlesztések könnyebbek lesznek
* ***Több nézet egy modellhez*** – Modelleknek több nézetük is lehet
* ***Tesztelhetőség*** - mivel a felelősségek tisztán szét vannak választva, a külön elemek könnyebben tesztelhetőek egymástól függetlenül

**Hátrányai:**

A MNV hátrányait általában a szükséges extra kódból adódnak.

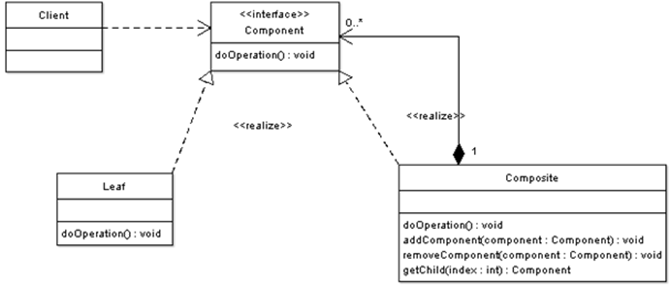
* ***Kód olvashatósága*** – A keretrendszer új rétegeket add a kódhoz ami megnöveli a bonyolultságát
* ***Sok boilerplate kód*** – Mivel a programkód 3 részre bomlik a ebből az egyik fogja a legtöbb munkát végezni a másik kettő pedig az MVC minta kielégítése miatt létezik.
* ***Nehezebben tanulható*** – A fejlesztőnek több különböző technológiát is ismernie kell az MVC használatához.

5. **Tervezési minták. Összetétel (Composite), Egyke (Singleton), Megfigyelő (Observer) minták.**

Az informatikában a programtervezési mintának nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Céljuk az, hogy leírást vagy sablont nyújtsanak. Segítik formalizálni a megoldást.

**Összetétel (Composite):**

Az **összetétel** minta az írja le, hogy az objektumok egy csoportját ugyanúgy kell kezelni, mint egy adott objektum példányait külön-külön. Az összetétel itt arra utal, hogy fa struktúrába szervezzünk objektumokat így reprezentálva a rész-egész hierarchiákat. Az összetétel minta lehetővé teszi, hogy a kliensek az önálló objektumokat és összetételeket egységes módon kezeljék.



A *Component* egy olyan interfész, amelyet mind az összetett osztálynak, mind a levélelemnek implementálnia kell. A *Leaf* osztály reprezentálja a tovább már nem osztható elemeket, míg a *Composite* osztály objektumai további komponensekből épülnek fel.

**Egyke (Singleton):**

Az **egyke** egy olyan programtervezési minta, amely egy objektumra korlátozza egy osztály létrehozható példányainak számát. Abban az esetben érdemes használni, amennyiben a rendszer működése rendellenes lenne, vagy össze is omolna abban az esetben, ha több objektumot is lehetne példányosítani. **Nem lehet publikus konstruktora**, csak akkor hasznos, ha egyetlen objektumra van szükségünk.

**Megfigyelő (Observer):**

A **megfigyelő** mintában egy objektum, melyet alanynak hívunk, listát vezet alárendeltjeiről, akiket megfigyelőknek hívunk és automatikusan értesíti őket bármilyen állapotváltozásról, többnyire valamely metódusuk meghívásán keresztül. Többnyire elosztott eseménykezelő rendszerek kialakításakor használjuk. A Megfigyelő minta kulcsfontosságú része az ismert Model-View-Controller (MVC, Modell-Nézet-Vezérlő) architekturális modellnek. A Megfigyelő mintát számos programozási könyvtár és rendszer alkalmazza, többek között szinte minden GUI toolkit.

A **megfigyelő minta akkor hasznos**, amikor egy objektum egy másik objektum állapotában érdekelt, és tudatában kell lennie (értesítést kell kapnia) az ebben bekövetkező változásokról. A megfigyelő mintában megfigyelőnek (*Observer*) hívják azt az objektumot, amely ellenőrzi egy másik objektum, az alany (*Subject*) állapotváltozásait.

