## 1. Szoftver projekt fejlesztés lepései

Tehát mi határoz meg egy projektet ? Pénz, Idő és Minőség stb.
Minden projektnek a következő lépéseken kel végig futni: Initialization, Planing, Executing, Monitoring and Controlling, Closing.
Mit történik minden egyes fázisban? Milyen dokumentumok születnek? Kik az érintett emberek?
Ezen kívül, a jobb jegy érdekében, illik megemlíteni azt, hogy van a Vízesés modell, vagy pl az agile modellek, melyek másképpen járják be ezeket a lepéseket. Előnyök és hátrányok.
Konkrét példa: a vízesés nagyon jó abból a szempontból, hogy ezeket szépen sorrendbe járja be, tehát mindenki tudja, hogy mondjuk most tesztelünk, és akkor semmi mással nem foglalkozunk. A probléma az, hogy a kliens nagyon későn látja meg a termekét. Ha egy liftet készítünk, ahol a követelmények nem változnak érdemes ezt használni.
Ezzel szemben az agileos megközelítések, iteráló és inkrementáló, ezeken a fázisokon ciklikusan mennek végig (végig iterál, egy ilyen ciklus kb 2 hetet tart és sprinteknek is nevezik) és minden ciklusban egy kicsit tervez, egy kicsit kivitelez, ..minden ciklus végeredménye egy inkrement lesz , az az a felhasználó számara egy működő és használható szoftver. Ha egy weboldalt készítünk, ahol a követelmények naponta változnak érdemesebb ezt használni.

## 2. Követelmény specifikáció.

Ki készíti? Kinek szól? Mit tartalmaz? Mire használjuk? (pl. időt lehet becsülni a követelmények alapján)
Milyen követelmények vannak? Felhasználói követelmény, Rendszer követelmény (ezen belül funkcionális és nem funkcionális). Érdemes példákat adni - akar a saját államvizsga dolgozatodból.
Illetve lehet beszelni a jól követelmények tulajdonságairól: egyértelmű, tesztelhető, szükséges, teljes, implementáció nélküli stb.

## 3. UML diagramok. Használati eset diagram (dinamikus kép). Osztály diagram (statikus kép). Mik az UML diagramok? Mire használjuk az UML diagramokat?

A használati eset vagy use case diagram egy dinamikus képet ad a rendszerről, külső szemlélők szemszögéből.
Nincs benne a rendszer belső működése. Szerepköröket azonosít, pl diák, tanár, titkárnő, akik más más formában szeretnék kapcsolatba lépni a rendszerrel. A szerepkőröket aktoroknak nevezzük, és pálcika emberekkel rajzoljuk.
A cselekvést (ugye dinamikus, pl a -titkárnő- -kinyomtatja névsort-, a -tanár- -jegyet ad-) kis ellipszisekbe írjuk.
Nagyon közkedvelt diagram, mert human readable, magyarán ezt a kliens is megérti és a programozó is.
Fontos megjegyezni, hogy a use case diagram a palcika emberek mellett tartalmaz egy leirast is. Ez az jelenti, hogy minden egyes usecaset dokumentálunk: ki indítja a cselekvést, milyen előfeltételek vannak, mi történik az optimális esetben, mi történhet a kevésbe szerencsés esetben stb.

Az osztály diagram a programozóknak szol. Kell ügyelni, hogy mit mivel jelölünk, pl egy

öröklődést, vagy egy tartalmazást (kompozíció, aggregáció). Fontos kiemelni, hogy csak a teljes osztálydiagramra szoktak mindent feltüntetni.
Tehát a privát elemeket nem szoktak, illetve érdemes egy modul osztálydiagramját elkészíteni. Ha túl sok elem van rajta akkor nem átlátható.
Fontos, hogy az osztálydiagramot oda lehet adni egy programozónak, hogy implementálja.
Érdemes kiemelni azt is, hogy az UML diagramok nyelv függetlenek, nyugodtan lehet az osztálydiagramot elkészíteni C++ vagy java nyelveken, sőt bármelyik OOPt támogató nyelven.

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

A rendszer architektúráját kell definiálni. Ez a big picture, amin minden fontos és elengedhetetlen komponens rajta van.
Ezek madártávlatból készítjük, azaz nem a részletek a fontos, hanem a nagy komponensek.
Több architektúra típus van: pl a rétegelt architektúra (gondolok most a .NETre, vagy az Android SDKra, de akar mondhatnám az OSI modellt hálózatokból), vagy pipe and filter (pl a labview vagy matlab simulink esetén), client server stb.
Mindeniknek van előnye és hátránya: pl a rétegelt estében jól elkülönített rétegek vannak, könnyű dolgozni rajta párhuzamosan, könnyű tesztelni, illetve könnyű kicserélni egy réteget anélkül, hogy zavarná a többit.
Hátrány pl az, ha egy funkcionalitás túlnyúlik a rétegeken, vagy egy magas elemnek szüksége van egy nagyon alacsony szintű információra (pl. billentyűzet beolvasás, szépen függvényeken keresztül le kell menni a legalacsonyabb szintekre, majd ugyanúgy vissza kell menni - de ez mind idő)
Az MVCről röviden le kell írni, hogy hol használják, és miért van szétbontva 3 komponensre. Mit csinál a model, a view és a controler külön külön.

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

A tervezési minták azok inkább guidlineok, olyan sablonok, melyek kiállták az idő próbáját és jól bevált módszereket javasolnak gyakori problémákra.
Háromféle van: strukturális (ahol egyes komponensek közti tartalmazási kapcsolatok vannak leírva pl a composite), creational (olyan módszerek, melyekkel létrehozunk objektumokat, pl. singleton), illetve a behavioral (ami a viselkedést írja le, pl az observer, vagy az iterator pattern - gondolok itt pl a listák elemein végig szaladó iteratorra).
Composite, Singleton, Observer estében hol használjak, és mi az alap ötlet?
Composite: a rész-egész kapcsolatát emeli ki: pl a -katona- osztályból származik a gyalogos és szakasz (azaz több -katona-), így bármilyen katonát hozza tudunk adni a szakaszhoz.
A singleton - vigyáz arra, hogy csak egyetlen példány legyen létrehozva, privát konstruktor. Nagyon kell vigyázni, hogy ez egy multithreaded környezetben elég nehéz.
Observer esetében: ez a klasszikus publisher-subscriber mintán alapszik, azaz én mind publisher készítek egy eseményt, pl ebed időkor, megkongatom a harangot, a subscriber feliratkozik erre az eseményre, és amikor bekövetkezik (szól a harang) akkor értesítve lesz. Gyakori példák erre: a formok eseteben OnSaveButtonClicked, egy esemény amire mi feliratkozunk.
Az érdekessége az, hogy mi, mint felhasználók, nem kell tudjunk a Button osztályról, nem is érdekel, mi csak annyit akarunk, hogy amikor bekövetkezik mi is tudjunk róla.