Tartalom

1. A fejlesztői eszköz kiválasztásánál milyen szempontokat érdemes figyelembe venni és miért? 3

2. Jellemezd egy hatékony tervezési, fejlesztési és javítási mikrociklust (mind a hármat)! Mire érdemes odafigyelni? 3

3. Adj kritikát a vízesés és az evolúciós szoftverfejlesztési modellekre! 4

4. Adj jellemzést a különböző szerződés modellekről! 5

5. Írd le, hogy a különböző szerződés modellek mire motiválják a résztvevőket! 5

6. Írd le, hogy a különböző szerződés modellek kockázataira milyen kockázat-mérséklési technikákat ismersz! 5

7. A különböző szerződés modellek milyen elvárásokat támasztanak a módszertannal szemben? Mit gondolsz az alkalmazásban megjelenő rugalmasságról? 6

8. A tesztelésbe milyen szerepkörű résztvevők vannak bevonva? Mi a tipikus feladatuk a teszteléssel kapcsolatban? Jellemezd, hogy a projekt különböző fázisaiban mik a módosítások költségei! 7

9. Mi az ellenőrzés és mi a validáció? Mit tartalmaz a tesztterv? Röviden jellemezd az elemeket! Mit értünk tesztelési stratégián? 7

10. Mi a unit tesztelés alapkoncepciója? Mik egy unit teszt tipikus lépései? 8

11. Ismertesd a jó unit teszt ismérveit, röviden jellemezd őket! 8

12. Röviden jellemezd az NUnit keretrendszert, ismertesd a főbb attribútumokat, használatukat, ellenőrző módszereket! 8

13. Jellemezd a moq keretrendszert! Add meg a főbb osztályokat, mutasd be a legfontosabb kódolási mintákat (nem kell feltétlenül kód, elég a koncepció)! 9

14. Mi jelentenek az IoC és DI rövidítések? Mik a fogalmak jellemzői? Mit jelent ebben a kontextusban a resolution, resolution root, injection? 10

15. A DI keretrendszerek tipikusan milyen életciklus modelleket támogatnak? Mutasd meg a dryIoc keretrendszer néhány kapcsolódó funkcióját! 10

16. Írd le, hogy egy adatbázis alapú alkalmazáshoz milyen módon érdemes unit teszteket készíteni és miért? 11

17. Jellemezd a Repository mintát! Mi az előnye, hátránya más megoldáshoz képest? Milyen viszonyban van az ORM technológiákkal? 11

18. Jellemezd a property és konstruktor alapú DI megoldásokat! 12

19. Jellemezd a különböző Factory mintákat és Service Locator mintát DI szempontból! Hasonlítsd össze a két megoldást! 12

20. Írd le, hogy az alkalmazást hogyan kell strukturálni ha unit tesztelni szeretnénk? 13

21. Foglald össze az Structured Systems Analysis And Design Method (SSADM) módszer lényegét, fázisait, előnyeit és hátrányait! 16

22. Foglald össze a követelményelemzés céljait, módszereit, azok előnyeit és hátrányait! 17

23. Mutasd be a követelményelemzés során végzett aktivitásokat és azok jellemzőit! 18
24. Mutasd be a követelményelemzés során előtérbe kerülő különböző követelménytípusok jellemzőit! 18
25. Mutasd be a követelményelemzés során előtérbe kerülő kihívásokat! 19
26. Foglald össze az üzleti elemzés céljait! 19
27. Foglald össze az üzleti elemzés területeit, technikáit és jellemzőit! 20
28. Mutasd be a szoftvertervezés során alkalmazott alapelveket és koncepciókat! 22
29. Mi a szoftvermodellezés szerepe a szoftvertervezés során? Milyen módszereket, technikákat ismersz? 23
30. Miért használunk a modellezés során eltérő absztrakciós szintű modelleket? Mutass rá példákat! 23
33. Mutasd be a szoftverfejlesztési folyamat lépéseit, röviden elemezd őket! 25
34. Hasonlítsd össze a vízesés, a spirál és az iteratív fejlesztési módszereket! 25
35. Mutasd be az iteratív és inkrementális fejlesztési módszerek jellemzőit! 26
36. Mutasd be a Capability Maturity Model Integration (CMMI) különböző területeinek céljait és jellemzőit! 26
37. Mutasd be a Capability Maturity Model Integration (CMMI) különböző szintjeit és azok jellemzőit! 27
38. Foglald össze az agilis módszerek értékeit és elveit! 30
39. Foglald össze az agilis tervezés jellemzőit és szintjeit! 31
40. Mutasd be az agilis módszerek során alkalmazott kiadás tervezést és annak jellemzőit! Térj ki a felhasználói sztorik szerpére! 31
41. Foglald össze az agilis iterációk jellemzőit és az iteráció tervezés lépéseit! 32
42. Mutasd be a napi stand-up esemény és a “Kész, kész” (done, done) jellemzőit! 32
43. Foglald össze az eXtreme Programming (XP) jellemzőit és eszközeit! 33
44. Foglald össze az eXtreme Programming (XP) értékeit és elveit! 34
45. Mutasd be a Scrum fejlesztési folyamatot és jellemzőit! 35
46. Mutasd be a Scrum ceremóniák jellemzőit! 35
47. Mutasd be a Scrum szerepköröket és a sprint tervezés jellemzőit! 36
48. Mik az előnyei a verzió kezelő eszközök használatának? Milyen verzionálási modelleket ismersz, mik ezek előnyei, hátrányai? 40
49. Ismertesd a központosított és elosztott verziókezelés előnyeit, hátrányait! 40
50. Milyen elágaztatási stratégiákat ismersz? Jellemezd őket röviden! Jellemezd Git legfontosabb fogalmait! 41
51. Jellemezd a git legfontosabb objektumait, adj példát egy módosítási művelet során végbemenő változásokra! 44
57. Jellemezd a szervezeti stratégiát és viszonyát a projekthez! 47
58. Jellemezd a lineáris-funkcionális szervezeti formát! 48

59. Jellemezd a projektorientált szervezeti formát! 48
60. Jellemezd a mátrix szervezeti formát! 49
61. Írj a projekt tipológiáról! 50
62. Mi a projekt karakterisztika, kik egy projekt résztvevői? 50
63. Sorold fel a PMBOK folyamatcsoportjait és tudásterületeit! 50
63. Mik a SOLID elvek? Röviden ismertesd mind az ötöt! Válassz ki egyet és jellemezd részletesen! 53
64. A szoftverfejlesztési folyamat tevékenységei. Foglald össze és röviden mutasd be a szoftverfejlesztési aktivitásokat! 53
65. Mutasd be a szoftverrendszerek, alkalmazások értékeit! 53
66. Mutasd be a szoftverprojektek életciklusának elemeit, azok jellemzőit! 53

Szoftverfejlesztési módszerek és paradigmák kérdéssor vizsgára 2014/15 tavasz

Fejlesztő eszközök hatékony használata

1. A fejlesztői eszköz kiválasztásánál milyen szempontokat érdemes figyelembe venni és miért?

- Kik használják, egyedül vagy az egész csapat, milyen meglévő ismereteik vannak, milyen az eszköz tanulási görbéje, rövid vagy hosszú távra kell
- Eszközgyártó: nagy cég esetén lassú innováció de megbízható, kis cég esetén gyors innováció de könnyen csődbe mehet, opensource esetén nem biztos a support
- Sosem 1 eszközt használunk, milyen másik eszközzel tud integrálódni, import, export, élő kapcsolat

2. Jellemezd egy hatékony tervezési, fejlesztési és javítási mikrociklust (mind a hármat)! Mire érdemes odafigyelni?

- Tervezési:
    - funkció:, van-e nem specifikált rész, külső függőség
    - architektúra: van-e már helye a funkciónak, hibakezelés, naplózás megoldott-e
    - felhasználói felület: mi a pontos használati eset, ergonómikus-e
- Fejlesztési:
    - hány mini-taszkot csinálsz meg 2 alkalmazás indítás között? Ha 1-2őt akkor lassú lesz, ha sokat, fontos mindegyiket tesztelni
    - Hibaüzenetek alapos elolvasása, InnerException megtalálása
- Hibajavítási:
    - hibákat ne egyesével, hanem burst-ösen javítsuk
    - Edit&Continue használata

- ha valamit többször kell javítani old meg, hogy minél könnyebben el lehessen jutni az adott részhez pl. teszteléshez beégetett adatok

Ami kimaradt a diasorokból:
- .NET hasznos tanácsok

Módszertanok

Módszertan: csapat által használt lépések összessége, melyek kritikusak a projekt sikeressége szempontjából, úgy mint munkaszervezés, tervezés, megvalósítás (implementálás), tesztelés, telepítés és karbantartás.

3. Adj kritikát a vízesés és az evolúciós szoftverfejlesztési modellekre!

Vízesés modell:
- Lépések: 1.Követelmény, 2.Specifikáció, 3.Tervezés (architektúrális és részletes), 4.Megvalósítás és integráció (implementáció), 5.Tesztelés (verifikáció és validáció), 6.Telepítés, 7.Karbantartás
- Gyakran nem az készül el amire szükség van, mivel sokszor nem kapunk visszajelzéseket a termékről addig, amíg az el nem készült (evolúciósnál folyamatos visszacsatolás prototípusokon keresztül)
- A folyamatosan változó követelmények sok újratervezést igényelnek, amely sok kidobott időt foglal magában, túl hosszúvá nyúlhat a fejlesztési fázis
- A tervezők gyakran nincsenek tisztában az implementációs részletekkel
- Néha egyszerű tervváltoztatás helyett bonyolult implementáció készül el
- A tervező eszközök nem integrálódnak jól az implementációt segítő eszközökkel
- Túl hosszú lehet a teljes fejlesztési fázis

Evolúciós modell:
- Több ciklus, mindegyik a termék egy részéért felel, így követelmények egyszerűen módosíthatóak. Legnépszerűbb az **Agilis** fejlesztés (Scrum, Extreme Programming)
- Agilis értékek (megoldások az evolúciós problémákra):
  - Egyének és interakció inkább, mint folyamatok és eszközök
  - Működő szoftver inkább, mint jól dokumentált szoftver
  - Felhasználó bevonása inkább, mint szerződések tárgyalása
  - Reagálás a változásokra inkább, mint egy terv követése
- Megrendelő oldalról költség és határidő szempontjából nehezen tartható
- Alkalmazás előállításának költségei tipikusan nagyobbak, utólag nehéz elszámolni vele a sok módosítás és iteráció miatt (nincs fix fejlesztési terv az elejétől, sokat változik)
- A megrendelő nincs rákényszerítve a specifikáció végiggondolására (vízesésnél előre elkészül)

4. Adj jellemzést a különböző szerződés modellekről!

Projekt alapú:

- Szerződésben rögzített specifikáció (feature set), és ellenérték
- Kockázat: fejlesztőnél, csak a legvégén fizetik ki ha leszállítja a feature set-et
- Megrendelő: nincs mozgástere, ha a végén elkészül amit kért, ki kell fizetni

Költség alapú:

- Szerződésben fejlesztői munka/erőforrások időarányos ára (óradíj)
- Kockázat: megrendelőnél, bármi is lesz a projektből, a ráfordított időt ki kell fizetni, minőségi garancia nélkül
- Megrendelő: van mozgástere, bármit kérhet, megcsinálják

5. Írd le, hogy a különböző szerződés modellek mire motiválják a résztvevőket!

Projekt alapú:

- Fejlesztők: feature freeze, semmi változtatás az eredeti specifikáción, merev ellenállás
- Megrendelő: minél gazdagabb funkció fejlesztése
- Probléma: az előzetes specifikáció sosem lehet teljes (megrendelő nem tudja vagy nem veszi a fáradtságot, nem az írja aki használni fogja, változó környezet), mivel hiányos, szükséges lenne a változtatás, de az nehézkes
- Megoldás: az ajánlatba valamilyen puffer beépítése a rugalmassághoz, lépcsőzetes tervek és kifizetések, de ez lassú

Költség alapú:

- Fejlesztők: minél több, elhúzódó munka, változtatások, addig sem megy élesben, nem derülnek ki a hibák
- Megrendelő: alul specifikálás, hiszen közben bármikor módosítható
- Probléma: konkrét specifikáció híján rengeteg nagy költségű változtatás, amelyek akár architektúrálisak is, nem készül el a kívánt alap funckionalitás sem, a költségek irreálisan nagyra nőhetnek
- Megoldás: közös tervezés és projekt vízió, előzetes költség becslések

6. Írd le, hogy a különböző szerződés modellek kockázataira milyen kockázat-mérséklési technikákat ismersz!

Projekt alapú:

- Ajánlatba beépített puffer a Fejlesztői oldalon
  - Verseny környezetben kockázatos
- Rugalmasság, mindkét oldalon
  - Barter megállapodások: nem módosul a teljes projekt összeg
- Lépcsőzetes kidolgozás és ajánlatok
  - Részletes, kifizetett tervek
  - Hátránya: lassú átfutás, sok döntési pont

Költség alapú:

- Minden agilis módszertan nagy hangsúlyt helyez a tervezésre, közös vízióra!

- A termék elhelyezése a Megrendelő stratégiai koncepciójában, életében
- Előzetes (nem kötelező érvényű) költség becslések

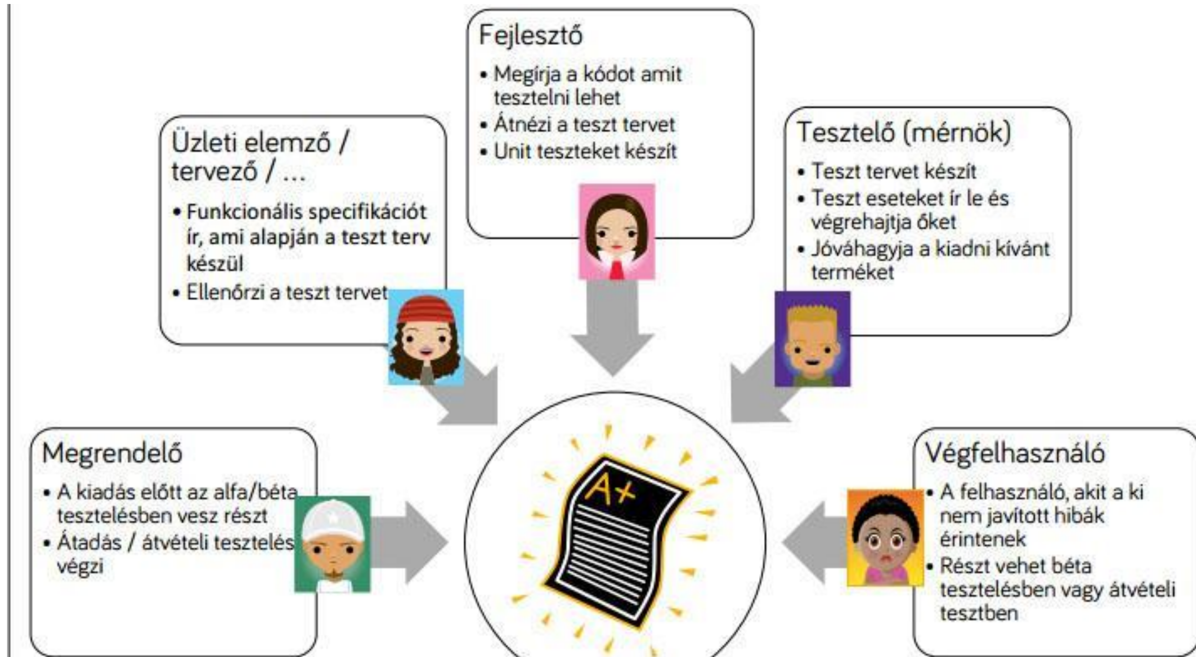
7. A különböző szerződés modellek milyen elvárásokat támasztanak a módszertannal szemben? Mit gondolsz az alkalmazásban megjelenő rugalmasságról?

- A projekt alapúnál szükséges előre elkészíteni a feature set-et, tehát egy kellően részletes tervet kell előre elkészíteni, amely alapján a szerződés létrejön, tehát ez a klasszikus vízesés modell módszertanra épít jobban. Ahogyan a vízesés módszer is kellően rugalmatlan, úgy ez a szerződés típus is, de puffer-el javítható.
- Költség alapúnál a megrendelő nincs rákényszerítve, hogy előre átgondolja a követelményeket, specifikációt, valamint menet közben kérhet bármilyen változtatást amely akár architektúrális jellegű is lehet, tehát ez sokkal közelebb áll az evolúciós (agilis) módszertanhoz. Kellően rugalmas mind a módszertan mind a szerződés.

Itt nem maradt ki semmi a diasorokból.

Tesztelés

8. A tesztelésbe milyen szerepkörű résztvevők vannak bevonva? Mi a tipikus feladatuk a teszteléssel kapcsolatban? Jellemezd, hogy a projekt különböző fázisaiban mik a módosítások költségei!



A javítás költsége attól függően, hogy hol és mikor találjuk meg a hibát

	Mikor ismerjük fel a hibát...				
Hol van a hiba	**Spec. elemzés**	**Tervezés**	**Megvalósítás**	**QA**	**Kiadás után**
Követelmény	1x	3x	5-10x	10x	10-100x
Terv	-	1x	10x	15x	25-100x
Megvalósítás	-	-	1x	10x	10-25x

9. Mi az ellenőrzés és mi a validáció? Mit tartalmaz a tesztterv? Röviden jellemezd az elemeket! Mit értünk tesztelési stratégián?

Validáció: “A megfelelő szoftvert készítjük?” (LZ: A jó szoftvert készítjük?)
Verifikáció: “A jó irányba haladunk?” (LZ: Jól készítjük-e el a szoftvert?)
Az elfogadási teszt a validációról szól, a többi az ellenőrzésről.
Tesztterv:

- teszteléshez tartozó követelményrendszer
- tesztterv sablon:
    - teszt célja
    - szkóp: tesztelt és nem tesztelt funkciók listája

    - tesztelési stratégia
    - tesztesetek listája
    - nyitott kérdések

Tesztelési stratégia:

- Leírja a használni kívánt tesztelési módszert: black/gray/white box teszt
- Megadja milyen módszerrel végezzük és milyen keretrendszert használunk hozzá
- Milyen előfeltételei vannak a teszteknek: infrastruktúra, architektúra, input adatok, külső függőségek pl. szolgáltatás

10. Mi a unit tesztelés alapkoncepciója? Mik egy unit teszt tipikus lépései?

Unit teszt: egy darab kód, amely egy másik darab kódot ellenőriz.
SUT: System Under Test
CUT: Class/Code Under Test

- Alkalmazás **egyetlen** funkcióját teszteli
- **Izoláltan**, minél kevesebb külső függőséggel, ehhez interfész alapú programozás
- Minden esethez külön unit teszt
- Automatikusan futtatható legyen
- Tipikus felépítés:
    - Setup
    - Excercise
    - Verify
    - Teardown

11. Ismertesd a jó unit teszt ismérveit, röviden jellemezd őket!

- Megbízható (nem kell manuálisan ellenőrizni, hogy biztosan jó-e), hosszú távon futtatható
- Karbantartható:
    - minimális a logika benne
    - kerüli a privát belső állapotok ellenőrzését (refaktorálásnál változik)
    - izolált, minél kevesebb külső függőséggel
- Könnyen, jól olvasható
- Egyszerű (kicsi a tesztelt kód)
- Automatizált és megismételhető, bárki tudja futtatni
- Gyorsan lefut
- A tesztek nem épülnek egymásra

12. Röviden jellemezd az NUnit keretrendszert, ismertesd a főbb attribútumokat, használatukat, ellenőrző módszereket!

- JUnit keretrendszerből nőtte ki magát, ma már sok nyelvhez elérhető
- Nyílt forráskodú, élő közösség van mögötte
- Jól integrálódik a Visual Studio-ba

- Tipikus NUnit teszt felépítés: Setup > Excercise > Verify > Teardown
- Teszt leírók (attribútumok):
    - TestFixture: teszt osztályra
    - Test: teszt metódusra
    - Category: [Test, Category(“My test category”)]
    - SetUp: teszt futtatáshoz szükséges inicializálás
    - TearDown: visszaállítás az eredeti állapotra
    - TestCase: különböző tesztesetek futtatása egymás után
    - Theory: előre megadott Datapoints minden elemére a teszt lefuttatása

```
[TestFixture]
public class BasicAtt
{
    [Test]
    public void Passi

    [Test, Ignore("I
    public void Ignor
```

- NUnit ellenőrzés szintaktika:
    - Classic Assertions
    - Fluent Constraints
    - AssertionHelper: Expect
    - Constraints: Is, Has, Throws
    - pl: Assert.That(b,Is.LessThan(a));

- Paraméterezett teszteket is lehet írni

13. Jellemezd a moq keretrendszert! Add meg a főbb osztályokat, mutasd be a legfontosabb kódolási mintákat (nem kell feltétlenül kód, elég a koncepció)!

Mocking keretrendszer: fake objektumok dinamikus létrehozása futásidőben, interfész/alaposztály helyettesítés, viselkedésbeállítás, futtatás, ellenőrzést csinál

- Jelenleg a legtöbbet használt ilyen keretrendszer, egyszerű, gyorsan tanulható, korszerű
- Erősen típusos: LINQ kifejezések string-ek helyett
- Nincs record/replay minta
- Klasszikus Setup minta (record/replay-t nem támogatja)
- Mock<T> központi osztály, T típus példányát fogjuk mockolni
    - Setup metódus: melyik metódus milyen bemenetre milyen választ adjon
    - A T típusú mockolt példány az Object propertyn keresztül érhető el
    - Hívások ellenőrzése: Times.Never(), Times.AtLeastOnce()
    - Inputoknak intervallumokat meg lehet adni
    - propertyket stubként megadni: mock.SetupProperty(f=>f.Name, “..”)
    - propertyket mockolni: mock.Setup(f=>f.Name).Returns(“...”)
- Viselkedése:
    - Strict (true mock) mód: mindig kivételt dob, ha úgy hívják ahogy nem lett specifikálva
    - Loose: nem dob kivételt, mindig alapértelmezett értéket ad vissza (0, null)
    - Megadható, hogy az alaposztályt hívja meg, ha nincs a viselkedés felüldefiniálva (CallBase)
    - A viselkedés a konstruktorban adható meg
- Rekurzív mockok: a mock által visszaadott objektumok automatikusan mockok lesznek

- Egyebek: mock factory, protected tagok, események támogatása, callbackek támogatása, több interfész implementálása

14. Mi jelentenek az IoC és DI rövidítések? Mik a fogalmak jellemzői? Mit jelent ebben a kontextusban a resolution, resolution root, injection?

Inversion of Control (IoC), Dependency Injection (DI)

- Objektum gráfok létrehozása automatikusan
- Függőségek automatikus, rekurzív kielégítése
- Az interfészhez tartozó konkrét típusokat deklaratívan adhatjuk meg a konténer példányon

"The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependencies from your code." Stackoverflow

Magyarázat, mintakód: http://stackoverflow.com/questions/3058/what-is-inversion-of-control

IoC

Általános fogalom, bármire utalhat, amikor a vezérlés megfordul. Nem a komponens vezényel, hanem őt használják fel, hívják meg, irányítják, változtatják meg a viselkedését.

DI

- Szoftver tervezési minta: a függőségek átadásra kerülnek a függő objektumba
  - Az átadás lehet manuális vagy automatikus (injection)
- Minden külső függőséget írjunk le egy interfésszel, így cserélhető az implementáció, pl. megkaphatja konstruktorban (a lecserélés helye „seam")
- A minta elemei
  - A függőség implementációja
  - A függő/kliens objektum
  - Az interfész
  - (injektor objektum)
- Fogalmak:
  - Resolution: konténeren hívott Resolve metódus, amely visszaadja a létrehozott objektumgráfot
  - Resolution root: létrehozott gráf gyökere
  - Injection: függő objektumok létrehozása és átadása (pl. konstruktorban)

15. A DI keretrendszerek tipikusan milyen életciklus modelleket támogatnak? Mutasd meg a dryloc keretrendszer néhány kapcsolódó funkcióját!

A DI konténer szabályozza a létrehozott objektumok életciklusát. A regisztrációnál adható meg az életciklus modell.

- "normál" módon viselkedő objektumok
- Singleton minta
- Szálanként egyetlen objektum
- UoW támogatása – például HTTP kérésenként egy objektum létrehozása

Dryloc keretrendszer :

- Scope: Külön konténer objektum, ami követi a benne InCurrentScope módon létrehozott objektumokat, amik megszűnnek a konténerrel együtt
- Késleltetett példányosítás: Lazy<T> vagy Func<T> létrehozása esetén a példány nem jön létre, csak amikor azt elkérjük, így áttehető a létrehozás pillanata a megfelelő scope-ba
- A scope-ok automatikusan egymásba ágyazódnak, a konténer követi őket
- Választható, hogy hány aktív scope legyen
- Factory delegate-ek támogatása regisztrálásnál

16. Írd le, hogy egy adatbázis alapú alkalmazáshoz milyen módon érdemes unit teszteket készíteni és miért?

- Közös vagy minden fejlesztőnek saját adatbázis?
    - Közös: eleinte kisebb költség, de később minden fejlesztőnek más kiinduló állapotra lehet szüksége, tesztek elbukhatnak
    - Saját: minden fejlesztőnek saját kiinduló állapot, hátrány, hogy mindenkinek saját szkript, db restore, setup kód stb..
- Közös induló állapot vagy tesztenként saját?
    - Előre feltöltött: eleinte kisebb költség, új tesztek új állapotokat kívánnak meg, nehéz hozzá igazítani a régiekhez, nagy csapatban nehéz koordinálni
    - Tesztenként saját: minden tesztnek felkell építenie a sajátját, ez lassabb indulást eredményezhet
- Explicit cleanup/tranzakció alapú/üres db:
    - explicit kis költségű, de ha debug közben kilépünk, nem fut le a cleanup (induláskor cleanup…)
    - tranzakció alapú esetén egyszerűen nem commitáljuk el a végén
    - üres db esetén a teszt felhúzza saját magának egy szkripttel, lassan indul

Az ideális adatréteg:

- Unit tesztenként saját induló állapot
- Üres memória adatbázisban, kódból előállítva

17. Jellemezd a Repository mintát! Mi az előnye, hátránya más megoldáshoz képest? Milyen viszonyban van az ORM technológiákkal?

http://www.remondo.net/repository-pattern-example-csharp/

Széles körben használt az adatréteg leválasztására.

- A interfészen explicit jelennek meg az alsó réteg szolgáltatásai, funkciói
- Adatrétegtől független üzleti logika
- Specifikus hely, ahol az adatok elérésének módját módosítani kell
- Az adattáblák helye
- Elrejti a részleteket
- Több alternatív megvalósítása van
- Absztrakció az adatréteg fölött, könnyen cserélhető másik megvalósításra
- Domain logika és az adatelérés szétválasztása

- Fejlesztés jobb párhuzamosítása
- Unit tesztek támogatása
- Objektumorientált API
- Hátrányok, kritika:
    - még egy újabb réteg
    - API beszivároghat az üzleti logikába
    - Bonyolítja a kódot, hiszen „minden” lekérdezést explicit meg kell írni, nem használható a linq
    - „Visszatérés” a tárolt eljárások világába

ORM vs. Repository: mások a célok
- Repository: tárolással kapcsolatos összes funkció absztrakciója és egységbe zárása, architektúrális minta
- ORM: támogatott relációs adatbázisok elérésének absztrakciója, repository-ban használva egy implementációs részlet

18. Jellemezd a property és konstruktor alapú DI megoldásokat!

Konstruktor alapú:
- Előnye, hogy jól láthatóak a kötelező függőségek
    - A property injection használható az opcionális függőségek megadására
- Sok paraméter lehet
    - Áthidalható, ha egyetlen paramétert kap, amiben minden szükséges függőséget átad a hívó – de akkor elveszítjük az deklaratív függőség leírást
- Új függőség új paramétert jelent, ami miatt az összes unit teszt módosítandó
    - Vagy újabb .ctor-t kell bevezetni ami hívja a többit
- IoC konténerrel érdemes használni…

Property alapú:
- A függőségeket nem konstruktorban, hanem külön propertykben adja meg a hívó kód
- Tipikusan opcionális függőségek esetén jó megoldás
- Nem explicit a függőségek leírása
    - Egy új függőség bevezetését az összes meglévő unit tesztben kézzel, hibajelzés nélkül kell végig vezetni
- Nem látni jól, hogy melyek a külső függőségek, nehezen olvasható
    - Az osztálynak lehet egy csomó egyéb property-je, nem tudni, melyiket kell beállítani…

19. Jellemezd a különböző Factory mintákat és Service Locator mintát DI szempontból! Hasonlítsd össze a két megoldást!

Cél: a kliensről leválasztani a kompozíciós logikát

Factory minták:
- Factory method (simple factory)
    - általában egy metódus implementáció, egy objektumot hoz létre a paraméterek alapján
- Absztrakt factory minta
    - Több factory leszármazott van, amelyek közül az egyik kiválasztott tipikusan több kapcsolódó objektumot hoz létre

    - Kiemelhető közös inicializációs logika az ős factoryban
    - A konkrét típusokhoz tartozó specifikumok a leszármazottakban

Tiszta factory minták hátrányai:

- Tipikusan nem újra felhasználhatóak, létrehozó kód nehezen általánosítható
- Legtöbb implementáció fordítás idejű típusinformációra épít
- Nehezebb tesztelni

Service Locator minta:

- Egyetlen (singleton) objektum, ami tartalmazza az összes szükséges szolgáltatást
    - Inkább meglévő (létrehozott) objektumokat ad vissza – a regisztrációs adatbázisból (registry)
    - A szolgáltatások létrehozása lehet statikus (belekódolt), dinamikus (futásiőben lehet konfigurálni mire van szükség) stb.
    - De akár használhat Dependency Injection-t is
- Előnyök:
    - Új függőség bevezetése egyszerűbb, kevesebb módosítás
    - Kisebb kód, mintha rengeteg paramétert kellene konstruktorban megadni
    - A hívási hierarchia mélyén lévő osztályok könnyen tudják elérni a szükséges függőségeket
    - Az implicit függőségi hibák kiszűrésére megoldás a unit testing
- Hátrányok:
    - Függőségek nem jelennek meg explicit módon, a kódban imperatív módon van benne, hogy a Locatort milyen paraméterekkel hívja meg
- Új függőség bevezetésénél: lefordul a kód, de az összes unit test hibás lehet ami csak futási időben derül ki

Service Locator vs DI:

- SL esetén a függőség elkérése explicit, lehet breakpoint-ot rakni, de nehezebb a függőségeket látni, nehezebb újrafelhasználni
- DI-t nehezebb debugolni, de könnyebben átláthatóak a függőségek (konstruktor)

Összehasonlítás:

- Az egyik példányosítva csak egy adott típusú (vagy abból leszármazott) objektumot ad vissza (Factory)
- A másik példánya viszont több típusút tud előállítani (SL)
- A factory által visszaadott példány mindenképpen új
    - Így annak a hívó a tulajdonosa
- A ServiceLocator viszont nem biztos, hogy új példányt hoz létre
    - Ilyen értelemben IoC jellegű megoldás
    - A kapott példányt lehet, hogy mások is használják (singleton stb)

https://msdn.microsoft.com/en-us/library/dd458907.aspx

20. Írd le, hogy az alkalmazást hogyan kell strukturálni ha unit tesztelni szeretnénk?

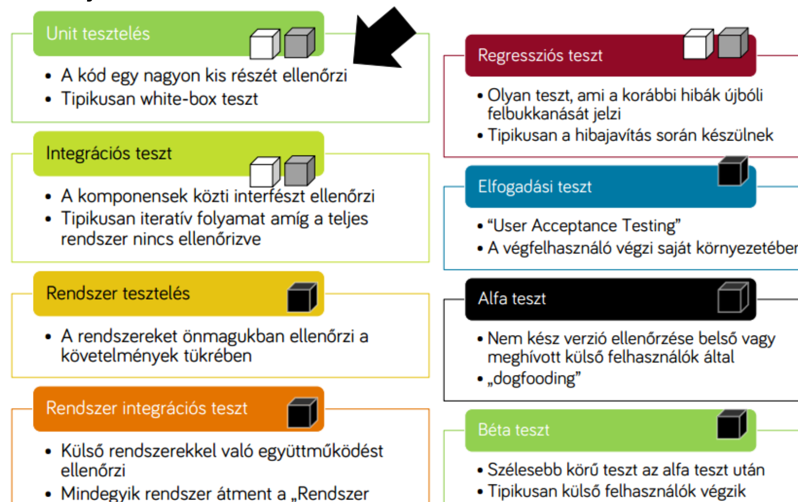
Milyen módszerek segítenek ebben? Mik a különböző megoldások előnyei, hátrányai?

- Interfész alapú fejlesztés
- Megjelenítés, üzleti logika, adatelérési réteg megfelelő izolációja

- Külső függőségek minimalizációja, ahol szükséges, ott IoC vagy DI -vel, pl. konstruktorral megoldani, hogy könnyen tesztelhető legyen mock object-el
- Lehető legkisebb egységekre bontani a kódot, így 1 egyszerű unit test tud tesztelni egy funkciót, gyorsan, átláthatóan

Ami kimaradt a diasorokból:

- Box analógia: white/black/grey
- A tesztelés szintjei:



- Állapot tesztelés: fekete doboz
    - A külső függőségeket behelyettesítettük stubokkal
        - Egyszerű, logika nélküli osztályok fix válaszokkal
    - A teszt sikeressége a tesztelt objektum állapotán múlik
        - Gyakran külön metódusok (accessor osztályok) kellenek a belső állapot lekérdezéséhez
- Vislkedés tesztelés: fehér doboz
    - A felhasznált komponenseket megfelelő sorrendben és paraméterekkel hívta-e meg
        - A hivatkozott komponensek mockolva vannak
    - Tipikusan mockokkal dolgozik
    - Mock (test spy): olyan objektum, ami a tesztelt kód (CUT) interakciója alapján dönti el, hogy a teszt sikeres-e
    - A teszt sikerességét a mock osztály dönti el!
- Test double terminológia:
    - **Dummy object:** Sose használt, paraméterként átadogatott objektum
    - **Fake object:** Működő, de leegyszerűsített implementáció, például memória adatbázis valódi helyett stb.
    - **Stub:**
        - Beégetett visszatérési értékeket ad a teszt során.
        - Inkább állapot verifikációra szolgál. Néha extra metódusokkal rendelkezik az állapot lekérdezéséhez.
    - **Mock object**
        - A valódi osztály működését imitálja.
        - Lényegi különbség van a setup és verifikációs fázisban, egyébként ugyanúgy többnyire fix értékeket ad vissza. Nem állapot verifikációval dolgozik.

    - **Mole:** Tetszőleges metódus hívás elirányítható. Futtató környezet (CLR) szintű megoldás.
- Unit of Work minta (UoW)
    - Nyilvántartja egy üzleti tranzakció során módosított/létrehozott objektumokat és koordinálja a módosítások véglegesítését
    - Egyéb, egy üzleti tranzakcióhoz tartozó funkciók, aspektusok megvalósítása
        - Naplózás, hibakezelés, tranzakció kezelés stb.
    - Egy UoW-ban megvan az összes szükséges repository stb.
    - A minta nem mindig explicit módon jelenik meg
- Tesztelés II közepén hosszú összefoglaló rész (~33-42)
- Mocking viselkedés: Teszetelés II 55-től

Specifikációs módszerek és üzleti elemzés

21. Foglald össze az Structured Systems Analysis And Design Method (SSADM) módszer lényegét, fázisait, előnyeit és hátrányait!

- 1980, Brit kormányzatban kormányzati szabványként alkalmazzák információs rendszerek fejlesztésében, elkülönült egységekre osztja fel az információs rendszer fejlesztésének munkáit, kvázi egyfajta vízesés módszer, dokumentum-vezérelt
- Minden lépést a korábban befejezett lépés eredményére épít, ragaszkodik az eredeti modellhez, rideg, szigorú struktúra, mely célja kontrollálni a folyamatokat

Technikák:

- Logical Data Modeling: A rendszerrel kapcsolatos adatelvárások azonosítása, modellezése és dokumentálása, az eredmény egy adatmodell
- Data Flow Modeling: A rendszerben történő adatmozgások azonosítása, modellezése és dokumentálása
- Entity Behavior/Event Modeling: Entitások viselkedésének modellezése

Fázisok:

- Stage 0 - Feasibility study: projekt megvalósíthatósági elemzés. Kisméretű projekt esetén elhagyható. Vizsgálandó területei: technikailag kivitelezhető? pénzügyileg, üzletileg a vállalat megengedheti, hogy végig vigye a projektet? szervezetileg, az új rendszer kompatibilis lesz a régi rendszerekkel? Etikailag, társadalmileg elfogadható?
- Stage 1 - Investigation of the current environment: minden esetben létezik korábbi verzió, ez lehet akár papír és ceruza is. Elemezni kell a már meglévő rendszert, ennek előnyeit és hátrányait, korlátait és lehetőségeit felismerni. A régi rendszer adja az új követelményének az alapját, mely hiányosságait kijavíthatjuk, új lehetőségeket megvalósíthatunk. Felhasználók bevonása fontos az alapos megismeréshez. Végeredmény a felhasználói és követelmény katalógus, aktuális szolgáltatások leírása, adatszótár.
- Stage 2 - Business system options: üzleti megoldásvariációk kidolgozása, ötletek, javaslatok, mi maradjon a régiben, mi legyen teljesen új, mely funkciók kerüljenek be, költségek és megtérülés (haszon), mi lesz az új rendszer hatása
- Stage 3 - Requirements specification: a stage1 és stage2 alapján elkészül a rendszer logikai specifikációja, nem tartalmazza a konkrét megvalósítást, csak a rendszer viselkedését. Funkció definíciók, entitás relációk. Végeredmény a teljes követelmény specifikáció.
- Stage 4 - Technical system options: stage1 fizikai implementáció iránya, figyelembe veszi a szükséges emberi erőforrásokat, költségeket, elosztottságot, felhasználói felületeket. Mindezeknek összhangban kell lennie az üzleti feltételekkel.
- Stage 5 - Logical design: kimenete az implementáció független felhasználói felülettel kapcsolatos elvárások, rendszerrel való fő interakció elemei, események rendszerre való hatása. Stage3 eredményeit felhasználva kerül kialakításra. Fázis eredménye a logikai terv, melyben adatkatalógus, logikai adatstruktúra, logikai működési modell található.

- Stage 6 - Physical design: minden logikai specifikáció valódi hardver és szoftver elemekre kerül lefordításra, valódi technikai fázis. Valós adatbázis struktúra kialakítása, függvények pontos struktúrája és implementálási módja stb..

Előnyök, hátrányok:

- üzlet metodológián alapuló tanulmányozása, az elvárások szerint mélyebb, rendszerezettebb ismereteket ad az üzleti folyamatokról, adatok szerkezetéről, mely egy korrektebb rendszerhez vezethet.
- Az üzlet és folyamatai folyamatosan változnak, emiatt frissíteni kell a fázisok eredményeit, amely plusz feladatokat és késést eredményez.
- Az SSADM esetén minden fázis elkezdését meg kell előzze a korábbi fázis befejezése.

22. Foglald össze a követelményelemzés céljait, módszereit, azok előnyeit és hátrányait!

- *"Annak azonosítása, hogy a felhasználók mit várnak el egy szoftver rendszertől"*
- *"Requirements are "designing the right thing" as opposedto software engineering's "designing the thing right"* (Boehm, 1981)
- A Requirements engineering az a folyamat, amely során kialakítjuk, dokumentáljuk és karbantartjuk a szoftverrendszerrel kapcsolatos elvárásokat
- A vízesés modellben a követelményelemzés az első fázisa a teljes fejlesztési folyamatnak
- A későbbi fejlesztési módszertanok (pl. RUP, XP,Scrum) feltételezik, hogy a követelményelemzés folyamatosan jelen van a rendszer fejlesztésének teljes élettartama alatt

Három fő tevékenységet tartalmaz:

- Eliciting requirements: Üzleti folyamatok azonosítása és dokumentálása, interjúk
- Analyzing requirements: A követelmények elemzése, annak eldöntése, hogy kellően egyértelműek, teljesek, konzisztensek
- Recording requirements: különböző formában történő dokumentálás

Módszerek:

- Stakeholder azonosítás és interjúk
  - Stakeholder bárki, aki kapcsolatban van a rendszerrel
  - Az interjú jól bevárt eszköz a követelményelemzéshez
  - Az interjúk személyes jellege egy oldott környezetet teremt, ahol fókuszált, hatékony a kommunikáció
- Szerződés jellegű követelménylista
  - Egy hagyományos módja a követelmények dokumentálásának
  - Összetett rendszerek esetén nagyon sok oldalas dokumentumot jelent
  - Erősségei
    - A követelmények egy kipipálható listáját adja, szerződést formál a projekt szponzor és a fejlesztők között
    - Nagy rendszerek esetén egy magas szintű leírást jelent, amelyből az alacsonyabb szintű követelmények származtathatók
  - Gyengeségek
    - Nagyon hosszú, nehezen olvasható
    - A részleteket elfedi, a teljes követelményhalmaz felfedése eltolódik a fejlesztés és a tesztelés fázisokra

        - Egy lista elkészítése nem garantálja a teljességet
- Use-case-ek
    - A rendszerrel szembeni funkcionális elvárások dokumentálásának egy struktúrája
    - Forgatókönyvek egy halmaza: a rendszer és a felhasználó, valamint a rendszer és más rendszerek közti interakció leírása
    - Nem tartalmaz technikai zsargont, a végfelhasználó, a szakterület nyelvezetét használja
    - Gyakran a követelményelemző és a stakeholder-ek együtt szerkesztik őket
    - Egyszerű eszközei a rendszerrel szembeni elvárások leírásának: nem definiálnak belső működést, implementációs részleteket. A feladat elvégzéseshez szükséges lépésekre koncentrálnak.
    - A felhasználó célja alapján készül
- Prototípus készítése
    - példa alkalmazás, segíti az új rendszer céljainak demonstrálását
    - segíti az követelmények pontosítását és véglegesítését
    - Wireframe és Mockup készítés: bevált módszerek, de nem ugyanaz a 3!
    - Támogatja a tervezési döntéseket
    - Az alkalmazás korai megismerése csökkenti a későbbi változtatások számát

Prototípus készítés: > példa alkalmazás, > segíti az új rendszer céljainak demonstrálását, > követelmények pontosítását és véglegesítését

23. Mutasd be a követelményelemzés során végzett aktivitásokat és azok jellemzőit!

- Requirments inception: követelmények felderítése
- Requirments identification: követelmények azonosítása
- Requirements analysis and negotiation: követelmények elemzése és egyeztetése, konfliktusok feloldása
- Requirements specification (Software Requirements Specification) – követelmények dokumentálása
- System modeling: a rendszer modellezése (különféle jelölésrendszerekkel, pl. UML)
- Requirements validation: követelmények validálása, annak ellenőrzése, hogy a dokumentált elvárások és a modellek konzisztensek, valamint, hogy teljesítik a stakeholder-ek elvárásait
- Requirements management: követelménymenedzsment, a változási kérések kezelése a fejlesztés és az üzembe helyezés során

24. Mutasd be a követelményelemzés során előtérbe kerülő különböző követelménytípusok jellemzőit!

- Felhasználói követelmények (Customer Requirements): Működésbeli elvárások a rendszerrel szemben
- Architekturális követelmények (Architectural Requirements)
- Strukturális követelmények (Structural Requirements)
- Viselkedésbeli követelmények (Behavioral Requirements)

- Funkcionális követelmények (Functional Requirements)
- Nem funkcionális követelmények (Non-functional Requirements)
    - Azon nem funkcionális követelmények, amelyek alapján a rendszer működése (nem a viselkedése) értékelhető
- Teljesítménybeli követelmények (Performance Requirements)
    - Egy mérték, amely alatt egy művelet lefut (mennyiségi, minőségi, lefedettségi, időbeni, készültségi mérték)
    - A követelményelemzés során iteratívan kerül pontosításra, hogy “mennyire jól kell működnie” a rendszernek, más tényezők figyelembe vételével (a rendszer életciklusa, a kritikusság foka, más rendszerekhez való kapcsolódás)
- Tervezési követelmények (Design Requirements)
- Származtatott követelmények (Derived Requirements)
    - Gyakran, magasabb szintű követelményekből származtatott elvárások.

25. Mutasd be a követelményelemzés során előtérbe kerülő kihívásokat!

- A felhasználók különféle módon akadályozzák a követelmények összegyűjtését
    - Nem tudják mit szeretnének vagy nincs egy tiszta képük az elvárásaikról
    - Nem tudnak elköteleződni egy definiált, leírt funkcióhalmaz mellett
    - Új követelményeket erőltetnek annak ellenére, hogy a költség és az ütemezés rögzített
    - A felhasználókkal való kommunikáció lassú
    - Nem vesznek részt az interjúkon, egyeztetéseken
    - Technikailag alulképzettek
    - Nem értik meg a fejlesztési folyamatot
    - Nem ismerik az aktuális technológiát
- Mindez olyan helyzetekhez vezethet, ahol a felhasználói követelmények folyamatosan változnak, még akkor is, amikor a rendszer vagy termékfejlesztés már elkezdődött

26. Foglald össze az üzleti elemzés céljait!

- Célja az üzleti igények azonosítása és megértése, az üzleti problémákhoz megoldások keresése, stratégiai irányok segítése, stratégiai célok elérésének támogatása
- Vállalat/szervezet elemzés
    - Üzleti architektúra készítése és karbantartása
    - Megvalósíthatósági tanulmányok készítése
    - Új üzleti lehetőségek azonosítása
    - Üzleti lehetőségekhez kapcsolódó scope definiálás
    - Üzleti forgatókönyvek előkészítése
    - Kockázati elemzések készítése
- Célok:
    - Megoldások létrehozása
    - Elegendő eszköz rendelkezésre bocsátása a robosztus project menedzsmenthez
    - Hatékonyság javítása, a veszteség csökkentése
    - Alapvető dokumentáció létrehozása
    - Projekt hatékénységának növelése, időre befejezés

27. Foglald össze az üzleti elemzés területeit, technikáit és jellemzőit!

Az üzleti elemzés területei:

- Követelménytervezés és menedzsment
    - Magas prioritású követelmények azonosítása
    - Változáskezelés
- Követelmények összegyűjtése
    - Brainstorming
    - Dokumentum elemzés
    - Fókuszált csoportmunka
    - Interfészelemzés
    - Interjúk
    - Workshopok
    - Felmérések
    - Felhasználói feladatok felmérése
    - Folyamatok leképzése
- Követelményelemzés és dokumentálás
    - Követelmények specifikálása olyan részletességgel, hogy azok sikeresen implementálhatók legyenek
- Elemzés
    - Architektúra elemzés
    - Üzleti folyamatok elemzése
    - Struktúra elemzés
    - Tárolás, adatbázis és adattárház elemzés
- Dokumentálás
    - Szöveges- pl. user story-k, amelyek bizonyos információkat összefoglalnak
    - Mátrix – pl. követelmények táblázata a prioritások megjelölésével
    - Diagramok - pl. adatfolyam az egyik struktúrából a másikba
    - Wireframe – pl. egy weboldal tartalma
    - Modellek
- Követelmények kommunikálása
    - Technikák, amelyek segítenek a stakeholderekkel megérteni a követelményeket és, hogy azok hogyan kerülnek implementálásra
- Értékelés és validálás
    - A javasolt megoldás helyességéről való meggyőződés
    - A megoldás implementálásának támogatása
    - Az implementálás során a hiányosságok kezelése

Az üzleti elemzés technikái:

- A SWOT elemzés 4 attribútuma
    - Erősségek (**S**trengths) – Mik az előnyök? Mit végzünk jelenleg jól?
    - Gyengeségek (**W**eaknesses) – Mely dolgok javíthatók? Mit végzünk rosszul?
    - Lehetőségek (**O**pportunities) – Milyen jó lehetőségek kínálkoznak a szervezet részére?

    - Veszélyek (**T**hreats) – Milyen nehézségekkel néz szembe a szervezet? (pl. kulcs terület, ahol a versenytárs jól teljesít)
- PESTLE: külső környezeti elemzés, amely vizsgálja a különféle, a szervezetre hatással bíró külső faktorokat
- 5 Whys:
    - Iteratív kérdés-válasz technika, célja a probléma körüli ok és okozat felderítés
    - Az elsődleges cél a gyökér (alap) probléma/hiba felderítése
- MoSCoW: a követelmények priorizálása, a követelmények egymáshoz viszonyításával
    - **M**ust have – egyébként a teljesítés hibás
    - **S**hould have – egyébként alternatív megoldást (workaround) kell keresni
    - **Co**uld have – hogy növeljük az elégedettséget
    - **W**ould like to have in the future – de most nem

Ami kimaradt a diasorokból:
Az üzleti elemzők többnyire specializálódnak a három terület valamelyikére:
- Strategist: a szervezet stratégiai céljainak, a környezetének az elemzése, a felső vezetés támogatása
- Architect: folyamatok, erőforrások elemzése, üzleti folyamatok áttervezése
- Systems analyst: az IT rendszerek jobb megértése és elvárásainak definiálása, cél az IT beruházások optimalizálása, erőforrások megfelelő hasznosítása

Üzleti folyamatok javítása:
1. Selection of process teams and leader: 2-4 fős csapatok, a csapat tagjai más-más részlegről, az adott folyamatban érintettek, csoportvezetőt választanak.
2. Process analysis training: A csapat tagjai üzleti elemzés és dokumentálási technikákból egy képzésen vesznek részt
3. Process analysis interview: A folyamatban érintett kollegákkal lefolytatott interjúk, a folyamat struktúrájának megismerése
4. Process documentation: Folyamattérkép készítése az interjúk alapján. A korábbi folyamatleírások tanulmányozása és integrálása. A lehetséges folyamatjavítások integrálása.
5. Review cycle: A folyamon dolgozó kollegák véleményezik a leírást. Iteratív folyamat: több review kör lefolytatására lehet szükség
6. Problem analysis: A folyamat problémáinak elemzése, a folyamat javítási lehetőségeinek származtatása.

Szoftver tervezési módszerek

28. Mutasd be a szoftvertervezés során alkalmazott alapelveket és koncepciókat!

A folyamat, amely során a szoftvertermék specifikációja készül el, mindazon aktivitás, amely a követelményspecifikáció elkészítése és a fejlesztés elkezdése között van.

- A szoftvertervezés eredményei modellek és dokumentáció (terméktípustól függően más és más jellegű és mennyiségű)
- A tervezés lehet platform-függő vagy platformfüggetlen
- A tervezés egy megoldás kidolgozása egy adott problémára a rendelkezésre álló képességek/tudás figyelembe vételével
- A célkörnyezettől függően más-más terv születhet ugyanarra a feladatra

Alapelvek:

- A szoftvertervezés folyamat és modell is egyben
- A tervezési folyamat lépések sorozata, mely során a tervező a szoftvertermék minden aspektusát részletesen kidolgozza
- A tervezési folyamat nem egyszerűen egy recept, kreativitás, jelentős tapasztalat, a "jó szoftverhez" való érzék/ráérzés, a minőség iránti elköteleződés mind kritikus tényezői a kompetens tervezésnek
- A tervmodell különböző absztrakciós szinteken mutatja be a rendszert más és más aspektusból
- Alternatív lehetőségek vizsgálata, azok erőforrás igényeinek, és a kapcsolódó kapacitások figyelembe vételével (csőlátás ne legyen)
- A tervmodell egy eleme gyakran több követelményhez is tartozik, a követelmény-terv kapcsolatokat igazolni kell
- Meglévő tervezési minták használata.
- A szoftverterv struktúrája, amennyire az lehetséges, imitálja a szakterület struktúráját
- Egységes szabályok, stílus használata: mintha egy kollega készítette volna az egészet.
- Változás elfogadásának tervezése
- A nem várt körülményekre is fel kell készülni, lassú degradáció
- A tervezési absztrakciós szint magasabb, mint a fejlesztési
- Számos eszköz, koncepció segít megmérni a terv minőségét, használjuk!
- Időt kell szánni a terv ellenőrzésére! A koncepcionális kérdéseken kell legyen a hangsúly!

Koncepciók:

- Abstraction – általánosítás, melynek során csökkentjük az információ tartalmát, adott célnak megfelelőkre koncentrálunk
- Refinement – kifejtés, részletek kidolgozása
- Modularity – a szoftverarchitektúra komponensekre bomlik, ezeket nevezzük moduloknak
- Software Architecture – a szoftver teljes struktúrája, valamint az a megközelítés, ahogy a struktúra a rendszer koncepcionális integritását adja
- Control Hierarchy - a program felépítése, a felépítésből következő program struktúra
- Structural Partitioning – horizontális felbontás : fő funkcionalitások alapján; vertikális felbontás: a vezérlés és a feldolgozás szeparálása

- Data Structure – az egyes adatelemek közti logikai kapcsolat
- Software Procedure – az egyes modulok feldolgozási képessége
- Information Hiding – az egyes modulokon belül tartsuk azon információt, amelyre más moduloknak nincs szükségük

29. Mi a szoftvermodellezés szerepe a szoftvertervezés során? Milyen módszereket, technikákat ismersz?

Modellezés szerepe:

- Magasabb absztrakciós szint
  - „Madártávlat"
  - Kódból nehézkes megérteni az általánosabb koncepciókat
  - Kóddal nehéz leírni őket
  - Viszont kezelni kell őket
- A csoport tagjai közti kommunikáció
  - A modell szemléletes
  - A modell szabványos (UML)
- CASE eszközök
  - Kód és modellgenerálás
  - Kód-modell szinkronizáció

Tervezés modellezéssel:

- Segít megérteni, tisztázni és kommunikálni a szoftvertől elvárt követelményeket.

- Egy részben működő alkalmazás (prototípus) érdemben stimulálja az egyeztetéseket. A modell egy hatékony módja ezen beszélgetések eredményeinek rögzítésére.

Modelltípusok és felhasználásuk

- Dependency graphs – Függőségi gráf
  - A kód felépítését és a kódon belüli kapcsolatokat segít áttekinteni. Függőségek, struktúra megértését segíti
- Layer diagrams – Rétegezett diagram
  - Segíti az alkalmazás struktúrájának definiálását: rétegek, blokkok, explicit függőségek. A layer diagram és a kód által definiált függőségek (automatikusan) összehasonlíthatók. A függőségi konfliktusok elkerülését/felderítését segíti
- UML model – UML modell
  - Számos nézet: osztály, komponens, use case, aktivitás, szekvencia diagramok. Testre szabható, így illeszthető az alkalmazás szakterületéhez
- Sequence diagrams – Szekvencia diagramok
  - Segít vizualizálni, hogy a kód hogyan implementál egy adott metódust/funkciót.
- A Domain-specific language (DSL) – Szakterületi nyelv
  - Egy jelölésrendszer speciális célokra.

30. Miért használunk a modellezés során eltérő absztrakciós szintű modelleket? Mutass rá példákat!

A különböző szereplők emberi természetüknél fogva is máshogy látják az üzleti világot. Az első feladat ezen eltérések feloldása.

Ami kimaradt a diasorokból:
Tervezési megfontolások:

- Compatibility – más termékekkel, korábbi verziókkal való együttdolgozás
- Extensibility – az új elemek/funkciók hozzáadása a meglévők jelentős módosítása nélkül tehető meg
- Fault-tolerance – a szoftver hibatűrő képessége, mennyire tud talpra állni az egyes komponensek hibáit követően
- Maintainability – a karbantarthatóság mértéke: hibajavítás, funkcionális módosítás
- Modularity – a moduláris szoftverek jól definiált, független komponensekből állnak. Jobb karbantarthatósághoz vezet. Az egyes komponensek külön-külön fejleszthetők és tesztelhetők
- Reusability – kevés módosítással adhatók hozzá új funkciók
- Robustness – stressz alatt is jól teljesít, nem várt bemenetekre fel van készülve. Pl. nem dől össze kevés memória esetén
- Security – rosszindulatú támadásokkal szemben ellenálló
- Usability – a felhasználói felület használható a célfelhasználók számára, pl. alapértelmezett értékek ki vannak töltve
- Performance – a felhasználók által elfogadható válaszidőn belül elvégzi a feladatát
- Portability – ugyanazon szoftver használható más-más környezetekben
- Scalability – a megnövekedett felhasználószámhoz vagy adatmennyiséghez jól adaptálódik

UML cuccok

<u>Módszertanok, Klasszikus módszertanok</u>

33. Mutasd be a szoftverfejlesztési folyamat lépéseit, röviden elemezd őket!

- Követelményelemzés
- Specifikáció készítés
- Tervezés
- Implementálás
- Tesztelés
- Debuggolás
- Telepítés / Üzembe helyezés
- Karbantartás

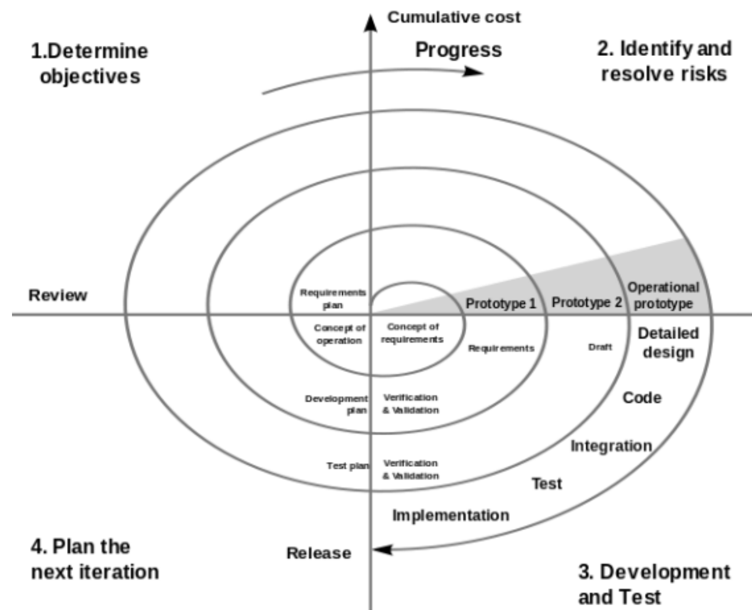
34. Hasonlítsd össze a vízesés, a spirál és az iteratív fejlesztési módszereket!

Vízesés:

- Folyamat, amelynek során a fejlesztés ebben a sorrendben végzi a következő lépéseket:
    1. Követelmény specifikálás (követelmény elemzés)
    2. Szoftvertervezés
    3. Implementálás és integrálás
    4. Tesztelés
    5. Üzembe helyezés (és telepítés)
    6. Karbantartás

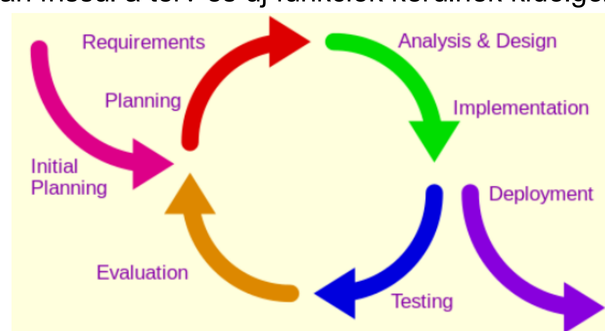
Spirál:

- Kulcs karakterisztika: kockázat menedzsment a fejlesztés megfelelő fázisában
- A projekt egyedi kockázatai alapján veszi át más folyamatmodellek elemeit (inkrementális, vízesés, evolúciós modellek)
- Négy fő aktivitást ismétel ciklikusan (spirál formájában):
    1. Tervek kialakítása: célok, elvárások, megkötések
    2. Kockázatelemzés: kockázatot jelentő tényezők azonosítása és eltávolítása
    3. Implementáció: alkalmazásfejlesztés és verifikáció
    4. Következő iteráció tervezése



Iteratív:

- A rendszer ismétlődő ciklusok (iteratív) és egy időben kisebb részek kidolgozásával (inkrementálisan) fejlődik
- Minden iterációban frissül a terv és új funkciók kerülnek kidolgozásra



35. Mutasd be az iteratív és inkrementális fejlesztési módszerek jellemzőit!

Lásd előző pont

36. Mutasd be a Capability Maturity Model Integration (CMMI) különböző területeinek céljait és jellemzőit!

- A CMMI egy folyamatokat javító modell, amely teljesítménybeli kérdések javításban segít, tetszőleges iparágra adaptálható.
- Útmutatásokat és javaslatokat ad a szervezeti problémák/szűk keresztmetszetek azonosítására és a hatékonyság javítására.
- Célja segíteni az üzleti célok azonosításában és elérésében (mérhető teljesítménybeli célok)

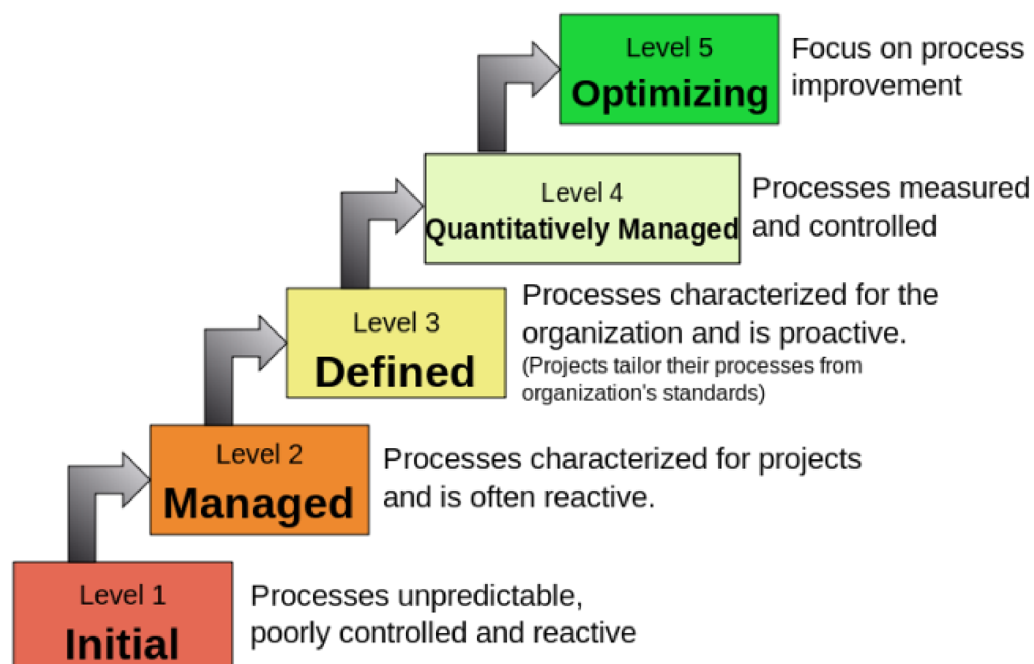
A CMMI folyamatmodell keretrendszerek jelenleg három területen érhetők el:

- The CMMI for Development (CMMI-DEV)
  - Termékfejlesztési folyamatok javítása

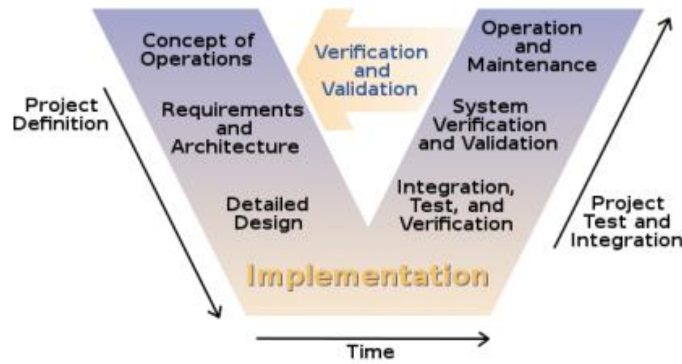
    - Hatékonyság, teljesítmény minőség ösztönzése
- The CMMI for Services (CMMI-SVC)
    - A végfelhasználók számára szolgáltatásokat létrehozó és fenntartó szervezetek folyamatait segíti
- The CMMI for Acquisition (CMMI-ACQ)
    - Beszerzés, szállítmányozás, termék és szolgáltatás integrálását végző folyamatok javítása

Minden CMMI modell
- Egy folyamatjavító megközelítés, ami a hatékony folyamatok alapvető építőelemeit adja
- Útmutatót ad a javításokhoz egy csapat, egy projekt, egy részleg vagy teljes szervezet számára. Segít kialakítani a folyamatjavítással kapcsolatos célokat
- Prioritásokban támogat, minőségi folyamatokat alakít ki

37. Mutasd be a Capability Maturity Model Integration (CMMI) különböző szintjeit és azok jellemzőit!



Ami kimaradt:
- **V-modell:**



- **RUP: Rational Unified Process**
  - Best practice-ek: tervezéshez, implementáláshoz, és hatékony projektmenedzsmenthez
  - RUP építőelemek:
    - Szerepkörök (who) – A szerepkör számos képességet, kompetenciát és felelősséget definiál.
    - Eredménytermékek (what) – Az eredménytermék (Work Product) a feladatok eredményeit jelentik, beleértve a teljes folyamat során keletkezett modelleket és dokumentációt
    - Feladatok (how) – A feladat egy szerepkörhöz rendelt munkaegységet definiál, melynek használható eredménye van
    - Minden iteráción belül a feladatok kilenc területre oszlanak fel, 6 mérnöki, 3 támogató
  - Fázisok: minden fázis egy vagy több iterációt tartalmaz
    - Inception (Kezdet) – A projektötlet kialakítása (a csapat eldönti, hogy érdemes-e a projektet folytatni/érdemben elindítani, valamint, hogy milyen erőforrás igénye lesz)
    - Elaboration (Kidolgozás) – A projekt architektúra és a szükséges erőforrásigény kidolgozása (a fejlesztők átgondolják a lehetséges alkalmazási területeket, valamint a fejlesztés költségeit)
    - Construction (Kidolgozás) – A project kidolgozásra kerül (a szoftver tervezése, fejlesztése és tesztelése)
    - Transition (Átadás) – A szoftver publikus kiadása, átadása (végső simítások, visszajelzések alapján történő korrekciók)
- **TDD: test driven development**
  - unit teszteket használ a fejlesztési munka vezérlőmotívumaként, mini ciklusa: Red-Green-Refactor
  - RED: Azzal kezdünk, hogy megírjuk a tesztet ami a kódot ellenőrzi, ez elsőre elhasal
  - GREEN: Megírjuk a kódot, teszt zöld lesz
  - REFACTOR: A kód működik - alakítsuk át úgy, hogy megfeleljen az elvárásoknak
  - tipikusan nagyon rapid
  - előnyei:
    - A kód minősége hasonló a hagyományos módszertannal készült kódéhoz – Pedig kevesebb tervezés volt az elején
    - Ritkán kell debuggolni – Ha nem működik, akkor kevés kódot kell csak pofozgatni

- Bár több kódot írunk (a tesztkóddal együtt), a fejlesztési idő mégis gyakran kevesebb
- Kevesebb hiba – a tesztek valójában a specifikáció leírása kódban
- A végén egy működő kód van, amihez rengeteg teszt tartozik
- A kódot később bátran lehet módosítani, bővíteni, hiszen a tesztek jelzik, ha valamit elrontunk

Agilis fejlesztési módszerek

38. Foglald össze az agilis módszerek értékeit és elveit!

- Egyének és interakció inkább, mint folyamatok és eszközök
- Működő szoftver inkább, mint jól dokumentált szoftver
- Felhasználó bevonása inkább, mint szerződések tárgyalása
- Reagálás a változásokra inkább, mint egy terv követése

Értékek:

- Kommunikáció
    - „What matters most in team software development is communication"
    - Minden probléma visszavezethető a kommunikációra
        - Programozó – Programozó
        - Programozó – Ügyfél
        - Manager – Programozó
    - Egy cél: minden fejlesztő ugyanúgy lássa a rendszert, ahogy a majdani felhasználók is látni fogják.
- Egyszerűség
    - „What is simplest thing that could work?"
    - Az agilis módszer arra fogad, hogy...
    - YAGNI •
    - Az egyszerűség segíti a kommunikációt
- Visszacsatolás
    - „Az optimizmus szakmai ártalom a programozóknál, és a visszajelzés rá a gyógyír."
    - „Elsőre jól?"
        - Nem tudjuk hogyan
        - Ha tudjuk, holnap már lehet a jó rossz
        - Sok és gyors visszacsatolással tudunk közel kerülni a jó megoldáshoz.
- Bátorság
    - Tenni valamit a félelem ellenére
        - Betartani a YAGNI-t
        - Refaktorálni
        - Revertálni
    - Vagy nem tenni
    - Főleg a többi értéket támogatja
        - Kimondani jót és rosszat
        - Eldobni a rossz megoldásokat
        - Igazi konkrét válaszokat keresni
- Respect
    - Ha nem törődünk egymással, akkor...
    - Ha nem törődünk a projekttel, akkor...
    - „I am important and so are you"

Elvek:

- Humanity: Business and personal needs
- Economics: Somebody has to pay for all this.
- Mutual Benefit: Every activity should benefit all concerned
- Self-Similarity: Copying a structure of a solution to new context
- Improvement: In software development „perfect" is a verb, not an adjective
- Diversity: Conflict comes with diversity
- Reflection: Good teams don't just do their work, they think about "how" and "why" 19
- Flow: Steady flow of valuable sw
- Opportunity: Problems as opportunity for change
- Redundancy: Redundancy of practices
- Failure: If you can't succeed, fail
- Quality: Quality is not a control variable
- Baby Steps: Many small steps rapidly looks like a leap
- Accepted Responsibility: Responsibility cannot be assigned

39. Foglald össze az agilis tervezés jellemzőit és szintjeit!

Agilis tervezés:

- Csapatmunka
- Rövid iterációk
- Prioritás
- Adaptáció

Tervezés célja:

- Kockázatok csökkentése
- Bizonytalanság csökkentése
- Döntéstámogatás
- Bizalomszerzés
- Információátadás

A tervezés szintjei:

- Vízió: mit szeretnénk elérni?
  - Dokumentum: mi a cél, miért jó, siker kritériumai
- Kiadás tervezés
- Iteráció tervezés
- Stand-up

40. Mutasd be az agilis módszerek során alkalmazott kiadás tervezést és annak jellemzőit! Térj ki a felhasználói sztorik szerpére!

Kiadás tervezés:

- Gyakori kiadások
  - Nem feltétlenül a végfelhasználónak
  - ~3 hónap
  - NEM lehetetlen: https://drive.google.com/#my-drive (http://www.writely.com)
  - A felhasználó jobban átlátja, könnyebben veszi át

        - Korábban problémás volt
- Egyszerre egy projekt

Felhasználói sztorik:

- Céljuk, hogy tervezni tudjunk
- Olyan dolgot ír le, amit a csapatnak meg kell valósítania
- Alistair Cockburn: „ígérvény egy jövőbeni megbeszélésre”
- Megrendelő számára értékkel rendelkezik
- Teljesítési kritérium
- Mini vízió
- [VALAKI]ként azt szeretném elérni, hogy [CÉL] azért, hogy [MIÉRT].

41. Foglald össze az agilis iterációk jellemzőit és az iteráció tervezés lépéseit!

Iteráció:

- Az iteráció az agilis projekt szívverése
- Iteráció hossza: 1 vagy 2 hét
- Időkorlát
- Haladás mértéke
- Az iteráció nem előzi meg a problémát, csak felszínre hozza

Lépései:

- Demo
- Visszatekintés
- Iteráció-tervezés
    - Sebesség
    - A legértékesebb sztorik a kiadás tervezési tábláról
    - Mérnöki feladatokra bontás
        - Technikai, nem felhasználó-centrikus
        - Szoftvertervezés
    - Becslés
        - Ideális óra
    - Néhány óra
    - Kiadás tervezés nem itt történik •
    - A felhasználó és a termékmenedzser már az előző iteráció közben kifejti a sztorikat
- Vállalás
- A sztorik megvalósítása
- A kiadás elkészítése

42. Mutasd be a napi stand-up esemény és a “Kész, kész” (done, done) jellemzőit!

Stand-up:

- Naponta 1x (2x)
- Max. 10-15 perc

- Napi tervezés
- Felállva!!!
- Ha másképp nem megy
  - Mit csináltam tegnap?
  - Mit fogok csinálni ma?
  - Mi akadályoz meg abban, hogy haladjak?

“Kész, kész” (done, done):
- Megtervezett
- Lekódolt
- Integrált
- Tesztelt
- A build lefut
- A termék installálódik
- Migrálódik
- A végfelhasználó ellenőrizte és elfogadta
- Megfixált

43. Foglald össze az eXtreme Programming (XP) jellemzőit és eszközeit!

Az XP egy fejlesztési módszertan kb. 9 fős fejlesztői csoportok számára.
Az XP legfontosabb jellemzői:
- A programozók párokban programoznak.
- A fejlesztést a tesztesetek irányítják.
  - Először tesztelünk, utána kódolunk. Amíg nem fut az összes teszteset, nem vagyunk készen. Amikor az összes teszt fut, és nem jut eszünkbe olyan teszt, ami esetleg hibát jelezne, kész vagyunk az új funkciók hozzáadásával.
- A párok nem csak a teszteseteket teszik futtathatóvá, hanem a rendszer tervezését is előremozdítják. A változtatások nem csak egy adott területet érintenek. A párok szervesen részt vesznek a rendszer analízisének, tervezésének, implementációjának és tesztelésének értékesebbé tételében. Értékesebbé teszik a rendszert bárhol, ahol szükséges.
- Az integráció közvetlenül követi a fejlesztést, beleértve az integrációs teszteket.

Eszközök?
- Kód ellenőrzés - párprogramozás
- Tesztelés – mindenki mindig tesztel
- Tervezés – mindenki mindig tervez (refactoring)
- Egyszerűség
- Integráció – folyamatos integrálás
- Architektúra – mindenki finomítja
- Rövid iterációk

Technikák:
- Planning game:
  - Üzleti szakemberek eldöntik a hatókört, prioritásokat, kibocsátott verzió összetételét, a kibocsátott verziók dátumát
  - A műszaki szakemberek eldöntik a becsléseket, üzleti döntések következményeit, folyamatot, részletes időbeosztást

- Kis méretű kibocsátott verziók
- Az egyszerű terv
    - minden tesztet futtat
    - nincs duplikált logika
    - kifejezi a programozó szándékát
    - a legkevesebb osztályt és metódust tartalmazza
- Tesztelés
- Kódátírás (refactoring)
- Programozás párban
- Közös kód
- Folyamatos integráció
- 40 órás munkahét
- A megrendelő rendelkezésre áll helyben
- „Az egyszer és csakis egyszer” szabály

44. Foglald össze az eXtreme Programming (XP) értékeit és elveit!

Négy érték:

- Kommunikáció
- Egyszerűség
- Visszacsatolás
- Bátorság
    - A hiba kijavítása – nem hackelés
    - A nem szükséges kódot eldobjuk
    - Ha valami átláthatatlan, újraírjuk

Alapelvek:

- Gyors visszacsatolás
- Az egyszerűség feltételezése
- Növekményes változtatás
- A változás felvállalása
- Minőségi munka

További elvek:

- Taníts tanulni
- Kis kezdeti befektetés, utána fokozatosan
- Konkrét tapasztalatok a kockázatelemzésben
- Nyílt, egyenes kommunikáció
- Ne fáraszd a programozókat fölöslegesen
- Vegyük figyelembe a környezetet
- Becsületes mérőszámok

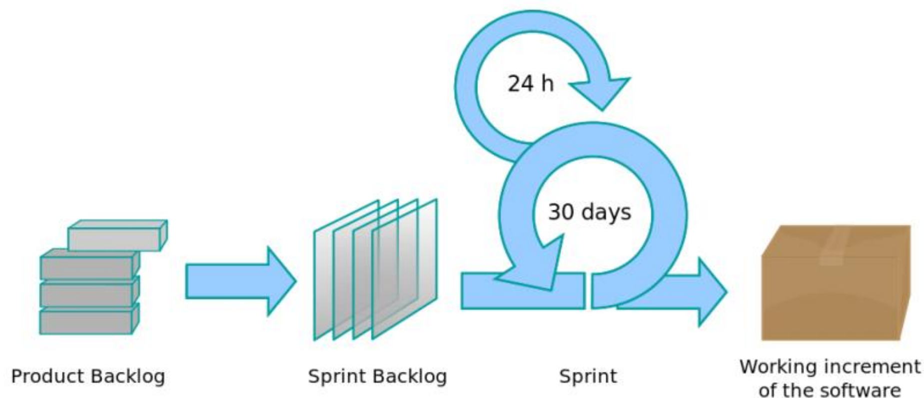
Fontosabb alapelvek (diasor más részéről…)

- Kommunikáció (épít a csoport jó kapcsolatára)
- Egyszerűség (ha lehet, apróbb változtatások)
- Visszacsatolás (sok teszt)
- Lelkesedés (iteratív fejlesztés, idő a jó kódra)
- 40 órás munkahét
- A felhasználó a bevonása a fejlesztői csoportba

45. Mutasd be a Scrum fejlesztési folyamatot és jellemzőit!

Folyamat:

- Minden sprint (2-4 hét) során a csapat egy működő szoftveregységet hoz létre.
- A sprint során megvalósítandó funkciók a Product Backlog-ból (termék teendőlistája) kerülnek ki, ami az elvégzendő munka magas szintű követelményeiből álló, fontossági sorrendbe állított lista.
- A Product Owner közli a csapattal, hogy a teendők listájából melyek azok, amiket leghamarabb szeretné, hogy elkészüljenek.
- Ezután a csapat eldönti, hogy ezek közül melyek azok, amelyeket a következő sprint során meg tud valósítani, és ezek megvalósítására ígéretet tesz.
- A sprint folyamán a Spring Backlog-ot nem lehet megváltoztatni, a sprint során elvégzett tevékenységek rögzítettek.
- Amint a sprint a végéhez ért, a csapat bemutatja az elkészült funkciókat (demo).



Jellemzők:

- A Scrum egyik legfontosabb alapelve az, hogy felismeri és elfogadja, hogy a megrendelő a fejlesztés során meggondolhatja magát a követelményekkel kapcsolatban, és a váratlan változások nem kezelhetők könnyen a hagyományos, előzetes tervezési fázison alapuló módszerekkel.
- Ezért a Scrum gyakorlati megközelítést választ, és elfogadja hogy nincs lehetőség a probléma teljes megértésére és definiálására. Inkább azt próbálja maximálisan elősegíteni, hogy a csapat gyorsan meg tudja valósítani a funkciókat és gyorsan tudjon reagálni a változó követelményekre.

46. Mutasd be a Scrum ceremóniák jellemzőit!

4 megbeszélésfajta, Scrum, Sprint planning, Demo, Retrospective

Scrum: napi megbeszélés

- A megbeszélés ideje 15 perc
- Állva: ez elősegíti, hogy a megbeszélés ne húzódjon el
- Minden nap ugyanazon a helyen és ugyanabban az időpontban tartják
- A megbeszélés során minden résztvevő ugyanazokat a kérdéseket válaszolja meg:
    - Mi az, amit a tegnapi megbeszélés óta csináltam?
    - Mi az, amit a mai nap tervezek csinálni?
    - Vannak-e akadályok, amik gátolnak a cél elérésében? (Az akadályok elhárítása a Scrum Master feladata.)

Demo:
- Annak áttekintése, hogy mely munkák készültek el és melyek nem.
- Az elkészült munka bemutatása a terméktulajdonos és a fejlesztésben érdekeltek részére (demo).

Retrospective:
- A csapattagok véleményt alkotnak az elmúlt sprintről. A vélemény lehet egy puszta benyomás is, nem kell kidolgozott, szilárd álláspontnak lennie.
- Javaslatokat tesznek a folyamatok továbbfejlesztésére. A javaslatoknak nem kell kisérleltnek lenniük, a kidolgozás nem a visszatekintés része.
- Két kérdés merül fel a megbeszélésen:
    - Mi az, ami jól ment a sprint alatt?
    - Mi az, amit a következő sprint során jobban lehetne csinálni?

47. Mutasd be a Scrum szerepköröket és a sprint tervezés jellemzőit!

Szerepek :
- Product Owner
    - A megrendelőt képviseli
    - Biztosítja hogy a csapat az üzleti szempontból fontos dolgokkal foglalkozzon
- Scrum Master
    - Feladata hogy elhárítsa az akadályokat amelyek gátolják a csapatot abban, hogy a sprint célját megvalósítsa
    - Nem a csapat vezetője (a csapat önszerveződő)
- Team
    - A csapat azért felelős hogy a termék elkészüljön
    - 5-9 főből áll

Sprint Planning:
- Elvégzendő feladatok kijelölése a termék teendőlistájáról (product backlog) a terméktulajdonos közreműködésével.
- A sprint teendőlistájának előkészítése, amely a teljes csapat figyelembevételével részletezi az egyes részfeladatok időszükségleteit.
- Annak meghatározása és kommunikálása, hogy mennyi feladat elvégzése várható el az aktuális sprint során.

Ami kimaradt a diasorokból:
Az agilis fejlesztés 12 pontja:
1. A legfontosabb dolog a megrendelő kielégítése úgy, hogy a kezdetektől folyamatosan értékes szoftvert szolgáltatunk.
2. Üdvözöljük a változtatási kérelmeket, még a fejlesztés késői szakaszában is. Az agilis folyamatok a megrendelő versenyelőnye érdekében hasznosítják a változtatásokat.
3. Működő szoftver gyakori kiadása (pár hét – pár hónap) minél rövidebb időszakokkal.
4. Az üzleti megrendelőnek és a fejlesztőknek napi szinten együtt kell működniük.
5. A projekteket jól motivált egyénekre kell építeni. Adjuk meg nekik a környezetet és a támogatást, és bízzunk bennük, hogy a rájuk bízott feladatot elvégzik.
6. A leghatékonyabb és leghatásosabb módja az információáramlásnak a csapaton belül és azon kívül a személyes beszélgetés.

7. A haladás mértéke a működő szoftver.
8. Az agilis folyamatok a fenntartható fejlesztést támogatják. A megrendelők, a fejlesztők és a végfelhasználók egy állandó sebességet tudnak fenntartani gyakorlatilag végtelen ideig.
9. A technikai tökéletességre és a jó dizájnra való törekvés növeli az agilitást.
10. Az egyszerűség – az el nem végzett munka maximalizálása – alapkövetelmény.
11. A legjobb architektúrát, követelményt és dizájnt önszerveződő csoportok tudják kitermelni.
12. Bizonyos időközönként a csapat megvitatja, hogyan lehetne hatékonyabb és ennek megfelelően hangolja a viselkedését.

Kockázati tényezőkre példák (+ az XP válaszai erre)

- Késés az ütemtervhez képest: eljön az átadás napja, és közölnünk kell a megrendelővel, hogy a szoftver nem fog elkészülni további hat hónapon belül.
  - az XP rövid kibocsátási ciklusokat ír elő, legfeljebb néhány hónapot, így késés csak korlátozott mértékben fordulhat elő.
  - Egy kibocsátási ciklusban az XP 1-4 hétig terjedő iterációkat használ a felhasználó által kért funkciók implementálására, így egészen pontosan számon tudja tartani a projekt előrehaladását.
  - Az XP előírja, hogy a legfontosabb funkciókat kell legelőször implementálni, ennélfogva a kisebb értéket képviselő funkciók maradnak ki a kibocsátott verzióból.
- A projektet leállítják: számos csúszás után a projektet megszűntetik anélkül, hogy egyáltalán valaha is termék lett volna belőle.
  - az XP arra kéri a megrendelőt, hogy a lehető legkisebb kibocsátott verziót kérje, ami a leghasznosabb a szakterület számára
- A rendszer összekuszálódik: a szoftverből termék lesz ugyan, de néhány év elteltével a változtatás költsége és a hibaarány olyannyira megnő, hogy az egész rendszert ki kell cserélni.
  - Az XP létrehoz és karbantart egy átfogó tesztcsomagot, amelyet minden változtatás után újra és újra lefuttat (naponta többször), ezáltal egy összehasonlítási alapot biztosít a minőség folytonos biztosítására.
  - Az XP mindig elsőrendű állapotban tartja a rendszert, így a rossz megoldások nem halmozódhatnak fel.
- Hibaarány: a szoftverből termék lesz, de olyan magas a hibaarány, hogy nem használják.
  - Az XP nem csak a programozók szempontjából tesztel, akik minden függvényhez teszteket írnak, hanem a megrendelők szempontjából is, akik minden funkcióhoz írnak/kérnek/rendelnek tesztet.
- Szakterületi félreértés: a szoftverből termék lesz, de nem oldja meg a szakterület eredetileg megfogalmazott problémáját
  - Az XP előírja, hogy a megrendelő szerves része legyen a fejlesztői csapatnak.
  - A projektspecifikáció a fejlesztés során folyamatosan alakul, így amit a fejlesztők és a megrendelők megtanultak, visszatükröződhet a szoftverben.
- Szakterületi változások: a szoftverből termék lesz, de az általa megoldott szakterületi probléma helyét már hat hónapja átvette egy másik, sokkal sürgősebb szakterületi probléma.

    - Az XP lerövidíti a kibocsátási ciklust, így kevesebb változás történik egy kibocsátott verzió fejlesztése alatt.
    - Egy kibocsátott verzió fejlesztése alatt a megrendelő a még el nem készült funkciókat bármikor helyettesítheti új funkciókkal.
    - A fejlesztők észre sem veszik, hogy egy újonnan felfedezett vagy egy évek óta specifikált funkción dolgoznak
- Nem kívánt funkciókban gazdag: a szoftver számos, esetleg érdekes funkciókat tartalmaz, amelyek mindegyikét élvezet volt implementálni, de a megrendelőnek egyik sem jelent nagyobb bevételt.
    - Az XP kitart amellett, hogy csak a legmagasabb prioritású feladatok kerülnek sorra.
- Személyi változások: két év elteltével az összes jó programozó megutálja a programot és elmegy a cégtől.
    - vállaljanak felelősséget + ad visszajelzést
    - Az arra vonatkozó szabályok, hogy ki készíthet, illetve változtathat becsléseket, egyértelműek. Így kisebb az esélye annak, hogy a programozót frusztrálja, hogy a nyilvánvaló lehetetlen teljesítésére kérték.
    - Az XP elősegíti a csapatok közti emberi kapcsolatokat
    - Végül az XP tartalmaz egy explicit modellt a személyi változások kezelésére. Az új fejlesztőket segíti abban, hogy fokozatosan egyre több felelősséget vállaljanak, segítséget kapnak

XP fejlesztés:

- Planning game:
    - Felfedezés (Exploration) > Írj egy történetet > Becsüld fel a történetet > Darabold fel a történetet (mi lényeges, mi nem)
    - Bevállalás (Commitment) > Rendezés érték, kockázat szerint > Mennyi idő kell hozzá? > A kibocsátott verzió összeállítása
    - Irányítás (Steer) > Iterációk > Hibajavítás (mégis több idő kell hozzá, módosítás) > Új történet > Újra felbecsülni a tervet
- Tervezés
    - Egyszerűség
    - A tanultakat vissza kell fordítani a tervbe
    - Rövidítsük le az iterációs ciklus idejét, amennyire csak lehet
    - Kis kezdeti befektetés
    - Ne tervezz a jövőre, mert lehet, hogy soha nem jön el
- Kódolás
    - Legfontosabb, amit a kódból kinyerhetünk, az a tanulás
    - Jó minőségű kód: világos és érthető – lehetőség a kommunikációra
    - Kódátírás (refactoring) – ha tanultunk valamit az eddigiekből, írjuk át!
    - Kollektív tulajdon – bárki módosíthat bárhol
    - Folyamatos integráció
    - Nincs duplikált kód!!!
- Tesztelés
    - A teszt jelenti azt, hogy a kód készen van
    - Nem egy teszt: az összes lehetséges dologra kell tesztet írni, ami nem működhet
    - Programozás + teszt: gyorsabb, mint csak programozni

Srcum artifacts:

- Product Backlog: Termék teendőlistája prioritás szerint rendezve
- Sprint Backlog: Sprint teendőlistája
- Burn down chart: napi eredmény-kimutatás

Scrum kifejezések:

- story - a termék funkcióinak magas szintű, megrendelőközpontú leírása
- product backlog - a projekt során megvalósításra váró teendők listája, fontossági sorrendben
- sprint backlog - konkrét feladatok a következő sprintre
- backlog item - teendő
- sprint (futam) - a sprinttervezés során kiválasztott teendők megvalósítására szánt rövid iteráció (2-4 hét)
- scrum - rövid napi találkozó, ahol megbeszélik az eredményeket, az akadályokat és a következő teendőket
- sprint planning session - megbeszélés, amelyen a következő sprint teendőit definiálják
- sprint retrospective - visszatekintés, célja a fejlesztési folyamat gyengeségeinek elhárítása, a hatékonyság javítása. Minden csapattag elmondja a véleményét az utolsó futammal kapcsolatban, és a csapat megegyezik hogy mit változtatnak a fejlesztési folyamaton következő futam során.
- burn down chart - kimutatás a napi eredményekről a futam során

<u>Forráskód-kezelő eszközök</u>

48. Mik az előnyei a verzió kezelő eszközök használatának? Milyen verzionálási modelleket ismersz, mik ezek előnyei, hátrányai?

Előnyök:

- Potenciális ütközések kezelése
- Biztonsági mentés
- Visszaállás adott verzióra
  - Például kiadott éles verzió javítása
  - Félresikerült módosítások visszavonása – rövid/hosszú távon
- Változások követése
- Felelősök követése
- Hozzáférés szabályozás
- Elágazás és összefésülés

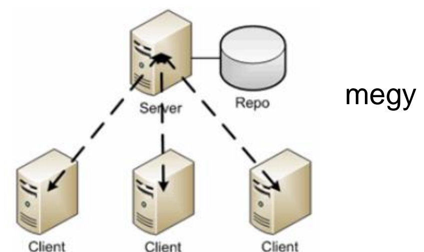
Verzionálási modellek:

- Zárolás (pesszimista): Lock-Modify-Unlock
  - Nagy csapatnál kényelmetlen, várni kell
  - Offline működés nem támogatható
  - Mindig a legfrissebb verzióval dolgozunk, a változásokat le kell tölteni szerkesztés előtt
  - Bináris, office fájlokhoz ideális lehet
- Zárolás nélkül (optimista): Copy-Modify-Merge
  - Merge-elni kell
  - Nem kell várni, bármi módosítható
  - Offline működik
  - Nem kell a változásokat beszinkronizálni szerkesztés előtt

49. Ismertesd a központosított és elosztott verziókezelés előnyeit, hátrányait!

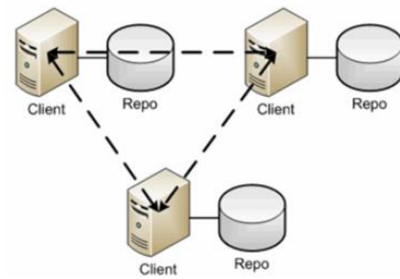
Központosított:

- Szerver – kliens működés
- Minden változtatás a központ repositoryn keresztül megy
- Egyszerű
- Teljes kontroll a hozzáférés fölött
- Több, jobban működő kliens
- Bizonyos műveletek szervert igényelnek



Eloszott:

- Minden felhasználónak van a saját repositoryja
- Az önálló repositoryk hálózatként működnek
- A megosztáson kívül minden helyi művelet, nincs szerver függőség (SPoF)
- A teljes history helyben lehet
- Egyszerűbb elágaztatás
- Bonyolult
    - Felhasználóknak nehezebb, kevés jó eszköz



50. Milyen elágaztatási stratégiákat ismersz? Jellemezd őket röviden! Jellemezd Git legfontosabb fogalmait!

Célok:

- Hatékony munkavégzés biztosítása
- Munkafolyamatok izolálása
- Jogosultság osztás

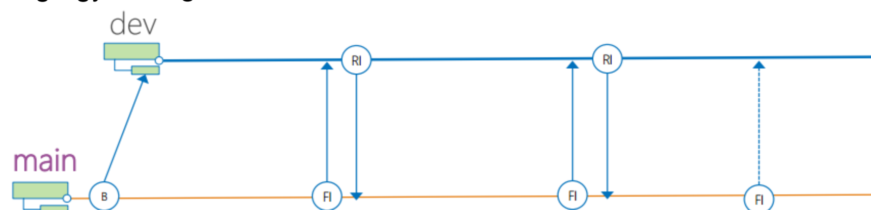
Ág típusok:

- Main
    - A Development és Release ágak találkozása
    - Mindig fordul, a QA csapat ezt használja
- Development
    - Független a többi ágtól, a fejlesztés normál ütemben folyhat
    - Gyakori merge javasolt
- Release
    - Kiadott verziók támogatása
    - A hotfixeket vissza kell vezetni a Main/Dev ágakba

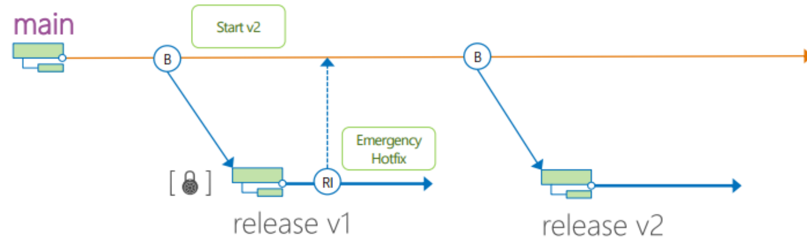
Stratégiák:

- Main only
    - Egyszerű
    - Címkék jelzik a fejlesztés diszkrét állapotait
        - A címkék helyéről bármikor indítható új ág ha szükséges

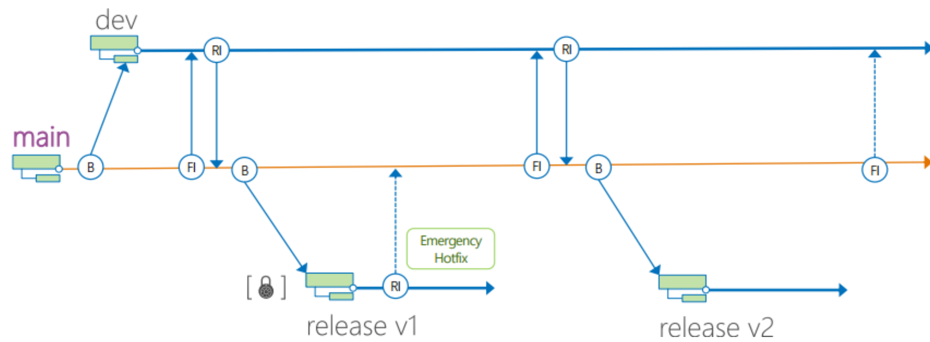
- Development isolation
    - Egy vagy több párhuzamos ág fut Main mellett
        - Akár minden fejlesztőnek egy külön ág….
    - Ezek lehetnek feature fejlesztések, bugfixek stb.
    - Mindig egy támogatott release verzió van, a Mainből



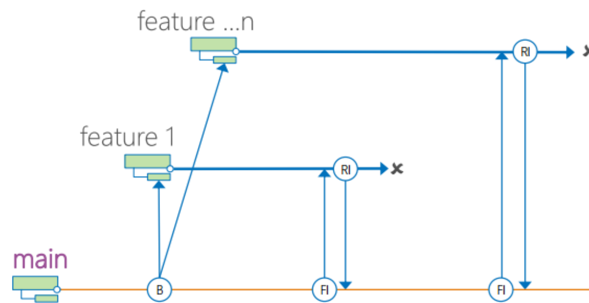
- Release isolation
    - Több release ág, párhuzamos release támogatás
    - A release ágakon ritkán van változtatás
        - Legfeljebb hotfix



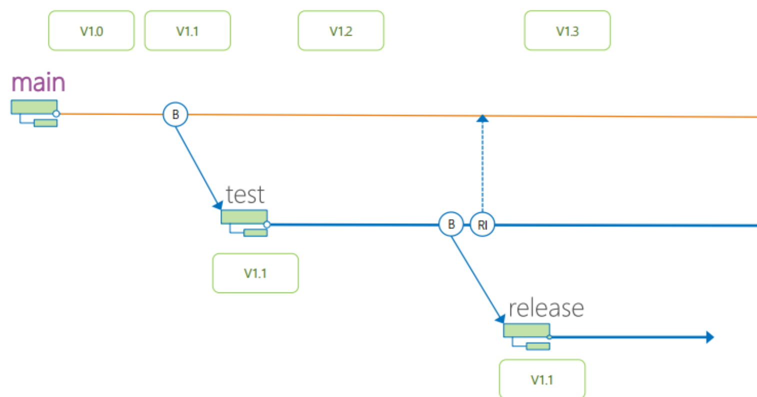
- Development and release isolation
    - A két előző stratégia kombinálása
        - Párhuzamos release-ek karbantartása
        - Izolált fejlesztési ág(ak)



- Feature isolation
    - A feature-ök külön ágat kapnak
    - Akkor mozgatható a Mainbe, amikor készen van
        - Vagy amikor a Megrendelő kéri



- Code promotion
    - Vízesés jellegű, minőség alapú ágak
    - Hosszú teszt ciklus



Branching alternatívák:

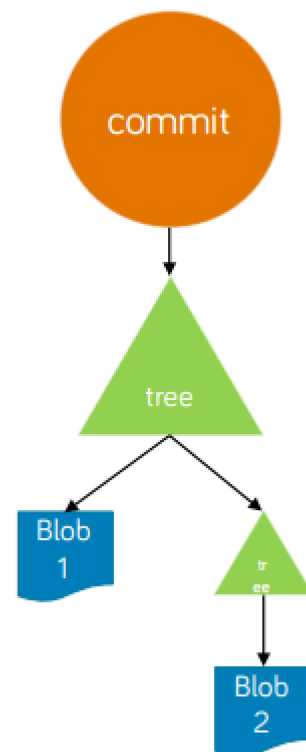
- Shelving / stashing
    - “Elmentjük későbbre”
- Labeling / tagging
    - Megjelöljük az állapotot
    - Később is lehet belőle leágazni

Git fogalmak:

- **Working tree:** egy mappa a fájlrendszerben, ami hozzá van kötve egy repositoryhoz (van benne egy .git almappa)
- **Commit:** a working tree egy snapshotja az időben valamikor. A commit szülője a HEAD.
- **HEAD**: azt mutatja, hogy mi van a working treeben (checkoutolva) :
    - Lehet egy branch: akkor a commit művelet után ezt a branchet kell frissíteni
    - Vagy egy specifikus commit is lehet checkoutolva
- **Branch**: valójában csak egy commit neve / referenciája. A commitok szülői jelentik a fejlesztés folyamatát
    - master: a fő fejlesztési ághoz tartozó branch
- **Tag**: hasonlóan egy commit neve, de nem változik, mindig ugyanarra a commitra hivatkozik
- **Repository:** a commitok gyűjteménye (ahogy a mappa kinézett a múltban), akár ezen a gépen akár másikon
- **Index / staging area**: a változások köztes tárolója, a commit ezeket véglegesíti, így rugalmas
- **Git reflog:** log minden parancsról

51. Jellemezd a git legfontosabb objektumait, adj példát egy módosítási művelet során végbemenő változásokra!

Git: nyílt forráskódú elosztott verziókezelő rendszer, tartalomra fókuszál, nem fileokra



A gitben objektumok/entitások vannak, mindegyiknek van azonosítója (a tartalmából generálva) és típusa, a parancsok ezeken az objektumokon dolgoznak. A .git/objects mappában vannak, fomátumuk zlib, a fájlnév az azonosító.
"The git blob type is just a bunch of bytes that could be anything, like a text file, source code, or a picture, etc."

Blob:

- Egy fájl tartalma
- Minden fájlhoz számolható egy azonosító: tartalom + hossz SHA1 hash-e
- Az azonosító 40 karakter, de 6 általában elég
- Ha a fájl megváltozik, más lesz az azonosító
    - A git csak olvasható tartalommal dolgozik!
- Az azonosító ugyanaz, bárhol van a fájl
    - Akár a fában / mappa hierarchiánban máshol
    - Akár másik repositoryban vagy akár gépen …
- A fájl tartalma blob-ban van tárolva
    - Nincs hozzá metaadat (név se), csak tartalom
- Fájl / mappa alapú
    - Kivételek a .gitignore fájlban adhatók meg

Tree:

- A mappa hierarchia leképezése
- A blobok (fájlok) a fa levelei
- Egy csomópont tartalmazza a „mappa":
    - leveleiként a blobok (fájlok) azonosítóit
    - a hozzájuk tartozó metaadatokat (például név)
    - benne lévő további gyerek csomópontokat a fából

Commit:

- A working tree tartalma egy adott időpontban
- Egy commit tartalmazza
    - Az index gyökerének azonosítóját
        - Index: ahol a commit tartalma „összeáll"
        - Gyökér: tree node, a gyökér mappa
    - Ki végezte a commitot (email címmel)
    - Mikor (időzónával)
    - Mi az előző (szülő) commit azonosítója
        - Vagy több commité, ha összefésülés történt
    - Commit üzenet

Tag:

- Egy tetszőleges objektum neve >
    - Létrehozóval, időpecséttel
- Tipikusan commitekre hivatkozik
- Például egy release adott verziójának megjelölésére szolgál

-

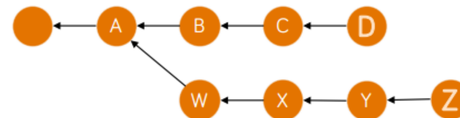
Módosítási művelet:

- Változtatás, új file:
  - Minden commit végén az index kiürül, várja az új változásokat
    1. Megváltozik egy fájl a working treeben a diszken
    2. Az -add paranccsal bekerül az új blob az indexbe
    3. A mappához tartozó tree objektum megváltozik, mert benne más a fájl mutató
    4. Minden szülő mappa megváltozik, mert más a mutató, egészen a gyökérig
- Változtatás, átnevezés:
  - A blob marad a régi, hiszen a tartalom nem változott meg
  - A tree objektum új lesz, mert más a neve
  - A szülők a gyökérig ugyancsak újat

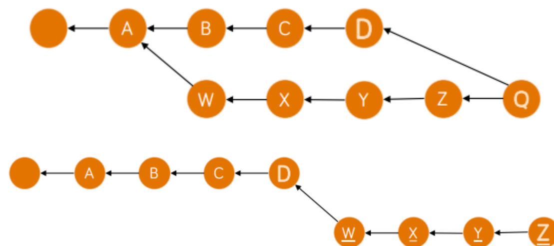
52. Jellemezd a git rebasing és a merge műveleteit!

Merge:

- Munka folyik mindkét ágon



- A Z ágon végzett munkát szeretnénk visszafűzni a D ágba



Rebasing:

- Munka folyik mindkét ágon
- A Z ágon végzett munkát szeretnénk tovább vinni, mintha megszereztük volna a legújabb változtatásokat a D ágon
- Csak lokálisan működik
- A rebasing során megváltozik az egyes commitok tartalma
  - A -> W helyett D -> W lesz a tartalma
  - És az összes többi commit tartalma is megváltozik
- Ha a W..Z-ből bárki más leágazott, akkor gond van, hiszen ezek a commitok megváltoznak

Rebasing vs. merge:

nagyon jó tutorial video rebaseről https://www.youtube.com/watch?v=cSf8cO0WB4o

- rebasing: egy javítás vagy más helyi munka tovább vitele ha a szerver változott
  - Helyi ág, amiből nem nőttek más ágak
- merge: vissza szeretnénk vezetni a helyi változásokat a főágba

Ami kimaradt a diasorokból:

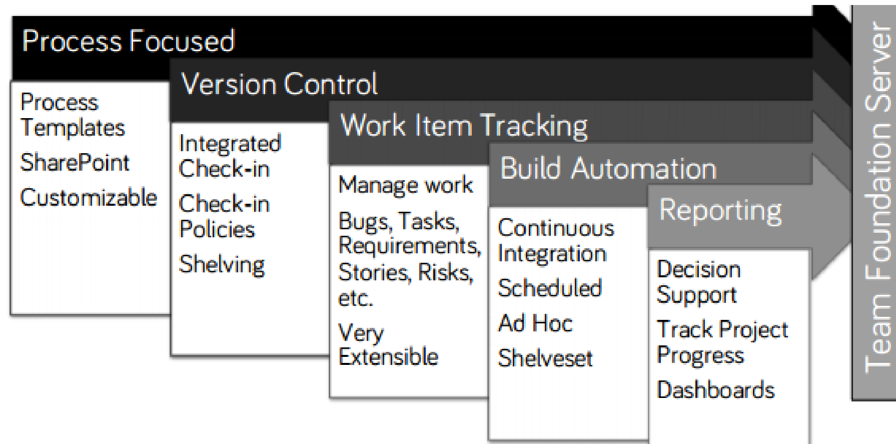
Branching antipatterns:

- Merge paranoia: merge-ölés elkerülés minden áron
- Merge mania: állandó merge-ölés és konfliktus feloldás
- Big Bang merge: a merge-ölés a fejlesztés végére marad ahol az összes ágat egyszerre fűzik össze
- Never ending merge: sose ér véget a folyamatos újabb fejlesztések miatt

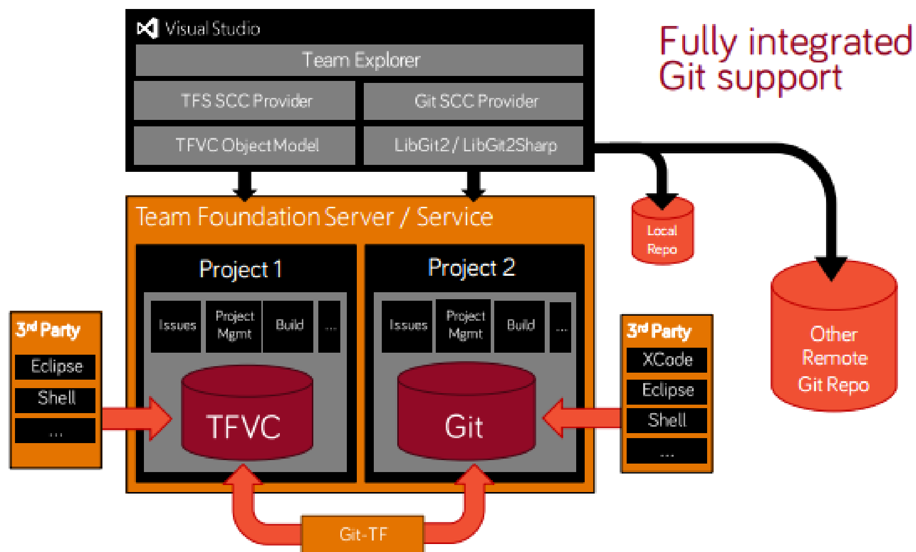
- Wrongway merge: rossz helyre kerül a merge
- Branch mania: új ágak születnek ok nélkül
- Mysterious branches: senki sem tudja miért vannak az ágak
- Cascading branches: az ágak sose kerülnek visszafűzésre a főágba
- Development freeze: nincs fejlesztés amíg a merge meg nem történik
- Berlin Wall: az ágak elválasztják a fejlesztőket ahelyett, hogy a munkájukat választanák szét

TFS:

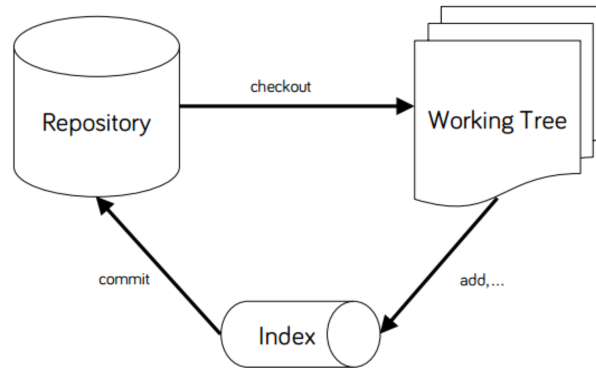
Szolgáltatásai:



Architektúrája:



Git folyamatok:



Git vs TFS:

- A git snapshotokkal dolgozik, a TFS (és mások) különbségekkel
- Műveleteik teljesen mások:
  - Team Foundation Server:
    - Get-Latest: a szerverről leszedjük a gépre a legfrissebb verziót – lehet, hogy merge-ölni kell a helyi változtatásokkal
    - Check-out: a fájl helyileg szerkesztés alatt van: megváltozott a legutolsó szerver szinkron óta
    - Check-in: a helyi módosításokat feltöltjük a szerverre – előtte mindig Get Latest van… > Changeset: egy checkinhez tartozó változások (fájlok, műveletek)
    - Shelve pending changes: polcra tesszük a helyi módosításokat, később le lehet őket tölteni
  - git
    - add: változás „jelzése” a helyi repository felé
    - commit: a helyi változások eltárolása a helyi repositoryban
    - clone: távoli repository lemásolása
    - pull (fetch+merge/rebase): távoli változások mergelése a lokális repositoryval
    - push: helyi változások feltöltése a szerverre

Projektmenedzsment

Projekt: minden olyan tevékenység, amely egy szervezet számára olyan egyszeri és komplex feladatot jelent, amelynek teljestési időtartama (kezdés és befejezés) valamint teljesítésének költségei (erőforrások) meghatározottak és egy definiált cél (eredmény) elérésére irányul.

Projektmenedzsment: A projektmenedzsment erőforrások szervezésével, foglalásuk tervezésével és azok irányításával foglalkozó szakterület egy projekt sikerének érdekében.

57. Jellemezd a szervezeti stratégiát és viszonyát a projekthez!

A szervezeti stratégia lényegében a környezet kihívásaira adott válasz.

- Stratégiai célok: fejlődés, változás tudatos alkalmazása, kívánatos jövőbeli állapot, döntések kiválasztása, erőforrások előteremtése

- Lehetőségek, irányok megfoglamazása: külső környezet és változások (makrogazdaság), belső adottságok(erős/gyenge pontok, különböző erőforrások), külső és belső érdekcsoportok elvárásai (tulaj, fogyasztó, stb.)

Stratégiától a projektig:

- Hierarchikus rendszer: jövőkép
  - célkitűzések (kvalitatív)
  - Konkrét célok (vegyes)
  - Stratégiai programok és akciók (kvantitatív)
- Feltételrendszer kialakítása:
  - Realizálás: projektek
  - Projekt segítségével kerül a szervezet az egyik állapotból a másikba
  - Projekt menedzsment: a stratégia egy-egy jól körülhatárolható, komplex, egyszeri feladata

Szervezeti forma: Egyértelmű, hogy egy adott projekthez megfelelő számú (elég idővel rendelkező) és megfelelő képzettségű embernek kell együtt dolgoznia, a forma megmondja hogyan (milyen jellegűek a szervezetben a projekthez tartozó koordinációs, döntéshozási irányok).

58. Jellemezd a lineáris-funkcionális szervezeti formát!

- A tevékenységeket a funkcionális szervezeti egységek végzik, például: oktatás, kutatás, bérszámfejtés stb.
- A projektvezető a felsőbb vezetésnek felel
- Nincs projektmenedzseri hatáskör, nem allokálhat erőforrást, nem utasít, nem ellenőriz (ezeket csak a felsővezető végezheti), csak koordinátori szerepe van, információs központ
- A felelősség és hatáskör nincs összhangban
- Kicsi koordinatív kapacitás, standard tevékenységfolyamatok vannak

Előnyök:

- Azonos szakmai kompetenciával rendelkezők együtt dolgoznak, munkaidejük hatékonyan kihasználható
- Egymás segítségét közvetlenül igénybe vehetik
- Tapasztalatok megőrzése, átadása, újrahasznosítása

Hátrányok:

- A projektvezető hatásköre kisebb, mint a felelőssége
- A funkcionális szakmai szempontok dominálnak a projektszempontok előtt
- A projektfeladatokkal lassabban haladhatnak az egyes funkcionális területek, mert közben folyamatosan operatív feladatokat látnak el
- A több területet érintő döntések nehézkesek
- Az információ áramlás nehézkes, mert különálló funkcionális területek között kell kommunikálni

59. Jellemezd a projektorientált szervezeti formát!

- Elkülönült szervezeti egység végzi a projekt teljesítését

- A szervezeti egység összegyűjti a projekt teljesítéséhez szükséges embereket a különböző funkcionális területekről a projekt időtartamára (a projekt futása alatt ez dinamikusan változhat)
- A projektvezető a felsőbb vezetésnek felel
- Döntési, utasítási, ellenőrzési joga van
- Felelősség és hatáskör azonos szinten
- Jó koordináció, rossz skálázhatóság

Előnyök:
- Nincsenek prioritási problémák
- Az szükséges erőforrásokat a projektre lehet koncentrálni
- Hatékony információ áramlás

Hátrányok:
- A projekt érdek háttérbe szoríthat más, a funkcionális területek által képviselt, az egész vállalatot érintő érdekeket
- Ha a tagok nem teljes munkaidejüket fordítják a projektre, akkor a hatékonyságuk csökken
- A projekt szervezet ideiglenes, így a vele való azonosulás nehézkes
- A felhalmozott tapasztalatokat nehéz megőrizni és újra felhasználni egy másik projektben

60. Jellemezd a mátrix szervezeti formát!
- Megosztott hatáskörök
  - Funkcionális szervezeti egységek teljesítenek (hogyan és ki?)
  - A projektmenedzsmentnek is van hatásköre (mit és mikor/ra?)
- Kompromisszumok a funkcionális terület vezetői és a projekt vezetők között: (kiegyensúlyozott mátrix vs. gyenge/funkcionális mátrix vs. erős/projekt mátrix)
- Sérülékeny de van kohézió
- Jó koordinációs kapacitás

Előnyök:
- Nincsenek prioritásbeli konfliktusok, azt a vezetők tisztázzák
- Azonos szakmai kompetenciák koncentrációja nagyobb hatékonyságot eredményez
- Stabil struktúra biztosítja a tapasztalatok újrafelhasználását
- Hatékonyabb információáramlás mint a funkcionális megközelítésnél

Hátrányok:
- A kompromisszumok törékenységet visznek be a megoldásba, ha gyakori felsőbb szintű döntésre van szükség az lassítja a projekt folyamatot
- Ha egy kolléga sok projektben vesz részt akkor a gyakori fókusz váltás rontja a hatékonyságot
- Ilyen esetben a elkötelezettség (projekt), csapatépítés (funkcionális terület) is nehézségekbe ütközik

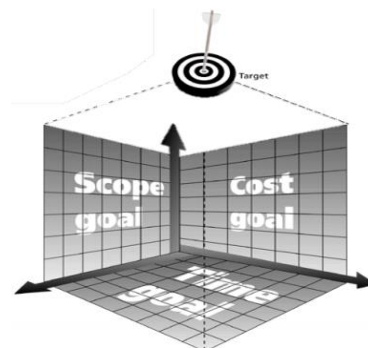
Választani a szervezeti formák között:
- Lineáris funkcionális
  - Kis projekt, kevés kommunikáció a különböző területek között
- Projekt
  - Komplex együttműködés, nagy bizonytalanság

- Nagy projekt esetén a projekten belül jelenhet meg a lineáris funkcionális bontás
- Mátrix
  - Nagyon nagy, nagyon komplex projektekhez

61. Írj a projekt tipológiáról!

- Beruházási projektek: létesítmény
  - Új létesítmény jön létre vagy kerül módosításra
  - Műszaki paraméterekkel egyértelműen jellemezhető
  - Fizikai teljesítés, materiális jellegű
  - Kevéssé prototipizálható
- Kutatási és fejlesztési projektek: termék, technológia
  - Új termék, technológia vagy javítás (technológia: minden olyan eljárásmód, amely meghatároz egy tevékenységfolyamatot)
  - Többnyire rögzíthető kvantitatív módon
  - Szellemi erőforrások a meghatározóak
  - Prototípusok készülnek a termékhez
- Szellemi szolgáltatási projektek: működési körülmények, keretfeltételek
  - pl: privatizáció, ISO minősítés megszerzése, …
  - Nem jól kvantifikálható de modellezhető
- Külső, belső projektek: ki a megvalósító / közreműködő

62. Mi a projekt karakterisztika, kik egy projekt résztvevői?



Karakterisztika:

- cél (eredmény, követelmények, funkciók) - MIT
- időtartam (kezdet és vég határidők) - MIKORRA
- költségkeret (erőforrások) - MENNYIÉRT
- Megfelel a minőségi követelményeknek
- Elégedett a projekt szponzor is

Résztvevők (stakeholders):

- akik a projektbe bevonásra kerültek vagy érintettek a projekt aktivitásai által
  - Projekt szponzor
  - Projekt menedzser
  - Projekt csapat
  - Támogató személyzet
  - Ügyfelek
  - Felhasználók
  - Beszállítók
  - Ellenfelek

63. Sorold fel a PMBOK folyamatcsoportjait és tudásterületeit!

- PMBOK: The Project Management Body of Knowledge

- Széles körben elfogadott projekt menedzsment tudás, best practices gyűjtemény folyamatok leírásával, részletes ajánlásokkal, sok projekttípussal (általános, nem konkrét, testre kell szabni)
- Nemzetközi standard, 4 évente gyakorló szakemberek frissítik

Folyamat:

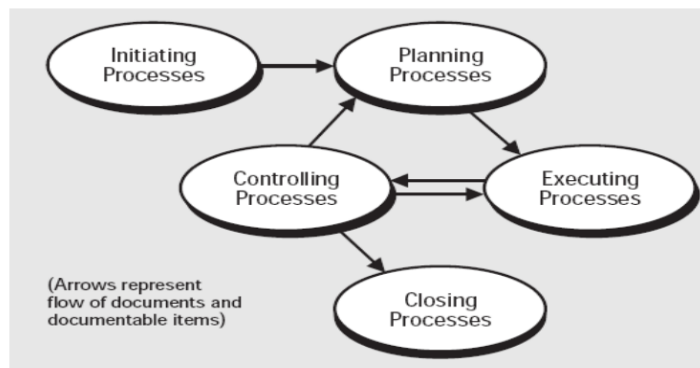
- valamilyen eredményre vezető tevékenységek sorozata
    - Bemenetek: dokumentumok vagy dokumentálható elemek, amikkel dolgozni kell
    - Eszközök és techniák: azok a mechanizmusok, melyek segítségével a bemeneteket kimenetekké alakítjuk
    - Kiementek: a folyamat eredményeit jelentő dokumentumok vagy dokumentálható elemek

Folyamatcsoportok:

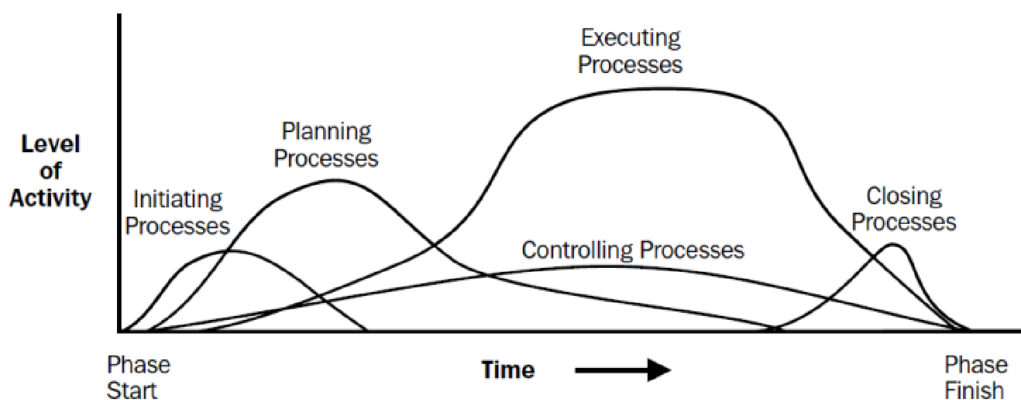
- A különböző folyamatokat egy csoporton belül a bemenetük és kimenetük kapcsolja össze

1. Kezdeményezés
2. Tervezés
3. Végrehajtás
4. Követés/felügyelet
5. Zárás

Összefüggések közöttük:



Átfedéseik:

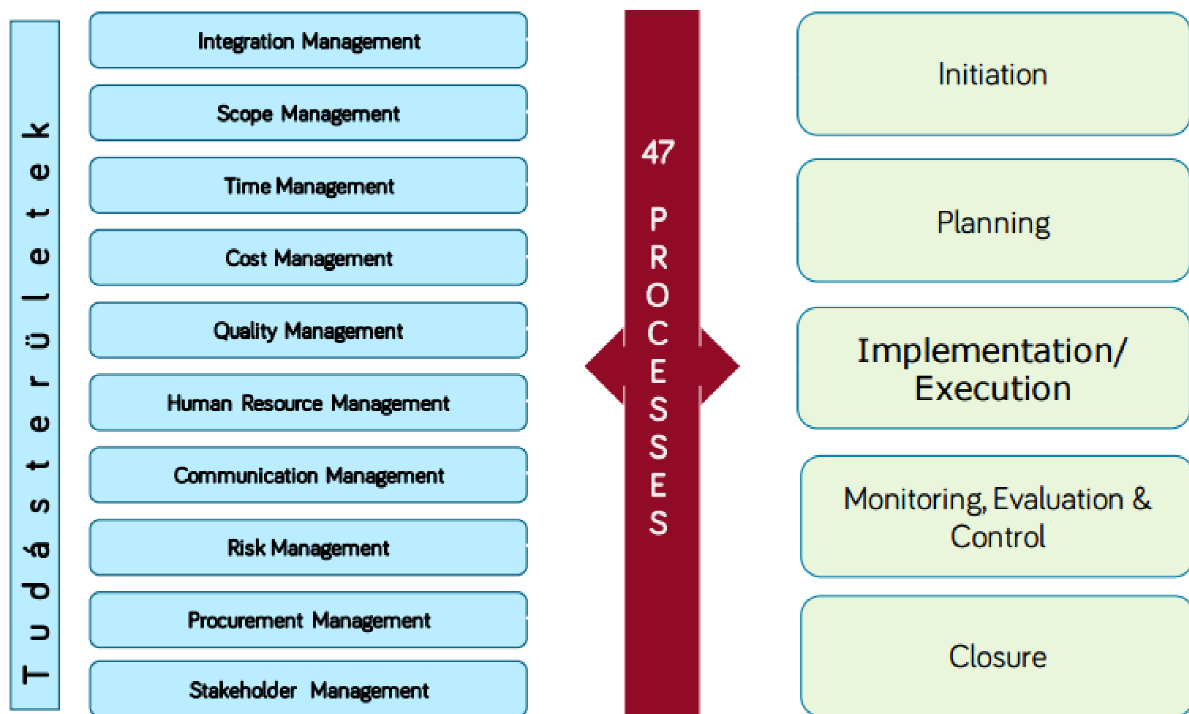


Tudásterületek:

- Az a tudás, amivel egy projekt vezetőnek rendelkeznie kell, hogy konzisztens módon sikeres projektjei legyenek, amelyek megfelelnek a megrendelő elvárásainak, terjedelem, idő, költség és minőség tekintetében

1. Integráció menedzsment
2. Terjedelem kezelés
3. Ütemezés
4. Költség kezelés
5. Minőség menedzsment
6. Emberi erőforrások kezelése
7. Kommunikáció
8. Kockázat kezelés
9. Beszerzés menedzsment
10. Projekt érintettek kezelése

Együtt a kettő:



Ami kimaradt a diasorból: 67-től a végéig gyakorlatilag
PMBOK vs Scrum/Agile: 67-87
Folyamatcsoportok részletezése

63. Mik a SOLID elvek? Röviden ismertesd mind az ötöt! Válassz ki egyet és jellemezd részletesen!

- **Single Responsibility** – mentális modell egyszerűsége és érthetősége a cél, komponensek és dekomponálás, interfészek helyes használata
- **Open closed**: nyitottság a kiterjeszthetőségre, de zártság a módosításra, absztrakció
- **Liskov Substitution**: a származott osztály nem változtathatja meg az ősben definiált viselkedést, szerződés elő és utófeltételekkel a metódusokra
- **Interface segregation**: kis interfészeket használjunk ne kényszerítsük felesleges metódusok implementálását
- **Dependency Inversion**: Egy magasabb szintű modul ne függjön egy alacsonyabb szintűtől, mindkettő függjön absztrakcióktól, használjunk absztakciókat és intefészeket. Az absztakiók ne függjenek a részletektől, de fordítva lehet.

64. A szoftverfejlesztési folyamat tevékenységei. Foglald össze és röviden mutasd be a szoftverfejlesztési aktivitásokat!

- **Követelményelemzés**: közös kép kialakítása a megrendelő és szállító között a termékről
- **Specifikáció** készítés: a projekt célja, feleősök, határidők, átvételi tesztek, felhasználók, ..
- Tervezés: architektúra, interfészek, részletes adatbázisterv, használati tervek, prototipizálás (iteráció a felhasználói visszajelzések alapján)
- **Implementálás**: munka dekomponálása, fejlesztési tervek
- **Tesztelés**: spec szerint működik? Visszatérünk korábbi pontokba
- **Debuggolás**: bugok javítása
- **Telepítés**/üzembe helyezés: jóváhagyott verzió telepítése, terjesztése
- **Karbantartás**: hibajavítás, tesztelés a termék későbbi életére (igen csak költséges)

65. Mutasd be a szoftverrendszerek, alkalmazások értékeit!

Karbantarthatóság, a folyamatos tovább fejleszthetőség és szállítás képessége.
Új szoftverek legfőbb jellemzője a folyamatosan változó követelmények.
Értékek a fejlesztésben: csapatmunka kódminőség, szoftver életciklus, fenntartható fejlesztése folyamat.

66. Mutasd be a szoftverprojektek életciklusának elemeit, azok jellemzőit!

Megvalósíthatósági tanulmány, analizálás, tervezés, implementáció, tesztelés, felhasználás, karbantartás

----

- **Inception** (Kezdet) – A projektötlet kialakítása (a csapat eldönti, hogy érdemes-e a projektet folytatni/érdemben elindítani, valamint, hogy milyen erőforrás igénye lesz)
- **Elaboration** (Kidolgozás) – A projekt architektúra és a szükséges erőforrásigény kidolgozása (a fejlesztők átgondolják a lehetséges alkalmazási területeket, valamint a fejlesztés költségeit)

- **Construction** (Megépítés) – A project kidolgozásra kerül (a szoftver tervezése, fejlesztése és tesztelése)
- **Transition** (Átadás) – A szoftver publikus kiadása, átadása (végső simítások, visszajelzések alapján történő korrekciók)