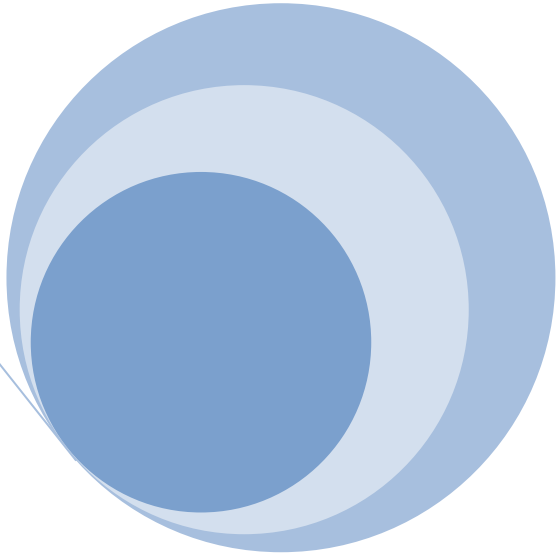
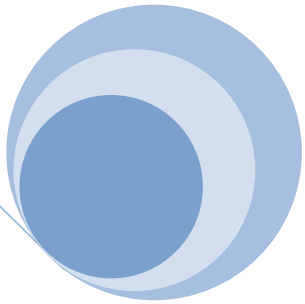




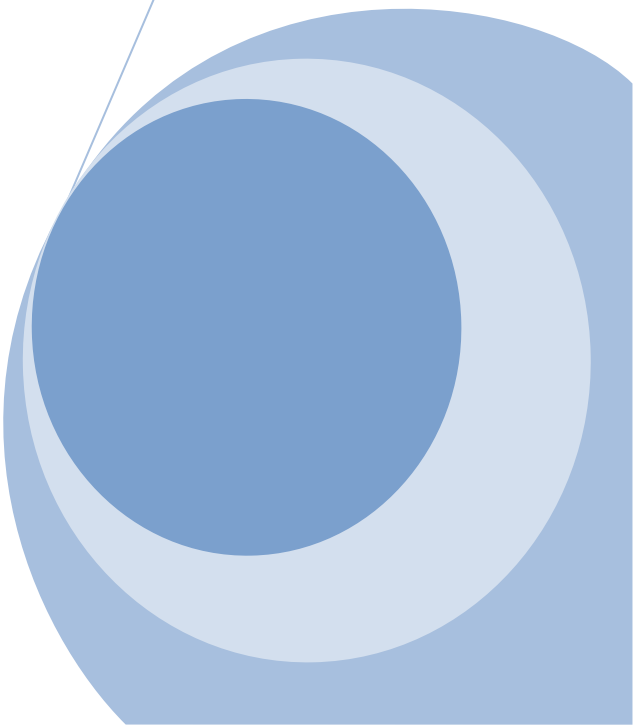
A Rendszerfejlesztés Technológiája

Dr. Juhász Pici előadása

Vízió, Követelmények feltárása, Elemzés, Architekturális tervezés, Tervezés, Implementálás, Tesztelés, Üzembe helyezés, Üzemeltetés, Karbantartás, Evolúció, Üzemen kívül helyezés

Fésüs Miklós, Pálóczi Árpád

2009/2010



A rendszerfejlesztés technológiája

1. előadás 2008. szeptember 9.

Ajánlott irodalom: Ian Sommerville – Szoftverrendszerek fejlesztése

- Vízió
- Követelmények feltárása
- Elemzés
- Architekturális tervezés
- Tervezés
- Implementálás
- Tesztelés
- Üzembe helyezés
- Üzemeltetés
- Karbantartás
- Evolúció
- Üzemen kívül helyezés

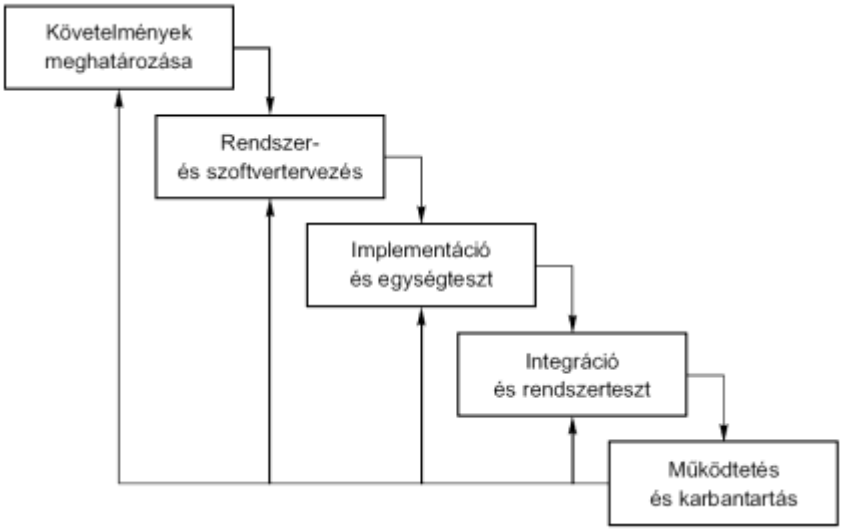
Vízió

A nagyméretű szoftverek esetében több probléma is felmerül. A nagyméretű szoftvereket csapatokban fejlesztik, s ezek időnként változhatnak. Az eredetileg megadott követelmények is megváltozhatnak.
A szoftverfejlesztés egy folyamat, ezen belül fázisok vannak, a folyamatot magát menedzselni kell.
Heterogenitás: A szoftverfejlesztés heterogén, mivel a fejlesztés egyszerre több környezetben történik, más infrastruktúrában, más platformokon, vannak elosztott rendszerek, illetve más-más technológiában is készülhetnek(Java,.Net).
Időkényszer: Ezek mellett ott az időkényszer, vagy a szoftvergyár, vagy a megrendelő oldaláról.

Bizalom és bizalmasság kérdése:

- El kell hinnünk, hogy a szoftver úgy működik, mint azt a készítő állítja.
- Fel kell tételezni, hogy a gyártó nem él vissza az általa elérhető információkkal.
- Mindez kemény szakmai, és etikai felelősség az informatikusoknak.

Folyamatmodellek: absztrakt modellek, közelítik a folyamatot, a fázisokkal kapcsolatban is mondanak valamit.
Vízesésmodell: 70-es években alakul ki, egy szekvenciális modell.
Fázisokat szigorúan egymás után kell végrehajtani, és minden modell egy dokumentummal zárul, ami dönt arról, hogy érdemes e folytatni.

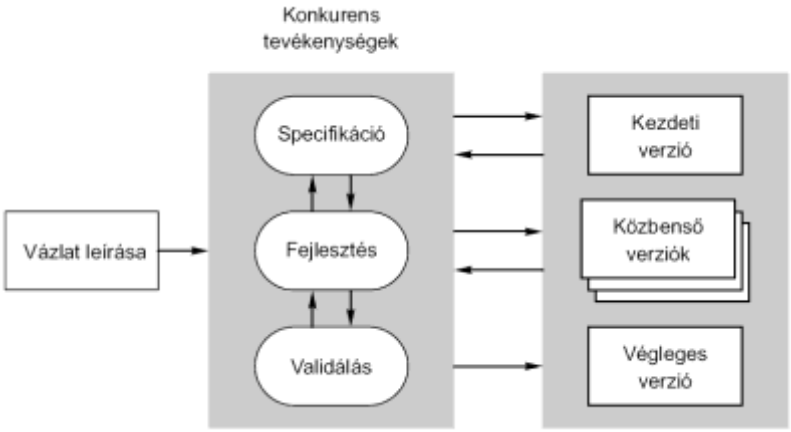


Használható: ha a követelmények teljesen világosak, ez csak kisméretű szoftvereknél fordul elő.
Hátránya:Viszonylag sikeres volt, elég sokáig, amíg meg nem bukott, mivel egy követelmény változás esetén lehet elölről kezdeni az egészet ami pont olyan mint ha felfelé mennénk egy vízesésen☺.

Evolúciós modell: a szekvencialitás tagadásaként jön létre, párhuzamosságot helyezi a középpontba. Prototípusokat állít elő, kétféle van:

- feltáró prototípuskészítés
- eldobható prototípuskészítés

Feltáró prototípuskészítés:



Megértettük a követelmények egy részét, ezt specifikáljuk, implementáljuk, validáljuk, és előáll a szoftver, majd elölről kezdjük az egészet, és így tovább, míg az összes követelménnyel meg nem csináltuk.
Verziósorozat lesz belőle, mely egyre több funkciót fed le.
Eldobható: Előáll a kezdeti követelmények alapján egy prototípus, és a rendelővel ez alapján pontosítunk, majd ezt eldobjuk, és újat csinálunk egészen addig míg jó nem lesz.

Előnyei:

Hamar működő szoftver áll a rendelkezésre
Először a lényeges dolgokat implementáljuk, vagy a megértett követelményeket.
Van mindig használható eredmény.

Hátrányai:

Nem lehet tudni, hogy mikor van meg az összes követelmény, nincs teljes követelményrendszer.
A teljes folyamat nem látszik, nehezen tervezhető, illetve nehezen időzíthető és költségelhető.
A prototípusok általában szegényes architektúrájúak.

Újrafelhasználási/Komponens modell:

Komponensek újrafelhasználására épül.

Legegyszerűbb:

Adott esetben, ha nincs újrafelhasználható komponens, akkor megpróbálom a követelményeket módosítani úgy, hogy jó legyen.

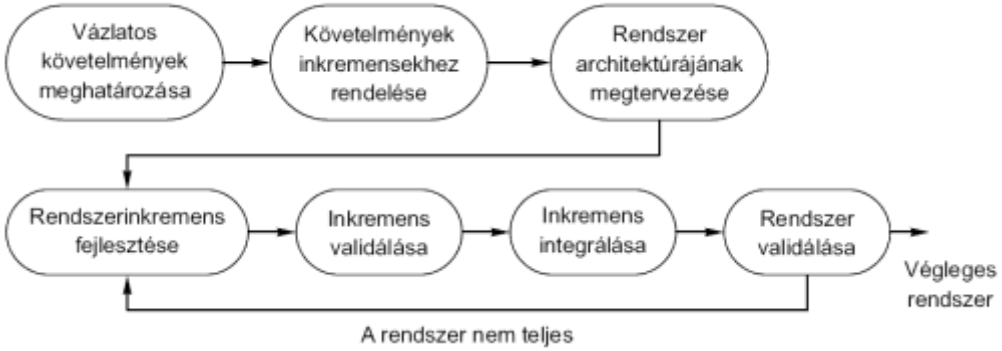
Újrafelhasználásorientáltság: Eleve komponensekre bontom, és ezeket újrafelhasználhatóan készítem el.

Előny: Gyors fejlesztés, piacra kerülés, ha a komponenseket rendesen készítem el, akkor bizalmat is szül, mert ha jól működik, akkor legközelebb már nem kell validálni.

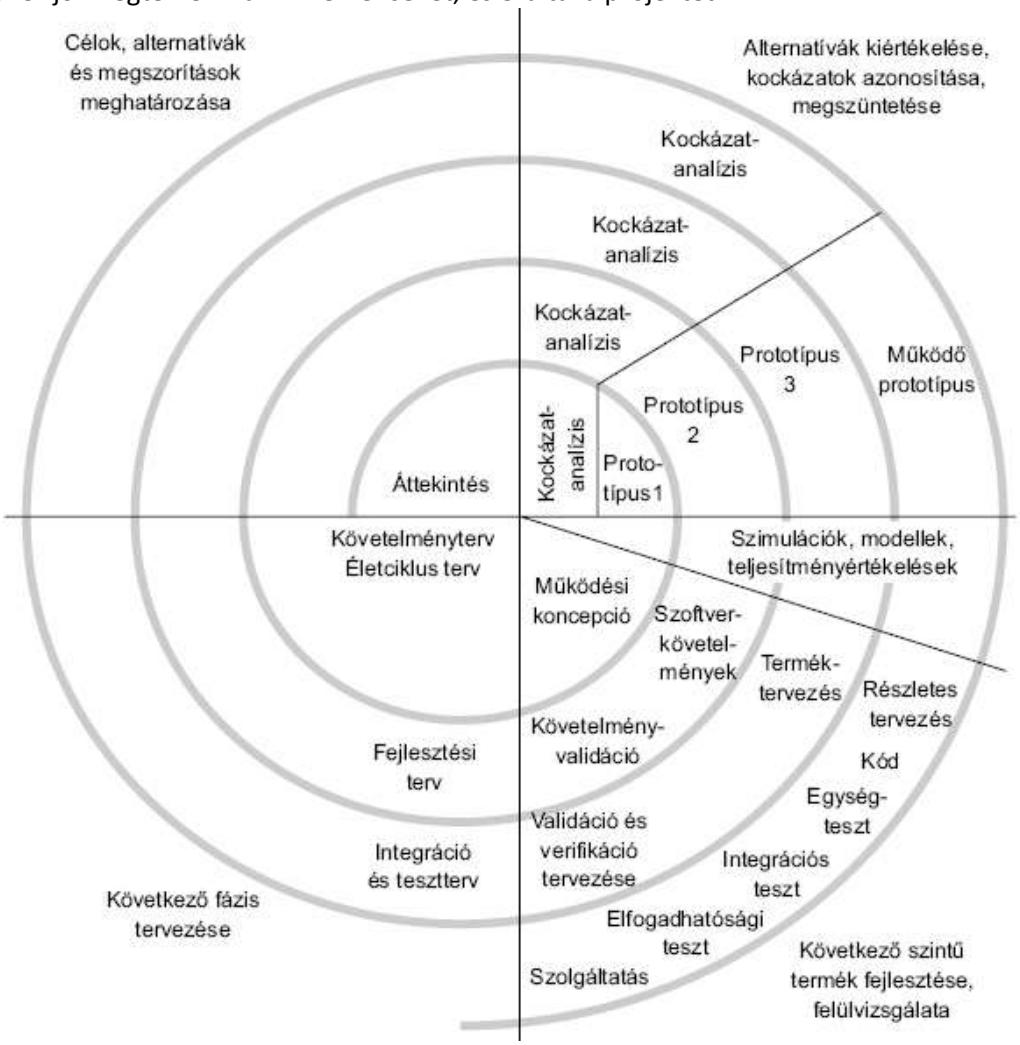
Hátrány: egy az egybe komponens=követelmény megfeleltetés nem valószínű hogy van, a komponenseket meg kell találni, és publikálni.

Iterációs modellek.

Inkrementális modell: az evolúciós modell továbbfejlesztése. Indul egy vázlatos követelményből, és a feltáró prototípuskészítés alapján létrehoz egy minimális rendszert, és ezt mindig egy inkremenssel bővíti. Az inkrementsek kicsik és behatárolhatóak.



Előnye: az evolúciós modellhez képest itt van rendszer architektúra előre, és nem pedig közben változik.
Hátránya: Nehéz jól megtervezni az inkrementseket, és ezáltal a projektet.



Irányzat: tart az agilis módszertan felé: nincs követelmény, az inkrementsek extra kicsik, a felhasználóval együtt készülnek, és a rendszer folyamatosan változik.

Követelmények

Követelmények dokumentációja

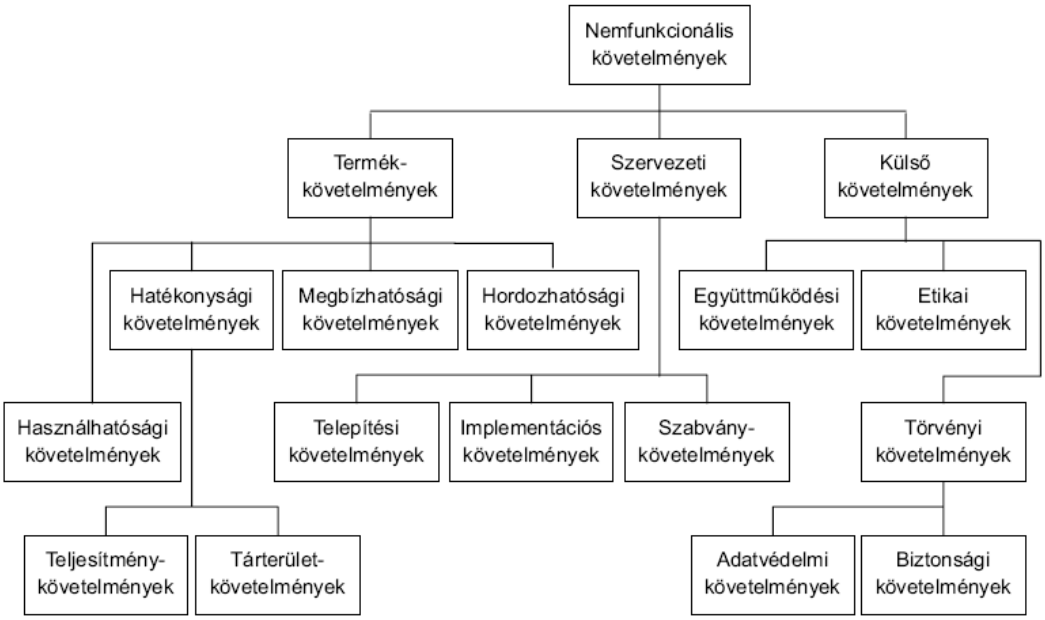
Szolgáltatások és megszorítások együttese. Vannak felhasználói és rendszerkövetelmények.
A felhasználói követelmények igazából absztrakciók, azt adja meg, hogy a felhasználó mit vár el a rendszertől, tulajdonképpen egy formalizált elvárásegyüttes.
Rendszerkövetelmények: részletes dokumentumok, konkrétabb, a felhasználói követelményeket bontja le, s ebből indul ki majd a tervezés.

Három féle követelmény van:

1. funkcionális
2. nem funkcionális
3. szakterületi

Funkcionális: egyértelműen rendszer szolgáltatásaihoz kapcsolódnak, hogy milyen bemenetre milyen kimenet stb… Elviekben teljesnek és ellentmondásmentesnek kéne lennie. Meg lehet adni, hogy milyen eszközt, vagy szabványt használjon, kevés elemhez kapcsolódnak.
Nem funkcionális: Könnyebben megfoghatóak, eredendő rendszertulajdonságok, nem kapcsolódnak a rendszer részeihez általában. Ezek bizonyos szituációkban sokkal lényegesebbek. Három részre bontható:

1. ***Termék követelmények:*** Rendszerre vonatkoznak (használhatóság, hatékonyság, megbízhatóság, hordozhatóság, teljesítmény, tárterület)
2. ***Szervezeti követelmények:*** Nem a rendszerre van, hanem a környezet elvárásai, akik használják (telepítési, implementációs, szabvány)
3. ***Külső követelmények:*** jogi, politikai, gazdasági, etikai, törvényi, adatvédelmi, biztonsági



A követelményeket verifikálni illetve validálni kell. a funkcionálisakat viszonylag könnyű, a nem funkcionálisat már nehezebb. Ha metrikával van megfogalmazva akkor mérhető és könnyű, csak az a gond, hogy a legtöbb dologhoz nem tudunk metrikát társítani.

Rendszercél
A rendszernek a tapasztalt ellenőrök számára könnyen használhatónak kell lennie, és javallott olyan módon szervezni azt, hogy a felhasználói hibák száma a lehető legkisebb legyen.

Verifikálható nemfunkcionális követelmény
A tapasztalt ellenőrök képesek legyenek az összes rendszerfunkció használatára egy kétórás képzés után. A képzés után a tapasztalt felhasználók által elkövetett hibák száma átlagosan ne haladja meg a napi kettőt.

Tulajdonság	Mérés
Sebesség	Feldolgozott tranzakciók/másodperc Felhasználó/esemény válaszidő Képernyő-frissítési idő
Méret	Kbyte RAM-chipek száma
Könnyű használhatóság	Képzési idő Help képernyők száma
Megbízhatóság	Hibák átlagos bekövetkezési ideje A rendelkezésre nem állás valószínűsége Hibák bekövetkeztének aránya Rendelkezésre állás
Robusztusság	A hiba utáni újraindulási idő A hibát okozó események százalékos aránya A hiba okozta adatmeghibásodás valószínűsége
Hordozhatóság	A célrendszerfüggetlen utasítások százalékos aránya A célrendszerek száma

A Nem funkcionális követelmények gyakran ellentmondanak maguknak, vagy a funkcionálisakkal is, pl.: sebesség⇔méret, könnyű használat⇔robosztusság. Ezért néha nem is akarjuk megkülönböztetni a funkcionális és nem funkcionális követelményeket.
A funkcionális követelményeknél gyakran fordul elő olyan követelmény, hogy a rendszer mit nem csináljon.

Felhasználói követelmény: úgy kell megfogalmazni, hogy a rendszer külső viselkedését határozzák meg. Beszélt nyelven van megfogalmazva, ezt egészítjük ki táblázattal, űrlappal illetve diagrammal. Ez nem feltétlen egyértelmű, a funkcionális és nem funkcionális követelmények keverednek illetve szuperponálódnak.
Megoldás: félreértések csökkentése végett formalizált, szigorúbb nyelve, illetve hogy derüljön ki hogy mit kell tudnia a rendszernek, és mit szeretnénk ha tudna.

Rendszerkövetelmény: strukturáltabb természetes nyelvek, vagy formalizált nyelven, erős grafikus jelölés, diagramok. Néhány esetben matematikai jelölés van, főleg realtime illetve kritikus rendszereknél.
hátránya: hogy meg kell érteni, előnye hogy nem lehet félreérteni, és könnyű belőle rendszert fejleszteni.

Funkcionális követelmény leírás:

1. az éppen megadott funkció vagy egyed leírását
2. bemeneteit és azt, hogy azok honnan származnak
3. kimeneteit és azt, hogy azok hova kerülnek
4. utalást arra, milyen egyéb egyedek használtak (az *igény* részben)
5. a végrehajtandó művelet leírását
6. funkcionális megközelítés használata esetén egy előfeltételt, amely kijelöli, minek kell teljesülnie a függvény hívása előtt, és egy utófeltételt, amely azt adja meg, minek kell teljesülnie a függvény hívása után
7. a művelet mellékhatásainak leírását (ha vannak)

Manapság a követelménydokumentáció részei az interfész specifikációk is, azaz, hogy a rendszer a környezetével hogyan kommunikál. Azt is meg lehet adni, hogy a rendszer egyes alrendszerei hogy kommunikálnak.
Megadható szervezeti követelmény még:

- procedurális interfész (API)
- kommunikációs adatszerkezet (ezt matematikai jelöléssel szokták megadni), ez generálható az API-ba.
- adatreprezentációs kérdés, itt szabványt szokás megadni.

A Végén előáll egy követelményspecifikáció. Vannak követelményszabványok is, pl.: IEEEE/ANSI 830
Ez egy metaszabvány: javaslat, hogy hogyan nézzen ki, és tippeket, trükköket ad.

IEEEE/ANSI 830

1. Bevezetés
   1.1. A követelménydokumentum célja
   1.2. Termék felhasználási terület
   1.3. Definíciók, betűszavak, és egyéb rövidítések.
   1.4. Hivatkozások
   1.5. A dokumentum hátralévő részének áttekintése.
2. Általános leírás.
   2.1. A termék áttekintése.
   2.2. A termék funkciói.
   2.3. Használati jellemzők.
   2.4. Általános megszorítások.
   2.5. Feltételezések és függőségek.
3. Speciális követelmények (tulajdonképpen itt van leírva minden)
4. Függelék
5. Tárgymutató

Használatos:

1. Előszó: (*tulajdonképpen verziókezelés)*
2. Bevezetés: *(szól a dokumentumról, és mindenről ami a user szempontjából lényeges)*
3. Szójegyzék: *(jel- és defincíció-magyarázat)*
4. Felhasználói követelmények definíciói: *(Funkcionális és szakterületi absztrakt követelmények)*
5. A rendszer felépítése: *(Architektúrára vonatkozó, tervezést elősegítő információk)*
6. Rendszer követelmény specifikáció: *(Funkcionális, Nem funkcionális és szakterületi, ez a részletes)*
7. Rendszer modellek
8. Rendszer evolúció: *(Hogyan készülünk fel majd a változásra)*
9. Tárgymutató
10. Függelék: *(ábrák és egyebek)*

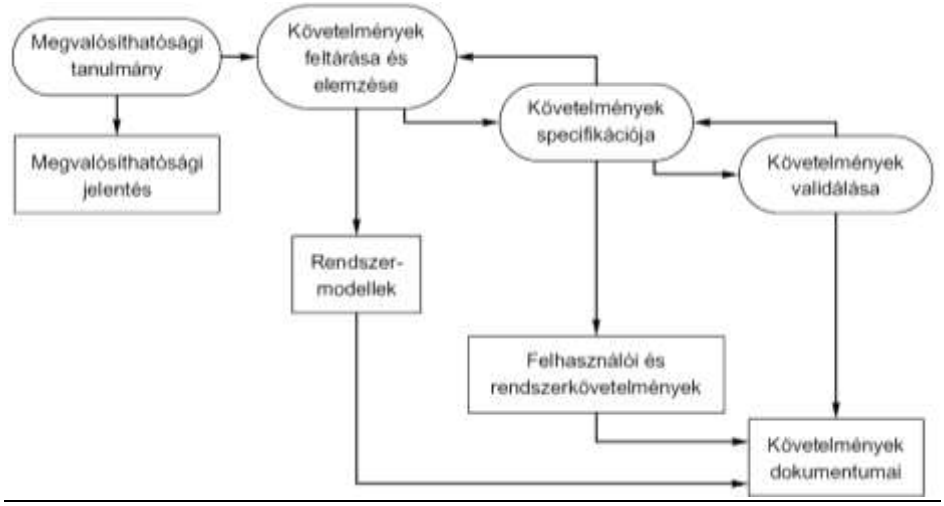
Minden rendszerhez van egy vízió, az, ahogyan a megrendelő elképzeli.
Minden fázis lépésekre oszlik, minden fázis végén van egy végtermék. Minden fázis minden lépésében technikát, esetenként modellt használunk. Mögöttük technológiák vagy szabványok is lehetnek.

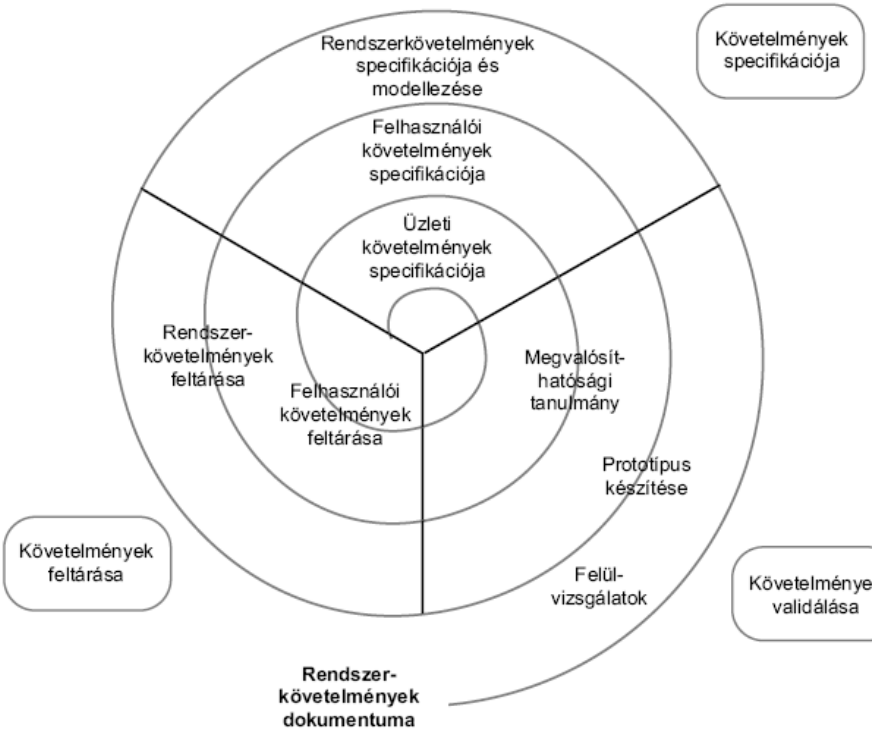
Követelménytervezés: 3. előadás 2008-09-23

Meghatározzuk és dokumentáljuk a követelményeket. Ez a fázis 4 lépésre osztható:

1. Megvalósíthatósági tanulmány (felmérés) előállítása.
2. Követelmények feltárása és elemzése
3. Specifikáció
4. Validálás

A rendszer víziója alapján egy 2-3 hét alatt elkészülő olyan dokumentáció, amely alapján az illetékesek eldöntik azt, hogy egyáltalán érdemes e belekezdeni a fejlesztésbe.





Megvalósíthatósági tanulmány: 3 kérdésre kell, hogy választ adjon:

1. Kifejlesztendő rendszer támogatja-e a vállalat üzleti célkitűzéseit?
    a. Válasz lehet az, hogy nincsenek célkitűzései, vagy homályosak, és inkább elvetik, mint támogatják a célkitűzéseket
    b. Vannak, de az üzleti követelmények specifikációja nem jól sikerül
    c. Világosak a célkitűzések, csak beleszól a kis-nagy politika.
2. A megvalósítandó rendszer adott időintervallumban adott költség mellett és a rendelkezésre álló technológiával megvalósítható e? (mikorra, mennyiért, mibe)
3. A kifejlesztendő rendszer integrálható e a vállalat pillanatnyi rendszereihez?

A megvalósíthatósági tanulmányt külső szakembernek illik elkészíteni, ennek elkészítése során konzultál a menedzsmenttel, a felhasználóval, és az informatikusokkal.
Erről jelentés készül, és a dokumentum alapján figyelembe véve a pro és kontra dolgokat, dönteni kell.

Feltárás és elemzés: a fejlesztők végzik, a megvalósíthatósági tanulmány alapján a követelmények összegyűjtése, és elemzése. Kulcsfigurákra kell támaszkodni. A kulcsfigurák mindenekelőtt a végfelhasználók, azaz közvetlen használják a rendszert, emellett a közvetett használók (menedzsment), és valamilyen értelemben kulcsfigurák az ehhez a rendszerhez kapcsolódó rendszerek is, és a megrendelő informatikusai, amennyiben vannak.

Problémák:

- a kulcsfigurák saját vagy vállalati szakzsargont használnak, és az informatikusnak ezt meg kell értenie.
- a kulcsfigurák nagyon gyakran nem tudják hogy mit akarnak, egyrészt üzletileg, másrészt nem tudják, hogy informatikailag mi valósítható meg és mi nem, vagy mi könnyen és mi nehezen.
- különböző kulcsfiguráknak különböző követelményei vannak, és az összeset fel kéne tárni, és összefésülni.
- Nagyon sokszor a követelmények feltárását a politika is befolyásolja
- A követelményeket úgy kell feltárni, hogy a rendszer működésekori követelmény legyen nem az előállítása előtti, mert változik a politikai és gazdasági és egyéb helyzet.

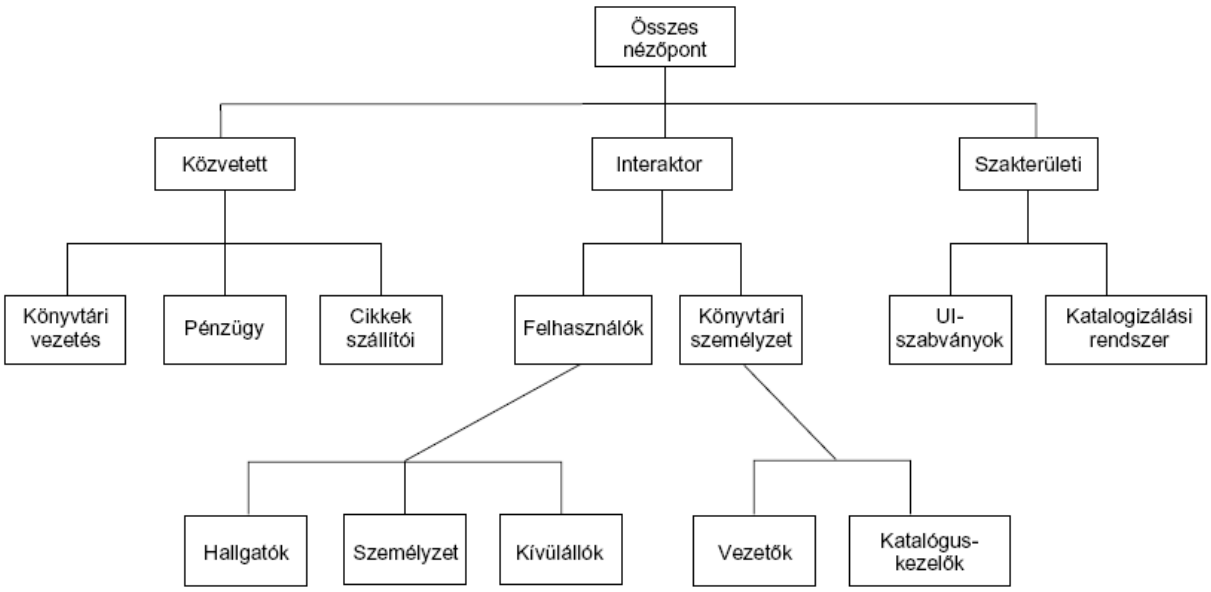
Követelmények osztályozása és szervezése: a követelményeket kategorizálni kell, csoportosítani kell a hasonló vagy azonos követelményeket, lehetőleg homogén csoportokat kell megfogalmazni, amelyek alapján majd a rendszer architektúra majd megtervezhető.
Sorba állítás: rangsorolást jelent, prioritás.
Megtárgyalás: a követelmények ellentmondásosak, és ezeket a kulcsfigurákkal újra kell tárgyalni.

Követelmény felderítési technikák:

Nézőpont-orientált feltárás: A kulcsfigurákat osztályozzuk, a különböző kulcsfigurák más-más nézőpontokat jelentenek, 3 nézőpont csoport van:

- interaktor nézőpont: azok az emberek és rendszerek, amelyek közvetlenül érintkeznek a kifejlesztő rendszerrel. Nekik speciális igényeik.
- Közvetett nézőpont: azok a kulcsfigurák, akik csak áttételesen kapcsolódnak a rendszerhez, a rendszer bizonyos funkcióinak eredményét használják (tipikusan a döntéshozó menedzserek)
- Szakterületi nézőpont: elsősorban szabványokat jelent

Minden nézőpontból kell valakit szerezni.



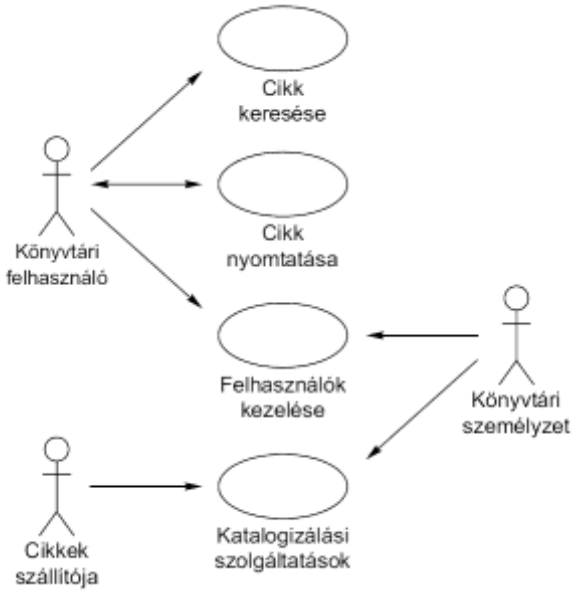
Interjú: az ügyfélhez interjúzni megyünk. Személyesen kikérdezzük. Van zárt és nyílt interjú.
Zárt interjú: egy kulcsfigura egy előre megformázott kérdéssort kap, amire válaszolnia kell. csak azokra kapunk választ, amit feltettünk
Nyílt interjú: nincs ilyen kérdéssor, hanem beszélgetünk a megrendelővel. (eléggé ad hoc☺) Ekkor az ott eszünkbe jutó kérdéseket is fel tudjuk tenni.
Általában a kettő keverékét használják. Az is gond, hogy az adott szakterület kulcsfiguráinak elég sok minden triviális.
Forgatókönyv: Az interjú eredményének dokumentálására a forgatókönyv az eszköz. A forgatókönyvbe csatolok vissza az ügyfélhez, és azzal szembesítem. Arra való, hogy egy interakció sorozatot formálisan leírjunk vele. Ez lehet strukturált szöveg, és/vagy diagram.

Forgatókönyv tartalma:

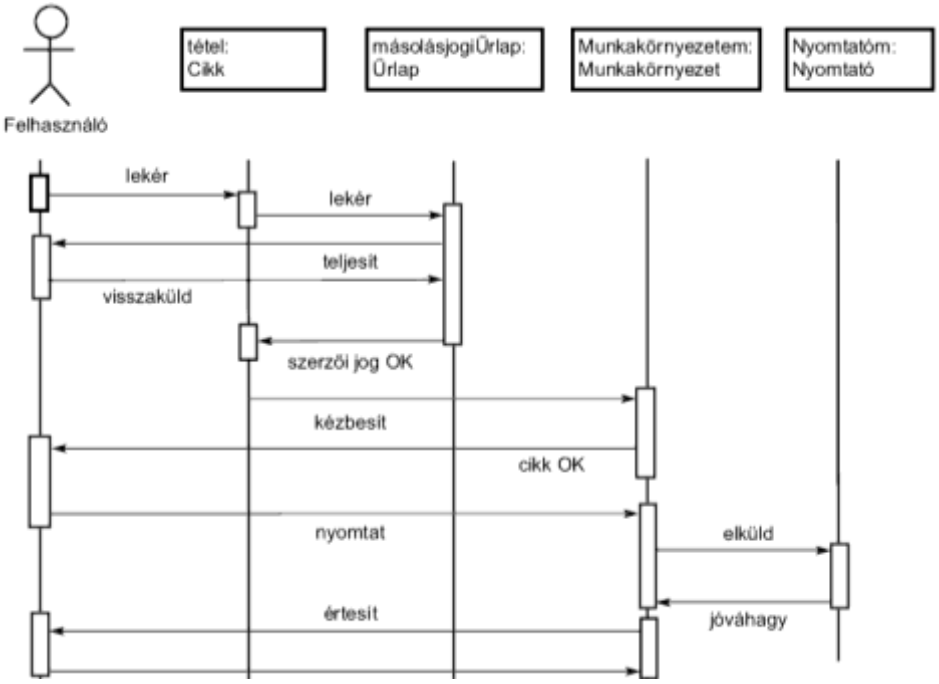
- Annak a megfogalmazása, hogy mit várunk el a rendszertől
- Az interakció sorozat normál menetének a leírása.
- Annak a leírása, hogy a normál menethez képest mi romolhat el és ekkor a rendszer mit tesz. Magyarul a kivételkezelés.
- Az adott interakció sorozattal egy időben zajló párhuzamos tevékenységek leírása.
- A rendszer állapotának leírása az interakció sorozat végén.

Use Case (Használati eset): Actorok vannak és interakciók. A use case a normál menet leírását célozza.

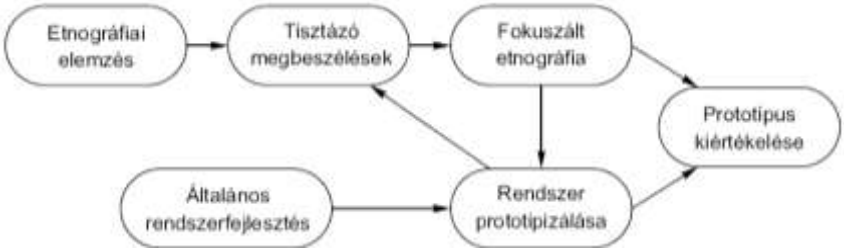
UseCase Diagram



Szekvencia diagram:



Etnográfia(tobechecked-untrusted) –magyarul megfigyelés
Mindenféle dokumentumot begyűjtünk a szóbeli beszélgetés mellett. Megnézzük, hogy hogyan dolgoznak a végfelhasználók. Az interaktor nézőpontokra vonatkozik. Ebből a megfigyelésből szűrünk le következtetéseket, hogy a rendszernek milyen követelményeket kell majd teljesítenie.
Olyan területen hasznos, amit másként nem lehet kideríteni. Általában van egy formális kép arról, hogy egy vállalat hogy működik, közben meg valójában nem feltétlen így van, ezt interjú során nem lehet kideríteni, mert nem beszélnek/tudnak róla. A végfelhasználók is figyelembe veszik, hogy a másik ember hogy dolgozik, és ő ahhoz igazodik. Bizonyos dolgokra viszont abszolút nem alkalmas, például szakterületi és közvetett nézőponti követelmények felmérésére.Az etnográfiát és az eldobható prototípusfejlesztést nagyon gyakran ötvözik.



Követelmények validálása:

A validálás szempontjai:

- Validitás ellenőrzések: a felhasználói követelményekre vonatkoznak, azt nézzük meg, hogy a követelményekben benne vannak e azok a funkcionalitások, amelyekre tényleg szüksége van a felhasználónak.
- Ellentmondás-mentesség: Az összes követelmény vonatkozásában
- Teljesség-ellenőrzés: Nincsen meg sose az összes követelmény, tehát elég nehéz, de próbálni kell közelíteni.
- Megvalósíthatósági ellenőrzések: Az egyes követelményeket nézzük olyan szempontból, hogy időben, pénzben, technológiában.
- Verifikálhatóság ellenőrzése: azt nézzük meg, hogy annak az ellenőrzése majd, hogy egy adott követelmény kielégíti a rendszert, hogyan történhet majd, tudom-e ellenőrizni egyáltalán, hogyan tudom bizonyítani.

Validálási technikák:

1. prototípuskészítés
2. követelmények felülvizsgálata: a specifikáció átnézése. Az a javallott hogy felhasználóval együtt. Viszont nem biztos hogy ez hatékony.
3. Teszteset generálás. Az agilis esetén a teszteseteket azelőtt kell generálni, hogy a tervezést elkezdjük. Amihez nem tudok generálni, azzal valami baj van.

Követelmények menedzselése:

Klasszikus módszertanok is azt vallják, hogy a követelmények változnak, ennek a lekövetését értik a menedzselés alatt. Implementáció változtatását kell végrehajtanunk. Természetes változtatáson (gazdasági, politikai, jogi) túl még két dolog van:

- Követelmény feltárásánál a kulcsfigurák sokfélesége miatt soha nem mondhatjuk, hogy teljes a követelményrendszer
- A rendszerek úgy fejlesztődnek ki, hogy aki a döntést hozza, nem az fogja napi szinten használni a rendszert.

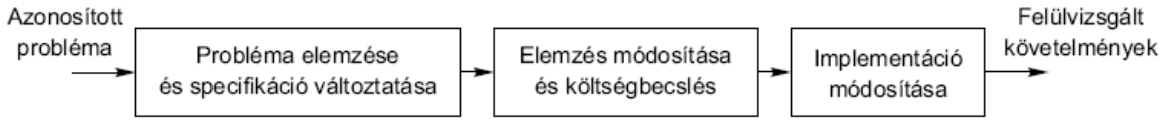
Tartós követelmények: valószínűleg teljesen felderíthetők, nem változnak pl.: hallgatók, tanárok stb…
Átmeneti követelmények: szinte biztos, hogy változni fognak pl.: kreditelőírások stb… , kiderítésük nehéz, ezekre kell igazából koncentrálni.

Követelménytipus	Leírás
Átalakuló követelmények	Azok a követelmények, melyek együtt változnak azzal a környezettel, melyben a szervezet működik. Például kórházi rendszerek esetén a betegellátás pénzalapja változhat, ezért szükség lehet különböző kezelési információk összegyűjtésére.
Felmerülő követelmények	Azok a követelmények, melyek aközben keletkeznek, hogy az ügyfél a rendszer fejlesztése folyamán egyre jobban megérti a rendszert. A tervezési folyamat új felmerülő követelményeket hozhat napvilágra.
Következmény-követelmények	Azok a követelmények, melyek a számítógépes rendszer bevezetéséből következnek. A számítógépes rendszer bevezetése megváltoztathatja a szervezet folyamatait, és a munka új módjait tárhatja fel, melyek újabb rendszerkövetelményeket hoznak létre.
Kompatibilitási követelmények	Azok a követelmények, melyek a szervezeten belüli külön rendszerektől vagy üzleti folyamatoktól függenek. Mivel ezek változnak, a megrendelt vagy átadott rendszerekkel szemben támasztott kompatibilitási követelményeket szintén ki kell alakítani.

Követelmények menedzseléséhez meg kell még csinálni:

- A követelményspecifikációban minden egyes követelményt azonosítani kell.
- Kell egy változtatáskezelési belső szabvány.

Változtatás kezelése:



Ha felmerül egy új probléma: akkor meg kell vizsgálni, hogy probléma vagy nem probléma, és ezután a követelményspecifikációt kell változtatni, majd ezt kell elemezni, hogy mennyibe kerül időben és pénzben. Implementáció után ismét verifikálni és validálni kell.

Nyomon követhetőség: a követelmények egymásra hatása, illetve a későbbi termékekben való megjelenése. Nyomon követhetőségi információk alkategóriái:

1. források nyomon követhetősége: az adott követelmény melyik kulcsfigurához köthető, mely kulcsfigura javasolhat pl.: követelményváltoztatást.
2. Követelmények nyomon követhetősége: ez jelenti a kapcsolatokat.
3. Hatások nyomon követhetősége: a követelmények mely későbbi termékhez kapcsolódhatnak.

A technológia automatizált eszközöket ad a követelménymenedzseléshez.

Nyomon követhetőségi mátrix (F – Függési kapcsolat, K – valamilyen kapcsolat

Követelményazonosító	1.1.	1.2.	1.3.	2.1.	2.2.	2.3.	3.1.	3.2.
1.1.		F	K					
1.2.			F			K		F
1.3.	K			K				
2.1.			K		F			F
2.2.								F
2.3.		K		F				
3.1.								K
3.2.							K	

Rendszermodellek:

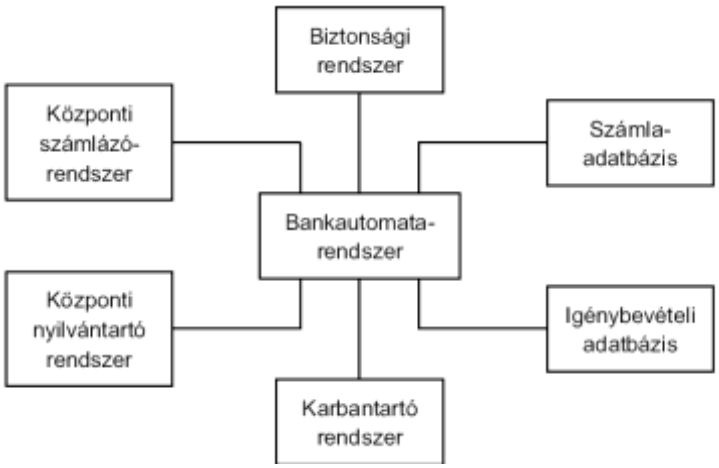
A követelményspecifikáció természetes nyelven íródik általában és ez egészül ki diagramokkal, ábrákkal. Van még egy absztrakciós eszköz, vizuális eszközök, ezek a rendszermodellek. Igazából olyan eszközök, melyek az architektúrális tervezés felé mutatnak. Alkalmasak arra, hogy mind a felhasználó, mind a fejlesztő jól értsenek.

Szemléleti módok:

- Külső szemlélet:, milyennek látszik, milyen környezetbe illesztjük be a rendszert
- Viselkedési szemlélet: A teljes rendszer bizonyos viselkedési dolgait írjuk le.
- Strukturális szemlélet: Rendszerfelépítés, ami egy belső szempont.

Az utóbbi kettő kemény hatással van a rendszerarchitektúrára.

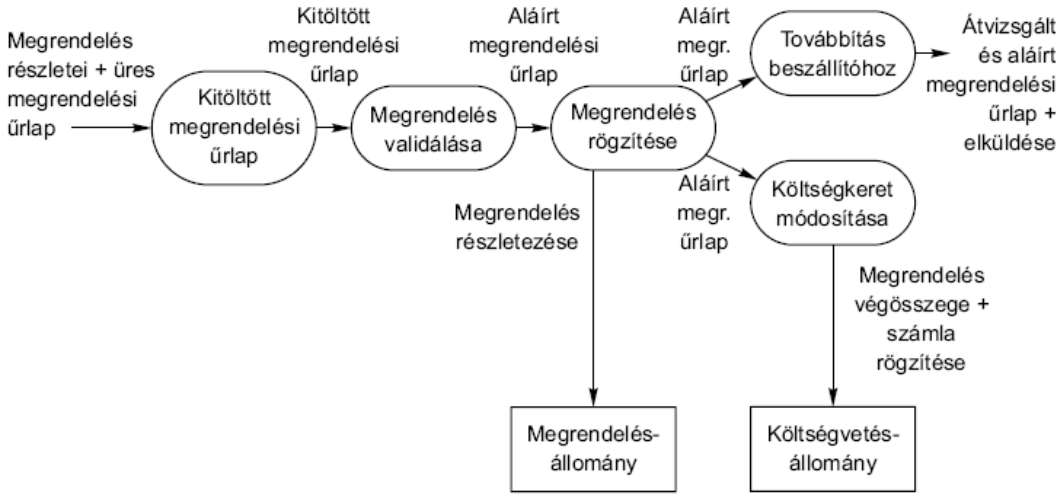
Környezeti modell: A rendszer határait definiáljuk vele, lehatároljuk. Egy új rendszert be kell illeszteni egy működő környezetbe, ezt is célozza. A későbbiekre vonatkozóan azért lényeges, mert a rendszeren belüli alrendszereket is lehatárolja. Rendszereket, alrendszereket téglalapokkal ábrázoljuk, és vonalakkal vannak összekötve.



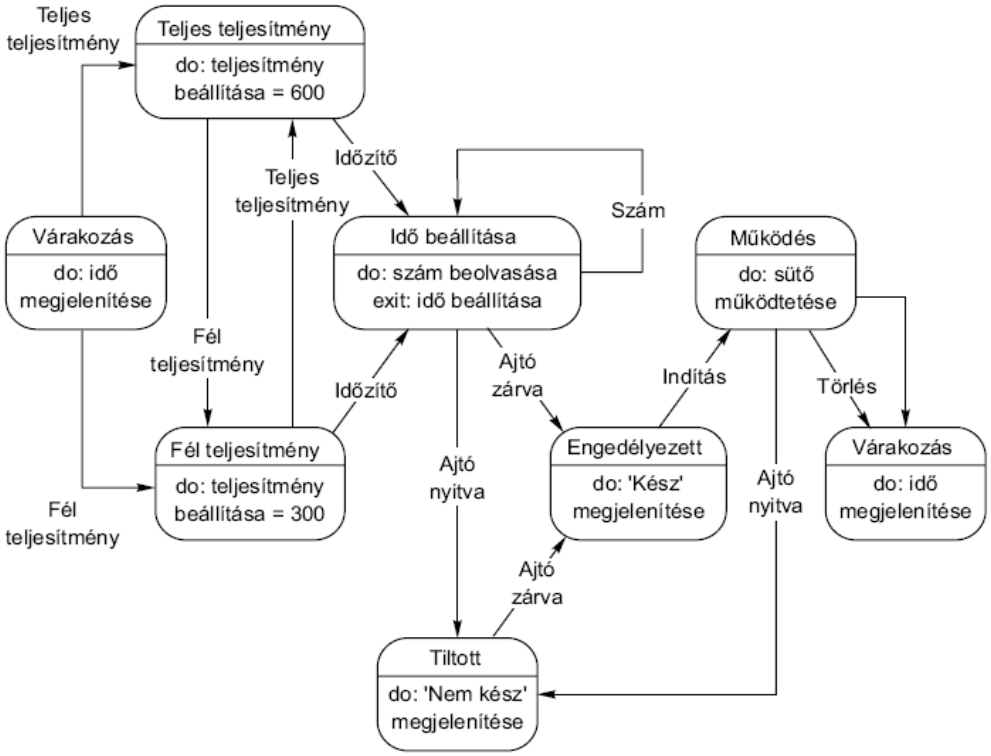
Viselkedési modell: sok lehet belőle.

- Adatfolyam modell, strukturált módszertanok középpontjában áll.
- OO módszertan alapmodellje az állapot-átmenet modell. Az UML-ben is van Állapot-átmenet diagram.

Adatfolyam modell: A rendszerek nagy része adatokat dolgoz fel. Az adatfolyam modellek azt a viselkedést modellezik, ahogy az adatok áthaladnak a rendszer egy részén, és közben milyen transzformáció-sorozaton mennek végig. Funkció-orientált modell. Olyan absztrakciós eszköz, amit viszonylag jól megértenek a felhasználók (ez a legnépszerűbb ☺). Egy transzformációs lépést egy lekerekített téglalap szemléltet. Vannak benne úgynevezett tárolók, ezeket téglalap jelzi. Ezeket nyilak kötik össze, és a nyilak mentén áramlanak az adatok. Párhuzamos és szekvenciális folyamatokat is lehet modellezni.



Állapot-átmenet modell: tulajdonképpen egy véges-automata (elvileg determinisztikus (míg le nem implementáljuk)) modell. A rendszer állapotait modellezzük lekerekített téglalapokkal, megnevezzük az állapotot. Az állapotokban valamilyen inger éri a rendszert, és annak a hatására egy másik állapotba megy át, ezt a viselkedést modellezi.

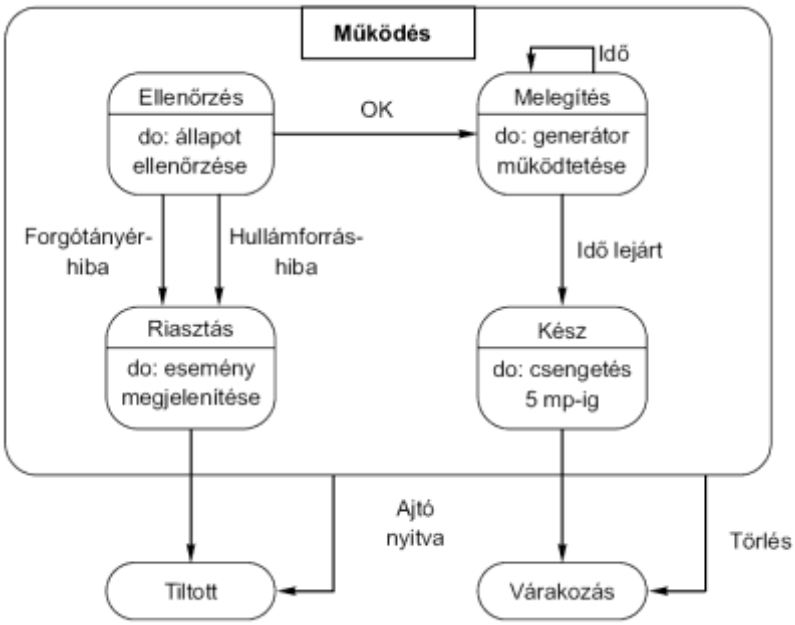


A diagramot ki szokták egészíteni szöveges információkkal az egyes állapotokról, és az egyes ingerekről.

Állapot	Leírás
Várakozás	A sütő várakozik az adatok bevitelére. A kijelző a pontos időt mutatja.
Fél teljesítmény	A sütő teljesítménye 300 wattra van állítva. A sütő a „Fél teljesítmény" szöveget jelzi ki.
Teljes teljesítmény	A sütő teljesítménye 600 wattra van állítva. A sütő a „Teljes teljesítmény" szöveget jelzi ki.
Idő beállítása	A melegítési idő a felhasználó által megadott értékre van állítva. A kijelző a választott melegítési időt mutatja, és az idő beállítása után frissítődik.
Tiltott	A sütő biztonsági okokból tiltott állapotban van. A belső lámpa világít. A kijelző a „Nem kész" feliratot mutatja.
Engedélyezett	A sütő üzemkész állapotban van. A belső lámpa nem világít. A kijelző a „Kész" feliratot mutatja.
Működés	A sütő működés alatt áll. A belső lámpa világít. A kijelző a visszaszámlált időt mutatja. A melegítés befejezésekor a csengő öt másodpercig jelez. A belső lámpa világít. A kijelző a „Melegítés elvégezve" feliratot jeleníti meg a csengő hangjelzése mellett.

Inger	Leírás
Fél teljesítmény	A felhasználó megnyomta a fél teljesítmény gombot.
Teljes teljesítmény	A felhasználó megnyomta a teljes teljesítmény gombot.
Időzítő	A felhasználó megnyomta az egyik időzítőgombot.
Szám	A felhasználó számgombot nyomott meg.
Ajtó nyitva	A sütő ajtaja nyitva van.
Ajtó zárva	A sütő ajtaja be van zárva.
Indítás	A felhasználó megnyomta a start gombot.
Törlés	A felhasználó megnyomta a törlés gombot.

Ökölszabály mind a két fajta diagramnál, hogy egy A4-es lapnyi diagram áttekinthető… Onnantól nem igazán. Nagy diagramoknál az állapot-átmeneti modell bevezeti a szuperállapotot, és azt alkalmazza egy csoport helyett, és azt másik A4-es lapon kifejti.

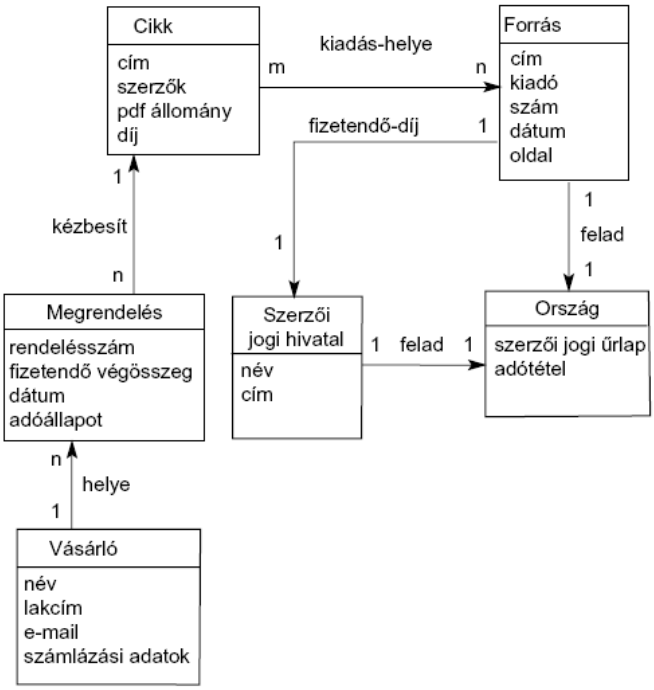


Adatmodellek:

Strukturális modellek, de az adatok struktúráit és a köztük lévő kapcsolatot írják le. A modellezés mindig egy sémát eredményez. pl.: ER-modell (szemantikus modell).Vannak változatai pl.: EER-modell. Ezek a szemantikus modellező eszközök ott vannak a technológiákban.

Funkcionális adatmodellek: funkcionális paradigmában lévő adatmodellek.
OO-adatmodellek: pl.: az OMG OO adatmodellje amely mögött az UML van.
OR: Objektum relációs adatmodell: pl.: SQL99/2003/2008.

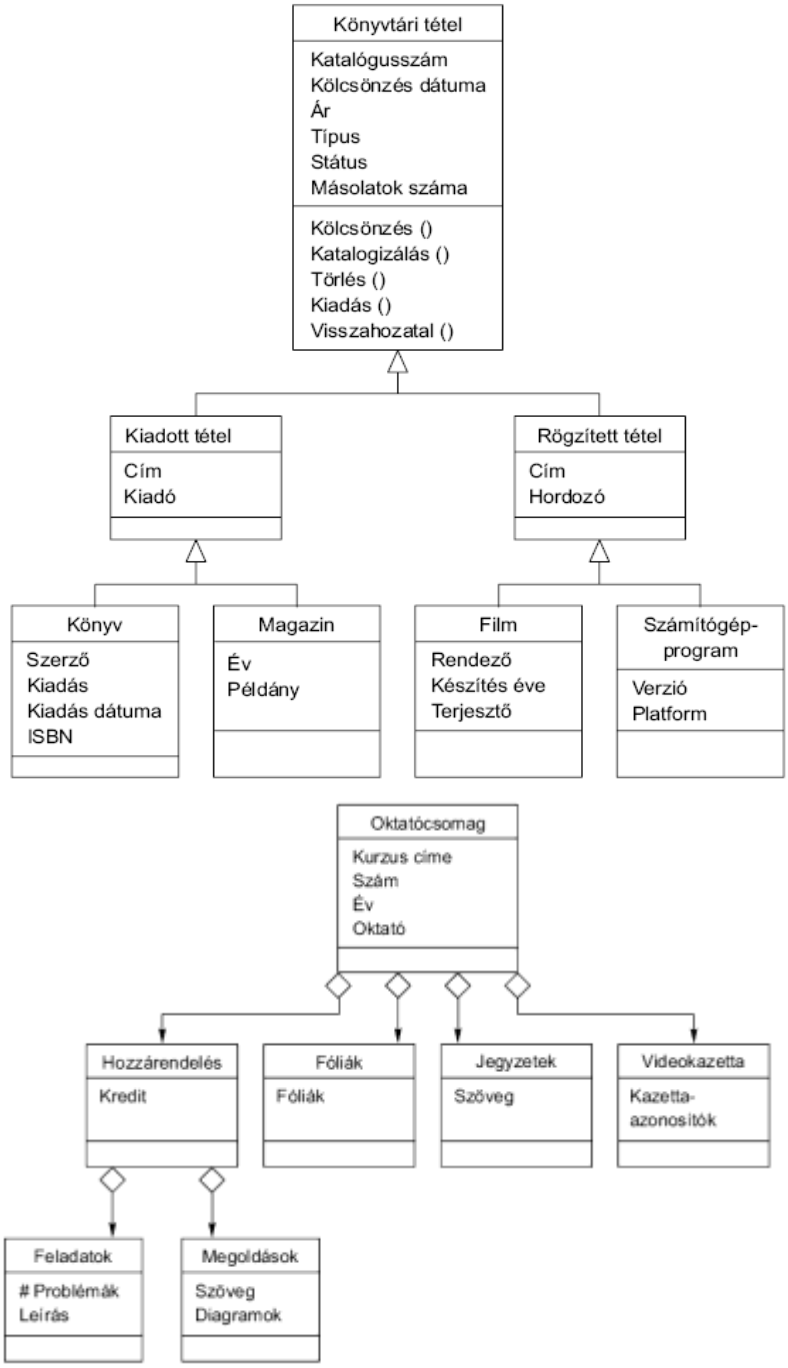
PÉLDA: Szemantikus modell:



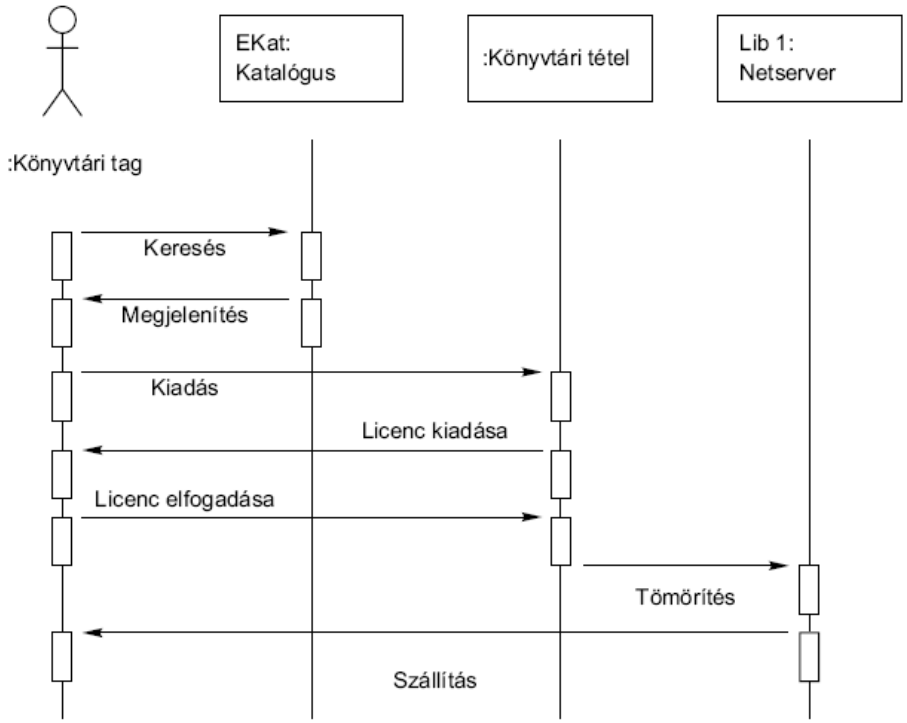
Adatszótár: A koncepcionális sémák mögé s, a sémán megjelenő fogalmi magyarázat.

Név	Leírás	Típus	Dátum
Cikk	A LIBSYS-t használók által megrendelhető publikált cikk részletei.	Egyed	2002. 12. 30.
szerzők	A cikk szerzőinek neve, akik majd a díj egy részét kapják.	Tulajdonság	2002. 12. 30.
Vásárló	A cikk másolatát megrendelő személy vagy vállalat.	Egyed	2002. 12. 30.
fizetendő-díj	1:1 kapcsolat a Cikk és a Szerzői jogi hivatal között, aminek a díjat kell fizetni.	Kapcsolat	2002. 12. 29.
lakcím(Vásárló)	A vásárló lakcíme. Minden szükséges számlázási információhoz ez lesz használva.	Tulajdonság	2002. 12. 31.

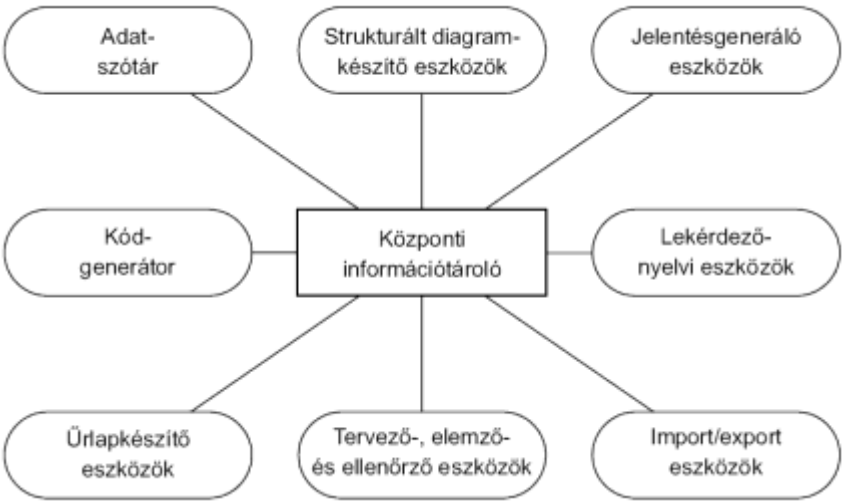
PÉLDA: UML

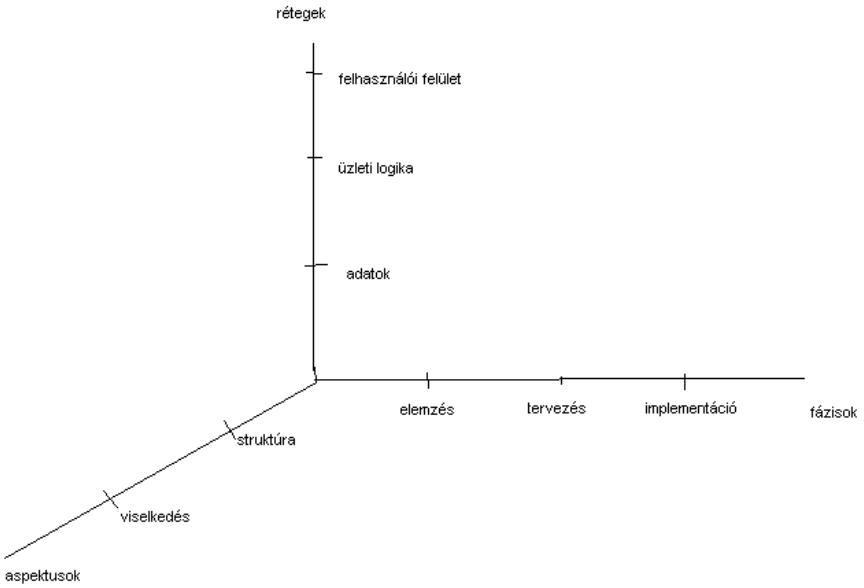


PÉLDA: Szekvenciális modell:



Automatizált eszközrendszer, technológia ilyen eszközöket szolgáltat:





Rétegek: Az adatok modellezése és kezelése önmagában egy olyan dolog, ami felér egy teljes rendszerfejlesztéssel. Egy kis adatbázis fejlesztés a nagyon belül. A rendszer viselkedését alapvetően az üzleti logika határozza meg. Az üzleti logika meghatározására a követelmények szolgálnak. Az üzleti logika egy funkcionalitás, az üzleti logika implementálása egy külön kérdés, hogy hol meg hova, mert elosztott rendszerek vannak, szerver-kliens, ezek fölött ott van az operációs rendszer, erre rájön az alkalmazás, kérdés, hogy a platformhoz hogyan alkalmazkodik, és hogy a rendszer architektúra hogyan követi le a rétegeket. A felhasználó pedig user interfészen keresztül találkozik a rendszerrel. Majdnem kizárólagos kommunikáció ezeken keresztül történik.

Architektúrális Tervezés

5.előadás 2008-10-07

Részben szolgálja a rendszerkövetelmények megértését, validálást. A fejlesztő és a megrendelő közti kommunikációt. A tervezés első szintje. Jól megtervezett rendszer a mai technológia mellett valószínűleg jól (de legalábbis jobban) fog működni.

Architektúrális minták: lásd előző félév…
Ezek régen kialakultak, és erős a minták újrafelhasználása, az architektúrális elemek felhasználása.
Egy rendszer alrendszerekből áll, ezek modulokra tagolódnak.
Alrendszerek: viszonylag önálló rendszerelemek, amelyek kommunikálnak más alrendszerekkel. Lényeges az alrendszerek interfésze. A kommunikáción túl nem vesznek igénybe szolgáltatást más alrendszerektől.
Modul: csak egy alrendszeren belül. Szolgáltatásokat nyújtanak, és vesznek igénybe más moduloktól.
pl.: Tanulmányi rendszer egy alrendszer.
Szemcsézettség: csak alrendszereknél. Ha a rendszerelemek kicsik és sokan vannak, akkor finom szemcsézettség, ha kevés és nagy, akkor durva szemcsézettség.
Architektúrát befolyásolják az Nem Funkcionális elvárások:
Teljesítmény: ha a teljesítmény a kritikus, akkor durva szemcsézettségű rendszer
Védelem: ha a védelem a kritikus, akkor rétegezett, és a legbelső rétegekben van a védelem bezárva. Durva szemcsézettségű rendszer
Rendelkezésre állás: A rendszer architektúra redundáns elemeket tartalmaz. Közepes szemcsézettséget jelent.
Karbantarthatóság: Finom szemcsézettségű architektúra felé tolja el a rendszert, mert így minél kevesebb elemet kell kicserélni.
Több elvárás is lehet kritikus, ekkor kompromisszumot kell kötni, vagy priorizálni a követelményeket, vagy pedig valaki majd jól eldönti☺.

Architektúra specifikációja az tipikusan diagram, kiegészítve szöveges leírásokkal.
ADL: Architektúra leíró nyelvek, informatikai eszköz, ezt a megrendelő garantáltan nem érti. Ráadásul vagy nem elég absztraktak, vagy pedig eléggé és akkor már az informatikus sem érti. Ezért ezekről leszoktak.

Az architektúrális tervezés kemény szellemi folyamat, nem lehet automatizálni, igazából egy döntés sorozat.

Kérdések:

- Tudunk e újrafelhasználni valamilyen architektúrális mintát?
- Hogyan alakítom ki, hogy osztott rendszer t tudjak csinálni?
- Mire optimalizálom a rendszert?
- Milyen szemcsézettségű lesz a rendszer?
- Mik lesznek az alrendszerek és a modulok?
- Milyen, mikor mennyi lesz a felhasználóval a kommunikáció? Magyarul az architektúra validálása.
- Mi lesz a specifikációs eszköz?

Rendszerarchitektúra összetevői:

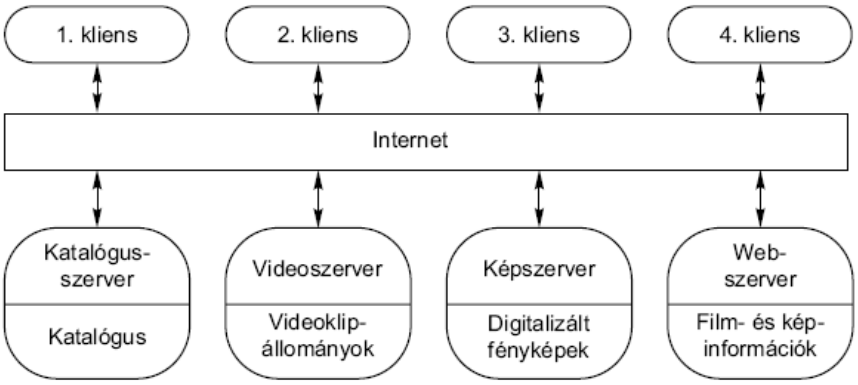
A rendszernek van egy statikus szerkezete. Statikus struktúramodell. Az alrendszerre és modulra való bontást tartalmazza.
Van egy dinamikus folyamatmodell, amely azt modellezi, hogy hogyan kell leképezni a rendszert úgy, hogy majd működőképes folyamatok legyenek mögötte, amikor majd működik. Hogyan kell szétosztani pl.: processzorokra.
Interfészmodell: amivel kommunikálnak az alrendszerek, vagy a modulok.
Kapcsolatmodell: elsősorban a kommunikációra koncentrál. Az alrendszerek és modulok milyen adatokat milyen formában és hogyan adnak át.

Architektúrális minták

Tárolási modell: Az adatok tárolására vonatkozik, adatbázis vonatkozásban. Hogyan tudom szétosztani az adatbázist? Védelmet hogyan oldom meg?
Alapvetően két mintát tartalmaz:
Megosztott adatbázis: egyetlen alrendszer felelőssége az adatok kezelése, és minden más alrendszer ezt az egyetlen megosztott adatbázist látja és használja. Egyetlen adatmodell van, ennek a kezelése az adott alrendszer feladata.
Előnyei: Nem kell mozgatni az adatokat az alrendszerek között. Egyetlen adatmodell van, konzisztencia, védelem, karbantartás egyszerű.
Hátrányai: Minden alrendszernek ugyanazt az adatmodellt kell használnia, egy heterogén rendszernél nagyon nehéz közös adatmodellt kitalálni. Evolúciót az adatmodell generálja, mert ahhoz nem lehet hozzányúlni, mert az összes alrendszert érinti.
Dedikált adatbázis: Minden alrendszernek saját adatbázisa van. Minden adatrendszernek saját adatmodellje van
Előnyei: Külön adatbázis, tehát külön modell lehet, nem kell közös adatmodell.
Hátrányai: Adatokat kell mozgatni, ezért akkor szokták használni, hogy ha az előállító alrendszer fel is használja az adatokat. Evolúció: ha adatokat kell mozgatni, és változik az adatmodell, akkor egy idő után nagyon sok idő megy el az adatkonverzióra.

Kliens-szerver architektúra:

Szerver: szolgáltatásokat nyújtanak bárkinek, aki a szolgáltatást igénybe akarja venni, és nem tudják, hogy ki akarja igénybe venni a szolgáltatást. Általában egy példányban létezik.
Kliens: igénybe veszik a szerverek által nyújtott szolgáltatást, ismerni kell, hogy honnan veheti igénybe, akárhány példányban létezhet.



Előnyei: az elosztottság, kifejlesztése, evolúciója könnyen történhet, új szerver és kliens bevétele könnyen megy, az agilis és iteratív fejlesztések könnyen alkalmazhatóak. Ha a szolgáltatás interfész marad, akkor a szerver könnyen lecserélhető. Nem keveredik a szerver és a kliens, valaki vagy szerver, vagy kliens. Általában osztott adatbázist használó tárolási modell van.

Rétegezett modell:

Egy réteg csak az alatta levő és felette levő réteggel van kapcsolatban, csak azzal kommunikál. Az alatta levő rétegtől kér szolgáltatást, és a felette lévőnek ad szolgáltatást.
Viszonylag merev modell, előnye az, hogy a rétegezett modell kiválóan támogat egy evolúciós prototípus alapú fejlesztést, vagy egy inkrementális közelítést. Biztonságosság és védelem nagyon könnyű.
Az alsó rétegekbe be tudom kódolni a platformfüggő dolgokat, és ezt kicserélve a rendszer már hordozható is.
Viszont nagyon merev modell, bizonyos rendszereknél nem jó pl.: interaktív rendszer nem lehet réteges, vagy bizonyos esetekben nem lehet mindig az, hogy csak a mellette lévő rétegektől kérhet szolgáltatást.
Ez a 3 minta alkalmazható a rendszerre és az alrendszerekre is.

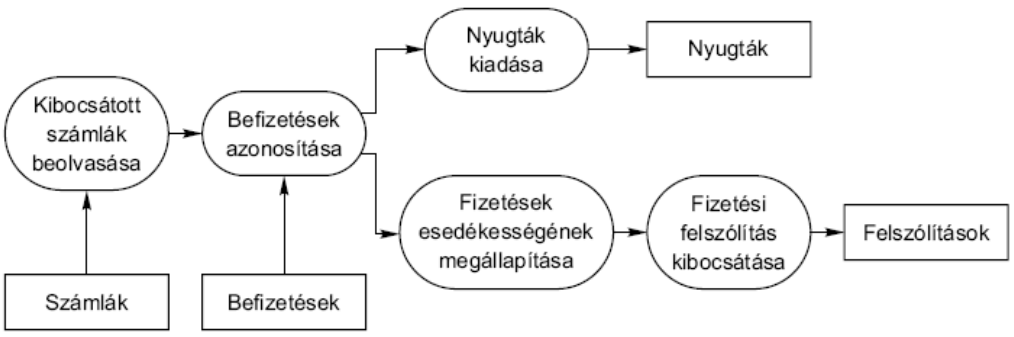
Moduláris felbontás modellek: A következő két minta alrendszerekre való,

OO felbontás:

Az alrendszereket a lazán kapcsolódó objektumok egy halmazára robbantom szét. Az objektumok szolgáltatásokat nyújtanak és vesznek igénybe. Az objektumokat a követelmények alapján hozom létre. A követelmények alapján úgy hogy az objektumokat megfeleltetem valamely követelménynek, valós világ elemet képvisel általában, pl.: hallgató, oktató. Ezek után kategorizálok, vagyis létrehozom az osztályokat. Ezeken belül általánosítás, azaz absztrakciós hierarchiát alakítok ki.
Előnyei: Magas absztrakciós szint, természetes módon lehet objektumokat képezni.
Hátrányai: az objektum túlságosan finomszemcséjű. Ha nagy az alrendszer, akkor túl sok objektum keletkezik, és a lazán kapcsoltság előbb-utóbb átláthatatlanná válik.

Funkció-orientált-csővezeték-modell:

Struktúrált módszertanokból jön, az adatfolyam modell lecsapódása architektúrális szinten. Alrendszert, vagy modult funkciók sorozataként képzelünk el. Ezek transzformációk, és az architektúra ezt követi le, ez alapján építjük fel.
Előnyei: az evolúció viszonylag egyszerű, természetesen megérthető, sok funkció újrafelhasználható
Hátránya: inkább modulon belül, mint alrendszeren belül van, egy alrendszer architektúrája bonyolulttá válik ilyen megközelítés mellett. Alacsony az absztrakciós szintje. Azonos adatmodellt kell vallania a funkcióknak.

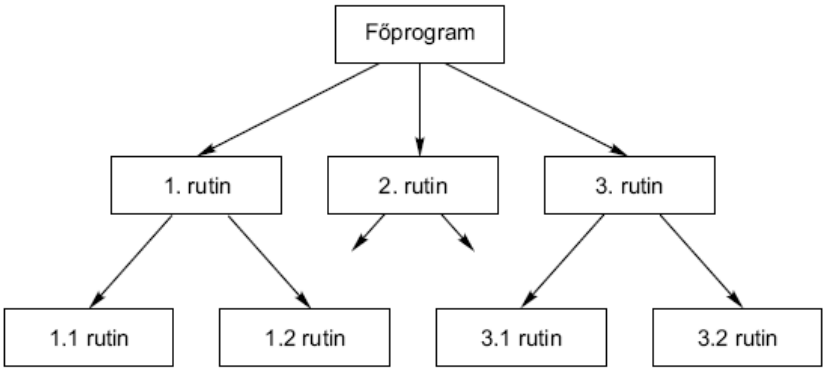


Vezérlési modellek:

Arról szól, hogy az alrendszerek működése az hogyan történjen, azaz hogy a szolgáltatások elérhetőek legyenek.

Központosított vezérlés:

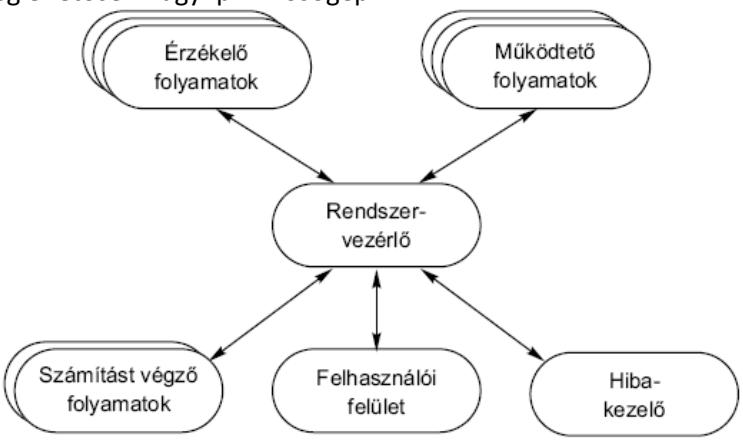
Van egy kitüntetett alrendszer, amely koordinálja az összes többi alrendszer működését.
Ezen belül:
Hívás visszatérés modell, amely a magasszintű nyelvek alprogramhívásának megfelelő.



Ez egy statikus, merev vezérlési modell, ez előnye és hátránya is.
Előnye: nagyon könnyű kezelni, normális működés esetén nincs semmi gond.
Hátránya: Kivételek esetén hajlamos összeomolni, és össze is omlik☺

Másik:

Gyengén valós idejű rendszerekben szokás alkalmazni, ezek azok a rendszerek, ahol a valós idő nem annyira valós. Az időkorlát az meglehetősen nagy. pl.: mosógép.

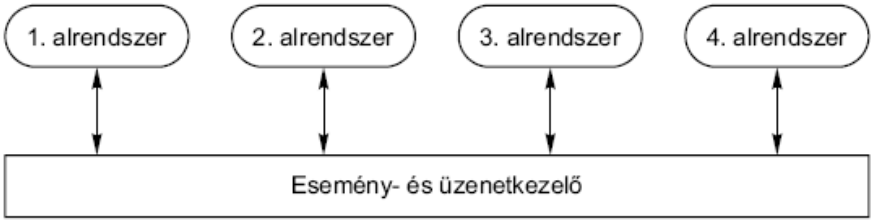


Eseményalapú vezérlés:

Az esemény az mindig külső esemény, azaz inger jön a rendszer környezetéből, és az adja az eseményt, ez bármilyen adat megjelenése lehet.

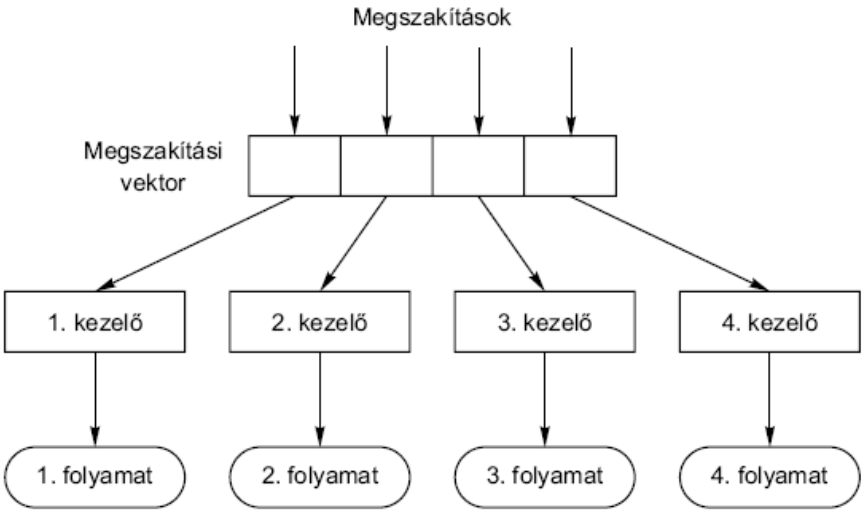
Eseményszórás:

Van egy esemény és üzenetkezelő, az egyes alrendszerek feliratkozhatnak bizonyos eseményekre. Ha bekövetkezik az a valami, akkor egy üzenetet küld az alrendszereknek, amelyek érdeklődtek☺
Előnye: egy esemény egyszerre akárhány alrendszert befolyásolhat, és akár egyszerre is tudnak rá reagálni.
Hátránya: Az alrendszerek nem tudnak egymásról, és hogy a másik hogyan reagál egy adott eseményre.



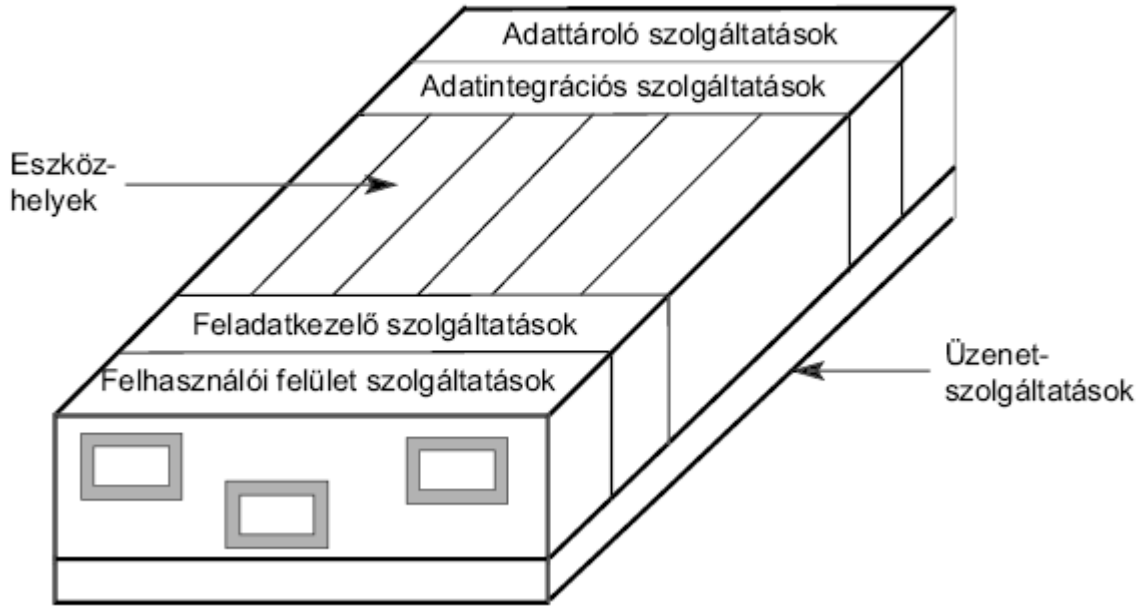
Megszakítás vezérlés:

Szigorúan valós idejű rendszerekben csak ez működik. A megszakítása azonnal odakerül egy alrendszerhez, amely azonnal elindul egy kezelőfolyamatot.



Szakterületi architektúrák:

Nem általánosak, szakterülethez kötőnek, kettő van belőlük:
Általános modell: olyan modellek, melyek az adott szakterületen lévő alkalmazások architektúrájának absztrakciójaként jönnek létre. Olyan architektúrák, amelyekhez ezek után illeszkednek az adott szakterület alkalmazásai. pl.: OSI tehát ezek általában szabványok, általában „de jure” szabványok.
Referencia modell: jóval absztraktabbak, mint az általános modellek. Ezek eleve absztrakciók, idealizált modellek. Az adott szakterület alkalmazásai nem ezt követik ☺, tulajdonképpen csak összehasonlításra való.
Szabvány-referencia architektúrák: „de facto” szabványok, szintén összehasonlításra való.



Ez egy komponens alapú rendszer, az eszközhelyekre lehet bedugdosni a komponenseket.

Osztott rendszerek architektúrái:

Előnyei:

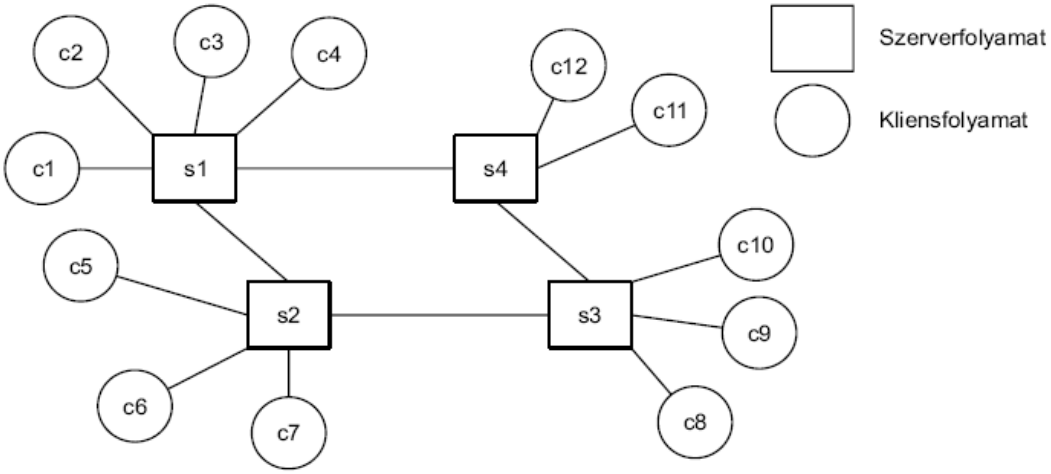
- Erőforrás megosztás
- Nyílt rendszerek tehát könnyű a bővítés és a csatlakozás.
- Konkurens rendszerek.
- Skálázhatóság: Ha megnő az igény pl.: a terhelés, akkor új erőforrások kapcsolhatóak be, illetve átcsoportosíthatóak.
- Hibatűrés: redundancia miatt.

Hátrányai:

- Bonyolultság, a rendszerek alatti platform lehet heterogén.
- Biztonság: A hálózat miatt sokkal sebezhetőbb.
- Menedzselhetőség: Sok hardverelem, sok szoftverelem, a környezet heterogén, sokkal nehezebben kezelhető, mint egy elosztott rendszernél.
- Megjósolhatatlanság: a hálózat nem mindig determinisztikusan viselkedik

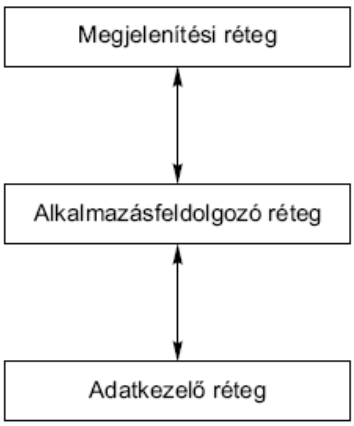
Kliens-szerver architektúra, Osztott objektum architektúra: mind a kettő nagyon elterjedt. Mindkettőnél kérdés az, hogy a platform heterogenitása miatt, a rendszer elemek között biztosítani kell a kommunikációt, az együttműködést. Erre szabványos middleware(köztes szoftver) eszközöket dolgoztak ki.

Kliens-szerver architektúra:

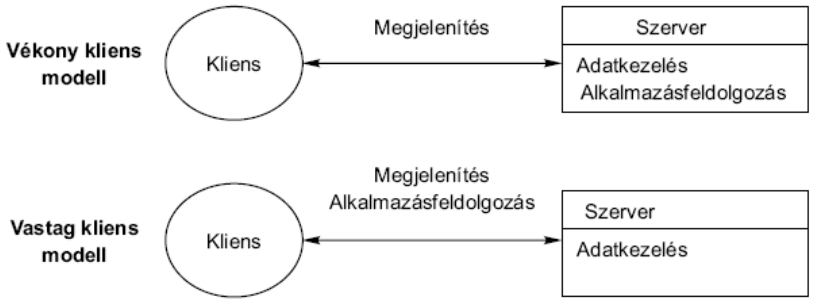


Mindig az alkalmazás logikára kell építeni, és nem a platformra. Van szerverfolyamat, kliensfolyamat, ezek lehetnek egy, de akárhány gépen is, ez nem szempont.

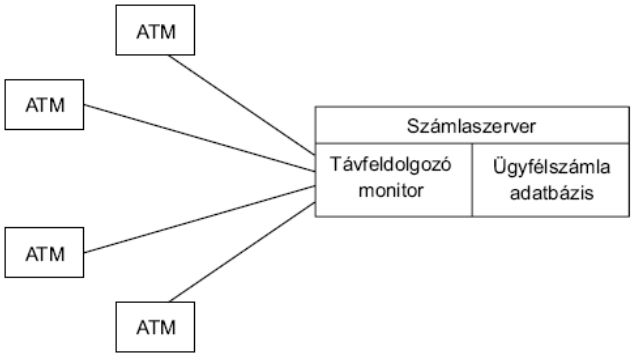
Rétegek:



Kliens szerver megoldásként szokásos a kétrétegű kliens-szerver megoldás, ennek két közelítése van:

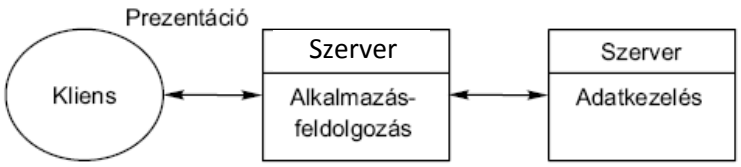


A kliens oldal mindig felelős a megjelenítésért. A vastagnál az üzleti logika a kliensnél van kódolva. A vékony megoldásnál nagyon nagy az adatforgalom, a terhelés a szervernél van. A vastag esetén ki van használva a kliens hardvere is. A vékony megoldás jól alkalmazható akkor, ha egy központosított rendszert akarok elosztottá tenni. A vékony jóval nehezebben skálázható. Kliens folyamatból sokkal több van, mint szerverfolyamatból. A vastag sokkal jobban skálázható, ami azonban probléma, hogy ha sok a kliensfolyamat, akkor ez menedzselési problémákat vet fel.

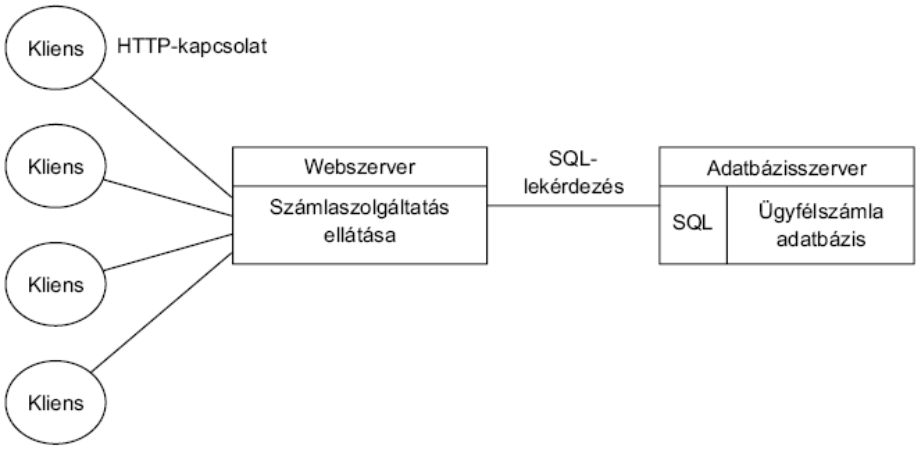


A két megoldás között van egy olyan technikai lehetőség, hogy a szerveroldalon az alkalmazás mobil kódokat tartalmazhat, pl.: applet, vagy acitveX, ezt letölthetjük, és a böngésző szépen csinálja, ezáltal lehet kliensoldalra áttenni üzleti logikát.
Szélsőséges megoldások: az összes folyamat egy gépen fut, vagy az összes kliens és szerver folyamat külön gépen fut.

Háromrétegű:



Mai tipikus megoldás, pl.: egy internetes bank:



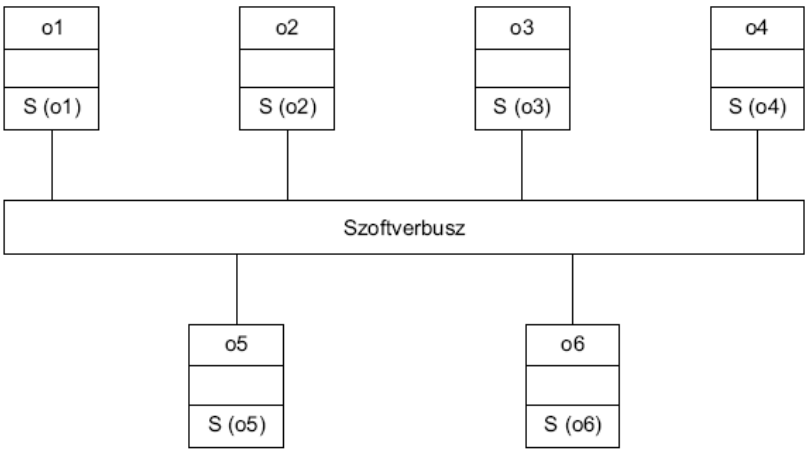
Jobban skálázható, az elosztás sokkal jobban megtehető, és könnyebb továbblépni a több réteg felé. Mert így csak marad a kliensnek az a feladata, hogy prezentáció, és ennyi. Lehet, hogy több adatbázis van mögötte, és akkor lehet beletenni egy integrációs szervert.

Architektúra	Alkalmazások
Vékony kliens elven működő kétrétegű kliens–szerver architektúra	Az ősrendszerek alkalmazásai, ahol az alkalmazásfeldolgozás és az adatkezelés szétválasztása nem célravezető. A számításintenzív alkalmazások kevés adatkezeléssel vagy adatkezelés nélkül, mint például a fordítóprogramok. Adatintenzív alkalmazások (böngészés és lekérdezés) kevés alkalmazásfeldolgozással vagy alkalmazásfeldolgozás nélkül.
Vastag kliens elven működő kétrétegű kliens–szerver architektúra	Azok az alkalmazások, amelyekben a kliensoldali alkalmazásfeldolgozást egy kulcsrakész rendszer (például Microsoft Excel) biztosítja. Számításintenzív alkalmazások adatkezeléssel (például grafikus megjelenítés). Viszonylag stabil végfelhasználói funkciókkal és jól megalapozott rendszerkezeléssel rendelkező alkalmazások.
Három- vagy többrétegű kliens–szerver architektúra	Kliensek százaival vagy ezreivel rendelkező széles körű alkalmazások. Azok az alkalmazások, ahol mind az adatok, mind pedig az alkalmazás változékonyak. Különböző forrásokból származó adatok integrálását végző alkalmazások.

Ősrendszerek azok, melyek központosítottak, architektúrájuk egyszerű vagy nincs is, és bármilyen nyelven készülhettek. Viszont az ősrendszerek egy része meglehetősen jól működik és jól. Ezeket be kellene integrálni.

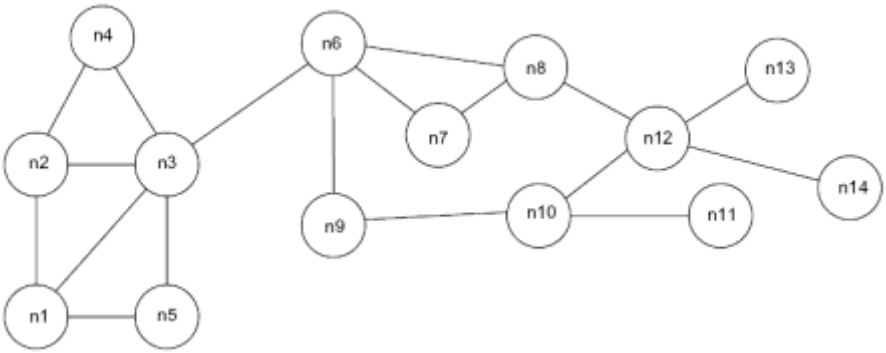
Osztott Objektum Architektúra:

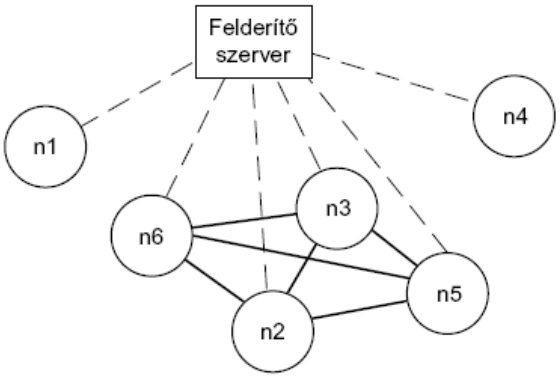
Lényeges különbség a kettő között, hogy a kliensek és a szerverek szerepe aszimmetrikus. A szerverek szolgáltatásokat nyújtanak, és a kliensek veszik igénybe, és szerepük nem felcserélhető, nem összemosható. Az objektum architektúrában nincs ilyen, az objektum viselkedik és kész☺. Absztrakciós szinten ez jobb közelítés, intuitíven nem biztos.



A szoftverbusz(ORB – Object Request Browser) az egy köztes szoftver. Az OO üzenetalapúságát veszi át. Semmit nem tudnak a másik objektumról, csak annyit, hogy van valamilyen szolgáltatás, amit ő igénybe vehet, és nyilván van egy amely ezt nyújtja, és elküldi az üzenetet a szoftverbusznak, és a szoftverbusz feladata, hogy kiválassza azt aki ezt majd megválaszolja. Lehet olyan is, hogy olyan okos az objektum, hogy tudja a másik OID-jét, de a továbbítás feladata akkor is a szoftverbusz feladata marad.
CORBA (Common ORB Architecture) szabvány van mögötte. Ez egy konkrét szabványos ORB megvalósítás.
Előnye: az architektúrális tervezés szintjén nem kell foglalkozni azzal, hogy a szolgáltatás helyét és miként jét eldöntsük. Azt se kell mondani hogy kliens vagy szerver. Skálázhatósága jóval nagyobb: Új objektum, új szolgáltatás, funkció. Hibatűrése sokkal nagyobb: többszörösítem, átmozgatom az objektumokat.
Meg lehet tenni azt, hogy kifejlesztek üzleti/szabvány objektumokat. A két architektúra összehozható. Mindkettő osztott objektum architektúrával jelenik meg.

P2P architektúra: Olyan erőforrás megosztás, mely a kihasználatlan kapacitásokat próbálja kihasználni, pl.: elosztott számításokhoz. Nagy méretű hálózatban lévő gépekre telepíthető.

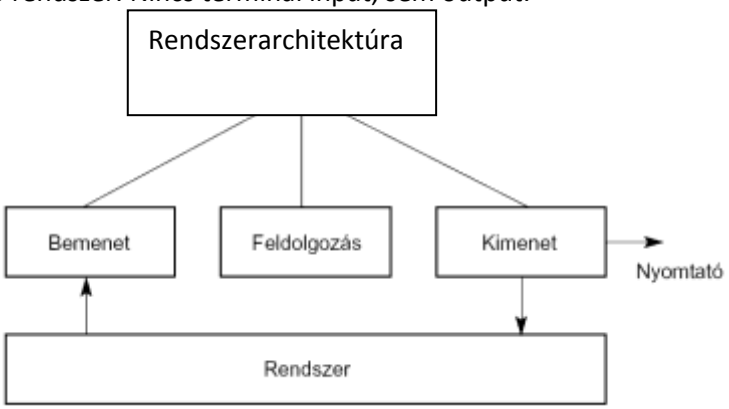




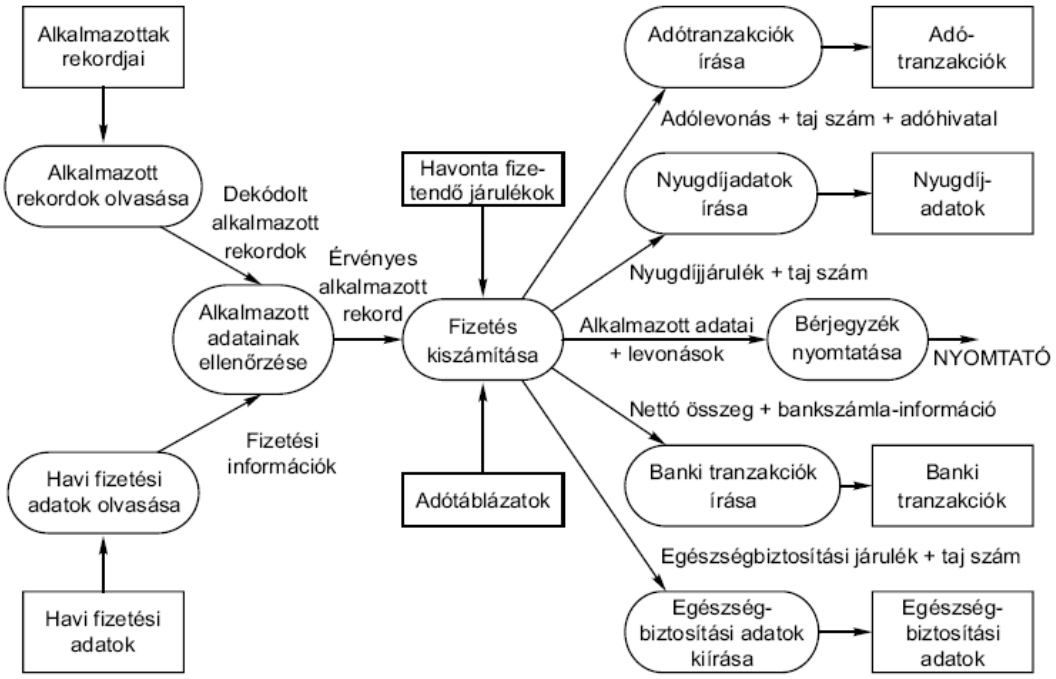
Alkalmazás architektúrák:

Az alkalmazások nagyon különbözőnek tűnnek. Azonban nem feltétlen: bizonyos architektúrális minták annyira felhasználhatók, hogy egy tetszőleges alkalmazás architektúrája ezek alapján kialakítható.
Ezek alapján készíthető egy check lista, ami alapján megnézhető, hogy mennyire jó, vagy nem jó az általam készített architektúra. Az architektúrális minta ismeretében a fejlesztést megszervezhetem úgy, hogy több csapat párhuzamosan fejlesszen alrendszereket. Ha kész szoftvert veszek, akkor a kész szoftverek architektúráját össze tudom hasonlítani ez alapján.

Adatfeldolgozó alkalmazások: adatvezérelt, nem interaktív rendszerek, kötegeltek, adatintenzívek, például egy bérszámfejtő rendszer. Nincs terminál input, sem output.

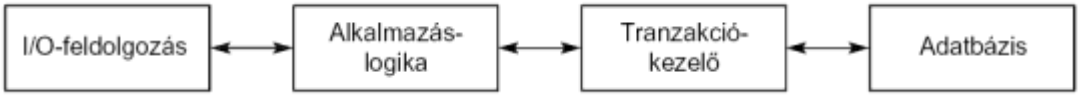


Ráadásul az egyes architektúrális elemeken belül ugyanez a minta alkalmazható, tetszőleges mélységben.
Bemenet: Adatokat szűri, javítja, validálja. A feldolgozó feldolgozza, megy az output sorba, az meg vagy kinyomtatja, vagy visszarakja a rendszerállományba. Tipikusan függvényorientált, állapotmentes architektúra. Ezek az elemek egymással kommunikálnak, de állapotot nem vesznek át.

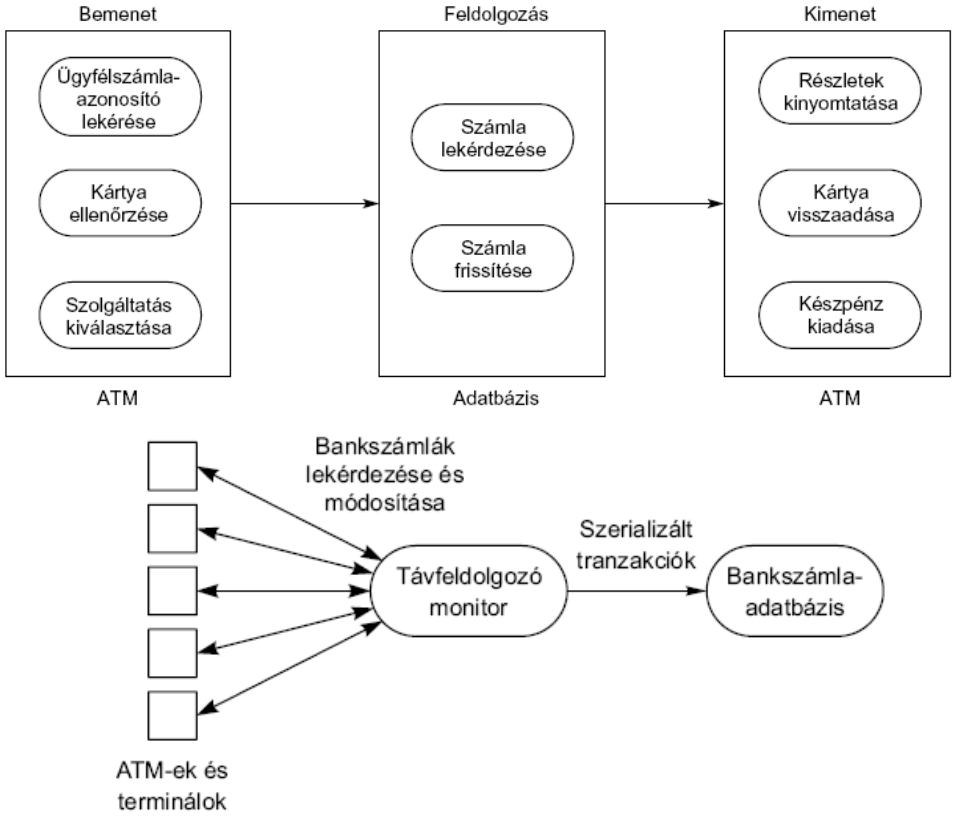


Tranzakciófeldolgozó alkalmazások

Interaktívak, konkurensek, és általában aszinkron jellegűek.



Ez és a kötegelt nem annyira tér el, mert pl.: a kötegeltben összegyűjtjük a tranzakciót, és este kötegelve dolgozza fel. pl.: ATM

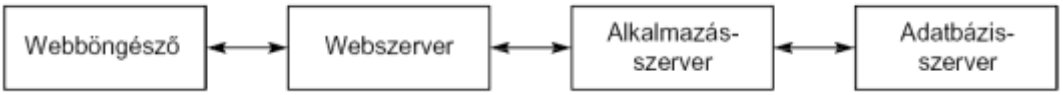


Eseményfeldolgozó alkalmazások
Nyelvfeldolgozó alkalmazások

Információs és erőforráslefoglaló rendszerek architektúrája:

Felhasználói interfész réteg – tipikusan egy böngésző
Felhasználói kommunikációs réteg – autentikációs elem, lekérdezés kezelő elem, információkézbesítő vagy erőforráskézbesítő elem.
Információkezelési réteg – erőforrás lefoglaló elem, ehhez kell egy szabályhalmaz, pl.: jogosultság-kezelő elem.
Adatbázis (tranzakció-kezelő réteg)

Órarendkészítő, Könyvtári rendszer, Jegylefoglaló rendszer stb…

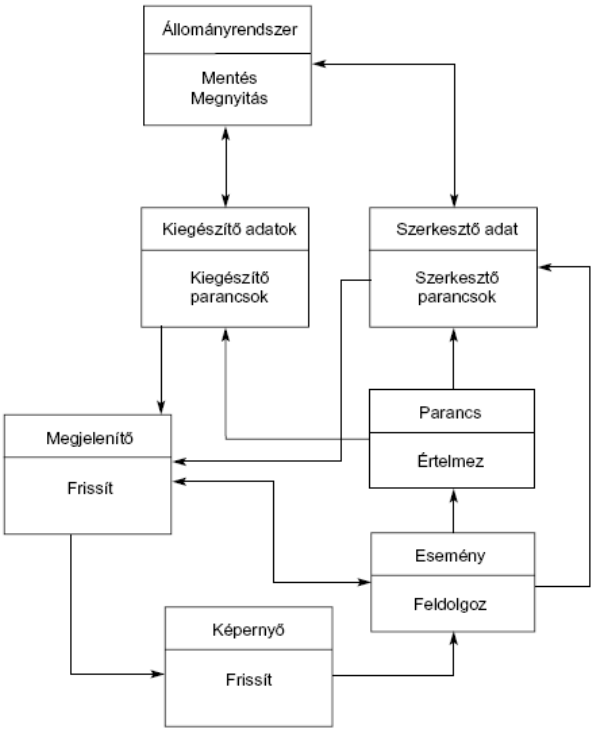


Lehet az, hogy egy gépre pakolom a 4 réteget.
A mai megoldás az, hogy webszolgáltatásokat implementálok. Finom szemcsézettségű rendszert fejlesztek, és webszolgáltatásokat integrálom.

Eseményfeldolgozó rendszerek:

Egyik féle az esemény a rendszer környezetében keletkezik, ezek a *real-time* rendszerek.
A másik az esemény a felhasználói felületen következik be, irodai alkalmazások tipikusan, szokás ezeket *szerkesztő rendszereknek* is hívni:

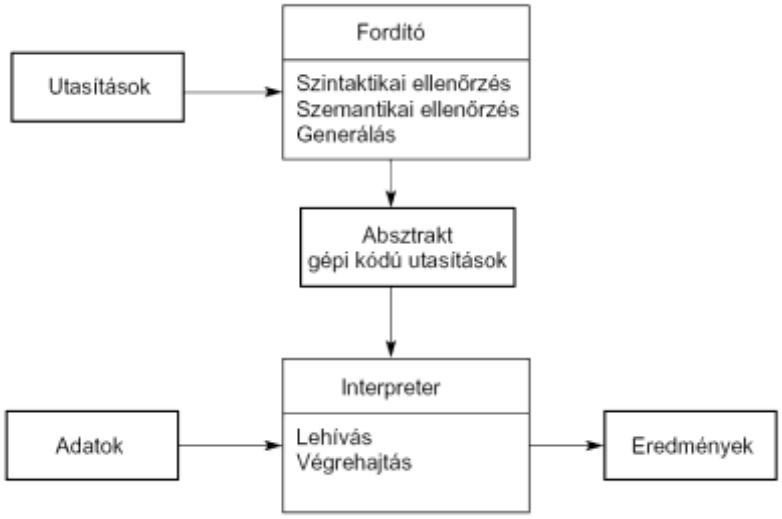
- Dokumentumokkal dolgoznak, tipikusan szöveg, kép, hang, film
- általában egy felhasználós rendszerek
- nincs tranzakció
- általában nincs osztott adatbázis, vagy csak ritkán, de akkor is speciális.
- adatintegritási konzisztencia problémák mások, vagy kevesebbek.
- nagyon gyors válaszidőt tételeznek fel, ezért az aktuálisan szerkesztett dokumentum a memóriában helyezkedik el, így memóriaműveletekkel manipuláljuk, s ez biztosítja a gyorsaságot, viszont ugyebár nem árt néha elmenteni.
- A munkamenetek hosszúak→automatikus mentés.



Képernyő: az architektúrális elem, amelyhez az események kötődnek, időnként frissíteni kell, az itt manipulált dolgok váltják ki az eseményeket.
Esemény feldolgozás: az esemény hathat közvetlenül a dokumentumra, vagy generálhat egy parancsot, amelyet értelmezni kell, például előveszek valamilyen stílust. A dokumentum változását vissza kell csatolni a képernyőre, kell lennie olyan elemnek, amely automatikusan ment, és kell egy állománykezelési elem, amely megnyit, ment stb…
Tipikusan nincs központi elem, hanem ezek párhuzamosan működnek, és közvetlenül kommunikálnak egymással, és adott esetben bizonyos rendszerelemek redundánsan kell, hogy tudjanak működni.

Nyelvfeldolgozó rendszerek architektúrája

Egyaránt ideértjük a természetes nyelv feldolgozó rendszereket, illetve a formális nyelv lefordító rendszereket.



A fordítóprogram architektúrája szabvány architektúra. Lexikális elemző, szintaktikai elemző, szemantikai elemző és kódgeneráló. Van egy tároló, ami egy szintaxis fa.
Az IDE-k megjelenésével folyamatból átmegy tároló architektúrába, ahol megjelenik egy absztrakt szintaxis és egy konkrét szintaxis leírás. Környezet érzékeny szerkesztő, formázó, optimalizáló, nyomkövető.

TERVEZÉS

Objektum-orientált tervezés

A való világot szétrobbantjuk egymással lazán összefüggő objektumokra, az összes jellemzőjükkel együtt, aktív objektumok. Ezeken az objektumokon építem fel az architektúrát, a tervet az implementációt, és minden mást.
Az OO elemzésnek az a feladata, hogy megtalálja a probléma objektumokat, ezek a való világ objektumai. Az OO tervezés az olyan absztrakciók kialakításáról szól, ahol a probléma objektumok megjelennek a tervben, de mellettük megjelennek megoldásobjektumok, ezek szorosan együttműködnek, az implementáció mindkettő implementációját jelenti valamilyen OO nyelven. Az OO közelítés nagy előnye az, hogy az elemzés, tervezés, implementáció során az átmenet sima, nincsenek leképezések, rések, nem kell terminológiát váltani, ugyanaz a fogalomrendszer megy végig. Ha netán még OO adatbázisrendszert teszek alá, akkor még ábrázolás szintjén is objektumok maradnak az objektumok. Egy nagy hátránya van, hogy az objektum túlságosan kicsi, mert objektumokban gondolkodni nagyon kis szemcsézettségű rendszert tételez fel, az osztályok száma hihetetlenül megnőhet.
Ökölszabály: Az a rendszer, ahol az osztályok száma 100 felé nő, az már nagyon komplex.

Pozitív dolgok:

- Bezárás: az implementáció be van zárva, csak az interfész érdekel, az implementációt el lehet halasztani jó messzire. Párhuzamos fejlesztést nagyon segíti, mert interfészek segítségével mehet, ez a karbantartást is nagymértékben elősegíti, mert ha az interfészt nem érinti, akkor mindegy, hogy hogyan van implementálva.
- UML
- OO módszertanok: RUP – nagy méretű projektekhez valók, majd jönnek az agilis módszertanok, ezek kis és közepes méretűekhez jók.

Hogyan lehet azonosítani a problémaobjektumokat?

1. Követelményspecifikáció szövegét vegyük elő és a következőket csináljuk: nézzük meg a főneveket. A főnevekből lesznek az attribútumok és az objektumok. Ha valahol azt tudjuk megállapítani, hogy a főnévnek nincsenek jellemzői, pl.: jegy, akkor valószínűleg attribútum, de ha vannak, akkor valószínűleg objektum lesz, pl.: oktató. Eztán szedjük az igéket, ezekből lesznek a szolgáltatások és műveletek, pl.: jegybeírás.
2. Az adott szakterület fogalmait, szerepköreit, interakcióit kell megismerni nagyon jól, ezt nem lehet egy fejlesztésen keresztül megtenni.

Mindkét technika alkalmas arra hogy adatmodellt gyártsak?

3. A rendszer viselkedését derítsük föl, vagy a jelenlegi rendszer, vagy a leendő rendszer viselkedését.
4. Forgatókönyv alapú technika, a követelményeket is forgatókönyvek segítségével készítjük el, akkor a forgatókönyv segítségével azonosítom a problémaobjektumokat.
5. Vegyes technikák alkalmazása.

Egy OO terv specifikálása UML-ben történik, a terv statikus és dinamikus modelleket és terveket tartalmaz. Osztályok és a közöttük lévő viszonyok jelennek meg. A dinamikus tervben viselkedés jelenik meg, valóban objektumokkal foglalkoznak.
OO interfész-specifikáció: egy objektumnak több interfésze lehet. Az OO interfésztervezést szerződés alapú tervezésnek hívják. Az implementáció bármilyen lehet. Az UML nem alkalmas interfész tervezésre.

Felhasználói felület tervezése

A felhasználói felületek alapvető szerepet játszanak, mert a felhasználó csak ezt látja, csak ezen keresztül találkozik a rendszerrel. Fizikai és mentális igények alapvetően középpontban kell, hogy legyenek.

Mentális

Mágikus 7: a rövidtávú memória olyan, hogy nagy általánosságban 7 dolgot tud mejegyezni. Tehát ha a UI arra kényszerít hogy egyszerre többet kezeljek akkor a felhasználó nem lesz képes kezelni.
Az ember elfárad, hibázik, tehát fel kell készíteni a UI-t hogy az embert minél kevésbé fárassza, másrészt arra, hogy hibázik, és ezt a hibázási lehetőséget ki kell küszöbölni minél hamarabb.
Az emberek sokfélék mentálisan, tehát a UI-okat az ízlések is befolyásolják, ki mit szeret látni, meg mit nem, az információk hogyan nézzenek ki ez sokféle lehet.

Fizikai

Az ember kézmérete, színtévesztés, bizonyos fizikai hiányok például rövidlátó, egykarú, szóval lényeg a felhasználói felület akadálymentesítése. Az ember jobb alulra koncentrál először☺ már aki.

Ökölszabályok:

- Felhasználói jártasság elve: a felületen olyan terminológiát használjunk, amely terminológiát a felhasználók értenek, ismernek, alkalmaznak. Olyan objektumokat használjunk, amely egyértelmű,
- Felhasználói felület konzisztenciája: Két ugyanolyan dolog ugyanazt jelentse, ugyanúgy kelljen használni. Ha valamihez egyszer kell klikkelni, akkor mindenhol egyszer kelljen. Egy alkalmazáson belül alacsony szintű konzisztencia. Ha külön alkalmazásokban is így van, akkor az magas szintű konzisztencia.
- Minimális meglepetés elve: Van valami működési profil, ami ki alakul használat közben, és jobb esetben nem történik más, amikor ugyanazt csinálom☺, másrészt ha igen akkor az nem nem
- Visszaállíthatóság elve: az ártalmas tevékenységeket meg kell erősíteni. Ellenőrző pontok képezése.
- Felhasználói támogatás: súgó. Egy jó súgó az egy sok belépési ponttal rendelkező szöveggráf. Megtervezése nem egyszerű dolog.
- Felhasználói sokszínűség elve: színek, bettűméret

Felhasználói interakciók

Amelyekkel a felhasználó kommunikál a rendszerrel, parancsot ad stb…

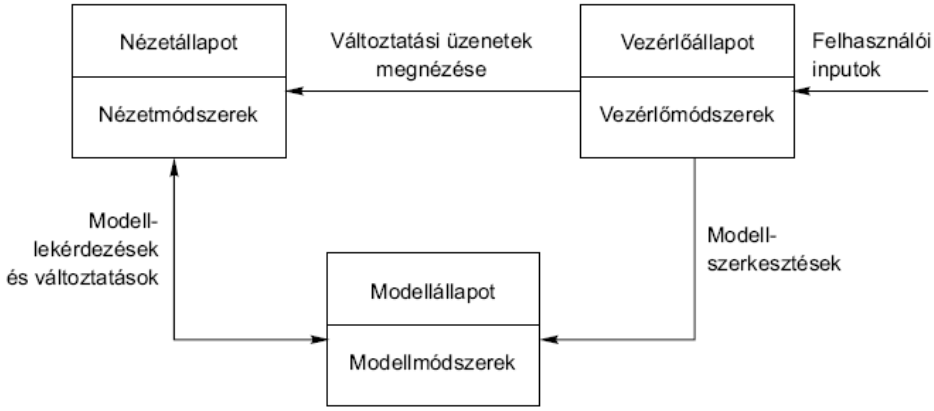
- Közvetlen manipuláció: ekkor a felületen valamilyen objektumok jelennek meg, ezek általában ikonok, de más is lehet, olyan felületelemek, amelyet a felhasználó valamilyen eszközzel közvetlenül tud manipulálni. Ez fog eredményezni valamilyen hatást.
  Előny: Intuitív, könnyen tanulható és alkalmazható, bizonyos esetekben nem implementálható.
- Menürendszer: A menü megfelelő elemét kiválasztva idézi elő a felhasználó a megfelelő hatást.
  Előny: Kevés gépelést igényel, könnyű a használata, olyan dolgok is megfoghatók, amik objektummal nem.
  Hátrány: a menürendszer bonyolulttá válhat, a hetedik almenüben nem feltétlen tudjuk, hogy mi van☺, egy idő után lassú bizonyos felhasználóknak.
- Űrlap kitöltés: Adatokat elsődlegesen bevinni a billentyűzetről tudunk. Az űrlap tulajdonképpen csak a struktúrát jeleníti meg.
  Hátrány: A gond ott kezdődik majd, hogy ergonómia, vagy pedig hogy ha rosszul van megtervezve, vagy például kitölthetetlen.
- Parancsnyelv: grafikus világ előtti kommunikáció.
  Előnye: legrugalmasabb, jobban, mint a közvetlen manipuláció, tömör.
  Hátrány: meg kell tanulni, rossz a hibakezelés, ha valamit, elgépeltem akkor elgépeltem☺
- Természetes nyelv: nem szabványos, de próbálják elterjeszteni. Egyrészt egy speciális parancsnyelv, ami magyarul van, és mindezt írjuk. A másik hogy beszélünk korlátozott szókészlettel.
- Webes felület: ott is ugyanezeket az elemeket használjuk csak valamilyen script nyelvvel. Nagggyon divatos☺

Alkalmazás interakció

Amikor az alkalmazás próbál velünk kommunikálni.

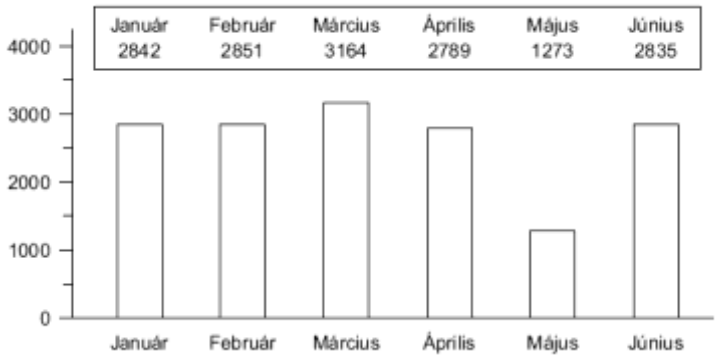
- Adatmegjelenítés
- Üzenetmegjelenítés

MVC minta:

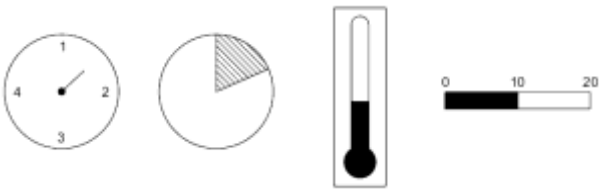


A felhasználó oldaláról figyelembe veendő tényezők:

- Ha az adatok numerikusak, akkor az abszolút érték, vagy a relatív érték az, ami érdekes.
- Kell-e valamit tenni egy felhasználónak, hogy ha egy érték megváltozik?
- Kell-e az, hogy valamelyik megjelenített információt közvetlenül manipulálni tudja?
- Numerikus vagy szöveges információt kell-e megjeleníteni. A numerikus megjelenhet szövegesen, vagy grafikusan is. Numerikusnál a szöveg, relatívnál a rajz a preferált



Különböző kedvenc megjelenítések:



Színek a felületen: ökölszabályok

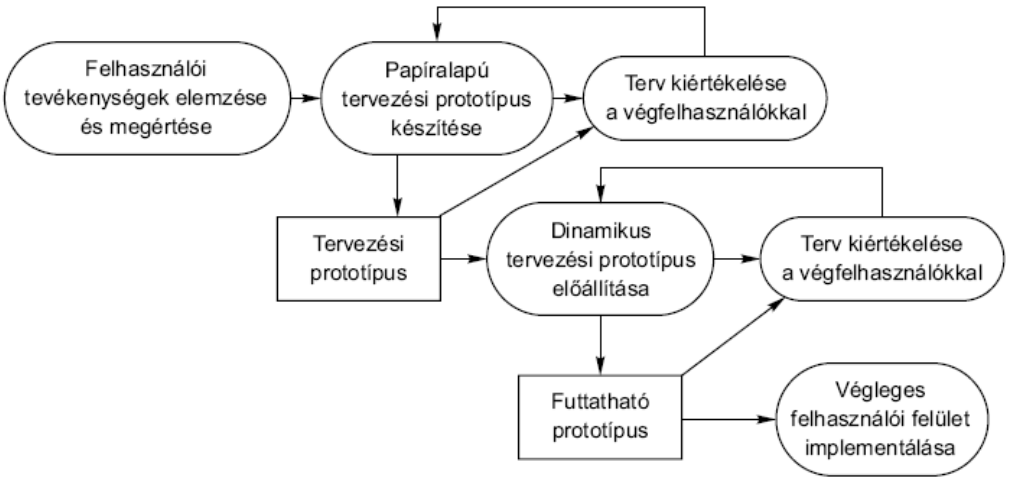
- a felületen a különböző színek alkalmazásánál tanúsítsunk önmérsékletet, tehát kevés színt használjunk a felületen, ez kb. 3-4, de itt is maximum a 7.
- A színt próbáljuk következetesen használni, bizonyos felhasználói interakciókhoz kössünk színt, amennyiben arra szükség van, pl.: figyelmeztetés, de azért ne kössünk mindegyikhez.
- A színek megválasztása az nem informatikai probléma, hanem dizájn, másrészt kognitív probléma, mert az emberek különbözőek.
- Több pasztellszínt ne használjunk egyszerre, mert szemrontó.
- Bizonyos színek kizárják egymást.
- A színek megjelenítése eszközfüggő: A képernyő és a kivetített szín különböző. A nyomtatott és a felhasználói felület színe különbözhet a való világban.
- Konzervatív módon bánjunk a színekkel, színkombinációkat gondoljuk végig.

Üzenetek a felületen

Amivel a rendszer működés közben informálja a felhasználót. Megtervezésük:

- A felhasználót mindig informáljuk, ha valami történik, de egyébként ez nem derül ki a felületből.
- Ne legyen olyan szituáció, amikor a felhasználó nem tudja hogy mi történik.
- Az üzeneteknek figyelembe kell vennie a környezetet, azt amiben használják a rendszert. Terminológiában, megfogalmazásban, hosszában.
- Figyelembe kell venni a felhasználók tapasztalatát, az üzeneteket konfigurálhatóvá tenni.
- Az üzenet stílusa: segítőkésznek kell lennie, és nem pedig sértőnek, vagy gúnyosnak, vagy viccesnek lennie.
- Kultúra: adott esetben figyelembe kell venni, hogy milyen körben működik a rendszer, tegeződés/magázódás, egyik se vagy mind a kettő. Arab/Ázsiai/Amerikai stílus.

Felhasználói felületek tervezése:



A felhasználók felmérése során derül ki az, hogy milyen jellegű felhasználókat kell támogatnia a rendszernek. A felületre koncentrálunk a rendszer nélkül.
Elemzésnél: feladatelemzés, etnográfia, interjúk és megfigyelés.

Feladatelemzés:

Csinálunk egy hierarchikus feladat felépítést a felhasználó oldaláról, azaz hogy mit végez és az milyen alfeladatokra bontható. Itt is megjelenik a forgatókönyv, ez az elemzést is jól szolgálja.
Interjú: olyan interjú, ami a felhasználóra vonatkozik,

Prototípuskészítés

Felhasználói felületek prototípus alapú közelítése, agilisnál előtérbe kerül, de már előtte is használják. Először csak papíron és nem leprogramozva, magyarul felhasználói felület makettet csinálsz a felhasználóval közösen. Elképzeléseiket próbáljuk pontosítani, majd ebből jön a gépi prototípus. Mind a két típusú prototípust használják, de általában evolúciósat szoktak. A felhasználóval közösen nagyon gyorsan el tudunk jutni egy olyan felülethez, ami általa jól használható, és ez által is pontosítjuk a funkciókat is.

Prototípus készítése:

Tipikusan scriptprogramozás, másik lehetőség hogy vizuális nyelveket használunk. Harmadik közelítés, hogy webes megoldást alkalmazunk. Mind a 3 technika alkalmazható együtt.
Értékelés: ha tudunk hozzá metrikát kapcsolni, akkor jó. Ezek általában webes felület metrikák, a gond az, hogy egy részük szubjektív.

Ezért inkább minőségi, mint mennyiségi jellemzők vannak, néhány szempont:

- Tanulhatóság: Mennyi idő alatt jut el a végfelhasználó odáig, hogy a napi munkájában úgy tudja használni a felületet, hogy hasznot termeljen a cégnek, ez metrikálható.
- Sebesség: A felhasználói interakciók és az üzenetek sebessége, ami nem feltétlen a felhasználói felületen múlik, hanem a felhasználói felület által előidézett funkció sebességén.
- Robosztusság: Mennyire hibatűrő a felhasználói felület. Ez esetenként mérhető, máskor meg nem.
- A Megelégedettség (User Experience): Egy bizonyos határon túl teljesen szubjektív.

A felhasználói felületek értékelésénél kognitív és művészeti ismereteket is be kell vetni:

- Kérdőív: olyan szempontokat kell belefogalmazni, hogy az tényleg értékeljen.
- Etnográfia: a használat közben megfigyelni a felhasználót, adott esetben szem, kéz vagy szájmozgást☺. Pillanatfelvétel, vagy videofelvétel.
- A szoftverbe olyan kódrészleteket illesztek be, amely profilt készít a használatáról. Figyelni kell rá, hogy a profil valóban értékelhető legyen.
- A felhasználói felülethez odarakunk egy olyan funkciót, amely segítségével a felhasználó kifejezheti a véleményét.

Ergonómia

A felhasználói felület kényelmesen, kevés fáradtsággal, élvezetesen lehessen használni. Nem igazán mérhető, tipikusan minőségi dolog.

- Lehetőleg egymás alá
- helyközöket kihagyni közöttük
- a hasonló dolgokat csoportosítani
- Kultúra specifikus: jobbra le menü nem feltétlen jó az araboknak.

Felhasználói dokumentáció:

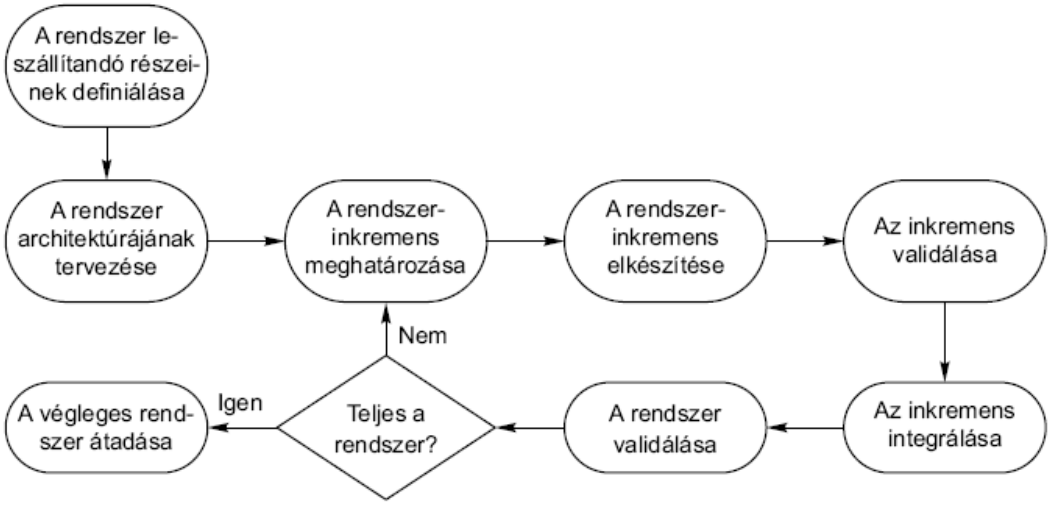
Rosszabb esetben a rendszer része, amely általánosságban a következő dokumentumokat tartalmazza:

- Funkcionális leírás: kívülről látja a rendszert, és a rendszer szolgáltatásait írja le nagyvonalakban. Rövid dokumentum, a főnököknek és a vásárlóknak szól.
- Bevezető kézikönyv(User manual): egy általános felhasználó szempontjából vizsgálja kívülről a rendszert, sokkal részletesebben. Ezekből általában az informatika hiányzik. Menük, képernyőképek stb…
- Referenciakézikönyv(crazykönyv): belülről szemléli a rendszert, ebben már benne van az informatika, ami ahhoz szükséges, hogy a rendszert használó cég informatikusai alappal tudják szidni a rendszert.
- Adminisztrátori kézikönyv(nagy rendszerek esetén): ez a rendszeradminisztrátoroknak szól, teljesen belülről nézi, üzemeltetési kézikönyvnek is hívják. Benne van a konfigurálás.
- Telepítési útmutató: a telepítéshez szükséges.

Ezek mind egybevonhatók, kivétel a funkcionális.

Gyors alkalmazásfejlesztés: RAD(Rapid Application Developement)

a 90-es évekre az lett az alapvető irányelv, hogy minél hamarabb történjen a fejlesztés, a szoftvert minél hamarabb igénybe lehessen venni. Itt alakult ki az iteratív, inkrementális fejlesztés.
A rendszerfejlesztés minden egyes fázisát összemossuk, vagyis párhuzamosan történnek, a rendszer bizonyos elemeire végbemennek, és utána a többire.



Az inkremensekre vonatkozik a specifikáció, tervezés, implementáció, és nem a teljes rendszerre.
Az inkremens validálásába a felhasználó keményen be van vonva. A felhasználónak aktívan részt kell vennie a fejlesztésben: a meghatározásban és a validálásban.
A felhasználói felület fejlesztése valamilyen speciális fejlesztői környezetben automatikusan, vagy félautomatikusan történik.
A rendszer méretétől függetlenül alkalmazható a modell, és azon belül a fázisok, mert a rendszerhez képest az inkremens nagyon kicsi.
Az egyik probléma az egésszel a felhasználó: vagy alkalmas, vagy nem arra, hogy részt vegyen a fejlesztésben.
Ha csak az inkremens részéről van specifikáció és tervezés, akkor az egész rendszerre vonatkozó nincsen. A rendszerhez nincsenek dokumentációk, és verifikációs problémák lépnek fel. Kérdés, hogy a megrendelő mi után fizet? Mert ha nincs rendszer specifikáció, akkor munkaórát tud fizetni, és nem tudja kifizetni a rendszert, így aztán a szerződések problémásak, és maga a projekt is problémás. A rendszer nincs verifikálva, mert nincs mihez, és ez jogi problémákat vethet fel.

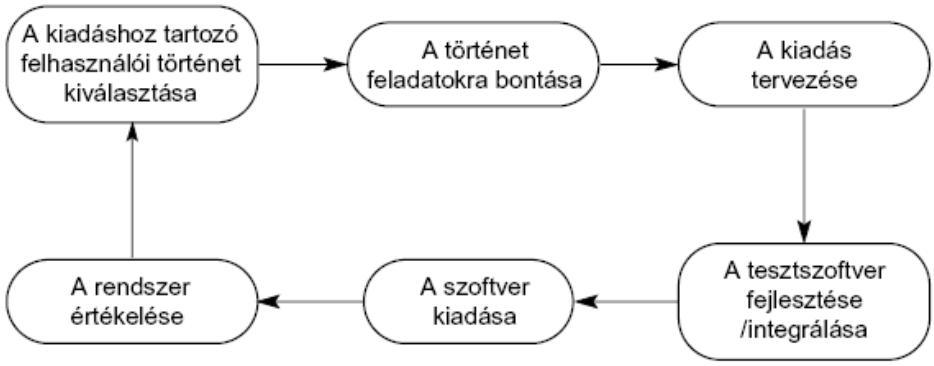
Agilis módszertanok

A 90-es évek végére kialakulnak az Agilis módszertanok, a klasszikus módszertanok tagadásaként. A 80-as nagy rendszerei kritikus rendszerek, ahol fontos a verifikáció. A 90-es években megjelennek kis cégek főleg a webrobbanás után, és a nagy módszertanok nem igazán felelnek meg, mert mindig változik a rendszer, és ekkor kezdenek el másképpen gondolkozni.

- XP – Extreme Programming(Kent Beck – Implementációs minták)
- ASD
- Scrum
- Ambler mint személy
- RUP mintájára megjelenik az AUP(Agile Unified Process – 2000)

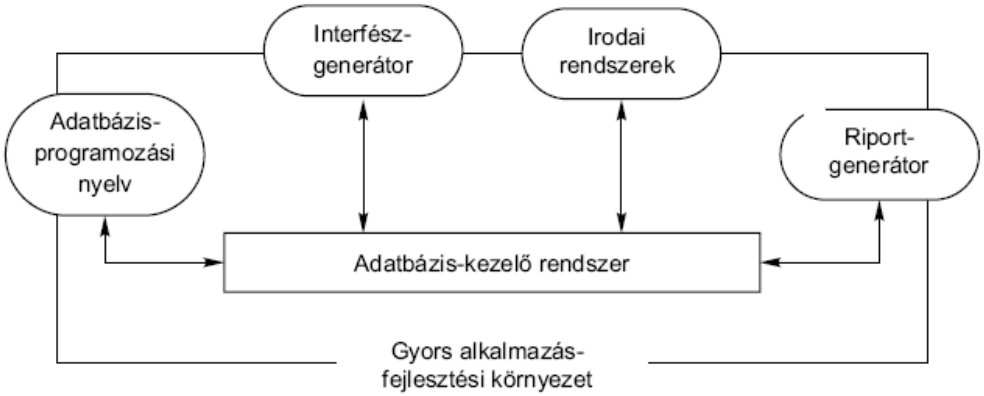
Extreme Programming

Páros programozás, Előrehozott tesztelés



Előrehozott tesztelés: A felhasználó és a fejlesztő History Card-okat gyárt, amik forgatókönyvek. Ez lesz az alapja az összes többinek, és a történet feladatokra bontása után kezdik el tervezni a következő részt, és utána rögtön a tesztek megalkotása történik, tehát először teszteket tervezünk, aztán fejlesztünk, és utána azonnal teszteljük. A forgatókönyv alapján van a tesztelés és a verifikálás.
Páros programozás: a fejlesztők egyenrangúak, és a fejlesztést mindig két ember végzi, birtokolja a szoftvert. Így azonnal van egyféle átvizsgálási tevékenység, azonnal megvan a visszacsatolás, egy refactoring.
Hátránya: nagyon nehéz összeszedni két azonos stílusú, képességű embert. A stílust nehéz felvenni, mert az individuális dolognak alakult ki.

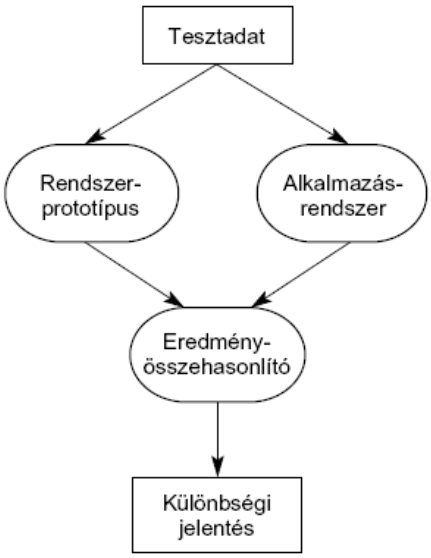
RAD:



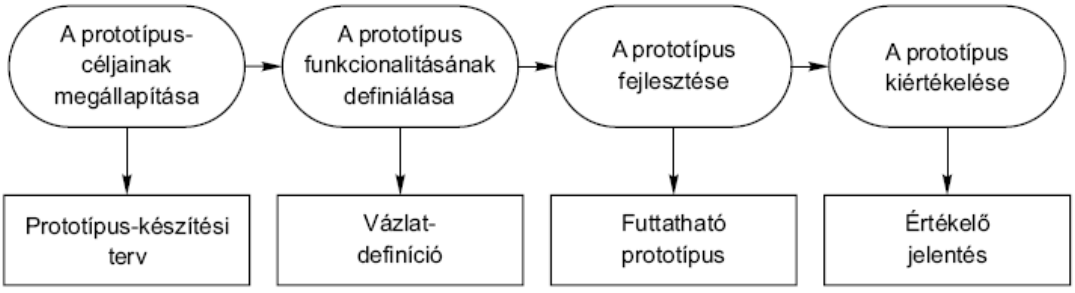
Adatbázis programozási nyelv: eszközöket biztosít az adatbázis használatára
Interfész generátor: űrlapok, felületek készítésére és megjelenítésére alkalmas, ma elsősorban honlap generátor, űrlapok segítségével kommunikálunk.
Irodai rendszerek: pl.: egy szövegszerkesztő a jelentések, vagy egy táblázatkezelő a numerikus adatok elemzésére.
Jelentésgenerátor: az adatbázisban lévő információkból jelentéseket lehet előállítani.
A RAD rendszerek általában vizuális programozási eszközöket is tartalmaznak, a rendszer interaktív fejlesztéséhez nyújtanak segítsége. Az alkalmazásfejlesztők interaktívan készítik a rendszert úgy, hogy az interfészt képernyők, mezők, gombok, és menük megadásával definiálják.

Prototípus(összefoglalóan):

Eldobható és evolúciós. Használhatunk követelmény feltárásnál, ez elsősorban eldobható. Prototípust használhatunk felhasználói felület tervezésénél, ez általában evolúciós, de használhatunk az implementációnál is, elsősorban a tesztelésnél, ez egyfajta verifikálási technika, kimondottan a tesztre alkalmazva.



A prototípus készítés általában:



Újrafelhasználás alapú tervezés

Újrafelhasználás alapú tervezés és fejlesztés növeli a szoftver minőségét. Több szinten értendő.

1. Alprogram, osztályok felhasználása
2. Komponens újrafelhasználás, ahol a komponens egy tetszőleges szoftverelem, amelynek egyetlen jellemzője, hogy nem önállóan használható.
3. Alkalmazás: önállóan teljes alkalmazás felhasználása.

Koncepcionális újrafelhasználás: nem kódot, hanem egyéb akár kódban megjelenő egyéb ismeretet, tudást. Például tervezésben a tervezési minták.

Az újrafelhasználás előnyei:

- megbízhatóság: a felhasznált elemek vagy koncepciók kiérleltek.
- kisebb kockázat, mintha elölről kezdeném
- jobban felhasználhatóak a szakemberek, mert az adott szakterületen megismert koncepciók kerülhetnek előtérbe????
- Szabványosan történhet.
- lerövidíti a projektet->kisebb költség

Hátrányai:

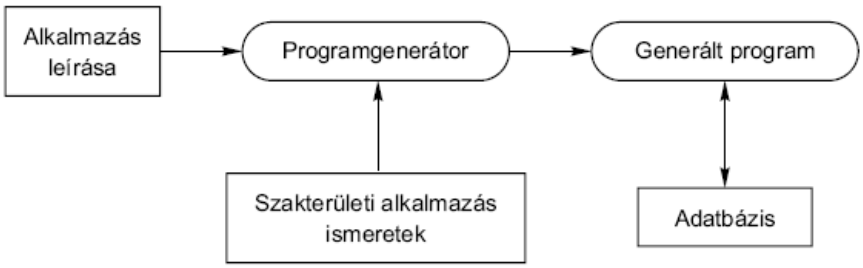
- az újrafelhasználható elemet lehet, hogy meg kell venni
- nő a rendszer karbantartásának költsége is.
- az újrafelhasználható elemek egy része mögött nem biztos, hogy van eszköztámogatás, így az integrálás nehéz.
- más kódját újrafelhasználni nem feltétlen biztonságos.
- a felhasználandó elemekről tudni kell, másrészt olyan kell, ami jó, valószínűleg adoptálni kell.

Aspektusorientált: az OO világ statikusságát próbálja megszüntetni.
Ősrendszerek becsomagolása: amiket kifejlesztettek valamilyen nyelven, és sok éve működnek, és ezek fölé húzunk egy mai technológiával készült részt, ami ezt használja.

Az újrafelhasználás függ:

- A kifejlesztendő rendszerrel kapcsolatos követelményektől.
- A rendelkezésre álló technikákról.
- Attól, hogy milyen értelmű újrafelhasználás, lásd ábra.
- Függ a szakemberek szaktudásától.
- Csak részben függ a szakterülettől.

Programgenerátor: ez egy koncepcionális újrafelhasználás, ahol a generátorba van beépítve az a szakmai tudás, amit újrafelhasználunk. A szakterületi ismeretek nagyon fontosak. Az üzleti alkalmazásoknál fontos igazán.



Előnye, hogy automatizált, a gond viszont az, hogy ezek általános szoftverek, és ha nem általános kell, akkor nem jó, ekkor egy vázat generálunk, amit aztán módosítunk.

(Komponens)Keretrendszerek

Szabvány objektumokat gyártanak objektumgyárak, és ebbe épül bele a szakterületi tudás, és ezt építjük be. Ez nem igazán valósult meg, mert az objektum túl kicsi. Ezután keretrendszerek jöttek létre, ami absztrakt és konkrét osztályok együttese, és magát a keretrendszert használjuk fel újra.
Infrastruktúra keretrendszerek: a rendszer infrastruktúrájának kialakítását célozzák, pl.: kommunikációs vagy felhasználói felület gyártása. Köztes termék keretrendszerek.
Vállalati alkalmazási keretrendszerek: szakterület specifikus, pl.: pénzügy, vagy telekommunikáció.

COTS-integráció (Commerce Of The Shelf): dobozos szoftverek integrálása. Például egy tanulmányi rendszer. Alapvető kérdések:

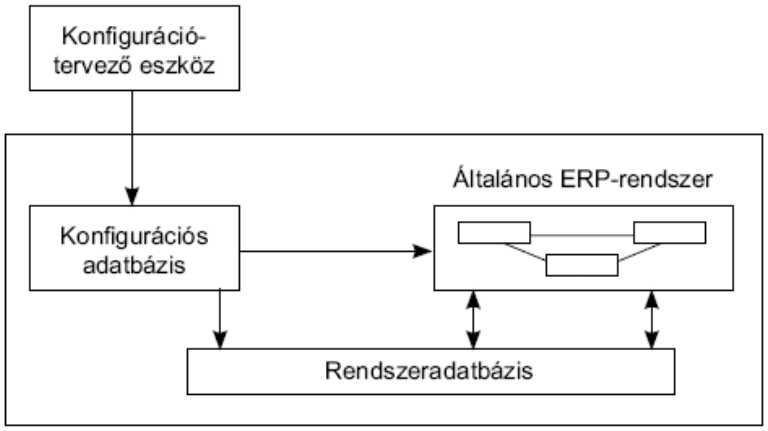
- Milyen funkcionalitásokat nyújt a rendszer?
- Hogyan tudom integrálni a saját rendszerembe a terméket?
- Milyen illesztők kellenek, vagy vannak meg?

A probléma ott van, hogy egy dobozos szoftver fejlődésének irányát nem tudom befolyásolni, illetve hogy a támogatások azok nagyon nyűgösek, pl.: megszűnik a cég, ami fejlesztette a rendszert. Lehet, hogy a támogatásért sokkal többet kell fizetni, mint a rendszerért. Ki kell próbálni, hogy ténylegesen alkalmas-e arra, amire kellene. Ha beépítek egy dobozos terméket a rendszerbe, akkor utána nehéz lecserélni.

Alkalmazás termékvonalak: Architektúra újrafelhasználását jelenti. Azonos architektúrájú, de specifikált alkalmazások. A specialitások lehetnek:

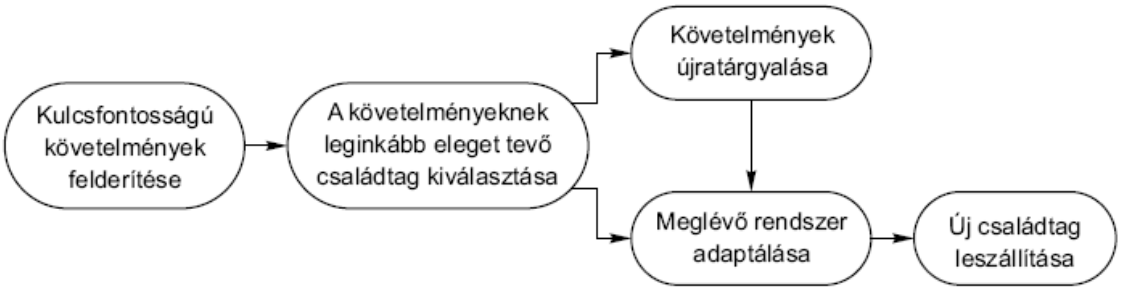
- Platform specializáció.
- Környezeti specializáció, vállalati környezet.
- Funkcionális specializáció, más-más környezetben a funkcióknak máshogy kell működniük.
- Folyamatspecializáció: máshogy mennek a folyamatok, pl.: közbeszerzés:Laptop beszerzésből szkenner.

Konfigurálhatóság: nem 10 termék van, hanem egy és azt lehet konfigurálni, és ekkor a konfigurációban van a specializálás. Az újrafelhasználás ott van, hogy az adott szakterületen milyen konfigurálási lehetőségeket kell biztosítani. Van tervezési idejű, és kihelyezési idejű konfigurálás.
A vállalatirányítási(ERP) rendszerek tipikusan ilyen konfigurációs rendszerek.



pl.: SAP: a vállalatot konfiguráljuk az SAP-hez.

Alkalmazáscsalád új tagjának kifejlesztése:



Komponens alapú fejlesztés: az objektum kicsi, ezért inkább komponens. Az alkalmazásokat komponensekből rakom össze, úgy hogy azokat összedugdosom(LEGO☺). A komponens az egy cserélhető valami, csak kompatibilisnek kell lenni.

CBSE – Component Based Softvare Engineering

90-es évek második felében jelenik meg az OO világ csúcsán, arra próbál meg választ adni, hogy az objektum túl kicsi, nem születnek meg az üzleti objektumok, ezek helyett üzleti komponensek.

Legfőbb elvei:

- a komponensek függetlenek, és absztrakt entitások abban az értelemben, hogy élesen elkülönül az interfész és az implementáció, és az interfészt látjuk, az implementációval nem foglalkozunk
- komponens szabványok alakulnak ki, amelyek meghatározzák, hogy hogyan kell kinéznie az interfészeknek, és ez alapján meghatározzák a komponensek kommunikációját.
- A komponensek elosztottak, lényeges a köztes réteg.
- CBSE a teljes fejlesztési folyamat.

Problémák:

- A komponenseket megcsinálják, kérdés, hogy megbízható-e a komponens. Adott esetben a dokumentáció alapján sem lehet eldönteni ezt, csak hogy ha leteszteltem a rendszeremmel.
- A komponenseket független cégek minősítsék, és tanúsítvánnyal lássák el, ez az alapötlet, de ez nem alakul ki, mert rengeteg gond van, pl.: hogy lehet egy komponenst tanúsítani.
- A komponensek eredendő tulajdonságait nem az interfész, hanem az implementáció határozza meg, ezek nem biztos, hogy dokumentáltak, és nem biztos, hogy kiderülnek.
- A komponenseket nekem kell alkalmaznom, kérdés az, hogy olyan komponensre van e szükségem.

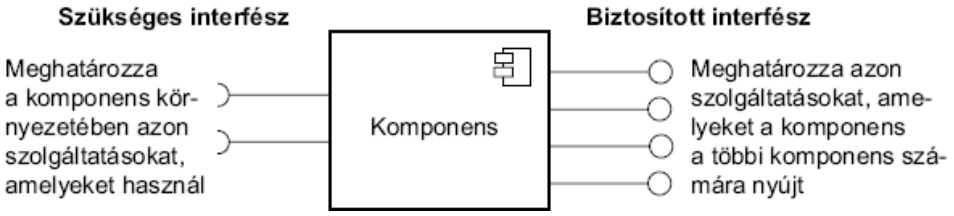
Egy-egy vállalat el kezd CBSE alapon fejleszteni, és aztán a saját komponenseit használja, tehát megmarad céges keretek között.

Komponens modellek:

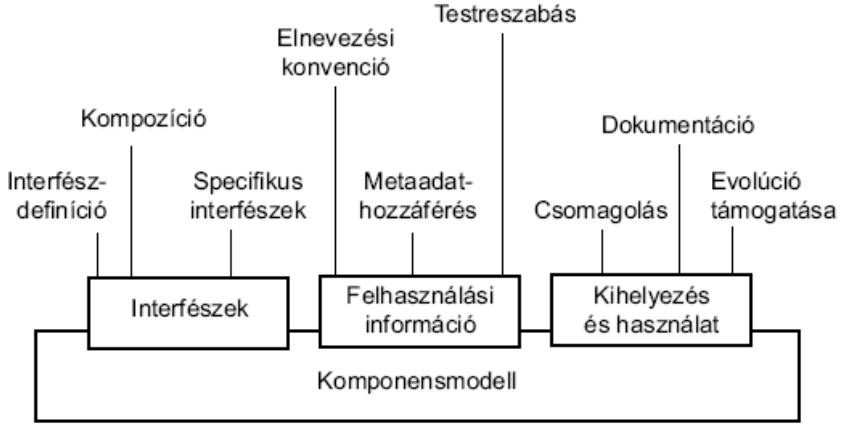
A komponens állapotmentes. Behozza, és általánossá teszi az állapotmentes szemléletet. Egy komponensből akárhányat hozok létre, azok megkülönbözhetetlenek.
A komponens definiálhatatlan (vizsgán nem is lesz):

Definíció:

A komponens egy szoftveralkotó elem, amely megfelel egy komponensmodellnek, önállóan kihelyezhető, módosítás nélkül beilleszthető az összeillesztési szabványnak megfelelően.
Szypasski-tól származik az egész:
Egy szoftverkomponens az összeillesztés egy egysége, amely csak szerződésszerűen meghatározott interfészekkel, és explicit kontextus függőségekkel rendelkezik. Önállóan kihelyezhető, és alkalmas harmadik fél által történő összeillesztésre.
Lényege: követ egy komponensmodellt, önálló, összeilleszthető, kihelyezhető.
Kihelyezhetőség: Önmagában zárt dolog, akárhányszor kihelyezve ugyanaz marad. Szemben az objektummal a komponenst nem fordítom bel e a kódba.

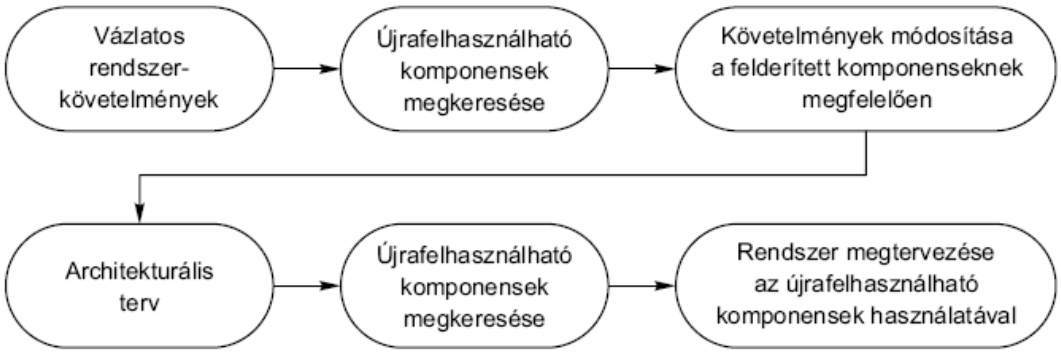


Szolgáltatásokat nyújt meg használ, ezekhez két interfész van, a szükséges és a biztosított interfész.
Az objektum mindig példányként jelenik meg, a komponensnek nincs típusfogalma.
OMG: CORBA, A Microsoft COM,COM+,DCOM, Java: EJB, IDL – Interface Definition Language
A komponens soha nem kezel kivételt. A kivétel része az interfésznek, és specifikálni kell.

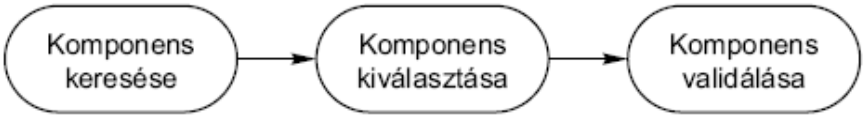


A köztes réteggel való együttműködéstől eltekintve a teljes funkcionalitást kell tudnia nyújtani, de egyébként be van csomagolva, tehát fekete doboz, és csak az interfészek vannak.

Megszületnek komponensek, és megszületnek komponenstárolók, és itt érhetőek el a komponensek.
Kialakulnak speciális komponensek, és kialakulnak vállalati tárolók, de nem igazán alakulnak ki szakterületi komponenstárolók. Újrafelhasználható komponenseket próbálunk gyártani, specifikáljuk az interfészt, függetlenné és zárttá tesszük.
A baj az, hogy ellentmond egymásnak az újrafelhasználhatóság, és az alkalmazhatóság, mert annál jobban újrafelhasználható, minél jobban általános, és annál kevésbé alkalmazható.
A komponens egyfajta eszköz arra, hogy az ősrendszerek kezelését megoldjam. Tehát ha nem akarom kidobni, hanem használni akarom a funkcionalitását, akkor becsomagolom egy komponensbe, felé specifikálom az interfészt, és innentől használható.



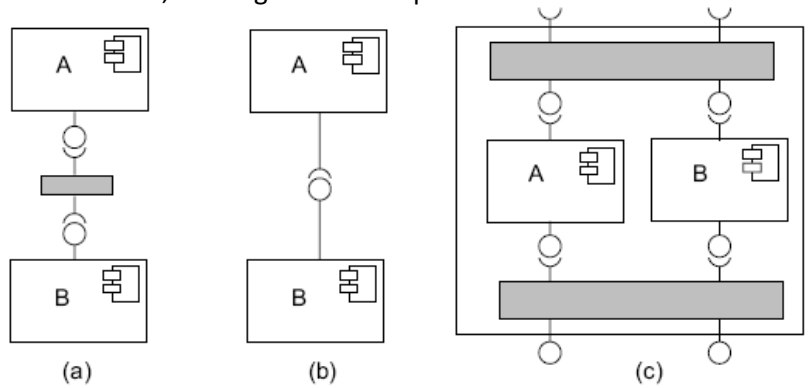
A komponens modellek vagy nem mondanak sok mindent, vagy nem beszélnek arról, hogy a komponenst a tárolóban hogy kell elhelyezni. A validálás külön kérdés, mert csak tesztelni tudom, a kódba nem látok.

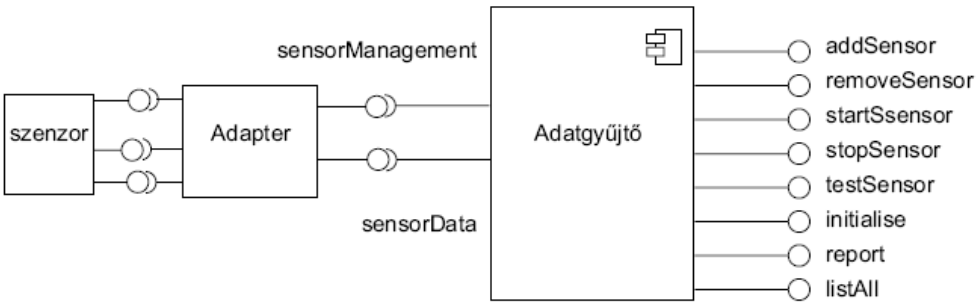


Komponens kompozíció

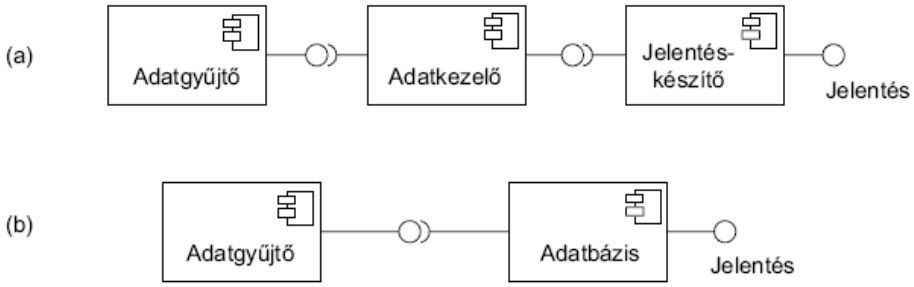
A komponensek összeillesztésének folyamata egy rendszer létrehozása érdekében. Ha rendelkezésre állnak újrafelhasználható komponensek, akkor a legtöbb rendser létrehozható lenne, az újrafelhasználható komponensek egymással, a speciálisan elkészített komponensekkel, és a modell keretrendszer által nyújtott komponenstámogató infrastruktúrával történő kompozícióval.
3 féle közelítést különböztetnek meg a modellek:

1. Szekvenciális kompozíció: Az összetett komponensben az alkotó komponensek végrehajtása szekvenciálisan történik. (a) Az egyes komponensek biztosított interfészei kerülnek összefogásra. A komponensek közti kapcsolat megteremtéséhez plusz kódra van szükség.
2. Hierarchikus kompozíció: Egy komponens közvetlenül meghívja a másik komponens által nyújtott szolgáltatásokat. (b) Az egyik komponens biztosított interfésze egy másik komponens szükséges interfészével kerül összekapcsolásra.
3. Additív kompozíció: Két vagy több komponens egy új komponens létrehozásának érdekében kerül összeépítésre. A kompozit komponens interfészei az alkotó komponensek interfészeinek összefogásával állnak elő, szükség esetén a duplikált interfészek eltávolításával (c)



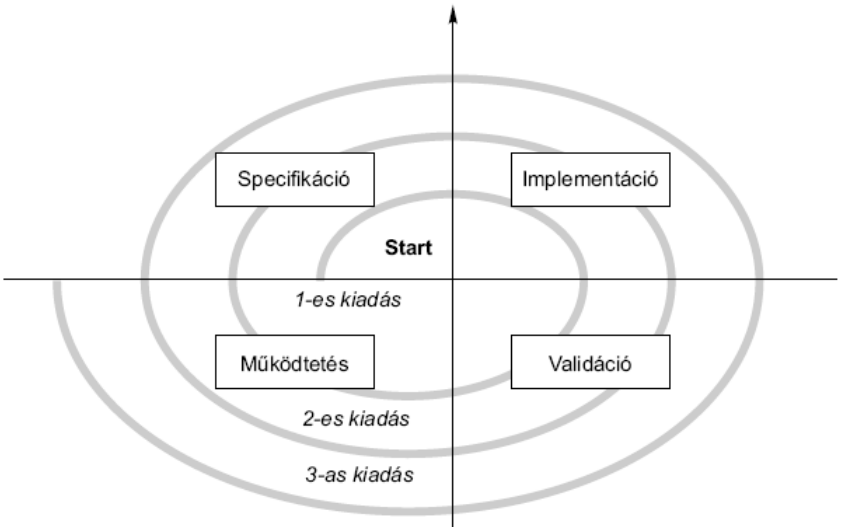


A kompozíciók különböző előnyökkel és hátrányokkal rendelkeznek.

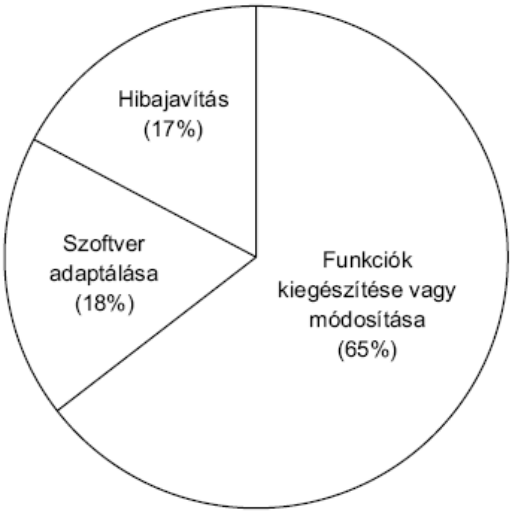


EVOLÚCIÓ

Az evolúció minden esetben változtatást jelent. Hosszú életű nagy rendszerek esetén az evolúció válik alapvetővé a nagy fejlesztési folyamatban.



Minden lépést ugyanaz a cég fejleszt, a teljes életciklus egy helyen van. Egyszerre több spirál megy, kettő vagy három, tehát egy kiadásakor már egy másik vagy harmadik is mehet.



Szoftverkarbantartásról akkor beszélünk, amikor valaki kifejleszt egy rendszert, letelepíti és átadja a működtetőnek. Ez a rendszer is változik, a karbantartást végezheti a fejlesztő is de soha nem ebben az értelemben.

A Lehmann törvények(Bélády is): nagy rendszerek evolúciója:

- Folyamatos változás: Egy ténylegesen használt nagy rendszer szükségszerűen változik, különben egy idő után használhatatlan lesz. A rendszer változásával a rendszer mindig egyre komplexebb lesz. Nagyobb infrastruktúra kell, adott esetben refaktorálni kell a rendszert.
- Önszabályozó folyamat: A rendszer mérete, az egyes kiadások közt eltelt idő, és az egyes kiadásokban felfedezett hibák száma az invariáns.
- A rendszer fejlődése viszonylag konstanssá válik, függetlenül attól, hogy mekkora erőforrásokat fordítunk a rendszerre.
- A rendszer élettartama alatt a változások mértéke az egyes kiadások között konstans.

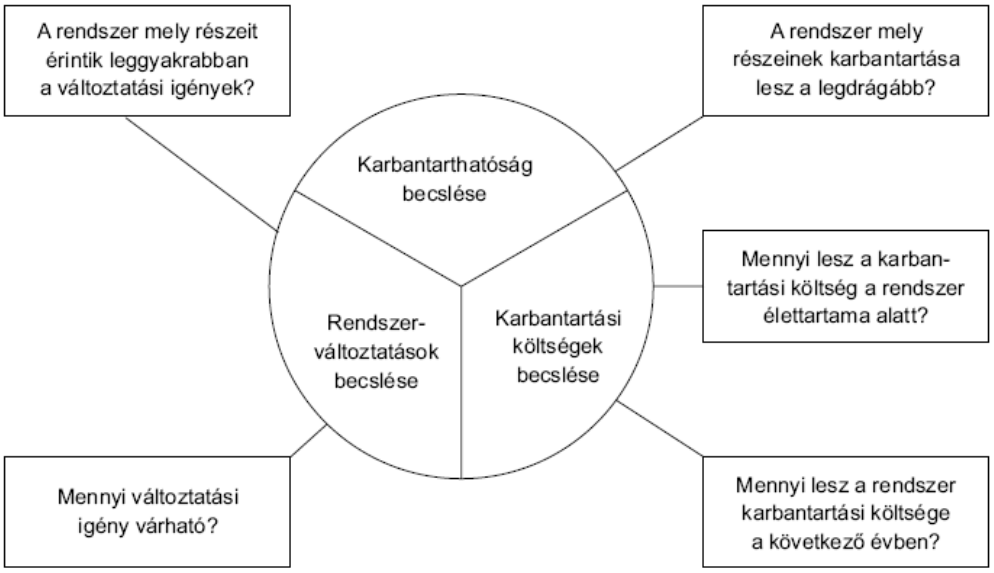
Szoftverkarbantartás:

A szoftver sohasem egész és hibátlan. Új funkciók beiktatása is idetartozik, mert változnak a követelmények, a jogszabályok.
Ezek az arányok az elmúlt évtizedekben nagyjából állandóak maradtak. A kérdés a karbantartásnál az, hogy milyen költséggel jár ez, illetve milyen tényezők befolyásolják a költséget.

Tényezők:

- A karbantartás elválik a fejlesztéstől, és a karbantartást nem az végzi feltétlenül, aki a rendszert fejleszti. Így meg kell ismernie a rendszert, mielőtt nekilát, ez költséges.
- Általános, hogy szándékosan leválasztják a fejlesztésről a karbantartást, és az külön szerződéssel történik, még akkor is, ha a fejlesztő végzi a karbantartást. A karbantartás jóval kevésbé körvonalazható szerződésben, mint a fejlesztés, kivéve, ha eleve tudják, hogy milyen hibákat kell majd kijavítani☺.
- Ahogy a program öregszik, úgy korrodálódik. Romlik a struktúra, architektúra, pláne ha már új funkció hozzá lett adva, vagy karban lett tartva, másrészt, ha nem nyúlunk hozzá, akkor értéktelenné válik egy idő után. Refaktorálni kell a rendszert.

Az üzemeltető cégnek becsülni kell a karbantartást, ez idő meg pénz meg ember. A változtatások számát kellene megbecsülni:

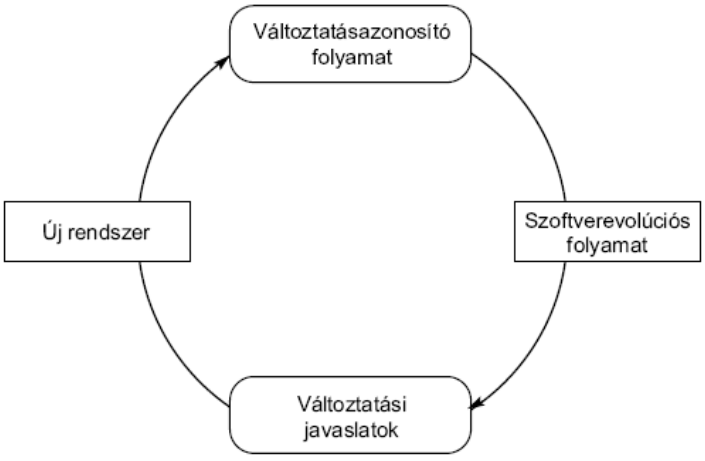


A becsléshez felhasználhatóak a komplexitási metrikát, a karbantartási metrikát. A rendszer egyes elemeinek költségbecslésénél tudjuk használni a komplexitási metrikákat.

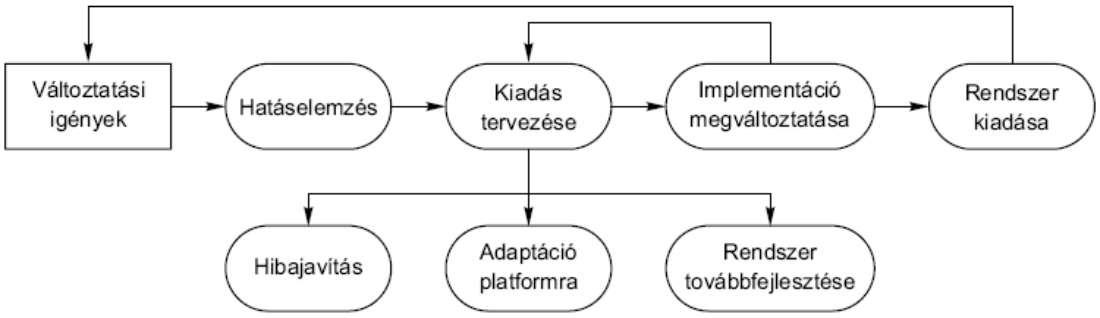
Tudunk használni folyamat metrikákat:

- Hibajavításból származó karbantartási kérések számát
- Hatásanalízis elvégzéséhez szükséges átlagidő: van egy adott rendszerelem, amelyet módosítani kell, akkor meg kell nézni azt, hogy a rendszer más elemeire ez milyen hatással lesz. Általában nem igaz, hogy egy elem önmagában módosítható és nincs kihatása. Ha ez nagy, vagy nő, akkor egyre bonyolultabb, drágább és időigényesebb a karbantartás.
- Hány olyan változtatási kérés van, amit nem sikerült realizálni. Ha ez nő, akkor problémás a karbantartás.
- Egy változtatási kérés implementálásának az átlagideje, azaz mennyi idő alatt sikerül realizálni.

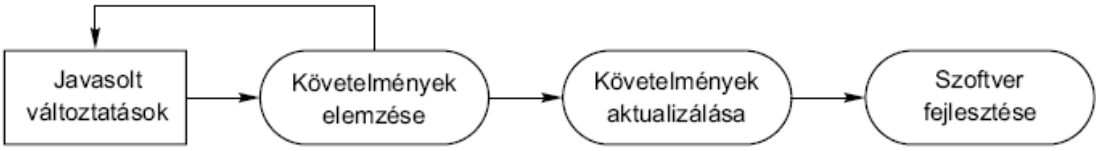
Változtatások kezelésének folyamata: ciklikus, és nem kör☺



A változtatási folyamat:

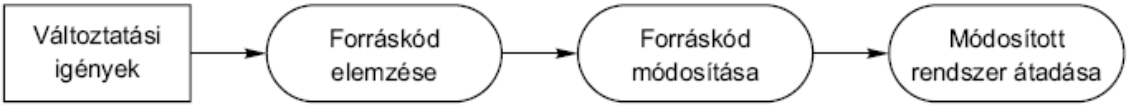


Hatáselemzésben benne van az is, hogy a változtatási kérelmet elutasítják, mert drága, vagy nincs ember, vagy kiszámíthatatlanná válna a rendszer általa.
Ha elfogadják, akkor mehet a tervezés, majd esetleg adoptáció, implementáció, verifikálás, majd megjelenés, és aztán kezdődik elölről.
A változtatás általában nem egy változtatás, hanem jóval több változtatási kérelem együtt.

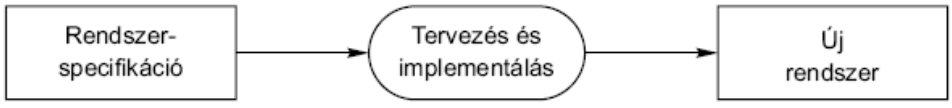


Lehet, hogy a követelmények változnak, de lehet hogy a változtatások igazodnak a követelményekhez. Jöhetnek a kulcsfigurától, felhasználóktól, informatikusoktól stb. A hatáselemzés a legkeményebb fázis, utána bizottság dönt, hogy melyiket viszik végig, és hogy mikor ütemezik be.
Sürgősségi változtatások, amit végre kell hajtani, mert különben a rendszer működése kerül veszélybe, pl.:

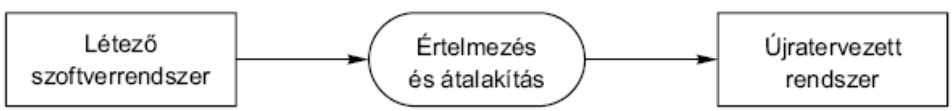
- Rendszerhiba keletkezik, ami az egész rendszert veszélyezteti.
- Üzleti igények változtak meg, különben pozíciót veszít a cég.
- Platformon történő változások.



Ha a rendszer elég sokáig működik, akkor ősrendszer lesz belőle, elég sokszor volt karbantartva, elavult technológiailag, és esetleg leromlott az architektúrája, struktúrája.
Ugyanakkor egyre jobban validált, a rendszer funkcionalitása egyre jobb, viszont egyre nehezebben, és költségesebben tartható karban. Ekkor felmerül az egyik dolog, hogy tervezzük újra a rendszert.
Előre tervezés: követelmények, tervezés, implementáció. (Forward Engineering)
Újratervezés: van egy ősrendszerem, aminek leromlott az architektúrája, akkor újratervezem a rendszert. ez a ReEngineering.
Visszatervezés: Van egy rendszer, terve meg sose volt, meg doksi, és akkor a kódból csinálok dokumentációt, ez Reverse Engineering

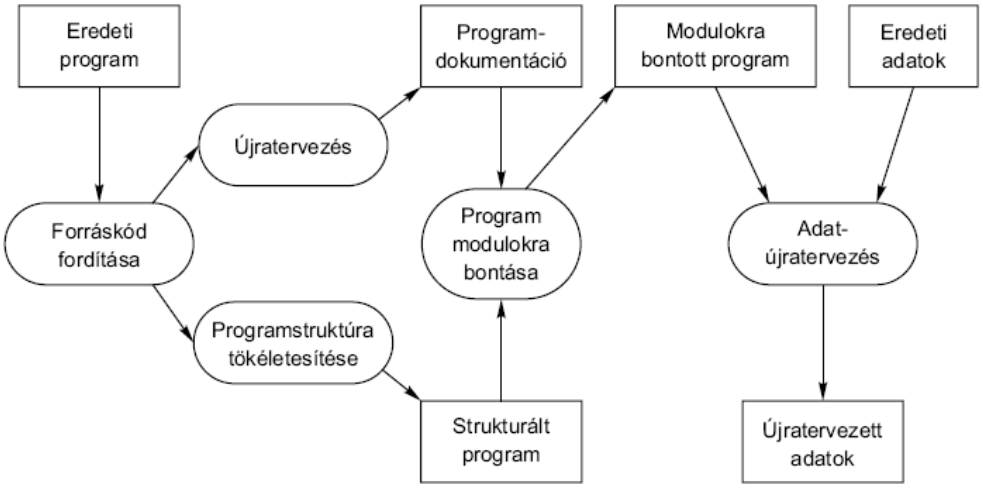


Előretervezés

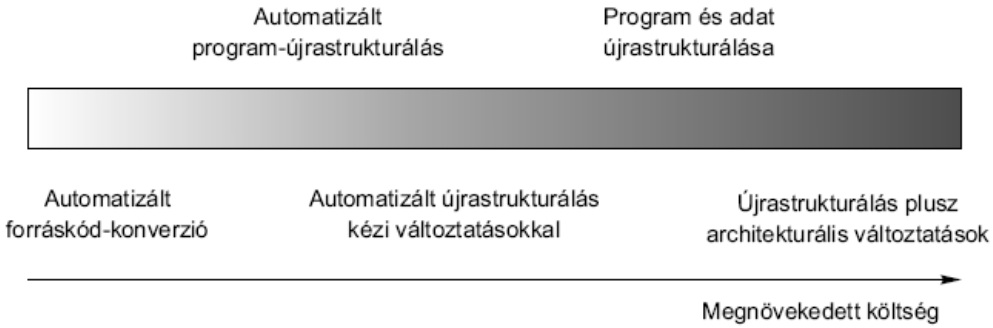


Újratervezés

Mindegyikhez megvannak az automatizált eszközök.
Az újratervezésnél a rendszert megtartom, csak újraimplementálom, ez egyfajta adoptációs változtatásnak tekinthető. Jóval kisebb a kockázat, mint ha újraírnám a rendszert. Ez működik, és validáltan működik, és a költség is kisebb, ha megfelelő szakemberek tervezik újra a rendszert.



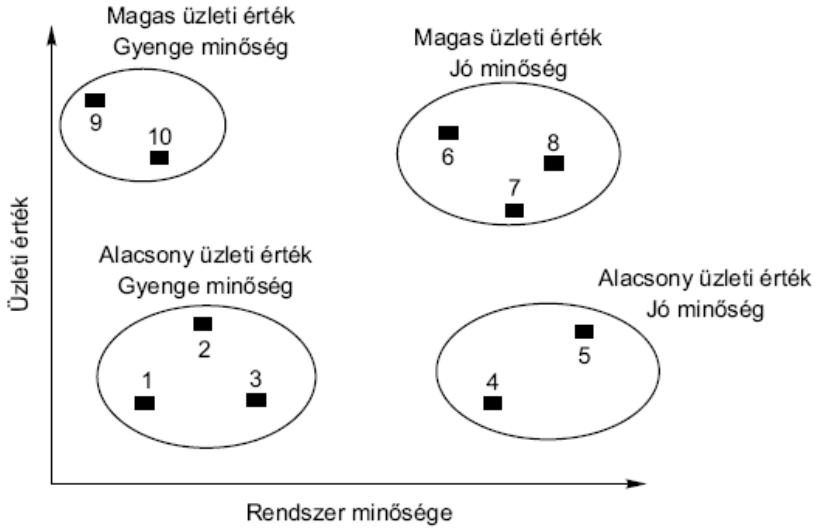
Megvan a forrásszöveg, akkor az adott nyelv jelen fordítójának érthetővé kell tenni, majd jön egy refaktorálás, és utána jön az újratervezés általában a forráskód alapján. Az adatok struktúráját is újra kell tervezni valószínűleg. A végén kialalkul egy javított struktúrájú, de változatlan funkcionalitású rendszer.



Az egész becslése a COCOMO2.

Az ősrendszerekkel kapcsolatban egy cég 4 féle döntést hozhat:

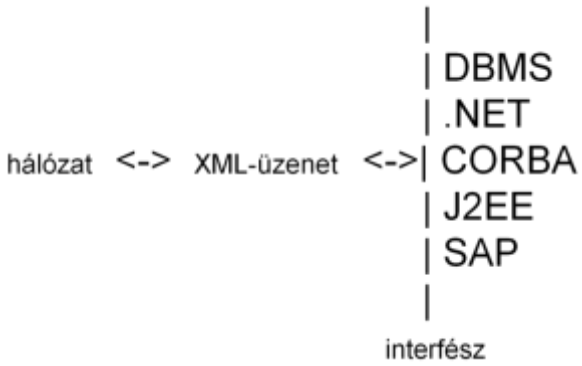
1. Kidobják
2. Változatlan formában fenntartják
3. Megváltoztatott formában továbbüzemeltetjük, például újratervezzük, vagy komponenst csinálunk belőle, vagy szolgáltatást.
4. Lecseréljük egy új rendszerre.



1,2,3:Kidobni
4,5: Hagyni úgy ahogy van
6,7,8: Továbbvinni, mai technológiával
9,10: kidobjuk vagy újraírjuk vagy becsomagoljuk

WEBSZOLGÁLTATÁSOK:

XML-alkalmazásokat képeznek le programokra, objektumokra, adatbázisokra, üzleti függvényekre.



Van egy XML dokumentum, ezt küldözgetem a weben, és a szolgáltatások között ott van valami, valami adatbázis kezelő, és valamilyen technológia, ami a szolgáltatásokat biztosítja, egy jól meghatározott interfészen keresztül. Elküldök egy XML-t és XML jön vissza. Szabványok definiálják az üzenetek formáját, az interfészt, a leképezést, hogy a szolgáltatást hogy kell publikálni, megtalálni és használni.

Kétféle megoldás van:

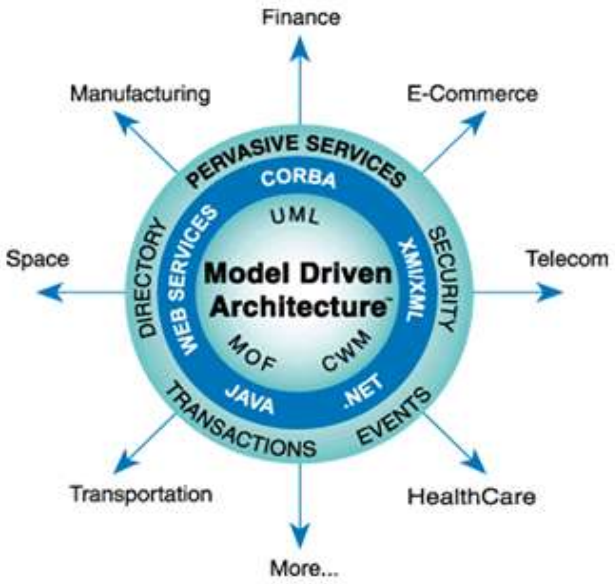
- RPC – online megoldás, egyetlen dokumentumot küldünk, amit egyből le lehet képezni és már jön is a válasz
- Batch – aszinkron megoldás, dokumentum orientált megoldás, az XML dokumentum leképezése egy jóval bonyolultabb dolog, elküldöm és nem várom meg a választ.

W3C Szabványok:

- XML interakciós minták, és protokollok definiálását szolgálja
- WSDL – Web Service Description Language: szolgáltatás leíró nyelv, interfészek interakciós minták, és leképezésprotokollok definiálását szolgálja.
- UDDI – Universal Description, Discovery and Integration : XML alapú technológiai gyűjtemény, amely a webszolgáltatások kommunikációját írják le. Hogy lehet megtalálni, és felhasználni.
- SOAP – Simple Object Access Protocol : A webszolgáltatások publikációját, tárolását elérését írja le.

MODEL DRIVEN ARCHITECTURE

Az MDA lényege, hogy az újrafelhasználást felviszi az elemzés elé, legalábbis próbálja. Felviszi a kódgenerálást architektúrális tervezési szintre, tervből kód.



CWM mai legkorszerűbb adatbázis szabványnak tekinthető.

Három modellt definiál:

CIM – Computation Independent Modell
PIM – Platform Independent Model
PSM – Platform Specific Model
itt van még a PLATFORM MODELL
CÍM: szakterületi absztrakt modell, lényegében független az informatikától, egy szakterületi fogalomszótár.

Két almodellje:

- vállalati
- információs

Hidat képez a követelményeket megadó kulcsfigurák, és a követelmények feltárását végző informatikusok között.
PIM: absztrakt modell, implementáció független elemeket tartalmaz. A kifejlesztendő rendszert abból a szempontból írja le, hogy hogyan segíti a modell az üzleti életet. Technológia független virtuális gépnek tekintik, amit majd implementálni kell különböző technológiákban.

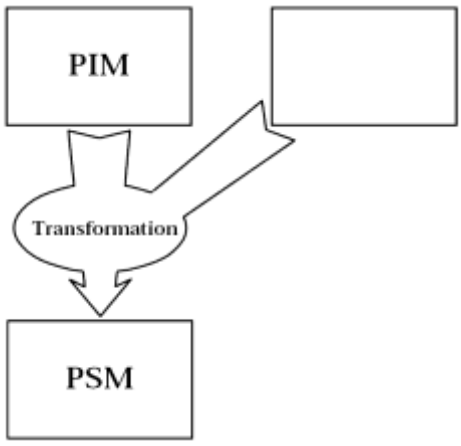
Almodellje:

- számítási modell

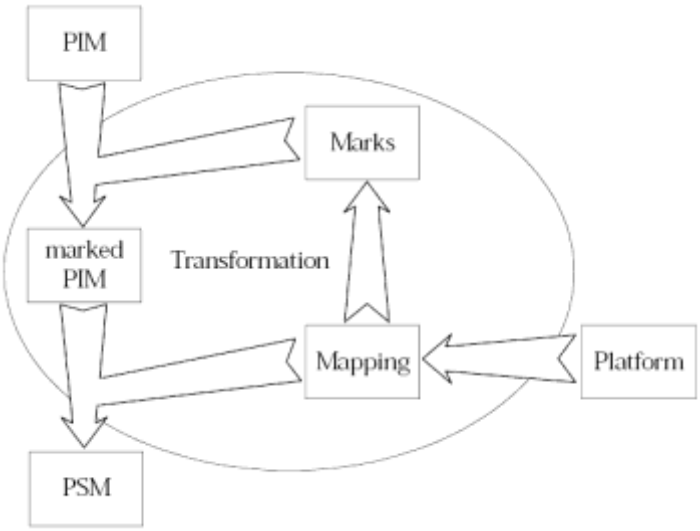
PSM: az elérhető implementációs technológiákon alapszik. Az adott technológia elemeit, eszközeit, fogalmait használja. Egy rendszer vonatkozásában egy CIM és egy PIM létezik, de akárhány PSM. Ezeket kell megadni, formalizálni.
PLATFORM MODELL egy konkrét platformhoz kötődik, a PIM-ből a PSM-et ezen információk segítségével lehet előállítani, konkrét technikai információk. A kód generálása ebből automatikusan történik.

MDA metamodell közelítés felé hajlik, de nem zárja ki a metanyelvek használatát sem. Kérdés az hogy hogy lesz a PIM-ből PSM, és hogy a PSM-ek átjárhatók e. PSM-ek között bridge-ek vannak. Vertikálisan és horizontálisan a leképezéseket automatizálja a szabvány.

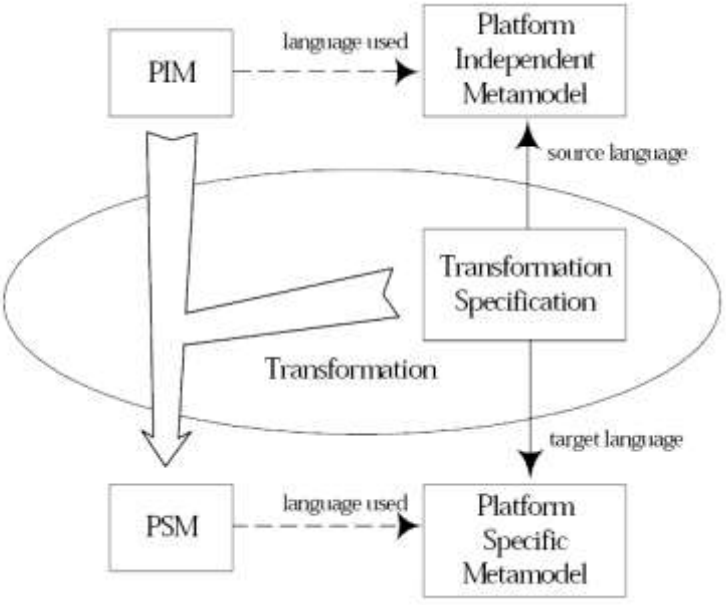
Az MDA az implementációs, tervezési folyamatra nem mond semmit. Bekerül az összes nagy tervezési folyamatba. RUP, XP stb…



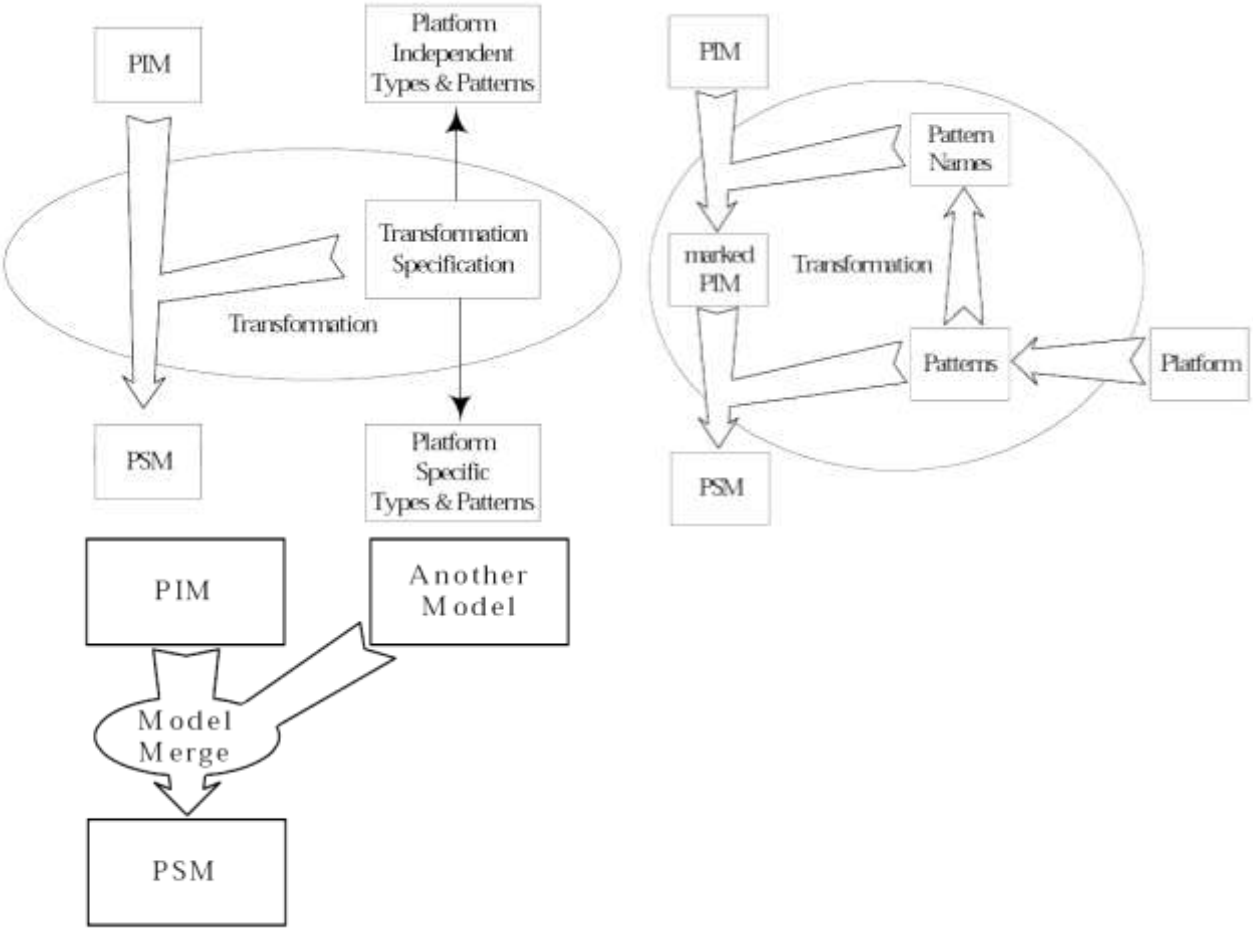
PIM leíró nyelvét transzformáljuk a PSM leíró nyelvére



Lényege: Modell elemeket képez le modell elemekre. Egy PIM több PSM-re leképezhető, és van egy megjelölt PIM modell. A PIM modell elemeihez szerepköröket rendel, és ezek alapján mehet az automatikus transzformáció.



Leírjuk a PIM és a PSM metamodelljét, és a PIM-ből képezzük a PSM-et. Ennek a speciális esete az, hogy megvan a metamodell, PIM és PSM példánynak tekinthető, és akkor a példányokat képezzük le.



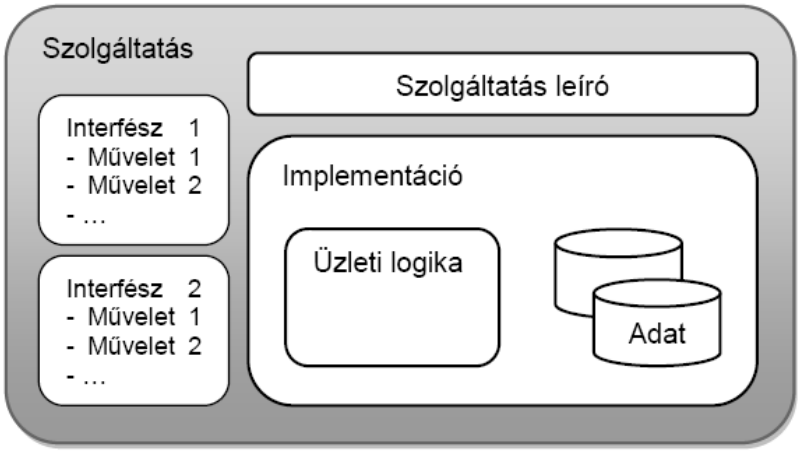
A konkrét technológiák ezek valamelyikét építik be. Egyes technológiák nem képezik le, hanem a PIM-ből automatikusan kódot generálnak. Absztrakt modellből automatikus kódgenerálás.

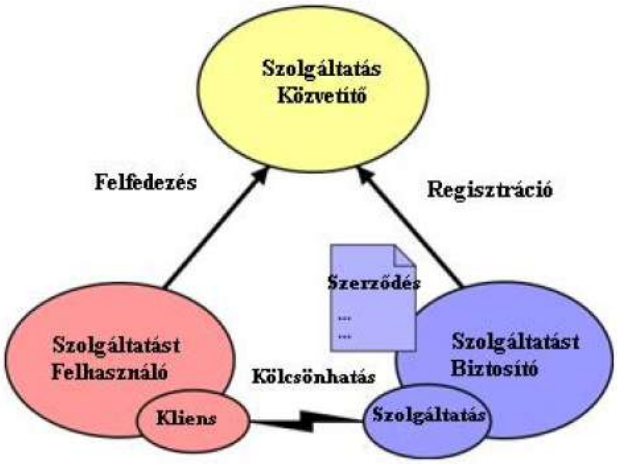
Az MDA előnye az, hogy az újrafelhasználás nagyon magas szintű, a gond az, hogy a generált kód lényegesen nem jobb, mint amikor az első FORTRAN kódot generálták, annyival jobb, hogy nem egy nyelvhez kötődik. A minőség ott kezd gond lenni, hogy ha egy teljes rendszert akarok generálni.

objektum->komponens->szolgáltatás
objektum: kódújrafelhasználás
komponens: funkció újrafelhasználás
szolgáltatás: a mai értelemben véve az folyamat újrafelhasználást jelent.

SOA: szolgáltatás orientált architktúra.

Egyesek szerint nem architektúra. Nem informatikai dolog legelső szinten. A szolgáltatás egy absztrakt entitás. A Webszolgáltatás szolgáltatás, de a szolgáltatás az nem webszolgáltatás.



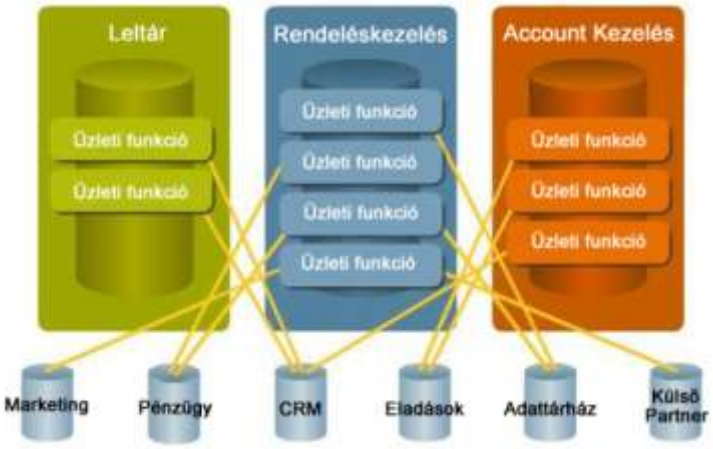


A szolgáltatás orientált megoldások. A szolgáltatások lazán kapcsolt szerkezetben működenk.

Három szerepkör van:

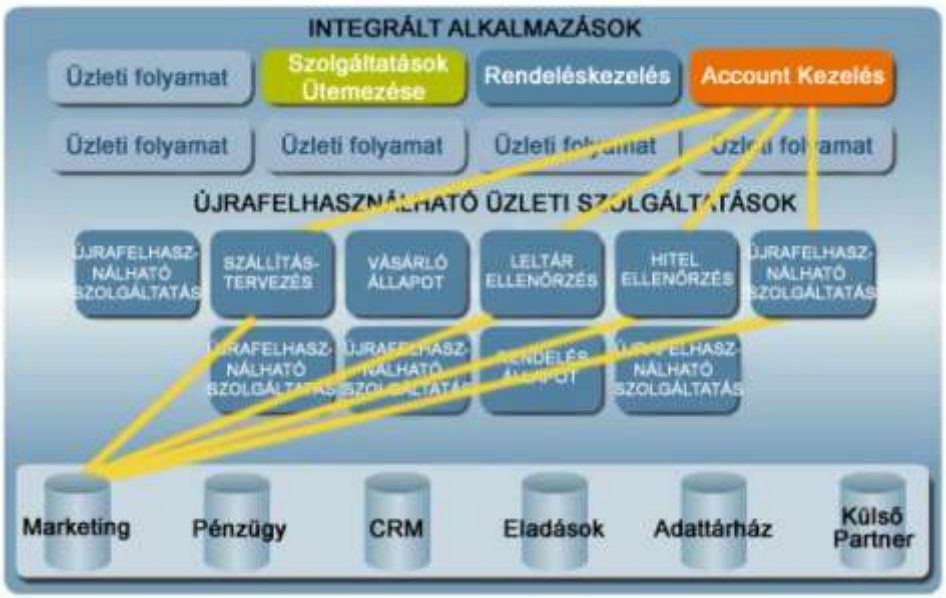
1. szolgáltató
2. igénybe vevő
3. közvetítő: rajta keresztül történik a szolgáltatás, a regisztrálás, felfedezés is.

A szolgáltatást egy szerződés alapján lehet igénybe venni és implementáció nem látszik.



Vállalat informatikai politikáját dönti el. Egy vállalaton belül folyamatok vannak, ezekhez üzleti logika kötődik, a folyamatokat mindenféle informatikai alkalmazások segítik. Az ábrán egy SOA előtti struktúra van.

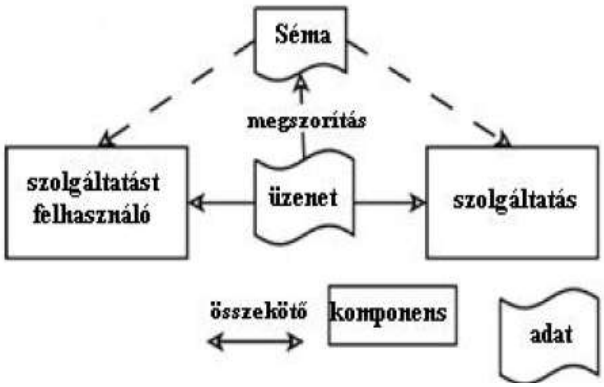
A SOA egyféle rendszerintegrációs megoldást nyújt. „Mindent soásítunk”.



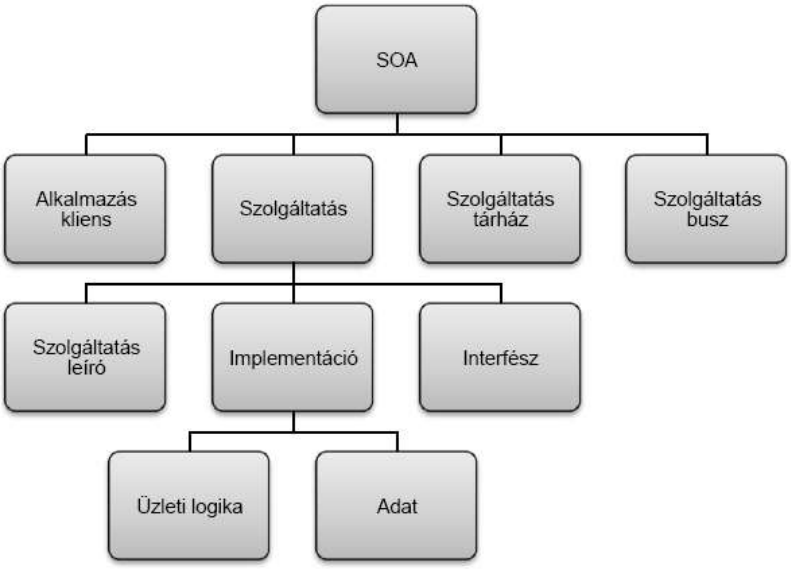
Folyamatok vannak, a folyamatokra úgy képzeljük el, hogy szolgáltatásokat fűzünk fel, és ezeket minden eddigi részrendszer igénybe tudja venni, vagy minden részrendszerből lehet szolgáltatás, amit igénybe lehet venni. A rendszert úgy fejlesztem tovább, hogy szolgáltatásokat definiálok.

SOA-sítás:

1. Megmondja, hogy mik a folyamatai
2. eldönti, hogy mi mögé akar szolgáltatást tenni vagy sem
3. jöhet az informatika, mögé teszek egy szolgáltatást.

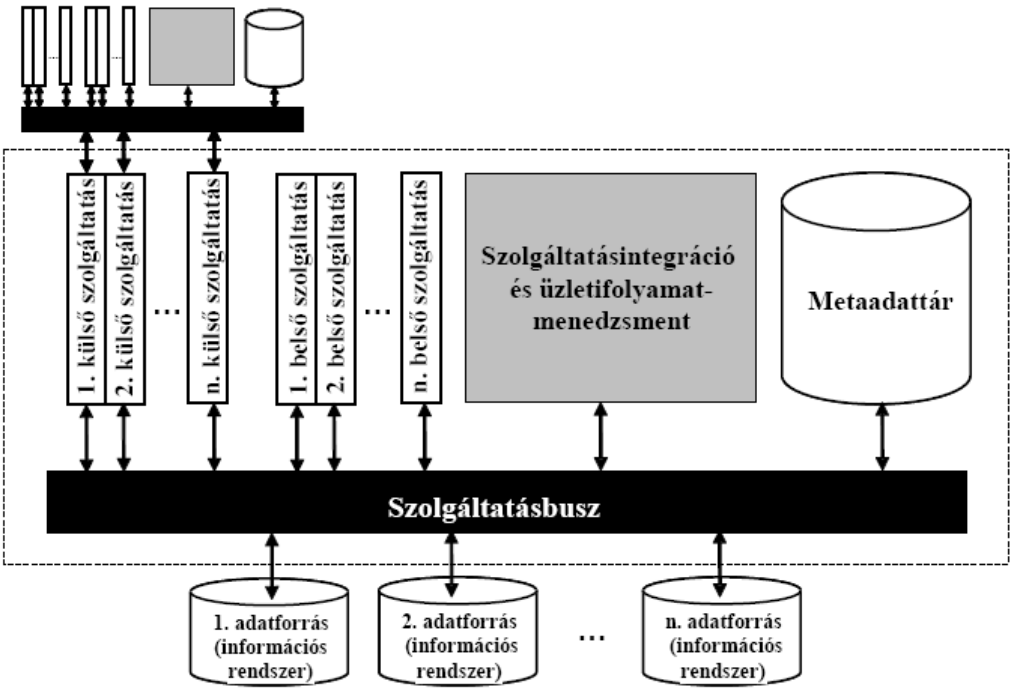


Másikfajta megközelítés



SOA négy alapvető komponense:

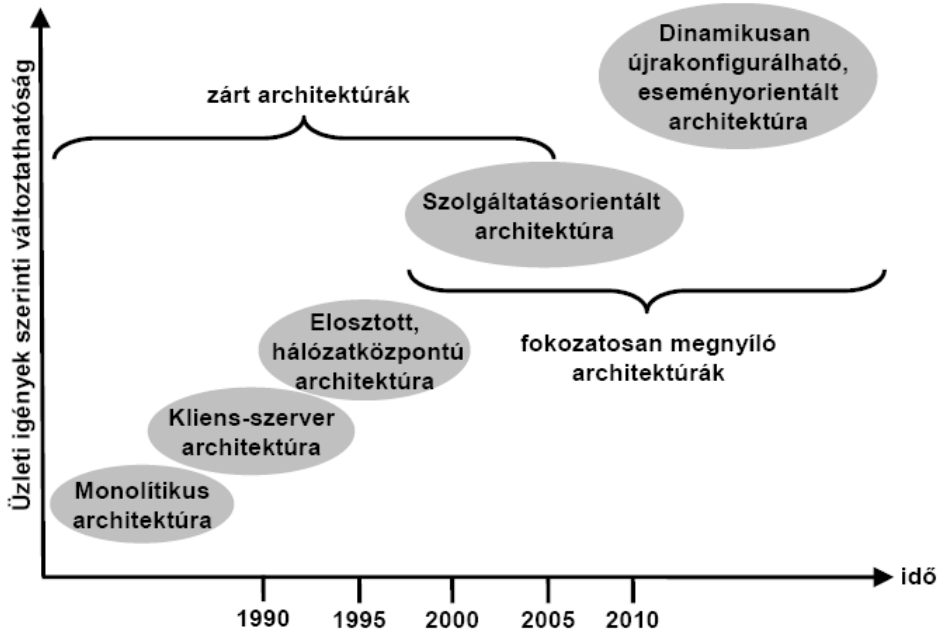
A szolgáltatásokat a szolgáltatás busz közvetíti. Az architektúra erre vonatkozik.



Ezt a szolgáltatás buszt kell nagyon jól definiálni. Ma minden magára valamit adó cégnek megvan az ajánlata, hogy hogyan lehet SOA-t építeni:

1. Folyamat leíró és feltáró rendszer
2. Szolgáltatás kezelő eszközrendszer, interfész és szolgáltatás definiálására
3. Szolgáltatás busz megvalósítás

A szolgáltatás határai mindig explicitek, tehát nagyon jól definiált, hogy mettől meddig tart, és mi tartozik a hatáskörébe.
A szolgáltatások önálló entitások, autonóm módon működnek, a külvilág számára absztrakt entitások.
A szolgáltatások viszonyát egy dolog szabályozza, és ez a szerződés.
Kis és nagy cégek NEM használnak SOA-t. Nagy cégeknél nem deríthetők fel annyira a folyamatok hogy megérje, mert ezek az eszközök istentelenül drágák. Kis cégeknél csak az ár miatt, meg túl kicsi a folyamat.
53%-a önmagától SOA-sított. 25% még nem de akar, 22% meg nem is akar. A Mobil világ az nem SOA-sít.Aki komplett lefedi az egész vállalati infrastruktúrához a tetőtől a legaljáig: MICROSOFT és az ORACLE



SOP: szolgáltatás orientált programozás.
Grid: egyfajta SOA megoldást ad.

WOA
szolgáltatás után -> ontológia:szakterületi koncepció újrafelhasználásával tudunk kódot generálni. Már van rá szabvány.

„Things should be made as simple as possible, but no simpler”
Einstein
„Nature wasn’t designed but debugged into perfection”