Tartalomjegyzék

Vizsgatémakörök
a „Szoftver Tervezés és Technológia” c. tárgyból

1. A szoftverkrízis (elozmények, tünetek, okok, megoldások)

1.1. Elozmények:

A mai fogalmak szerint gyenge hardver. Gyenge fejlesztoi szoftver ellátottság.
Monolitikus programozás. Nehezen becsülheto elore a szükséges eroforrás mennyisége. Nehezen határidozheto a feladat. Körülményes a team-munka, nem teljes köru a tesztelés (triviális hibák még futáskor is kibukhatnak (interpreterek)). Nagyon nehézkes a program módosítása, bovítése. Személyiség függo programok.

1.2. A tünetei:

Megbízhatatlanok a szoftverek (nem tudja speckót, nem szuri az inputot, nem robosztus). Nehezen alkalmazkodnak a konfiguráció változásához.
A fejlesztés nehézkes (részprogramok összeállítása nehézkes, körülményes a tesztelés, nem hatékony a team-munka).
Nem informatívak a szoftverek (nincs korrekt hibajelzés, merev input oldal, stb.).

1.3. Az okai

A minoségi követelmények változása. Futási ido minimalizálás. Tárigény minimalizálás. Felhasználóbarát felület. Feltétlen megbízhatóság. Könnyu karbantarthatóság. Könnyu továbbfejleszthetoség. Gyors, olcsó kivitelezhetoség. Határidok pontos betartása. Egyéniség független fejlesztés.
A szoftverek méretének növekedése maga után vonta a komplexitás növekedését (min. négyzetes az összefüggés). A fejlesztési módszerek nem tartottak lépést a változással. A felhasználói környezet változása (a felhasználók száma, felkészültsége, az alkalmazási körülmények) gyökeresen megváltoztak.

1.4. Megoldások:

A szoftverkészítés technologizálása. Új elvek, módszerek és eszközök kifejlesztése (CASE). Szoftverszabványok bevezetése, szoftverminoségbiztosítás. Új programozási paradigmák alkalmazása:

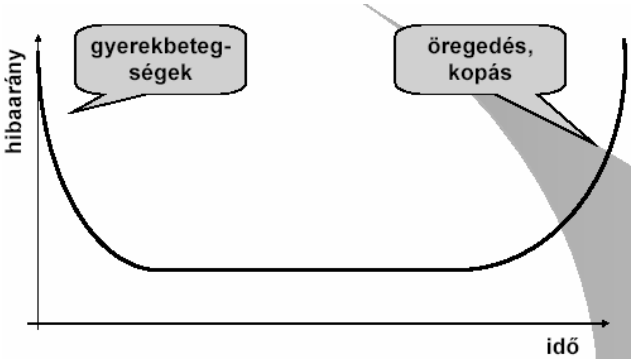
- Moduláris programozás („Divide et Impera”)
- Strukturált programozás (SP, SD, SA)
- Objektum-Orientált programozás (OOP, OOD, OOA,OOAD)

2. A szoftver jellemzoi (a termék eloállítás jellemzoi, hiba-aránygörbék, költségösszetevok, költségarányok)

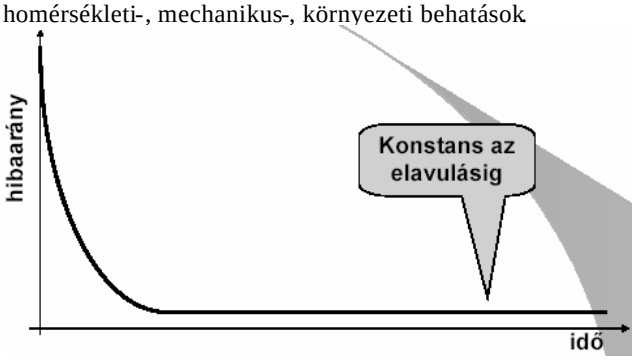
2.1. A termék eloállítás jellemzoi

A hagyományos termékek készítésének folyamata: Analizs → Vázlatos tervezés → Részletes tervezés → Fizikai megvalósítás.
A szoftver termék esetében eltéro a megvalósítás fázisa. Nincs „átültetés” az anyagba. Nem kell figyelembe venni a tervezés során, az anyagjellemzoket, a megmunkálási módok jellemzoit. A szoftver „nem kopik el”.

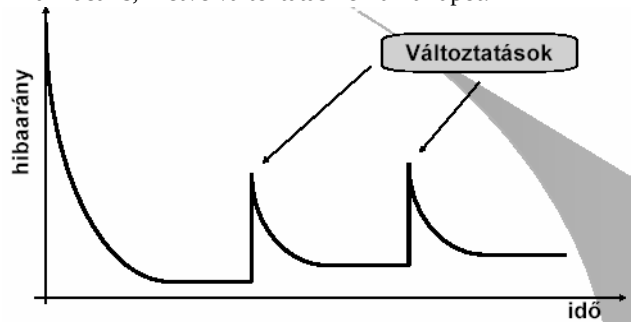
2.2. Hiba-aránygörbék



A gyerekbetegségek okai: tervezési-, gyártási-, szerelési hibák hibás anyagok, IC-k, stb. Az elkopás okai: öregedés, homérsékleti-, mechanikus-, környezeti behatások.



Ez az ideális, illetve változtatás nélküli állapot.



A változtatások újabb hibákat eredményeznek, melyek a termékben összegzodnek.

2.3. Költségösszetevok, költség arányok

Követés költségei **Követés nélküli költségek**

3. A szoftverfolyamat modelljei I. (modelltípusok, a modellezési koncepció, célok, a vízesés modell)

3.1. Modelltípusok

- Munkafolyam modell (tevékenységek sorrendje, bemenetei, kimenetei, függoségei)
- Az adatfolyam vagy tevékenység modell (mindegyik tevékenység valamilyen adattranszfomációt hajt végre. Azt reprezentálja, hogyan alakul át a folyamat bemenete kimenetté az emberek, vagy számítógépek által végrehajtott tevékenységek során).
- A szerepkör /cselekvés modell (a szoftverfolyamatban résztvevo emberek szerepköreit és felelosségüket ábrázolja.)

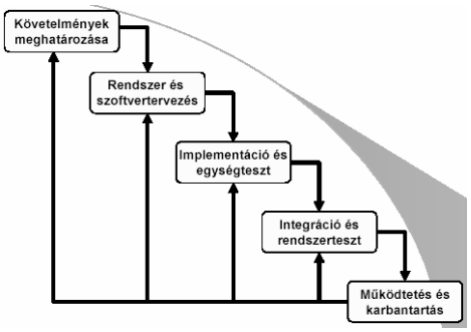
3.2. A modellezési koncepció

A rendszer megértést segíto absztrakció, elhanyagolja a lényegtelen részleteket, a modell kizárólag az eredeti rendszer lényegi elemeit tartalmazza. (LÉNYEGI ELEM - lényeges a vizsgálat szempontjából)

3.3. Célok

A komplexitás csökkentése. Kommunikáció az ügyféllel. A rendszer vizsgálata megépítése elott, vizualizálás

3.4. A vízesés modell



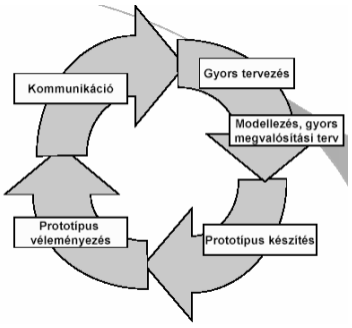
Jelemzok: szisztematikus, szekvenciális, top-down modell. A hagyományos mérnöki szemléletet követi, a leginkább elterjedt (legrégibb) modell

Problémák: a valós projektek ritkán követnek szekvenciális modellt, nehezen valósítható meg az iteráció, az egész modell a specifikáció minoségétol függ, a projekt elején meglévo kezdeti bizonytalanságot nem tudja kezelni, nagyon késon lát a megrendelo muködo programot.

Elonyök: modellt ad a különbözo fázisokhoz tartozó módszerek elhelyezésére, logikus, könnyen értheto, kézenfekvo modell. Sok tapasztalat halmozódott fel

4. A szoftverfolyamat modelljei II. (az evolúciós modell, a formális rendszerfejlesztés)

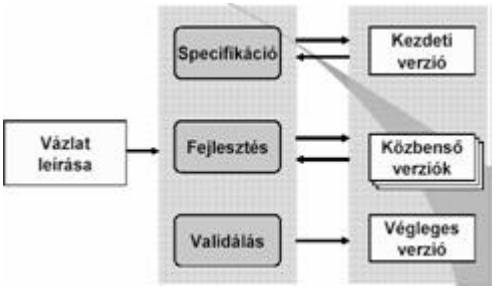
4.1. Az evolúciós modell



Egy kezdeti implementációt (prototípust) a felhasználókkal véleményeztetünk és soksok verzión keresztül addig finomítjuk, amíg a megfelelo rendszert el nem érjük.

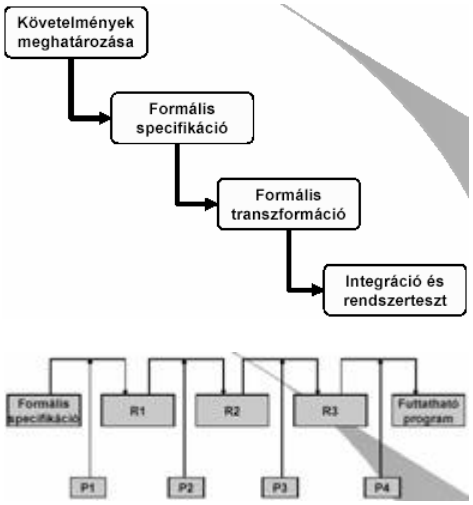
Prototípus típusok: Termék prototípus, Rész-prototípus, Interfész prototípus

Típusok: Feltáró fejlesztés Eldobható prototípus készítése



Problémák: A folyamat nehezen menedzselheto. A rendszerek struktúrája nem megfelelo. Minoségbiztosítási problémák. Speciális eszközök és technikák igénye.

4.2. A formális rendszerfejlesztés



A formális rendszerfejlesztés jellegében a vízesés modellhez hasonlít. Lényege a specifikáció futtatható formába transzformálása formális matematikai eszközökkel

A leglényegesebb különbségek a vízesés modell és formális modell között: A követelményspecifikáció részletes matematikai jelölésekkel leírt formális specifikálás. A tervezés, implementáció, egységteszt egy transzformációs folyamat (a formális specifikáció transzformációk során programmá válik).

A transzformációs folyamat során a kezdeti formális specifikáció transzformálódik egyre jobban kifejtett, de matematikailag korrekt rendszerreprezentációvá. Minden egyes lépés további finomítás mindaddig, amíg a formális Specifikáció vele ekvivalens programmá nem válik.

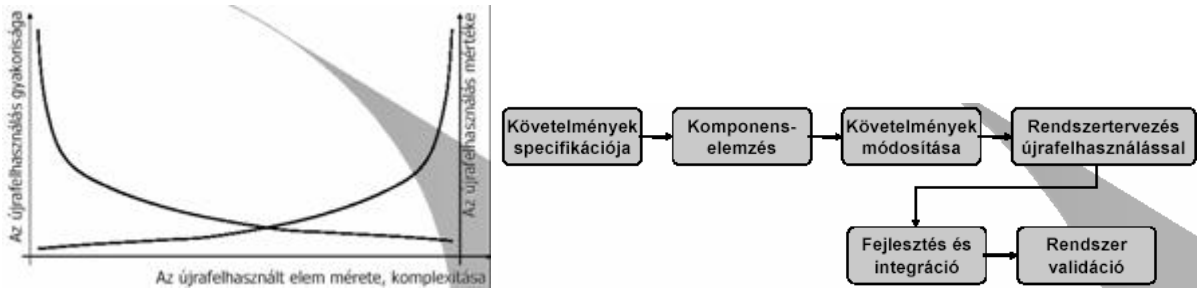
Elonyök: A transzformációk korrektek, így könnyu a verifikációjuk, a sok kis lépés után a program lényegében a specifikáció szisztematikus transzformációja.

Annak bizonyítása, hogy a program megfelel a specifikációnak, más paradigmákkal szemben itt egyszeru, hiszen sok kis ellenorzéssé csökken a transzformációk során.

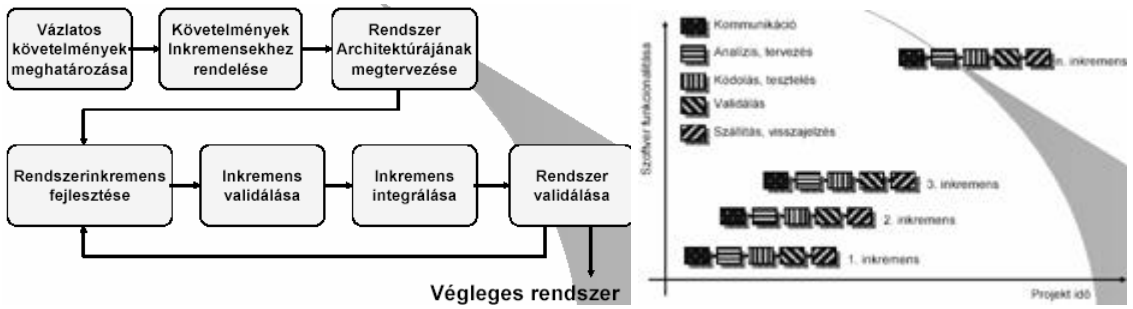
Használata nagyon elonyös olyan rendszerek, vagy részrendszerek esetében, ahol elsodleges a rendszer biztonsága, megbízhatósága és annak autentikus bizonyítása.

5. A szoftverfolyamat modelljei III. (újrafelhasználás orientált fejlesztés, inkrementális fejlesztés, RAD modell)

5.1. Újrafelhasználás orientált fejlesztés

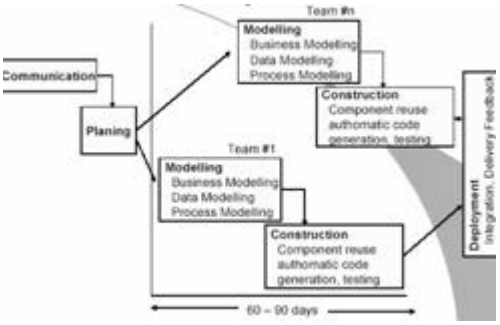


5.2. Inkrementális fejlesztés,



Elonyei: A szoftver már menetközben használhatóvá válik a megrendelo számára (kritikus követelmények teljesülnek eloször). A megrendelok használhatják a korábbi inkrementeket, mint prototípusokat a késobbi inkrementekhez. Kisebb a kockázata a projekt kudarcának, biztonságos a fejlesztés. A legkritikusabb funkciókat teszteljük legtöbbet (azok készülnek el elobb).

5.3. RAD modell

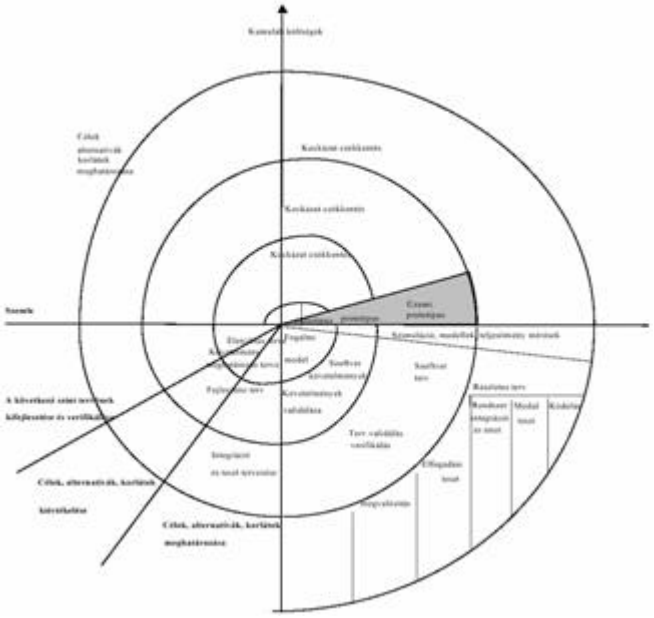


Különbözo teamek dolgoznak párhuzamosan a tervezés fázisától a különbözo funkciókon. A team tagjainak sokoldalúnak kell lennie.A megrendelo szakemberei is részt vesznek a teamek munkájában. Modern fejlesztoi környezetet használ. RAD-et támogató tool-ként hirdetik magukat: Microsoft Visual Studio .NET, Sun Java Studio Creator, BEA Web Logic Workshop, Borland C# Builder, IBM WebSphere Studio Application Developer

Problémák: Nagy projektek esetén nagy a humán eroforrás igény az elegendo számú RAD-team felállításához. Ha a rendszer nehezen modularizálható, a komponensek elkészítése problematikus. Magas muszaki kockázat esetén a RAD nem muködik biztonságosan

6. A szoftverfolyamat modelljei IV. (spirális fejlesztés, Agile Software Processes, XP)

6.1. Spirális fejlesztés



A spirál minden ciklusa négy szektorra bontható: 1. Célok kijelölése (célok, alternatívák, megszorítások, kockázatok meghatározása). 2. Kockázat becslése, csökkentése (kockázat elemzés, kockázati tényezok csökkentésére intézkedés kidolgozása). 3. Fejlesztés és validálás (a következo szintu termék fejlesztése, a kockázat analízis eredményének függvényében fejlesztési modell választása). 4. Tervezés (a következo fázis, ciklus megtervezése).

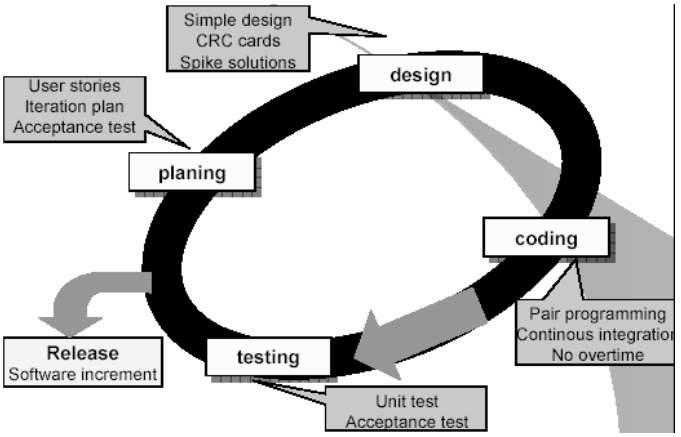
6.2. Agile Software Processes

A legfontosabb elvek: Elso rendu szempont a megrendelo maradéktalan kielégítése. Flexibilitás a követelmények változásával szemben. Muködo szoftver gyors átadása (inkrementálisok). Az üzleti szakembereknek és a fejlesztoknek napi kapcsolatban kell lenniük. Motivált szakembereket gyujts a projekt köré, teremtsd meg a feltételeket a munkájukhoz. Szorgalmazd a szemtol-szembeni párbeszédet a teamen belül az információ cserére. A haladás legjobb mértéke a muködo szoftver. „Agile process” fenntartható fejlodést ösztönöz. A fejlesztoknek és a megrendelonek egy állandó ütemet kell fenntartani a fejlesztési folyamatban. A kiváló muszaki színvonalra és a jó tervre folyamatosan ügyelni kell. Egyszeruség: csak az igazán fontos feladat elvégzése. Az önszervezodo, aktív teamektol származnak a legjobb megoldások (szerkezetekre, követelményekre, tervekre). Rendszeres idoközönként a teamnek önvizsgálatot kell tartani és viselkedését esetleg módosítani.

Ezen elveket követo módszerek: Extreme Programming (XP), Adaptive Software Development (ASD), Dynamic Software Development Method (DSDM), Scrum, Crystal, Feature Driven Development (FDD), Agile Modeling (AM)

6.3. eXtreme Programming

Szabályok:



Planning: User stories are written. Release planning creates the schedule. Make frequent small releases. The project velocity is measured. The project is divided into iteration. Iteration planing starts each iteration. Move people around. A stand-up meeting starts each day. Fix XP when it breaks.

Designing: Simplicity. Choose a system metaphor. Use CRC cards for design sessions. Create spike solutions to reduce risk. No functionality is added early. Refactor whenever and wherever possible.

Coding: The customer is always available. Code must be written to agreed standards. Code the unit test first. All production code is pair programmed. Only one pair integrates code at a time. Integrate often. Use collective code ownership. Leave optimization till last. No overtime.

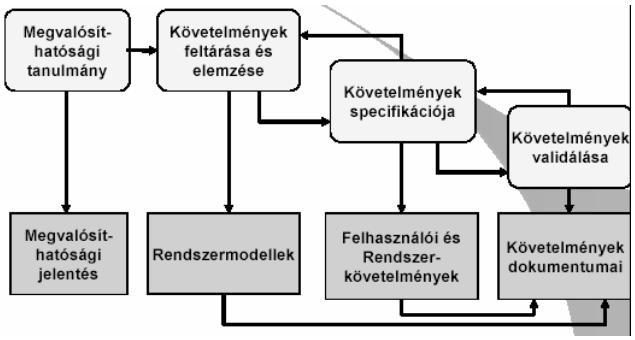
Testing: All code must have unit tests. All code must pass all unit tests before it can be released. When a bug is found tests are created. Acceptance tests are run often and the score is published.

7. Szoftverspecifikáció (fázisok, folyamat, technikák)

7.1. Fázisok

A követelmények tervezésének fázisai: Megvalósíthatósági tanulmány készítése, gyors gazdasági, muszaki elemzés a megvalósításra és az üzemeltetésre vonatkozóan. Követelmények feltárása és elemzése, meglévo rendszerek vizsgálata, modellek, prototípusok készítése, konzultáció a megrendelovel. Követelmény specifikáció készítése, a rendszerkövetelmények és a felhasználói követelmények megfogalmazása, egységes dokumentumba foglalása. Követelmény validáció, a követelmények valószeruségének, konzisztenciájának, teljességének vizsgálata, hibák keresése

7.2. Folyamat



7.3. Technikák

Elozetes interjú módszer jellemzoi: a személyek nem ismerik egymást, tartanak a félreértésektol, hatnak esetleges régi rossz tapasztalatok, ugyanakkor mindkét fél sikerorientált. Az elozetes interjú célja a jég „megtörése”. Módszer: Eloször kötetlen beszélgetés, informálódás a rendszer alapveto kérdései iránt. (Ki fogja használni a rendszert? Mi lesz az elonye a rendszer használatának?). Ezután a rendszer jellemzoivel kapcsolatos kérdéskör kerül elo (Hogy jellemezné, milyen ismereteket kell majd magánviselnie az elkészült rendszernek? Milyen környezetben kell majd üzemelnie a rendszernek?). A harmadik kérdéscsoport az interjú hatékonyságára irányul. (Lényegretöro kérdéseket tettem fel? Van valami, amit meg kellen még kérdeznem?)

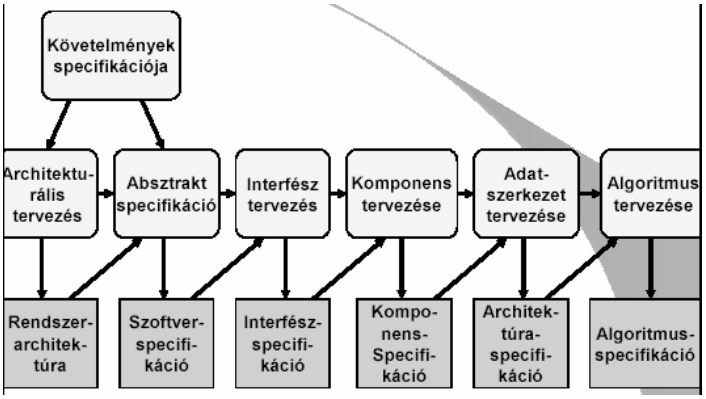
FAST (Facilitated Application Specification Techniques): Probléma: A megrendelo és a vevo tudat alatt „mi és ok” csoportokat képez és ebben gondolkodik. Nem alakul ki igazi team munka (félreértések, lényeges info elvész, stb.). Megoldás: FAST. Az analízis és specifikáció korai fázisát, az információgyujtést támogató team-orientált szisztematikus módszer.
Lényege: A felhasználók és fejlesztok közös teamet hoznak létre a probléma identifikációjára, a megoldás elemeinek kijelölésére, a megoldással szemben támasztott követelmények elozetes meghatározására.
Alapveto jellemzok: Az üléseket semleges fél vezesse. Az ülés elokészítésének, a meghívottak köre kialakításának szabályai rögzítve legyenek, a napirend rögzítse a teendoket, de adjon lehetoséget az ötletek szabad kifejtésére.Elnököt kell kijelölni az ülés irányítására. Definíciós mechanizmus használata ajánlott (táblázat, tábla, Pinwand stb.)
FAST-ülés elokészítése: minden résztvevot megkérnek, hogy készítsen: a rendszert körülvevo környezet objektumainak listája, a rendszert alkotó objektumok listája, azon eljárások, függvények listája, amelyek manipulálják ezen objektumokat. Kényszerfeltételek és muködési kritériumok listája (a listáknak nem kell komplettnek lenni).

8. Szoftvertervezés (a folyamat tevékenységei, általános modell, általános elvek)

8.1. A folyamat tevékenységei

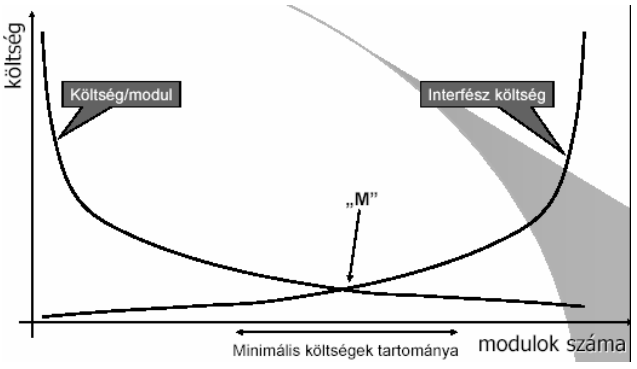
1. Architekturális tervezés: A rendszert alkotó alrendszerek és azok kapcsolatainak azonosítása, dokumentálása.
2. Absztrakt specifikáció: Minden egyes alrendszer szolgáltatásainak absztrakt specifikálása, peremfeltételekkel, megszorításokkal együtt.
3. Interfész tervezés: Minden egyes alrendszer interfészének megtervezése, dokumentálása.
4. Komponens tervezés: A szolgáltatások komponensekhez rendelése, a komponensek interfészének meghatározása.
5. Adatszerkezet tervezés: A rendszer implementációjában használt adatszerkezetek részletes tervezése.
6. Algoritmus tervezés: A szolgáltatások biztosításához szükséges algoritmusok részletes megtervezése.

8.2. Általános modell



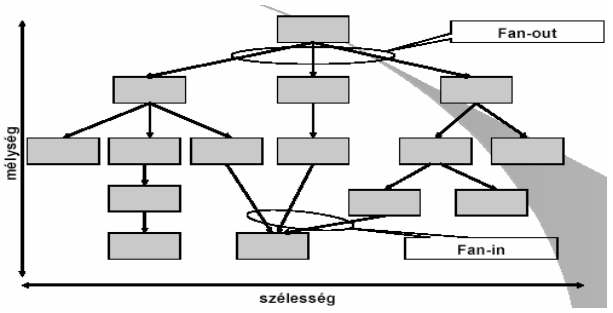
8.3. Általános elvek

Absztrakció: Az absztrakció segít koncentrálni a lényegesre, elhanyagolva a lényegtelent, mindhárom területen alkalmazható (data design, procedual design, architectural design)
Finomítás: Párhuzamosan finomítható program és adatszerkezet (pl: rendezés). A finomítás a funkcionális primitívekig tart. Minden finomítási lépés egy-egy döntést igényel (strukturáló objektum és szempont).
Modularitás (kohézió, csatolás): A kohézió a modul



kompaktságát, integritását, feladatorientált „tisztaságát”, belso összetartását ill. homogenitását (a feladat szempontjából) fejezi ki. A kohézió egy spektrumként fogható fel a gyengétol az erosig. Törekedni kell a minél erosebb kohézióra. A csatolás a modul kapcsolatának intenzitását, erosségét Egy spektrumként adható meg a laza csatolástól a szoros csatolásig. Törekedni kell a minél lazább csatolás kialakítására.

Programszerkezet kialakítás: Ajánlások a programszerkezet kialakításánál: Minimalizálni kell a fan-outokat és fan-ineket. A mélység és a szélesség ne legyen 7-8-nál nagyobb. Törekedni kell az egy bemenetu és egy kimenetu modulokra



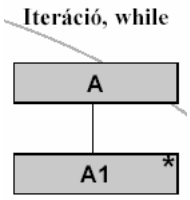
Információrejtés: Lényege: A program elemek (modulok) csak az interfészen keresztül kapcsolódnak akülvilághoz. Belso változók, rutinok, eljárások nem érhetok el a külsok számára. Elonye: Jól definiált interfész. Minoségi biztonság modulok cseréje, javítása esetén (a hiba is be van zárva)

9. Tervezési módszerek I. (kategorizálás, a Jackson féle módszer)

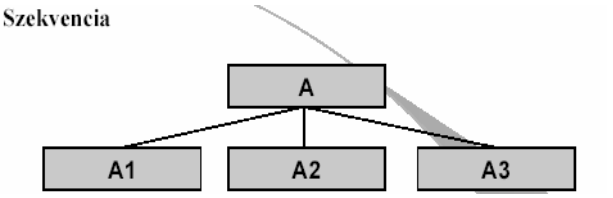
9.1. Kategorizálás, módszerek

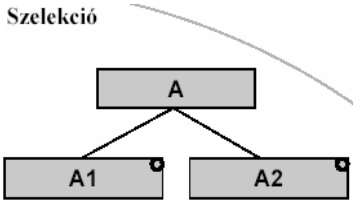
Adatszerkezet orientált: JSP (Jackson)
Adatfolyam orientált SA (DeMarco), SD (Yourdon, Constantine,
Objektum orientált OOSE (Jacobson), OMT (Raumbught &all), UML (Raumbught, Booch, Jacobson)

9.2. a Jackson féle módszer

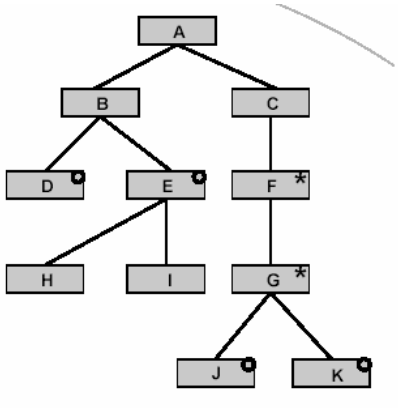


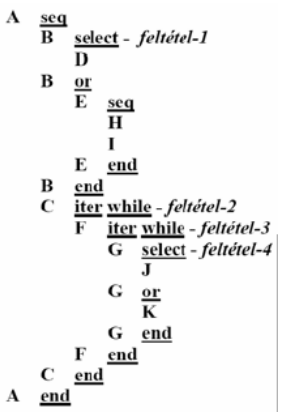
Struktúráló objektum: adattér
Strukturáló szempont: szemantikus szint
Alkalmazott elvek: a probléma hierarchikus (top-down) lebontása, a lépésenkénti finomítás módszere, kizárólag elemi részgráfok alkalmazása (Böhm és Jacopini tétel alapján: szekvencia, szelekció és iteráció használata)





Alkalmazott formalizmusok: szerkezeti ábra (a program- és adatszerkezethez), funkcionális leírás (pszeudokód szeru) a programszerkezet leírásához.
A módszer tervezési lépései:
1. Az adatszerkezetek elkészítése. A feladat szövegének elemzése alapján elkészítjük a bemeno és kimeno adatszerkezeteket a három alapelem (szekvencia, szelekció, iteráció) segítségével. (Annyi bemeno és kimeno adatszerkezetet ábrázolunk, ahány fizikailag létezo állomány van. Sorrend nem számít.)
2. A programszerkezet elkészítése az adatszerkezet alapján
3. A tevékenységek összeállítása „összeírjuk" az összes felmerülo tevékenységet valamint feltételt és sorszámmal látjuk el.
4. A tevékenységek elhelyezése a programszerkezetben. Meg kell keresni azt a programelemet, amelyhez az adott tevékenység a legközelebb áll.

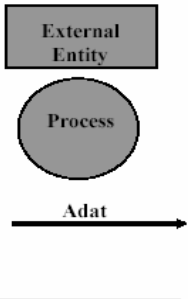


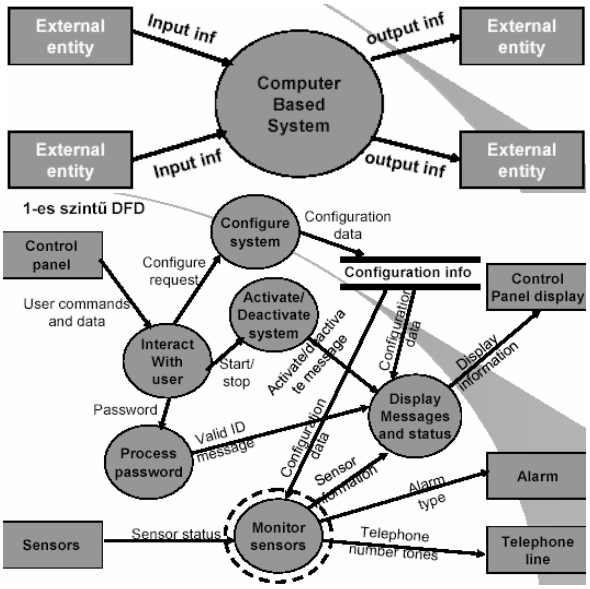


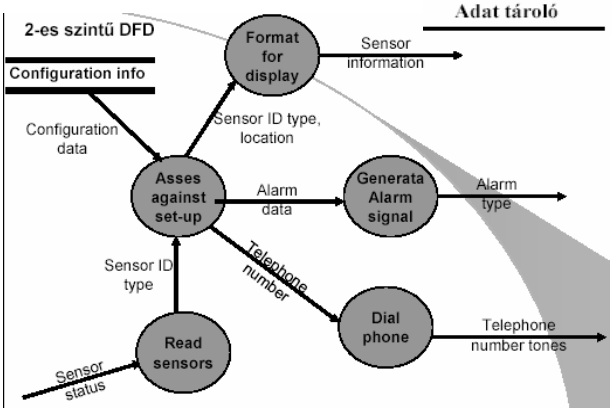
5. A funkcionális leírás elkészítése Leírjuk a teljes programot (pszeudókódban) a célnyelvtol független formában (konkrét tevékenységeket, feltételeket, logikai kapcsolatokat tartalmaz)

10. Tervezési módszerek II. (Strukturált analízis és tervezés)

10.1. Strukturált analízis



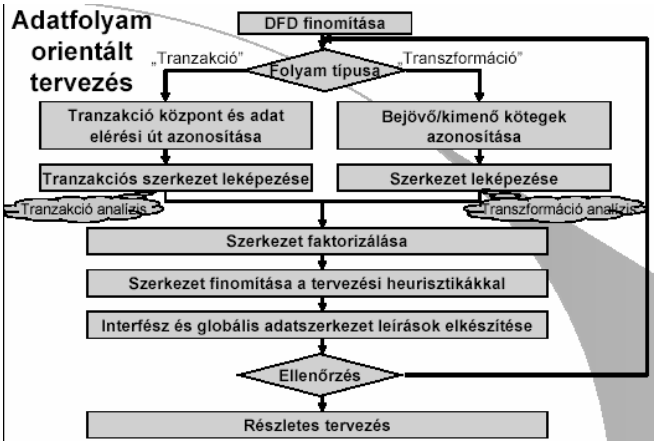




Jellemzoi: 20 éves fejlodési folyamat eredménye modellezési technika, adat és vezérlési áramlást és tartalmat ír le, a funkcionalitás és viselkedés szerint particionál

A 0-s szintu DFD csak egy buborékot tartalmaz, ami nem más, mint a rendszer szoftver. Ezt szokás még: Context model, vagy Fundamental system Model-ként is nevezni. A DFD egyértelmuen maghatározza, melyek a külso egyedek, mi az, ami része a rendszernek és mi az, ami nem. (Hol vannak a rendszer határai.) A DFD az információ áramlását írja le, nem blokk diagram! (sorrendiséget nem tartalmaz). A DFD-t finomítjuk, a buborékokat felvágjuk. (meddig finomítunk, melyek a finomítás szabályai). A DFD-t kiegészítjük két leírással: Data Dictionary, Adatszótár (DD), Process Specification (PSPEC) A folyamat szöveges megfogalmazása. (Algoritmust, megszorításokat, és egyéb részleteket tartalmazó kiegészíto leírás.) (A PSPEC megadásának formái)

10.2. Strukturált tervezés

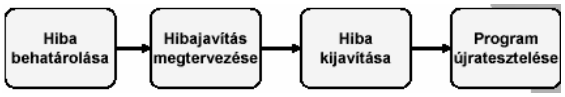


Alkalmazhatóság: igen szélesköru, hiszen a problémák jelentos része leírható információáramlással (DFD)

A DFD típusai: 1.) transzformáció folyam (bejövo folyam - transzformációs folyam - kimeno folyam) 2.) tranzakció folyam (egy „tigger" adat tranzakciókat indít el lásd.) Tervezési heurisztikák: 1.) A programszerkezet kialakításakor vegyük figyelembe a csatolást és a kohéziót /modulok (funkciók) összevonása, szétválasztása/. 2.) Kerüljük a nagy Fan-Out-tal rendelkezo szerkezeteket. Inkább a kiegyensúlyozott, arányos szerkezetekre törekedjünk. 3.) A hatást kiváltó modul vezérlési körében legyen az a modul, amire hatással van. 4.) A modul-interfészek komplexitásának csökkentésére, a konzisztencia növelésére kell törekedni. 5.) Törekedjünk „emlékezet nélküli" modulok kialakítására (azonos inputra mindig azonos választ ad). 6.) Törekedjünk az „egy bemenet-egy kimenet" modulok létrehozására. 7.) A modulok csoportosítása különbözo kényszer feltételek miatt. pl.: portabilitás.

11. Programozás, tesztelés, szoftverevolúció (a belövés folyamata, szoftvervalidáció, tesztelési folyamat, fázisok, dokumentumok, az evolúció folyamata)

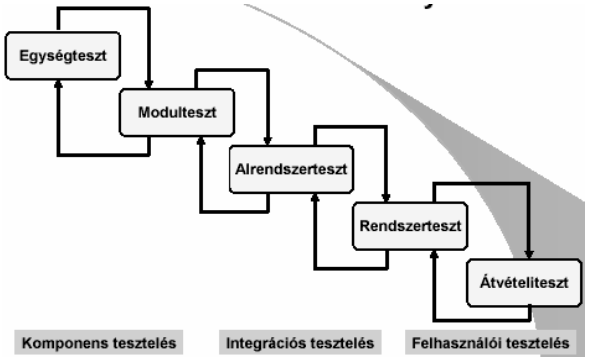
11.1. A belövés folyamata



11.2. Szoftvervalidáci

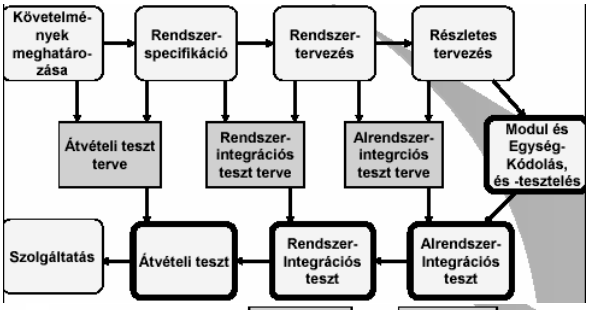
11.2.1. Tesztelési folyamat szakaszai:

Egység teszt (a rendszerkomponensek egymástól független tesztelése [tesztágy szükségessége]). Modul teszt (az egymástól függo komponensek egységbe zárt rendszerének önálló tesztelése). Alrendszer teszt (az alrendszert alkotó modulok egymás közötti kapcsolatának ellenorzése [tipikus az interfész probléma]). Rendszer teszt (az alrendszerek integrált egysége az alrendszer. Tesztelése során az alapveto rendszertulajdonságok vizsgálata történik). Átvételi teszt (a rendszer tesztelése valódi adatokkal, az elore rögzített vakidációs kritériumok teljesülésének ellenorzése).

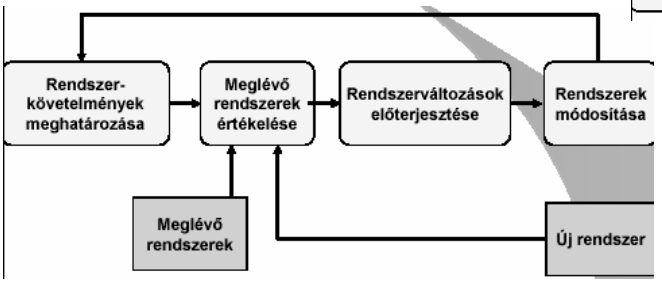


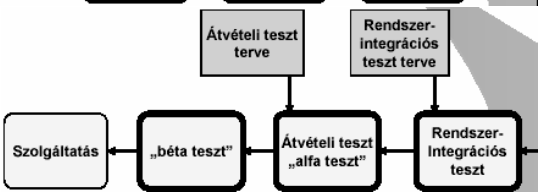
11.2.2. Fázisok, dokumentumok

Az átvételi tesztet szokás „alfa tesztnek” is nevezni. Egyedi megrendelo esetén az alfa teszt a megrendelo bevonásával történik. Nem egyedi szoftverek esetében az alfa tesztet általában egy béta teszt is követi.



11.2.3. Az evolúció folyamata





12. Projekt menedzsment, projekt tervezés (a menedzsment kérdései, a projekt elokészítése, a terv típusai, a tervkészítés folyamata, a terv szakaszai, mérföldkövek a folyamatban)

12.1. A menedzsment kérdései

Projekt tervezése, ütemezése, mérése, becslése. Kockázat menedzsment, kockázat kezelés. Projekt követés és ellenorzés. Team-szervezési megoldások

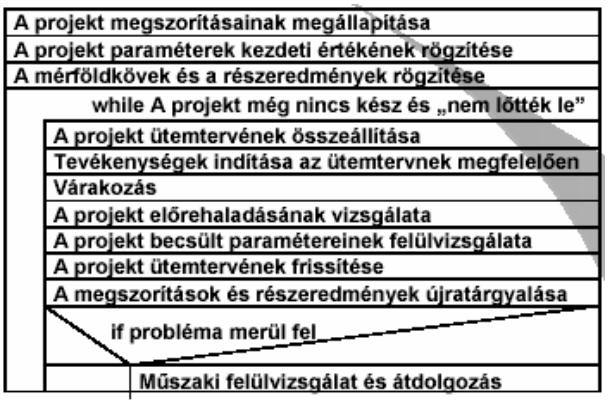
12.2. A projekt elokészítése

A szoftver projekt elokészítése: A megrendelo és a fejleszto közösen meghatározzák: a projekt céljait, a megvalósítandó rendszer fo funkcióit, kvantitatív méroszámokat a funkciók jellemzésére. Alternatív megoldási módokban állapodnak meg a tervezési tér növelése érdekében költség, határido, eroforrás korlátok tisztázása

12.3. A terv típusai

Minoségi terv (a projektben használandó minoségi eljárásokat és szabványokat tartalmazza). Validációs terv (a rendszer validációhoz szükséges megközelítési módot, eroforrásokat és ütemterveket határozza meg). Konfigurációkezelési terv (a konfiguráció kezeléséhez szükséges eljárásokat és szerkezeteket tartalmazza). Karbantartási terv (a rendszer karbantartásához szükséges követelményeket, költségeket tartalmazza). Munkaero-fejlesztési terv (a projektteam tagjainak szakmai fejlodését tartalmazza)

12.4. A tervkészítés folyamata



12.5. A terv szakaszai

1. Bevezetés Meghatározza a projekt célját, alapveto jellemzoit, a menedzselés megszorításait (eroforráskorlátok, idokorlátok)

2. Projekt szervezet Megadja a projekthez rendelt team vagy teamek összetételét, a tagok szerepköreit, feladataikat, felelosségüket

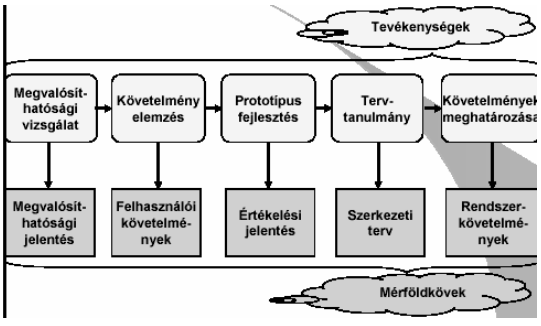
3. Kockázatelemzés Számba veszi a lehetséges projekt kockázatokat, elkerülésükhöz szükséges stratégiákat, bekövetkezésük esetén szükséges tevékenységeket.

4. Hardver és szoftver eroforrás követelmények Rögzíti, hogy a fejlesztéshez pontosan milyen hardver és szoftver eszközökre van szükség. Amennyiben ezeket be kell szerezni, úgy tartalmazza a költségbecslést is.

5. Munka felosztása Tartalmazza a projekt tevékenységekre bontását az egyes tevékenységekhez tartozó részeredményekkel és mérföldkövekkel együtt.

6. Projekt ütemterve Megadja az egyes tevékenységek közötti függoségeket, a mérföldkövek eléréséhez szükséges becsült idot, valamint a tevékenységekhez rendelt becsült emberi eroforrásokat.

7. Figyelési és jelentéskészítési mechanizmusok (projekt követés és ellenorzés) Meghatározza a menedzser által elkészítendo jelentéseket, azok leadási idejét, valamint az alkalmazandó projekt követési, ellenorzési technikákat).



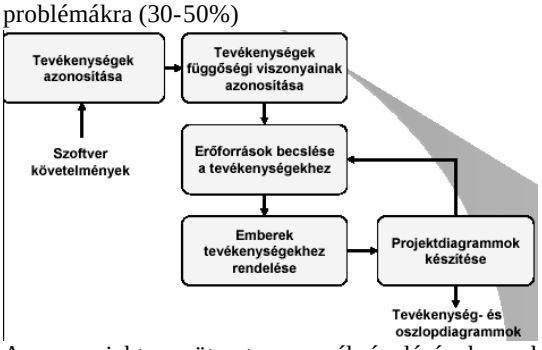
12.6. Mérföldkövek a folyamatban

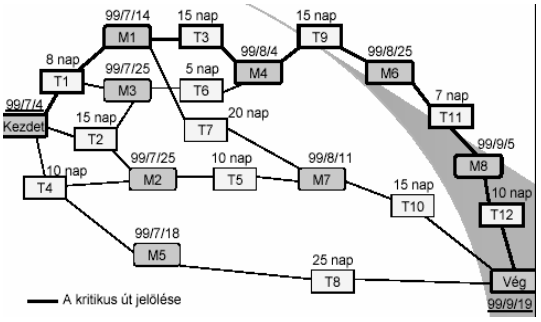
A mérföldkövek alkalmazása egy eszköz a projekt követésére. A mérföldkövek tipikusan a projekt egyes szakaszainak végét jelzik (esetenként ritkább, máskor surubb bontásban). A mérföldkövekhez jelentések (report) tartoznak. A mérföldkövek kapcsolódhatnak belso részeredményekhez, vagy külso, leszállításra kerülo részeredményekhez (pl: specifikáció, stb.).

13. A szoftver projekt ütemezése

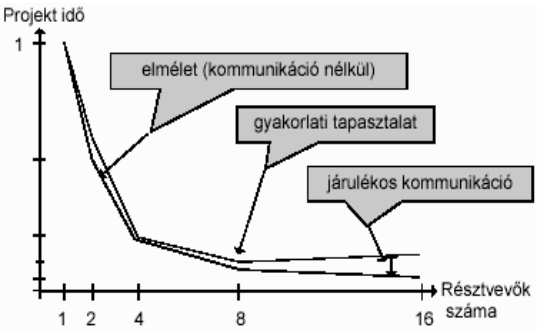
Az ütemezéshez ismerni kell: A tevékenység egységeket, Idoigényüket, Eroforrásigényüket, Függoségeiket, A folyamatok, részfolyamatok sorrendiségeit, A párhuzamosítható tevékenységeket, Egyéb korlátokat, illetve peremfeltételeket.

Ökölszabályok: Ha a tevékenység 8-10 hetet meghaladna, szét kell bontani. Mindig tervezzünk tartalékot a nem várt problémákra (30-50%)





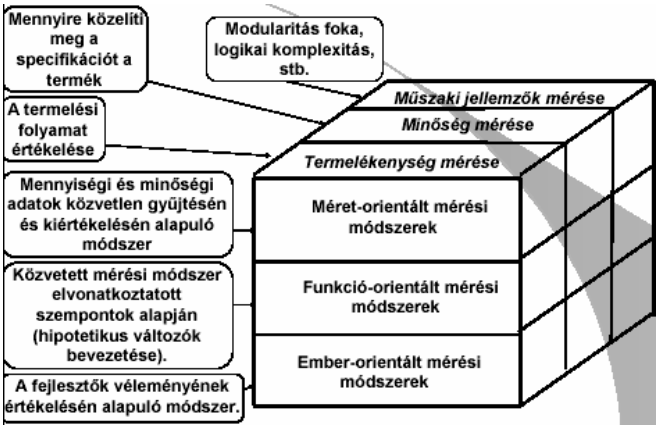
A projekt ütemterv ábrázolásának lehetoségei: Oszlopdiagramok (Tevékenység diagramok): Tartalmazza az adott tevékenység felelosét (munkavégzojét) és a tevékenységek kezdési és befejezési idopontjait. Tevékenységhálók: A tevékenységek közötti függoségeket ábrázolja.



Az emberi eroforrások ütemezésének kérdése: példa: Adott 4 fejleszto 5KLOC/év kapacitással. Ha egy csoportban dolgoznak -> 20 KLOC/év !!! Ebbol lejön azonban a kommunikációs veszteség, ami 250 LOC/év !! (4x5000) - (6x250) = 18500 azaz 4625 LOC/fo. Ha hozzáadunk még két embert a csoporthoz: (6x5000) - (15x250) = 26250 azaz 4375 LOC/fo. Nem érdemes csoportban dolgozni!? Érdemes mert: sokszor a projekt méretei miatt nem is lehet másként megoldani, jobb a minoség és a megbízhatóság mint a „magányos farkasok” esetében nem fog mindenki, mindenkivel „beszélgetni”.

Ütemezési módszerek: PERT (Program Evaluation and Review Technique), CPM (Critical Path Method). Lehetové teszik: kritikus út meghatározását, a feladatok legvalószínubb ido-ütemezését, ido-ablakok meghatározását (legkorábbi, legkésobbi befejezés / kezdés, indítási ido-intervallumok).

14. Szoftver projekt mérése



A szoftver mérése a szoftver mint termék, illetve a szoftver készítési folyamat szignifikáns jellemzoinek számszerusítésével foglalkozik (szoftver metrikák). A szoftver metrikák lehetnek: Vezérlési metrikák (a szoftver folyamattal kapcsolatosak, pl: hibák száma, ráfordítások, idok). Prediktor metrikák (a szoftver termékkel kapcsolatosak, pl: a modul ciklomatikus komplexitása). A vezetoi döntéshozatalt mindkét típus befolyásolhatja

A mérések több célt szolgálhatnak: A termék mérése esetén a termék minoségének értékelése. A folyamat mérése esetén az alkalmazott módszerek, eszközök hatékonyságának értékelése, a fejlesztok termelékenységének elemzése, a következo projektek becslési fázisához bázis-értékek kialakítása, a meglévo értékek javítása.

Méret-orientált mérési módszer: Közvetlen mérési módszer, a mérés tárgya az elkészült termék és a készítés folyamata. A módszer jellemzoi: mennyiségi méroszámokon alapuló egyszeru módszer (oldalak száma, sorok száma, emberek száma, stb.), nem igényel tapasztalatot, szaktudást a kiértékelés, nem veszi figyelembe a komplexitást, a szoftver sajátosságait, csak azonos típusú projektek összehasonlítására, becslésére használhatók fel a méroszámok

Projekt neve	Ráfordítás [ember/év]	Költség [ezer$]	KLOC	Doku. [oldal]	Hibák száma	Részt-vevők száma
A	30	141	11	500	19	4
B	70	200	30	1500	61	8
C	40	155	21	700	5	6
.	.	.	.	.	.	.

$$\text{Termelékenység} = \frac{\text{KLOC}}{\text{Ráfordítás}} \qquad \text{Minőség} = \frac{\text{Hibák száma}}{\text{Ráfordítás}}$$

$$\text{Fajlagos költség} = \frac{\text{Költség}}{\text{KLOC}} \qquad \text{Fajlagos doku} = \frac{\text{Doku}}{\text{KLOC}}$$

Funkció-orientált mérési módszer: A „LOC számolása” helyett a szoftver funkcionalitása kerül elotérbe. A módszer alkalmas a komplexitás kezelésére. A Function Point (FP) valamilyen számszerusítheto jellemzobol származtatható. Tipikus jellemzok, melyeket a módszer alapul vesz: felhasználói inputok száma (adat), felhasználói outputok száma (adat), felhasználói interakciók száma (vezérlés), fájlok száma külso interfészek száma (pl.: hálózat)

Jellemzők	Szám	Súlyfaktor			Érték
		Egyszerű	Átlagos	Komplex	
Felhasználói inputok száma		3	4	6	
Felhasználói outputok száma		4	5	7	
Felhasználói interakciók száma		3	4	6	
Fájlok száma		7	10	15	
Külső interfészek száma		5	7	11	
Teljes összeg					

$$\text{Termelékenység} = \frac{\text{FP}}{\text{Ráfordítás}} \qquad \text{Fajlagos költség} = \frac{\text{Költség}}{\text{FP}}$$

$$\text{Fajlagos doku} = \frac{\text{Doku}}{\text{FP}}$$

A módszer lépései: 1., A számértékeket tartalmazó táblázat kitöltése. 2., A komplexitás értékelése: Módszer: 14 kérdésre adandó 0..5 értéku, osztályzat jellegu válasz összege adja a SUM(Fi), a komplexitást kifejezo tényezot. (max. érték = 70). Példák a komplexitást kifejezo tényezokre: Mennyire szükséges üzembiztos mentés? Mennyire szükséges az osztott feldolgozás? Mennyire legyen a kód újrafelhasználható? 3., A táblázat és a válaszok alapján FP meghatározása FP= [Teljes összeg] x [0.65 + 0.01 x SUM(Fi)] A komplexitástól függoen a [teljes összeg] 0.65 ... 1.35-tel módosulhat. Jellemzok: adat alapú, ami azért elonyös, mert az adatok a fejlesztés korai szakaszában már ismertek, míg a LOC csak a végén, programozási nyelv független, részben szubjektív, a LOC alapúval szemben az értékelés szaktudást, tapasztalatot igényel

Objektum-orientált mérési módszer: Az OO projektek esetén is alkalmazható pl. az FP alapú mérési módszer. Javasolt paraméterek: A szcenáriók száma (a felhasználó és az alkalmazás közti interakciókat írják le – erosen korrelál az alkalmazás méretével). Kulcs osztályok száma (az OO analízis korai szakaszában már definiált, erosen független osztályok – korrelál a számuk a rendszer kifejlesztéséhez szükséges ráfordítással, a szükséges tesztesetek számával. Támogató osztályok száma (nem tartoznak közvetlenül a rendszer probléma specifikus funkcióihoz pl. UI osztályok, szerviz osztályok – számuk jó indikációt ad a rendszer kifejlesztéséhez szükséges ráfordításra, és a reuse lehetoségére. A kulcs osztályokat támogató osztályok átlagos száma GUI alkalmazás esetén a támogató/kulcs osztály arány 2-3, nem GUI alkalmazás esetén 1-2 (mivel a kulcs osztályok már az analízis korai fázisában ismertek, így következtethetünk a teljes alkalmazás méretére. Alrendszerek száma – a rendszer komplexitásának jó indikátora

Use-Case-orientált mérési módszer: Potenciális elonyök a use-case-ek metrikákban történo felhasználása esetén: A szoftver folyamat korai szakaszában rendelkezésre állnak. A rendszer funkcionalitását (legalább a felhasználó oldaláról) jól jellemzik. Programozási nyelvtol, környezettol függetlenek. Számuk arányos a rendszer nagyságával és a szükséges tesztesetek számával. Problémák, ami miatt még nincs széleskörü használat: A use-casek mérete, bonyolultsága, nem szabványosított, erosen függ az alkalmazott absztrakciós szinttol. Nehéz megadni a use-case / ráfordítást

15. Projekt becslés (általános fogalmak, dekompozíción alapuló technikák, tapasztalati becslési modellek, becslési bizonytalanság)

15.1. Általános fogalmak

A becslés tárgya: ráfordítás, költség, idotartam. A becslést nehezíto tényezok:a projekt nagysága, komplexitása, idokorlát, eroforráskorlát, emberi (mennyiségi, minoségi), hardver (speciális egységek: analizátor, stb.), szoftver eszközök

15.2. Dekompozíción alapuló technikák

Mindegyik technika alapja a bonyolult, nagy rendszer szétbontása kezelheto részekre. A kiindulás minden esetben a feladat vázlatos, lehetoleg számszerusített leírása (scope).
I. LOC vagy FP orientált technikák lépései: A., Meghatározzuk az egyes modulok LOC-ját (három értéket adunk meg: optimista, valószínu, pesszimista). B., Képezünk modulonként egy súlyozott átlagot E=(optimista + 4 x valószínu + pesszimista)/6. C., Kitöltjük a táblázatot D., Meghatározzuk a becsült költség és ráfordítás értékeket Az eltéro súlyok a különbözo modultípusok komplexitás különbségeit fejezik ki. Az értékek az elozo projektek méréseibol származnak és projektenként finomíthatók. A módszer jellemzoi: A LOC orientált részletesebb, finomabb felbontást igényel. Az FP orientált nagyobb, komplexebb egységeket képes kezelni. A módszer nagy tapasztalatot, jó „érzéket" kíván, hiszen a projekt elején tényszámok nincsenek. A becslésnél nagy a bizonytalanság.
II. Ráfordítás becslés módszer: Egy „rutinos öreg róka" megbecsüli az egyes tevékenységek ember-hónap ráfordítás igényeit minden modul esetében. A kiindulás ennél a módszernél is a projekt vázlatos leírása. A módszer jellemzoi: A különbözo komplexitású modulok különbözo súllyal szerepelnek. Az egyes tevékenységek súlyozása eltéro. Igen nagy rutint igényel (indirekt becslés), jól kell ismerni a rendszer típusát és a fejlesztocsapatot. Jól használható az elozo módszer kontroljaként.

Modul neve	LOC			Számított érték
	optimista	valószínu	pesszimista	
A	1800	2000	2500	2050
B	700	1000	1200	983
C	5000	6000	8000	6166
D	900	1200	1700	1233
Összeg				10432

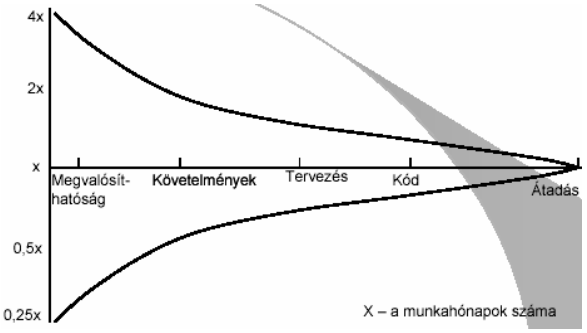
Modul neve	Tevékenységek ráfordításai [ember-hó]				Összeg
	Analízis	Tervezés	Kódolás	Tesztelés	
A	1.5	2.5	0.5	3.5	8
B	2	5	1	4	12
C	3	8	1	8	20
D	2	4	0.5	3.5	10
Összeg	8.5	18.5	3	19	50
Fajlagos ktg. [eFt]	300	200	60	120	
Költség [eFt]	2.550	3.900	180	2.280	8.910

Modul neve	Számított LOC	Becsült Ft/LOC	Becsült LOC/hó	Költség [Ft]	Ráfordítás [ember-hó]
A	2050	800	270	1.640.000	7.6
B	983	1200	75	1.179.600	13.1
C	6166	700	300	4.316.200	20.5
D	1233	1500	110	1.849.500	11.2
Összeg				8.985.300	52.4

15.3. Tapasztalati becslési modellek

Jellemzoik: Általában méretbol indul ki. Tartalmaz exponenciális komponenst a méret – komplexitás viszony kifejezésére. Az algoritmus miatt nem okvetlenül pontos a becslés, nagyban függ a paraméterek jó becslésétol. Általános formája: Munka = A x méret^B x M, ahol a „Munka" a ráfordítás, „A" a helyi szervezetet, illetve a fejlesztett szoftver típusát jellemzi, „méret" a forrássorok száma, „B" a Projekt nagyságát fejezi ki (tipikusan 1 .. 1,5) és „M" pedig a folyamat-, termék-, és fejlesztési jellemzok

15.4. Becslési bizonytalanság



16. COCOMO 81, COCOMO 2

COCOMO = Constructive Cost Model, Jellemzoi: Jól dokumentált. Szabadon hozzáférheto. Széles körben használt. Hosszú ideje használt, több verziót fejlesztettek ki, sok tapasztalat halmozódott fel.

16.1. COCOMO 81

Jellemzoi: Feltételezi a vízesés modell alapú fejlesztést. A szoftver túlnyomó többsége még nem létezik. A modell szintjei: Model-1 (alap COCOMO) egyszeru, egyváltozós modell, a sorok számán alapszik. Model-2 (középszintu COCOMO) a LOC-on kívül szubjektív költségtényezok határozzák meg a ráfordítást. Model-3 (fejlett COCOMO) a Model-2 jellemzoit a szoftver technológia fázisaira (analízis, tervezés, stb.) szétbontva határozza meg.

A COCOMO projektosztályai: Kis méretu projekt: relatív kis méretu projekt, egyszeru feladat, kis team, jól felkészült szakemberek. Közepes méretu projekt: közepes bonyolultságú feladat, közepes méretu projekt, vegyes összetételu team. Nagy méretu projekt: bonyolult hardver körülmények között, kényszer-feltételek mellett fejlesztendo projektek (pl.: légi -irányítási rendszer).

Projekt mérete	a_b	b_b	c_b	d_b
kis	2.4	1.05	2.5	0.38
közepes	3.0	1.12	2.5	0.35
nagy	3.6	1.20	2.5	0.32

COCOMO Model-1: $E = a_b \times (KLOC)^{bb}$; $D = c_b \times (E)^{db}$; ahol: „E" a ráfordítás [ember-hónap] „D" a projekt idotartama [naptári hónap]

Projekt mérete	a_i	b_i
kis	3.2	1.05
közepes	3.0	1.12
nagy	2.8	1.20

COCOMO Model-2: $E = a_i \times (KLOC)^{bi} \times EAF$; ahol: „E" a ráfordítás [ember-hónap], „EAF" a költség befolyásoló tényezok. Költséget befolyásoló tényezok: Termék jellemzoi: igényelt szoftver megbízhatóság, felhasznált adatbázis mérete, a termék bonyolultsága. Hardver jellemzok: run-time kényszerfeltételek, memória kényszerfeltételek, Virtuális gép kényszerfeltételek, igényelt válaszido korlátok. Személyi jellemzok: analizáló szakember kapacitás igény, szoftver technológus szakember kapacitás igény, alkalmazói gyakorlat, virtuális gép gyakorlat, programozási nyelv gyakorlat. Projekt jellemzoi: szoftver toolok használata, szoftver technológiai módszerek használata, igényelt fejlesztési ütemezés. Kiértékelés: mind a 15 kérdésre egy 6 fokozatú skála segítségével, illetve táblázat alapján EAF megállapítása (tipikusan: 0.9-1.4).

16.2. COCOMO 2

Jellemzoi: Elismeri a szoftverfejlesztés különbözo szemléleteit (prototípus készítés, komponens alapú fejlesztés, 4GL, stb). A modell szintjei itt már nem a becslések részletesebbé válását jelentik. A szintek a szoftverfejlesztési folyamat tevékenységeihez kapcsolódnak.

A COCOMO 2 modell azonosított szintjei: Korai prototípuskészítési szint (a méretet objektumokkal becsli, a ráfordítás mértéke méret/termelékenység képlettel számítható ki.). Korai tervezési szint (a rendszerkövetelményekhez valamilyen kezdeti tervezetet készítünk. A becslések funkciópontokon alapulnak, amelyet a forráskód sorainak számává alakítunk át. A képlet alakja marad az egységes formátumú, a szorzóknak egy egyszeru halmaza kapcsolódik hozzá). Posztarchitekturális szint (a rendszer architekturájának elkészülte után már aránylag pontos becslés teheto a szoftver méretére. Itt a becslések már több szorzót tartalmaznak, ezekkel a személyi kapacitások, a termék és a projekt jellemzoi fejezhetok ki.)

17. Kockázat menedzsment

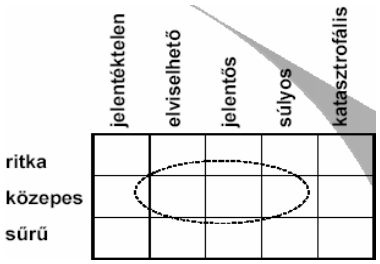
A kockázati kategóriák csoportosítása (kockázat típusok): Projektkockázat (kihatással van a teljes projektre pl: eroforrások rendelkezésre állása, ütemterv betartása, költségvetés tartása, kollégák kiválnak a projektbol, vezetoségváltás). Termékkockázatok (kihatással van a fejlesztés alatt álló termék minoségére, teljesítményére, pl.: fejlesztoeszköz elégtelensége, elvárások jelentos változása). Üzleti kockázatok (a szoftver fejlesztését végzo szervezetre hat ki, pl: konkurens termék kerül a piacra, megváltozik a cég stratégiája, technológiaváltás).

A kockázatkezelés fázisai: 1., Kockázat azonosítás (a lehetséges kockázatok közül meghatározza a valószínu kockázatokat). 2., Kockázat elemzés (megbecsüli a kockázat bekövetkezésének valószínuségét és kihatását). 3., Kockázat tervezés (intézkedési tervek készítése a kockázat csökkentésére, illetve a teendokre a bekövetkezésének esetére). 4., Kockázat figyelés (állandó monitorozás és terv revízió).

A kockázatkezelés folyamata és dokumentumai:

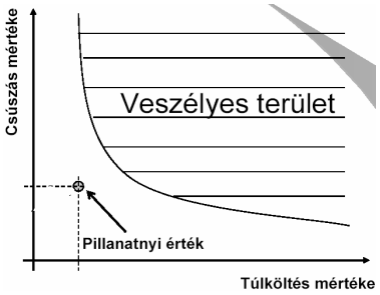
Kockázat azonosítás: Cél: kiválasztani a lehetséges kockázatok közül a potenciális kockázatokat. Módszer: checklist alkalmazása. Eredmény: Potenciális kockázatok listája. Példák kockázatokra: Muszaki-, technológiai-, emberi-, szervezeti-, hardver-, eszköz-, követelmény-, becslési kockázat

Kockázat elemzés: cél: a potenciális kockázatok vizsgálata két szempontból: bekövetkezésének valószínusége és következményei. A kockázat elemzés lépései: A bekövetkezés valószínuségének számszerusítése (alkalmazható skálák: bináris, minoségi, mennyiségi). A bekövetkezéskor fellépo következmények leírása. A kockázat kihatásának becslése. A kockázat elemzés pontosságának megfogalmazása



A kihatás mértéke lehet: jelentéktelen, elviselheto, jelentos, súlyos vagy katasztrofális. A bekövetkezés gyakorisága lehet: ritka, közepes vagy suru.

Kockázat tervezés: Kidolgozandó a kockázat elkerüléséhez szükséges stratégiák, és a bekövetkezésekor elvégzendo teendoket tartalmazó tervek. Kockázat elkerülési stratégiák: azon tevékenységeket és megszorításokat tartalmazza, melyek segítségével csökkenteni igyekszünk a kockázat bekövetkezésének valószínuségét. (pl: bizonytalan elemek lecserélése megbízhatóra). Kockázat minimalizációs stratégiák: célja a kockázat bekövetkezésekor, annak kihatásának csökkentése (pl.: redundanciák, átfedések, tartalékok beépítése). Vészhelyzeti tervek: a kockázati veszélyeztetettség fokára vonatkozó referencia szinteket, a kockázat bekövetkezésekor szükséges teendoket, a helyzet megoldásához tartalékolandó eroforrások leírását tartalmazza.



Kockázat figyelés, (követés): A teljes projekt során szükséges tevékenységek: az azonosított kockázatok bekövetkezési valószínuségének figyelése, a kockázat elkerülési terv végrehajtásának ellenorzése, szükség esetén a tervek módosítása, kockázati esemény bekövetkezése esetén vészhelyzeti terv végrehajtása, hatás ellenorzése. Nagy projektek esetén az azonosított kockázatok száma: 30-40 (felkészülés, eroforrás tartalékolás drága)

Általános szabály: A kockázatmenedzsment költsége ideális esetben a projekt költségvetésének 3-5 százaléka. Ha a kockázat menedzsment költsége eléri a projektköltség 15 százalékát, akkor meg kell gondolni, hogy szükséges-e egyáltalán (a kockázat menedzsment, vagy a projekt maga).

18. Team-szervezési megoldások

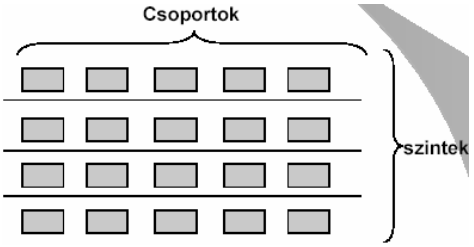
18.1. Géniusz elv

Jellemzoi: A programozás „hoskorára” jellemzo. Az átlagos programozók egy kiváló képességu, vagy autokrata egyéniségu programozó (vezeto) keze alá dolgoztak. Nem volt tervezés (feladat szétbontás, integrálás, stb.) mindent a vezeto döntött el. Az o képességei határozták meg a program minoségét. Fejletlen információs rendszer (gyakorlatilag nincs). A dokumentáció gyakran utólag készül.

18.2. Homogén csoport elv

Jellemzoi: A programozás egy késobbi fázisára jellemzo. Az elv, hogy nem kell vezeto a csoportba (demokratikus muködés: a csoport minden tagja egyenrangú). Döntéseket közösen hoztak, a modularizálást közösen végezték el. A programozás egyénileg történt, az integrálásnál jött az „integrációs BUMM”. A modulok „személyesek” a teljes program személytelen. Az információs rendszer nem fejlett, információ áramlás nincs.

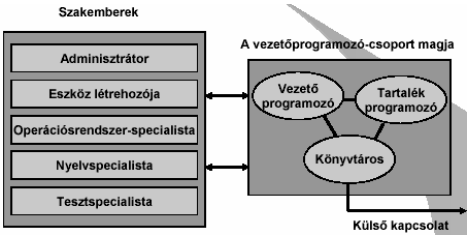
18.3. Horizontális szervezési elv



Jellemzoi: A programozási, programtervezési módszerek fejlodésével kialakult a specializálódás, szakosodás (külön tervezo szakember, külön programozó szakember, stb.). A programozás egyes részfeladataira, tehát specializált csoportok rendszere alakult ki: rendszertervezok, rendszerszervezok, programozók*, kódolók*, tesztelok. A *-gal jelölt csoportok minden munkánál, illetve szoftverháznál megtalálhatók, míg a többi esetleges.

A feladatot mindegyik csoport a saját szempontjai szerint feldolgozza és továbbítja az alatta lévo szintekre. Fejlett információs rendszer és információ továbbítás. Az egyes csoportok közötti kapcsolat pontosan szabályozott.

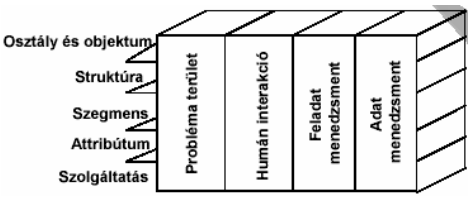
18.4. Vezeto programozó-csoport elv



Jellemzok: Az igen jó képességu programozók hatékonyságának növelése érdekében magas szintu kiszolgálásuk. A csoportot egy három tagú mag alkotja, amelyhez idolegesen kapcsolódnak egyéb szakemberek. Dinamikus, a feladathoz jól illeszkedo struktúra.

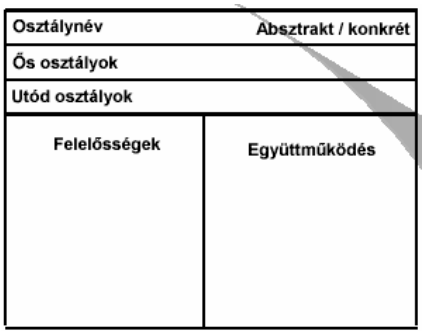
19. Jelentosebb OO módszertanok (OOAD, RDD, OOSE, OOD, OMT)

19.1. OO analízis és tervezés (OOAD)



Jellemzoi: szerényebb eszközrendszerrel rendelkezo, több szintu, több komponensu fejlesztési módszertan. Az *alkalmazott szintek:* osztály & objektum, struktúra, szegmens (subject), attribútum, szolgáltatás (service). *A figyelembe vett komponensek:* probléma-terület, humán interakció (párbeszéd), feladat (task) menedzsment, adat menedzsment.

19.2. Responsibility Driven Design (RDD)

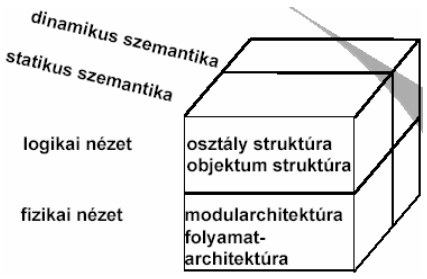


Jellemzoi: Új szemléletu módszer, melyre jellemzo az antropomorph megközelítés. A rendszert egymással együttmuködo és „szövetkezo” objektumok (ügynökök) alkotják, melyek mindegyikéhez jól meghatározott funkció és felelosség kötodik. Az objektumok közti kommunikációt „szerzodések” (contract) rögzítik.

19.3. Object-Oriented Software Engineering (OOSE)

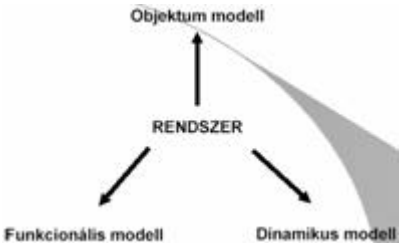
A módszer öt modellt használ az analízis és tervezés fázisaiban: követelmény modell, analízis modell, tervezési modell, implementációs modell, teszt modell. Use Case alapú megközelítés: felhasználók azonosítása, szerepek meghatározása, interakciók számbavétele, rendszer viselkedésének.

19.4. Object-Oriented Design (OOD)



A rendszer statikus leírására használt diagrammok: Osztálydiagram, Objektumdiagram, Moduldiagram, Folyamatdiagram. A dinamikus viselkedés leírására használt diagramok: Állapotdiagram, Idozítés diagram (Timing diagram),

19.5. Object Modeling Technique (OMT)



A rendszert három különbözo nézetbol három modell képezi le: Objektum modell, Dinamikus modell, Funkcionális modell. Az *Objektum modell* a rendszer struktúráját, szerkezetét ábrázolja: Osztály diagramm, Objektum diagramm. A *Dinamikus modell* a rendszer viselkedését képezi le: Szcenárió, Esemény diagramm. A *funkcionális modell* a rendszer funkcionalitását reprezentálja: Adatfolyam diagramm.

20. UML (fogalmak, keletkezése, jellemzoi, célok, tulajdonságok, kritikája, építokövei,)

20.1. Fogalmak

„Az Unified Modeling Language (Egységes Modellezo Nyelv) rendszerek elemeinek specifikálására, megalkotására és dokumentálására szolgáló vizuális nyelv. Általános célú modellezo nyelv, amely minden nagy objektum és komponens alapú módszerrel használható bármely alkalmazási területen (pl. egészségügy, pénzügy, telekommunikáció, légi irányítás) és implementációs platformon (pl. J2EE, .NET).”

20.2. Keletkezése

Alapvetoen három módszer egységesítésébol, összedolgozásából alakult ki: Grady Booch: Booch Methode (BOOCH), Jim Rumbaugh: Object Modeling Technique (OMT), Ivar Jacobson: Object-Oriented Software Engeneering (OOSE) és közremuködött az Object Management Group (OMG), az objektum-orientált szakma legjelentosebb szervezete.

20.3. Jellemzoi

Hamar de-facto szabvánnyá vált. A szoftveripar domináns modellezo nyelvévé emelkedett. Széles körben sikerrel alkalmazzák az egészségügytol az e-kereskedelemig. Széleskörü együttmuködés eredménye több vezeto cég között: pl. Hewlett-Packard, IBM, Microsoft, Oracle, Unisys…

20.4. Célok

A gyakorlatban jól használható a vizuális, kifejezo modellezo nyelv kialakítása. A modell bovítését és specializálását lehetové tevo mechanizmusok biztosítása a nyelvben. Programozási nyelvtol és fejlesztési módszertol független legyen. Formális alap biztosítása a modellezo nyelv megértéséhez. Az OO alapú CASE eszközök piacának növekedését ösztönzi. Magasabb szintu fejlesztési koncepciók támogatása. A létezo legjobb technikák integrálása

20.5. Tulajdonságok

Modellezo nyelv, nem fejlesztési módszertan. Grafikus szemléletet nyújt. Lehetové teszi szoftver-intenzív rendszerek specifikálását, konstruálását, vizualizálását és dokumentálását. Munkafolyamatok, szervezetek, üzleti tevékenységek stb. leírására is alkalmas. A gyakorlatban jól használható egyedi, osztott vagy konkurens rendszerek kezelésére.

20.6. Kritikája

Nem teljesen precíz szemantika, az interpretáció szubjektív lehet, ez megnehezíti a formális tesztelés fázisát. Osztott rendszereknél nem jól alkalmazható, nincsenek meg benne pl. a sorosítás vagy az üzenetküldés faktorai. Túlságosan dagályos, részletekbe meno, olyat is leír, amit a forráskód mutat a legjobban. Széleskörü alkalmazás, sok a hozzá nem érto.

20.7. Építokövei

Az UML szókincse három fo kategóriába sorolható, amelyek tovább bonthatók: Elemek: strukturális, viselkedési, annotációs, csoportos. Kapcsolatok: függoségi, társítási, általánosítási. Diagramok …

20.8. Kapcsolatok

A modellelemek közötti kapcsolatokat testesítik meg. Fajtái: Függoségi (Dependency): Egyik elem változása hatással van a másik elemre, jele: szaggatott nyíl --------->. Asszociáció, társítás (Association): Strukturális, szerkezeti összefüggés, jele: egyenes vonal ———. Generalizáció (Generalization): Általános-speciális kapcsolata, öröklodés, jele: üres háromszög feju nyíl. ———▷

21. UML – Use Case Diagramm

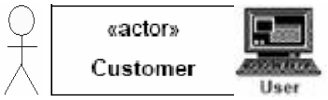
Eszköz a rendszer elvárt muködésének specifikálásához. Tipikusan a követelmények feltárására használják: mi az, amit a rendszernek tudnia kell. Kulcsfogalmak: aktor (Actor), használati esetek (Use Cases), tárgy (Subject), kapcsolatok (Connection).

21.1. A tárgy

A rendszer maga, olyan szempontból, ahogy a használati esetek alkalmazzák. Jelölése: egy téglalap, melynek határa a rendszer határa.

21.2. Az aktor

Felhasználók és más rendszerek, amelyek kapcsolatba lépnek a rendszerrel. Mindig a rendszeren kívül van. Szerepet definiál. Nem feltétlenül fizikai entitás. Jelölése: pálcikaember, osztályként, egyéb ikonnal, ami nem emberi voltára utal



21.3. Használati eset

A rendszer ajánlott viselkedését írja le. Nem utal a belso struktúrára. Eredménye: megváltozik a rendszer állapota, kommunikáció a környezettel. Aktorokkal muködik együtt. Speciális esetei: alapviselkedés variációja (include), kivételes viselkedés, hibakezelés (extend). Jelölése: ellipszis

21.4. Kapcsolatok

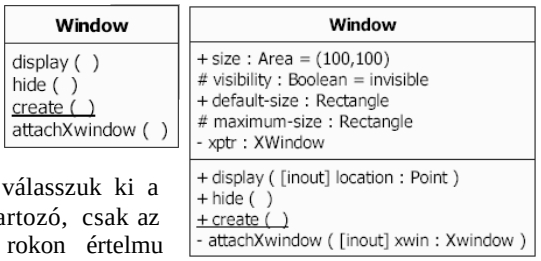
Kapcsolatok	Leírás	Jelölés
Association	A kommunikáció útja az aktor és használati eset között	Egyenes vonal vagy nyíl ———
Extend	Két használati eset között meghatározza a speciális, kiterjesztett viselkedés idejét és mikéntjét.	Szaggatott nyíl a kiterjesztéstől az alap felé. «extend»
Include	Két használati eset között. A tartalmazott eset az alap egy kiemelt része.	Szaggatott nyíl az alaptól a tartalmazott felé. «include»
Generalization	Általános használati esetből vagy aktorból származik egy specifikusabb.	Folytonos nyíl üres háromszögű fejjel. ———▷

22. UML – Osztály diagramm

A modell (egy részének) statikus nézetét írja le. Objektumorientált rendszer építoelemeit mutatja. A rendszer osztályait és a közöttük lévo kapcsolatokat írja le.

22.1. Osztály

UML-es szóhasználat: Attribútumok, Operátorok. Jelölés: három részbol álló téglalap, Név rész, Attribútum rész (opcionális), Operátor rész (opcionális). Láthatóságok: +public, -private, #protected, ~package.



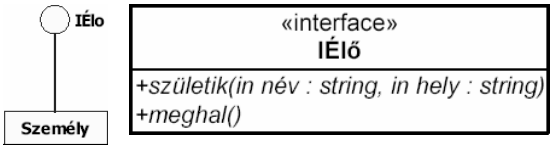
Osztálydeklaráció attribútumokkal:

Osztályszimbólum részletek nélkül:

Az osztályok meghatározásának menete: A leírásból (scope) válasszuk ki a foneveket. Szurjük ki az egyértelmuen nem a rendszerhez tartozó, csak az érthetoséget javító foneveket. Szurjük ki a szinonimákat, rokon értelmu foneveket. Keressünk rejtett foneveket (a szövegben explicite nem szerepel, de a kultúrából következik). Az így kapott jelöltekbol válasszuk ki, a rendszert alkotó objektumokat.

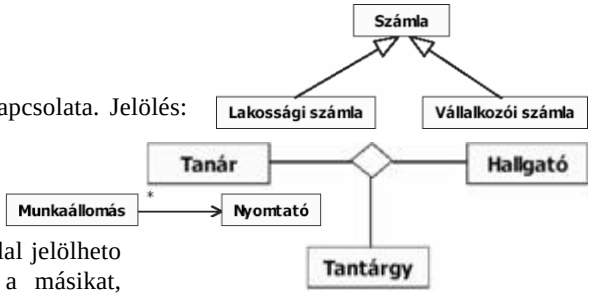
22.2. Interface

Operációk halmazát adja meg, melyet más osztályok megvalósítanak. Az interfészt megvalósító osztály és az interfész között realizációs kapcsolat van. Jelölés: Osztályjelölés + <<interface>> sztereotípia, kör. A dolt betus operátor- és osztálynevek az elem absztrakt voltát jelölik.



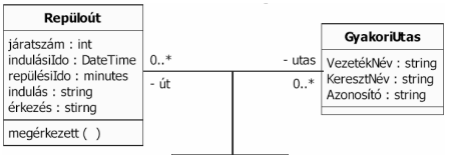
22.3. Generalizációs kapcsolat

Az öröklodést fejezi ki. Általános és specifikus osztály kapcsolata. Jelölés: Operátornál kettospont után a visszatérési érték van.



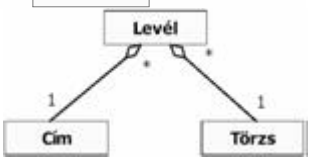
22.4. Asszociációs kapcsolat

Két osztály összekapcsolása a legáltalánosabb reláció. Nyíllal jelölheto a navigálhatósága: A nyíl irányában ismeri az egyik a másikat, multiplicitás jelöli, hogy hány példány vehet részt a kapcsolatban, az alapértelmezett az 1. Az objektumok szerepét a kapcsolaton jelölhetjük. Asszociáció megadása osztállyal: A kapcsolatra vonatkozó információkat tartalmaz. Osztályként adható meg, szaggatott vonallal kapcsolódik az asszociációhoz. Az osztálynak attribútumai vannak.



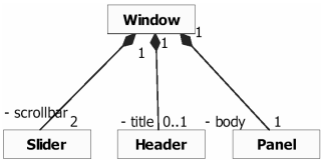
22.5. Aggregációs kapcsolat

Az asszociáció egyik formája. Az egyik osztály alárendeltje a másiknak. Tartalmazó-tartalmazott viszony, a tartalmazott túlélheti a tartalmazót



22.6. Kompozíciós kapcsolat

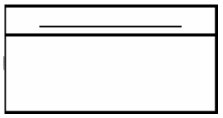
Az aggregáció erosebb formája. A részek életciklusa egybeesik a tartalmazóéval. Egy objektum csak egy tartalmazóhoz tartozhat. A tartalmazó objektum felelos a részek létrehozásáért és megsemmisítéséért. Osztott aggregáció: olyan aggregáció, ami nem kompozíció a tartalmazók osztozhatnak a részeken. A kompozíció definíciója jól kötodik az automatikus szemétgyujtés (garbage collection) fogalmához. Ha a tartalmazó objektum megsemmisül, a részre mutató is megsemmisül, és a rész elérhetetlenné válik. A szemétgyujto hatáskörébe kerül, visszaállítani lehetetlen.



23. UML – objektum és szekvencia diagramm

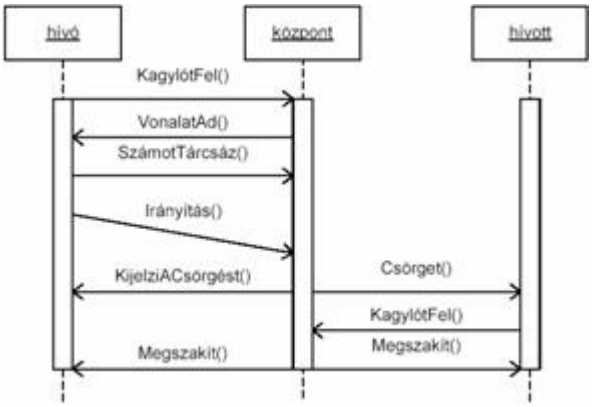
23.1. Objektum diagramm

Objektum: egy osztály egy példánya. Objektumdiagram: objektumokat és azok kapcsolatait mutatja egy adott pillanatban. A rendszer adott idoben vett állapotát reprezentálja. Az objektum minden attribútumának van értéke. Az objektum összeköttetésben áll más objektumokkal. Link: az asszociáció egy példánya.



Objektumok jelölése: Hasonít az osztálydiagramok ábrázolásához. Két része van: névrész, objektum vagy az osztály neve aláhúzva, attribútum rész értékekkel. A metódusok (operációk) feltuntetésére nincs szükség, hiszen azok minden objektumban azonosak.

23.2. Szekvencia diagramm



Viselkedési, interakciós diagram. Az objektumok interakcióit mutatja az ido függvényében. Elemei: objektumok, melyek részt vesznek az interakcióban és a kicserélt üzentek sorozata. Explicit mutatja az idot, de nem jelöli az osztályok kapcsolatát.
Jelölés: Aktivációs életvonal: az objektumhoz kapcsolódó muveletek hajtódnak végre. Vízszintesen az interakcióban részt vevo objektumok vannak. Egy objektum téglalap jelölése elhelyezkedhet a diagram tetején, vagy ahol létrejön. Az üzenetek különbözo típusúak lehetnek. „X” jelzi az objektum megszunését. Függolegesen jelenik meg az ido. Életvonal: az objektum idoben való létezését jelenti.
Üzenet típusok: Egyszeru üzenet: az aktív objektum átadja a vezérlést a passzív objektumnak. .
Szinkronizációs üzenet: a küldo blokkolt állapotba kerül, amíg a fogadó nem fogadta az üzenetet.
Randevú üzenet: a fogadó várakozik arra, hogy a küldo üzenetet küldjön neki.  Aszinkron üzenet: a küldo folyamat nem szakad meg, nem érdekli ot, hogy mikor kapta meg a fogadó az üzenetet.
Visszatérési üzenet: amikor az objektum a vezérlést visszaadja, gyakran nem tüntetjük fel
Feltételek, ciklusok: A diagramon belül beágyazott diagramrésszel ábrázoljuk. A részek alrészekre oszthatók (vízszintes szaggatott vonallal), ha szükséges (pl. feltételnél). A rész viszonyát az egész diagramhoz a sarkába írt cím jelzi.

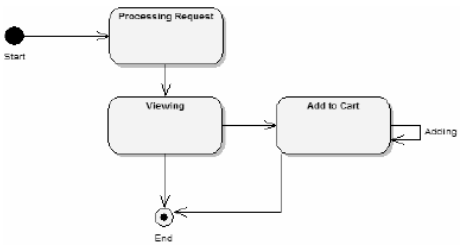
24. UML – Együttmuködési diagramm, állapot diagramm

24.1. Együttmuködési diagramm

Objektumok közti interakciót (párbeszéd) mutatja. Az objektumdiagram és a szekvenciadiagram keresztezése. Szabad elrendezést enged az objektumok között (a szekvenicadiagramtól eltéroen). Így könnyebb adott objektum interakcióit áttekinteni. Az üzenetek idorendben számozottak. Az UML 2.0-ban Communication Diagramnak hívják

24.2. Állapot diagramm

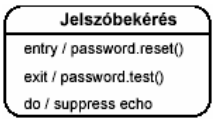
Elnevezés, eredet: David Harel dolgozta ki a statechart diagramok alapelvét. Az UML state machine fogalma ezen alapszik. Egyetlen objektum vagy interakció állapotainak sorozatát írja le. Az állapotgép forrásosztályhoz kapcsolódik, és példányainak viselkedését specifikálja
Az állapotdiagram elemei: Állapot, Belépési pont (entry point / Start), Kilépési pont (exit point / End), Átmenet: állapotok közötti kapcsolat (nyíl). Feltüntethetjük rajta: az esemény nevét, a feltételt, ami az esemény kiváltásához szükséges és a végbemeno tevékenységeket.



Állapotok: Egy állapot az objektum az attribútum értékeinek és csatolásainak absztrakciójaként értelmezheto. Egyszeru esetben minden különbözo attribútum érték külön állapotnak tekintheto. Általános esetben az attribútum értékek, és csatolások halmaza képez egy

állapotot. Azon attribútumok, melyek az objektum viselkedését nem befolyásolják, melyek a vezérlésmintára nincsenek hatással, figyelmen kívül hagyhatók. (A vizsgálat szempontjából nem fontosak!) Az objektum viselkedését tekintve az állapot egy idotartamot jelent, mely két esemény között telik el. Az események és az állapotok összetartoznak. Az események állapotokat határolnak, és fordítva, az állapotok eseményeket terminálnak. Úgy az állapotok, mint az események az absztrakciós szint függvényei.

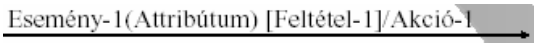
Állapot jelölése: Fontosabb részei: név, be- és kilépési tevékenység (entry/exit), belso tevékenység (do)



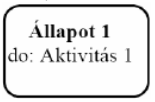
Események: Az események a külvilágból érkezo külso ingerek, melyek a rendszert „stimulálják”, melyekre a rendszernek meghatározott válaszokat kell adnia. Az események lehetnek belso ingerek, melyeket a rendszert alkotó objektumok bocsátanak ki. Az objektumok különbözo eseményekre adott válaszai függenek azon objektum állapotától, mely az eseményt észleli. A reakció lehet: nincs állapotváltozás, állapotváltozás, esemény küldése az eredeti objektumnak, esemény küldése egy harmadik objektumnak. Az események bekövetkezhetnek egy másik esemény elott vagy után (okozati összefüggés). Lehetséges azonban, hogy az események nincsenek hatással egymásra (nincs okozati összefüggés). Az okozati összefüggés nélküli eseményeket párhuzamos eseményekként kezelhetjük. (Az események idopontokat reprezentálnak, idotartam hozzájuk nem rendelheto.). Egy esemény egy információ átvitelként is tekintheto egyik objektumtól egy másik objektumhoz.

Feltételek megadása: Egy feltétel egy logikai függvény, mely igaz vagy hamis értéket vehet fel. A feltételek rendelhetok eseményekhez. Az események feltételei, mint „or” viselkednek. Egy „orzött” átmenet akkor aktív, amikor az esemény megtörténik, azzal egy idoben a feltétel is igaz. A feltételeket az eseménynév mögött [ ] jelek között adjuk meg.

Muveletek: Az események nem csak átmeneteket, hanem muveleteket is kiválthatnak. Muveletek: Akció (nincs idotartama, eseményhez kapcsolódik) Jelölése:



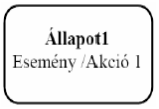
Aktivitás (idotartammal rendelkezik, állapothoz kapcsolódik). Jelölése:



Bemeneti és Kimeneti akciók: Lehetoség van arra, hogy akciókat egy állapotba való belépéssel, vagy abból történo

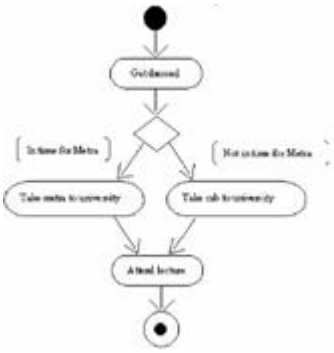
kilépéssel kössünk össze. Jelölés:

Belso akciók: Egy esemény akció végrehajtását kiválthatja anélkül, hogy az állapotot elhagyná. Ezen akciókat nevezzük belso akcióknak. Jelölés:



25. UML – Aktivitás (activity) diagramm, package diagramm

25.1. Aktivitás (activity) diagramm



Tevékenységek procedurális folyamatait modellezi. Eljárások szekvenciális és konkurens lépéseit írja le. Egy tevékenység összes lehetséges folyamatát mutatja. Használható: használati esetek részletezésére, rendszerszintu funkciók modellezésére. Fókuszál a tevékenységek sorrendjére, a feltételekre, melyek kiváltják a tevékenységeket

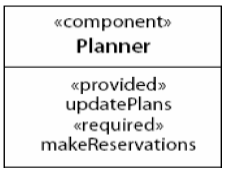
Jelölések: Tevékenységi állapot: objektum tevékenységét jelöli. Átmenet. ⟶ Kezdeti állapot. ○ Végállapot. ◎ Szinkronizációs vonal: ▬▬▬ párhuzamos folyamatok elején és végén használjuk. A tevékenységek oszlopokba rendezhetok objektumok vagy feladatok szerint

25.2. Csomag (package) diagramm

Jellemzoi: A rendszert a legmagasabb absztrakciós szinten bontja fel. A logikailag összetartozó UML elemeket (foleg osztályok) csoportosítja. A csomagok között függoségi kapcsolatokat (pl. egyik elem a másikban, változást idéz elo, üzenetküldés) ábrázolja. A csomagok egymásba ágyazhatók. Jelölés: négyszög füllel. A csomag megfeleloje Java környezetben a package, .NET környezetben a namespace

26. UML – Komponens diagramm, Konfigurációs (Deployment) diagramm

26.1. Komponens diagramm



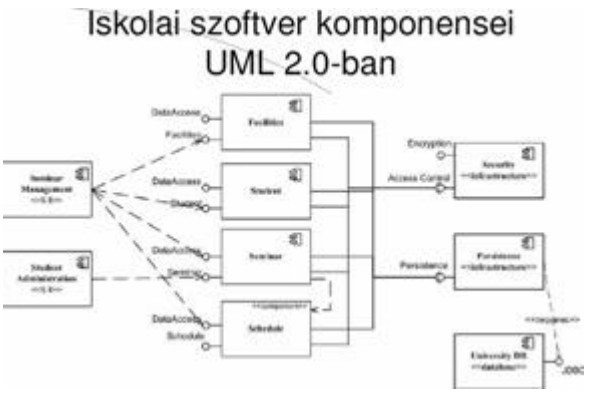
Története: A komponensek lényege új irányt vett az UML 2.0-ban az addigi fizikai szemlélethez képest. Itt a komponensek az elemek fizikai fogalmától elkülönültek, fogalmi modellként használhatók. Halvány különbség van a strukturált osztály és a komponens között.

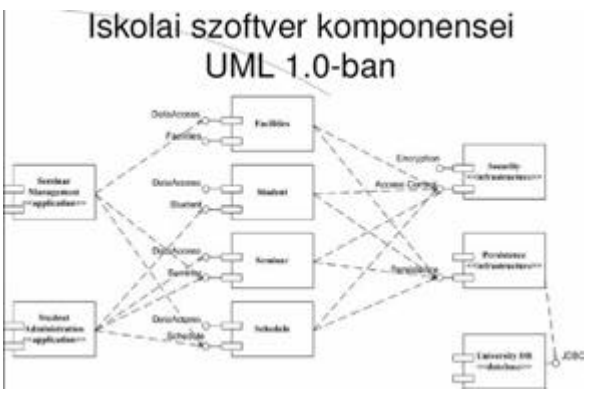
Komponens UML 2.0-ban: A fizikai és logikai rendszer azon moduláris részeit írja le, amelyek kifelé látható viselkedése jobban leírható, mint a megvalósításuk. A kifelé látható viselkedéseket interfészek halmaza reprezentálja. Két nézopontot mutatnak: definiálják a rendszer részeinek külso arculatát, megvalósítják a rendszer funkcionalitását.

Jelölése: Két nézet: Téglalap és a <<component>> kulcsszó. Téglalap benne a komponens ikonnal (az UML 1-es verzióiban ez jelölte a komponenst a téglalap helyett)

Interfészek: Provided interface: olyan összetartozó szolgáltatások halmaza, amelyeket a komponens elérhetové tesz. Required interface: a komponensnek meg kell kapnia a szolgáltatást.

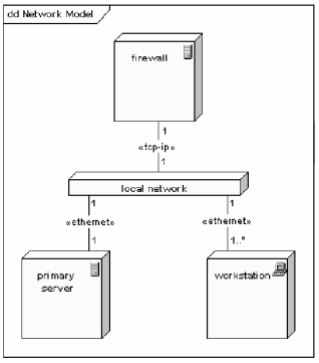
Az UML verziók komponens-diagramjainak összehasonlítása: Másképp jelölik a komponenst. A komponensek közti függoséget mind a ketto szaggatott nyíllal jelöli. Az UML1 egyfajta interfészt használ, amíg az UML2 másképp jelöli azrequired interfészt, amellyel egyben a függoséget is kifejezi



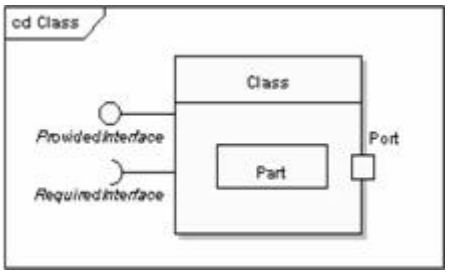


26.2. Konfigurációs (Deployment) diagramm

A rendszer futási ideju architektúráját modellezi. A hardverelemek (nodes) konfigurációját mutatja, valamint a szoftverelemek leképzését node-okra. Node: Eroforrást modellez (pl. számítógép, lemezmeghajtó). Eszközök és futtatható környezetek ilyenek. Legalább memóriával, gyakran feldolgozó képességgel rendelkeznek.



27. UML – Composite structure, Kollaborációs, Timing, Interaction Overview diagrammok

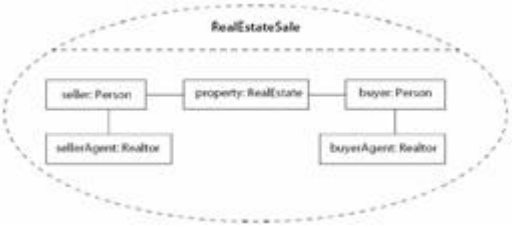


27.1. Composite structure diagrammok

Struktúra diagram. Az osztály, komponens stb. belso struktúráját, és a rendszer más részeihez való kapcsolódását mutatja. Az osztálydiagrammal ellentétben az osztályt alkotó elemek összetételét mutatja: interfészek, portok, részek, kollaboráció.

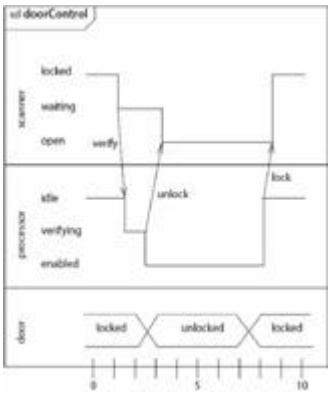
27.2. Kollaborációs diagrammok

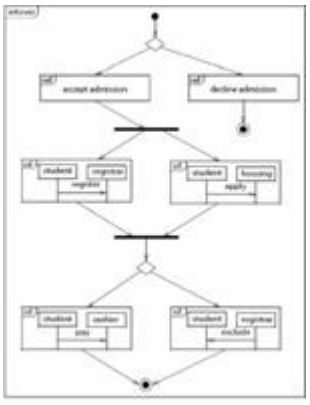
Semmi köze a Collaboration Diagramhoz, ami az UML 1.x-ben volt. Statikus struktúra. Együttmuködo részek struktúráját mutatja, amelyek együttesen látnak el kívánt feladatokat. Gyakran patternt valósít meg. Jelölése: szaggatott ellipszis a kollaboráció nevével



27.3. Timing diagrammok

Viselkedési interakció diagram. A szekvenciadiagram bemutatásának alternatív módja. Explicit mutatja egy életvonalon az állapotokban bekövetkezett változásokat. Valósideju alkalmazásoknál hasznos. Különbség a szekvenciadiagramhoz képest: A tengelyen balról jobbra az ido no. Az életvonalak függolegesen, elkülönült részekben találhatók. Az életvonal le-fel ugrál, az állapotok változását mutatva. Mindenegyes függoleges helyzet külön állapotot jelent. Az állapotok sorrendje nem feltétlen bír jelentéssel





27.4. Interaction Overview diagrammok

Viselkedési interakciós diagram. Az aktivációs diagram variációja, mely magában foglalja a szekvenciadiagramot. A szekvenciadiagram jelölését használja az aktivációs diagramból vett döntésekkel és elágazásokkal.

28. RUP I. (jellemzok, történelem, UML és RUP, módszertan, technológia, eszközök, emberek, szervezeti minták)

28.1. Jellemzok

Passz

28.2. Történelem

OMT (Object Modeling Technique) (1991) James Rumbaugh (General Electric 1994 Rational). Booch method (1991) Grady Booch (Rational 1981 a Rational alapítása óta). OOSE (Object-Oriented Systems Engineering) (1992) Ivar Jacobson (Ericson 1987, Rational Software 1995). UML (Unified Modelling Language) (1997). RUP (Rational Unified Process) (1998)

28.3. UML és RUP

Passz

28.4. Módszertan

Software Development Process Megmondja: Ki, mit, mikor, és hogyan csinál, hogy elérje a megadott célt. Az aktuális technológiákra, eszközökre, emberekre, szervezeti mintákra épít.

28.5. Technológia

Az eljárásnak technológiára kell épülnie! Programozási nyelvek, operációs rendszerek, számítógép rendszerek, hálózati lehetoségek, fejlesztoi környezetek… Melyek adott idoben a rendelkezésünkre állnak. Pl. a 80-as években a vizuális modellezés nem volt realitás, túl drága volt. Kézzel rajzolt diagramokat használtak. Ez nagyban korlátozta a munkát.

28.6. Eszközök

Az eljárásnak és az eszközöknek párhuzamosan kell fejlodniük. Az eszközök beépülnek az eljárásba. A széles körben alkalmazott eljárás támogathatja, irányíthatja az eszközfejlesztéseket.

28.7. Emberek

Az eljárás fejlesztoinek korlátozni kell a készségeket, amelyek szükségesek az eljárás használatához, hogy azokat a fejlesztok gyorsan elsajátíthassák. Sok területen beépíthetünk olyan technikákat, melyek egyszeri beruházást és képzést igényelnek pl. a konzisztencia megorzés számítógép alapú eszközöknél.

28.8. Szervezeti minták

A szoftverfejlesztok nem egyedül dolgoznak. A folyamat építoinek illeszteni kell az eljárást a napi valósághoz, a virtuális szervezetekhez, melyek egymástól nagy távolságra nagysebességu hálózatokon kommunikálnak. Résztulajdonosok, fizetett alkalmazottak, szerzodéses munkások, outsourcing, szoftverfejlesztok hiánya.

29. RUP II. (a RUP fázisai)

29.1. Elokészítés (Inseption)

Döntés a megvalósíthatóságról. Egyetértés a követelmény-feltárás eredményében. Költség, menetrend, prioritások, rizikófaktorok. Minden kockázat elemezve van és van mérséklo stratégia. Az áttekintés, a vízió kidolgozva. Kockázat lista. Iterációs terv: Tevékenységek idoigénye (Gantt diagram), Eroforrásigény, Függoségek (Gantt diagram), Részletes terv. Szójegyzék. A legkritikusabb használati esetek. Prototípusok.

29.2. Kidolgozás (Elaboration)

Az áttekintés és a követelmények rögzítve. A struktúra rögzítve. A tesztelés és az értékelés kulcsfontosságú elemei meghatározva. Az építés fázis ciklusainak terve részletesen kidolgozva. A terv adott szerkezetben történo megvalósítása az áttekintést eredményezi. Az aktuális költségek a tervezett költségeknek megfelelnek. Néhány végrehajtható prototípus kész. Feltárják a kritikus funkcionalitást, és a szerkezet meghatározó elemeit. Ismert és nyitott kockázatok csökkeno fontossági sorrendben speciális mérséklési lehetoségek feltárásával. Software Architecture Document: Különbözo szerkezeti nézetek (use case view, logical view, deployment view). Design Model: Komponensek meghatározva make/buy/reuse figyelembevételével. Data Model: fontos egyedek, kapcsolatok, táblák. Implementation Model: Implementation files, Subsystems. Vision: use casek és szerkezeti tervek alapján átdolgozva. Design guideline, Programming guideline. Software Architecture Document. Iterációs terv az építés fázishoz. Use case model: 80%-ban kész. Járulékos követelmények dokumentálva és áttekintve.

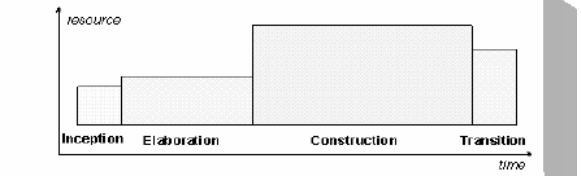
29.3. Építés

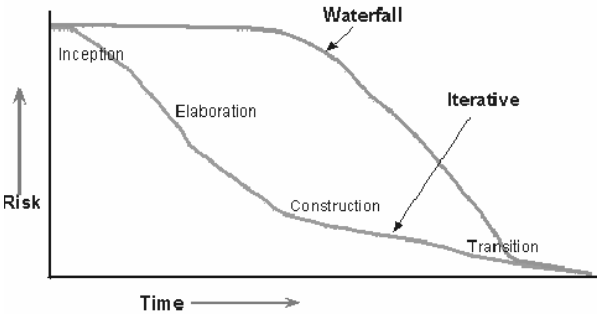
Az összes funkcionalitás kifejlesztve és alpha tesztelve, felhasználói kézikönyv és leírás. A rendszer kész a bétatesztelésre. Imlementation: Minden fájl készen tesztelve. Iterációs terv az átadás fázishoz. Design Model: frissítve, az összes követelménynek megfelel. Data Model: tables, indexes, object-orelational mappings, etc. Járulékos követelmények: frissítve, feltárva. Use Case Model: frissítve az építés fázis új use case elemeivel.

29.4. Átadás

A felhasználó elégedett? Az aktuális kiadások a tervezett kiadásokhoz képest elfogadhatók?

	Előkészítés	Kidolgozás	Megvalósítás	Átadás
Erőfeszítés	~5%	20%	65%	10%
Idö	10%	30%	50%	10%



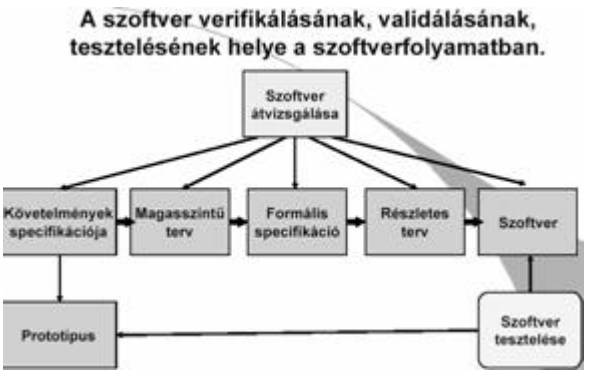


30. Verifikáció és validáció

A verifikáció és a validáció (V&V) azon ellenorzo és elemzo folyamatok összessége, amelyek célja annak vizsgálata, hogy a szoftver megfelel a specifikációnak. Ennek része a hagyományos értelemben vett szoftvertesztelés is.

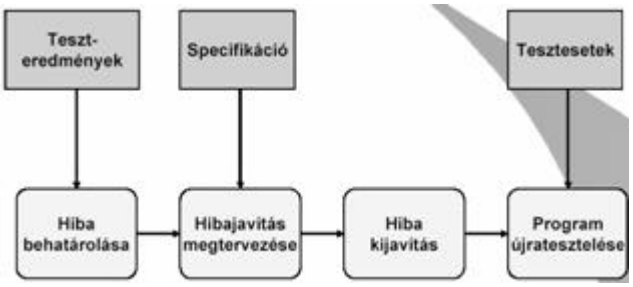
Verifikáció: A terméket jól készítjük el? (a termék megfelele-e a specifikációnak, illetve a funkcionális és nem funkcionális követelményeknek).

Validáció: A megfelelo terméket készítjük el? (a termék megfelele-e a vevo elvárásainak, ezért a validáció már a követelmények megfogalmazásánál kezdodik)



A V&V folyamaton belül 2 technika használható:1., A szoftver átvizsgálások: A rendszer fejlesztése során keletkezo dokumentumok (követelmény-dokumentáció, tervek, ábrák, forrássorok, stb.) ellenorzése. Ezek statikus technikák, mert szükséges hozzá a program futtatása (a korai szakaszban még nincs is program). 2., A szoftvertesztelések: Az elkészült szoftver ellenorzése különbözo tesztadatok futtatásával. Ellenorizzük a szoftver kimeneteit, a futás közbeni viselkedését és a teljesítményét. Ez úgynevezett dinamikus technika, mivel a szoftver futtatását igényli.
Átvizsgálási technikák: Programátvizsgálások, Automatikus forráskód elemzés, Formális verifikáció. A statikus technikával csak a program és a specifikáció közötti megfeleloségét tudja vizsgálni. Így nem vizsgálható pl.: a megbízhatóság, teljesítmény. A tesztelés nem nélkülözheto.
A tesztelések fajtái: Hiányosságtesztelés (célja a program és a specifikáció között meglévo hiányosságok felderítése. Célirányos tervezett vizsgálat.). Statisztikai tesztelés (célja a program teljesítményének, megbízhatóságának vizsgálata. Tükrözniük kell a valós felhasználói bemeneteket és azok gyakoriságát. Becslés adható a megbízhatóságra a muködés közben mért hibák alapján, illetve a teljesítményre a statisztikai tesztadatok feldolgozásánál rögzített paraméterek - pl.: futási ido, válaszido, stb - alapján)

30.1. A „belövési” folyamat:

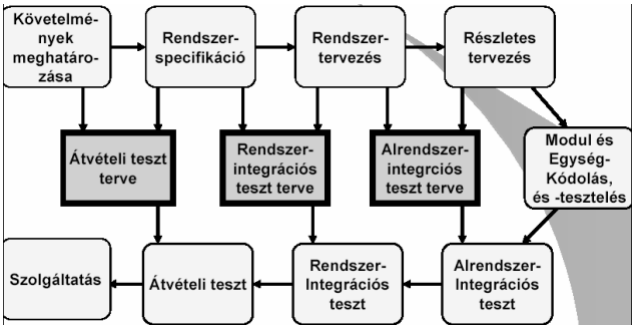


A V & V az a folyamat, amelyik megállapítja, hogy a szoftverben vannak-e hiányosságok. A belövés az a folyamat, amely behatárolja és kijavítja ezeket a hiányosságokat. Jellemzok: Igen magas fokú nyelvi környezet ismeretet igényel, Nehezen algoritmizálható, szabályok nehezen adhatók meg. Speciális célszoftverek segíthetik a hiba megtalálását. A hiba behatárolását követoen kijavítjuk, majd rendszert újra validáljuk. Ez lényegében a tesztek újbóli megismétlését jelent, amit szokás regressziós tesztelésnek nevezni. A regressziós tesztelés célja annak vizsgálata, hogy a hiba kijavítása során nem követtünk-e el újabb hibát. A regressziós tesztelés során elvben az összes tesztet megismételjük minden javítási lépés után. A gyakorlatban ez igen nagy ráfordítást igényelne, így csak a módosított rész és annak függoségeihez tartozó teszteseteket ismételjük meg. (Alapos teszt terv és dokumentáció szükséges ennek kivitelezéséhez)

30.2. V & V tervezése:

A V&V igen költséges folyamat, szükséges gondos megtervezése. A tesztterv jellemzoi: Áttekinto képet ad a rendszer tesztelési folyamatról. Rendelkezik a felhasználható eroforrásokról, Kötelezoen betartandó és irányadó szabványokat és eljárásokat tartalmaz a tesztelésre vonatkozóan, Ellenorzo listákat tartalmaz a tesztelok számára, információt ad a fejlesztést és az implementálást végzok számára is.

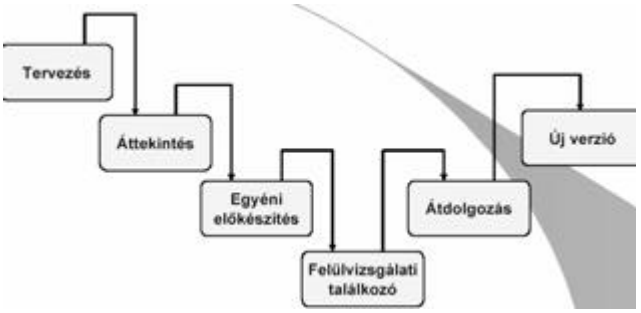
30.3. Szoftverek átvizsgálása



A szoftver átvizsgáláshoz nem kell programot futtatni, így a fejlesztés korai szakaszában, a programok implementációja elott, verifikációs technikaként használható. Azonban kiterjedhetnek a szoftverfolyamat bármely dokumentumára: követelmény specifikáció, vázlatos és részletes tervekre, adattervekre, teszttervekre, stb. Mit vizsgálhatunk? Rendszermodellt, Specifikációt, Magas szintu nyelven megfogalmazott metakódot, Bármilyen dokumentumot, mely a szoftverfolyamat része.
A szoftverátvizsgálás jellemzoi: Sokkal olcsóbb a tesztelésnél. Az egyes programelemek elszigetelten vizsgálhatók. Elhanyagolhatók a hibák kölcsönhatásai. A hiba detektálása közvetlenül történik (nem valamilyen rossz értékbol derül ki a hiba). Egyes vizsgálatok szerint az átvizsgálás nem csak olcsóbb, hanem hatékonyabb is, mint a tesztelés. Fagan szerint egy program hibáinak 60 százaléka felderítheto átvizsgálással. Az átvizsgálás során a minoség és a szabványoknak való megfeleloség is ellenorizheto. A dinamikus viselkedés vizsgálatára, a megbízhatóság becslésére nem alkalmas az átvizsgálás. Muködési hatékonyság, szuk keresztmetszet nem határozható meg segítségével. A

felhasználói felület validálása nem végezheto el az átvizsgálás során. Rendszerszinten a bonyolultság miatt gyakorlatilag nem alkalmazható.
Mi indokolja az átvizsgálások hatékonyságát? Egyetlen átvizsgálással több különbözo hiba is felderítheto, míg a tesztelés általában tesztenként csak egy hibát hoz ki. A felhalmozódott tapasztalatok segítenek a kritikus részek ellenorzésében. (Az átvizsgálást végzo, rutinos szakember „tudja, hogy mit kell nézni")
Kritikus programszerkezetek: Pointeres kezelés, több kilépési ponttal rendelkezo ciklusok, bonyolult vezérlési szerkezetek Az átvizsgálás nem helyettesíti a tesztelést, hanem kiegészíti azt!

30.4. Program átvizsgálások



A programátvizsgálás lényege: egy különbözo háttérrel rendelkezo tagokból álló csoport a program forráskódját gondosan, sorról, sorra átnézi. Az átvizsgálás célja a hiányosságok, hibák felderítése. Az átvizsgálás egy formális folyamat, melyet egy, legalább 4 fobol álló team végez. Különbözo szerepkörök definiáltak a csoportban. Fagan szerint: **Szerzo** (a program vagy dokumentum elkészítésért és a hibák kijavításáért felelos) **Olvasó** (felolvassa, illetve elmagyarázza a kódot vagy dokumentumot) Tesztelo (a tesztelés szempontjából vizsgálja a kódot) Moderátor (az átvizsgálás folyamatát szervezi, vezeti).
A programátvizsgálás hatékonyságának jellemzoi: Az áttekinto szakasz során kb. 500 LOC/óra tekintheto át. Az egyéni elokészület során kb. 125 LOC/óra vizsgálható meg. A találkozó során 90-125 LOC/óra vizsgálható át. Az AT&T-nél gyujtött adatok szerint: Egy 4 fos csapat 100 LOC-ot kb 1 embernapnyi ráfordítással vizsgál át. Ha átvizsgálás helyett tesztelnének, akkor több mint dupla ráfordításra lenne szükség.

30.5. Automatizált statikus elemzés

Az automatizált statikus programelemzo szoftverek a program forráskódját analizálva derítenek fel hibákat, hiányosságokat. (pl: nem inicializált változók, nem használt változók, a tartományon túlmutató adatértékek). Az automatizált statikus elemzovel felismerheto hibák: Adathibák (inicializálás, deklarálás, rossz értékadás, stb.). Vezérlési hibák (hibás vezérlési szerkezetek, ciklusok, nem hívott függvények, eljárások, stb.). Input/Output hibák (a típusnak nem megfelelo I/O form.). Interfészhibák (paraméterek típusütközése, stb.). Tárkezelési hibák. (védett területre írás, stb.). A statikus elemzés szakaszai: 1. **A vezérlés folyamatának elemzése** (többszörös be, illetve kilépési ponttal rendelkezo ciklusok vizsgálata, nem használt kódrészek felderítése, stb.). 2. **Az adathasználat elemzése** (inicializálás nélkül használt változók, kétszer ír egy változóba, de senki nem olvassa, deklarált, de fel nem használt változók felderítése). 3. **Interfészelemzés** (gyengén típusos nyelveknél használható, pl.: C, a függvény és eljárás deklarációval, paraméterekkel, visszatérési értékekkel kapcsolatos hibákat vizsgálja) 4. **Az információáramlás elemzése** (a bemeno és kimeno változók közötti függéseket deríti fel, a programban használt értékek származtatását gyujti ki, ami segítséget nyújthat az átvizsgálásokhoz.) 5. **Útvonalelemzés** (azonosítja a program összes lehetséges végrehajtási útvonalát, és kigyujti, az ezeken az útvonalakon végrehajtott utasításokat.).
Az automatizált statikus elemzokre különösen azon nyelveknél van szükség, amelyek gyengén típusosak, vagy a fordítójuk kevés ellenorzést végez. Nem helyettesíti az átvizsgálást, illetve a tesztelést, csak kiegészíti!