Tartalomjegyzék

Szoftvertechnológia

A szoftver egy rendszer azon része, amely nem kézzel fogható, azonban perces gyakorisággal találkozunk vele. Az informatikai rendszerek olyan szinten elterjedtek, hogy a legegyszerűbb, napi használatos eszközeinkben is megtalálhatók. Ezek az eszközök mindegyike rendelkezik szoftver elemekkel. A szoftver használat kikerülhetetlen és kritikus sikertényező.

Akkor mi is az a szoftver?

Definíció: szoftver az adatok és a programok összessége.

Számos esetben a közhiedelem a szoftver alatt csak a programokat érti, azonban a számítógép működéshez a programok általában nem elegendőek, hanem adatokra is szükség van. Az adat ebben az esetben nem csak azokat az állományokat fedi le, amelyeken a számítógép dolgozik, hanem azokat az adatelemeket is, amelyek a rendszer állapotát határozzák meg, illetőleg írják le.

Definiáljuk az adat és a program fogalmát is!

Definíció: az adat az információ fogalmi egysége, tények, fogalmak, jelenségek absztrakt formája.

Miért absztrakt? Mivel az adat a számítógép memóriájában van valamilyen kódolással tárolva, csak egy tükörképe annak, amit a valóságban jelent. Nem adunk meg mellé mértékegységet (bár jelezhetjük) és önmagában haszontalan.

Vegyük példának a szoba hőmérsékletét, jelen esetben ez az érték 23 Celsius fok. Ezt ábrázolhatjuk a memóriában ASCII értékként két bájton: 32_H 33_H, vagy egyetlen bájton (mert elfér) 17_H[1]. Nem adtuk meg mellé, hogy ez milyen mértékegységben került ábrázolásra, mert a kérdéses adatról tudtuk, hogy mit ábrázol.

Definíció: program az a véges számú utasításból álló sorozat, amely a számítógépet az adott feladatnak megfelelően vezérli.

1 Mindkét példát hexadecimálisan adtuk meg.

1. Szoftverek osztályozása

A szoftvereket osztályozhatjuk a következő szempontok szerint:

- felhasználás szerinti osztályozás,
- megbízhatósági igény szerinti osztályozás,
- futtatói – alkalmazói környezet szerinti osztályozás,
- szerzői jogok szerinti osztályozás.

Felhasználás szerinti osztályozás

Egy kérdéses szoftver felhasználás szerint lehet rendszerszoftver, fejlesztő rendszerek és lehet felhasználói szoftver.

A rendszerszoftver olyan program, illetve az esetlegesen ehhez tartozó adathalmaz, amely kezeli működteti a számítógépet[2]. Ide tartoznak az operációs rendszerek, monitor programok, eszköz kezelők és a rendszer kezelését lehetővé tevő segédprogramok.

A felhasználói szoftverek olyan programok és adatok, amelyek lehetővé teszik, hogy a felhasználó a gépet előre meghatározott célra használja, például szövegszerkesztésre.

A fejlesztő rendszerek olyan szoftver csomagok, amelyek lehetővé teszik, hogy szakemberek különböző programokat hozzanak létre[3].

1.1 Osztályozás felhasználás szerint

1.1.1. Rendszerszoftverek

A rendszerszoftverek a következő csoportokba sorolhatók:

- kisebb intelligenciával rendelkező eszközök működtetését biztosító programok és

2 Itt folyamatosan számítógépről beszélünk, de itt nem okvetlenül a klasszikus PC jellegű gépet kell érteni, hanem minden programozható eszközt, amely akár lehet egy CPLD, vagy FPGA is.

3 Az irodalom a fejlesztői rendszereket sokszor a rendszer szoftverek közé és néhány esetben a felhasználói programok közé sorolja. Véleményünk szerint ezeket a szoftver csomagokat célszerű külön kezelni.

eljárás csomagok (számos esetben ezeket firmware-nek nevezik),

- operációs rendszerek.

Sok olyan eszköz van forgalomban, amelyek rejtetten valamilyen szoftvert tartalmaznak. Tipikus példa erre akár egy dallam kapucsengő, vagy egy kombinációs ajtózár. Ezek az eszközök sok esetben diszkrét áramkörökből is felépülhetnek, de sokkal egyszerűbb, gyorsabb és olcsóbb valamilyen programozható eszközt – gyakorlatilag mikrokontrollert – beépíteni. Így lehetővé válik az elkészült termék igény szerinti változtatása is. Számos esetben előfordul, hogy egyazon hardverre épülő termékek lényegesen különböző célokra használhatók köszönhetően annak, hogy a programozhatóak. Ezek a programok rendszer programok. Sokszor előfordul, hogy a gyártók felépítenek egy szoftver vázat, vagyis elkészítenek alapvető funkciókat, amelyet az adott igény szerint használnak fel. Ez a filozófia már elég közel áll az operációs rendszerekhez (lásd alább).

A számítástechnika fejlődésével egyre nyilvánvalóbbá vált az a tény, hogy a számítógépek kezelését egyszerűsíteni kell, illetve lehetővé kell tenni, hogy általános felhasználható is képes legyen a gép használatára. További cél volt az is, hogy a gépet a célszerűség határáig ki lehessen használni. Ez az irányzat vezetett az operációs rendszerek kifejlesztéséhez.

Definíció: az operációs rendszer egy olyan programcsomag, amely kapcsolatot tart a felhasználóval és kezeli a rendszer erőforrásait.

Az első funkció nem igényel magyarázatot. Az erőforrást azonban definiáljuk.

Definíció: erőforrásnak nevezzük mindazon hardver és szoftver elemet, amely a számítógép működéshez szükséges.

Hardver erőforrások azok a szűkebben vett elemek, amelyek valóban a számítógép számítógép szerű[4] működéséhez szükségesek. Ezt a teljesség igénye nélkül processzorok, illetve processzor magok száma, memória, háttértár, stb.Szoftver erőforrások azok a szoftver elemek, amelyek a gép működéshez szükségesek, ilyenek a speciális eszközmeghajtók és a kezelő programok.

4 Elég nehéz meghúzni a határt, hogy az adott elem a számítógép erőforrása, hiszen akár a tápegység is ide sorolható. És igazából ez jogos is, hiszen egy szerverben a háttértárak tápellátása is elengedhetetlen. De ezeket már nem „szokás” az erőforrások közé sorolni.

1.1.2. Felhasználói szoftverek

A felhasználói szoftverek választéka nagyon szerteágazó. Ide tartoznak a közismert irodai programok, a különböző általános célú segédprogramok (pl. kalkulátor, rajzprogramok, zenei programok, otthoni alkalmazások), a játékok és néhány speciális szoftver, amelyekről a későbbiekben szó esik még.

1.1.3. Fejlesztő rendszerek

A fejlesztő rendszerek segítségével lehetőség van új szoftverek létrehozására, tesztelésére és üzembe helyezésére. A fejlesztő rendszerek bonyolultsága az egyszerű úgynevezett tool-chain -ektől a nagy bonyolultságú integrált fejlesztő környezetekig terjed.

Alapvető elemeik:

- szerkesztést biztosító felület,
- fordító program (ha szükséges),
- szimulátorok,
- nyomkövetést biztosító programok,
- amennyiben a fejlesztés idegen platformra történik a telepítéshez szükséges programok.

A fejlesztő rendszerekkel is részletesen foglalkozunk a jegyzet hátralevő részében.

1.2. Osztályozás megbízhatósági igény szerint

Az előzőekben már említettük, hogy a szoftver kritikus sikertényező, illetve alkalmazása elkerületlen. Így az élet minden területén találkozunk vele. De azt is könnyű belátni, hogy teljesen más követelményeket támasztunk megbízhatósági szempontból egy egyszerű játékprogram és egy vasúti biztosító berendezés szoftverrendszerével szemben.

Megbízhatósági szempontból a szoftvereket öt kategóriába sorolják és ezen belül több szempontot vizsgálnak abból az aspektusból, hogy milyen kockázatot jelent a szoftver

leállása, vagy hibás működése.

A szempontok:

- emberi sérülés és életveszély,
- környezeti kockázat,
- pénzügyi kockázat,
- erkölcsi kockázat.

A kategóriák:

5. Veszélybiztos rendszerek. Tömeges életveszély, környezeti katasztrófa veszélye, pénzügyi katasztrófa bekövetkezése és hatalmas erkölcsi kár.

   Ilyen rendszerek a vasúti biztosító berendezések, a repülésirányító szoftverek, a nem nyomott vizes atomerőművek, egyes vegyipari üzemek, orvosi automaták és robotok szoftverei.

4. Működésbiztos rendszerek. Emberi sérülésveszély lehetősége, komoly környezet szennyezés lehetősége, nagymértékű pénzügyi veszteség és nagymértékű erkölcsi kár.

   Ilyen rendszerek, a nyomott vizes atomerőművek, a gépkocsi fedélzeti rendszerek, az erőművi rendszerek, a közmű rendszerek.

3. Nagy megbízhatóságú rendszerek. Emberi sérülésveszély lehetséges, de nem valószínű, környezet szennyezés nem valószínű, a pénzügyi kockázat nagy, az erkölcsi kár nagy.

   Ilyen rendszerek a nagy adatbázisok és a nagy teljesítményű CAD rendszerek.

2. Általános megbízhatóságú rendszerek. Emberi sérülésveszély nincs, környezeti veszély nincs, pénzügyi kockázat közepes vagy alacsony, erkölcsi kár lehetséges.

   Ilyen rendszerekre legjobb példák az irodai rendszerek.

1. Megbízhatósági szempontból lényegtelen rendszerek. Emberi sérülésveszély nincs, környezeti veszély nincs, pénzügyi kár nincs, erkölcsi kár lehetséges.

Ilyen szoftverek lehetnek például a játékprogramok.

A 4. és 5. kategóriákat a magyar terminológiában biztonságkritikus szoftvereknek is nevezik.

1.3. Futtatói környezet szerinti osztályozás

A programok minden esetben valamilyen hardveren futnak. A hardver az esetek nagy többségében meghatározza a rajta futó szoftver jellemzőit. Ezzel a kijelentéssel azonban ellentétes az utóbbi idők azon törekvése, hogy a lehetőségekhez mérten a lehető legjobban elvonatkoztassunk a futtató hardvertől és a futtató környezettől és csak a feladatra koncentráljunk platform függetlenül.

Arról nem szabad elfeledkezni, hogy a futó program természetesen egy, vagy több processzoron fut. Ez az eszköz csak és kizárólag azt a bináris kódot képes lefuttatni, amely az adat, vagy utasításbuszán megjelenik.

A futtató környezet szerint az első felosztási lehetőség az, hogy a program önállóan fut a hardveren, vagy kap valamilyen operációs rendszer jellegű támogatást.

Vegyünk példának egy egyszerű klímavezérlést, ez megvalósítható:

- önállóan futó, úgynevezett natív programmal,
- valamilyen kis operációs rendszer támogatással, pl. FreeRTOS,
- valamilyen nagyobb teljesítményű hardveren, amely képes általános célú operációs rendszert futtatni, pl. Raspberry PI.

Az első megoldás pontosan a feladatra optimalizált program lehet, viszont mindenről nekünk kell gondoskodnunk.

A második megoldásban sokkal inkább koncentrálhatunk a feladatra, mert az ütemezést, időzítést, stb. a FreeRTOS elvégzi.

Az előző két kategóriában a klasszikus értelemben vett kezelői felület nem követelmény. Nagyon sokszor a felületet néhány LED és nyomógomb jelenti.

A harmadik esetben még feljebb léphetünk, mert sokkal szélesebb körű szolgáltatásokat

vehetünk figyelembe. Például a feladatot szkript nyelven is megoldhatjuk.

Ez alapján a futtató környezet alapján történő osztályozás célszerűen a következő lehet:

- kis hardveren történő futás,
- nagy gépen történő futás[5],
- önállóan program,
- program operációs rendszerrel.

Ezek a kategóriák bizonyos mértékig átfedhetik egymást.

1.4. Szoftverek csoportosítása szerzői jogok alapján

A szoftvereket csoportosíthatjuk a szerzői jogok alapján. Ezek a jogok azt szabályozzák, hogy mit tehetünk meg az adott szoftverrel.

- kereskedelmi programok: kereskedelmi céllal készülnek, pénzbe kerülnek, felhasználó lehetőségei erősen behatároltak.,
- freeware szoftverek: szabadon felhasználhatók és terjeszthetők,
- shareware programok ingyenesen beszerezhetők és terjeszthetők, de gyakran nem működnek teljesen (a teljes programért fizetni kell, amit ha nem teszünk meg), előfordulhat, hogy bizonyos idő vagy bizonyos számú indítás után nem lesz használható,
- trial programok kipróbálásra kiadottak. Hasonlóak a shareware programokhoz, de nem terjeszthetőek szabadon.
- fél szabad szoftverek: kereskedelmiek, de adott felhasználási célra vagy adott felhasználói csoportnak olcsóbbak,
- szabad szoftverek: az ingyenesnél sokkal többet kap a felhasználó és több joggal rendelkezik: ingyenesen beszerezhető, szabadon használható, terjeszthető,

5 Ide soroljuk a PC-ket is, illetve az alkalmazás processzorokkal (p. Arm Cortex A sorozat) rendelkező hardvereket is.

módosítható.

2. Felhasználói szoftverek

Az előzőekben már röviden említettük a felhasználói programokat, ebben a fejezetben részletesebben foglalkozunk ezekkel.

2.1. Irodai alkalmazások

A felhasználói programok egyik leggyakoribb típusa az irodai alkalmazások. Ezek nem egyetlen programként jelennek meg, hanem csomagként készítik ezeket.

Ilyen rendszerek például a LibreOffice, MS Office, KOffice.

A csomag általában a következő komponenseket tartalmazza:

- szövegszerkesztő,
- táblázatkezelő,
- prezentáció készítő.

Nem minden irodai alkalmazás része az:

- adatbázis kezelő[6],
- rajzprogram,
- kalkulátor,
- különböző adatok, képek, animációk.

Az irodai rendszerek szinte kivétel nélkül automatizálhatók, ami azt jelenti, hogy valamilyen programozási nyelven programozhatók. Ilyen programozási nyelv a Visual BASIC, a JAVA és a Python.

6 Akár az adatbázis kezelőt, akár a rajzprogramot tekintjük. Ezek az alkalmazások nem tekinthetők professzionális alkalmazásnak, de irodai célra megfelelők.

A szövegszerkesztő komponens egy dokumentum típusú szövegszerkesztő. Ez azt jelenti, hogy az elkészült állomány nem pontosan azt tartalmazza, amit a képernyőn látunk, hanem más információkat is.

Ezek az információk lehetnek formázási adatok, pl. szöveg méret, betű típus, paragrafusok zárása, stb. és lehetnek az szerkesztés menetével kapcsolatos információk.

A szöveg szerkesztőknél már alapkövetelmény úgynevezett WYSIWYG jelleg.

Definíció: a WYSIWYG (what you see is what you get), amit látsz, azt kapod. A képernyőn látható szerkesztési kép megegyezik a nyomtatandó formátummal.

A WYSIWYG nagy előnye, hogy a képernyőn is jól olvasható, tehát nem kell kinyomtatnunk az állományt a kényelmes olvasáshoz, hanem rögtön a monitoron olvashatjuk a dokumentumot jó minőségben.

Hátránya az, hogy bizonyos kompromisszumokat kell kötni. Az első, hogy a futtató hardver legyen kellően erős, a második, hogy a gép folyton helyettünk akar gondolkodni, ami néha nagyon idegesítő. Ez az oka, hogy számos más rendszer is elterjedt és bár ezek a rendszerek rettenetesen bonyolultnak tűnnek és sok lexikális tudást igényelnek mégsem kopnak ki a számítástechnikából[7].

A táblázatzekekő komponens egyfajta számoló táblát biztosít a felhasználó számára, amely segítségével vegyes adatokat lehet feldolgozni. Ilyen táblázat kezelő a LibreOffice Calc, az MS Excel, KSpread.

Az adatok lehetnek numerikus, szöveges, dátum, idő, pénzügyi és felhasználó által definiált formátumúak.

A műveletek cellánként és cella csoportonként is elvégezhetők. Az alapvető műveletek mellett sok függvény is rendelkezésre áll csoportokra bontva. A teljesség igénye nélkül ilyen függvénycsoportok a matematikai, statisztikai, logikai, dátum és idő, adatbázis, stb. függvények.

A táblázatkezelők a táblázatokban szereplő adatokból képesek grafikonokat készíteni, amelyek a táblázatban elhelyezhetők.

7 Ilyen rendszer a LaTeX.

Ezt a funkciót a szövegszerkesztők is tudják, azonban azok grafikon kezelése az esetek többségében nem interaktív. Tehát az adatok változását nem azonnal viszi át a grafikonra, hanem külön utasítást kell adnunk.

A táblázat kezelők is - hasonlóan a szövegszerkesztőkhöz - programozhatók a környezetre jellemző programozási nyelven.

A prezentációs komponens lehetővé teszi előadások készítését. Legismertebb képviselői: LibreOffice Impress, Power Point, Prezi[8].

A prezentáció készítők a hagyományos írásvetítő fóliákkal készült prezentációkat helyettesítik. Természetesen mivel számítógépen futó szoftverek nagyon sok plusz szolgáltatást nyújtanak. Ilyen szolgáltatások az áttűnések, különböző objektumok beillesztése, interakciós elemek beépítése a prezentációba. Kicsit részletesebben, beilleszthető kép, táblázat, videó, akusztikus elem, illetve gombok, rádiógombok, stb.

A fenti komponensek szinte kivétel nélkül megtalálhatók minden irodai rendszerben. A következő elemekre ez nem minden esetben igaz.

Az adatbázis komponens általában egy relációs adatbázis kezelést lehetővé tevő szoftver. Tipikus képviselője a LibreOffice Base, MS Access. Ezek az adatbázis kezelők számos adatbázis formátumot támogatnak, de kivétel nélkül ismerik az SQL adatbázis kezelő nyelvet.

Ezek a komponensek a felhasználó számára biztosítanak egy grafikus tervezői felületet, de lehetővé teszik a szöveges programozást is.

A rajz komponens egyes esetekben beépítésre került más komponensekbe, de néhány megvalósításban különálló szoftver elem. Ez a komponens kivétel nélkül vektorgrafikus rajzolást tesz lehetővé. Szolgáltatásai alapszintű rajzolást tesznek lehetővé.

2.2. Egyéb szövegkezelő alkalmazások

A gyakorlatban számos szövegformátum létezik, a teljesség igénye nélkül röviden ismertetjük ezeket.

PDF az Adobe Systems által kifejlesztett, dokumentumok tárolására alkalmas formátum.

8 A Prezi nem egy irodai rendszer része.

Képes szöveg, ábra és kép tárolására alkalmazás, eszköz és platform független formában. A PDF formátum lehetővé teszi, hogy interaktív elemeket is elhelyezhessünk a dokumentumban, így például lehetőség nyílik kérdőívek megvalósítására is.

A PDF formátum egy nyílt szabvány, tehát bárki jogdíjmentesen írhat alkalmazásokat, amelyek PDF formátumú dokumentumokat olvasnak vagy írnak.

Tipikus kezelő szoftverek: Adobe Acrobat, Adobe Acrobat Reader, Evince, stb..

PostScript egy gyakori, platform és eszköz független nyelv, amelyben az elemek között vannak vonalak, karakterek, különböző alakzatok és ábrák is, de bittérképes formátum is beilleszthető. Ekkor azonban a fájl méret jelentősen megnő. A PostScript formátum leginkább a nyomdatechnikában használatos és ott szabvánnyá is vált. Érdekes jellemzője, hogy a PostScript fájl normál ASCII karakteres formátumú.

A nyomtatók nagy többsége közvetlenül képes a PostScript formátumot értelmezni.

Tipikus kezelő szoftverei Evince, gv.

DVI (Device Independent, magyarul eszközfüggetlen) fájlformátum, A fájlformátum egy dokumentum vizuális tulajdonságait írja le például font, margók, de nem tartalmazza az esetleges beszúrt médiafájlokat, fontokat közvetlenül úgy, hogy ne függjön semmilyen megjelenítő eszköztől. A fájl tipikusan egy másik program bemenete, ami vizuálisan megjeleníthető alakra hozza. DVI driver lehet egy másik fájlformátumba konvertáló program is.

Tipikus megjelenítő alkalmazások: Evince, xdvi.

DJVU formátumot főleg szkennelt dokumentumok tárolására tervezték. A PDF-fel szemben a beolvasott lapokat általában kisebb fájlméret mellett jobb minőségben képes eltárolni.

Tipikus megjelenítő alkalmazás: djview.

mobi, epub, prc, azw ezek az úgynevezett e-book formátumok. Nem okvetlenül személyi számítógépekre vannak kitalálva, de természetesen a személyi számítógépek tudják

olvasni ezeket.

Tipikus alkalmazás: calibre.

2.3. Internethez kapcsolódó felhasználói programok

Ide tartoznak a böngésző programok, amelyekkel különböző web-es felületek érhetők el. Ezeknek a programoknak nagy többsége nagyfokú interakcióra képes és képes beágyazott alkalmazásokat kezelni. Ilyen beágyazott alkalmazás videó állományok lejátszása, hangkezelés, stb..

Egyre elterjedtebbek azok a játékprogramok, amelyek böngészőben játszhatók. Ehhez természetesen kellően erős számítógépre és gyors hálózat hozzáférésre van szükség.

Ilyen szoftverek: Firefox, Opera, Chromium-browser, Internet Explorer.

A következő jelentős kategória a levelező programok. Ezek szolgáltatásai az alapvető levél írás, olvasás, továbbítás mellett állományok küldése szabványos hálózati protokollok használatával. Hozzá kell tennünk, hogy számos levelezői felület elérhető már böngészőn keresztül, de számos dedikált levelező alkalmazás létezik. Ilyen alkalmazások például: Thunderbird, Outlook (2003-óta az MS Office része), Send-blaster.

FTP kliensek. Ezek a programok gyakorlatilag kimentek a divatból, manapság az FTP felületek web böngészőkön keresztül elérhetők. Egyetlen terület van, ahol beágyazott alkalmazásként előfordulnak, a felület kezelők, mint például a Mid-night Commander és a Krusader.

Csevegő programok. A helyzet hasonló az előző kategóriával, a web-böngészőkön belül adott web-oldalon egy beágyazott alkalmazásként csevegni.

2.4. Egyéb felhasználói szoftverek

Ide tartoznak azok a szoftverek, amelyek nem tekinthetők professzionális felhasználói termékeknek, de a felhasználók nagy része sűrűn használja. Ebbe a kategóriába tartoznak a zenelejátszók, a videó lejátszók, az egyszerű rajzprogramok, egyszerű szerkesztő programok és ide tartoznak a játékok is.

A játékok akár külön kategóriát is képezhetnének, hiszen egy általános célú felhasználói programhoz képest igen komoly erőforrás igénnyel rendelkezhetnek[9].

Professzionális felhasználói programok

Ebbe a kategóriába azok a szoftverek tartoznak, amelyek segítségével üzleti célra is lehet komoly termékeket előállítani. Ezek lehetnek:

- Rajzprogramok. Ezek kizárólag vektorgrafikus alkalmazások rendkívül széles szolgáltatás palettával. Ilyen program például Inkscape, Corel Draw.
- Képszerkesztő programok. Ezek a szoftverek lehetővé teszik képek utólagos szerkesztését. Ezek szintén számos szolgáltatással rendelkeznek. Tipikus képviselője: a GIMP és a PhotoShop.
- Videó szerkesztő programok. Lehetővé teszik videofelvételek szerkesztését. Ilyen program például a Openshot, a Kino, az Adobe Premier.
- Audió szerkesztő programok. Hang állományok szerkesztése lehetséges segítségükkel. Ilyen program például az Audacity, a Sound Forge.
- Matematikai programok. Különböző matematikai és ehhez haszonló feladatok megoldására szolgáló szoftver csomagok. Tipikus képviselője a Matlab, az Octave, a SciLab.
- CAD szoftverek. Ezekről a következő fejezetben részletesen kitérünk.
- Speciális területek szoftverei. Számos terület használ a területre jellemző szoftvereket. A teljesség igénye nélkül említsünk meg néhányat. Orvostechnikai felhasználások, például MR képfeldolgozás, repülésirányítás nyomkövető szoftverei, meteorológia mérésadatgyűjtő szoftverei.

2.5. CAE szoftverek

A CAE rendszerek képviselői a CAD, CAM, CASE rendszerek.

9 A játékok manapság szinte az összes lehetséges kategóriában megjelennek. Egyre jobban tejednek az Internetes alapokon működő játékprogramok.

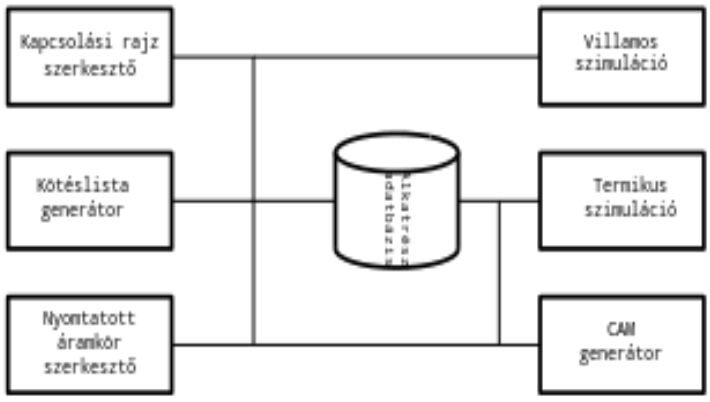
2.5.1. CAD rendszerek

A CAD a Computer Aided Design rövidítése. Ezek a rendszerek olyan szoftver csomagok, amelyek a mérnöki tervezést támogatják. A CAD rendszerek az adott mérnöki tevékenységre fókuszálnak, tehát vannak villamosipari, gépészeti, épületgépészeti, építészeti változatok. A kezdeti időkben ezek a rendszerek élesen elkülönültek egymástól, de mostanában egyre több átfedés található a különböző területek között.

A következő részben a CAD rendszerek felépítésével foglalkozunk.

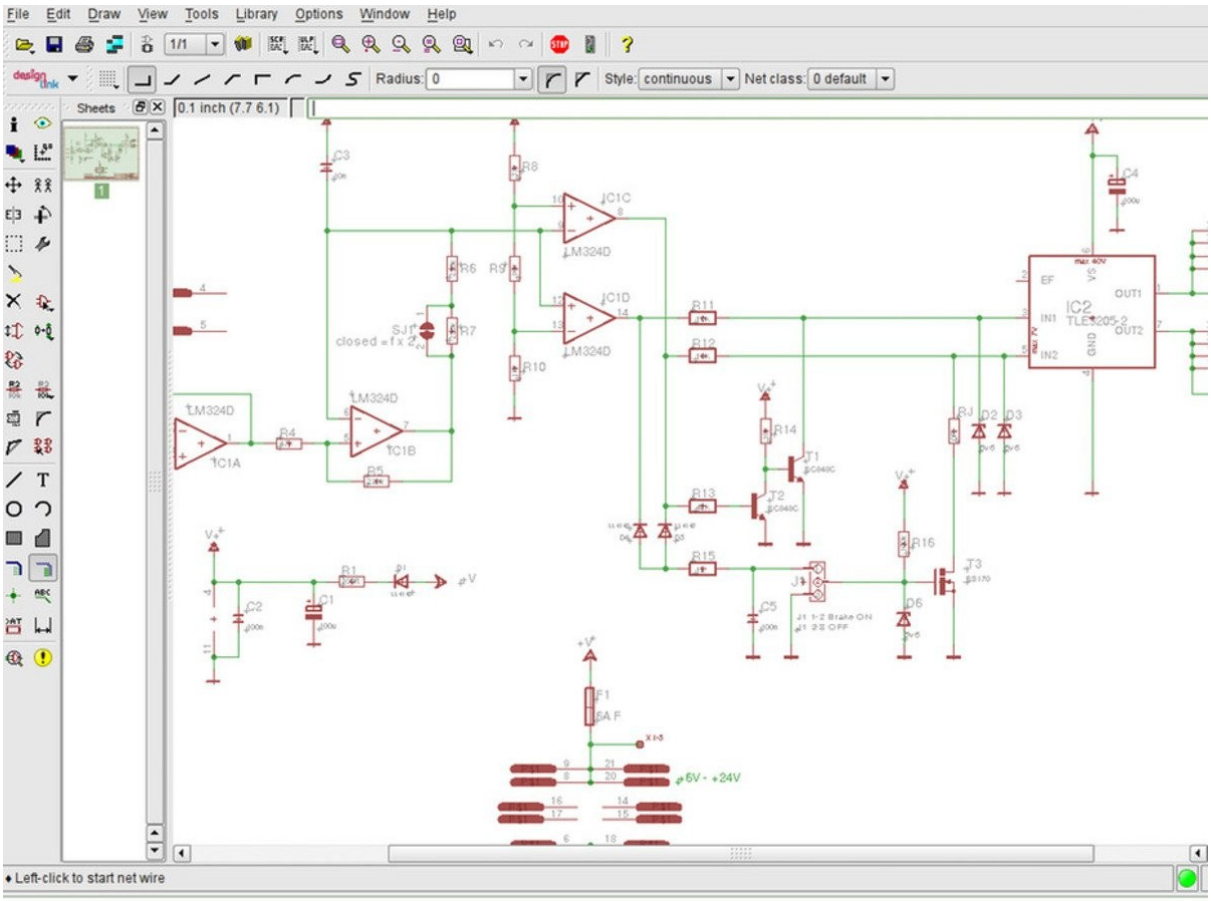
2.5.1.1. Villamos CAD rendszerek

A villamos CAD rendszerek a teljes termékfejlesztést támogatják a kiindulástól a kivitelezésig. A következő ábra mutatja egy általános villamos tervezőrendszer felépítését.



1. ábra: Villamos CAD rendszer

A kapcsolási rajz készítő modullal az elkészítendő áramkör kapcsolási rajzát lehet elkészíteni. Ebben a modulban az adatbázisban szereplő áramköri elemek kapcsolási rajz szimbólumaival és egyéb rajz elemek, mint például vonalak, átkötések segítségével a tervező a teljes kapcsolási dokumentációt el tudja készíteni. Manapság már elvárás, hogy a dokumentáció lapokra bontható legyen, így a nagyméretű készülékek dokumentációi is egyetlen egységként kezelhetők.

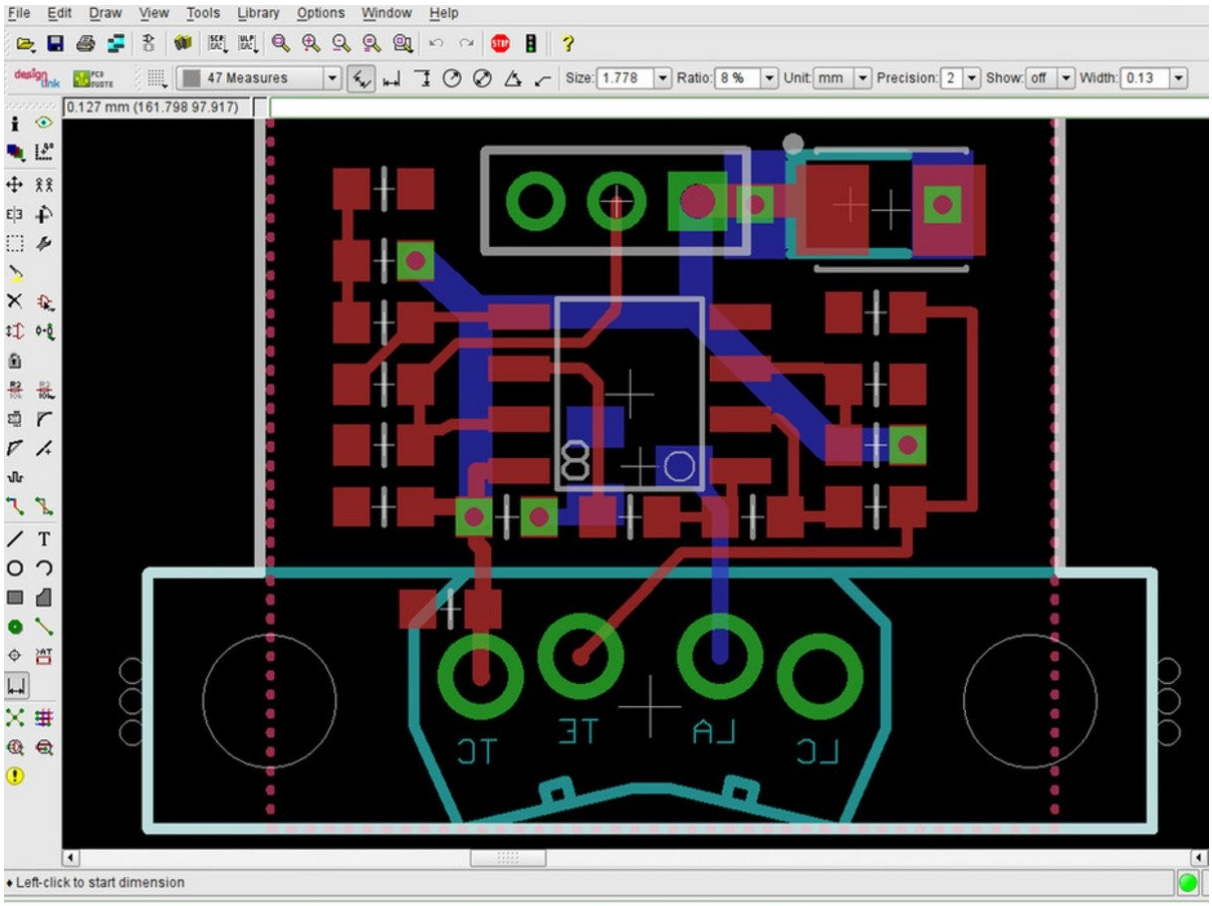


2. ábra: Kapcsolási rajz szerkesztő (forrás Internet)

A kötéslista generátor az elkészült kapcsolási rajzból egy kötéslistát állít elő. Az előzőleg megrajzolt kapcsolási dokumentációból ez a modul előállítja a kötéslistát. A kötéslista egy olvasható szöveges állomány további feldolgozásra.

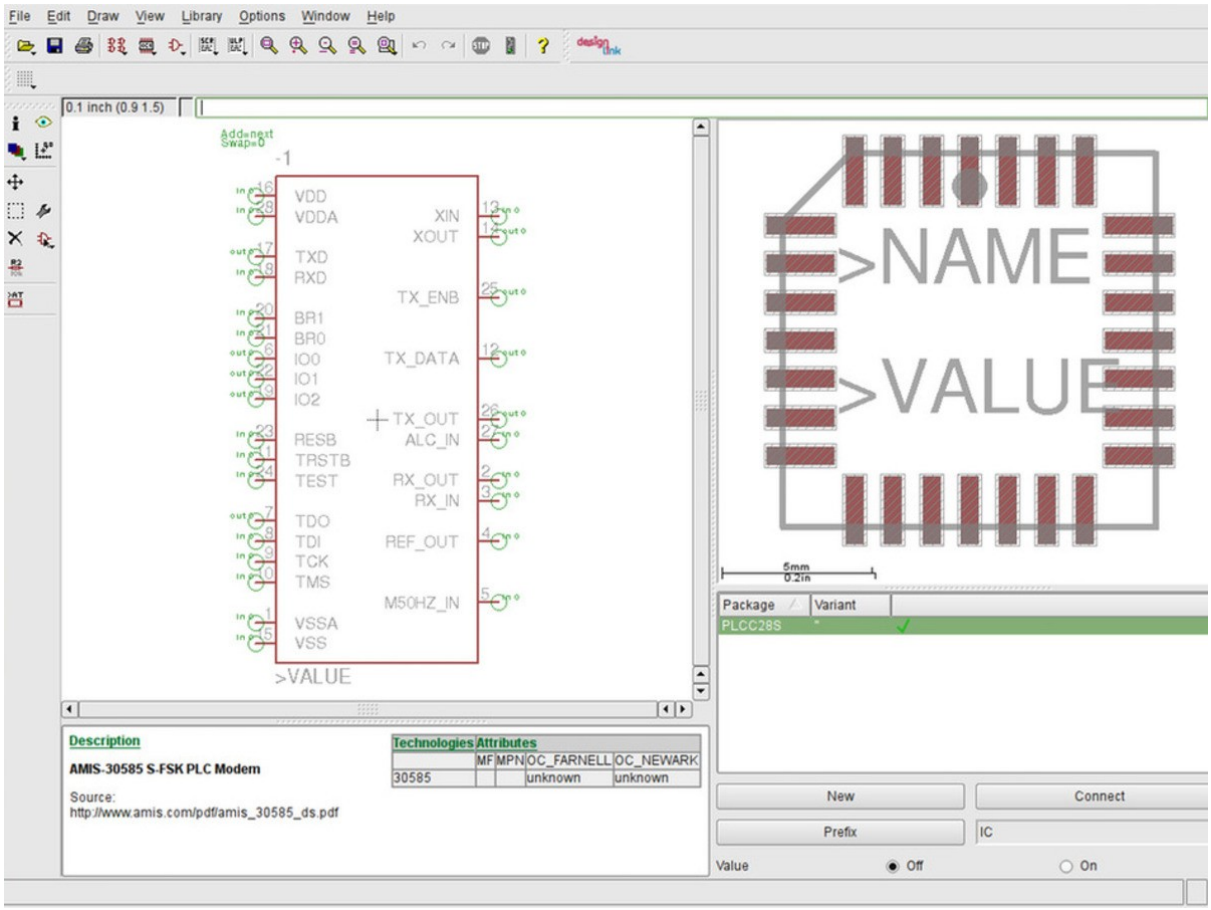
A nyomtatott áramkör szerkesztő a kötéslistából a nyomtatott áramkört készít. Ez a modul támogatja a kézzel történő és az automata huzalozás készítést is. Az automata huzalozást készítő program az előre beállított paraméterek alapján, mint vezetősáv vastagság, szigetelési távolság, furatátmérő, rétegek száma, stb. elkészíti. A tervező ezt később módosíthatja.

A nyomtatott áramkör szerkesztő a tervező által elkészített nyomtatási rajzot ellenőrzi is és az előző beállított paraméterek megsértése esetén üzenetet küld.



3. ábra: Nyomtatott áramkör szerkesztő (forrás Internet)

Az alkatrész adatbázist mindhárom előző modul használja. Itt találhatók az alkatrészek szimbólumai, az alkatrészek tokozásai, működése és termikus tulajdonságai.



4. ábra: Alkatrész adatbázis (forrás Internet)

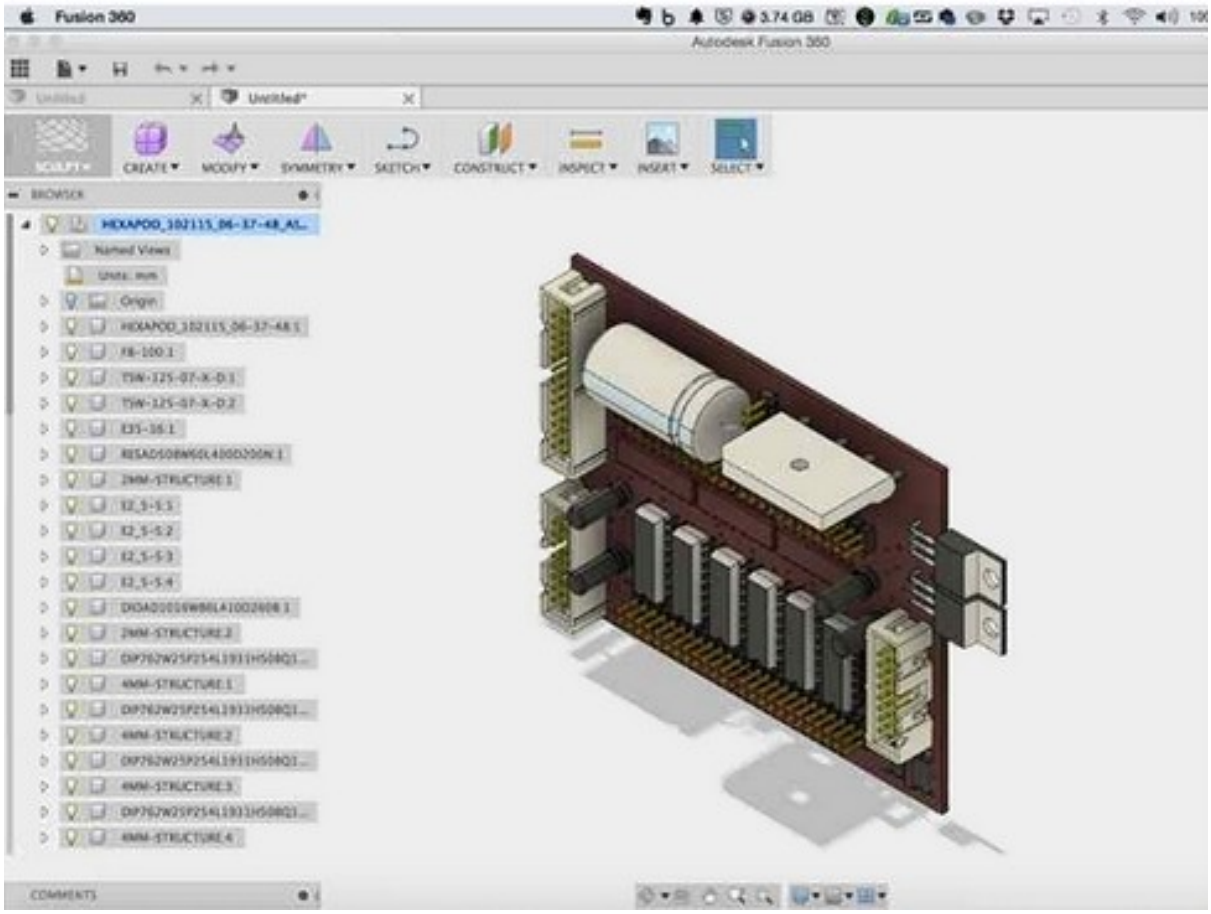
A villamos szimulációs modul a kötéslistából az adatbázis felhasználásával működési szimulációt hajt végre. Ez a modul is képes egy szöveges állományt készíteni, amely a későbbiekben feldolgozható.

A termikus szimulációs modul a kötéslistából az adatbázis felhasználásával az alkatrészek melegedését vizsgálja. Így ellenőrizhető az elkészült berendezés termikus viselkedése.

A CAM generátor a nyomtatott áramköri panel gyártó állományait állítja elő, úgymint a huzalozás, a forrasztó maszk, a feliratozás és a furatozási fájlok.

Néhány nagyobb teljesítményű villamos CAD lehetővé teszi a készülékdoboz megtervezését is.

Tipikus képviselői: Eagle, Altium Designer, stb..



5. ábra: Mechanikai rajz (forrás Internet)

2.5.1.2. Gépészeti CAD rendszerek

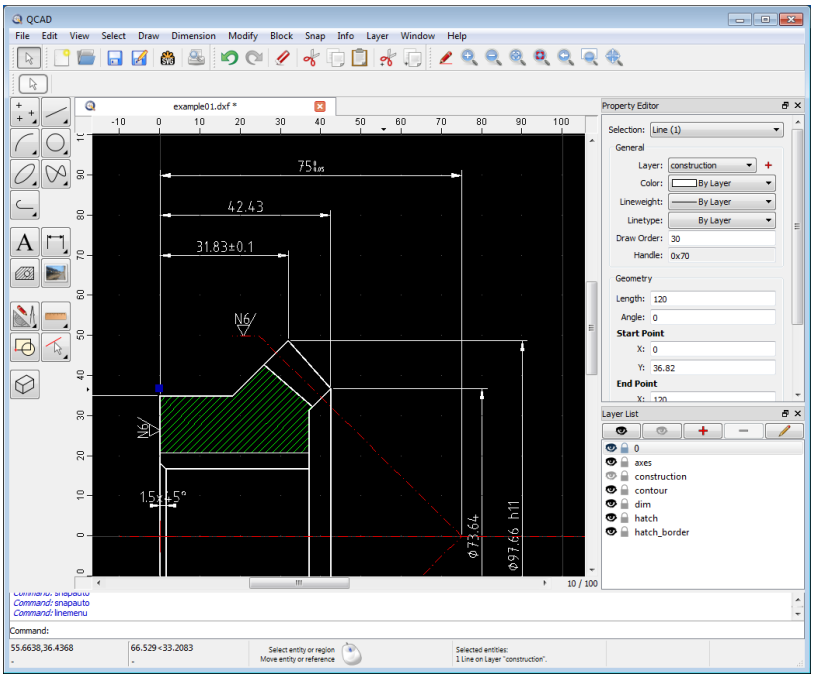
A gépészeti CAD szoftverek jóval szélesebb palettát biztosítanak a mérnököknek. Az egyszerűbb 2D rajzoló programoktól a 3D tervező szimulációs rendszerekig.

Az első kategória inkább a rajzolást segíti egy egyszerű alkatrész adatbázis támogatással. Ezek a szoftverek a hosszadalmas szerkesztést és a kihúzást szükségtelenné teszik és segítik a feladatra történő koncentrálást, viszont ritkán támogatnak szimulációs vizsgálatokat.

Tipikus képviselője: QCAD, LibreCAD.

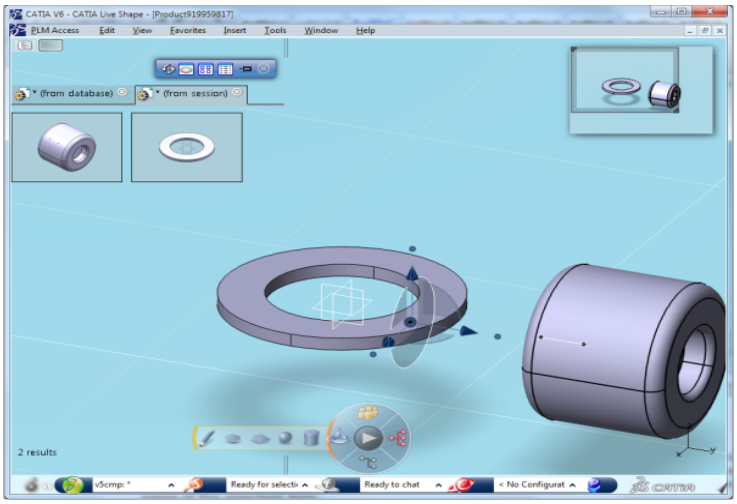
A második kategóriába tartozó rendszerek a rajzolás segítése mellett a készülő alkatrész, vagy termék térben történő megjelenítését támogatják. Legtöbbjük biztosítja az összeszereléssel kapcsolatos szimulációt és mozgásszimulációt. A legfejlettebb rendszerek szilárdságtani és

termikus szimulációt is biztosítanak.



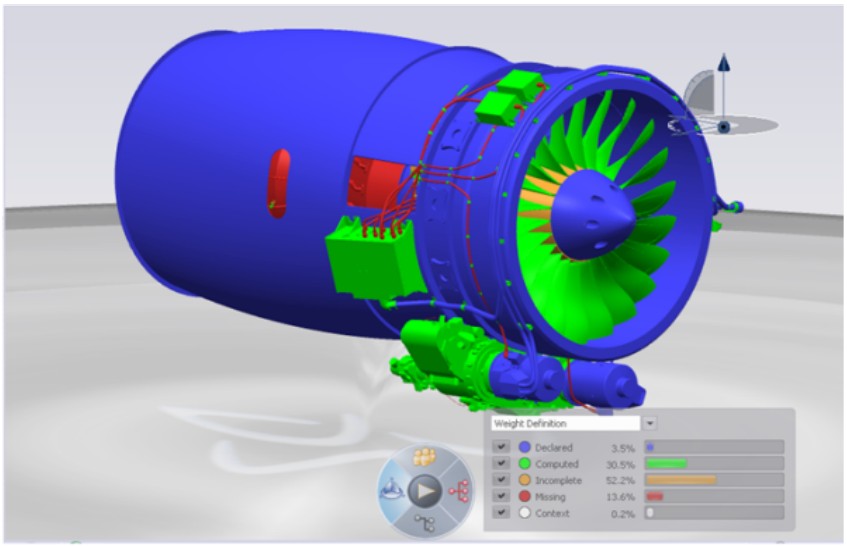
6. ábra: Tipikus 2D tervező rendszer képe (forrás Internet)

Ellentétben a 2D CAD rendszerekkel egyes termékek esetén még ergonómiai vizsgálat is végezhető. Nagy többségük speciális, vagy általános script nyelvekkel programozható.



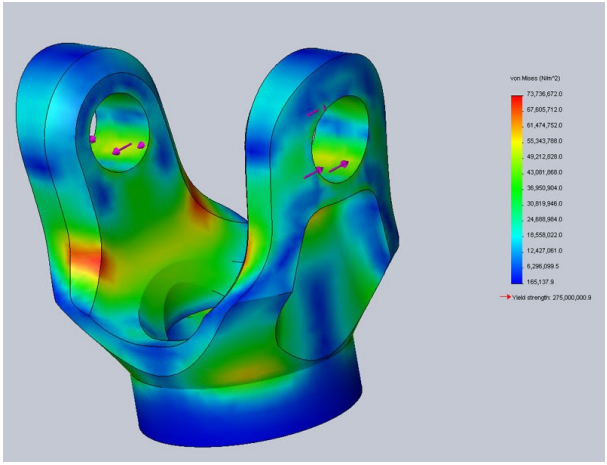
7. ábra: CATIA alkatrész rajz (forrás Internet)

A 3D CAD rendszerek tipikus képviselői: CATIA, AutoCAD, FreeCAD.



8. ábra: CATIA összeállítási rajz (forrás Internet)

Az építészeti CAD rendszerek sok mindenben hasonlítanak a gépészeti rendszerekhez. A 3D szerkesztés itt is elvárás, de plusz funkciókkal is rendelkeznek. Ilyen például a statikai

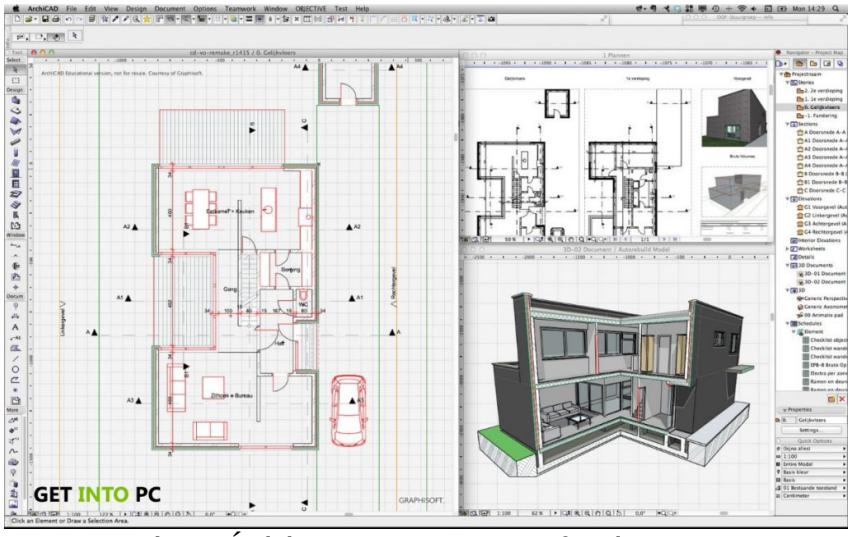


9. ábra: Q3 terhelési szimuláció (forrás Internet)

számítások elvégzése, illetőleg a kész épület belső „bejárása”.

Ezek a rendszerek szintén rendelkeznek igen széles adatbázis támogatással, mind az építő- és szerkezeti anyagminőség mind a beszerezhető elem készletet tekintve.

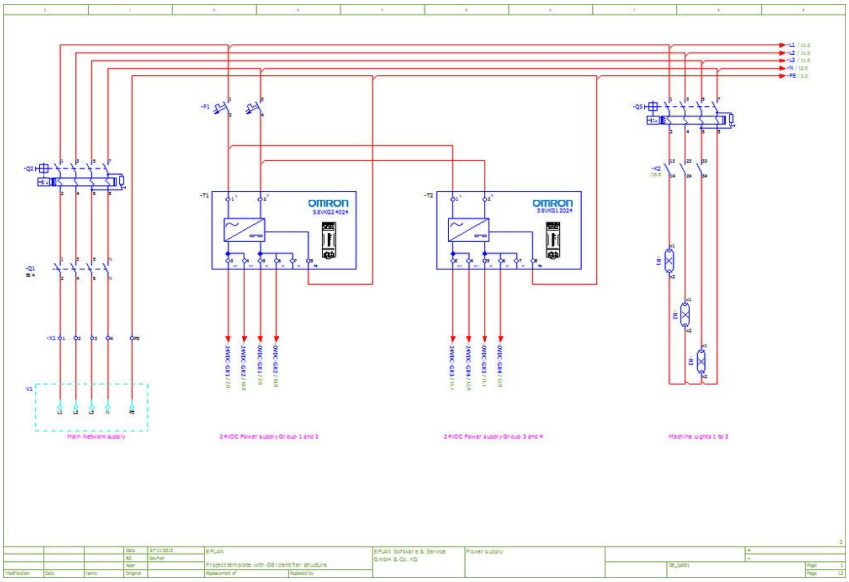
Az építészeti CAD rendszerek tipikus képviselője az Archicad.



10. ábra: Építészeti CAD rendszer (forrás Internet)

Az épületgépészeti CAD rendszerek épületek és területek teljes épületgépészeti tervezését teszik lehetővé minden tekintetben, például: vízellátás, villamos energia ellátás, szennyvíz ellátás, stb..

Tipikus képviselője az EPLAN.



11. ábra: EPLAN villamos rajz (forrás Internet)

2.5.2. CAM rendszerek

A CAD rendszerek a termékkel foglalkoznak, a CAM rendszerek - nagyon leegyszerűsítve - azzal a folyamattal, ahogyan ezt a terméket előállítják.

Legegyszerűbben a villamos CAM rendszerek működnek. A CAM rendszer számára a nyomtatási rajz a legfontosabb, mert ez tartalmazza a rétegek huzalozási mintáit, a furatozási értékeket és az alkatrészek beültetési helyét és orientációját. A gyakorlatban ezeket az adatokat konvertálja még a CAD rendszer szabványos állományokká, amelyeket a megmunkáló gépek közvetlenül fel tudnak használni.

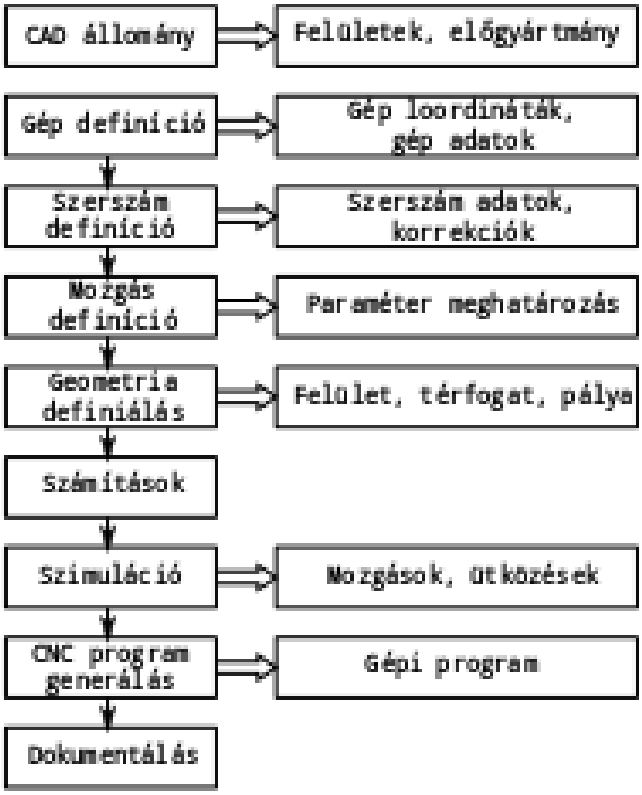
Ilyen állomány a fotoplotter által használt fájl, az esetek túlnyomó többségében egy Gerber állomány, a furatozási fájl, amely a furatok helyét és átmérőjét adja meg a CNC fúrógép számára, a körülmarás állománya, amely az elkészült panel kontúrmarás és belső marásainak CNC állománya, illetve a beültető gép számára a beültetési alkatrész és orientációs lista. A fenti állományok szabványos kimenetek.

A gépészeti CAM rendszerek jóval összetettebb rendszerek. A kérdéses technológia határoz meg számos betartandó paramétert.

Vegyünk példának egy marási feladatot, amely forgácsolási technológia, a minimálisan figyelembe veendő paraméterek és feladatok a következők:

- szerszámpálya tervezés,
    - szerszámpálya számítás figyelembe véve az anyagi jellemzőket,
    - szerszámkorrekciók számítása, figyelembe véve a szerszám geometriát és a korlátokat,
    - szerszámpálya optimalizálás,
- CNC program generálás,
    - a gép szabadságfokainak száma,
    - a gép utasításkészlet ismerete és használata,
    - megmunkálás optimalizálás.

Ez így leírva elég egyszerűnek tűnik, azonban a valóságban ez igen bonyolult lehet. Ez folyamábrán a következő:



12. ábra: CAM feldolgozás folyamata

Az építészeti CAD rendszerek nagyon ritkán, vagy egyáltalán nem rendelkeznek CAM interfésszel.

2.6. Adatbázis rendszerek

Definíció: az informatikában adatnak nevezzük azokat a kódokkal leírható "dolgokat" amelyek rögzíthetők, feldolgozhatók, kereshetők és megjeleníthetők. Az adat egy objektum, vagy fogalom egy meghatározott tulajdonságának "értéke".

Definíció: az adatbázis azonos jellegű, strukturált adatok összessége, amelyet egy tárolására, lekérdezésére és szerkesztésére alkalmas szoftvereszköz az adatbázis kezelő kezel.

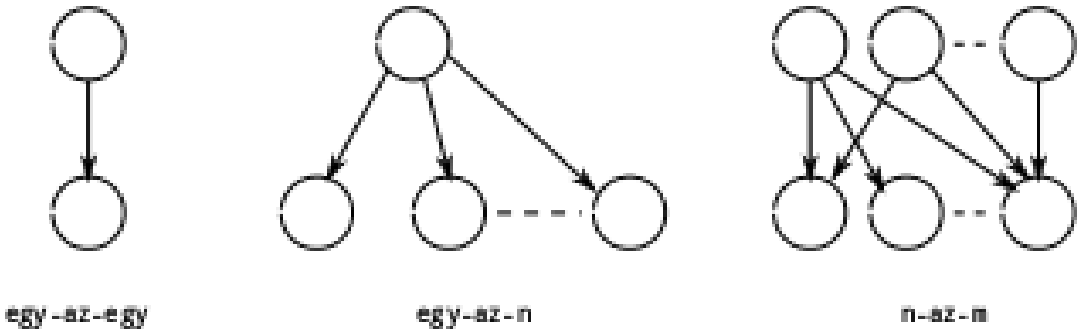
Az adatbázisok célja adatok megbízható, tartós[10] tárolása, és a lehetőségekhez mérten gyors

10 Perzisztens.

visszakereshetőségének biztosítása.

Definíció: az adatbázis-kezelő egy olyan szoftvereszköz, amely az adatbázisban tárolt adatok kezelését, karbantartását teszi lehetővé.

Az adatelemek egymáshoz többféleképpen kapcsolódhatnak, ez lehet egy-az-egy kapcsolat, egy-az-n kapcsolat, illetve n-az-m kapcsolat.



13. ábra: Kapcsolatok adatelemek között

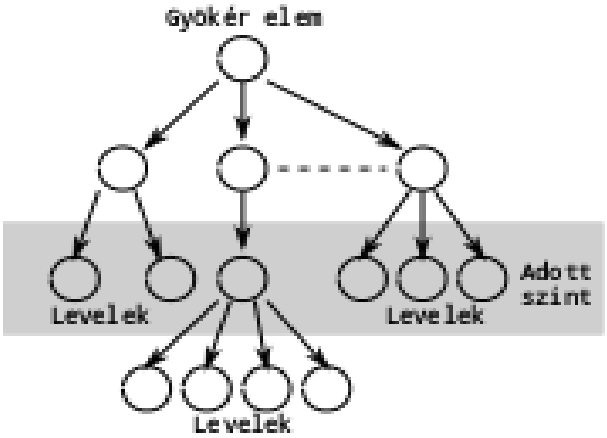
Az adatbázisok felépítése többféle lehet, ezek:

Hierarchikus adatmodell: gráfelméleti szempontból ezek az adatbázisok egy irányított fa struktúrát valósítanak meg. Ez azt jelenti hogy egy adott elemtől kezdődően lehet az adatbázisban keresni. Ilyen jellegű adatbázisok a felügyeleti rendszerekben és a CAD rendszerekben kerültek[11] felhasználásra.

Ennek a modellnek az az előnye, hogy csak a fa bejárással érhetők el az adatok. Ez igazából a hátránya is, azonos szinten lévő adatelemek között nagyon nehéz az „átjárás”.

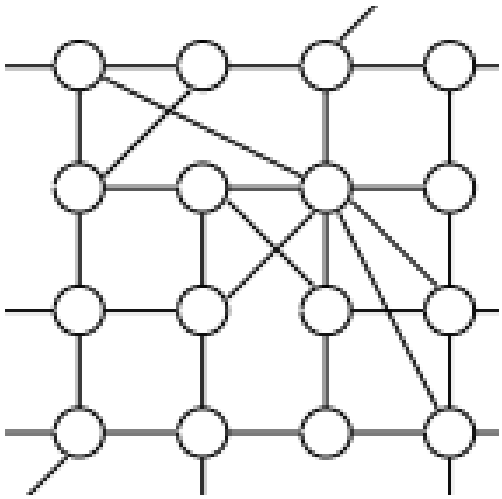
Az adatelemek között egy-az-n kapcsolat van.

11 Egy pár sorral lejjebb találhatók a relációs adatbázisok, ezekkel szinte minden modell felépíthető, ez az oka a múlt időnek.



14. ábra: Hierarchikus modell

Hálós adatmodell: ebben az adatmodellben az adatelemek között n-az-m jellegű kapcsolat van. Ez az adatmodell nagyon bonyolult, ezért nem elterjedt. Használata gyors szakértői rendszerekben lehetséges. Az adatelemek között n-az-m kapcsolat építhető fel.

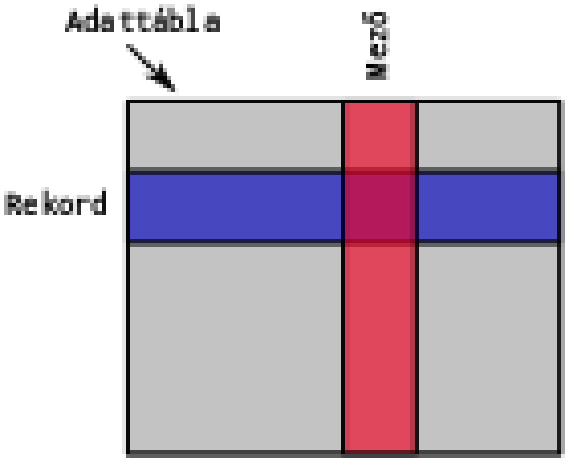


15. ábra: Hálós adatmodell

Objektum orientált modell: az objektum orientált programfejlesztési (OOP) módszertan eredményeit használja fel. A kérdéses adatelem hordozza magával azokat az eljárásokat, amelyek a feldolgozásához szükségesek. Ezek az adatbázisok és kezelő szoftvereik rendelkezhetnek az OOP nyelvek tulajdonságaival, úgymint: egységbezárás, öröklődés, többalakúság és plusz tulajdonságként az objektum azonossággal.

Relációs modell: a legelterjedtebb adatmodell. Alapja az adattábla, amelynek sorai az úgynevezett rekordok és oszlopai a mezők.

A rekordok egy „egyént” (entitást) írnak le a mezők ezeknek a tulajdonágait írják le.



16. ábra: Adattábla, rekord, mező

Az adattáblák adott „tulajdonságok” alapján összefűzhetők. Így igen komplex és hatékony adatbázisok építhetők fel. Tulajdonképpen az előzőleg ismertetett modellek az objektum orientált modell néhány tulajdonságának kivételével felépíthetők.

A relációs adatbázis modell kezelésre manapság a legelterjedtebb nyelv az SQL, amely egyfajta pszeudo szabványként jelenik meg. Ez azt jeleneti, hogy bizonyos SQL megvalósítások kismértékben különböznek egymástól, de lényegében működésük azonos.

Az SQL nyelv egy parancsnyelv, amely az adott felületről kézzel kezelhető, de ha automatizálni akarjuk valamilyen hordozó nyelvre van szükségünk. Ez a nyelv tetszőleges lehet, de a nagyobb környezetek biztosítanak közvetlen interfészeket.

Az adatmodelltől függetlenül minden adatbázis kezelő rendszer három résznyelvből tevődik össze[12], ezek:

DDL Data Definition Language adat definíciós nyelv, ez teszi lehetővé az adatok, adat

12 DATA BASE TASK GROUP javasolta ezt a felosztást.

táblák, adatbázisok és kapcsolatok definiálását.

DML Data Manipulation Language adat manipulációs nyelv, az adatok, adattáblák, adatbázisok és kapcsolatok feltöltését, változtatását és törlését teszi lehetővé.

DQL Data Query Language adat lekérdező nyelv, amely adott kritériumoknak megfelelően adat lekérdezést tesz lehetővé.

Az adatbázisok megvalósításukat tekintve lehetnek monolitikus adatbázisok és osztott adatbázisok.

A monolitikus adatbázisok logikai és fizikai felépítésüket tekintve egyetlen egységként jelennek meg. Ez azt jelenti, hogy az adatállományok egy egységként kerülnek kezelésre.

Ezeknek az adatbázisoknak az az előnyük, hogy egyszerűbbek, könnyebb az adatmodell tervezése és kevesebb hardver erőforrást igényelnek.

Hátrányuk az, hogy az adatbázis elérése nagy számú és távolsági kérés esetén lelassul, a kommunikációs igény megnövekszik és az adatbázis sérülése esetlegesen komoly adatvesztést okoz.

Az osztott adatbázisok esetén az adatbázis nem egyetlen helyen van felépítve, hanem az igényeknek megfelelően szét van osztva. Az adatbázis időben is osztott. Az előtérben lévő (front-end) adatkezelő elemek az adott helyhez tartozó adatokat tartalmazzák. Ezek az adatok az adott helyről egyetlen üzenetként kerülnek a háttér adatkezelőre (back-end).

Előnye, hogy a kommunikációs idő és költség lecsökken. Az adatok a front-end egységeken egy adott ideig tárolódnak redundánsan. Egyetlen front-end kiesése nem okozza a teljes rendszer leállását, könnyebb a tervezés. Nagyobb rendszereknél gyakorlatilag az egyetlen járható út, ha megfelelő teljesítményt várunk el.

Hátránya a jóval bonyolultabb felépítés és az adatok konzisztens[13] kezelésének megoldása.

13 A konzisztencia egy adatbázisban több adat közötti viszonyra utal, ahol egy adatelem előfordulásának tartalma megegyezik a másik azonos előfordulás tartalmával (Wikipédia).

3. Rendszerprogramok

3.1. Operációs rendszerek

A rendszerprogramok tipikus képviselői az operációs rendszerek. Ezek a szoftvercsomagok teszik lehetővé, hogy az összetettebb számítástechnikai eszközök felhasználhatók legyenek a kiválasztott célra.

Az operációs rendszerek több szempont szerint osztályozhatók, ezek:

- a felhasználás területe szerint,
- a futtatott feladatok és a felhasználók száma szerint,
- a kezelés módja szerint.

3.1.1. Operációs rendszerek osztályozása a felhasználás területe szerint

Az operációs rendszerek első körben két alapvető csoportba oszthatók: általános célú operációs rendszerek és speciális operációs rendszerek.

Általános célú operációs rendszerek azok a rendszerek, amelyekkel a felhasználó a mindennapokban találkozik. Tipikus képviselőik a Windows, Linux, iOs és BSD UNIX rendszerek. Biztosítják a felhasználói programok futtatását és a rendszer kezelését.

Kezelésük lehet grafikus, vagy karakteres, illetve mindkettő[14].

A speciális célú operációs rendszerek, mint a nevük is mutatja valamilyen speciális célra lettek kifejlesztve. Ez a csoport tovább osztható:

- hálózati operációs rendszerek. Ezek kimondottan számítógép hálózat kezelésére lettek kifejlesztve.

14 A kezdetleges általános célú operációs rendszerek karakteres felületűek voltak. Macintosh és Windows rendszerek terjedésével úgy tűnt, hogy örökre eltűnnek, de az utóbbi időkben a karakteres felületek használati lehetősége újra előtérbe került, lásd Windows 7. A Linux - UNIX rendszerek mindkét lehetőséget a kezdetektől fogva biztosították.

Érdekes tény az, hogy a manapság ez a kategória gyakorlatilag megszűnt és szerepét átvette azoknak a rendszereknek a csoportja, amelyek valamilyen általános rendszerből lettek kifejlesztve, tulajdonképpen erre a célra konfigurálták ezeket.

Tipikus képviselői a Novell szerverek 4.10 verzióig.

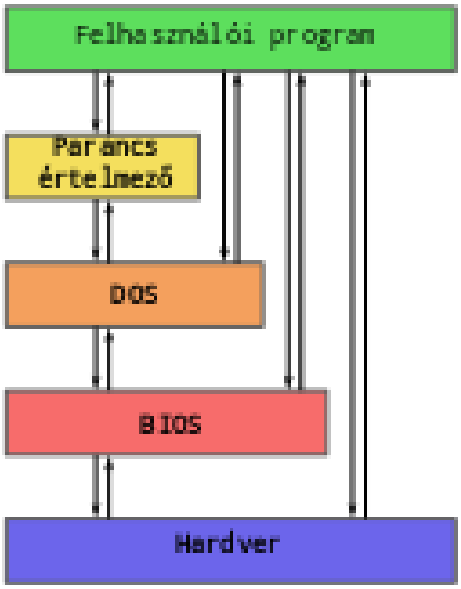
- Real-Time operációs rendszerek. Ezek a rendszerek fizikai eszközök és rendszerek kezelésére, irányítására fejlesztették ki.

  Ezeket a rendszereket találhatjuk az autóban a motor vezérlő elektronikában, az ABS-be, vezetési asszisztensben, a mosógépben, a klíma szabályozóban a vagyonvédelmi rendszerekben.

  Átlagos felhasználó mindennap találkozik ezekkel, de általában nem ismeri fel.

3.1.2. Operációs rendszerek felosztása a felhasználók száma és a futtatott feladatok szerint

SUST (Single User Single Task) egyfelhasználós – egytaszkos operációs rendszerek. Az adott számítógépen egyetlen felhasználó egyetlen feladaton képes dolgozni. Ennek tipikus képviselői a DOS különböző változatai (MS DOS, FreeDOS, CP/M stb.).



17. ábra: A DOS tipikus felépítése

A SUST operációs rendszerek a PC első húsz évében nagyon jelentős szerepet töltöttek be. Ennek oka kettős, egyrészt a számítógépek alapfunkcióit az elvárásoknak megfelelően kiszolgálták, másrészt a SUST jellegből adódóan a hardver elérése viszonylag egyszerűen történt. Ezért 2008-ban is lehetett olyan ipari PC-vel találkozni, amely DOS-t futtatott.

A DOS eltűnésének oka az ISA busz eltűnése és a PCI busz bevezetése, az USB csatlakozás és a számítógép hálózatok elterjedése volt. Ezeket már ezzel a technológiával nem lehetett hatékonyan kezelni.

A DOS felépítése a 17. ábrán látható. Részei:

- felhasználói program, nem része az operációs rendszernek,
- parancs értelmező, amely a felhasználó felé biztosít kezelési interfészt, a felhasználói program is használhatja,
- DOS az operációs rendszer magas szintű része, amely nem tart közvetlen kapcsolatot a hardverrel,
- BIOS az operációs rendszer alacsony szintű része, amely a DOS felől érkező hívásokat a hardver kezelési primitívekre bontja le,

- a hardver, amelyen a rendszer fut.

A DOS nagy előnye, hogy a felhasználói program szintjéről is a hardver közvetlenül elérhető. Így viszonylag egyszerűen megvalósítható olyan alkalmazás, amely esetén szükséges járulékos hardver alkalmazása.

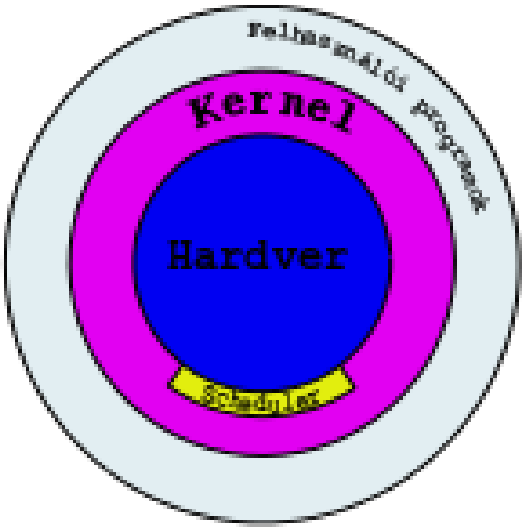
SUMT (Single User Single Task) egyfelhasználós – egytaszkos operációs rendszerek. Ezek a rendszerek átmeneti jellegűek voltak az SUST és MUMT[15] rendszerek között. Nem támogatták, vagy csak nagyon korlátozottan több felhasználó egyidejű tevékenységét, de azt lehetővé tették, hogy egy felhasználó több feladatot futtasson.

Tipikus képviselői. Korai OS2, Windows 3.1, Windows 95, 98, stb.

MUMT (Multi User Multi Task) többfelhasználós – többtaszkos rendszerek. Ebben a kategóriában egyidejűleg[16] több felhasználó is dolgozhat több feladaton. Ez azonban az előzőeknél jóval bonyolultabb rendszert feltételez, amely jóval komolyabb számítási teljesítményt is igényel.

A rendszer felépítését a 18. ábra mutatja.

15 Következő kategória.
16 Vagy látszólag egyidejűleg.



18. ábra: Többfelhasználós és többtaszkos rendszer

A rendszer részei:

- a felhasználói program, nem az operációs rendszer része,
- a kernel az operációs rendszer magja,
- scheduler, az ütemező, amely a rendszeren futó folyamatok ütemezését végzi,
- a hardver, amelyen rendszer fut.

Az ábrán látható, hogy a felhasználói szintet a kernel teljesen eltakarja a hardvertől. Ez azért szükséges, mert a folyamatok gyakorlatilag bármikor elindulhatnak, így bizonyos erőforrásokat esetleg több folyamat kezel, ami hibához vezetne.

Az ütemező alapvetően meghatározza a rendszer időbeli működését. Figyeli a folyamatok állapotát, prioritását és az ütemezési stratégiának megfelelően a folyamatokat leálltja, illetőleg elindítja.

Az ütemező két alapvető elven működhet, ezek:

1. megszakításos – pre-emptive – ütemezés. Ekkor minden folyamat azonos időszeletet kap, amely alatt futhat. Ha ez az időszelet lejár, az ütemező

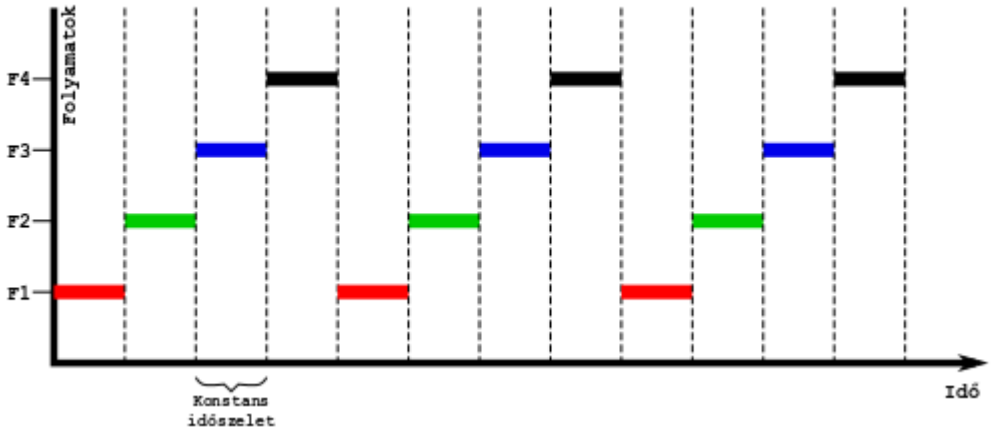
elveszi a hardvert a kérdéses folyamattól és egy másikat indít el.

2. kooperatív ütemezés. Ebben az esetben az ütemező csak elindítja a folyamatot, de a folyamat az, aki lemond a hardverről és hívja az ütemezőt. Ettől kezdve már az ütemező dönt arról, hogy melyik folyamatot indítja el.

Mindkét működési elvnek megvan az előnye és a hátránya.

- megszakításos ütemezés esetén, ha a folyamat valamilyen okból „lefagy”. Az ütemező az időszelet lejártakor a lefagyott folyamatot is megszakítja, így a teljes rendszer nem áll le. Viszont mivel a folyamat bármikor megszakítható nagyon sok adatot kell elmenteni azért, hogy a kérdéses folyamat zavar nélkül újra indulhasson.

  A 19. ábra vázlatosan bemutatja a megszakításos ütemezést.



19. ábra: Megszakításos (pre-emptive) ütemezés

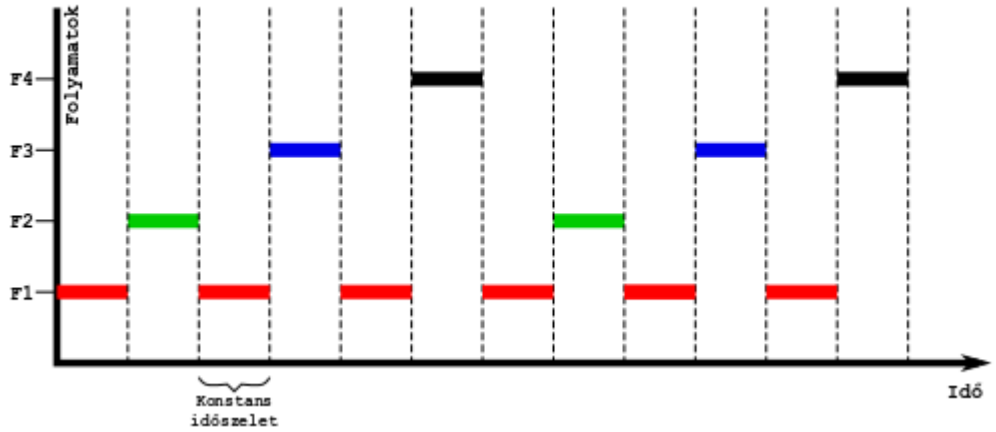
- Kooperatív ütemezés esetén a folyamat saját maga menti el a jellemzőit[17], de miután a folyamat hívja az ütemezőt egy lefagyás a rendszer leállásához vezethet.

17 Egy kicsit bonyolultabb, de nem nagyon. Csak ez egyenlőre kicsit korai lenne.

20. ábra: Kooperatív ütemezés

A 20. ábra a kooperatív ütemezés vázlatát mutatja.A prioritás kezelése szintén az ütemező feladata. Az ütemező azt a folyamatot fogja elindítani, amely az ütemezési stratégiának megfelel.

A 21. ábra megszakításos ütemezés esetén mutat be egy prioritásos ütemezést. Az F1 folyamat prioritása magasabb, mint a z F2, F3 és F4 folyamatok prioritása. Ezért az ütemező az F1 folyamatot gyakrabban indítja, mint a többit.



21. ábra: Prioritás kezelési példa megszakításos ütemezés esetén

4. Fejlesztő rendszerek

A fejlesztő rendszerek olyan szoftvercsomagok, amelyeket szoftverfejlesztésre használnak. Ezek a rendszerek bonyolultságukat tekintve igen változatosak a legegyszerűbb programkészlettől a nagy bonyolultságú programgenerálásra is alkalmas programrendszerekig terjednek.

A fejlesztő rendszerek kategóriái a következők:

- egyszerű fejlesztői szoftvercsomag (tool chain). Ezek a csomagok az adott feladathoz vannak hangolva, amit nagyon sokszor a programozó tesz meg. Egyszerűek, gyakran ingyenesek és tapasztalt szakember kezében rendkívül hatékonyak. Viszont kevés támogatást nyújtanak a programozónak, ami abban az esetben hátrány, ha a programozó nem rendelkezik kellő gyakorlattal.

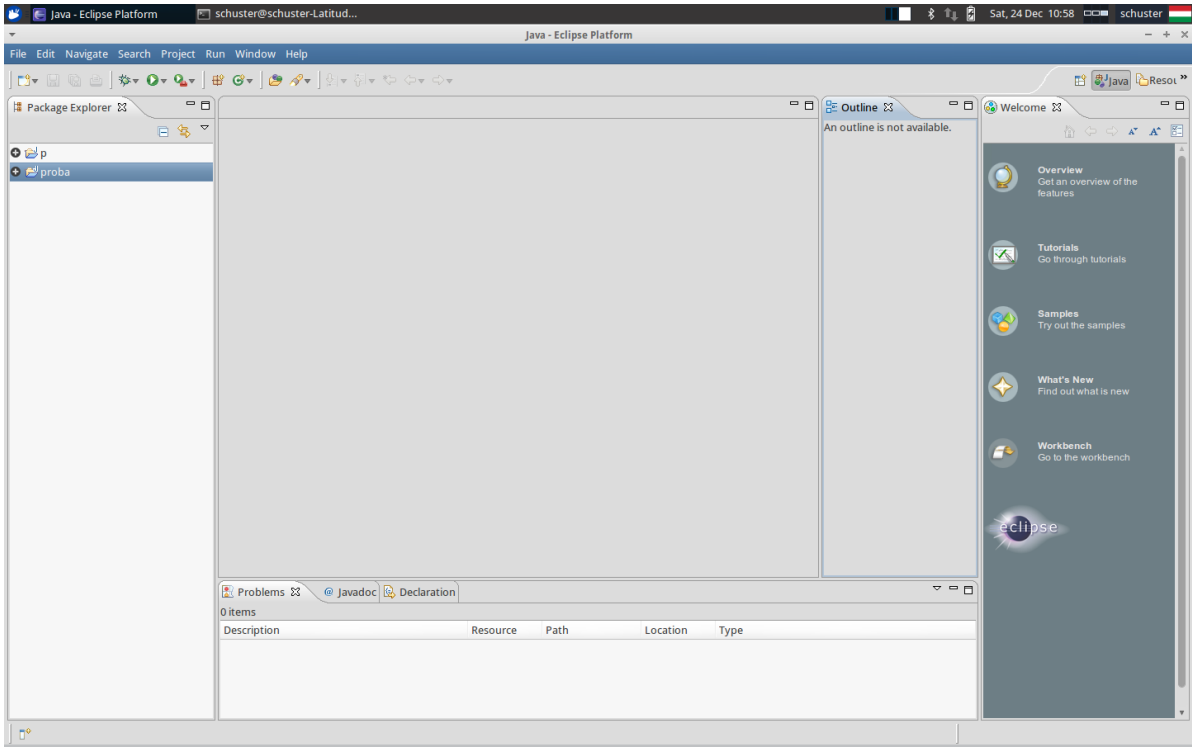
  Ilyen tool chain az gcc-avr programcsomag, amely tartalmazza a következőeszközöket:

  - gcc-avr C fordító és linker,
  - gdb-avr debugger,
  - avr-libc alapfunkciók könyvtára,
  - avarice programozó,
  - avra assembler az adott mikrokontroller családra.

- integrált fejlesztői környezetek. Az előző kategóriának a kiterjesztése, amely mögött ott vannak ugyanazok a szoftver eszközök, mint az előző kategóriában, de ezt egy – remélhetőleg jól összeállított felület fog össze. Ennek a rendszernek az a nagy előnye, hogy egységben kezeli a teljes fejlesztést, on-line segítséget képes nyújtani és számos esetben segíti a program forrásának létrehozását is. Hátránya az, hogy némileg lassabb lehet, mint az előző rendszer.

  Ilyen integrált környezet például az Eclipse. Az Eclipse a valóságban tényleg

egy héj, amit a kérdéses feladatra lehet hangolni. Nagy előnye, hogy platform független, tehát Linux, Windows és MAC rendszerekben is használható.



22. ábra: Eclipse fejlesztői felület

- nagy bonyolultságú CASE eszközök. Ezek a rendszerek egyel magasabb szintre emelik a fejlesztési folyamatot. Ez nagy vonalakban azt jelenti, hogy képesek szoftvereket előállítani már szabványos leíró állományokból. Ezáltal a szoftver fejlesztés időtartama jóval rövidebb lesz és várhatóan a szoftver megbízhatósága is javulhat.

  Ilyen eszköz például az Autosar fejlesztői eszközök[18].

A következő lépés a fejlesztői rendszereket alkotó programok ismertetése:

- szerkesztő program. Szerepe a forrásállományok és az egyéb szoftver elemek szerkesztése. Ezek az szerkesztő programok a korszerű rendszerek esetén legalább két szerkesztési funkciót ismernek. Az első funkció a forrás szöveg (programlista) szerkesztése a másik a beállító és vezérlő állományok szerkesztése.

18 Sajnos az Autosar eszközökről nincs képernyő képünk.

A forrás szöveg egy közvetlenül olvasható állomány, speciális vezérlő elemek csak az adott programozási nyelvre jellemzők.

A beállító állományok nem egyszerű szöveg állományok, általában valamilyen metanyelven készülnek, mint például xml, vagy Jason.

- Fordító program. Szerepe a forrásállomány teljes, vagy részleges fordítása. Amennyiben a fordítás részleges, akkor további feldolgozásra lesz szükség.
- Linker. Szerepe az előbb említett részlegesen fordított állomány, más állományok és előre elkészített - úgynevezett könyvtári állományok szerkesztése futtatható állománnyá.
- Debugger (teszter) program. Ez a program elősegíti a programok dinamikus tesztelését.
- Kód auditor. Szerepe a forráskód elemzése azért, hogy az előre beállított konvenciókat[19] a kód írója betartja-e, vagy sem.
- Kód lefedettség ellenőrző. Elemzi, hogy a kód a teljes program és adatterületet milyen mértékben használja az elkészült kód.

5. Szoftverek osztályozása megbízhatósági szempontból

Definíció: a szoftver a specifikációban megadott funkciókat milyen valószínűséggel teljesíti az előre meghatározott körülmények között.

A fenti definíciót figyelembe véve öt kategóriát határoznak meg a következő alszempontokat vizsgálva: emberi élet, környezeti hatások, pénzügyi hatások, erkölcsi következmény.

5. veszélybiztos rendszerek. Működési kiesés nem lehetséges, mert:

- emberi életben tömeges kár eshet, több haláleset és tömeges sérülésveszély fordulhat

19 Mint például a MISRA. A MISRA egy korlátozás a C és a C++ nyelvekre, hogy a félreértéseket és programhibákat megelőzzük.

elő,

- környezeti katasztrófa következhet be,
- pénzügyi katasztrófa következhet be,
- az erkölcsi kár hatalmas.

Példa az ilyen rendszerekre a repülésirányító rendszerek, vasúti biztosító berendezések, impulzus reaktorok irányító berendezései.

4. működésbiztos rendszerek. A rendszereknek igen magas megbízhatósággal kell rendelkeznie, mert hibás működés esetén:

- emberi életben kár eshet és sérülésveszély valószínű,
- környezeti katasztrófa lehetséges,
- nagymértékű pénzügyi kár lehetséges,
- az erkölcsi kár hatalmas.

Példa az ilyen rendszerekre a jármű fedélzeti rendszerek, az erőművi rendszerek beleértve a nyomott vizes atomerőművek irányítási rendszereit.

3. nagy megbízhatóságú rendszerek. Működésképtelenség esetén

- emberi életben eshet kár,
- környezeti katasztrófa nem valószínű, de környezet szennyezés lehetséges,
- pénzügyi hatás jelentős lehet,
- az erkölcsi kár jelentős.

Példa az ilyen rendszerekre a nagy országos adatbázisok és nyilvántartó rendszerek és banki rendszerek.

2. általános megbízhatóságú rendszerek,

- emberi életben nem eshet kár,

- környezeti hatás nem valószínű,
- pénzügyi hatás lehetséges,
- az erkölcsi kár jelentős lehet.

Példa az ilyen rendszerekre az irodai rendszerek, CAD rendszerek, átlagos ipari rendszerek.

1. lényegtelen megbízhatóságú rendszerek,

- emberi életben nem eshet kár,
- környezeti hatás nincs,
- pénzügyi hatás nincs,
- erkölcsi kár lehetséges.

Példa az ilyen rendszerekre a játék programok, otthoni rendszerek.

6. Életciklus és életciklus modellek

Idézet: „a szoftver mérnöki tevékenység eredménye, és mint ilyen tervezést igényel”. Ezt az idézetet 1991-ben egy hallgató írta le szakdolgozatában.

A szoftver fejlesztésnek jól meghatározható lépései vannak. A szakirodalom több lépést különböztet meg. A legelterjedtebb a következő lépéseket adja meg:

1. Követelmények meghatározása. Ez az a specifikáció, ami alapján a szoftver elkészül.
2. Tervezés. A szoftvert a fenti idézetnek megfelelően tervezni kell.
3. Megvalósítás. A szoftvert a megfelelő módszertan szerint, a rendelkezésre álló eszközökkel létre kell hozni.
4. Validáció. Ez a lépés a szoftver tesztelését és minősítését jelenti.
5. Karbantartás. A szoftver a leszállítás után még tartalmazhat hibákat, ezeket javítani

szükséges.

Sajnos ez a felsorolás nem elegendően részletes arra, hogy a folyamatot jól leírhassuk. Ezért a továbbiakban a Bundeswehr által 1992-ben definiált lépéseket használjuk.

1. Kockázat elemzés, specifikáció és szerződéskötés. Ebben a lépésben el kell dönteni, hogy az adott fejlesztő szervezet képes-e a feladatot elvégezni, illetőleg érdemes-e az adott feladatot elvállalni. Amennyiben az előző feltételek alapján a feladat elvégezhető a pontos specifikációt össze kell állítani.

   Sajnos ez sokszor nem egyszerű feladat. Ennek egyszerű oka az, hogy a megrendelő sok esetben nem szakember és csak halvány elképzelései vannak a feladat pontos megvalósításáról. Ekkor a fejlesztők feladata, hogy a specifikációt előállítsa. Azonban nagyon fontos az, hogy az elkészült specifikációt pontról – pontra ismertesse és jóváhagyassa a megrendelővel.

   Utolsó lépésként meg kell kötni a szerződést. Nagyon fontos, hogy a következő mondat szerepeljen a végleges szerződésben:

   „A szerződés tárgya a hivatkozott specifikációban leírt szoftver elkészítése. Minden eltérés a hivatkozott specifikációtól az új szerződés tárgyát képezi.”

2. Logikai tervezés. Ebben a lépésben a szoftver strukturális felépítését határozzák meg. Itt még nincsenek algoritmusok és adatszerkezetek, de a főbb modulokat meghatározzák és ezáltal a szoftver szétoszthtató a fejlesztői csoportok között.

3. Fizikai tervezés. Itt történik a konkrét, részletekbe menő tervezés. Ebben a lépésben tervezik a megfelelő adatszerkezeteket, tervezik az algoritmusokat és a konkrét működést.

4. Megvalósítás[20]. A konkrét kód előállítása ekkor történik a megfelelő eszközök és módszertan felhasználásával. A módszertanokkal később foglalkozunk részletesen.

5. Tesztelés. A tesztelés során a szoftverben lévő hibákat keresik és javítják. A szoftver teszteléssel szintén később foglalkozunk.

20 Nagyon sokan azt hiszik, hogy ez a lépés a szoftverfejlesztés. Azonban ez a szakasz csak 20-40%-a teljes életciklusnak.

6. Átadás. A szoftver elkészült, a megrendelőnek át kell adni. Ez sem a legegyszerűbb folyamat, mert a megrendelő számára lépésről lépésre a specifikációnak megfelelően bizonyítani kell a szoftver működését és a specifikáció megfelelőségét.
7. Követés és karbantartás. A szoftverben lehetnek hibák, amelyek a használat során derülnek ki. Ezeket a hibákat a szoftver készítője javítja. Erre a készítő törvényileg meghatározott ideig garanciálisan kötelezett. Előfordulhat az is, hogy a fejlesztő talál hibát az átadás után. Ekkor a megrendelőt értesíti és javítja hibát.

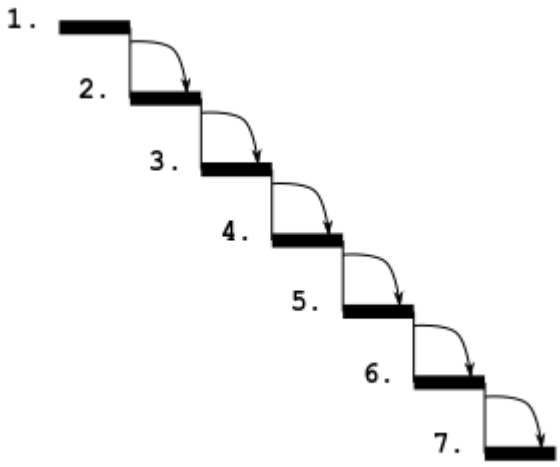
A fenti lépéssort felhasználva definiálhatók az életciklus modellek.

Vízesés modell. Egy teljesen szekvenciális modell, a lépések szigorúan egymás után követik egymást. Nem megengedett a visszalépés.

A modell előnye, hogy nagyon jól nyomon követhető a projekt haladása és becsülhető a projekt befejezése.

Hátránya az, hogy a tervezési hibákat a projekt és a szoftver viselni fogja. Ezek a hibák általában nem katasztrofális hibák, inkább a kezelhetőség és áttekinthetőség szempontjából kellemetlenek.

A vízesés modell általában egyszerű jól nyomon követhető szoftverek esetén használható, mint például egy PLC program esetén.



23. ábra: Vízesés modell

Az inkrementális fejlesztés során módunk van arra, hogy a rendszer problematikus részeit

fejlesszék ki először. Ha ez sikerrel járt, akkor ehhez hozzáfejleszthetik az újabb és újabb részeket.

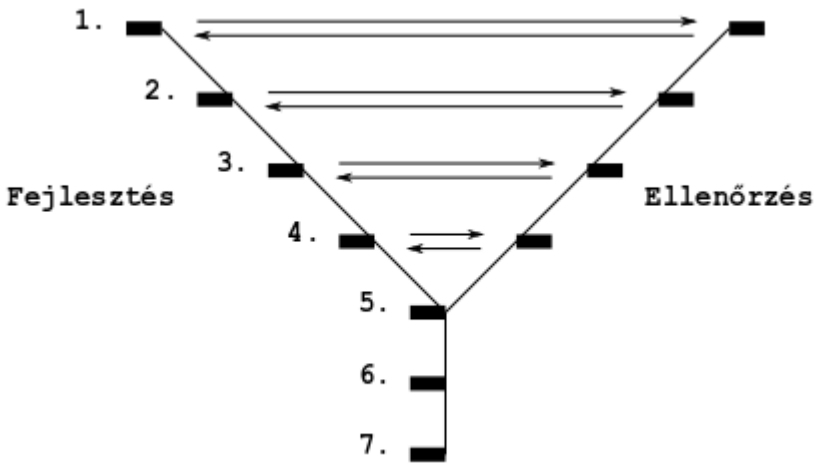
Sok esetben az is előfordul, hogy a megrendelő nem képes teljes specifikációt adni, de a fejlesztő már látja, hogy biztosan mely elemekre lesz szüksége, és ezeket fejleszti. Így lehetővé válik a specifikáció rögzítése után a gyorsabb kiszállítás.

Az inkrementális fejlesztési folyamatban ellentétben a vízesésmodellel a rendszer nem egyenletesen fejlődik. Bizonyos részek korábban kerülnek fejlesztésre. A két modell természetesen kombinálható.

V-modell. Hasonlóan szekvenciális modell, de a kritikus részeken a fejlesztői folyamat mellett egy ellenőrző folyamat is működik. A fejlesztők által előállított dokumentumokat a az ellenőrök folyamatosan vizsgálják és hiba, vagy eltérés esetén beavatkoznak. Ezáltal korai szakaszban sokkal nagyobb eséllyel felderíthetők a problémák.

Ez a modell szintén jól követhető és jobb a „kihozatala”, mint a vízesés modellé. Viszont láthatóan erőforrás igényesebb.

Felhasználása tömeges előállítású biztonság kritikus szoftverek fejlesztése során szokásos, mint például gépkocsik fedélzeti szoftverei.



24. ábra: V-modell

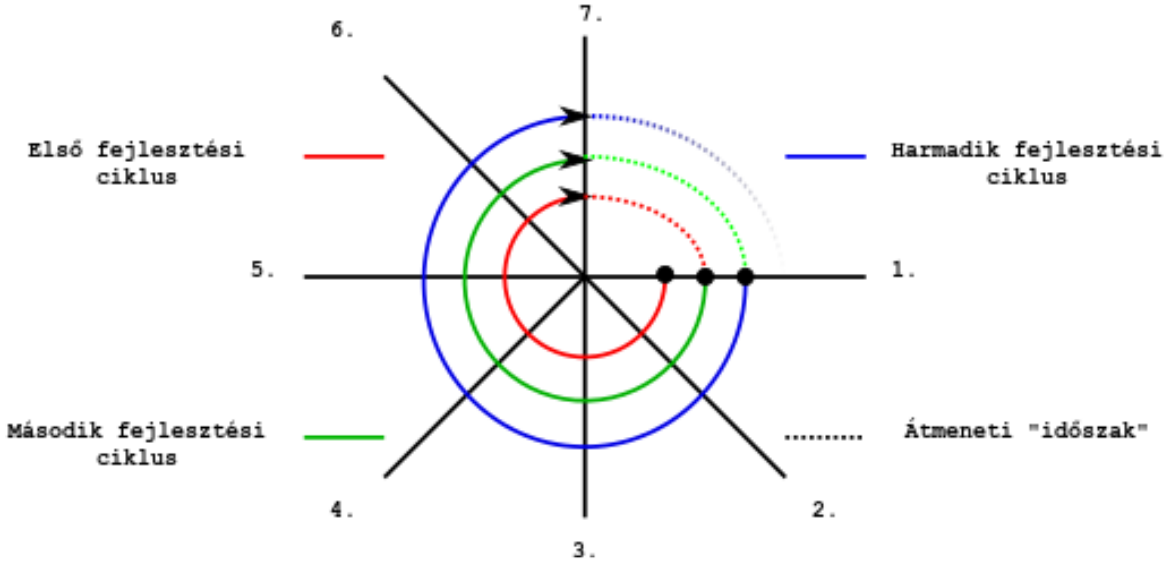
Spirál modell. Egy olyan életciklus modell, amely a teljes életciklust periodikusan ismétli.

Általában irodai szoftverek és operációs rendszerek fejlesztésénél használatos.

Néhány szakíró szerint ez nem egy önálló életciklus modell, mert az előző két modellt használják fel ciklusról ciklusra.

Előnye, hogy az első kört kivéve nem minden fejlesztést kell elölről kezdeni és lehetőség van hibák – akár tervezési hibák – javítására is.

Hátránya az, hogy jelentősebb javítások csak a következő ciklus végén várhatók.

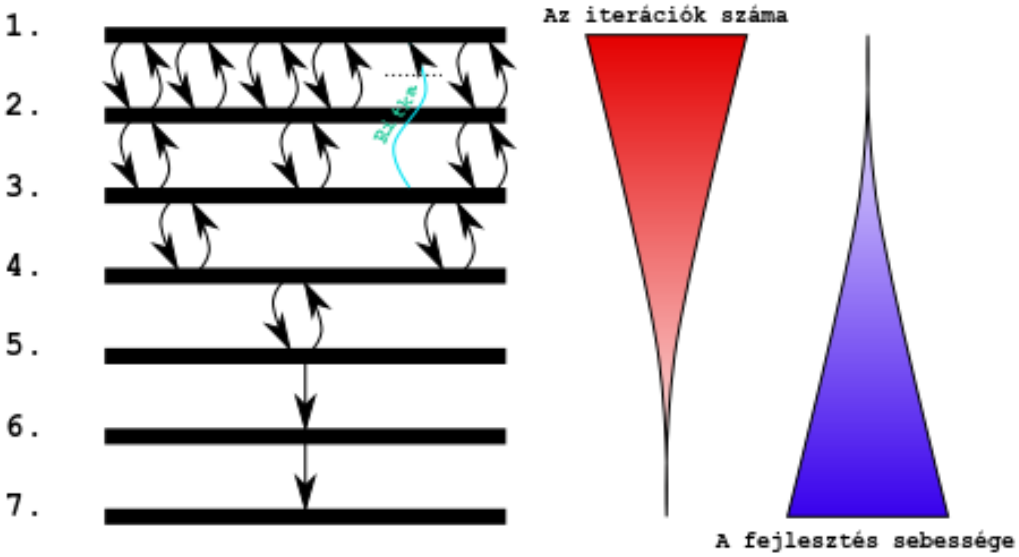


25. ábra: Ciklikus modell

Evolúciós modell. Az egyik leghatékonyabb modell. A kezdeti időszakban több elemzés, specifikáció, terv és javaslat készül. Ezeket versenyeztetik és a legjobbat kiválasztják és továbblépnek. Ha a tervezésnél problémát találnak visszalépnek az első szintre. Ezt minden egyes szintnél megteszik, ha szükséges. Nagyon ritkán, indokolt esetben két szintet is visszalépnek. Ez ahogy a projekt halad előre egyre ritkábban fordul elő. Ennek oka, hogy a tervezés nagyon jó, így egyre kevesebb módosításra van szükség.

Tehát eleinte a nagyon gyakori iteráció a jellemző, a későbbi szakaszban nagyon felgyorsul a projekt.

Tény, hogy ez a modell nagyon erőforrás igényes. Alkalmazása akkor célszerű, ha valamilyen egyedi terméket kell megvalósítani nagyon jó minőségben és kellő idő is rendelkezésre áll.



26. ábra: Evolúciós modell

7. Programozási modellek

A számítástechnikai fejlődése az 1960-as évek közepére egy olyan szintet ért el, hogy a hardverek fejlesztésének sebessége meghaladta a szoftverek fejlesztési lehetőségeit és a fejlesztés sebességét. Ezt a jelenséget szoftver válságnak nevezik és ez mind a mai napig tart.

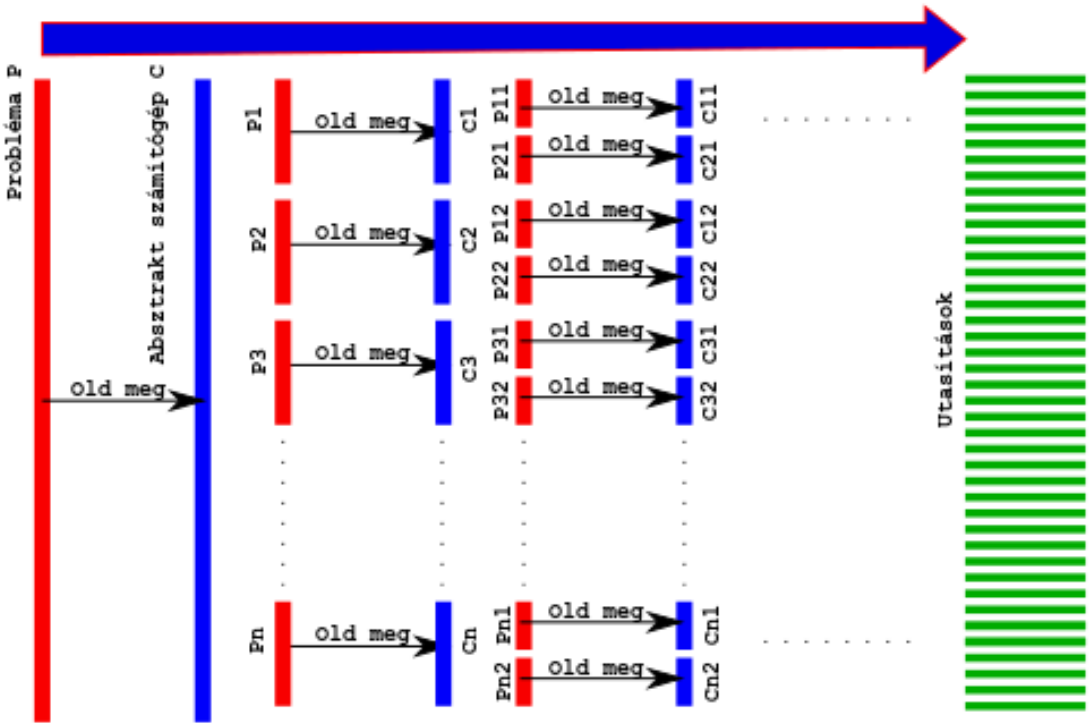
A fejlesztők viszonylag korán felismerték, hogy a gépi, majd assembly szintről egy magasabb absztrakciós szintre kell lépni és elkezdtek magas szintű programozási nyelveket fejleszteni. Ez a lépés természetesen segített, de nem oldotta meg a problémákat, mert igaz az, hogy a programozónak nem kellett már gépi szintű problémákkal foglalkoznia, de az előállított kód és az előállítás folyamata nem volt elég hatékony. Ennek lett az a következménye, hogy programozási módszertanokat fejlesztettek ki.

Programozás, kipróbálás modell. A programozó megír egy kódrészletet, majd teszteli és javítja. Alapvetően ez a módszer néhány száz sorig működhet, de ha a kód nagyobb a fejlesztés káoszba csap át, lelassul és az elkészült kód valószínűleg áttekinthetetlen, javíthatatlan és nagyon nehezen módosítható[21]. Ez az a probléma, amit a további módszertanokkal ki szeretnénk küszöbölni.

Strukturált módszertan. Az alapgondolat a következő: adott a probléma és adott egy absztrakt

21 Nem véletlenül nevezik a keletkezett programot spagetti kódnak.

számítógép. Az absztrakt számítógépet utasítjuk, hogy oldja meg a problémát. Nyilvánvaló, hogy ilyen számítógép nincsen. Ezért a problémát részproblémákra bontjuk és minden egyes részproblémát próbáljuk megoldani a hozzájuk tartozó absztrakt számítógépekkel. Valószínűleg ez sem fog menni. Ezért ezt az eljárást egészen addig folytatjuk, amíg utasítás szintre nem kerülünk. Ez az úgynevezett top – down tervezési modell.



27. ábra: A strukturált módszertan

A fizikai megvalósításban pont fordítva járunk el az utasítások szintjéről indulunk el és fokozatosan építjük fel a végső programot.

A strukturált programozási modell a felhasználható „szerkezeti” elemeket is korlátozza, összesen három alapvető „építőkockánk” van, ezek:

- szekvencia, amikor az utasításokat egymás után írjuk,
- szelekció, amikor valamilyen feltételtől függően a program futása elágazik,

- iteráció, amikor az ismétlődő feladatokat ciklusokkal oldjuk meg.

A módszertan tipikus képviselője a Pascal programozási nyelv.

Moduláris módszertan. A strukturált módszertan jelentős javulást eredményezett az elkészült kódban, Az egyetlen probléma, az volt, hogy nagyméretű kódot rendkívül nehéz volt írni a forráskód kezelhetetlen mérete miatt és nehéz volt több programozónak együtt dolgozni.

A moduláris módszertan lehetővé tette, hogy a a forráskódot modulokra bontsuk. Ezeket a modulokat önállóan lehet fejleszteni, tesztelni és járulékos nyereségként a modulokat a későbbiekben is fel lehet használni.

Lehetővé vált a bizonyos kódrészletek és változó területek a kód többi részlete elől történő elrejtése.

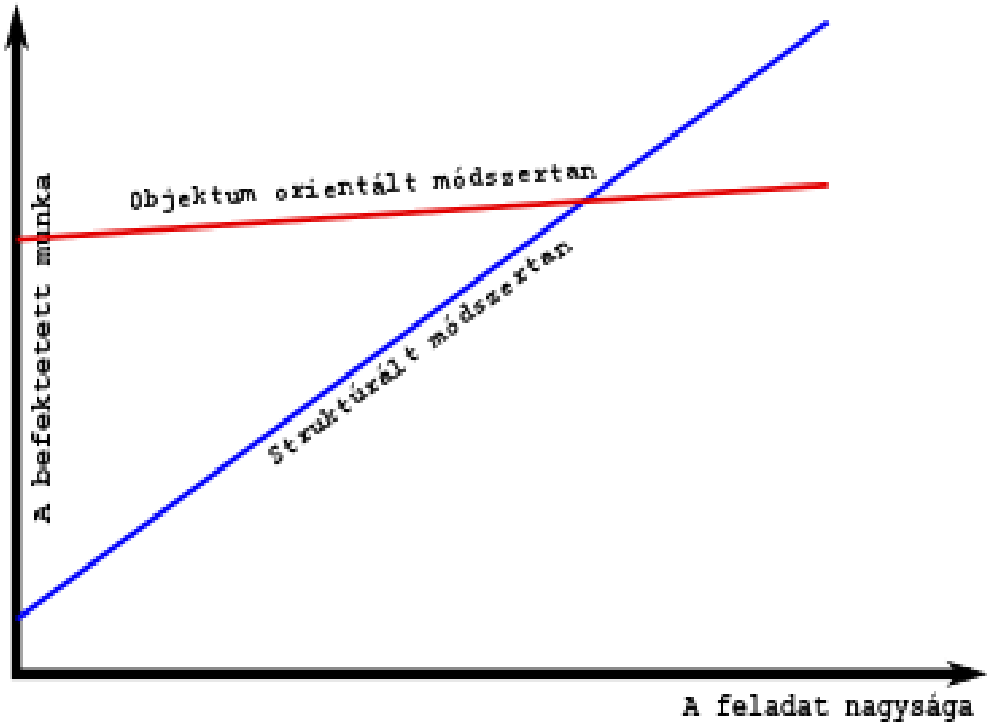
A módszertan tipikus képviselője a C programozási nyelv.

Objektum orientált módszertan. Az előző módszertanokat kibővítették három új tulajdonsággal, ezek:

- egységbezárás, ez azt jelenti, hogy az objektum önmagában tartalmazza a számára szükséges kód területet és az ehhez tartozó kódrészletet is,
- öröklődés, ez azt jelenti, hogy az objektumok (és az objektum osztályok) képesek ős – leszármazott relációban örökölni,
- többalakúság, ez azt jelenti hogy különböző eljárások azonos néven szerepelhetnek.

Az objektum orientált programozás nagyon hatékony lehet, ha sok közel azonos tulajdonsággal rendelkező program elemet kell előállítanunk, illetőleg olyan a feladat, ahol egy adott funkciót ellátó kódrészletet kell tovább bővíteni.

Az objektum orientált módszertan egyik tulajdonsága, hogy a működő kódhoz viszonylag sok előkészítő munkára van szükség, viszont jó tervezés esetén a kezdeti lassú kezdet után nagyon gyorsan lehet haladni. A Error: Reference source not found ezt mutatja.



28. ábra: A strukturált és objektum orientált módszertan összehasonlítása

8. Programozási nyelvek

8.1. Programozási nyelvek generációi

A programozási nyelvek hasonlóan a hardverhez folyamatosan fejlődtek. A szakirodalom jelenleg négy generációt különböztet meg.

1. generációs programozási nyelvek. Ide tartozik a gépi kód és az úgynevezett sor assemblerek.

   Előnyük, hogy nem igényelnek a fordításhoz komoly gépi teljesítményt, gyors a futásuk, mert a processzor közvetlenül hajtja végre az utasításokat.

   Hátrányuk, hogy gépfüggők, komolyabb feladatok esetén rendkívül bonyolulttá válik a program és ezért a fejlesztés nagyon hosszúvá válik.

2. generációs programozási nyelvek. Ide tartoznak a nagy teljesítményű makró assemblerekkel támogatott gépi nyelvek, A „korai” magas szintű programozási

nyelvek, pl. FORTRAN, ALGOL, COBOL, BASIC.

Előnyük, hogy áttekinthetőbbek, mint az előző kategória, a fejlesztés gyorsabb a magasabb absztrakciós szint miatt, kevesebb a hibázás.

Hátrányuk, hogy a fordításhoz már elengedhetetlen a számítástechnikai támogatás és a végrehajtás lassabb.

3. generációs programozási nyelvek. Magas szintű programozási nyelvek, amelyek specializáltak adott feladatra (az általános cél is feladat). Ilyen nyelvek például a PASCAL, C, C++.

   Előnyük a hatékony programozás támogatása.

   Hátrányuk a fokozott számítástechnikai támogatás igénye.

4. generációs környezetek. Ezek már nem nyelvek, hanem valóban környezetek, amelyek valamilyen harmadik generációs nyelvre épülnek, például JAVA, vagy C++. Ezek valamilyen felületen, esetleg valamilyen szabványos grafikus leíró nyelven keresztül előállítják a program kódját.

   Előnyük a nagyon gyors program előállítás.

   Hátrányuk az, hogy a keletkezett kód egyáltalán nem optimális sem méretben, sem futási sebességben. A programozónak nagyon kevés fogalma van a valós programvégrehajtásról.

8.2. Programnyelvek a kód szerkezete alapján

Imperatív programozási nyelvek. Ezek a nyelvek a klasszikus programozási nyelvek utasításokkal változókkal és vezérlési szerkezetekkel. A feladat megoldásának algoritmusát a programozónak kell megalkotnia.

Funkcionális nyelvek. Ezek a nyelvek a formális matematika jelölésmódját követik. Ezért ezt az alapelvet követve a funkcionális nyelvet tisztán függvényekre alapozzák.

Az ilyen nyelvekből hiányoznak az imperatív nyelvekben megszokott vezérlési szerkezetek (iteráció, szelekció), műveletként csak a függvények összehasonlítása,

feltételes kifejezés és rekurzió[22] áll a programozó rendelkezésére.

Logikai nyelvek. Más néven szabály alapú nyelvek. Ezek a programozási nyelvek a formális logika és a halmazelmélet alaptörvényeire épülnek.

A logikai programban a programozó szabályokat definiálhat. A program feladata olyan eset keresése, amely az adott kérdésre a programból logikailag következik.

8.3. Programnyelvek futtatás szerint

A programozási nyelveken előállított programok több módon futhatnak a számítógépen.

Interpreteres nyelvek. Az elkészült programot a futtató rendszer sorról sorra értelmezi a program futása alatt.

Az ilyen értelmező megírása viszonylag egyszerű, viszont a program futása nagyon lassú. A másik probléma, hogy a program formai hibái esetleg nem derülnek ki elég korán és így nagyon sok idő elveszhet.

Előny lehet az esetleges platform függetlenség.

Ilyen nyelv az alap BASIC, UNIX SHELL script.

Fordított nyelvek. A forráskódot a fordító program közvetlenül futtatható kóddá fordítja.

Az ilyen programok futnak a leggyorsabban. A formai hibák korán kiderülnek.

Hátrány az, hogy a lefordított kód nagyon hardver függő.

Ilyen nyelv például a PASCAL, C, C++.

Pszeudokódra fordított nyelvek. Egy átmeneti megoldás, ahol a forráskódot egy átmeneti bináris állományra fordítják, amit egy virtuális gép futtat.

Előny, hogy a formai hibák fordítás időben kiderülnek. A futás gyorsabb, mint az interpreteres esetben.

Lassabb, mint a fordított eset.

22 Rekurziónak azt nevezik, amikor egy függvény, vagy eljárás meghívja önmagát. Matematikai példa az aktuális eredmény tartalmazza az előző eredményt.

Ilyen nyelv például a korai JAVA. Most már van olyan hardver, amely közvetlenül képes a pszeudokódot futtatni.

Script nyelvek. Forrásban tárolt nyelvek, amelyeket a futás előtt automatikusan fordítja binárisra a rendszer.

Előny a platform függetlenség és hordozhatóság, továbbá a viszonylag gyors futási sebesség.

Hátrány az, hogy nem olyan gyors, mit a fordított típus[23].

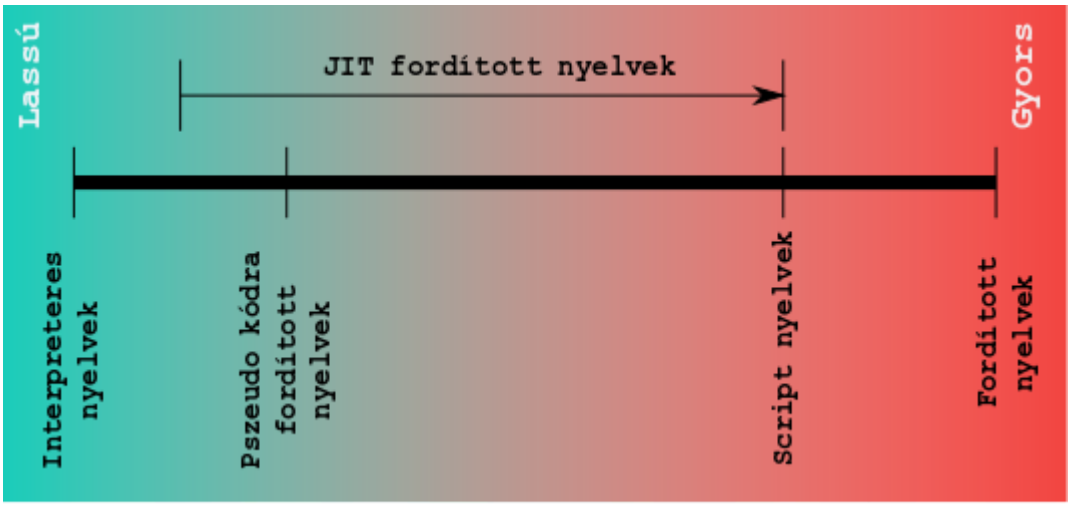
Ilyen nyelv például a PERL, PYTHON, PIKE.

JIT fordított nyelvek. Ez egy keveréke az előzőeknek. A futtatás kezdetén egy értelmező fut, de a fordított kódot tárolják és a későbbiekben ez a tárolt bináris fut.

Előnye, hogy a program azonnal elindul és hosszabb futású programok felgyorsulnak a kezdeti lassú futás után.

Hátránya a nehezen becsülhető futási sebesség.

Ilyen programnyelv például a C# és az utóbbi időben számos script nyelv.



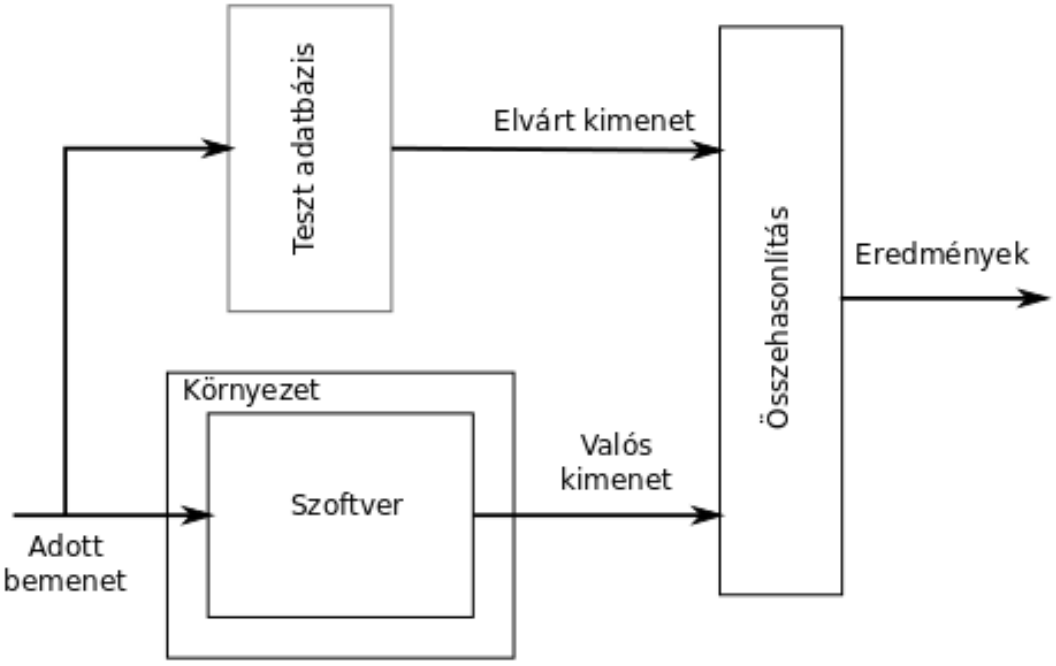
29. ábra: Programozási nyelvek futási sebessége

23 Vannak olyan fordító programok, amelyek a script nyelvű forrást binárisra fordítja és a továbbiakban ez a bináris hozzáférhető.

9. Szoftvertesztelés

A szoftverek rendkívül bonyolult rendszerek lehetnek. Tapasztalati tény, hogy ha egy rendszer bonyolult, akkor hibákat tartalmaz. Ez sajnos elkerülhetetlen. Ezért a szoftvert tesztelni kell[24].

Sajnos az is tapasztalati tény, hogy a szoftvert nem lehet teljesen tesztelni. Ennek igazolására vegyük a következő példát és a 30. ábrán látható struktúrát!



30. ábra: Egyszerű teszt vázlata

A 30. ábrán látható rendszer elemei:

- a szoftver, amelyet tesztelünk adott környezetben és feltételek mellett,
- teszt adatbázis, amely a helyes eredményt szolgáltatja,
- összehasonlító rendszer, amely a helyes eredményt és kapott eredményt összehasonlítja.

24 Egyszer találkoztunk egy gyárigazgatóval, aki kijelentette, hogy neki nincsen szüksége tesztelésre, mert minőséget gyárt. Jól van. „Köszönjük Emese!”

A mintapéldában olyan szoftvert tesztelünk, amely három darab 16 bites számot ad össze belső állapotok nélkül.

Ekkor lehetséges bemeneti állapotok száma: $2^{48} = 2.8147497674 * 10^{14}$.

A szükséges adatbázis mérete: $3 * 2^{48} = 8.4442493012 * 10^{14}$

Tegyük fel, hogy a fenti rendszer egy másodperc alatt egy millió tesztet hajt végre, ekkor a teszt kerekítve 281474977 másodpercet vesz igénybe. Egy napban 86400 másodperc van, így a teszt szintén kerekítve3258 napig, mintegy 8.9 évig tart. És ez csak három darab 16 bites változó volt, tehát ez a program rendkívül egyszerű volt.

A szoftverek komplexitása sokkal, de sokkal nagyobb. Ez azt jelenti, hogy egy ilyen jellegű teljes, vagy más néven kimerítő tesztet nem lehet végrehajtani. Ezért különböző eljárásokat és módszertanokat fejlesztettek ki.

A szoftvertesztelés két alapvető módszere a statikus és a dinamikus tesztelés.

Statikus tesztelés lényege az, hogy a programot nem futtatják, hanem „csak” olvassák. Ez persze nem egy sima kódolvasást jelent, hanem megfelelő, a tesztelési célnak megfelelő módszerekkel. Tehát a statikus tesztelés főbb szempontjai:

- a forráskódot nézzük át az összes lehetséges állapot figyelembevételével,
- minden egyes unit átnézésre kerül a követelmények szerint.
- forrás kód ellenőrzése egy választott teljesen módszeres eljárás,
- a kódot és nem a szerzőt vizsgáljuk,
- a vizsgálat tervszerűen, módszeresen történik,
- a siker alapja: "oszd meg és uralkodj", viszonylag kis részeket vizsgálunk azért, hogy ne maradjon ki semmi.

Dinamikus tesztelés esetén a kód futtatásra kerül és a tesztelési tervnek megfelelően vizsgálják a szoftver működését.

- a programrészt futtatjuk és kimenetet figyeljük,

- a kimeneti eredmények alapján megítéljük a program minőségét,

A tesztek teljesítésének szükséges feltétel, hogy minden rész jól működjön, de nem elégséges.

A tesztek befejezésének okai:

- a teszt nem mutat újabb hibákat, a következő két lehetőség fordulhat elő:
  1. valóban nincs hiba a programban,
  2. a teszt nem képes kimutatni a hibát.
- a tesztelésre szánt idő lejárt,
- ideje a terméket átadni.

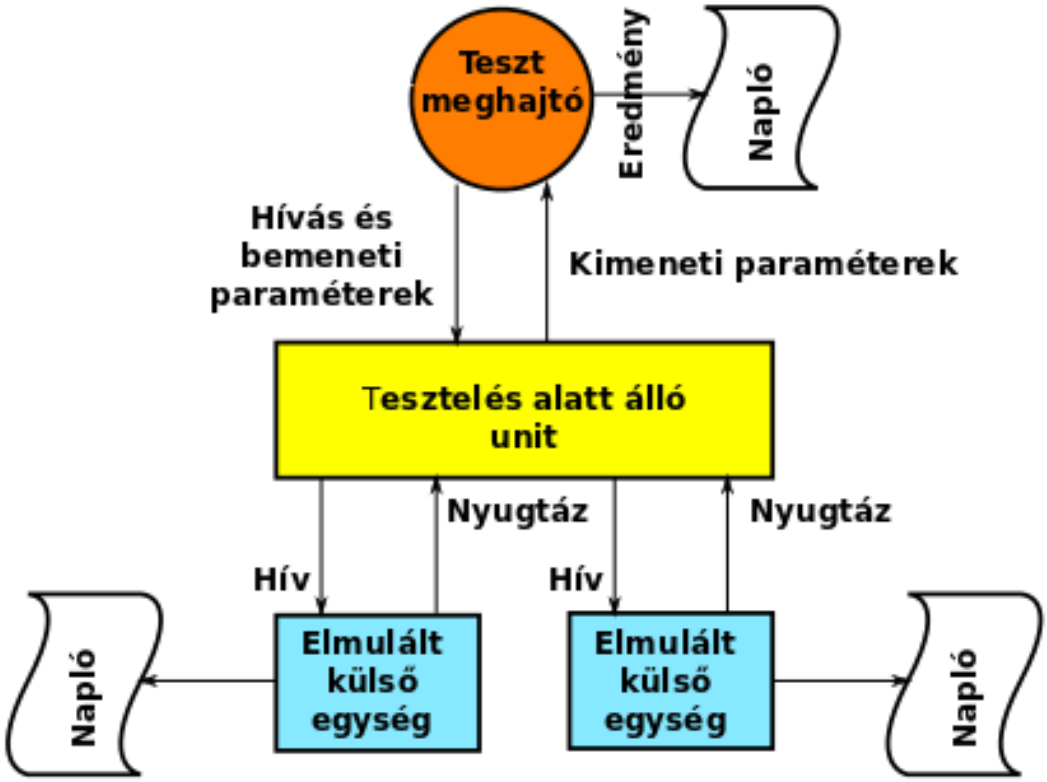
A két tesztelési eljárás nem zárja ki egymást, nem is helyettesíti, hanem kiegészíti egymást. Célszerűen a statikus tesztelés megelőzi a dinamikust tesztelést. A tapasztalat azt mutatja, hogy a felderített hibák 90 – 95%-a még a statikus szakaszban kiderül és a statikus teszt eredményei bemenetként szerepelhetnek a dinamikus tesztnek.

9.1. Tesztelési szintek

Unit tesztelés. Minden olyan programrészlet, amely bármilyen módon működésre képes unitnak (program egységnek) tekinthető. Ez lehet függvény, eljárás, vagy komplett program modul. A unitok tesztelése lehetővé teszi, hogy viszonylag korán kiderüljenek a funkcionális hibák.

A hatékony unit tesztelés megköveteli a jó programtervezést. A unitoknak célszerűen jól szeparáltnak kell lenniük, hogy elősegítsék azok tesztelhetőségét[25].

25 Igen, igen fontos, hogy a működés mellett a kérdéses unit jól tesztelhető legyen. Igazán jól tervezett szoftver működési rétegekre bontható, hasonlóan az OSI modellhez.



31. ábra: A dinamikus unit tesztelés vázlata

A 31. ábra elemeinek magyarázata:

- a tesztelés alatt álló unit, ezt a programegységet teszteljük,
- teszt meghajtó, olyan külső szoftver, vagy hardver egység, amely a kérdéses unit számára bemeneti adatokat állít elő[26],
- emulált külső egység, a unit kimeneteit fogadó hardver, vagy szoftver egység,
- naplók, minden bemeneti és kimeneti adat időbélyeggel rögzítésre kerül.
- Az időbélyeg, (ez ugyan nincs az ábrán, de megemlítettük) egy akár 0.1 μs felbontású idő adat.

A unit tesztelés számos eljárást alkalmaz, azomban ezek ismertetéséhez okvetlenül szükséges programozási ismeret.

26 Ez az egység 99.99%-ban szoftver.

Rendszer integrációs teszt. Az elkészült unitokból fel kell építeni a kész szoftvert. A unitok egymással interfészeken keresztül kommunikálnak.

Egy történet: 1999 szeptember 23-án a NASA elvesztette a kapcsolatot a Mars Climate Orbiter űrszondával. Szeptember 15-én egy pályakorrekciót hajtottak végre, melynek eredményeként az űrszondát a Mars felszínétől mintegy 226 km magasságra kellett volna vinni. Ehelyett a szonda magassága 96 és 104 km közé esett, amelynek eredményeként az űrszonda megsemmisült.

Az előző eseményhez hasonlóan egy másik űreszköz a Mars Polar Lander szintén megsemmisült.

A fenti történetben szereplő nagyértékű beredezések azért semmisültek meg, mert két szoftver unit közti interfész hibásan működött. A két modult két különböző szoftver cég készítette, az egyik unit angolszász mértékegységekben dolgozott (Lockheed Martin), míg a másik SI mérékegység rendszert használt (NASA).

A teljesség igénye nélkül a lehetséges interfész hibák:

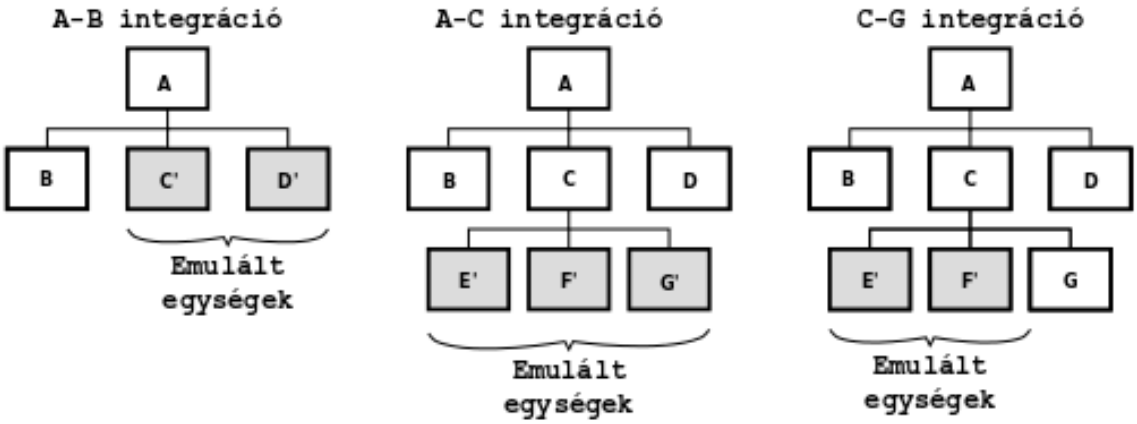
- konstrukciós hiba, az interfész nem úgy épül fel, ahogy az eredetileg tervezve lett,
- hibás funkció, az interfész szoftverhibát tartalmaz,
- a funkció helye rossz, az interfész nem a megfelelő két unitot köti össze,
- funkció változás, valamelyik funkció megváltozott, de az interfészt nem módosították,
- új funkció, egy új funkciót helyeztek el a rendszerben, de nem érhető el,
- az interfész hibás kezelése, másként kezeljük az interfészt, pl. rossz adatformátumot adnak át,
- adatstruktúra változás, az egyik unit más formátumú adatot vár, mert a unit megváltozott, de az interfészt nem módosították,
- hibás hibakezelés, az interfész a hibás adatokat és a hibák kezelését

kivételkezeléssel próbálja kiküszöbölni, illetőleg jelezni, ez a funkció is lehet hibás

- változó hibakezelés, az előző alapján a kivétel kezelési igény megváltozott, de az interfész nincs rá felkészítve,
- hibás támogatás, az interfész is használhat unitokat, ez is lehet hibás,
- inicializálási hiba, az intefrésznek lehetnek belső állapotai és ezt rosszul inicializálták,
- validációs hiba, vagy adatkorlát, az egyik unit tud olyan adatokat adni, amelyet az interfész adatábrázolás miatt nem képes kezelni,
- időzítési, vagy teljesítmény hiba, az intefész nem képes az előírt idő alatt az adatszolgáltatásra,
- hardver - szoftver problémák. számos esetben az interfész hardver függő, ha a hardver nem képes a szoftverrel együttműködni, akkor kerül elő ez a hiba.

9.1.1. Integrációs eljárások

Top-down (fentről-lefelé) integráció. A unitok összeépítése felülről lefelé történik, ezért a tesztelés is ebben az értelemben történik.



32. ábra: Top-down integráció néhány lépése.

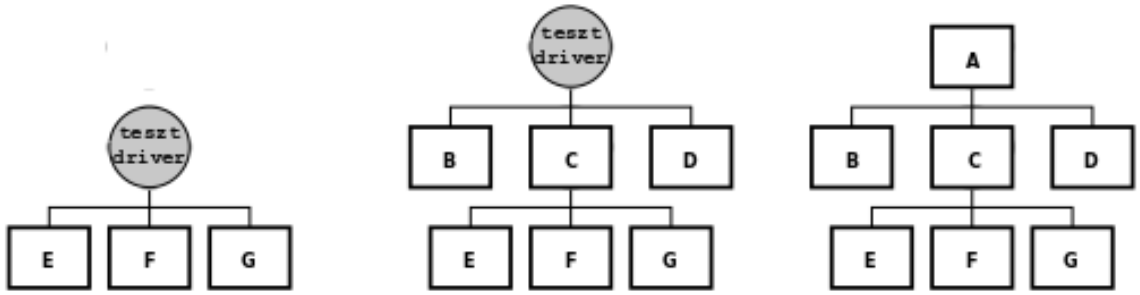
Előnyök:

- a rendszer integrációs teszt folyamatosan figyelembe veszi az integrációs folyamatot, mivel a nem integrált unitokat emulálni szükséges és ez csak rendszerszintű ismerettel működik,
- az interfész hibák elkülönítése viszonylag egyszerű, mert unitokat teszteltnek, tehát hibátlannak tekintjük, az emulált unitokat mi állítjuk elő, ezért tetszőleges – a célnak megfelelő – elemeket építhetünk be.
- A már korábban integrált unitokat fel lehet használni.

Hátrányai:

- előfordulhat, hogy lényeges rendszerfunkciók kimaradhatnak az emulációk miatt,
- az emulációk tervezése igen bonyolult lehet.

Bottom-up (lentről-felfelé) módszer. Az előző módszer ellentéte, a rendszert a „legalul lévő” unittal kezdjük és innen haladunk felfelé.



33. ábra: Bottom-up eljárás

Előnyei:

- a viszonylag egyszerű teszt meghajtókkal sokkal könnyebb felderíteni az alacsony szinten lévő hibákat,
- mivel az alacsony szinten lévő unitok gyakrabban kerülnek meghívásra, ezért célszerű először ezeket tesztelni.

Hátrányai:

- jelentős hibák – főleg tervezési hibák – felderítése eltolódhat a tesztelési folyamat végére.
- rendszerszintű tesztek nem végezhetők el a részlegesen integrált rendszer esetén[27].

Inkrementális módszer. A már elkészült és inkrementálható unitokat az elkészülés után elkezdik összeszerkeszteni és tesztelni. Így a rendszer összeállítása viszonylag korán megkezdődhet.

Jellemzői:

- várhatóan gyorsabb a termék várható ”kiszállítása",
- nagyobb hangsúlyt fektet a kis lépésekre,

27 Ez nagyon érezhető objektum orientált esetben.

- a teszt esetek számát folyamatosan növeli,
- teszteli a rendszert az automatikusan újrafuttatott teszt esetekkel,
- ösztönzi a hibák 24 órán belüli javítását,
- az eredeti verzió megtartható és ha kell vissza lehet rá térni,

Látható, hogy a verziókövetés nagyon fontos, hogy az esetleges, közben felmerült és véletlenül beépített hibákat meg lehessen találni és esetlegesen visszatérni a kiindulási állapothoz. Általában az utolsó 7-10 verziót tartják meg.

9.2. Rendszer integrációs teszt kategóriák

Interfész integrációs tesztek. Ez a teszt az összes külső és belső interfészt teszteli az érintett unit esetén.

Funkcionalitási teszt. Minden modul esetén a be és kimeneti interfészek bevonásával működési teszt.

End-to-end validáció. A tejes rendszer integrációja utáni tesztelés.

Interfész stressz teszt. Az interfészt időben és adatmennyiségben megterhelik addig, amíg az interfész nem kezd hibásan működni.

Állóképességi teszt. Az adott interfészt névleges terheléssel hosszú időn keresztül terhelik.

9.3. Rendszer teszt

A szoftver elkészült, a kiszállítás előtt természetesen tesztelni kell. A rendszerteszt típusai:

Alapvető teszt. Ez a teszt lépés vizsgálja a szoftver installálhatóságát, konfigurálhatóságát és egyéb úgynevezett külső tulajdonságokat, például a rendszer indulási tulajdonságait.

Funkcionalitási teszt. A tesztelés a specifikáció alapján pontról pontra ellenőrzi a szoftver működését. Ez a teszt nagyon szerteágazó lehet a szoftver jellegétől függően.

Vegyük példának egy ügyviteli szoftvert. Adott a specifikáció, ennek alapján professzionális tesztelők végig mennek a kezelésen. Következő lépésként olyan

emberrel teszteltetik a szoftver, aki használni fogja. Ez az ember, vagy ember csoport nem a specifikációt nézi, hanem a használhatóságot, ekkor általában néhány hiba kiderül. A legutolsó lépésként képzetlen „felhasználó” fogja a rendszert tesztelni, minimális oktatással, ha a szoftver ezt a lépést is túléli, akkor biztonságosan működik[28].

Robusztussági teszt. Ez a teszt azt vizsgálja, hogy a szoftver a különböző hibákat milyen mértékben viseli el.

A teszt során a tesztelők hibákat állítanak elő. Bizonyos hiba szintig a szoftvernek reagálnia kell. A hibák lehetnek kezelési, erőforrás és szoftver hibák.

Interoperábilitási teszt. Azt tesztelik, hogy a szoftver milyen mértékben képes más szoftverekkel együttműködni. Például képes-e a vágólapot kezelni, vagy a kimenetei szabványosak-e, mondjuk xml formátumúak-e.

Teljesítmény teszt. Vizsgálja a szoftver teljesítmény jellemzőit, a válaszidőket, az erőforrás használatot.

Skálázhatósági teszt. Vizsgálja a szoftver beállíthatóságát a környezeti, erőforrás és a földrajzi viszonyoknak megfelelően.

Stressz tűrési teszt. A teszt a szoftvert olyan körülmények közé helyezi, amelyek már befolyásolják a működését, illetve csökkentik az erőforrásokat egészen addig, míg a szoftver ezt elviseli.

Terhelési és stabilitási teszt. A szoftvert hosszú ideig maximális terheléssel üzemeltetik. Itt merülhetnek fel azok a hibák, amelyek egy rövid működés során nem léphetnek fel.

Például memória szivárgás, a rendszer lassulása, véletlenszerű hibázás.

Regressziós teszt. A szoftver stabilitását vizsgálja, miközben új elemeket installálnak, illetőleg a régieket karbantartják.

Dokumentációs teszt. Ez a teszt nem közvetlenül a szoftverről szól, hanem az elkészült termék dokumentációját vizsgálja. A dokumentáció elengedhetetlen, ezért ennek

28 Tapasztalat az, hogy profik által agyontesztelt rendszert egy képzetlen felhasználó másodpercek alatt jéggé tudja fagyasztani.

ellenőrzik olvashatóságát, teljességét és kezelhetőségét.

Regulációs teszt. A szoftvert abból a szempontból vizsgálja, hogy az megfelel-e a törvényi előírásoknak és a területi szokásoknak.

10. Szoftver minőség

A szoftverek sokkal nehezebben megfogható termékek, mint más hagyományos műszaki termékek. Ennek oka az, hogy nehéz olyan mércét találni, amely objektív módon méri a szoftver tulajdonságait.

Ezért olyan szabványokat hoztak létre, amelyek az előbb említett mércét definiálják. Ennek egyik – talán legjelentősebb – termék alapú képviselője az ISO 9126. Ez a szabvány azért is érdekes, mert más szabványok, mint például az ISO 14598 hivatkozzák.

Az ISO 9126 a már elkészült szoftverből indul ki és egy szempontrendszert hoz létre. A szempontrendszer négy alapvető kérdésre keres választ, ezek:

- minőségi modell,
- külső mérési pontok,
- belső mérési pontok,
- használati szempontok.

A méréshez hat minőségi karakterisztikát definiál:

1. funkcionalitás,
2. megbízhatóság[29],
3. használhatóság,
4. hatékonyság,

29 Erről a témakörről sokkal részletesebben fogunk beszélni a jegyzet későbbi fejezetében.

5. karbantarthatóság,
6. hordozhatóság.

Ezek a szempontok további alszempontokra bomlanak.

Funkcionalitás. A szoftvertermék megfelel-e az eredeti célkitűzésnek, illetve a specifikációnak. Ehhez elvárás, hogy a specifikáció kellőképpen definiált és teljes.

Alszempontok:

- Megfelelőség funkciók szempontjából. A szoftver milyen mértekben teljesíti a specifikációban lefektetett szempontokat.
- Pontosság. Adott a kitűzött cél, illetve a feladat. A szoftver ezt milyen pontosan teljesíti (például irányítástechnikai, CAD, vagy matematikai szoftverek esetén).
- Interoperabilitás. A szoftver milyen mértékben képes más szoftverekkel együttműködni (például vágólap használat, ki- bemenetek kompatibilitása).
- Megfelelőség a szakmai szokásoknak és törvényeknek. Vannak berögződött kezelési szokások, ezektől nem illik, illetve nem célszerű eltérni[30]. Vannak bizonyos szoftverek, amelyeknek meg kell felelniük törvényi előírásoknak (például számviteli, számlázó, vagy orvostechnikai alkalmazások).
- Biztonság. A szoftver mennyire védett illetéktelen használatnak, illetve adatlopás ellen és így tovább.

Megbízhatóság. A szoftver az előre specifikált körülmények között milyen valószínűséggel működik helyesen.

Alszempontok:

- Érettség. Milyen gyakorisággal következik be a szoftver meghibásodása.
- Hibatűrés. Ha valamilyen hardver, vagy szoftver meghibásodás történik, akkor

30 Egy jól ismert PLC gyártó egy régebbi szoftverében eltért a szokásoktól. Egy sor bevitele az ENTER billentyű helyett az INSERT billentyűvel történt és nem csak ez volt az eltérés. Azt a PLC-t szörnyű volt PC oldalról programozni.

ezt a szoftver mennyire képes tolerálni, illetve adott hiba esetén milyen mértékben marad működőképes.

- Magához térés képessége. Bekövetkezett egy hiba, képes-e a szoftver az adott hiba megszűnésével újra működőképessé válni, illetve ez mennyi idő alatt történik meg.

Megérthetőség. A szoftver vizsgálata a használati szempontok szerint.

Alszempontok:

- Megérthetőség. A felhasználó mennyire képes a szoftvert megérteni. Ha a szoftver „ránézésre” kezelhető, akkor megfelel ennek a szempontnak.
- Tanulhatóság. A szoftver funkciói, kezelési lépései, eredményei mennyire könnyen tanulhatók, ez a tanulás mennyi időt vesz igénybe.
- Kezelhetőség. A szoftvert hogyan lehet kezelni. Például egy sokszor használt funkciót mennyire könnyen lehet elérni. Használható-e grafikus felület, vagy csak parancs sor, vagy mindkettő (legjobb).

Hatékonyság. A szoftver időbeli viselkedését és erőforrás használatát vizsgálja.

Alszempontok:

- Erőforrás használat. A szoftver háttértár, memória, processzor idő használatát vizsgálja.
- Időbeli viselkedés. A szoftverek alapvetően utasításokat hajtanak végre. Ez időbe kerül. Vannak olyan szoftverek, amelyek működése erősen időhöz kötött – lásd real-time rendszerek – de egy általános ügyviteli szoftver esetén is lényeges az időbeli viselkedés.

Karbantarthatóság. A szoftver azon tulajdonsága, hogy a hibák felderítése és a hibák javítása mekkora erőfeszítést igényel.

Alszempontok:

- Analizálhatóság. A szoftver dokumentációi mennyire teljesek, mennyire jól

kezelhetők, illetve a szoftver strukturáltsága milyen.

- Cserélhetőség. Mennyire könnyű a szoftver bizonyos komponenseinek cseréje.
- Stabilitás. Karbantartási szempontból a szoftver mennyire stabil. Például verzió frissítés milyen mértékben befolyásolja a működést.
- Tesztelhetőség. Ha a szoftver jól dokumentált és jól strukturált, továbbá a szoftverben megfelelő helyeken jelzéseket helyeznek el, akkor a tesztelhetőség javul.

Hordozhatóság. A szoftver mennyire képes „idomulni” a környezet és a követelmény változásához.

Alszempontok.

- Adaptációs képesség. Alkalmazkodás a környezet és a követelmény változásához. Például a szoftver ezután más hardveren fut.
- Installálhatóság. Mekkora erőfeszítés és mennyi idő a szoftver installálása.
- Megfelelés a hordozhatósági szempontoknak. A szoftver képes-e más konfiguráción futni, például Linux és Windows alatt is.
- Kicserélhetőség. A szoftver kicserélhető-e teljes mértékben.

Az ISO 9126 egy hatékony „mérőeszköz”, de sajnos arra nem ad receptet, hogyan lehet a szoftvert „jóra írni”. Erre mind a mai napig nincs egyértelmű recept.

A szoftver minősége az elkészült szoftver alapján már a megalkotott szabványok alapján már mérhető, azonban azt is jó lenne megbecsülni, hogy a készülendő szoftver minősége milyen lesz.

A probléma az, hogy még nincs mit mérni. Ekkor az az egyetlen lehetőség, hogy a szoftvert előállító szervezetet vizsgálják. Erre ad egyfajta becslést a CMM (Capability Maturity Modell) vagyis a képesség érettség modell.

A CMM öt minőségi szintet állít fel, ezek:

- kezdeti, vagy kaotikus szint,
- ismételhető szint,
- meghatározott szint,
- menedzselt szint,
- optimalizált szint.

A CMM szintek meghatározásához a cég teljes szervezeti egészét vizsgálják. Úgy tekintik a céget mint amelyben, hogy csak egyetlen folyamat van, a szervezeti szintű folyamat, ez maga a szoftverfejlesztési folyamat, amelybe beletartoznak:

- a szoftverfejlesztésben részt vevő emberek,
- a szoftverfejlesztésben alkalmazott technológia,
- a szoftverfejlesztésben alkalmazott módszerek,
- a szoftverfejlesztésben alkalmazott eszközöket jelenti.

A szervezeti szintű folyamatnak bizonyos jellemzői / összetevői vannak, ezen jellemzők alapján dönthető el, hogy a szervezet milyen érettségi szinten áll.

Kezdeti vagy kaotikus szint. Tipikusan kezdő cégekre jellemző a tipikus túlvállalás. Ez azt jelenti, hogy több feladatot vállalnak be, mint ami a teljesítő képességük, illetve elvállalnak olyan feladatokat, amelyek jelentősen meghaladják a tudásukat.

Ennek egyenes következménye az, hogy a folyamatosan túllépik a határidő és a költségkeretet. A siker egyes személyek lététől és „hősiességétől” függ.

Egy sikeresen befejezett projekt nem jelenti azt, hogy a következő is sikeres lesz.

Az ilyen szervezetek vagy nagyon gyorsan a következő szintre lépnek,vagy megszűnnek.

Ismételhető szint[31]. Ezen a szinten már megjelenik a szoftverek konfiguráció kezelése és a cégnél létezik minőségbiztosítás. A projekteket nyomon követik, tervezik és létezik követelmény kezelés.

31 Ezt a szintet az irodalom több helyen menedzselt szintnek hívja, de mi az „ismételhető” elnevezést használjuk azért, hogy ne keverjük össze a 4. szinttel.

A cég vezetése megfelelő és kellő hozzáértéssel rendelkezik.

Ezek a szervezetek már képesek a piacon működőképesek maradni és fejlődni a következő szintre.

Egy elindult projekt sokkal nagyobb valószínűséggel lesz sikeres, mint az előző esetben.

Meghatározott szint. Minden pozitív jellemzőt „örököl” az előző szintről és ezeket javítja és kibővíti. Termék szintű menedzselés megjelenik, a különböző fejlesztői csoportok közötti kommunikáció követelmény, kölcsönös belső auditok és szemlék megjelennek. Szervezeti szintű folyamatszemlélet megjelenik.

Lényeges szempont, hogy a munkatársakat folyamatosan képzik.

Fontos megjegyzés, hogy ezek a cégek már nagyrészt specializáltak. Tehát nem foglalkoznak minden rendelés megkereséssel, csak azzal, ami a profiljuk.

Menedzselt szint[32]. A szoftver minőség menedzselése. A folyamatokat mennyiségi szempontból menedzseli. A spontán, vagy tervezett folyamatváltozások statisztikai vizsgálata.

Ennek következménye, hogy a folyamatok nagy mértékben felgyorsulnak.

Optimalizált szint. A folyamatváltozások menedzselése és optimális folyamatok keresése. A technológiai változások menedzselése, hibamegelőzés. Az esetleges eltérések, gyenge teljesítmények és hibák okainak folyamatos keresése.

A fentieknek köszönhetően a fejlesztési ciklusidők tovább rövidülnek és a fejlesztés biztonsága nagymértékben javul.

Érdekes megfigyelés, hogy csak az első szint ugorható át bizonyos segítséggel. A többi szint elérése nagyrészt folyamatos fejlődés következménye. Az is érdekes megfigyelés, hogy a felső két szint elérése nem csak tudásszint, hanem kulturális háttér függvénye is. Erre a következő példa is jó példa.

Egy jó nevű szoftver fejlesztő cég létrehozott egy leányvállalatot egy szoftverfejlesztésről

32 Ezt a szintet az irodalom gyakran mennyiségileg menedzseltként is említi.

híres ázsiai országban megcélozva az optimalizált érettségi szintet. Ez sehogy nem sikerült. Nagyon komoly erőfeszítésekkel a meghatározott szintet voltak képesek tartani[33].

11. Szoftver megbízhatóság

Átlagos megfigyelő azt gondolhatja, ha egy szoftver egyszer már hibátlanul futott, akkor az többé nem hibázik. A valóság ettől eltér.

Esetek:

- 2015 május 9-én egy Airbus A400M Atlas szállítógép berepülési feladata közben Seville közelében földnek ütközött és megsemmisült, a fedélzeten tartózkodó hat emberből négyen életüket vesztették, ketten súlyosan megsebesültek. A gép felszállás után röviddel a négy hajtóművéből háromban működési problémák léptek fel. A hiba oka szoftver probléma.
- 2015 május 1-jén az FAA[34] kiadott egy figyelmeztetést és egy utasítást egy szoftver hiba kijavítására, amely a teljes villamos rendszer kiesését okozhatja Boeing 787 Dreamliner-ek esetén. A hibát a gép generátor vezérlőjében találták.

Mivel a szoftverek – hasonlóan az emberi irányítókhoz - döntéseket hoznak, a különböző környezeti hatások ezeket is megzavarhatják. Egy több ízben hibátlanul futó vezérlő program, amely az Ariane 4 rakétán hiba nélkül futott az Ariane 5 első repülésekor a vezérlés elvesztéséhez, végül a annak megsemmisüléséhez vezetett. Az ok egyszerű: egy változó túlcsordult.

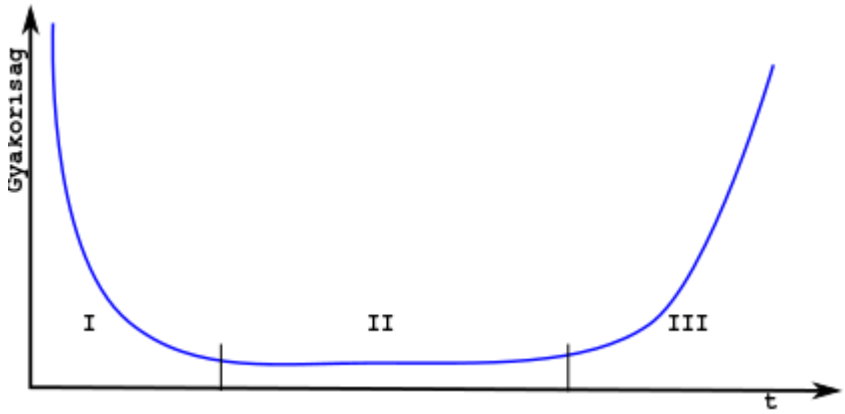
Definíció: a szoftver megbízhatóság egy adott, meghatározott időtartamon belül, adott környezetben a kérdéses szoftver hibamentes működésének valószínűsége.

Definíció: eltérés (failure) a felhasználó által észlelt váratlan szoftver viselkedés.

Definíció: Hiba (fault) az eltérés által okozott szoftver jellemző.

33 Sem cég nevet, sem országot nem nevezünk meg.
34 FAA Federal Aviation Administration az USA szövetségi légügyi hivatala.

A szoftverek ellentétben a klasszikus anyagi rendszerekkel megbízhatósági szempontból másként viselkednek.



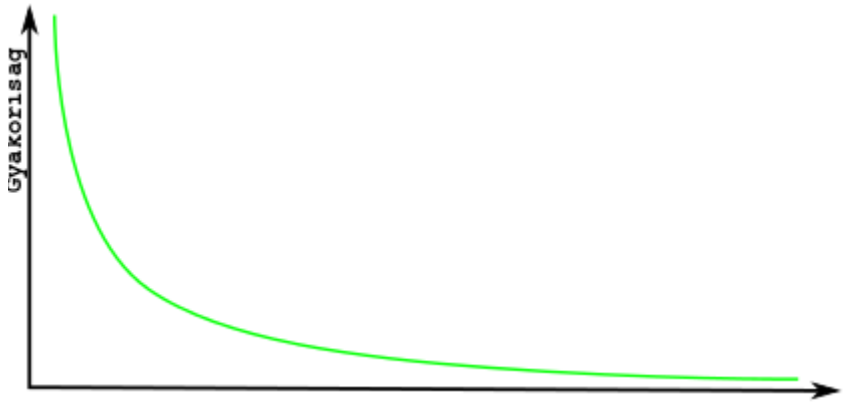
34. ábra: Anyagi rendszer hibagyakoriság görbéje.

Az I. szakasz a beüzemelési szakasz, ekkor jönnek ki a rendszer „gyerekbetegségei”. A II. szakasz a normál üzemi szakasz, ekkor a különböző terhelések hatására jelentkező meghibásodások egy viszonylagosan egyenletes és alacsony frekvenciával jelentkeznek. A III. szakasz az elöregedési szakasz, ekkor a meghibásodások egyre gyakoribbak.

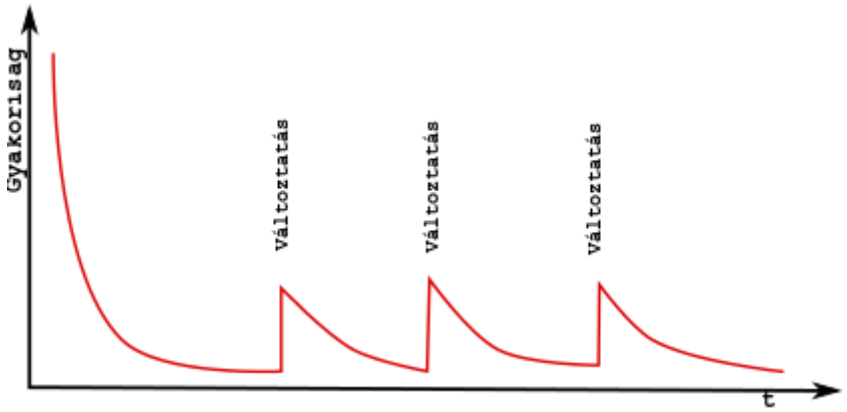
A szoftverek esetén nincs kopás, ezért a 35. ábrán látható görbét várnánk el. A szoftver használata közben egyre több eltérés és hiba kerül felszínre és ezek kijavítása egyre csökkenő hiba előfordulást okoz.

Sajnos a gyakorlat nem ezt mutatja, hanem inkább a 36. ábrán látható görbét tapasztaljuk. Ennek egyszerű oka az, hogy minden egyes beavatkozással - ami lehet csak egy egyszerű javítás, módosítás, vagy esetleg egy úgynevezett upgrade[35] - új hibákat vihetünk be a rendszerbe.

35 Időszakos „frissítés”.



35. ábra: Elméleti megbízhatósági görbe szoftverek esetén.



36. ábra: Gyakorlati megbízhatósági görbe.

A szoftverek megbízhatóságát az alábbi tényezők befolyásolják:

- emberi tényező. A programok és a szükséges adatok előállítóinak tapasztaltsága és képessége a hibamentes munkára. Hasonlóan szükséges a tesztelést végző személyzet képzettsége.
- az alkalmazott fejlesztési modell. Alapvetően a mostanság divatos fejlesztési alapelvek agile biztonságkritikus fejlesztésekben kerülendők.
- a menedzsment alkalmassága. Sokszor felmerülő probléma az alkalmatlan menedzselés. A menedzser típusú vezetők sok esetben nem értenek az adott szakterülethez – tisztelet a kivételnek – és olyan dolgokat várnak el a fejlesztőktől, amely, vagy értelmetlen, vagy teljesíthetetlen.

- az alkalmazott fejlesztői környezet hibamentessége. Számos esetben előfordul, hogy egy hibátlan forráskód a fordító program, vagy az alkalmazott programkönyvtárak hibája miatt nem működik megfelelően.
- a tesztelő környezet hibamentessége. A tesztelés lényeges része a szoftver életciklusának, így megbízhatósági szempontból lényeges kérdés. A tesztek nagy részét szükséges automatizálni, főleg, ha az állóképességi és időtartam teszt. Amennyiben a tesztelési környezet hibás, vagy hiányos egyértelműen belátható, hogy az eredményt befolyásolja.
- a hardver környezet kiválasztása. A szoftver hardver nélkül semmit nem ér. Ha a hardver kellő platformot nyújt az elvégzendő feladatokhoz elegendő tartalékkal, akkor van esély a szoftver hibátlan futására. Előfordult már olyan eset, hogy egy tesztelt hardver – szoftver konfiguráció a kiválasztott szimulációs környezetben és a valóságban több évig jól működött, de egyszer csak kifogyott a rendszer hardver a tartalékaiból és a rendszer leállt, illetve hibásan működött.

11.1. Szoftver megbízhatóság mérőszámai

A szoftver megbízhatóságot befolyásoló tényezőkre mérőszámokat kellene meghatározni. Azonban míg más mérnöki területeken ezek kiválasztása viszonylag egyszerű és egyértelmű, addig ezen a területen az objektív befolyásoló tényezők megtalálása igen bizonytalan.

A befolyásoló tényezők a teljesség igénye nélkül a következők:

- A szoftver komplexitása. Egyszerű mérőszámnak tűnik, de nem az. A kérdés az, hogy milyen mutató alapján próbáljuk ezt a jellemzőt meghatározni. Ez lehet:

    1. a program kódsorainak száma. Valóban alkalmazott mérőszám a LOC[36], illetve a KLOC[37] a (Kilo Line Of Code).

       Ez az adat kiindulásnak jó, de nem igazán mérvadó. Például: ha egy viszonylag hosszú program egyetlen szálként fut, annak komplexitása nyilvánvalóan nem éri el azon program komplexitását, amely esetleg több

36 LOC Line of Code programsorok száma.
37 KLOC Kilo Line of Code ezer programsorok száma.

szálon fut.

2. a szoftver rendszer fájljainak száma. Tapasztalt programozók sok esetben pont a túlzott komplexitás elkerülése miatt a működő rendszert több működő processzre szedik szét azért, hogy uralják a bonyolultságot és particionáltan tesztelhetővé tegyék a különböző részek tesztelését.

3. a párhuzamosan futó feladatok száma. Ez egy elég jó mérőszám lehet, de ez sem elegendő egyedül.

4. az alkalmazott IPC[38] elemek száma. Ezek a LOC mérőszámmal együtt már elég jó becslést nyújtanak, de közel sem teljeset. Az IPC elemek száma viszont utalhat a különböző speciális meghibásodások előfordulásának valószínűségére.

- A programozók tapasztaltsága. Szinte megoldhatatlan és mérhetetlen mérőszám. A szoftverek nagy része nem is egy programozó, sőt nem is azonos csoport tevékenységének eredménye. Így ez az egyébként igen fontos mérőszám nem becsülhető.

- A projekt menedzsment mérőszámai. Közismert tény, hogy egy jól irányított projekt nagyobb valószínűséggel eredményez jó eredményt. Amennyiben a projekt menedzselése támogató és normálisan értékelő, akkor a fejlesztői munka sokkal nyugodtabb és eredményesebb, mint például egy folyamatos értelmetlen számon kérő esetben.

A probléma továbbra is az, hogy erre konkrét mérőszámot nem tudunk megadni.

- Minőségi szempontok. Adott vállalatnál a létező minőségbiztosítási módszerek és a figyelembe vett szabványok is nyújthatnak támpontot a várható minőségre. Illetőleg a fejlesztő cég CMM besorolása is.

- Hiba mérték. Ez lenne a legfontosabb besorolás, mert ez ad pontos megbízhatósági

38 IPC Inter Process Communication feladatok közötti kommunikáció. Több feladatos, vagy többszálas rendszerek elemei közötti kommunikációs és szinkronizációs elemek. Ezek helytelen használata funkció kieséshez, vagy a teljes rendszer leállásához vezethet.
Az IPC részletes ismertetése túlmutat ennek a jegyzetnek a témakörén.

mutatót. A megbízhatóság időfüggő. Megbízhatósági szempontból a következő tesztelési típusok jöhetnek szóba:

1. Stressz tűrési teszt. A szoftvert (rendszert) olyan körülmények közé helyezi amely annak működését jelentősen befolyásolja, például az erőforrásokat csökkenti.
2. Terhelési és stabilitási teszt. A szoftvert maximális terheléssel hosszú ideig vizsgálja.
3. Megbízhatósági teszt. Hasonló az előzőhöz, de átlagos terhelés mellett végzik a tesztelést.
4. Regressziós teszt. Azt vizsgálja, hogy a szoftver stabil marad-e, ha új elemek jelennek meg a rendszerben, illetve a rendszer valamely része hibásan kezd működni.

Ezeknek a teszteknek az a legnagyobb baja, hogy hosszú ideig tartanak, hiszen statisztikai jellegű eljárások. A kapott eredményeket a következő mérőszámokban fejezik ki [9]:

1. A hiba bekövetkezési valószínűsége adott igényre (POFOD[39]). Alapvetően egy adott bemenetre mekkora a hiba bekövetkezési valószínűsége.
2. A hiba bekövetkezési frekvenciája (ROCOF[40]). Ezt adott időtartamra adják meg.
3. Az átlagos hiba bekövetkezés közti eltelt idő (MTBF[41], vagy MTTF[42]).
4. Rendelkezésre állás. A rendszer egy adott időtartam alatt mennyi ideig áll rendelkezésre és mennyi ideig tart a karbantartása.

További probléma a tesztelési környezet helyessége. Nyilvánvaló, hogy a hosszú távú tesztelést teljes egészében nem végezheti ember. Ezt automatizálni kell. Ha a teszt környezet hiányos, akkor bizonyos szoftver részletek nem kerülnek tesztelésre, illetőleg az automata teszter nem vesz észre eltéréseket.

39 POFOD Probability of Failure on Demand.
40 Rate Of Occurrence Of Failures.
41 Mean Time Between Failures.
42 Mean Time To Failure.

11.2. Hiba és eltérés

Sajnos a magyar nyelvben a failure és a fault angol kifejezésekre a fordítás minden esetben a hiba. Ezért kissé önkényesen fordítottuk a failure kifejezést eltérésnek és a fault kifejezést hibának.

Már a definícióból kiderül, hogy egy működési eltérés nem minden esetben okoz hibát. Elképzelhető, hogy egy avatott szemlélő észreveszi, hogy az adott rendszer kis mértékben eltér az előírt működéstől, de ez nem biztos, hogy a rendszer eredő működési jellemzőiben látszik. Az az eltérés, amely már rendszer szinten látható, az már hiba.

A hibák besorolása a következő lehet [2]:

- nem jelentős (lényegtelen),
- jelentős (kritikus),
- ismert,
- ismeretlen.

Nem jelentős hiba esetén, akár ismert akár ismeretlen, nem várhatunk komoly rendszer működési zavart. Ha azonban a hibát sikerült felderíteni, akkor záros határidővel azt javítani kell. Nem biztonságkritikus esetben egy kockázatelemzés után előfordul, hogy a hiba javítatlan marad.

Példaként említhetünk kisebb színezési hibákat.

Valahol a hiba és az eltérés közé sorolnánk a Boeing 787-esek generátor problémáját. Egyértelműen komoly működési probléma, de ehhez a hajtóműnek valószínűtlenül hosszú ideig kell működnie.

Ezen hiba javítása szükséges, de nem kell a 787-es flotta földre parancsolása.

Kritikus hiba esetén a javítás nem halasztható, azt azonnal végre kell hajtani. A legnagyobb biztonsági problémát az ismeretlen kritikus hibák okozzák. Nem ismert, hogy milyen körülmények között következnek be és nem ismert hatásuk sem. Lásd Sevilla-i katasztrófa.

Első pillanatra nyilvánvalónak tűnik a hibák azonnali javítása, azonban a javítás is hordozza a

hiba lehetőségét. Rendkívül fontos a javított szoftver alapos átfogó ellenőrzése.

Természetes módon felmerülő kérdés, hogy miért nehéz az eltérések és a hibák felderítése. A válasz a következő pontokban adható meg:

1. a hiba csak bizonyos esetekben számos körülmény együttes fennállása esetén következik be,
2. a hiba normális körülmények között nem következik be, vagy hatása észrevétlen marad,
3. a hiba folyamatosan fejlődik, majd egy adott szint elérésekor rendszerszintű problémát okoz.

Az első eset tipikus konkurens rendszerek esetén. Hibás tervezés esetén adott konstellációban a teljes rendszer, vagy egyes részei blokkolttá válnak.

A második esetben a rendszer a lehetséges állapottér olyan állapotába kerül, amelyre nincs felkészítve és rosszabb esetbe erre választ is ad. Ilyen szerencsétlen eset lehet az úgynevezett halott kód futása[43].

A harmadik hiba nagyon banális lehet. A rendszer valamilyen erőforrást folyamatosan és észrevétlenül fogyaszt. Erre tipikus példa a naplózás háttértárra, amely a tárolóterület elfogyásához vezet, vagy a memória szivárgás.

11.3. A megbízhatóság növelése

Elsődleges cél a megbízhatóság növelése. A lehetséges módszerek a következők:

1. Helyes tervezési módszerek alkalmazása. Biztonságtechnikai és kódolási eljárások betartása, mint például MISRA[44], illetve az ide vonatkozó szabványok figyelembe vétele IEC 61508[45].
2. Az előállított adatok és programok minél kimerítőbb tesztelése akár azon az áron is,

43 Halott kódnak nevezzük az olyan kódrészletet, amely normál körülmények között nem fut, viszont a programban szerepel.

44 Motor Industry Software Reliability Association az autógyártók által létrehozott ajánlás, amely a programozókat „korlátozza” abból a célból, hogy az eltérő programozási stílusok ne okozzanak félreértéseket és ezáltal hibákat.

45 IEC International Electrotechnical Commission nemzetközi elektrotechnikai szabványügyi szervezet.

hogy egy időben több párhuzamosan futó tesztelési projekt fut. Fokozottan érvényes ez a hosszú távú, terheléses és regressziós tesztekre.

3. Az előállított kód legyen hibatűrő, kisebb eltéréseket és hibákat képes legyen korrigálni azért, hogy ezek ne vezessenek komolyabb rendszer működés eltéréshez.
4. Olyan szoftver egységek használata, amelyet már többször felhasználtak és hibátlannak minősültek. Ez sem tekinthető végső megoldásnak. Ennek oka, hogy a felhasznált elemek olyan körülmények közé kerülhetnek, amelyben nem működhetnek helyesen.
5. Az előállított kód tartalmazzon biztosítékokat az esetleges komolyabb hibák időben történő felderítésére és ezen hibák kialakulásának megakadályozására.

A probléma továbbra is fennáll, minden redundancia jellegénél fogva növeli a komplexitást. A komplexitás növekedése viszont a hiba lehetőségét hordozza magában. Volt már arra példa, hogy egy jól működő rendszert a biztonságot szolgáló szoftver elem tett használhatatlanná.

A szoftver gyártók és felhasználók a kockázati elemeket még a szoftver kibocsájtása előtt szeretnék meghatározni. Erre az előbb említett módszerek nem alkalmazhatók.

A megoldás a szoftver előállítójának megfigyelése. Az előállító várhatóan több szoftvertermékkel rendelkezik, amelyből éles körülmények között számos példány fut. Az ilyen szoftverelemek adatainak összegyűjtésével már értékelhető statisztikai mintával rendelkezünk.

De mi van akkor, ha az előállított szoftver egyedi példány, vagy nagyon kis számban készül. Például egy mesterséges hold fedélzeti szoftvere.

Ekkor nincs más lehetőség, mint a kérdéses cég előzőleg megírt szoftvereinek vizsgálatára. Azok alapján már tudunk statisztikai analízist végezni. Ezáltal a megírandó kód megbízhatóságára képesek leszünk megbízható becslést nyújtani. Ez a gondolat párhuzamot mutat a CMM-el, ahol szintén az előállító szervezet kerül vizsgálatra.

Az összes eddigi fenti módszer valamilyen időalapú statisztikát feltételez. Ezeknek a módszereknek az a legnagyobb hibája, hogy valóban a futási időt tekintik alapnak. Azonban a hardverek futási sebessége lényegesen különbözhet, akár ezerszeres sebességi eltérések is

elképzelhetők még azonos szoftver alapú rendszerekben is. Például a QNX operációs rendszer futhat egy egykártyás vezérlő, de futhat összetett számítógép rendszeren is. Ekkor az idő nem szerencsés független változó a statisztikai vizsgálatokra. Ezért újabban a végrehajtott programsorokat tekintik a független változónak, amely sokkal jobban tükrözik a megbízhatósági jellemzőket.

A statisztikai vizsgálatok mellett új kutatási témaként megjelent a fraktál alapú vizsgálat, amelyből a szoftvert előállító szervezetre lehet következtetéseket levonni.

Felhasznált irodalom

Administration, http://mdp.ivv.nasa.gov/index.html

Dr. Schuster Görgy: CMM előadás

Dr. Schuster György: ISO9126 előadás

Dr. Schuster György: Szoftver tesztelés kurzus 1-9

http://istqbexamcertification.com/what-is-a-software-testing/

http://mars.jpl.nasa.gov/msp98/lander/

http://www.bbc.com/news/technology-32810273

http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/

http://www.ministryoftesting.com/2016/01/what-do-software-testers-do-version-0-1/

http://www.seattletimes.com/business/boeing-aerospace/faa-orders-787-safety-fix-reboot-power-once-in-a-while/

http://www.slideshare.net/RiantSoft123/different-types-of-software-development-model

http://www.space.com/14242-russia-spacecraft-phobos-grunt-crash-earth.html

http://www.space.com/14242-russia-spacecraft-phobos-grunt-crash-earth.html

https://en.wikipedia.org/wiki/2015_Seville_Airbus_A400M_Atlas_crash

https://en.wikipedia.org/wiki/Operating_system

https://en.wikipedia.org/wiki/Qantas_Flight_72

https://hu.wikipedia.org/wiki/Fobosz-Grunt

https://hu.wikipedia.org/wiki/Markov-folyamat

https://hu.wikipedia.org/wiki/Weibull-eloszlás

https://users.ece.cmu.edu/~koopman/des_s99/sw_relia bility

https://www.google.hu/gfe_rd=cr&ei=NMyAWPKbO8mEaIy_hKAJ#q=software+life+cycle+models+bundeswehr&start=10

Infotech Ltd, Oxford-New York-Toronto. Pan, Jiantao (1999): Software Reliability,

Kun, I., Szász, G., Zsigmond, Gy. (2002): Minőség és megbízhatóság I.-II., LSI, Budapest.

Long, J. (ed.) (2008): Metrics Data Program, National Aeronautics and Space

Mellor, P., Bendell, A. (eds) (1986): Software Reliability. State of the Art Report, Pergamon

Porkoláb, Z., Pataki, N., Sipos, Á. (2006): Jelenleg használatos szoftvermetrikák összehasonlító elemzése, Tanulmány, GVOP-3.2.2.-2004-07-0005/3.0, Budapest.

Schuster György - Terpecz Gábor Inkrementális fejlesztés és tesztelés biztonságkritikus rendszerek esetén

Schuster György, Terpecz Gábor, Tokodi Dániel Szoftver megbízhatóság Repüléstudományi közlemények 2016 2. szám