Módszeres programozás
avagy
Programozási módszertan

1.4. kiadás
írta Fábián Zoltán
Budapest, 2005

1 Bevezetés

Amikor elkezdtem programozással foglalkozni, még a Commodore 64-es gépek voltak divatban, és az amatőrök kedvenc programozási nyelve a BASIC volt. Hamarosan a nyelv minden csínjával-bínjával tisztában voltam, majd rávetettem magam a gép gépi kódú programozására is. Sokat programoztam. Mégis nagy gondom volt, hogy körülbelül 1000 sor hosszú programok írásakor a programjaim már nem sikerültek igazán, tele voltak hibával, megmagyarázhatatlanul viselkedtek, áttekinthetetlenekké váltak. Ekkor kezdtem el az egyetemet, az informatika szakot. Az első évek egyikében volt egy „Módszeres programozás” című tárgyunk. A kurzus végére rájöttem, hogy korábbi programjaim nem is működhettek hibátlanul, mert a kezemben nem volt semmi módszer, amitől hibátlanok lehettek volna.
A problémák megoldásának módszerét szeretném megosztani az olvasóval. Természetesen a kurzus címe mi más is lehetne, mint „Módszeres programozás”.
A jegyzet során feltételezem, hogy a jegyzet használója bizonyos elemi programozási ismeretekkel rendelkezik, legalább egy nyelven írt már működőképes programokat. Azt is javaslom a kedves olvasónak, hogy a jegyzetben leírt módszereket, eljárásokat próbálja ki a gyakorlatban is. A jegyzet anyaga az oktatás során csiszolódott ki.

Fábián Zoltán fz@szily.hu

Budapest, 2005. január 27.

2 Bevezető, avagy mért kell módszeresen programozni?

Jegyzetsorozatunknak ebben a részében arra keresünk választ, hogy miért és hogyan kell nagyobb lélegzetű programozási feladatokat megoldani. Betekintést kapunk nagyobb rendszerek fejlesztésének módszereibe is.

Nem kívánunk teljes áttekintést adni a programozás módszertanáról, és nem kívánunk hosszas elméleti fejtegetésekbe bocsátkozni, azokat meghagyjuk az egyetemek és főiskolák programozói kurzusainak, de azt világosan szeretnénk leszögezni, hogy a programozás folyamata – nem nyelv-specifikus, mindig ugyanazokon az elveken alapul. A programozás alapvetően egyfajta gondolkodási, algoritmizálási, feladat-megoldási eljárás, azon kívül alkotó munka.

A programozók között is vannak kezdők és haladók. Nyilván először megnézzük, hogy milyen is a kezdő?

2.1 A monolitikus programozás

A programozás tárgyalásánál nem kezdhetünk úgy egy könyvet, „hogy már az ókori görögök is ....", mivel az ókori görögöknek nem volt még számítógépük, de igaz ma már történelem, de az 1950-es évektől kezdve azért voltak már ilyen gépek, voltak programozók és írtak programokat is.
Az időszak mai szemmel egyszerű programjai más technikával készültek, mint a maiak, talán sokkal kevésbé voltak összetettek ezek a programok. Az mindenesetre igaz, hogy ezek a programok általában egy-egy programozó kezétől születtek. Azt lehetett mondani, hogy

egy program – egy programozó.

Az ellátandó feladatok nem voltak túl bonyolultak, az egy programozó a megoldások algoritmusait átlátta, a programokat lekódolta, a programok általában lineáris felépítésűek voltak, egy-egy elágazással és egy-egy ciklussal. A programokat viszonylag könnyen és gyorsan meg lehetett írni. A programoknak nem volt belső struktúrájuk. Ezek voltak a monolitikus programok és a programozási módszert **monolitikus programozásnak** hívjuk.

2.2 A kezdő programozó - frontális támadás módszere

A kezdő programozó miután megírt több apró – egyenként néhány tucat soros - programot, gyakran azt hiszi, hogy akkor néhány ezer soros programok a fejlesztése is ugyanaz lesz, mint a rövidebbeké, csak tovább tart. Ő a monolitikus programozás időszakánál tart.

A kezdő programozó a programozás tanulása során felcsipegeti a tudásmorzsákat, a trükköket, a megszakítási címeket és a spéci megoldásokat. Esetleg optimalizálási megoldásokról is hall, sőt dokumentálásról is olvasott.

Egy új, esetleg összetettebb feladat megoldásához is úgy viszonyul, mint a korábban megírt rövidebb programokhoz.

Nagy vonalakban átgondolja az adatszerkezetet, a megoldási módszert, amit a feladat elején még nem lát. Arra majd időben kitalál valamilyen megoldást – gondolja magában. Rövid tervezgetés után, előveszi a szeretett és jól megtanult programozási nyelvet (4GL vagy egyéb fejlesztő eszközt) és elkezdi írni a programot, az első szótól folyamatosan haladva az utolsó felé. Természetesen, mivel a program hosszabb, mint szokott, ezért gyorsan újabb és újabb eljárásokat talál ki a feladatok megoldására, újabbnál újabb alacsonyszintű rendszerkezelő programrészletet dolgoz ki. Lehetőleg a program minden részével foglalkozik – frontális támadás módszere -, mivel csupa globális változót használ, és az esetlegesen meglévő programmodulok ezeken a változókon keresztül kommunikálnak. A program írása az utolsó sor utáni END.(csukó kapcsos zárójel vagy RETURN, stb.…) szavakkal végződik.

Mivel az így megírt program általában nem működik, a fejlesztő mágus hamarosan megunja az állandó javításokat. A programot javítani senki sem tudja, még ő sem. A javítások újabb problémákat vetnek fel.

Viszonylag gyorsan lehet kis programokat fejleszteni, de a programot már az első pillanattól a fejében kell tartania fejlesztőnek.

2.3 A moduláris programozás

Köztudott, hogy egy ember egy bizonyos bonyolultság után a teendőket nem tudja átlátni, csak ösztöneire, megérzéseire hagyatkozhat. A programokat ösztönből azonban nem lehet megírni.

A moduláris programozás azt jelenti, hogy a problémát olyan részfeladatokra bontjuk, amelyeknek a bonyolultsága már nem okoz gondot, amit már egy modulban – monolitikusan meg tud írni egy programozó, azaz csökkentjük a probléma bonyolultságát.

Ha több ember együtt dolgozik egy munkán, akkor az elvégzendő feladatot szintén részekre kell bontani, és a részeknek az összekapcsolását, összekapcsolódását meg kell tervezni. Itt is a megfelelő megoldás az, hogy a programokat bontsuk modulokra. A részek közötti együttműködési felületet interface-nek hívjuk, a programozási módszert **moduláris programozásnak**.

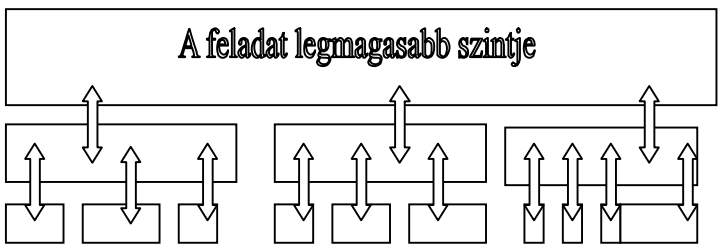
A továbbiakban felmerül a kérdés, hogy milyen elvek alapján, hogyan bontsuk részekre a programot?

2.4 Top - Down dekompozíciós módszer

A Top – Down módszer abból indul ki, hogy a feladatot nagyobb viszonylag egymástól független egységekre bontja. Ezek az egységek csak **jól definiált interface**-eken kommunikálnak egymással, más kapcsolat az egyes modulok között nem lehet.

Az egyes részek fejlesztői kívülről csak dokumentált definíciókat használhatnak fel.
Ha egy részleg a fejlesztés során rájön, hogy a mások által is használt külső paraméterek nem illeszkednek megfelelően az általa megoldandó feladathoz, akkor két dolgot tehet:

1. Saját hatáskörében kidolgozza a külső kapcsolatok olyan belső értelmezését, amely már a céljainak megfelel, de kifelé csak a szabványos felületen érintkezik a többi modullal.
2. A többi modul fejlesztőjével értekezve javasol egy olyan megoldást a közösen használt külső paraméterekre, amelyek mindenki számára megfelelőek.



A feladat legmagasabb szintje

A feladat részekre bontva

A feladat további finomítása

Mind a két módszernek vannak előnyei és hátrányai. Az első esetben a kérdéses modul megvalósítása az optimálisnál bonyolultabb lehet, következésképpen lassabb, esetleg a szükségesnél nagyobb erőforrást lefoglaló. Ebben az esetben a belső modul és a külső között lennie kell egy konvertáló modulnak is, amely, amely a belső, megváltoztatott szabványok által adott eredményeket átfordítja a külső kapcsolatok által megkívánt formára. További problémák forrása lehet, ha a belső módosítás esetleg csak részben teljesíti a többi modul által megkívánt feltételeket. Ha a többi modul kíván változtatásokat a közös paramétereken, akkor a belső nem egyeztetett megvalósítás akadálya lehet a közös, módosított paraméter rendszernek.

Előnyére válik a megoldásnak, hogy nem kíván egyeztetést a többi modul megvalósítójával, belső ügy maradhat, a módosítás a többieket nem feltétlenül érinti.

A második módszer mindenképpen szimpatikusabb, ugyanis a módosítások közkinccsé válása során az esetleges más által kezdeményezett módosítások is átgondoltabbá válnak. Az egyeztetés néha persze lehetetlen és mindenesetre több átgondolást kíván, de akkor később kisebb a feltételütközések lehetősége, az egyes modulok kapcsolódási pontjai továbbra is szabványosak maradhatnak, a teljesítmény az elvárható optimális közelében mozog.

A Top-Down módszer lényege, hogy a megfogalmazott **részfeladatokat további részfeladatokra osztjuk**, amelyeket további részfeladatokra és így tovább. A megoldandó problémát ésszerű határig bontjuk egyre alacsonyabb szintű.

Meddig folytassuk a részekre bontást?

Általános recept nincsen, de az a szint biztosan az alsó határ, amikor a modult már programozói szinten átlátjuk. Előfordulhat, hogy egyes modulokat lebontunk, míg más modulokat készen kapunk kidolgozva. A részekre bontás folyamatában figyelni kell az egyensúlyra. Törekedni kell arra, hogy a megoldandó feladat minden részén egyenletesen haladjunk előre a részekre bontás folyamatában. Ezt az **„párhuzamos finomítás" elvének** hívjuk. Az elv alkalmazásánál a finomítással együtt finomodik az adatstruktúra is. Tehát az adatmodell és az eljárásmodell együtt közeledik a végső cél felé.

Hogyan lehet kódolásnál a módszert alkalmazni?

Az elkészülő program vázát építjük fel, a bejelentkező menüt írjuk meg, a fő funkciókat megjelenítjük a képernyőn. Ha például a program elején megjelenik egy menü, akkor megírjuk azokat a modulokat, amelyek a legfontosabbak, pl. fájl megnyitás, keresés, bezárás, a többi modult beírjuk a programba, de csak „üres" eljárásként, azaz a nevét adjuk meg és az eljárás átveszi az esetleges paramétereket. Ha szükséges, akkor előre megadott teszteredményeket ad vissza.

A kódolásnál is figyelni kell arra, hogy a modulok megírását is lehetőleg egyenletesen végezzük. Ne fordulhasson olyan eset elő, hogy a program egyes részei kiválóan működnek, míg más részek esetleg alig vannak megírva.

Megjegyzés:

A Top- Down módszer rövid kis programoknál nem igazán hatékony, mivel a végeredmény csak viszonylag sok fejlesztői befektetés után jelenik meg. Nagy projektek megalkotásánál viszont a kezdeti hosszabb előkészítő munka meghozza a gyümölcsét. Kevesebb lesz a logikai, adatszerkezeti hiba, a nyelvi függés és az improvizatív megoldás a programban. Áttekinthetőbb lesz a program kód is. Könnyebb kialakítani a következetes névmegadási konvenciókat is.

A legtöbb programozási nyelven meg lehet valósítani az ilyen fejlesztést, sőt a 4GL nyelvek egyenesen támogatják az ilyen fejlesztést azáltal, hogy a megfelelő objektumok megalkotásánál felkínálnak előre megírt programmodul vázakat, amelyeket később csak ki kell tölteni a megfelelő tartalommal.

-

2.5 Bottom-Up kompozíciós módszer

Ez a programfejlesztésnek egy másik sokat alkalmazott módszere. Elsősorban rövid, speciális programoknál célszerű használni. Lényege, hogy alulról felfelé építjük fel a programokat. Megírjuk az elemi építőköveket, algoritmusokat, majd azokból építjük össze a magasabb szintű struktúrákat, majd a programot összeállítjuk.

Ennél a módszernél a programozónak teljes rálátással kell rendelkeznie a megoldandó problémára. A részletek megoldásánál ügyelnie kell az esetleges kapcsolatokra is.

Elsősorban hardverközeli programozásnál lehet használni a módszert, ahol is a legfontosabb a hardverhez illeszkedő programrészek megírása, és ezek a programrészek esetleg alapvetően befolyásolják a program többi részének működését is. A hardverrel kapcsolatos fejlesztéseknél az is fontos, hogy a programozó viszonylag gyorsan meggyőződjön elgondolásai helyességéről, ezért olyankor nem is szokott, nem is akar komplett felhasználói felületet adni a programjának, hiszen a egyszerűbb paraméterezés a programok funkcióinak ellenőrzésére éppen elegendő. A hardverek programozásánál eleve léteznek olyan módszerek, amelyek a strukturált programozás szabályait felrúgják, ún. trükköket alkalmaznak.

Megjegyzés:

Illusztrációként álljon itt egy példa: Assembly nyelven írt programok esetén tipikus, hogy a rövidebb kód érdekében egy RETURN parancsot lehagynak. Az alábbi példában egy meghívott rutin meghív egy másik eljárást:

„Szabályos” példa	*„Trükkös” példa*
;belépési pont	*;belépési pont*
…..	*…..*
…..	*…..*
JSR Másik_Eljaras_cím	*JMP Masik_Eljaras_cim*
RETURN	

A fenti példában a RETURN utasítást megspórolták, mivel a „Masik_Eljaras_Cim”-en kezdődő rutin végén feltehetőleg van egy RETURN, ami majd visszaviszi a programunkat az eredeti hívási helyre. Ez a trükk 1 byte megtakarítást és ami még fontosabb egy RETURN végrehajtásának megtakarítását eredményezte. A RETURN parancs során több belső művelet zajlik le, ami a program sebessége szempontjából nem lényegtelen, hogy hányszor zajlik le.

A Down - Top módszer igazából nem alkalmas nagyobb rendszerek tervezésére, mivel a fejlesztő először a részletekkel törődik, majd abból építene egységes egészet, ami általában nehezen sikerül neki.

2.6 Vegyes módszer

A fent említett két módszer tisztán a gyakorlatban csak nagyon ritkán jelenik meg. A leggyakoribb eset az, hogy a program fejlesztése során a program vázát a Top Down módszerrel tervezik meg, és finomítják ameddig a hardver-közeli, rendszer-közeli részekhez nem érnek, ugyanakkor bizonyos sebesség- vagy memóriakritikus részeket a Down - Top módszerrel oldják meg. Az így kialakult részeket, azután úgy illesztik össze, hogy mind a Top – Down módszer során megalkotott egységes felületekbe beilleszkedjenek a Down Top módszer konkrét megoldásai.

2.7 További programozási elvek

A továbbiakban olyan programozási elveket fogalmazunk meg, amelyek akár a program algoritmusának elkészítése, akár a kódolás során jól használható elvek.

2.7.1 Taktikai elvek

Niklaus Wirth egy cikkében az alábbi elveket fogalmazta meg a programok modulokra való bontásáról:

A párhuzamos finomítás elvét.
Korábban már említettük

Visszatérés az ősökhöz
A programok fejlesztésének bármely szintjén világossá válhat, hogy az általunk választott megoldás nem vezet célhoz. Ekkor olyan pontra kell visszatérni a programozás bármelyik szakaszában, ahonnan lényeges változtatást tudunk végrehajtani az eredeti megoldáshoz képest.

A döntések elhalasztásának elve
A programozási feladatok gyakran annyira összetettek, hogy a programozó nem láthatja át a megoldás minden aspektusát. Ekkor kell felhasználni az elvet. Mindig úgy programozzunk, hogy egy időben csak egy problémát oldjunk meg.

Döntések nyilvántartásának elve
Ha a program fejlesztése közben egy ponton szűkítjük a továbbiakban egy adat értelmezési tartományát, vagy a program többi részére is kihatással levő döntést hozunk, akkor azt a döntés pillanatában dokumentálni kell és a továbbiakban az illető adatokra való minden hivatkozás során figyelembe kell venni az értelmezési tartomány szűkítését. Például, ha egy adatbevitel során csak a pozitív egész számok jöhetnek szóba, akkor az algoritmusban és a kódolásnál is a bevitel helyén kell a bevihető adatok körét szűkíteni.

Az adatok elszigetelésének elve
Egy program fejlesztése során a programot egymással kapcsolatot tartó struktúrákkal valósítjuk meg. A program futása közben felhasznált adatok között vannak olyanok, amelyek a teljes programra tartoznak, és vannak olyanok, amelyek csak egy részfeladat megoldása közben szükségesek. Azokat az adatokat, amelyeket a program bármely részében fel akarunk használni globális adatoknak kell definiálni és azokat, amelyeket csak egy programmodul belsejében használunk, lokálisaknak kell definiálni. A ciklusokban, megszámlálásokban és egyéb hasonló helyeken igénybe vett változókat munkaváltozóknak hívjuk és a programozás minden szintjén ragaszkodni kell az azonos elnevezésekhez és természetesen mindig lokális változóknak kell őket definiálni.

Nyílt architektúra elve
Egy programozási feladatot igyekezni kell mindig a lehető legáltalánosabban megfogalmazni. Ez a későbbiekben a program módosításának, karbantartásának könnyítését eredményezheti.

A döntés elrejtésének elve
Mindig csak olyan finomítást hozzunk, amelynek a kihatása minél kisebb területre korlátozódik, lokális. Ha változtatnom kell valamit a programomban, akkor annak minél kisebb kihatása legyen.

2.7.2 Taktikai elvek

A technológiai elveket az algoritmusok írásánál és a kódolásnál is jól használhatjuk.

Bekezdéses struktúrák használata
Akár algoritmust írunk, akár kódolunk az elkészülő programszöveg olvashatósága elsőrendű feltétel. A ciklusok, elágazások, eljárások magját mindig célszerű egy tabulátorral beljebb írni és a struktúra azonos szintjén levőket azonos oszlopban elkezdeni.

Barátságos programok írása
A programoknak a felhasználó felé olyan képet kell mutatniuk, amely a felhasználó kegyeit keresi. A programnak magáról mindig el kell mondania azt a szükséges információt, ami elegendő a kezdő felhasználónak is a program használatához.

A képernyőn keresztüli adatbevitelnél megfelelő tájékoztató szövegnek kell megjelennie, továbbá a képernyőre kiírt adatoknak is a megfelelő kontextusban kell megjelennie.

Megjegyzések használata
A programszövegek – algoritmusok – írása közben szükséges a szövegbe olyan részel beírása, amely a nehezebben követhető programrészek működését szavakkal is megmagyarázza. Ez elengedhetetlen, mivel általában a programok fejlesztői sem emlékeznek a program minden részére megfelelően pár hónappal a fejlesztés után. A komment számukra is megkönnyíti a javításokat, illetve azok számára is, akik korábban nem foglalkoztak az adott feladattal.

Menütechnika használata
Azt hiszem ma már minden kisebb feladatot végrehajtó programnak is lehet menüje vagy menürendszere. A felhasználó dolgát megkönnyítjük vele.

Bolondbiztosság
Minden jól megírt program az adatok bevitelekor leellenőrzi, hogy a bevitt adat megfelel-e a programban előírt értéktartományoknak, és ha nem, akkor megfelelő üzenet után újra bekéri az adatokat. A jó program oly módon kezeli a programokat, hogy hibás adatbevitel esetén se száll el. Azt szokták mondani, hogy ha egy 6-8 éves gyerek egy programot nem tud hibás leállásra bírni, akkor tekinthető a program bolond-biztosnak.

3 A moduláris programozás előnyei

A moduláris programozással írt programok előnyei nyilvánvalóak

- Részprogramok könnyen áttekinthetők
- Könnyebben megírható
- Könnyebben tesztelhető
- Több modul írható egy időben (párhuzamos problémamegoldás)
- Könnyebben javítható
- A modulok szabványosíthatók
- Modulkönyvtárakban tárolhatók
- Újrafelhasználhatók

A moduláris programozás felhasználásával a programozó jól működő programokat írhat – de ezt semmi sem garantálja.

4 Struktúrált programozás

A fenti elvek alkalmazásával már használható programokat lehetett írni, de nem lehetett bizonyíthatóan helyes programokhoz jutni. A 70-es évek elején párhuzamosan, de egymástól függetlenül több programozási irányzat alakult ki, amelyek végülis egy irányba a struktúrált programozás kialakulásához vezettek

4.1 Dijkstra: Hierarchikus programozás

A strukturált módszertan legabsztraktabb változatát Dijkstráék (holland) dolgozták ki. Hierarchikus programozásnak hívták ezt a módszert.

A strukturált programozás átveszi a moduláris programozás top-down módszerét. A megoldandó feladathoz a program egy absztrakt programsorozat határértékeként alakul ki. Ebben a sorozatban egy későbbi absztrakt program egy őt megelőző program egy változtatásával áll elő úgy, hogy tekintjük valamely tevékenységét, és fokozatosan finomítjuk. A megelőző program valamely tevékenységét kifejtjük (részfeladat). A sorozat minden absztrakt programja mellett ott van egy absztrakt számítógép, amelynek utasításkészlete megegyezik a programban használt utasításokkal. Végül egy konkrét gép konkrét utasításkészletére készül el a program.

4.2 Mills: Funkcionális programozás

Ugyanaz az elve, mint a Dijkstra féle programozásnak, csak itt az eljárásmodell az elsődleges, az adatmodell másodlagos. Tevékenység mellett osztja szét a problémát részproblémákra. A feladat határozza meg a program szerkezetét.

4.3 Wirth: A Programok részekre való bontásának elvei

Niklaus Wirth a programozás modulokra való bontásánál használt elveket fogalmazott meg (2.7.1 fejezet)

4.4 Jackson és Warnier: Adatorientált programozási módszertan

A top-down módszert vallja, de az adatmodell az elsődleges. Ha van egy probléma, fel kell derítenem az általa érintett adatok szerkezetét. A program szerkezetét az adatok szerkezete határozza meg.
Hátránya: csak adatfeldolgozási területen alkalmazható.

4.5 Boehm és Jacopini

'66-ban publikálnak egy cikket, amelyben leírják, hogy minden algoritmus felépíthető a következõ három vezérlési szerkezet segítségével:

- Szekvencia (Soros program)
- Szelekció (Elágazást tartalmazó program)
- Iteráció (ciklust tartalmazó program)

1968-ban Mills '68-ban bebizonyította a fenti állítást és az alábbiakkal egészítette ki:

1. Egy program akkor jó, ha a szerkezete leírható egy szekvenciaként, amely szekvencián belül a fenti három vezérlési szerkezet megengedett.
2. Egy szekvenciaelembe a külvilágból egy ponton lehet belépni és egy ponton kilépni.
3. Ha egy program ilyen, akkor az strukturált. Kellemesen dokumentálható, módosítható.

A Jackson féle elv azon alapul, hogy az adatszerkezetek szintén leírhatók e három vezérlési szerkezet segítségével, ugyanis egy állomány nem más, mint rekordok iterációja, egy fix rekord pedig mezõk szekvenciája. Az adatszerkezetek meghatározzák a program szerkezetét.
A fenti elvek nem engedik meg a goto utasítás használatát, ugyanis a 2. pontot sértik. Ennek ellenére az utasítás a legtöbb programozási nyelvben megengedett maradt, bár használatát csak speciális esetben engedélyezik és nem ajánlják.

A **strukturált programozás** módszertana a '70-es évek elején analízis módszertanná, rendszerfejlesztési, tervezési technológiává válik. A 1970 - 1980-as évek uralkodó módszertana. A struktúrált programozás a modulok és az adatstruktúrák összefüggését tartja fontosnak a program tervezése során.

4.6 Objektum-orientált programozás - OOP

A hetvenes években a programok bonyolultsága tovább nőtt és a grafikus megjelenítők használatával előtérbe kerültek új problémák. Az alábbi rövid felsorolással az OOP elterjedésének állomásait szeretnénk felvázolni:

- 1969 Alen Key dilplomamunkája során egeret használt, ikonokat, ablakokat és menürendszert
- 1972 konkrét elképzelések OOP és SMALTALK programozási nyelvről
- 1981 Eiffel + eljárásorientált nyelvek egy részében OOP eszközrendszert vezettek be
- 1980-as évek végén a C programozási nyelv bővítéseként létrejött a C++, majd a Pascalt is kibővítették OOP elemekkel.
- Az 1990-es évek közepén megszületett a Java, teljesen OOP nyelv.
- 1990-es évek vége. Majdnem minden modern programozási nyelvnek megszületik az Objektum orientált kiterjesztése.

Az OOP nyelvek nem matematikai elveken, inkább a bonyolultság növekedésének gyakorlati kezelése céljából jöttek létre.

Az OOP alapja
Az adat és a funkcionális modell nem elválasztható. Az adatok és a rajtuk végrehajtható műveletek elválaszthatatlanok.

Az OOP az alábbi fogalmakat használja

Objektum (A változó általánosítása)

- Van **állapota** (attribútum), az állapotot tetszőlegesen komplex adatok segítségével írjuk le (adatelemek + szerkezet).
- Van viselkedése ez **módszereknek** (metódus) hívjuk (függvények és eljárások írják le.)
- Az objektumok állapotának lekérdezésére (mi az értéke)
- Egyik állapotból másik állapotba vivő módszerek (értékadás)
- Összehasonlítás Egy objektum csak önmagával azonos és minden mástól különböző.
    - Azonos állapotban vannak-e
    - Ugyanarról az objektumról van-e szó

Osztály (az adatttípus fogalmának általánosítása)

- Az azonos attribútumú és metódusú objektumok együttese az osztály
- Az objektumok az osztály példányai. Amikor egy osztály egy konkrét példányát létrehozzukj a programban, ezt hívják példányosításnak.

Bezárás (A hatáskör fogalmának általánosítása)

- olyan eszközrendszer, mely segítségével megmondhatom, hogy az osztály attribútumaiból és metódusaiból kívülről mi látszik. (igazi OOP-nél attribútumok általában nem látszanak, a metódusok közül a publikusak, de csak a specifikációs részük)
- Absztrakció: mely során a specifikáció és az implementáció elválik, de ezt szabadon előírhatom.

Öröklés (Az újrafelhasználhatóság csúcsa)

- **szuper osztály** elsődleges a kapcsolatban, alosztály csak hozzá kapcsolódhat.
- **Alosztály**
    - Örökli a szuperosztály attribútumait és metódusait
    - Új attribútumokat és metódusokat definiálhatunk ezen kívül
    - Elhagyhatunk
    - Átnevezhetünk attribútumokat és metódusokat
    - Megváltoztathatjuk a láthatósági viszonyokat
    - Felülbírálhatja a metódusok implementációit.

Lehetőségek:

- Egy alosztálynak egy szuperosztály lehet (1-szeres öröklődést támogató nyelvek) A hierarchia egy Fával írható le.
- Egy alosztálynak több szuperosztálya lehet (többszörös öröklődést támogató nyelvek) A hierarchia gráffal írható le.

Polimorfizmus (többalakúság)

- **Példány polimorfizmus** (egy konkrét háromszög példánya a háromszög osztálynak is és a poligon osztálynak is.)
- **Metódus polimorfizmus**: Az alosztály újraimplementálhat egy metódust kérdés melyik kód fut le. Ezt a kötés mondja meg.

Kötés

- Korai kötés: fordításkor eldől, hogy a meghívott módszerhez melyik kód tartozik.
- Késői kötés: futás közben dől el, hogy a módszerspecifikációhoz melyik kód tartozik.
- Minden példány tudja melyik osztály közvetlen példánya (aktuális példány)
- Minden példány egy jól definiált példány aktuális példánya
- (előny, pl. Ha egy módszer (A) hív egy másikat (B) a másikat újraimplementálom és meghívom (A)-t. ez esetben az újraimplementált fut le az első esetbe az eredeti)

Üzenet (az objektumok közötti kapcsolat)

- Osztályokat definiálok
- Elhelyezem az öröklődési hierarchiákat
- Osztályokon belől objektumokat származtatok
- A program futása közben az objektumok működnek, hatnak egymásra, üzeneteket küldenek egymásnak, válaszolnak az üzenetekre.
- Az üzenet egy metódus meghívása (eljárás vagy függvényhívás)

Objetumorientált nyelvek lehetnek

Tiszta

- Csak az objektumorientált eszközrendszer van
- Van egy standard osztályhiearchia. A programozó mindig ezt a hiearchiát bővíti, ezekhez fűz objektumokat...
- Minden eszköz objektum (az osztály, a módszer és az attributum is)
- Smalltalk, Eiffel

Hibrid

- Eljárásorientált nyelvek objektumorientált eszközrendszerrel kiegészítve.
- C++

5 Nagyobb rendszerek fejlesztésének lépései

5.1 Egy nagyobb rendszer fejlesztésének megkezdése, előkészítése

Nagyobb, több hónapos vagy éves munkát is igénylő rendszerek tervezésénél figyelni kell sok olyan szempontra is, amely nem merül fel a rövidebb programok fejlesztésekor. Először is a „nagy rendszer” fejlesztője leggyakrabban nem saját szórakozásából áll neki a rendszer kifejlesztésének.

A rendszer fejlesztéséhez kell egy Megrendelő és a Fejlesztő. A Megrendelő egy bizonyos célt el szeretne érni a fejlesztendő szoftverrel, általában a munkáját szeretné könnyebbé tenni, amire esetleg még pénzt is hajlandó áldozni. Itt van az első buktató.

Sajnos a szoftverek felhasználói általában hozzá vannak szokva, hogy az általuk használt programok szinte ingyen állnak rendelkezésükre, hiszen szerte a világon az illegális szoftverhasználat elterjedt jelenség. Az illegális szoftver ingyen van. Azt is tudják, hogy egy nagyobb általános célú programcsomag, pl. egy Office bizonyos esetekben a CD áránál nem kerül sokkal többe, így azt hiszik, hogy egy számukra egyszerűnek tűnő programrendszer kifejlesztése is kb. ugyanaz a nagyságrend. Nem akarunk most itt gazdasági számításokba bocsátkozni, de egy multinacionális szoftverfejlesztő cég azért adja olyan olcsón a termékét, mivel több millió példányt ad el. Attól annak a fejlesztése több százezer mérnökórába került!

A Fejlesztőtől az előzetes puhatolózások során árat kérnek. A Fejlesztő ekkor még nem tudja megbecsülni a munka mennyiségét, ezért érdemben nem is tud nyilatkozni, nem is szabad nyilatkoznia. A munkájának értéke a tervezési folyamat során válik mind a két fél számára világossá. A fejlesztési folyamat értékét több féle módon lehet megközelíteni.

1. A fejlesztéshez szükséges idő alapján. Ekkor a fejlesztésre fordított munkaidőt beszorozzuk egy egységnyi óradíjjal és így kijön a fejlesztés értéke. Ez a díj a Megrendelőknek általában sok, így nem jöhet létre a kapcsolat. A megrendelés értékének felső becslésére megfelel.
2. A fejlesztés során létrejövő megtakarítás alapján. Ez konkrét megrendelés és konkrét fejlesztés esetén a megrendelés előtt nem határozható meg egzaktul. A becslés során bármelyik fél rosszul járhat. Nem célszerű használni ezt a módszert, csak akkor, ha konkrét számokkal kimutatható az eredmény.
3. Megéri-e a fejlesztőnek. A fejlesztő megbecsüli a fejlesztési erőforrásokat és megvizsgálja, hogy mennyiért éri meg neki a fejlesztés. Alapul azt veheti, hogy ha nem fejlesztene, akkor az alatt az idő alatt mekkora értéket tudna termelni. Ez a legbizonytalanabb módszer és gyakorlatilag alku kérdése ezen az alapon értéket mondani.
4. Talán a legjobb módszer, hogy a fejlesztés teljes volumenére a fejlesztés elején nem mondunk semmilyen végleges árat, hanem a fejlesztés adott lépéseihez kötünk értéket, így a megoldás első teljes, dokumentált terve egy sarokpont. Annak az elfogadására adunk árat. Ekkor már amúgy is eléggé körvonalazódnak a feladatok. Az első tesztváltozat átadására, a javított változat átadására majd a végleges változat átadására ütemezzük a többi értékelést. Minden egyes nagy lépést dokumentálni kell.

Szakértő

A fejlesztés során a Megrendelő részéről is sok munkát kell befektetni a kívánt cél eléréséhez. Az első kívánalom a Megrendelővel szemben, hogy jelöljön ki egy személyt (vagy többet), akik a fejlesztés megvalósítása során a Megrendelő részéről Szakértőként működhetnek, azaz rálátásuk van a megoldandó feladatra, a Fejlesztő feladatra vonatkozó szakmai kérdéseire válaszolni tudnak, és a fejlesztés során elkészülő részeredményeket tesztelni is tudják (képesek rá). Elvárható a Szakértőtől, hogy számítástechnikai kérdésekben a megfelelő szinten otthon legyen, hiszen sokszor olyan technikai jellegű kérdéseket is el kell tudnia bírálni, amelyeknek a konkrét feladathoz nem vagy csak áttételesen van köze.

Alapkövetelmény, hogy a Fejlesztő technikai kérdésekben csak a Szakértővel konzultál. A megrendelő oldalán több személynek is lehetnek ötletei. Ezek az ötletek a tervezés és a project előrehaladása során sokszor már megvalósult részekkel kerülnek ellentétbe. A Fejlesztő ezekre az ötletekre csak akkor reagálhat, ha a Megrendelő részéről a Szakértő már eldöntötte, hogy az ötlet beleillik-e az addig megvalósult elképzelésekbe vagy akadályozza a megvalósulást. A Szakértő a szűrő, ami számára néha meglehetősen kellemetlen helyzetet teremthet.

A Fejlesztő is a Szakértőnek adja át az elkészült részeket és a Szakértő dolga, hogy az elkészült részeket átvegye és tesztelje. A folyamat végén a szakértő lesz az a személy (lesznek azok a személyek), aki a rendszert érti és bizonyos hibák esetén javítani is tud.

Bizalmas adatkezelés
Az előkészítő megbeszélések alatt és később is a Fejlesztő bizalmas adatok, tesztadatok birtokába juthat. Azokat teljesen megbízhatóan, bizalmasan kell kezelnie.

Az első megbeszélések
A Fejlesztőnek a munka elején több alapos megbeszélést is kell folytatnia a Megrendelővel, annak tisztázás végett, hogy mi is a pontos feladat. Az első megbeszélések általában csak tapogatózó jellegűek, amikor is a felek tisztázzák a Megrendelés feltételeit, és körvonalazzák a feladatot.

Gyakori eset, hogy a Megrendelő és a Fejlesztő nem érti egymást, mivel az egyik félnek nincs meg az a fajta tudása, ami másiknak. Az első megbeszélések idején le kell tisztázni azokat a fogalmakat, amikről a fejlesztés során szó lesz, mindenkinek le kell írni minden olyan információt, ami a közös munkát segíti. Gyakran a feladat véglegesen csak a második, vagy még későbbi konzultációkon tisztázódik.

Ügyvitelszervezés
Egy ügyviteli probléma számítógépes megvalósítása gyakran a Megrendelő ügyviteli rendszerének az újraszervezését is maga után vonja. Ha ennek a szükségessége felmerül, akkor a fejlesztőnek ragaszkodnia is kell az átszervezéshez.

Az első terv
Ekkor születik meg az első terv a feladat megoldására. Ez a terv a Megrendelő számára minél részletesebb, elsősorban vizuális információkat tartalmazó terv. Abból kell kiindulni, hogy a Megrendelő alapvetően a saját problémáit látja, a saját elképzeléseit érti. Az első terveknek olyanoknak kell lenniük, amit a megrendelő megért, azaz azt a nyelvet kell használni, amit ő is ért. Természetesen a fejlesztőnek a saját nyelvezetét és fogalomrendszerét is a terv mögé kell tennie, sőt azt használnia is kell.

5.2 A rendszer tervezése

5.2.1 A programspecifikáció

Bemenő adatok – input
A tervezés során meg kell határozni a rendszer bemenő adatait. Meg kell állapítani, hogy milyen bizonylatok, milyen adatai szolgáltatnak adatokat a fejlesztendő rendszer számára.

Meg kell állapítani, hogy ezek az adatok nem tartalmaznak-e ellentmondást, arra figyelmeztetni kell a megrendelőt.

Meg kell állapítani azt is, hogy nincsen – e többszörös adatbevitel a rendszerbe. Ezt, ha csak lehet, el kell kerülni.

Meg kell állapítani, hogy a bevitt adatok mindig egzaktak maradnak-e. Gyakori, hogy az adatokat a kézi nyilvántartásokban szépítik vagy becslik. Az ilyen adatközlés nem eredményezhet jó végeredményeket sem. Az adatok jósága érdekében célszerű szem előtt tartani a következőket:

1. Célszerű az adatokat bevitelkor valamilyen egységes formátumra konvertálni, különös tekintettel a keresésekben, szűrésekben és indexekben szereplő adatokra. Az ilyen adatokat célszerű Nagybetűs formában tárolni és a bevitelnél eleve így konvertálni.
2. Hacsak lehetséges a program az adatokat előre megadott listákból várja.
3. Célszerű meghatározni a kötelezően kitöltendő adatok körét és a programozás során a megfelelő kóddal kell biztosítani a mindenkori kötelező bevitelt.

Törzsadatok
A törzsadatok azok az adatok, amelyek a rendszer használata során nem, vagy alig változnak. Ennek megfelelően a törzsadatok bevitele a többi adatok bevitelétől eltérő módú lehet. Adott esetben célszerű meglévő adatállományokat felhasználni erre a célra, vagy külön segédprogramokat szerkeszteni erre a célra. A törzsadatok bevitele is a munka értékének egyik paramétere. A törzsadatokat nem célszerű fixen beépíteni a végleges rendszerbe, mert azok az idők folyamán változhatnak.

Kimenő adatok – Output
A kimenő adatok általában képernyők, képernyős- és nyomtatott listák formájában jelennek meg. A két féle listának lehetőleg azonos, vagy hasonló formában kell megjelennie. A listák formájukban áttekinthetőknek és logikusaknak kell lenni.

5.2.2 Képernyőtervek

Nem hangsúlyozható eléggé, hogy a majdani felhasználó mennyire másképpen gondolkodik, mint a fejlesztő. Ennek megfelelően a legegyszerűbb formában a felhasználó a képernyőterveket érti meg. Számára a program tervezete elsősorban a képernyőterveket jelenti. Ha azon a feliratok megfelelően vannak elhelyezve, az adatok pedig a feliratoknak megfelelően jelennek meg, az a felhasználó számára maga a tökély. Éppen ezért a listákon nem szabad technikai jellegű feliratoknak, utalásoknak megjelenni, hanem a felhasználó fogalmait kell használni.

Célszerűen a menürendszereket is úgy kell megszerkeszteni, hogy azok a felhasználó fogalmainak megfeleljenek. Nem szabad olyan fogalmakat erőltetni, amelyek a felhasználók számára zavart okozhatnak, de a felhasználót nem szabad tudatlannak tekinteni. Azokat a fogalmakat, amelyeket a mindennapi életben használ, ha azok esetleg számítógépes fogalmak is, tudottnak lehet tekinteni.

5.2.3 Adatszerkezetek tervezése

A fentiek alapján már elkészíthető a rendszer adatszerkezete. Az adatszerkezetekre általános szabály nincsen. Az adatszerkezet az adatok tárolására szolgáló keret. Az adatszerkezet tervezésekor ajánlatos a következő elvekre figyelni.

1. **Egyszeres adatbevitel**, ne kelljen bevinni ugyanazt az adatot több helyen vagy több féle módon. **Egyszeres adattárolás**, ne legyen egy adat több helyen tárolva, és
2. A törzsadatokat úgy kell tervezni, hogy lekérdezésekben, listákban jól megjeleníthetőek legyenek.
3. Ha csak lehet, célszerű **kódokat használni szövegek** helyett. A kódok szerkezetét rögzíteni kell, esetleg formai előírásokat is be kell vezetni, hogy a kódok mindig megfelelő formátumúak legyenek. Inkább szöveg és szám keverékét használjuk, mint csak számot.
4. Egy adott mező méretének meghatározásakor arra kell figyelni, hogy szövegek esetén a mező mérete elegendő legyen a tipikus adatok tárolására – a kivételesen extra méretű adatokat lehet rövidíteni. Figyelni kell arra is, hogy bizonyos utasítások, adatbázis-kezelő parancsok a teljes adattartalmat átolvassák, hogy a megfelelő eredményt szolgáltassák. A **túl nagy mezők** az adatelérést lassítják esetenként jelentősen.
5. A **numerikus mezők** mérete az előrelátható legnagyobb értéket tudja fogadni, ezen belül numerikus eredménymezők esetén figyelni kell a nagyságrendek ugrására, továbbá a megfelelő mennyiségű tizedes helyre.
6. **Megjegyzés** típusú mezőket ne használjunk, ha nem muszáj. A megjegyzés majdnem minden nyelven feleslegesen sok helyet foglal el, sőt sokszor a megjegyzés adatbázis csak nő. Megjegyzésben keresni, indexelni, szűrni körülményes.
7. **Logikai mezőket** lehet használni, de figyelni kell arra, hogy mi is lesz a mező értelme.
8. Ha több adatfájl – táblázat - ugyanannak a mezőnek az adatait tartalmazza, akkor megfelel mutatókkal, **relációval** lehet az adathoz hozzáférni. A relációk lassítják az adatelérést, ezért csak a szükséges relációkat kell alkalmazni

5.2.4 Összefüggések az adatok között

Gyakori, hogy a különböző adatok összevetéséből számított új adatok jelennek meg a programban. Külön kérdés, hogy az így létrejövő új adatokkal mi legyen, töröljük őket, vagy a programra bízzuk az esetleges újraszámolást.

Olyan számított adatok esetén, ahol az eredményt időrendben egy adott állapotban rögzíteni kell, mint például importáru esetén az árfolyamot, a számított adatokat rögzíteni kell, és naplózni kell azokat a körülményeket – itt például a valutaárfolyamot – ahogyan a számított adat létrejött. Ha a kiindulási adatok és a kiszámítás módja nem változik, akkor a számított érték mindig újra előállítható, így az értéket nem kell rögzíteni, elegendő csak a kiindulási értékeket tárolni.

5.2.5 Felhasználói felület – user interface

Egy program lehet kiváló képességű, a feladatokat kiválóan oldhatja meg, de ha a felhasználói felület nem megfelelő, akkor a program használatától elmegy a potenciális felhasználók kedve. Ez indokolja azt, hogy a program tervezésekor különös figyelmet fordítsunk a felhasználói felületre.

Általában egy program indítására a program egy adott viselkedéssel válaszol. Ez a program alapbeállítása vagy idegen szóval **default** beállítása. Ehhez a fejlesztőnek meg kell határozni azokat az alaptulajdonságokat, amelyekkel a program rendelkezik – egyéb rendelkezés hiányában. Az alap-viselkedést meg lehet változtatni, mégpedig a következő módokon:

1. paraméteres indítás

Ekkor a program a parancssorban megadott paraméterek hatására a default viselkedéshez képest más tulajdonságokkal kezd működni. A paraméterek parancssori használata a nagygépes operációs rendszerektől ered. A PC-s világban a UNIX operációs rendszeren át a DOS-os programok is átvették a paraméterek használatát. Az ilyen programok használatának megtanulása felesleges terhet ró a felhasználóra, abszolút barátságtalan a kezelőfelület.

A hozzáértő felhasználó viszont így tudja a programok **legmegfelelőbb beállításait** használni és így tudja más programok által meghívva a kívánt viselkedésre bírni a legegyszerűbben. Ez a fajta felhasználói felület még a windowsos programok korában sem ment ki a divatból. Ennek a felhasználói felületnek szokás szerint az egyik alapparamétere a /?, ?, vagy a /h. Ezek a paraméterek szokás szerint a programok használatáról nyújtanak segítséget. Ezektől eltekintve az egyik legbarátságtalanabb felületet mutatják a felhasználó felé.

A paraméteres indítás esetén **hátrány** lehet a nem megfelelő paraméterek feldolgozása. Mi történjen nem megfelelő paraméterek használata esetén. Erre általános szabály nem létezik. Olyan választ kell adni, amely szöveges formában tájékoztatja a felhasználót a hibáról és annak megoldási módjáról. Az ilyen hibaüzenet nem jó: „Runtime Error, errorcode 1201 in line 1324”. Ha a programozó ezt a módszert használja, akkor vigyáznia kell, hogy:

- Csak azokat a paramétereket fogadja el a program, amelyek a program viselkedésén változtatnak;
- A paraméterek írásmódja tetszőleges legyen, azaz kisbetű – nagybetű nem számíthat;
- A DOS-ban a paraméterek előtt szokásosan a „-„ vagy a „/” jel áll, lehetőség szerint tartani kell ezt a konvenciót;
- A DOS-ban bizonyos jelek nem megengedettek a paraméterekben – ezeket nem szabad használni;
- A paraméterként adott karakterek, szavaknak utalnia kell a megadott funkcióra, anyanyelven vagy angolul, ráadásul a nyelv kérdésében következetesen az anyanyelvet vagy az angolt kell választani, keverni nem lehet őket.
- Célszerű fenntartani egy paramétert egy rövidebb vagy hosszabb HELP kiíratására. A HELP-nek akkor is célszerű megjelennie, ha hibás paraméterezést adtunk a programnak;
- Ha megoldható, akkor célszerű a paramétereket tetszőleges sorrendben bekérni és célszerű a paraméterek hiányában valamilyen default viselkedést előírni. Ez a akár a párbeszédes adatbekérés is lehet.

2. Párbeszédes felület

A program az indítás után megkérdezi a beviendő paramétereket, majd az eredményt kiírja képernyőre, általában a bevitt adatok alá.

Ez a fajta párbeszédes, üzengetős interface a nagygépes világból átöröklött, és a UNIX, DOS világában ma is meglévő kezelői mód. Ezek a programok egyáltalán nem tekinthetők „barátságosnak”, a felhasználónak könnyű rossz válaszokat adni a feltett kérdésre. Csak nehezen találhatók meg a programban adott lehetőségek. Ha adatbevitelkor hibás típusú adatot viszünk be, előfordulhat, hogy a program futás közbeni hibával elszáll „Runtime Error …” és a felhasználó csak találgathat, hogy mi volt a hiba oka.

Kezdő programozók által gyakran használt módszer. Tanuláskor a legegyszerűbben így lehet adatbevitelt létrehozni a program részére. Ha a programozó mégis ezt a felületet választja, akkor vigyáznia kell:

- A adatbevitel során a bevitt adatok típusát ellenőrizze a program és a bevitelkor figyelmeztessen a típus-, vagy nagyságrendi hibákra
- Hibás adatbevitel esetén kínálja fel az újrapróbálkozás vagy a kilépés lehetőségét
- A beviendő adatok előtt mindig írja ki pontosan, hogy mit vár a felhasználótól, és egyértelműen jelölje meg az adatbevitel helyét is a kurzorral

- Ha az adatbevitel során eldöntendő kérdést vagy kérdéseket teszünk fel, akkor mindig ugyanazokat a karaktereket válasszuk a válaszadásra – lehetőleg a válasz Igen/Nem anyanyelvi esetben I vagy N, angol Yes/No esetben pedig Y/N legyen – kisbetű, nagybetű ne számítson
- Az eldöntendő kérdés mindig pontosan legyen megfogalmazva és legyen kiírva a lehetséges válasz is;
- Ne fogadjon el más karaktereket az eldöntésnél
- A bevitt adatok után a program valamilyen választ adhat, ez jól elkülönülve jelenjen meg a képernyőn, lehetőleg a következő sorok egyikében, a sor elején kezdve
- Nem célszerű színeket használni, csak ha valamiért ki kell emelni egy sort. A színek a monokróm képernyőn esetleg nem látszanak elég jól.

A fenti két módszert nagyon gyakran együtt alkalmazzák DOS-os környezetben. Ha a paraméterek nincsenek megadva a parancssorban, akkor kérdéseket tesznek fel és így biztosítják a hiányzó adatokat.

3. Menüs Interface

A programok menüs kezelése mára elnyerte létjogosultságát. Mind alfanumerikus, mind grafikus környezetben a mai programok többsége elképzelhetetlen menük nélkül, ennek ellenére vizsgáljuk meg, hogy mikor nem szükséges menüket használni.

Alapvetően egy menüválasztást jelent több lehetőség közül, azaz ha nincsen választás, akkor menüre sincsen szükség.

Azokban a programokban, amelyeket egy adott célra fejlesztettek ki, felesleges menüket alkalmazni, hiszen a program elindítása eleve a kért funkciót is elindítja.

Ha egy programot egy másik programból indítunk el – pl. keretprogramból egy tömörítőt-, akkor a keretprogram hordozza a választásokhoz szükséges információt, tehát ekkor sem kell menü.

Minden egyéb esetben van a menüknek létjogosultsága.

Hogyan jelenjen meg egy menü?

A kezdő programozók legegyszerűbb menüje a következő:

Egymás alá felsoroljuk a menüpontokat, mindegyik elé egy sorszámot írunk, majd feltesszük a kérdést, hogy melyiket választja. A választástól függően egy CASE szerkezettel kiválasztjuk a megfelelő menüpontot, majd a megfelelő eljárást meghívjuk. Az eljárásból visszatérve a menüt újra kirajzoljuk és kezdődik minden elölről.

Ennek egy kicsit módosított változata, ha nem sorszámot adunk a menüpontnak, hanem karakter, illetve ha a menüpont egyik betűje kiemelt, akkor azt a billentyűt megnyomva indul a kérdéses funkció.

Ennél egy kicsit bonyolultabb megoldás, amikor a menü nem csak egy egyszerű felsorolás, hanem a menüpontok között a megfelelő irányú kurzormozgató billentyűkkel is lehet mozogni. Ekkor gondoskodni kell arról, hogy az aktuálisan kijelölt menüpontnak más legyen a színe, mint a többi menüpontnak, illetve monokróm monitoron legyen intenzívebb a megjelenése. Az ilyen menükben a nyilakon kívül célszerűen a Home és az End a menü első illetve utolsó pontjára ugrik, az ESC jelentése kilépés a menüből, az Enter elindítja az aktuális menüpontot.

Amikor egy menü megjelenik a képernyő közepén esetleg letakarva a mögötte lévő képernyőterületet, akkor azt előugró vagy angolul POP-UP menünek hívjuk. Ha egy menüpont kiválasztásával egy másik menü nyílik meg, akkor azt almenünek vagy SUB menünek hívják. Az így létrejött menüt és almenükből álló rendszert menürendszernek hívjuk és ennek a menürendszernek a kezdete a főmenü. Egy menürendszer mindig ábrázolható egy fa struktúrával is, ezért leírásokban a menürendszer egy almenüpontját egyértelműen meg lehet adni, ha a főmenütől kezdve felsorolom azokat az almenüpontokat, amelyek elvisznek a megfelelő funkcióhoz.

Minél több féle alternatívát használunk a menürendszerünkben, annál jobban meg kell szervezni a végrehajtó programot – ez trivialitás. Mindenesetre következetesnek kell lenni egy bonyolultabb menürendszer leprogramozásakor.

Ha több programunkban is akarunk használni menüt vagy menürendszert, akkor célszerű egyszer megírni egy teljesen általános menürendszert, amit a megfelelő paraméterezéssel később is tudunk használni.

Borland Pascalban, Borland C-ben, az elterjedt adatbázis-kezelőkben vannak ilyen könyvtárak, amelyek menüket jelenítenek meg, azokat bármikor nyugodtan fel lehet használni.

A bonyolultabb menürendszerű programokban a felhasználó elveszhet a sok menü között, ezért felmerült az igény, hogy a menüpontokhoz magyarázatot is kell fűzni. DOS-os környezetben az aktuálisan kiválasztott menüponthoz tartozó magyarázat általában a képernyő alsó sorában jelenik meg, míg Windowsos környezetben manapság alkalmaznak kis keretben megjelenő leírást, amikor az egérrel megállunk egy menüponton. Ezt angolul **tooltip-nek** hívják.

Külön feladatot jelent menükezelésnél az egér használata. Az egér mára általánosan elfogadott kezelőeszköz, de DOS-os környezetben a rendszer nem dolgozza fel az egér pozícióját, azt a saját programmal külön le kell programozni. Az egér kezelése végső soron azon alapul, hogy a megfelelő megszakítás meghívásával le lehet kérdezni az egér pozícióját és az egérgombok állását. A menürendszerben mindig kell lennie egy várakozó résznek, ahol billentyűlenyomásra várunk. Célszerűen ezen a helyen lehet beiktatni az egér események figyelését is, és ha az egér valamelyik billentyűjét lenyomjuk, akkor utána le kell kérni a pozícióját, meg kell vizsgálni, hogy menüponton áll-e és ha igen, akkor meg kell határozni a menüpontot. Innen több választás lehetséges. Az egyszerűbb esetben a program betölti a billentyűzet pufferbe a megfelelő billentyűt és hagyja a várakozó ciklust tovább haladni. A következő körben a ciklus észleli a karakter megérkezését és feldolgozza azt. Bonyolultabb esetben az egérkoordináták alapján a menüponthoz tartozó eljárást rögtön elindítja, azaz a billentyűzet és az egér menükezelése párhuzamos lesz.

A Windows menükezelése példamutató, de ott az operációs rendszer a felelős a billentyűzet és az egér események feldolgozásért. A Windowsban minden menüpontnak van egy belső sorszáma és a várakozó rendszer, akár billentyűleütésről van szó, akár egéreseményről, a menü belső sorszámát beteszi egy eseménypufferbe. A program saját várakozó ciklusa azt figyeli, hogy szerepel-e egy neki szóló menüsorszám az esemény-pufferben. Ha igen, akkor aktivizálja magát. Mivel ennek a részletes tárgyalásához az objektumokról, mint adattípusról is kell szólni, ezért a Windows menürendszerének további boncolgatását nem folytatjuk.

4. Gyorsító billentyűk

A régebbi DOS-os programoknál a menürendszer utolsó almenüjén keresztül lehetett egy-egy beállítási funkcióhoz eljutni. Gyakran az almenü szomszédos menüpontjának eléréséhez pedig elölről kellett kezdeni a bóklászást a menürendszerben. Ez megnehezítette a kezelést, ezért a menüket ellátták gyorsbillentyűkkel, amit angolul SHORT-CUT-nak hívnak. Ezek segítségével a menürendszer nélkül lehetett elindítani funkciókat. A gyorsító billentyűk használata csak sok gyakorlás után válik természetessé, hiszen ezeket az ALT-CTRL-SHIFT-Billentyű kombinációkat nehéz megjegyezni. Illetve minél bonyolultabbak a használt programok, annál jobban kell vigyázni, hogy a short-cut-ok mindig következetesen ugyanazt jelentsék. A programozásuk is nehezebb, hiszen a menürendszerben biztosítani kell azt, hogy bármikor a megfelelő short-cut hatására a megfelelő eljárás induljon el. Nem könnyű feladat. A gyorsítóbillentyűk használata azonban megkönnyítheti és meggyorsíthatja a programok használatát.

5. Párbeszédablakok

A menürendszerek használatának könnyítésére a párbeszédablakokat is használunk a felhasználói interface szokásos részeként. A párbeszéd ablakok (angolul DIALOG) olyan a képernyőn keretben megjelenő objektumok, amelyek több kezelőfelületet, gombot, beviteli helyet, kapcsoló, választási lehetőséget tartalmaznak. Az egyes kezelőszervek beállításai csak akkor válnak véglegessé, ha a párbeszédablakon megjelenő elfogadó nyomógombot – Ok, Rendben vagy valami hasonló feliratú – megnyomjuk. Ha a szintén általános Mégsem, Mégse vagy Cancel feliratú gombot aktiváljuk, akkor a beállítások nem véglegesítődnek. A párbeszédablak megjelenő kezelőszervei között a TAB és a SHIFT+TAB billentyűkkel lehet általában mozogni. DOS-os környezetben eléggé körülményes megvalósítani a párbeszédablakokat, de nem lehetetlen. Objektum orientált módszerrel programozva léteznek ilyen könyvtárak a Turbo PASCAL és a Turbo C nyelven is. Minden windowsos programozási nyelv természetesen rendelkezik ilyen lehetőséggel.

5.2.6 Segítségadás a programokban

A felhasználói interface-hez szorosan kapcsolódó, ámde mégis különálló téma a help – segítség kérdése. Kezdetben a segítség a felhasználói kézikönyvet, később az egyszerűbb programok használatakor a segítség a /? vagy hasonló paraméterrel történő indításkor egy vagy néhány oldalnyi legördülő szöveg megjelenését jelentette.

Ma már a segítség több mindent takar és több szintje is lehet. A legegyszerűbb valóban a megjelenő egy oldalnyi információ. Bonyolultabb programok, menürendszerek esetén ez már nem elég. Ilyenkor jelennek meg a helyzethez alkalmazkodó segítő rendszerek. A korábban már említett menüponthoz kapcsolódó egysoros információ lehet a segítő rendszer második lépcsőfoka, de ekkor még mindig fennáll a tudatlan felhasználó alulinformáltságának veszélye. A harmadik lehetőség, amikor a kedves felhasználó megnyomja az F1 billentyűt és egy ablakban megjelenik egy részletes szöveges, képes leírás, amelyben mindenféle, az adott menüponttal kapcsolatos tudnivalót leír a fejlesztő. A DOS-os Norton Guide vagy a Windows help rendszere ennél is tovább megy. Nem elég, hogy megjelenik a szöveg, de a szövegben utalások vannak más szövegrészekre is, amelyekre egyszerű kattintással lehet átlépni. Az ilyen HELP rendszert HYPERTEXT rendszernek hívják. Az ilyen rendszerek komoly programozást igényelhetnek.

5.2.7 A tervezés lépései.

Egy számítógépes rendszer megtervezése általában nem egyszerű feladat. A megvalósítandó rendszer rengeteg összetevőből állhat. Nem biztos, hogy csak egy egyszerű program kifejlesztéséről van szó, hanem gyakran hardverből, megfelelő operációs rendszerből, kifejlesztett szoftverből álló összetett rendszert kell megtervezni és megvalósítani, a kívánt feladatok elvégzésére. Az egyes komponenseket – a hardvert, az operációs rendszert és a szoftvert - sokszor nem is lehet elválasztani egymástól. A fejlesztés tervezése során nagyon fontos a célszoftver kifejlesztéséhez felhasználandó eszköz gondos kiválasztása. A következőkben az egyes komponenseknél figyelembe veendő szempontokat fogjuk megtárgyalni.

5.2.8 A megfelelő hardverháttér megállapítása.

A szükséges hardver függ a futtató operációs rendszertől, a használt fejlesztőeszköztől is.
Manapság a multitaszkos operációs rendszerek világában azt lehet mondani, hogyha egy számítógép egy adott operációs rendszert lassan futtat, akkor a felhasználói programokat is lassan futtatja majd.
Gyakori eset az, hogy a megrendelő a meglévő hardverparkot alkalmasnak tartja a feladat elvégzésére, ugyanakkor a fejlesztő a saját rendszerét használja tesztelésre és a fejlesztőnek kompromisszumokat kell kötni.. Ha a fejlesztő kompromisszumokat köt, nem veszi figyelembe az alkalmazandó operációs rendszer és fejlesztőeszköz kívánalmait, akkor saját magának nehezíti meg a helyzetét. Az elkészült rendszer mélyen a kívánt teljesítmény alatt fog teljesíteni.

Általában figyelembe kell venni a hardver tervezésénél a gép processzorát, memóriaméretet, a szükséges háttértár méreteket. Ezen kívül a hálózat meglétét vagy szükségességét, modem és egyéb szükséges eszközöket is figyelembe kell venni. Szükség esetén elő lehet írni megfelelő grafikus felbontást is.

Ma már gyakorlatilag DOS-os rendszeren futó programokat nem fejlesztenek, és a Windows 3.1 is kiment a divatból.
A Windows 95, Windows98, Millenium Edition Intel PII-es (Celeron) processzoron 32 MB RAM-mal elfogadható teljesítményt nyújtanak.

A Windows NT 4.0 kicsit nagyobb étkű rendszer. Minimálisan 16 MB optimálisan 64 MB RAM a memóriaigénye. A winchesteren több helyet foglal, mint a Windows95/98. Ezért célszerűbb nagyobb winchestert használni hozzá, mint egy megfelelő Windows 95-ös géphez tennénk. Ezek a szoftverek természetesen már a múlthoz tartoznak, bár még sokan használják őket.

A Windows 2000 kissé elavult már, de sokfelé élő rendszerek futnak ilyen operációs rendszeren 64 MB minimum, 256 MB Ram már elegendő, 500 MHZ-es P-II/Celeron.

A Windows XP minimum 128 MB RAM-ot használ és 256 MB elég jó neki, de 512 MB RAM-mal hasít igazán. 1-2 GHZ-es Celeron/P-III/P4 processzor célszerű alá.

A Linux freeware operációs rendszer, ugyanolyan hardveren jobban teljesít mint egy XP.

5.2.9 A megfelelő operációs rendszer

Az előző fejezetben felsorolt operációs rendszerek azok, amelyek a leggyakrabban szóba jöhetnek. Most azt vizsgáljuk meg, hogy melyik rendszer milyen alkalmazások esetén a legalkalmasabb a fejlesztés és a futtatás szempontjait figyelembe véve.

Meglepő, de nagyon sok esetben a DOS alkalmas futtató rendszer! Az okok meglehetősen sokrétűek.

A DOS 25 éve jelent meg! A BIOS és a DOS hívásai abszolút publikáltak, a fejlesztő rendszerek sokadik generációi készítenek DOS-os programokat. Az x86-os processzorok real módját már minden fejlesztő rendszer készítő oda-vissza kiismerte.

A DOS-os programok általában stabilak. A számítógépek gyorsan bebootolnak DOS esetén. A memóriakezelés gyors, mert kevés információt kell változtatni. A DOS-os driverek kis méretüknek köszönhetően általában assemblyben készültek, de a szabványok már olyan jól ismertek, hogy akik ilyen programokat készítenek volt idejük jó drivereket írni. A hálózatkezelése tökéletesen megoldott. Gyakorlatilag minden PC-s környezetben működő hálózatnak van kliensprogramja hozzájuk. Az MS-DOS 5.0 nagy áttörés volt stabilitás szempontjából is. Ettől a verziótól számítva a memóriakezelése is jónak mondható. A DOS 640 KB memóriájából 620-630 is szabaddá tehető egyes esetekben. A memóriaméretnek megfelelő programok gyorsan betöltődnek, gyorsan harcra készen állnak. Ha egy program mégis lefagy, akkor legfeljebb egy-két lezáratlan állomány marad a lemezen. Akár egy floppyról is lehet futtatni egy DOS-os rendszert, még akkor is ha az alkalmazás esetleg overlay technikát használ – RAMDISK használatával. Hálózati környezetben akár a szerverről is futhatnak az alkalmazások.

Fejlesztői szempontból célszerű DOS-os alkalmazást készíteni, mivel ekkor a multitaszkolás, nincsenek az alkalmazás alól más alkalmazások által kihúzott programok. A fejlesztő viszonylag egyszerűen programozhat kevert nyelven. (Ilyen célra leginkább az Assembly és a C nyelvű betétek alkalmasak.) Még az assembly nyelvű programok hibáinak a követése is megoldható viszonylag gyorsan. Az operációs rendszer erőforrásait könnyű elérni. Az operációs rendszer szolgáltatásait megszakításokkal, a környezeti változók alkalmazásával, a keresési útvonalak használatával könnyen felhasználhatjuk.

Hátrányok

A grafikus lehetőségeket igazán elfelejthetjük. Bizonyos fejlesztőrendszerek olyan programokat fordítanak, amelyek memóriaigénye közelít a 640 KB-hoz, azaz krónikus memóriahiányban kezdenek szenvedni az alkalmazások. Ezt SWAP és overlay technikával küszöbölik ki, illetve egyes linker programok alkalmassá teszik az általunk írt kódot a DOS ún. protected üzemmódjában való alkalmazásra. Ekkor a program részére nem csak a 640 KB memória áll rendelkezésre, hanem a gépben lévő összes XMS memóriát fel tudja használni a program.

Generálisan nem megoldott probléma a magyar nyelvű ékezetes üzenetek megfelelő kiírása a képernyőre és a nyomtatóra.

A Windows 3.1/3.11-es shell instabil a Windows kooperatív multitask koncepciója miatt. Ha a rendszerben lévő egyik program kisajátítja magának a processzort, azaz nem adja át a vezérlést a többi programnak, akkor a rendszer menthetetlenül összeomlik. A grafikus erőforrások kezelése sem problémamentes. A tapasztalat azt mutatja, hogy ha a szabad grafikus erőforrások mértéke 50% körülire csökken, akkor a Windows 3.1/3.11 instabillá válik. Ekkor a legalkalmasabb megoldás a rendszer kikapcsolása és újraindítása. Mindezek a problémák a memóriakezelés kompromisszumos voltával függnek össze, továbbá azzal, hogy a rendszer bizonyos célokra nevetségesen ki memóriaterületeket tart fent.

A Windows használata közben sok olyan dolog felmerül, ami a DOS-os programok használóit nem érinti. A multitask miatt a rendszerben elindított programok kárt okozhatnak más futtatott alkalmazás fájljaiban futás közben, esetleg megmagyarázhatatlan hibákat előidézve. A kezelés során az egér használata bizonyos esetekben komoly problémák forrása lehet. A fejlesztők is néha akarva-akaratlanul az egér használatát írják elő még akkor is, ha nem a legcélszerűbb. Mindazonáltal el kell mondani, hogy olyan munkahelyen, ahol a gépet office-szerű programok futtatásra is használják, ott a Windows kezelése nem jelent általában problémát.

A grafikus programok nagyobbak, mint a karakteres felületű programok, ezáltal lassabbak. Ezt tudomásul kell venni.

A nyomtatás és a képernyőképek a Windows-ban egyszerűen tervezhetők. Minden windowsos fejlesztőeszköznek létezik resource-ok létrehozására alkalmas programja, ha pedig nincsen, akkor használható vagy a Microsoft vagy a Borland széles körben elterjedt megfelelő programja. Ezek segítségével a windowsos programok menüje, a párbeszédablakok, a párbeszédablakokban lévő grafikus objektumok mind megtervezhetők és a programba beletehetők vagy DLL fájlként a főprogramhoz mellékelhetők. Ezekkel az eszközökkel a programok külső képe gyorsan létrehozható.

A nyomtatás tekintetében is egyszerű a helyzet. A nyomtatók rendszerszinten definiáltak, azaz a windowsos programok tervezőinek nyomtatáskor a nyomtató fajtájtól függetlenül, a Windows szolgáltatásait felhasználva nyomtathatnak. A modern 4GL rendszerek eleve képesek a kódot a képernyőn megjelenő objektumokhoz igazítani, így forrásszinten áttekinthetőbb programok készíthetők.

A windowsos programok futtatása esetén felmerül egy probléma. A különböző Windows rendszere, még azonos verziószámon belül is – nem kompatibilisek egymással. Bizonyos állományaik nyelvi jellemzőkkel rendelkeznek, amelyeket nem mindig lehet figyelmen kívül hagyni. Azonos nevű könyvtári fájlokat tartalmazhat a Windows és a fejlesztői program is. Az egyes könyvtári fájlok viszont a fejlesztés különböző stádiumaiban vannak, tehát nem ugyanarra képesek. A Windows egy fájl betöltésekor megnézi, hogy a memóriában létezik-e és ha igen, akkor még egyszer nem tölti be. Ez futás közben potenciális és nehezen felderíthető hibaforrás lehet. Célszerű stratégia mindig az azonos nevű és azonos funkciójú fájlok közül a legújabb dátumú vagy legújabb verziószámú használata. Ha a rendszerben eredetileg lévőnél régebbit helyezünk a rendszerbe, akkor az esetlegesen kijavított hibákat visszacsempésszük a rendszerbe és előre nem látható hibák jelentkezhetnek.

Egy programfejlesztő barátom egyszer azt mondta, hogy azért nem szeret Windowsban programozni, mert akkor nem tudja, hogy a háttérben mi történik. Erre én azt feleltem, hogy ha a program jól működik, akkor nem is kell. A probléma csak az, hogy a programok nem mindig a szabvány szerint működnek.

Végső soron akkor érdemes windowsos programot fejleszteni:

- ha memóriaproblémák lennének DOS-ban,
- ha barátságos kezelőfelületet szeretnénk,
- ha a nyomtatási kép grafikát igényel, vagy a programunk eleve grafikus jellegű dolgokat dolgoz fel vagy jelenít meg,
- ha programunknak más programokkal együtt kell működni,
- ha eleve a Windowsban meglévő szolgáltatásokat akarjuk használni,
- ha gyorsan akarunk fejleszteni, de nem érdekel minket a létrejövő alkalmazás mérete,
- ha másban nem lehet

A Windows95/98/Me alapvetően felhasználói és fejlesztői szempontból stabilabb rendszer mint a Win3.x.

A Windows95/98/ME 32 bites operációs rendszer és a kooperatív multitask helyett a preemptitív multitask vette át a szerepet a 32 bites programok esetében, a 16 bites programok esetében ugyanakkor megmaradt az előző rendszer multitaszkolása. Ha 32 bites programot fejlesztünk, akkor alkalmazhatjuk a a multithreading lehetőséget is. Ez azt jelenti, hogy a programunk futás közben elindíthat olyan programszálakat, amelyek a programunktól függetlenül másik taszkban futnak, de ugyanazt az adathalmazt használják. Az ilyen szálakat a programunk azután meg is tudja szüntetni. Ezzel a módszerrel a megszakítások használata mellett sokkal természetesebb módon lehet programozni háttérben futó tevékenységeket.

A Windows 98 augusztusában jelent meg. Lényegében a Windows 95 javított, stabilabb és kicsit módosított változatának tekinthető.

A Windows NT/Windows 2000/Windows XP belső felépítésüknek megfelelően a Windows 9x rendszereknél is stabilabb futtatási környezetet biztosítanak és manapság a legelterjedtebb fejlesztői platformnak számítanak.

OS/2 Warp-ot akkor használ az ember, ha szereti az OS/2 Warp-ot, vagy kénytelen vele dolgozni. Az OS/2 Warp-os programokban megvan minden, ami a Windows-ban, de a támogatása nem olyan széleskörű, mint a Windows-é. Ezzel szemben az OS/2-ben is lehet csodálatos dolgokat készíteni A Win3.1-re írt alkalmazások nagyon szépen futnak az OS2/Warp alatt.

A Linux rohamosan fejlődő környezetet jelent. Linux alatt a természetes fejlesztő nyelv a C illetve a C++.

A fenti operációs rendszerek támogatottsága és kezelhetősége fontos szempont.

A DOS használhatósága korlátozott, de nem bonyolult. A kezelését, esetleges hibakeresést sokan el tudják végezni..

A Windows 3.1/3.11 lejárt lemez.

A Windows 95/98/ME már lassan kimegy a divatból, sok programozó ismeri, használja.

A Windows NT és a Windows 2000 windows XP egyszerűen betonbiztos.

5.2.10 A felhasználható fejlesztőeszközök kiválasztási szempontjai.

Nyilvánvalóan az előzőekkel szorosan összefügg a fejlesztésre használható rendszer, vagy rendszerek kiválasztása. Hogy melyik rendszert válasszuk ki, arra nézve a következő elveket érdemes figyelembe venni.

Nézzük meg, hogy mennyire alacsony szintű – rendszer közeli – a feladat!
Rendszer közeli alkalmazások fejlesztéséhez Assembly, C, C++, esetleg a Pascal nyelv ajánlható. A létrejövő program a fenti sorrendnek megfelelő méretű lesz, a futási sebessége fordított lesz, a fejlesztéshez szükséges idő – egyenlő tudásszinteket feltételezve – Pascal esetén a leggyorsabb, és Assemblyben a leglassabb.

Kisebb segédprogramok fejlesztésekor az Assembly-t érdemes kihagyni, de a C és a Pascal továbbra is jó választás. Ezek általános célú nyelvek, minden elvégezhető velük, amire általában egy programban szükséges lehet.

Összetettebb alkalmazások fejlesztésére célszerű olyan fejlesztőrendszert keresni, amely az adott területre kihegyezett. Ekkor a harmadik generációs programozási nyelvek – C, Pascal – már nem elég hatékonyak.

Vizuális, grafikus programok esetén használhatjuk a Microsoft Visual Stúdió (Basic, C, C#, Java) a Sun féle Java, Borland C-Builder, Delphi, és még sok fejlesztő rendszer programjait.

Nézzük meg a feladat bonyolultságát!
Gyakran előfordul, hogy egyszerűbb feladat megoldására már létezik az adott szoftver. A kisebb adatbázis-kezelő célrendszereknek, és utility programoknak se szeri, se száma a világ szoftvertermésében. Célszerű először tájékozódni, hogy létezik-e az adott feladatra kész program, majd ha nem találunk, akkor álljunk neki a fejlesztésnek.

Egyszerűbb adatbázis-kezelési feladatok megoldására olyan fejlesztőrendszert célszerű választani, amelyben egyszerű sablonok adatainak kitöltésével lehet a megfelelő célszoftvert összeállítani. Ilyenek pl. a Microsoft Access, dBase, a Database Manager. Egyszerűbb alkalmazások összeállítása 2-3 órai munkával már megoldhatók.

Több adattáblát tartalmazó programok fejlesztésekor már több variáció is szóba jöhet, hogy csak a legismertebbeket említsük, Microsoft Access, CA-Visual Object, Delphi, Microsoft Foxpro/Visual Foxpro, Visual Dbase, stb…

A felsorolt rendszerek mindegyike 4GL rendszerű.

Microsoft SQL, Oracle, MySQL, stb...

Nézzük meg a futtatási platformot!
Bár a DOS-os programok egy része már fordítható és futtatható a fejlesztési fázisban Windows alól is, és ezért sokszor alkalmasabb Windows alatt DOS-os programot, mint DOS-os környezetben,de ha DOS-os alkalmazást fejlesztünk, akkor azt Windows alól futtatni nem ajánlatos.

A DOS-os programok súlyos korlátja a DOS memóriakezelése. A nagyobb DOS programok 500-600 kb-ot is lefoglalnának maguknak, ezért esetleg bizonyos memóriakezelési kompromisszumokat is szükséges kötni.

A DOS-ban lehetséges védett módban üzemeltetni 386-os processzorú gépet, ugyanúgy, mint a Windowsban. Ha van rá lehetőség, akkor olyan nyelvet és Linker-t kell választani, amely felkínálja a lehetőséget. A Borland Pascal, a Watcom C, a Clipper az Exospace és/vagy a Blinker linkerekkel tud védett módú programokat készíteni DOS alatt. Ezek a programok már képesek kezelni az Extended (XMS) és az expanded (EMS) memóriát is.

Ha a hardver megengedi, akkor készítsünk Windows9x/Me vagy NT/W2K/XP rendszereken futó programokat. A felhasználói felület sokkal barátságosabb lesz, gyorsabban ki is fejleszthető. A megfelelő hardvert, azonban biztosítani kell, azaz processzorral, memóriával és szabad winchester hellyel nem szabad takarékoskodni.
Célszerű, ha lehetséges kliens-szerver alkalmazásokat írni, mivel ilyenkor a számítási feladatok egy részét a szerver átvállalja a munkaállomásoktól.

5.2.11 A terv dokumentálása

A rendszerfejlesztés megkezdése előtt nagyon fontos feladat az elkészítendő rendszer paramétereinek rögzítése, azaz a terv dokumentálása. Ebben a dokumentumban minden olyan alapvető paramétert rögzíteni kell, amelyek a későbbiekben meghatározzák az elkészítendő rendszert. A terv dokumentációja védi a megrendelőt és a fejlesztőt is. A megrendelő ebben a dokumentumban tudhatja meg véglegesen, hogy mit is fog kapni a fejlesztési folyamat végén, míg a fejlesztő ebben a dokumentumban rögzíti le, hogy mit is kell neki nyújtania a fejlesztés során. A dokumentációnak tartalmaznia kell a következőket:

- A megkívánt rendszer bemenő és kimenő paraméterei.
  Milyen forrásadatok alapján, milyen listák, nyomtatási képek jelennek meg.
- A rendszer összes funkciójának leírását
  A fejlesztés során előre el kell tervezni, hogy milyen funkciókra akarjuk alkalmazni a rendszert. A fejlesztés folyamata közben az utólagos igények kivédésének ez a legmegfelelőbb helye.
- Az adatszerkezeteket.
  A felhasznált adatszerkezetek eleve meghatározzák, hogy milyen adatok nyerhetők ki egyszerűen a rendszerből. Az adatszerkezetek megtervezése a rendszer későbbi használhatóságának alappillére. Ha nem kellően átgondolt a rendszer, vagy kevés adat nyerhető ki belőle vagy túlságosan bonyolult vagy nem kellően rugalmas. A rendszerhez és a vele támasztott követelményekhez illeszteni kell az adatszerkezeteket. Arra is gondolni kell, hogy ha a rendszernek további fejlesztési irányai is lehetnek, akkor azokra a fejlesztésekre is gondolni kell, azaz biztosítani kell az adatszerkezetek olyan kiterjesztését, amelyeket a jelenleg kifejlesztett rendszerek továbbra is használhatnak, de a később kifejlesztett rendszer is felhasználhatja a jelenlegi adatszerkezetek adatait.
- Az esetlegesen előrelátható további fejlesztések valamiféle kereteit.
- A tesztadatok és az éles adatok felvitelének módját.
- A tesztelés módját, a helyesség megállapításainak kritériumait.
- A felhasznált fejlesztőeszközt, és a fejlesztőeszköz hardverét
- A futtatórendszer hardver és szoftverkritériumait.
  A szükséges hardverkonfigurációt, ami a processzor típusa, memória, szükséges terület a HDD-n, kell-e floppy vagy nem, és ha igen, akkor milyen, a megjelenítő minimális felbontását, egyéb kiegészítő eszközöket. A szoftvernél a szükséges operációs rendszert, és minden egyéb kiegészítő szoftvert, ami a futtatáshoz szükséges. Célszerű megjelölni a minimális és az optimális konfigurációt is.
- Hálózati alkalmazásoknál, a szükséges hálózati feltételeket, (Hálózati operációs rendszer típusa, kapcsolódás típusa, stb…)
- A tervben indokolni kell, a fenti szoftver-, és hardverválasztásokat.
- A tervnek tartalmaznia kell a megvalósítás ütemezését, szakaszokra bontva, határidőkkel, felelősökkel. Mindenképpen kell tartalmaznia a kapcsolattartó nevét.
- A tervben le kell fektetni, hogy a fejlesztés során előre nem látható módosításokat a felek hogyan kezelik, azaz ki a felelős, ki dönti el, hogy az esetleges módosítás mennyiben elfogadható és hogy annak a költségeit hogyan viselik. Azt is célszerű leírni, hogy nem teljesen a leírások szerinti működés vagy a fejlesztés csúszása hogyan befolyásolja az egész tervezetet.

5.3 Megvalósítás

Egy program fejlesztése programozói munka. A programozónak a fejlesztés megkezdésekor szüksége van minden keretfeltételre, amely a munka végzését segíti. A tervezés során már mindent tud, ami a rendszer bemeneteit, kimeneteit illeti. A tervben megadott eszközökkel, minél gyorsabban, minél hatékonyabban el kell készítenie a kívánt rendszert. Egy üzleti céllal kifejlesztett programban nagyon fontos, hogy a program szerkezete jól strukturált legyen, azaz célszerű a rendszert a tervezetben meghatározott módon, logikus sorrendben fejleszteni. A fejlesztés sorrendiségét több dolog meghatározhatja, ízlés kérdése, hogy ki milyen módszert követ.

A harmadik generációs rendszerekben a fejlesztés központi része egy jó editor, debugger és profiler rendszer. Ezeket a modulokat az integrált IDE rendszerek tartalmazzák. Az ilyen rendszerek nem támogatják közvetlenül a strukturált programtervezést, bár esetleg maga a nyelv kínálja a struktúrákat. Az ilyen fejlesztőeszközök esetén a programozónak magának kell gondoskodnia a fejlesztés során a hatékonyságról, azaz az adatszerkezetek logikus létrehozásáról, a változónevek következetes és áttekinthető használatáról, az eljárások

és függvények következetes használatáról. Célszerű az ilyen fejlesztőeszközök használatakor gondoskodni a különböző programrészek külön modulokba szervezéséről, az egyes modulok közötti interface-ek szabványos együttműködéséről.

A fejlesztést például célszerű úgy szervezni, hogy:

1. Az adatszerkezetek létrehozása, az adatokat közvetlenül kezelő programrészek létrehozása Célszerű olyan rutincsomag létrehozása, amely az adott programban és az esetlegesen később létrehozandó programokban is felhasználható vagy újra felhasználható. Ez természetesen nem egyszerű, meglehetősen sok általánosítást tartalmazhat, ami plusz munkát jelent, de az elején befektetett munka később sokszorosan megtérül, nem csak a konkrét fejlesztés során, hanem a tesztelés, hibajavítás során is. Ha léteznek már előre elkészített standardizált modulok, akkor azokat megfelelően össze kell gyűjteni, a konkrét feladat szerint esetleg módosítani. Ha az ilyen rendszereket módosítjuk, akkor figyelni kell az esetleges más programokban lévő kapcsolatokra is. A visszafelé érvényes kompatibilitás nagyon fontos szempont.
2. Célszerű már az elején kidolgozni azokat az eljárásokat, amelyek az adatok tesztelésével kapcsolatosak.
3. A következő lépésben például el lehet készíteni a program fő menüpontjait
   A modern 4GL fejlesztőeszközök kínálják a Top-Down módszer alkalmazását, gyakorlatilag másképpen nem is lehet őket hatékonyan használni. Ezekben az esetekben a fejlesztés sorrendjének célszerűen a képernyő, a listaformátumok, a menük megtervezésével kell kezdeni, majd a módszert követve kialakítani az egyre alacsonyabb szintű programrészeket, az egyes képernyőobjektumok tulajdonságainak meghatározásával.

4. Az egyre jobban kidolgozott részletek természetesen állandó kontrollt igényelnek. Minden egyes elkészült programmodult a lehetőségekhez képest azonnal ki kell próbálni, tesztelni kell. A felmerülő hibákat ki kell javítani. Ebben a szakaszban nagy hasznát veszi a fejlesztő, ha a tervezés során és a fő modulokban megfelelően strukturálta a programját, azaz csak a megfelelő interface-eken keresztül kommunikálnak az egyes programrészek. A hibakeresés is könnyebb így.
5. A munka során célszerű a legkritikusabb részeket megírni először. Azokat lehet ezeknek tekinteni, amelyek az adatok bevitelét, az adatok kezelését, az adatok helyességét biztosítják.
6. A későbbiekben az adatok megjelenítéséért felelős részek kerülhetnek sorra. Ezek a részek általában nem hatnak vissza az adatokra, azokat nem módosíthatják, tehát elrontani sem tudják.

A fejlesztés során gyakran jó szolgálatot tesznek az üres eljárások, procedúrák. Ezek olyan meghívható eljárások, amelyek a program későbbi szerkezetében benne lesznek, funkciójuk lesz, de a fejlesztésnek a korai szakaszaiban még nem kell őket kidolgozni. Általában egy megjegyzést meg lehet jeleníteni bennük, esetleg a később feldolgozandó paramétereket is át lehet adni nekik, de a kidolgozásukra majd csak később kerül sor.

A felhasználót folyamatosan tájékoztatni kell a fejlesztés eredményeiről, akár úgy is, hogy tesztváltozatokat hagyunk nála.

7. A legvégén a kényelemmel kapcsolatos még ki nem elégített igényeket célszerű sorra venni. Itt gondolunk elsősorban a Help, a gyorshelp, a gombok megfelelő feliratairа. Meg kell jegyezni, hogy a 4GL nyelvek ezekben is igen nagy szolgálatot tesznek, mivel az adatok tervezése során meg kell adni az adat típusán, méretén kívül elnevezést, magyarázatot stb…
8. Amikor úgy gondoljuk, hogy észen vagyunk a programunkkal, akkor egy módszeres teszten kell átesnie a programnak, amely minden funkcióját módszeresen végigteszteli. Ebben a szakaszban általában fény derül korábban észre nem vett hibákra is. Vannak hibakeresési, tesztelési módszerek, amelyeket a későbbiekben fogunk tárgyalni. Segítségükkel meg lehet találni a hibák nagy részét. A megtalált hibákat ki kell javítani, majd a javítás után újra tesztelni a megfelelő részeket.

A helyesen megtervezett programok egyes moduljai függetlenek, így az egyik részben keletkezett hibák javítása nem hat a más részekre. A nem eléggé meggondoltan tervezett programokban előfordul az, ahogy az egyik rész hibájának kijavítása más részekben okoz hibát. Az ilyen programok hibamentesek soha nem is lehetnek. Minél később derülnek ki az ilyen függőségek, annál nehezebb a programot normális állapotba hozni.

9. Ha nem találtunk hibát, akkor sor kerülhet egy elsődleges hatékonyságvizsgálatra. Itt vizsgálni kell, hogy az éles adatok mennyiségéhez mérhető adatok esetén a futtató gépeken milyen teljesítményt produkál a rendszerünk. A hatékonysági problémák oka akár az algoritmusok, akár az adatszerkezetek szintjén is kereshetők. Meg kell keresni a nem megfelelő hatékonyság okát, és a megfelelő beavatkozással meg kell megszüntetni. Természetesen utána újabb tesztperiódus következik.

Előbb- utóbb a fejlesztő nem talál több hibát. Ennek pszichológiai okai is vannak. Hiába törekszik valaki módszeresen dolgozni, de a fejlesztés során a figyelmét elkerülő részletek általában később is elkerülik a figyelmét. Ilyenkor a külső tesztelők általában rögtön kiszúrják a hibát. A felhasználó megkapja a program tesztváltozatát.

A szakzsargonban **Alfa változatnak** hívják a még alig tesztelt változatokat, illetve azokat, amelyeket csak maga a fejlesztő tesztelt és még fejleszteni akar rajta. **Béta változatnak** nevezik azt a programot, amelyben a fejlesztő már nem talál hibát és kiadja külsősöknek tesztelésre. Ezekben a programokban a fő funkciók már eléggé megbízhatóan működnek, de korántsem tekinthető befejezettnek a program semmilyen szempontból. (Állítólag a COREL cég vezette be a nemzetközi szoftveres köztudatba a Bétatesztelés fogalmát, mert volt idő, amikor a szoftverei az eladás után fél évvel kezdtek csak használhatókká válni, sok javítókészlet, és patch – olyan kisebb terjedelmű javítás, amely egy-két modult lecserél, esetleg a program bináris formájában kicserél néhány gépi kódú utasítást - kiadása után.)

5.4 Javított változat

A felhasználó tesztelése közben elkészítjük a felhasználói dokumentációkat, majd a hibalista kézhezvétele után megkezdjük a hibajavítást, majd újabb hibakeresési, tesztelési, esetleg hatékonyság-vizsgálati eljárás következik. Az így tesztelt programot újra átadjuk a Felhasználónak tesztelésre.

A fenti ciklusok addig folytatódnak, amíg a felhasználó ki nem jelenti, hogy a program a tesztadatokkal jól fut. Ekkor kell kipróbálni az éles adatokkal, majd ha azokkal is hibátlanul fut, akkor a programot átadhatónak lehet tekinteni, és a Fejlesztő átadja, a Felhasználó pedig átveszi.

5.5 Végleges változat, és továbbfejlesztés

A végleges változatot a Felhasználó üzemszerűen használja, majd egy idő múlva csak azt használja. A program használata közben persze kiderülhetnek újabb hibák is, amelyek súlyuktól függően hibás működést is eredményezhetnek vagy csak szépséghibának tekinthetők. Általában fél év üzemszerű használat kell ahhoz, hogy egy komolyabb fejlesztésről kiderüljön, hogy mindenben megfelel és hibátlan.

Gyakran fél-egy év használat során derülnek ki azok a hiányosságok, amelyek a programok továbbfejlesztését indokolttá teszik. A továbbfejlesztés során biztosítani kell az addigi verzió üzemszerű működését, illetve egy korábbi stabil változathoz való visszatérés lehetőségét. Ennek megfelelően az adatszerkezeteket módosítani csak nagyon indokolt esetekben szabad. A továbbfejlesztést azonban úgy kell tekinteni, mint egy új projectet, igaz a feladat általában nem olyan mértékű, mint korábban, és a körülmények is tisztábbak.

6 Algoritmusok

Az ember a fejében megbúvó gondolatokat tevékenységek sorozatára fordítja le. A fordítás eredménye egy tevékenység sorozat, amelyet fel tud vázolni magának valamilyen előre definiált leírt módszerrel. Ezt a rögzített cselekvéssorozatot hívják **algoritmusnak**. Az algoritmus elsősorban az embernek szóló valamiféle formális leírás. Az algoritmusok leírásának jól definiált módszerei vannak.

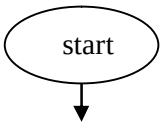
6.1 Algoritmuselíró módszerek, nyelvek

Három féle algoritmusleíró módszer terjedt el Magyarországon. Az első leírási módszert még a második generációs gépek idején találták ki, ma is sokan használják. **Ez a folyamatábra**. Ahogy a programozás egyre inkább tudománnyá vált a folyamatábra már nem felelt meg az egzakt és absztrakt ábrázolásnak, a programozók kitalálták a **struktogram**-ot. A struktogram precízen ábrázolja a folyamatokat, ugyanakkor olvasása esetenként nehézkes, a valódi programozástól kicsit elrugaszkodott. A programozás oktatásában vezették be a **mondatszerű** leírást **vagy pszeudo-kódot**. A felsorolt három algoritmus-leíró módszer egyenrangú, mind a három alkalmas arra, hogy kisebb-nagyobb rendszerek algoritmusait megfogalmazzuk segítségükkel.

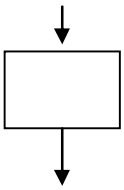
6.1.1 Folyamatábra

A folyamatábrák használatakor grafikai jelekkel jelöljük az egyes programozási egységeket. A grafikai jeleket nyilakkal kötjük össze, a program futásának megfelelő módon. Általában egy program vagy egy eljárás fentről lefelé vagy balról jobbra hajtódik végre, legalábbis a folyamatábra rajzolásánál erre kell törekedni. A szokásos feldolgozási lépéseket az alábbi módon szimbólumokkal jelöljük:

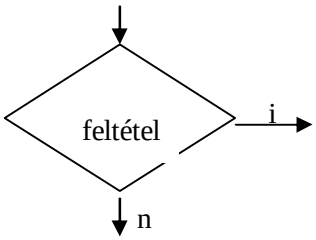
Az algoritmus elejét így jelöljük:



Egy utasítás, az utasításba beleírjuk az utasítást.

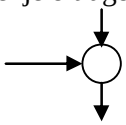


Elágazás - eldöntés



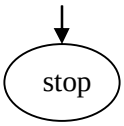
Ha a feltétel igaz, akkor az i-vel jelölt ágon a program, egyébként az n-nel jelölt ágon folyik tovább.

Ahol két végrehajtási szál találkozik, azt a következőképpen jelöljük:



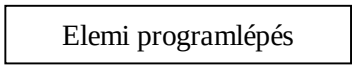
A ciklusok leírása külön nincsen, kézzel kell összeállítani.

A program végét a következő szimbólum jelöli:

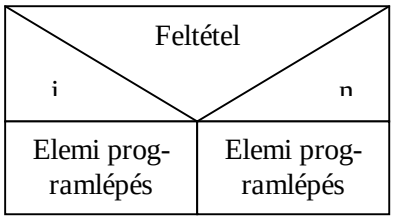


6.1.2 Struktogram

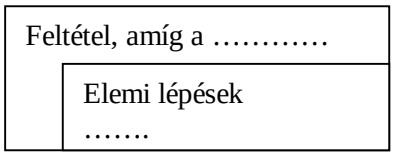
A struktogram algoritmus leíró módszer elsősorban a professzionális programozás oktatása során használandó. Ennek a leíró módszernek az előnye rendkívül egzakt módjában és tömörségében van. Talán kezdők számára nem annyira áttekinthető, mint a folyamatábra, de annak hibáit kiküszöböli.
A programban egy elemi lépést a következő jelzi:



Feltételes elágazás:



A ciklust az alábbi módon jelölhetjük:



6.1.3 Mondatszerű leírás

A mondatszerű leírásnak akkor van értelme, ha a programozó valamilyen szinten már tisztában van egy számítógépes program működésével, érti az utasítások egymásutániságának, az értékadásnak, az elágazásoknak, a ciklusoknak a fogalmát. Érti azt, hogy mit jelent egy eljárás hívása és mit jelent egy függvény hívása. Mint látjuk majd, az algoritmus-leíró nyelv egyfajta magyar nyelvű programozási nyelv. Ennek a nyelvnek egy oktatási célú megvalósítása a Budapesti Műszaki Egyetemen kifejlesztett ELAN nyelv.

A mondatszerű leírás a következő szintaktikai szabályokat tartalmazza:

Értékadás:

```
:=   A két oldalon azonos típusú értékek vannak
```

Feltételes elágazás:

```
Ha feltétel igaz akkor Utasítás
```

vagy

```
Ha feltétel igaz akkor
     Utasítások …
Elágazás vége
```

vagy

```
Ha feltétel igaz akkor
     Utasítások …
Különben
     Utasítások …
Elágazás vége
```

Ciklusok:

```
Ciklus cv :=kezdőértéktől végértékig lépésközzel
     ……
Ciklus vége
```

vagy

```
Ciklus amíg feltétel igaz
     ……
Ciklus vége
```

vagy

```
Ciklus
      ……
amíg feltétel
Ciklus vége
```

Kezdőérték adás, inicializálás

Ki:	**Kiíró utasítás**
Be:	**Adatbeviteli utasítás**
Tömb:	**Tömb definiálása**

```
Inicializálás(kezdőérték)  Bizonyos adatok kezdőértékének megadása
```

Program struktúrák

```
Program Név
      ……
Program vége

Eljárás Név(paraméterlista)
      ……
Eljárás vége

Függvény Név(paraméterlista)
      ……
      Nev := visszatérési érték
Függvény vége
```

Az eljárásokat és függvényeket az alábbi módon hívhatjuk meg a programokból:

```
Név( paraméterlista )
```

vagy

```
Érték := Név( paraméterlista )
```

6.2 Az algoritmusok dokumentálása

Az algoritmusok használatakor szükséges megadni a **programok**, az eljárások **specifikációját**, azaz milyen bemenő adatokat vár a program, azokra milyen feltételek érvényesek, továbbá az elvárt kimeneti adatokat is meg kell adni, azokra milyen feltételeket várunk, mikor tekintjük helyesnek és mikor hibásnak az eredményt.

Minden esetben szükséges a programban meghívott, nem triviális eljárások esetén leírni, hogy mit várunk a meghívott eljárástól. Az eljárás kifejtésekor pedig célszerű leírni az **eljárás fejléceként**, hogy az eljárás mit végez el, milyen típusú bemenő adatokat produkál és milyen kimenő adatokat várunk tőle.

A folyamatábrák és struktogrammok használata során az algoritmusokat célszerű bő magyarázattal ellátni tekintve, hogy az előbbi két leírás meglehetősen absztrakt. Az algoritmusleíró nyelv használata során a dokumentálás hozzávetőlegesen ugyanolyan mértékű kell, hogy legyen, mint a programok írásánál. Az nem teljesen magától értetődő zárt programrészek előtt tömören le kell írni a funkciójukat, esetleg annak magyarázatát, hogy miért pont úgy, azokkal az eszközökkel valósítjuk meg a folyamatot.

6.3 Elemi algoritmusok, programozási tételek

A fejezetcímben említett témák az „Adatszerkezetek és programozási tételek” c. jegyzetben találhatók meg.

7 A megvalósítás gyakorlati eszközei

Az algoritmusok általában még nem eléggé egzaktak, hogy valamilyen automatizmus feldolgozza őket, ezért azokat át kell ültetni egy programozási nyelv kereteibe.

A **programozási nyelveknek** előre lefoglalt szavai vannak. A lefoglalt szavak listáját **utasításkészletnek** hívjuk. Az utasításokat csak bizonyos formai szabályok betartásával használhatjuk, ezeknek a formai szabályoknak az összességét hívjuk a nyelv **szintaktikájának** neveznek. A szövegeket ezek alapján értelmezi és fordítja le a fordítóprogram, úgynevezett **gépi kódra**. A gépi kód az a számsorozat, amelyet a számítógép processzora értelmezni tud és végre tud hajtani.

Például:	*MOV AX,134*	*vagy*	*Gotoxy(11,13);*
	ADD AX,75		Writeln('Üdvözlöm Önöket!');

A bal oldali oszlop két sora a számítógép memóriájában 8 byte-nyi helyet foglal el, összead két egész számot, és egy memóriahelyre leteszi őket. Az egyszerűbb felhasználói programok is minimum néhány ezer byte hosszúak, azaz ilyen utasításokból jó néhány száz kell. Ezt hívják assembly programozásnak. Lassú és nehézkes, de a gép erőforrásait így lehet a legjobban kihasználni.
A jobb oldali oszlop két utasítása kiírja a képernyő 13. sorának 11. oszlopától kezdődően, hogy Üdvözlöm Önt. Ez is két soros program, de valami értelmeset is végrehajt.

7.1 Compiler, interpreter, p-kód

Egy tetszőleges nyelven megírt program lehetséges, hogy szintaktikailag helyes utasításokat tartalmaz, de nem biztos, hogy a program értelme dolgokat művel. A program gondolatmenetének hibátlanságát úgynevezett **szemantikai szabályokkal** ellenőrzik. Általában a rendszerek nem nagyon figyelnek oda a szemantikai szabályokra, de vannak olyan rendszerek, amelyek felismerik, ha a fejlesztő szintaktikailag helyes, de szemantikailag értelmetlen utasításokat ír le. Ezekre figyelmezteti a programozót és annak döntésétől függően, esetleg módosít a végeredményen. Gyakran előre kifejlesztett módszerek segítségével a programképes a végső futtatható változatot optimalizálni is.

Gyakorlatban ezt a tiszta rendszert a **compileres** rendszereknek hívják.

Ezekben a rendszerekben a fordítás során a compiler szintaktikailag ellenőrzi a programszöveget, és az esetleges hibákat kijelzi a fejlesztőnek, megjelölve a megfelelő helyet a programszövegben. Bizonyos esetekben a szemantikai szabályok alapján is ellenőrzi a szöveget a rendszer. A szemantikai szabályoknak való megfeleltetést általában nem jelzi kritikus hibáknak, azaz a program ezektől még lefordítható.

A forrásszöveget a compiler úgynevezett object formátumba fordítja le, vagy más néven **object kód**-ba. Az object kód az Intel által szabványosított köztes állapot a forrásszöveg és a gépi kód között. Az így keletkezett fájl a DOS - Windows rendszerekben a .OBJ kiterjesztést kapják. Ebben az állapotban az úgynevezett linker – szerkesztő – programok veszik át a további irányítást és az egyes programrészek .OBJ fájljaiból, továbbá az előre elkészített LIB fájlokból összeszerkeszti a végleges futtatható programot. A LIB fájlok szabványosan tartalmaznak több OBJ fájlt egy egységbe csomagolva. A LIB fájlok szerkesztéséhez a Microsoft LIB.EXE vagy a Borland TLIB.EXE programja alkalmas. A programok segítségével a LIB fájlokba új object kód helyezhető el, régi obj kód vehető ki, illetve a tartalmuk kilistázhatók.

Windowsos programok esetén még egy lépés szükséges, a programokhoz hozzá kell szerkeszteni a képernyőnn megjelenő objektumok forrását is. Ezt a Resource Compiler programok végzik el. Ilyen pl. a Microsoft BRC.EXE programja is.

A compileres rendszerek tagadhatatlan előnyei a következők:

- Mint látható, a compileres rendszerek a fejlesztés során fordítják le és tesztelik a forrásszöveget. Mivel ezek a rendszerek a fordítás során áttekintik az egész rendszert is, ezért a szintaktikai, szemantikai hibák többségét képesek kiszűrni az ilyen rendszerek.
- A keletkezett gépi kód a processzor típusára optimalizált lehet.

- A fordítónak meg lehet adni, hogy az elkészült gépi kód milyen ellenőrzéseket végezzen a futás közben. Memória verem ellenőrzés, memóriatúlcsordulás ellenőrzése, tömbhatárok ellenőrzése, adattípus megfeleltetés, stb – ilyen típusú ellenőrzések jöhetnek szóba, ezeket a fejlesztő kívánsága szerint befordítja a compiler a programba. Amikor a program már végső tesztelt, hibátlan állapotba kerül, akkor a fordítási opciókat átállítva a program a lehető legkisebb és leggyorsabb lehet.

Hátrányok is vannak persze:
- A compileres rendszerekben a szerkesztés, fordítás, tesztelés fejlesztési ciklus akár nagyon hosszú is lehet, hiszen a fordításkor a rendszer áttekinti a teljes fordítani valót. Persze léteznek olyan eljárások, amelyek az egyes modulok lefordításának, a forrásszövegek időpontjának elemzéséből meg tudják határozni, hogy mit kell újrafordítani és mit nem. Az újraszerkesztést, azonban általában nem lehet megspórolni. Lassabb gépeken egy teljes fejlesztési ciklus sok időt vehet igénybe.
- Nem lehet párbeszédes üzemben működtetni ezeket a rendszereket. A program futását megszakítva általában nem lehet folytatni a programot ugyanattól a ponttól, ha közben megváltoztattuk a program állapotát.
- Bár vannak hibakereső programok, de nem feltétlenül használhatók minden körülmények között.

A Microsoft Visual C, Borland Pascal, Borland C++, Delphi compileres rendszerek.

Interpreteres rendszerek:
Klasszikus ilyen rendszer a BASIC programozási környezet és nyelv. A rendszer lényege, hogy a program bevitelekor a rendszer egy szintaktikai ellenőrzést végez a bevitt utasításon. A program végrehajtása során a rendszer utasításonként elvégzi a következőket:
- Beolvassa a következő utasítást, és eldönti, hogy az utasításkészlet melyik utasítása.
- Ellenőrzi, hogy az adott utasítás szintaktikailag helyes-e, az átadott paraméterek típusa, esetleg mérete megfelel-e az utasítás kívánalmainak
- Ha hibátlan, akkor meghívja az utasításhoz tartozó előre elkészített kódrészletet.
- Figyeli, hogy a végrehajtás során nem jön-e létre valamilyen hiba. Ha hibára fut a rendszer, akkor hibakódot kell generálnia. Ha hibátlan a végrehajtás, akkor veszi a következő utasítást.

Hátrányok:
- A fejlesztő dolga annak megállapítása, hogy szemantikailag helyes-e a program.
- A program optimalizálása automatikusan semmilyen formában nem jön létre.
- Az így futtatott program sokkal lassabb, mint a megfelelő compilerrel lefordított gépi kódú program.
- A program futtatásához speciális futtatókörnyezet szükséges, amelynek tartalmaznia kell minden olyan segédállományt, amely az esetleg előforduló összes utasítás megfelelő gépi kódját tartalmazza. Ez a program méretét feleslegesen megnöveli, mivel olyan modulok is bent vannak a memóriában, amelyekhez tartozó utasítást nem is hajt végre a program. A programok mérete nagyobb, mint a megfelelő compilerrel fordított programoké.
- Az ilyen programok memóriaszervezése bonyolultabb, mint a compileres társaiké.
- A helyesen működő programok előállításának kisebb az esélye.

Előnyök:
- Az interpreteres rendszerekben a fejlesztési ciklus lerövidül, gyorsan lehet dolgozni.
- Az elkészült programrészleteket gyorsan ki lehet próbálni, tesztelni.
- Kisebb teljesítményű gépeken is ugyanolyan gyors a program, mint a nagyobbakon – (azaz ugyanolyan lassú)

P-kódú rendszerek
Ez egy öszvér megoldás. A fejlesztő rendszer compilere és linkere futtatható programot fordít, de a program bizonyos részei un. P-kóddá fodítódnak, ami átmenet a gépi kód és a programszöveg között. Ezt a kódot futás közben értékeli ki a rendszer. Önállóan futtatható EXE fájl az eredmény, de nem optimális a program sebesség és méret szempontjából. Az ilyen rendszerek igyekeznek a korábbi két rendszer előnyeit ötvözni.

A gyakorlatban a gépek teljesítményének növekedésével a compileres rendszerek előtérbe kerültek. Vannak olyan feladatok, amelyeket nem lehet interpreterrel megoldani, míg más feladatok megoldása csakis azzal képzelhető el.

A Pascal, a C, C++ nyelvek és más harmadik generációs nyelvek compileres rendszerekként valósulnak meg, míg a BASIC, Visual BASIC régebbi változatai, illetve a Microsoft Office termékcsaládjának mindegyik darabja interpreteres rendszer. A Word az Excel vagy az Access programozása interpreter közreműködésével jön létre.

A Clipper P-kódot hoz létre. A rendszer ezt a köztes kódot nem ellenőrzi futás közben szintaktikai és szemantikai szempontból. Csak a memóriakezelés, a változók értékhatárait ellenőrzi. Az így keletkezett program végrehajtási sebessége valahol a compiler és interpreter nyelvek között van, mérete is, mivel a p-kódra fordításkor csak azok a modulok kerülnek be a futtatható file-ba, amelyek benne vannak a fordított rendszerben.

7.2 A programozási nyelvek szintjei

Alacsonyszintű nyelvek - assembly
Ebben a nyelvben egy gépi kódú utasítás egy assembly utasításnak felel meg. A programok fejlesztése nehézkes, viszont a gép erőforrásait maximálisan ki lehet használni, a program a lehető leggyorsabb és a legkisebb helyet foglalja el. A mikroprocesszor típusától függően más gépfajtán más a nyelv is!

Középszintű nyelvek C, C++
Egy utasításhoz néha egy gépi utasítás, néha több tartozik. Viszonylag jól lehet kihasználni a gép adta lehetőségeket, de furcsa módon a C programok általában gépfüggetlenek. Van olyan program, amit a különböző gépfajtákra elkészítettek, mégis az eredeti programszövegük 70-80 %-ban azonos, mert C-ben írták őket. Ez assembly nyelven nem lehetne. Gyors a fejlesztés, a programok rövidek, gyorsak.

Magas szintű nyelvek - Pascal, dBase, Clipper, Fortran, BASIC
Ezeknek a nyelveknek az utasításai egészen összetett tevékenységeket hajtanak végre. A programok fejlesztése gyors, a gép tulajdonságait is ki lehet használni, mivel a nyelvekhez előre megírt programkönyvtárak állnak rendelkezésre. A programok mérete nagy, legalább néhányszor tíz kilobyte.

Nagyon magas szintű fejlesztőeszközök
Az utóbbi években elterjedtek olyan fejlesztőeszközök, amelyek segítségével gyorsan összetett alkalmazásokat lehet létrehozni, akár programozói tudás nélkül is. Ezeket 4GL (4. Generation Language - Negyedik generációs) nyelveknek hívják. Visual Basic, Clarion, ReMind, Magic, Borland Delphi, C Builder, Visual FoxPro, CA Visual Object stb.

Az elkészült alkalmazás természetesen nem a leggyorsabb és legkisebb lesz, amit csak ki lehet találni, de gyorsan elkészül és könnyen, egységesen felhasználható, mivel nem a programozó tudásán múlik ez a tulajdonsága.

Megjegyzések:

A robot vezérlése alkalmas arra, hogy segítségével elsajátítsa valaki az algoritmikus gondolkodás elemeit. Az algoritmusok minden eleme felfedezhető a robot tevékenységében.

Amikor a robot egymás utáni lépéseket tesz meg a vezérlő programban utasítások sorozatát kell kiadni.

Amikor figyelni kell egy tárgyat vagy képet és össze kell hasonlítani dolgokkal, akkor az elágazás vezérlési szerkezetet használjuk.

Amikor oda-vissza járkál, akkor ciklus vezérlési szerkezetet kell használni.

Az egyre bonyolultabb tevékenységeket részegységekre bontva eljárásokat írunk. A látás, információ bevitelével egyenlő, a beszéd pedig információ kivitel, azaz input-output műveleteket is használunk.

A C nyelvet B. Kernighan tervezte, mikor egy operációs rendszer kifejlesztésére kapott megbízást. Az assembly helyett írt hozzá egy fordítót, majd segítségével megírta a UNIX operációs rendszer egy változatát. A nyelv a professzionális fejlesztők között népszerű lett. Ennek továbbfejlesztése az objektum orientált nyelv a C++. Vegyes nyelvű programozás esetén az assembly-n kívül C-t szoktak használni. Általában más nyelvek elfogadják társnak, amikor vegyes nyelven írnak egy programot.

A Pascal nyelvet eredetileg N. Wirth a Zürich Műszaki Egyetem tanára oktatási célra tervezte. A Borland cég Turbo Pascal-jával lett világhírű a nyelv, ami a szabvány Pascal-nak a kiterjesztése. Mára általános célú nyelvként tetszőleges programozási feladatra használják.

A BASIC nyelv a hetvenes években lett népszerű a játékgépek miatt. Könnyű vele működő programot írni, ezért a kezdők előszeretettel használják. Soha nem szabványosították. A Microsoft által továbbfejlesztett Visual Basic az egyik leggyakrabban használt windowsos fejlesztőeszköz. A DOS is tartalmaz egy QBASIC nevű változatot.

A DBASE programozási nyelve kifejezetten adatbázis-kezelési feladatok megoldására való és a DBASE adatbázis-kezelő rendszerben lehet vele programokat írni és a rendszer segítségével futtathatók. Eredetileg ennek a fordítóprogramjaként született meg a CLIPPER programozási nyelv, ami mára önálló életet él és hatékonyabb mint az eredeti. Segítségével az alkalmazások önálló .EXE fájlként futtathatók. Professzionális nyelv.

7.3 A programozási nyelvek másik féle osztályozása

A programozási nyelveket szokás más módon is osztályozni. A legelterjedtebb és leggyakrabban használatos nyelvekben – Pascal, C, C++, Basic, stb. – van egy közös momentum. Ezek úgynevezett **procedurális-,** vagy más néven **Neumann elvű programozási nyelvek.** Közös bennük, hogy a programot egyfajta tevékenységek sorozataként képzelik el, ami mellesleg a leggyakrabban valóban a leghasznosabb is. Tulajdonságaik:

- Vannak változóik, vannak adatszerkezeteik.
- A program és a memória fizikailag közös memóriában helyezkedik el, azaz a program állapotát a memória állapota egyértelműen meghatározza. (Ha kimentjük a memóriatartalmat, majd kikapcsoljuk a gépet és bekapcsolás után visszatöltjük a memóriatartalmat, akkor a program ugyanonnan fut tovább)
- Az adatok beolvasása és kiírása a memóriában történő adatmásolás eredménye.
- Vannak vezérlési szerkezeteik.
- A programnak mindig van eleje, a programozó által megszabottan a program minden pillanatban ismert helyen fut, és a megadott eljárások lefutása után a program véget ér.

Sajnos ezek a nyelvek bizonyos feladattípusok megalkotására nem mindig célravezetők.

Képzeljünk el egy olyan esetet, amikor egy ipari robotot kell vezérelni egy számítógéppel. A robot állapota, helye, sebessége és még néhány paramétere a fontos. Az ilyen robotokat vezérlő számítógépek általában nem nagy teljesítményű PC-k, hanem a gép képességeihez tervezett fedélzeti számítógépek, a megfelelő célorientált nyelvvel. Az ilyen programozási nyelveket **automata elvű programozási nyelveknek** hívják. Tulajdonságaik:

- A program és a vezérléshez szükséges adatok elkülönülten vannak a számítógép különböző célú memóriaterületein.
- Az ilyen nyelveken nincsen változó, nincsen értékadás.
- Az ipari berendezés állapota egyúttal a program állapotát is jellemzi. A program változtatja a robot állapotát.

- A külvilág felől a program paramétereken keresztül kapja meg az indításhoz szükséges adatokat.

Mivel csak bizonyos célok végrehajtására hozzák létre a robotot, általában az ilyen rendszer csak adott típusú adatokkal tud dolgozni. Ennek a programozási nyelvnek a tipikus példája a LOGO programozási nyelv.

A harmadik nagy nyelvcsaládot a **függvényelvű programozási nyelvek** alkotják. Az ilyen nyelveken a programot függvényként képzelik el. A program futtatása során a programot alkotó függvényt értékeljük ki. Ezeknek a nyelveknek a tulajdonságai:

- Nincsenek változók, de a függvényeknek vannak paramétereik és van visszatérési értékük.
- Van feltételes elágazás, hiszen egy függvényt lehet úgy definiálni, hogy egy feltétel esetén egyfajta definíciója legyen, míg ha a feltétel nem igaz, akkor más definíció legyen értelmes.
- Van rekurzió, de nincsen ciklus, hiszen mint majd később látjuk minden ciklikus eljárást át lehet fogalmazni rekurzív eljárássá.
- Az adatbevitel egy függvény paraméterátadásával valósul meg, az adatok kiírása a függvények eredménye.

A programozási nyelvek egy speciális fajtáját jelentik a **logikai nyelvek**. A logikai nyelvek szerint a program egy kiértékelendő logikai formula. A program végrehajtása a formula kiértékelését jelenti. Tulajdonságai hasonlóak a függvényelvű programozási nyelvekhez, csak egy lényeges különbség van. Létezik bennük a visszalépéses keresés, a **backtrack** eljáráson alapuló kiértékelési rendszer.

Ennek hatására az ilyen nyelveken olyan kérdéseket lehet feltenni, hogy bizonyos alapadat halmazra felteszünk kérdéseket, és a program megadja a választ, hogy a kérdésre adott válasz igaz vagy hamis. Ezt oly módon teszi meg, hogy végigpróbálja az adathalmaz összes lehetséges állapotát, megvizsgálja, hogy a kérdés milyen választ ad, és ha igaz a válasz, akkor az adathalmaz pillanatnyi állapotát adja vissza eredménynek.

A logikai nyelvek családjának legelterjedtebb tagja a Prolog programozási nyelv. Az ilyen nyelvek egy kis kiegészítéssel rendkívüli dolgokra képesek.

Ha egy logikai nyelven meg lehet oldani, hogy a program a feltett kérdésekre adott válaszokat megjegyezze, akkor a program új adatok birtokába jut, az bővül azoknak az adatoknak a köre, amelyekre új kérdéseket lehet feltenni. Az ilyen nyelven megvalósított programok tehát egyre bővülő tudásbázisú programok — tanuló programok. Valójában ezek a programok a feltett kérdéseket és a rájuk adott választ programkód formájában elmentik, és az újabb futtatáskor az új kérdésre adott válasz már a rendszer adatai között van.

A LISP programozási nyelv alkalmas az ilyen feladatok megoldására. Egyetlen hibája, hogy nem compileres, hanem interpreteres nyelv, ennek megfelelően igazán nagy méretű programok nem futnak rajtuk elfogadható sebességgel. A Magyar Tudományos Akadémia mesterséges intelligencia kutatásával foglalkozó kutatói használják a fent felsorolt logikai típusú programozási nyelveket.

8 Programkódolás

A programozási feladatok megoldása során eddig nem vettük figyelembe az egyes programozási nyelvek speciális tulajdonságait. Sajnos az algoritmus-leíró nyelveken elkészített programokat még a kiválasztott programozási nyelveken kódolni kell. A kódolás során szembetalálkozunk az egyes programozási nyelvek eltérő tulajdonságaival, nem csak a szintaktikus szabályok, hanem a szemantikai szabályok terén is.

Eleve az algoritmus-leíró nyelvek igazából procedurális nyelveknek tekinthetők, azaz az algoritmus kódolása egy valódi programozási nyelvre legkönnyebben ilyen típusú nyelvekre megy.

8.1 Programozási tételek használata

A programozási tételek a felhasznált nyelvtől függően más és más alakot ölthetnek, amikor egy konkrét nyelven megvalósítjuk őket. Ennek megfelelően a programozási nyelvek leírásánál kerültük a nyelvfüggő részek használatát. A programozási tételek ismerete elősegíti, hogy bizonyos típusfeladatok megoldásánál felismerve a megfelelő programozási tételt, a programozó sokkal gyorsabban kódolja le az algoritmust.

Szűkebb memória vagy háttértár esetén a programozási tételeket nem lehet teljes mértékben használni. Ebben az esetben ragaszkodva a tételek lelkéhez, kerülő utakat kell keresni. Például, ha túl nagy annak a tömbnek a mérete, amin munkát akarunk végezni, akkor a tömböt helyettesíthetjük egy véletlen elérésű fájllal is, ahol a fájl rekordjai helyettesítik a tömb elemeit. Természetesen a program végrehajtási sebessége már jóval lassabb lesz, mintha az adat csak a memóriában foglalna helyet, és a kód is bonyolultabb lesz, de megfelelően hajtja végig a nagyobb adathalmazra is a kellő műveleteket.

Ha egy programozási nyelv nem támogatja a rekurziót, akkor a tanult módon a rekurzív algoritmusokat át kell és lehet írni ciklusokat tartalmazó formába.

8.2 Egyes programozási nyelvek eltérő kódolási lehetőségei, módszerei

A különböző programozási nyelvek rendkívül sokféle lehetőséget biztosítanak arra, hogy ugyanazt az algoritmust hány féleképpen lehessen kódolni. A továbbiakban a programozási nyelvek olyan eltérő tulajdonságaira szeretnénk felhívni a figyelmet, amelyek az algoritmus-leíró nyelven megírt programok kódolásánál problémákat okozhatnak.

Névadási szabályok

Az algoritmusok írásakor egy alapvető szabály, hogy beszédes azonosítókat használjunk. Az eltérő nyelvek az azonosítók számára eltérő előírásokat adhatnak.

- Gyakori a név hosszának maximalizálása. A BASIC programozási nyelv régebbi kiadásai, de a Standard Pascal, Assembly és a C régebbi implementációi maximalizálták a név hosszát. A BASIC esetén ez két karakter volt, a Pascal és a C esetén nyolc karakter. A Clipper programozási nyelv tíz karakterben maximalizálja az azonosítók hosszát. Manapság a Visual Basic, a Borland Pascal, és a C is legalább 20 karaktert engedélyez az azonosítók használatára.
- Kisbetű – nagybetű. A Pascal, a BASIC, a Clipper közömbös a betű írásának módjára, nagybetűre konvertálja belsőleg az összes azonosítót, míg a C/C++, Assembly, Java, Javascript, PHP és Visual Basic következetesen elkülöníti a kisbetűs és a nagybetűvel írott azonosítókat. Ez sokszor igen nehezen megtalálható hibaforrás, ha a programozó sokat használta a Pascal nyelvet.
- A BASIC nyelv korábban a változók definiálásakor a változó nevével megadta annak típusát is. Nem volt arra lehetőség, hogy tetszőleges nevet adhassunk a változónak. Kódoláskor sok hibát okozhatott.
- Beszédes azonosítókat kell használni. Ha a nyelv engedélyezi, akkor az algoritmusban használt hosszú neveket kell használni kivéve, ha interpreteres nyelvről van szó. Ez esetben valamilyen ésszerű kompromisszumot kell kötni a név érthetősége és végrehajtási idejének optimalizálása között. Célszerű betartani néhány jól bevált szokást:
  - az abc első betűit paramétereknek használjuk,
  - az i,j,k,l… betűket index, illetve segédváltozóknak használjuk,
  - az x,y,z.. koordináta kijelölésére alkalmas vagy segédváltozóknak.
  - a min, max azonosítókat, illetve ezeknek az összetételeit mindig valamilyen maximális vagy minimális érték kijelölésére célszerű használni.
  - a nagybetűs azonosítókat konstansokra használjuk.

Kezdőérték adásának szabálya
Az egyes programozási nyelvek az új változók létrejöttekor nem egyforma módon kezelik a változók kezdőértékeit. Különösen igaz ez az alacsony szintű nyelveken. Ha egy tömb kezdőértékeit kiolvassuk C nyelven, valószínűleg nagy meglepetésben lesz részünk. A tömb mindenféle memóriaszemetet fog tartalmazni. A Pascal, a Basic, a Clipper és még sok más nyelv a létrejött változó kezdőértékének az aktuális típusnak megfelelő 0-át, üres sztringet vagy hasonlót ad, de ebben nem lehetünk 100%-ig biztosak. Ennek megfelelően mindig kell kezdőértéket adni egy újonnan létrejött változónak, akár lokális, akár globális változóról van szó.

Nem minden programozási nyelv támogat minden típusú adatot
A különböző nyelvek más és más mértékben támogatják az eltérő adattípusokat. A kódolásnál ezt figyelembe kell venni.

- A lista adatszerkezetet közvetlenül nagyon kevés nyelv támogatja.
- Az Assembly nyelvnek természetesen csak a byte illetve az integer, illetve a mutató adattípus létezik. Semmilyen bonyolultabb adatszerkezet definiálására nincsen a nyelvben lehetőség.
- A BASIC például a rekordokat, a halmazokat illetve egyéb összetettebb adattípusokat nem támogat.
- A mutatókból felépített adatszerkezetek olyan szerkezetek, amelyek nem minden nyelvben támogatottak.
- A numerikus típusokra a Pascalnak, a C-nek több, de a Basic-nek csak egy lebegőpontos típusa van. A Clipper lebegőpontos típusa különös, ugyanis karakteres módon tárolja az adatokat, megjelölve a szükséges tizedesjegyek számát.
- Az egész numerikus típusok esetén a legtöbb nyelv ad egy célszerűen használható alapértelmezést, a kétbájtos, integer típust. Ha az értelmezési tartományba nem fér be az érték, akkor az adott nyelv lehetőségei között keresni kell egy ilyen típust. A C és a Pascal kínál hosszabb egész típusokat, de BASIC nem.
- A pointereket a Clipper, a BASIC, illetve a Visual Basic nem támogatja. Ennek megfelelően dinamikus adatfoglalás sem lehetséges a Visual Basic-ben
- A Pointerek használhatók és használandók a C, X++, Pascal nyelvekben..
- Új adattípusok létrehozására nincsen lehetőség több nyelvben, mint például a BASIC vagy a Clipper.
- Csak az objektum-orientált nyelvek támogatják az objektumok használatát. Ennek megfelelően a Borland Pascal, Borland C++, a Microsoft C++, Visual Basic, Java, Javascript nyelvek objektum-orientáltak. A Clippernek van objektum-orientált kiterjesztése (Fivewin), azaz definiálhatók és felhasználható benne objektumok.

A tömbök kezelése
A tömbök kezelése sem minden nyelvben egyforma. A Basic, Pascal, C, C++ nyelvekben előre kell definiálni egy tömb méretét, míg a Visual Basic, Clipper nyelvekben lehetőség van a futás közbeni tömbdefiniálásra, sőt a tömb méretének változtatására is. A Java, Javascript, PHP és Clipperben például új adatot lehet a tömb részévé tenni, régi adatot ki lehet venni, eközben a tömb többi adata megmarad, és a tömb mérete megfelelően változik. A Clipperben a tömbben nem csak azonos típusú adatok lehetnek. Menet közben is tudunk egy tömb részévé tenni más tömböket. Assemblyben nincsen tömb.

Sztringek mérete
A legtöbb nyelven a stringeknek a mérete korlátos, de nem egyformán. A BASIC, Visual Basic és a Pascal nyelvek csak a maximálisan 255 karakter hosszú stringeket támogatják. A C és a Clipper stringjeinek mérete maximálisan 64 kilobájt lehet.

Eltérő láthatósági szabályok
A változók láthatósága nagyon fontos kérdés. A Pascal, a C, a Clipper és a Visual Basic, Javascript, Java, PHP újabb verziói nagyon kifinomult láthatósági rendszerrel rendelkeznek, azonban ezek sem teljesen egyformák. Lényegében azonban az algoritmus-leíró nyelv szabályaihoz nagyon közel állnak

Eltérő feltételvizsgálat.
Az egyes programozási nyelvek a kifejezések kiértékelésekor több féle elvet követnek. Egyes nyelvek a kifejezésben szereplő minden műveletet, függvényt kiértékelnek, míg mások egy kifejezés kiértékelésekor csak addig vizsgálják a kifejezést, amíg az eredmény nem válik egyértelművé.

- Az aritmetikai kifejezéseket általában mindig kiértékelik a nyelvek. Sajnos ilyenkor jönnek elő a nullával való osztás, illetve a lehetetlen függvényargumentumok esetei is. Ezekben az esetekben gyakran a programozónak kell olyan kódot írnia, amely kizárja az ilyen eseteket.

- A logikai kifejezések kiértékelésénél a nyelvek általában beérik azzal, hogy mihelyt biztos végeredmény, a kifejezés többi elemét már nem vizsgálják.
- A kifejezések precedenciája (erősorrend – melyik lesz az először kiértékelt rész-kifejezés egy összetett kifejezésben) is hasonló. Ha egyenlő egy kifejezésben több rész precedenciája, akkor célszerűen zárójelezéssel lehet megváltoztatni a kiértékelési sorrendet. Ha nem vagyunk biztosak egy összetett kifejezés kiértékelési sorrendjével kapcsolatban, akkor célszerű használni a zárójeleket, inkább többet, mint kevesebbet.
- A Pascal eleve elvárja, hogy ha logikai kifejezéseket kapcsolunk össze, akkor rész értékeket zárójelbe kell tenni.
- A kifejezéseket általában a rendszerek balról jobbra értékelik ki. A Visual Basic néha a kód optimalizálása érdekében néha átrendezi a kiértékelési sorrendet. Ez alkalmasint problémákat és eltérő eredményeket okozhat.
- Assemblyben az összetett kifejezések kiértékelésére meg kell írni az arra alkalmas kiértékelő kódot.

Tömbhatárok vizsgálata

A programozási nyelvek másként viselkednek, ha az indexváltozók tömbhatáron túlra mutatnak. A legtöbb futás közbeni hibát generál és a program leáll, vagy hibakezelő ágra fut. A leállás természetesen hiba, amit korrigálni kell. Van olyan eset, hogy az index túlfut a határon, de mégsem okoz a programban valódi hibát, mert a túlmutatás során kapott adat soha sem lesz kiértékelve, és azt az adatot sohasem módosítják. Ebben az esetben a hibaüzenetnek nem célszerű megjelennie.

- A Pascal és a C nyelvben van olyan lehetőség, hogy a rendszer olyankódot generáljon, amely ellenőrzi a határokat, de ezt az ellenőrzést ki is lehessen kapcsolni. A BASIC, Visual Basic, Clipper és a legtöbb egyéb nyelv azonban szigorúan őrzi a tömbök határait.
- Ha bonyolult a kód és nem tudjuk pontosan megállapítani, hogy milyen körülmények között lépi túl a megadott területet a program, akkor a legegyszerűbb megoldás, hogy megnöveljük a tömb méretét annyival, amennyi a túllépés.

Ciklus típusok

A különböző nyelvek nem használják ugyanazokat a ciklus fajtákat. A kódolásnál erre figyelni kell. Az elöltesztelő ciklusokat a (`Ciklus amíg`) a legtöbb nyelv így ismeri, és ugyanúgy használja. A hátulesztelős ciklusoknál a Pascalnak a feltétel kiértékelése pont az ellentéte, mint a többi nyelvnek,

```
Ciklus
      ……
Amíg feltétel igaz
ciklus vége

repeat
 ……
Until nem feltétel
```

A megszámlálásos ciklusnál a legtöbb nyelvben megvan a lépésköz változtatásának a lehetősége, kivéve a Pascalt. Itt a `for` ciklus lépésköze csak +1 vagy –1 lehet.

Rekurzió léte

Nem minden programozási nyelv szereti használni a rekurziót, de a legtöbb nyelven azért meg lehet oldani. A Pascal, a C, a Clipper vagy a Visual Basic, PHP, Javascript, Java közvetlen rekurzióval is elboldogul, de az egyes nyelvek kényesek a rekurzió mélységére. A C, a Pascal esetén lehet állítani a verem méretét, így a rekurzió mélységét megbecsülve könnyen tudjuk alkalmazni azt. A BASIC nyelv nem tartalmaz rekurziót, illetve a változók elhelyezésére nem ad semmiféle módszert.

Az algoritmusok és a programozási nyelvek kapcsolata

Az eddigiek alapján nyilvánvaló, hogy az algoritmusok bár nyelvfüggetlenek, a programkódolásnál nem mindig vihetők át egy az egyben az adott programozási nyelvre, sőt esetenként az adott nyelven előnyösebb más, az eredeti algoritmussal egyenértékű kódot készíteni. Természetesen ez nem mindig szerencsés, figyelembe véve a későbbi javítások, hibakeresések tényét, de a programok hatékonyságának javításánál nagymértékben segíthet, ha a kódolásnál kihasználjuk a nyelv sajátosságait. Ha ilyen eltéréseket viszünk be a kódba, azt megfelelően dokumentálni kell.

A programszöveg strukturáltsága

A legtöbb esetben közömbös, hogy a program szövege egy szöveges fájl vagy több, de ha több fájl, akkor azok gyakran külön fordítási egységet is alkothatnak.

- A Pascal, a C, PHP, Java, Javascript és Clipper esetén alkalmazhatók az úgynevezett include fájlok, amelyek olyan kódot tartalmaznak, amelyet fordítás közben a fordító beemel a szövegbe és együtt fordítja le őket. Más esetekben a fordító több különálló fordítási egységet hoz létre, amely a program fordításának különböző fázisában, de a szerkesztés előtti fázisban szerkesztődik össze.
- A C, és a Clipper nyelvek igénylik úgynevezett header fájlok használatát, amelyben a rendszer előre lefordított programjai találhatók.
- A Windows alatti programozás nagy előnye, hogy a Windows egységes belső programfelépítésének köszönhetően lehet olyan programot alkotni, amelynek bizonyos részei más és más programozási nyelven készülnek, és a kapcsolat közöttük csak a futtatáskor jön létre. Ez az újrafelhasználható programkomponensek használatát nagyban elősegíti.

Memóriaméret

Az egyes nyelvek a fejlődés során különböző stratégiákat alkalmaztak a memóriakezelésre és ezzel kapcsolatban az egyes program- és adatmodulok méretére vonatkozóan. Ezek a stratégiák mindig az optimális sebességű kód megalkotását célozták. Tudvalevő, hogy ha egy programkód mérete nagyobb, mint 64 kb, akkor a memóriacímzésre nem elég két bájtot használni. Ebben az esetben a kód lassabb lesz.

- A Pascal, a BASIC és a Visual Basic azt választotta, hogy az egy függvénybe, eljárásba tartozó programkód mérete nem lépheti túl a 64 kilobájtot. Régebben a Pascal kód nem lehetett nagyobb ennél a méretnél.
- Az adatszegmensnél is hasonló korlátokat vezettek be. Az egy struktúrába tartozó adatok mérete nem lehet nagyobb 64 kilobájtnál.
- A C nyelv memóriamodelleket állított fel. A különböző memóriamodellek a kódra és az adatok méretére adnak felső korlátokat egymástól függetlenül, illetve adnak korlát nélküli memóriakezelést.
  Egy táblázatot adunk meg:

	Adatok mérete < 64 k	Adatok mérete > 64 k
Kódméret < 64 k	Small modell	Compact modell
Kódméret > 64 k	Medium modell	Large modell

  Tiny esetén az adat + kód mérete < 64 K
  Huge esetén minden modul statikus adatai külön 64 k-ban tárolódik, míg a Large modell esetén az összes statikus adat 64 K lehet csak
- A Clipper a C nyelv Large modelljét örökölte (C-ben fejlesztették). A Clipper használja az ún. overlay technikát, amikor a memória egy részét a winchesteren tárolva mindig a megfelelő kód és adatrészletet tölti be a memóriába, ezzel DOS alatt nagyobb programokat tud futtatni, mint a rendelkezésre álló DOS memória.

Alprogramok hívása

Az egyes programozási nyelvek más módot használnak a programok eljárásainak meghívására.

- Ez főleg a paraméterátadást tekintve különbözik. Ez alapvetően a Pascal és a C nyelv paraméterátadására vezethető vissza. A Pascal esetén mindig ugyanannyi és ugyanolyan típusú változónak kell lennie a hívás helyén és a hívott eljárás fejlécében, míg a C esetén nem baj, ha nem egyeznek meg a számok, a megfelelő változó értéke vagy elveszik, vagy ha nem kap értéket, akkor véletlenszerű értékkel töltődik fel. Ha egy átvett paraméter nem kap értéket, akkor csak arra kell vigyázni, hogy a nem meglévő értékét ne használjuk fel.
- A fentiekkel szemben a Pascal paraméterátadása gyorsabb, ezért a Windows alapú programozási nyelvekben különösen, ha más nyelvű fejlesztéseket is akarunk használni, akkor külön deklarálni kell minden nyelven a Pascal rendszerű paraméterátadást. Mivel a C nyelv ANSI féle szabványosítása nem a Pascal paraméterátadási konvenciót használja, ezért a Windows alá írt C nyelvű programok többek között ezért sem vihetők át egyszerűen más rendszerek alá

9 A programok tesztelése, hibakeresés

A programok tesztelésének célja, hogy a program vajon a bemenetekre a specifikáció alapján megfelelő kimenetet szolgáltatja-e. A specifikációnak megfelelő programot **helyes programnak** hívják. A programok tesztelése és a hibakeresés során arra törekszünk, hogy az eredeti specifikációnak minél jobban megfelelő, illetve megfelelő programot állítsunk elő. Olyan tesztmódszereket kell használni és olyan hibakereső eszközöket, amelyek a hibák nagy részét kiszűrik. Néhány tapasztalati ténybe, azonban bele kell nyugodni:

- A program hibáinak száma és súlyossága exponenciálisan nő a mérettel
- A hibajavítás után az összes tesztelést célszerű lefolytatni
- A hibát megszüntető okokat kell megtalálni és kijavítani
- Gyakran egy hiba megszüntetése több másik hiba megjelenését vonja maga után
- A program készítője a legrosszabb tesztelő. A fejlesztőn kívül mással is teszteltetni kell a programot.

A fenti tényeken kívül még egy továbbiról is kell szót ejteni. Nagyobb méretű programok esetén 100%-osan hibátlan programról nem lehet beszélni. A világon a legszéleskörűbb tesztelésnek a Windows95-öt vetették alá. A program kiadása többek között ezért csúszott majdnem egy évet. Mégsem lett hiba nélküli. A tapasztalat azt mutatja, hogy ha a kód 95 – 99%-a hibátlannak bizonyul, akkor a programot késznek kell nyilvánítani. A maradék hibák kijavításának költsége ugyanis minden tekintetben olyan nagy, hogy nem éri meg az összes hibát kijavítani.

A fentiek alapján a tesztelés kritériumai

- A jó tesztelés nagy valószínűséggel felfedi a hibákat
- A jó tesztelési eljárásoknak megismételhetőknek kell lenni
- Érvényes és érvénytelen adatokra is kell tesztelni
- Minden tesztesetet maximálisan ki kell használni, azaz a legtöbb hibát fel kell deríteni
- Fel kell tenni a kérdést, hogy **miért nem azt teszi** a program, amit kellene volna és **miért azt teszi**, amit nem kellett volna.

9.1 Statikus tesztelési módszerek

A statikus tesztelési módszerek a programkód vizsgálatán alapulnak. Ekkor nem futtatjuk a programot. Bár sokan azt gondolják, hogy a ami hardvereknél nincs nagy jelentősége ezeknek a módszereknek, hiszen a fordítók és interpreterek olyan jók, hogy mindenre választ adnak, minden hibát megtalálnak. Sajnos gyakori, hogy a fordítók nem találnak meg minden hibát, amint az alábbiakban látszik is. További probléma, hogy egyes rendszerek és fejlesztések esetén a fordítás folyamata olyan hosszú, hogy mindenképpen már a kódot is tesztelni kell. (Gyakori, hogy a fejlesztők olyan gépen dolgoznak – pénzhiány miatt, amelyek az adott fejlesztő rendszer követelményeit éppen csak kielégítik. Ilyenkor egy fordítás 5 –15 perc, de akár több óra hosszú is lehet.)

Kódellenőrzés
A legegyszerűbb lehetőség. Kinyomtatjuk, vagy a képernyőn átnézzük a kódot, miután begépeltük. Célszerű olyan editort használni, amely kiemeli az adott nyelv kulcsszavait, esetleg színnel vagy más módon elkülöníti az adatokat, az értékadó utasításokat. Ha lehet az program bevitelekor használni kell a strukturált írásmódot, ha akkor nem tettük meg, akkor utólag javítani kell a kódon. Figyelni kell, hogy az egyes programozási egységek kezdet és vége (begin … End, {…}, Procedure …. Return, stb…) megvan-e?

Szintaktikai ellenőrzés
A legtöbb fejlesztő eszköz ma már szintaktikailag ellenőrzi a program kódját és a megfelelő sorban ki is írja a hibaüzeneteket. Az interpreteres nyelvek gyakran már a programsor bevitelekor elvégzik az ellenőrzést, míg a compileres nyelvek csak a fordítás során.

Szemantikai ellenőrzés
Lehetséges, hogy egy szintaktikailag helyes program valójában nem azt teszi, ami a dolga lenne. Az interpreteres rendszerek szemantikailag nem ellenőrizhetik a bevitt kódot, hiszen általában nincsen a teljes rendszerre rálátásuk. Ekkor csak a programozó tudja végiggondolni, hogy programja valóban logikailag megfelelő, az alkalmazott algoritmusok valóban a kellő végeredményt adják, és a kódolás megfelel az algoritmusnak.

A compileres rendszerek esetén előfordul, hogy a fordító figyelmeztet bizonyos utasítások szemantikai problémáira. Gyakran találnak az ilyen rendszerek felesleges változókat, olyan kódrészleteket, amelyek sohasem futnak le, mindig biztosan azonos értéket felvevő változókat, stb. Az ilyen rendszerek használata sem teszi nélkülözhetővé a kézzel történő kódellenőrzést. Azok a compileres rendszerek, amelyek kódoptimalizálást végeznek, gyakran olyan assembly kódot hoznak létre az optimalizálás során, amely logikailag nem felelnek meg az algoritmusnak. Ekkor ki kell kapcsolni az optimalizálást.

Inicializálatlan változók
Meg kell keresni a kódban az inicializálatlan változókat, és kezdőértéket kell nekik adni. Sok váratlan hiba ilyen okokra vezethető vissza. Egyes rendszerek csak lefoglalják a változók számára a memóriaterületet, de nem adnak automatikusan nekik értéket. Ilyenkor az adott területen lévő véletlenszerű memóriaszemét lesz a kezdőérték.

Felesleges utasítások kiszűrése
Gyakran kódoláskor az algoritmusnak megfelelő kódot írunk, holott az adott nyelv ugyanazt a funkciót esetleg gyorsabban is meg tudja oldani. Az is előfordul, hogy például egy for ciklus ciklusváltozójának egy eljárásba való belépéskor külön értéket adunk. Ezekben az esetekben felesleges utasításokat helyezünk el, amelyek biztosan lassítják a programunkat.

Keresztreferencia táblázat
Ha a programunkban lévő változók értékeinek változását nem tudjuk követni, akkor célszerű keresztreferencia táblázatot készíteni. Erre a legtöbb fordító képes. Ez egy olyan táblázat, amely felsorolja, hogy az adott változó hol kap értéket, illetve hol történik hivatkozás rá a program során. Ennek alapján megállapíthatjuk, hogy mely változókat használjuk a leggyakrabban. Az interpreteres nyelvek egy részénél – BASIC – nem mindegy, hogy milyen sorrendben definiáljuk a változókat, a korábban definiált változókat a rendszer gyorsabban éri el.

Típuskeveredés
Egyes interpreteres nyelvek a bevitelkor nem ellenőrzik, hogy az értékadó utasítások két oldalán ugyanolyan típusú értékek szerepelnek-e. Más rendszerek, mint például a Clipper fordítási időben nem is tudja elvileg sem meghatározni egy változó típusát. Ezeknek a rendszereknek általában ez erősségük, hiszen futáskor dőlhet el, hogy az adott változó milyen értéket kap, „szabadabb” kódot lehet létrehozni, de egyúttal a tesztelésnél és hibakeresésnél hátrányuk, hiszen csak futás közben derülhetnek ki az esetleges problémák.

9.2 Dinamikus tesztelési módszerek

A programok hibáinak egy részét a statikus tesztelési módszerekkel ki lehet szűrni, de vannak olyan helyzetek, hogy csak a futás közbeni ellenőrzés segít. Hogy egy-egy teszt minél több tulajdonságot áruljon el a programról az alábbi módszereket lehet alkalmazni:

9.2.1 Fehér doboz módszerek

Az utasítások lefedésének elve
A program minden utasítását legalább egyszer végre kell hajtani. (Sajnos tapasztaltam már olyan esetet, amikor egy fejlesztő eszköz nem volt hibamentes és a leírások szerint jó program nem azt hajtotta végre, amit kellett.)

Döntések lefedésének elve
A programban lévő döntések minden következményét végig kell próbálni. A döntéseket igaz és hamis esetben is végig kell próbálni.

A feltételek lefedésének elve
A programban lévő feltételes elágazásokat minden feltételre ki kell próbálni, illetve az logikai összekötő műveleteket minden lehetséges helyzetre ki kell próbálni.

9.2.2 Fekete doboz módszerek

Ekvivalencia osztályok készítése
A lehetséges bemenő adatokat oly módon kell csoportosítani, hogy milyen kimenő adatot várunk tőlük. Ezeket ekvivalencia osztályoknak hívjuk. Nyilván ha egy ekvivalencia osztály egy elemére a helyes kimenetet kapjuk,

akkor az osztály többi elemére is helyes eredményt kell kapnunk. Minden ekvivalencia osztályra tesztelni kell a programot.

Határeset analízis
Ha a lehetséges bemenő adatok ekvivalencia osztályait helyesen is állapítottuk meg, és úgy találjuk, hogy az osztályokra megfelelő választ ad a program, még mindig meg kell vizsgálni, hogy az ekvivalencia osztályok határeseteit hogyan kezeli le a program. Gyakran az ilyen helyzetben adott hibás eredmény helytelen algoritmusra, gondolatmenetre vagy túlzott egyszerűsítésre vezethető vissza

Stressz teszt
A programokat biztosan rossz bemenő adatokkal is tesztelni kell. A programok fejlesztése során a fejlesztő általában feltételezi, hogy a felhasználó csak helyes dolgokat művel, pedig ez nem így van. A felhasználó sokkal gyakrabban téved, hibázik, mint azt a legtöbb fejlesztő képzelné.

9.2.3 Speciális tesztek

Hatékonysági tesztek
A programok tesztelésének utolsó fázisa, annak megállapítása, hogy milyen hatékony a program, illetve menynyire jól felhasználható. A tesztelések során meg kell állapítani, hogy a program a meghatározott hardveren egyáltalán fut-e, megfelelő sebességgel fut-e. Ha nem fut, vagy a sebessége nem megfelelő, akkor meg kell keresni azokat az okokat, amelyek a megfelelő futást megakadályozzák, és annak megfelelően kell módosítani a programot, akár az algoritmusok szintjére is visszamenve.

Biztonsági teszt
A programoknak stabilaknak kellene lenniük, nem szabadna előre látható okok miatt lefagyniuk. Ezekre valók a biztonsági tesztek. A programot nagyon kevés és túl sok adattal is tesztelni kell, hogy nincsenek-e memóriakezelési problémái, szándékosan a legelfogadhatatlanabb adatokat kell bevinni, szabálytalanul leállítani, hogy kiderüljön, vajon a program képes-e helyreállítani az ilyen esetekben is a működését.

Például adatbázis-kezelő rendszerek esetén fontos, hogy szabálytalan leállás után az adatok ne sérüljenek meg, és az újraindítás után folytatható legyen a munka)

9.3 Hibakeresési módszerek

A következőkben néhány általános jellegű módszert adunk a hibák megkeresésére.

Indukciós módszer
Az indukciós módszer lényege az, hogy keresünk egy olyan bemeneti adathalmazt, amire a program jól működik, majd ennek az adathalmaznak megkeressük azt a határát, amelyre szerintünk szintén működik a program. A kiterjesztésen belül még próbálkozunk. Ha mindent rendben találunk, akkor tovább tágítjuk a bemenő adatok körét és továbbra is vizsgáljuk, hogy az adatokra a program helyesen válaszol-e.

Ha bármilyen ponton úgy találjuk, hogy a program a bemeneti adatokra hibás választ ad, akkor megpróbáljuk leszűkíteni a bemeneti adatoknak azt a halmazát, amelyre hibás választ ad a program.

Egy példán keresztül szeretném illusztrálni az indukciós módszert. Ha egy program bizonyos bemeneti paraméterek hatására jól működik, pl 100, 101, 102 adatokra jól működik, akkor feltételezzük, hogy legalább ezerig jól fog működni. Megpróbáljuk ebben a nagyságrendben. Tegyük fel, hogy nincs probléma.

Ekkor feltesszük, hogy a program az 1000-es nagyságrendbeli adatokkal is jól működik. Ha például 1000, 1001, 1002-re nem jól működik a program, akkor keresünk még ugyanabban a nagyságrendben, új adatokat, például 1099-et, amelyre feltevésünk szerint jól kellene működnie. Ha arra az adatra jól működik, akkor a hiba az ezer környéki tesztadatokra jellemző, ha hibázik, akkor nyilvánvalóan a nagyobb adatokra is hibásan reagál.

Dedukciós módszer
Ha a programunk bizonyos bemeneti adatokra hibásan, más adatokra jól reagál, akkor megvizsgáljuk, hogy a bemeneti adatokban mi az a közös tulajdonság, ami szerint a végeredmény osztályozódik. Ha találtunk ilyen tulajdonságot, akkor jósolunk, és a jóslat alapján új tesztadatokat állítunk elő, majd teszteljük a programot. Ha a tesztadatokra a várt eredmények születtek, akkor feltételezésünk jó volt, a hiba valóban a bemenő adatok közös tulajdonságai alapján kereshető meg. Ha nem a feltételezéseknek megfelelő eredmények születnek, akkor

nem sikerült megtalálni az eredeti bemeneti adatokban a csoportképző faktort, tehát új tulajdonságot kell keresni rajtuk. Például:

Egy program a 3, 5, 7, 11 adatokra jó, a 10, 12 adatokra hibás eredményt ad. Feltételezzük, hogy a jó bemeneti adatokban a közös az, hogy a számok prímszámok vagy páratlan számok. Kipróbálunk olyan értékeket, amelyek prímek, például a 17 és kipróbáljuk a 2-t is. Az egyik feltételezésünket ki fogja ejteni a két próba.

Visszalépéses módszer
Ennek a módszernek az a lényege, hogy a program végeredményét vizsgálom, hasonlítom össze a specifikációban megjelöltekkel és a programban visszafelé lépkedve keresem meg a hibás lépést.

Tesztesetek felhasználása
A program specifikációja alapján teljes körű, körültekintően összeállított teszthalmazt készítünk és a tesztesetek felhasználásával, valamint a kapott végeredmény felhasználásával következtetünk a hiba okára.

9.4 Hibakeresési eszközök

A modern fejlesztő rendszerek majd mindegyike rendelkezik már valamilyen hibakereső szolgáltatással. A régebbi rendszerek a program futása közben beálló hibára gyakran egy memóriacím, esetleg a stack, kiírásával reagáltak és általában az operációs rendszer nem túl széleskörű szolgáltatásait használták erre a célra. Később megjelentek az olyan fejlesztő rendszerek, amelyek a hiba megjelenésekor kiírták, hogy a forrásszöveg melyik sorában, milyen jellegű hiba történt, akár egy hibakóddal, akár szövegesen is.

A mai integrált fejlesztőeszközök némelyike képes arra, hogy a hibánál kijelezze a hibás utasítást, adjon tippet a hiba okára, írja ki a verem állapotát, a változók pillanatnyi értékét. A továbbiakban átnézzük, hogy melyek azok az eszközök, amelyeket a fejlesztő felhasználhat hibakeresésre.

Nyomkövetés
A program végrehajtása során a nyomkövető eszköz kiírja, hogy melyik utasításnál, melyik sorban fut éppen a program. A Borland Pascal vagy Borland C/C++ esetén a program a képernyő egyik ablakában megjeleníti a program kimenetét, a másik ablakában pedig megjeleníti a forrásszöveget és jelzi, hogy éppen hol tart a program. A legtöbb komolyabb hibakereső rendszer képes arra, hogy az így futó programot bármelyik pillanatban leállítsa.

Debugging
A **debugger** eredetileg a futtatható programok gépi kódú visszafejtésére alkalmas eszközt és a **debugging** magát a visszafejtés folyamatát jelentette. Manapság már nem mindig az assembly nyelvű kód vizsgálata a célszerű. Vannak olyan eszközök, amelyek alkalmasak arra, hogy ha egy program fejlesztési nyelvét ismerjük, akkor képes legyen visszaállítani az eredeti forráskódot több-kevesebb sikerrel. Ilyen eszköz létezik assembly, Pascal, C, Clipper nyelvekre – tudomásom szerint.

Töréspontok elhelyezése
Ha egy nagyobb program működését vizsgáljuk, a nyomkövetés alkalmazása esetleg annyira lelassíthatja a program működését, hogy elviselhető időn belül nem jutunk el a vizsgálat alá vont helyre. Ilyenkor segít az, hogy töréspontokat helyezünk el a programban. Ezeken a pontokon a program megáll és megvizsgálható az állapota. A hibakereső rendszer tulajdonságaitól függően azután a töréspontoktól kezdve a program folytatható vagy félbeszakad. A nagyobb tudású rendszerek a képernyőtartalmat elmentik a töréspont előtt, majd visszaállítják a töréspont után.

Részeredmények, állapot kiíratása
Gyakori hibakeresési lehetőség. A program bizonyos vizsgált részeire kiíró utasításokat teszünk, amelyek tájékoztatnak a program, vagy csak bizonyos változók pillanatnyi állapotáról. Az állapot kiíratása során célszerű megvizsgálni az adott programozási egységben szereplő változók értékét, az adott eljárásnak átadott paraméterek és a visszaadott paraméterek értékét. A kiíratás összekapcsolható töréspontok elhelyezésével is.

Lépésenkénti végrehajtás
A töréspontok után a programot egy ideig lépésenként is végrehajtathatjuk. Ilyenkor a programkódban lépünk egyet majd figyeljük a program új állapotát. Figyelünk a képernyőtartalomra, a változók és paraméterek értékére, a fájlok állapotára. A lépésenkénti végrehajtás során gyakran eljárást vagy függvényt hívunk meg a forráskódban. Ilyenkor általában két választásunk van. Vagy a meghívott eljárást, függvényt is lépésenként hajtatjuk végre, vagy a kérdéses részt a program teljes sebességgel végzi el. Nyilván, ha biztosak vagyunk az

adott függvény, vagy eljárás hibátlanságában, akkor az utóbbit választjuk. A lépésenkénti végrehajtást a program teljes sebességű további futtatásával is lehet általában kombinálni.

Tervezés fázisába való visszalépés
Ha végképpen nem találjuk meg a hibákat az eddigi „nagyágyúkkal”, akkor bizony vissza kell lépni a tervezés fázisába és vagy algoritmus szinten, vagy kódolási szinten újra kell gondolni a hibás részt, hátha valami elemi szemantikai hiba okozza a hibás működést.

Ciklusok befejeződésének vizsgálata
Sok hiba felderítésének módja, a ciklusok befejeződésének vizsgálata. A programkódok legalább 30% ciklusokból áll, és a ciklusokba helytelen kilépési feltételei, vagy a megszámlálásos ciklusoknál a ciklusváltozó kezdő és végértékének helytelen meghatározása az oka az olyan fajta hibáknak, amelyek nem minden esetben fordulnak elő. Ezt a vizsgálatot a határeset analízissel is célszerű összekötni.

Változók értékének menet közbeni kiiratása

Állapotdefiníció
Egy gyakran használható módszer. A programozó a programkód vagy algoritmus alapján meghatározza, hogy bizonyos kezdőértékek alapján a programnak egy adott helyen milyen állapotba kell kerülnie. Ha eltérést tapasztal a megkívánt és a tapasztalt állapot között, akkor a hiba okát nyilván a kettő között kell keresni és nyilván a különbséget kell tüzetesen megvizsgálni.

9.5 A tesztelők személye, szervezett tesztek

Már korábban is említettük, de most részletesebben szólunk a tesztelők személyéről. Köztudott, hogy egy program fejlesztője egy idő múlva a triviális hibákat sem veszi észre. Ennek könnyen belátható okai vannak. Egy probléma megoldásának során a fejlesztő gyakran elgondol bizonyos algoritmusokat, amelyeknek nem minden részletét tisztázza le, majd ha a megvalósítás során az algoritmus hibái nem lesznek nyilvánvalóak, akkor gondolatban az algoritmus kérdéses részét jónak fogadja el. Megkönnyebbül attól a gondolattól, hogy az algoritmusnak az a része jó. Ettől az állapottól csak keservesen tud megszabadulni és csak úgy, ha újra elemzi, és újra megalkotja a kérdéses részt. Sajnos az embernek a hiba felismeréséig, megtalálásáig nehéz eljutnia.

A programozás során szinte elkerülhetetlen, hogy a programozó kisebb – nagyobb mértékben ne térjen el az eredeti elgondolásoktól. Ekkor az algoritmus és a kód már nem fog teljesen megfelelni egymásnak. Ez is okozhat olyan hibákat, amelyeket később a fejlesztő nem vesz észre.

A fejlesztés során gyakori, hogy a részletkérdésekbe annyira beleássa magát a fejlesztő, hogy nagyobb összefüggéseket nem fedez fel.

Milyen módon lehet kikerülni ezeket a csapdákat, hogyan lehet a tesztelést, a hibakeresést hatékonyabbá tenni.

- A tesztek és a hibakeresés során nem elég a képernyőn vizsgálni a forrásszöveget, ki is kell nyomtatni. A hagyományos papír alapú vizsgálódás sok olyan összefüggést feltár, ami a képernyőn nem válik nyilvánvalóvá.
- Mindig kell olyan tesztelő személyeket találni, akik az adott fejlesztési fázisban nem vettek részt, az adott kódrészletet nem ismerik. A fejlesztőnek hagyni kell a másik személyt, hogy amennyire lehet, egyedül értse meg az adott részletet, és egyedül keresse meg a kérdéses hibát.
- A tesztelő személyek között kell lennie olyannak, aki számítástechnikai, fejlesztői szempontból vizsgálja a programot és kell lennie olyannak is, aki a felhasználó szakmai szempontjából nézi a programot. A fejlesztő és a majdani felhasználó nem egyforma módon közelíti meg az adott problémát, ennek megfelelően nem ugyanazok a dolgok fontosak egyik számára, mint a másik számára.
- A legjobb tesztelés az éles adatokon, éles helyzetben történő tesztelés. Egy program elkészülte után az igazi felhasználók visszajelzései lesznek igazán a felhasználói teszt szempontjából lényegesek.

10 Hatékonyságvizsgálat, optimalizálás

A programok hatékonyságát több mérőszám tudja csak leírni. A program hatékonyságának legfontosabb kritériumai,

- A program futásának sebessége,
- A program mérete
- A program bonyolultsága.
- Manapság a szoftverek széleskörű elterjedtségének köszönhetően a programok hatékonyságához még hozzájön a programok felhasználhatóságának, barátságosságának szempontja is.

E fenti kritériumok közül sajnos az alábbi összefüggések állnak fent:

- A programok futásának sebessége és a program bonyolultsága általában egymásnak ellentmondó fogalmak.
- A program mérete arányos a program bonyolultságával.
- A program mérete fordítottan arányos a program futásának sebességével.

10.1 Rendszerek hatékonyságának megállapítása

A programok hatékonyságának megállapításánál a legjobban mérhető fogalom a futási idő megállapítása. A futási idő megállapításánál mindig több mérést kell végezni, és figyelni kell az egyes futási időket. Csak így lehet megbízhatóan kiszűrni a számítógépen futó alkalmazások, az operációs rendszer és a hardver egyéb paraméterének változásait. Azt is célszerű megállapítani, hogy különböző konfigurációkon hogyan működik a program, melyek azok a paraméterek, amelyek a program maximális hatékonyságát, sebességét biztosítják.

A mérések során fel kell jegyezni az **átlagos**, a **maximális** és a **minimális** futási időt. Természetesen az átlagos idő lesz a legjellemzőbb a program sebességére.

10.1.1 Egzakt módszerek

A program sebességét úgy tudjuk megállapítani, hogy a mérendő rész elején és végén megmérjük a gép belső órájának időpontját, majd a két időt kivonjuk egymásból.

Vannak olyan szoftverek, amelyek segítségével a futási időket egzakt módon mérni lehet. Ezek a **profiler**-nek nevezett programok. Segítségükkel a program optimális futása szempontjából sok különböző mérést lehet elvégezni. A Borland Pascal része egy ilyen profiler program.

Speciális operációs rendszer szolgáltatások hívásából mégis megállapítható a program futás közbeni mérete.

Egy program barátságossága nem mérhető egzakt módon.

10.1.2 Kézi módszerek

A programok sebességét sokszor a forrásszöveg hiányában nem tudjuk mérni egzaktul. Ekkor stopperrel egy batch állományból indítva lehet lefuttatni a programot és mérni a sebességet. Ha a program futása túl gyors, akkor a batch fájlba egy olyan ciklust kell beiktatni, amely kellően sokszor fut le ahhoz, hogy a program sebessége mérhető legyen. Ha megmértük a program sebességét, akkor meg kell mérni a batch file futásának sebességét is, így kaphatjuk meg a program futásának sebességét.

A program memóriában elfoglalt méretét gyakran nem tudjuk megállapítani, de a háttértáron elfoglalt méretből, az adatállományok méretéből, esetleg következtethetünk rá.

A program bonyolultsága nem egzakt fogalom, ezért mérése nem is lehet egzakt. Viszonyítani kell a feldolgozandó adatok bonyolultságát a programkód összetettségéhez. Ha a kettő között kiugró aránytalanságot találunk, akkor a kód túl bonyolult.

Egy program barátságosságának mérése csak több, különböző képzettségű tesztelő tesztelése után állapítható meg. Célszerű a programot egymástól független személyekkel teszteltetni, és előre leírni azokat a szempontokat, amelyeket pontozniuk kell egy előre megadott skála szerint. Előre el kell döntenünk, hogy az egyes kategóriákat a végső értékelésnél milyen súllyal vesszük figyelembe. A több személy az egyéni ízlésbeli különbségeket, míg a különböző képzettség a képzettségből fakadó hibákon való átsiklást küszöböli ki. Ha van hasonló feladatokat ellátó másik program, akkor célszerű a két program összehasonlító tesztelése is.

10.2 Globális optimalizálás

A tesztelések során kiderülnek a program gyenge pontjai, elsősorban a méret és a sebességbeli hiányosságok. A program sebességére és méretére több módszer lehetséges.

A programban legfontosabb, hogy a program egésze működjön optimálisan, ezért elsősorban globálisan kell megtalálni azokat a tényezőket, amikkel javítani lehet a program sebességét.

- Meg kell érteni a program során alkalmazott algoritmus és annak megfelelően, esetleg gyorsabbra cserélni azt.
- Csökkentsük a ciklusok végrehajtási számát (Például, ha N egész szám közül keressük a prímeket, akkor N/2 helyett elegendő csak négyzetgyök N-ig keresni a prímeket)
- A ciklusok végrehajtási idejét csökkentsük
- Fel kell használni a feladat speciális tulajdonságait, (pl. ha a keresés rendezett halmazon történik, akkor a gyorskeresés $\log_2 N$ lépés alatt zajlik le átlagosan, míg lineáris keresésnél csak N/2 lépés alatt.
- Matematikai elvek, ismeretek, definíciók használata.
- Kisebb méretű, gyorsabban feldolgozható adattípusok használata. Ne használjunk különböző típusokat egy kifejezésben
- Függvények ismételt kiértékelése helyett – ha nem változik a függvény értéke menet közben – temporary változóban tároljuk a függvény értékét.
- A feltételeinket egyszerűsítsük
- Az adattípusokat gondosan válogassuk meg, a helykihasználás és a feldolgozás sebességének figyelembevételével.
- Bekapcsoljuk a fordítókba beépített kódoptimalizáló opciókat. Ennek az a veszélye, hogy bizonyos esetekben a rendszer nem azt a gépi kódú programot hozza létre, amit mi elterveztünk.

10.3 Lokális optimalizálás

Ha már kifogytunk a globális optimalizálás ötleteiből, akkor jó szolgálatot tehet, ha megvizsgáljuk, melyek azok a programok, amelyekben a legtöbbet tartózkodik a program. Ezeket az eljárásokat kell gyorsabbá tenni, ekkor az egész program futása felgyorsul. Milyen módon?

- Átvizsgáljuk az algoritmust és megnézzük, hogy nincsen-e gyorsabb, helyettesítő algoritmus.
- A speciális esetek kiszűrése. Az eljárásban konkrét értékadással vagy egyéb módon kizárjuk a speciális eseteket, a szélső értékeket külön kezeljük le, nem próbálunk általános megoldást adni ezekre az esetekre.
- Hatékonyabb programkódolási technikát alkalmazunk. A tapasztalat azt mutatja, hogy a hatékonyabb programkódok egyúttal nehezebben követhetők is. Elsősorban C nyelven van annak jelentősége, hogy milyen kódot ír le a programozó, mivel a leírt kódtól nagymértékben függ a gépi kódú végeredmény.
- Egyes fordítók a fordítás során keletkezett kódot optimalizálják, a programozó által leírt kódhoz képest gyorsabb assembly kódot hoznak létre. Sajnos egyes esetekben a kódoptimalizálás váratlan eredményeket is hozhat, ezért kellő körültekintéssel kell alkalmazni.
- Rendszerközeli betéteket alkalmazunk (Assembly – csínján kell bánni vele!)
- Trükköket alkalmazunk, amely kihasználja a processzor, az operációs rendszer vagy a nyelv egyes speciális tulajdonságait. (Vigyázni kell vele, könnyen érthetetlenné és visszafejthetetlenné tehetjük a programunkat)
- Kisebb méretű, gyorsabban feldolgozható adattípusok használata. Megjegyzendő, hogy a PC-ken az Integer adatokat dolgozza fel leggyorsabban az operációs rendszer.
- Ha már leteszteltük a programot, akkor kikapcsoljuk a fordítóprogram hibaellenőrző kapcsolóját, ami gyorsabb és kisebb programkódot hoz létre.

10.4 Hatékonyság transzformációk

Legyenek az alábbi definiciók:
F – Feltétel
U1, U2, U3, ... utasítások

Ha F akkor
  U1;U2
különben
  U1;U3
	U1
	Ha F akkor U2
	különben U3
	
	Ha az F-et nem módosítja az U1.
	
	Ha F akkor U2
	különben U3
	U1
	Ha az utasítások sorrendje nem számít.

Ha F akkor U1;U3 különben U2;U3	Ha F akkor U1 különben U2 U3 Mindig megcsinálhatjuk.
Ha F1 és F2 akkor U	Ha F1 akkor Ha F2 akkor U Akkor, ha F1 rövid, F2 bonyolult. (A Pascalban beállítható, hogy ne értékelje ki F2-t, ha F1 hamis.)
Ha F1 akkor U1 különben Ha F2 akkor U2	Ha F1 akkor U1 különben U2 Ha F1= NOT F2.
Ha F akkor U1 Ha F akkor U2	Ha F akkor U1;U2 Ha U1-nek nincs hatása F-re.

Ciklus amíg CF
  U1;U3
Cvége
Ciklus amíg CF
  U2;U3
Cvége
	Ciklus amíg CF
	  U1;U2;U3
	Cvége
	Ha U1 és U2 függetlenek egymástól, és nincsenek hatással CF-re, legfeljebb U3-ra.

U
Ciklus amíg Cf
  U
Cvége
	Ciklus
	  U
	amíg CF
	
	Mindig megcsinálhatjuk, de van, hogy nem célszerű.

Ciklus amíg CF
  Ha F akkor U
Cvége
	Ha F akkor
	  Ciklus amíg CF
	  U
	  Cvége
	
	Ha F nem változik a ciklus futása alatt.

Ciklus amíg CF
  U1;U2
Cvége
	U1
	Ciklus amíg CF
	  U2
	Cvége
	
	Ha az U1 végrehajtása független a ciklustól és saját maga korábbi végrehajtásától.

11 Dokumentáció

11.1 A dokumentáció formája

Fel lehet tenni a kérdést, hogy a dokumentáció milyen formában jelenjen meg, papíron vagy elektronikus úton. Mind a két megoldásnak vannak előnyei és hátrányai.

A papíron lévő dokumentáció áttekinthetőbb, jobban lehet benne böngészni, ismerősebb a használata egy kezdő felhasználónak vagy a rendszerrel most ismerkedőnek, de bizonyos speciális információkat nehezebb megtalálni benne, még akkor is, ha tartalomjegyzék, és tárgymutató van benne. Nem utolsósorban egy több száz oldalas könyv papír- és nyomtatási költsége tetemes lehet.

Az elektronikus dokumentációban könnyebb szavakra, kifejezésekre keresni, az elfoglalt tárterület minimális, tartalomjegyzék tárgymutató alapján lehet keresni benne, és papírköltség sincsen, ugyanakkor a problémakörrel frissen ismerkedők nehezebben tudják megtenni az első lépéseket, mivel nincs olyan áttekintésük a dolgokról.

A célszerűség azt diktálja, hogy általában legyen elektronikus és papír alapú dokumentáció is egy rendszerhez, figyelve arra, hogy a rendszer felhasználásának különböző fázisaiban más és más a célszerű formátum. Ha a dokumentáció terjedelme nem túl nagy, akkor mindenképpen mind a két formában célszerű közreadni, ha a méret indokolja, akkor pedig meg kell teremteni annak a lehetőségét, hogy az elektronikus dokumentációt megfelelő formában ki lehessen nyomtatni.

A „nagy” szoftveres cégek, mint a Microsoft, Novell, Lotus stb… a felhasználói és a fejlesztői dokumentációkat manapság csak elektronikus úton mellékelik a szoftverez, de olyan formában, hogy azt bárki kinyomtathatja. Ezekez néha külön is meg lehet vásárolni, súlyos tízezrekért.

11.1.1 Az elektronikus dokumentáció szokásos eszközei

Szövegszerkesztő

Bármilyen dokumentáció is készül, egy szövegszerkesztő alapvetőt alapvetően használni kell. Néha célszerű egyszerű ASCII editort használni, de figyelni kell a kódlapokra. DOS-os 437-es kódlapot használó editor a Windowsban nehezen olvasható fájlokat eredményez, míg 852-es kódlapot használva a 437-es kódlapban nem lehet olvasni az állományokat és ekkor még nyomtatásról nem is beszéltünk.

Norton Guide

Régebben programozók körében elterjedt volt a DOS-os Norton Guide formátumú dokumentáció. Ez egy speciális formátum, a fájl kiterjesztése NG. A dokumentációt egy menürendszeren keresztül lehetett használni. A szövegekben nem, de a szövegek alján lehettek hivatkozások a szöveg más fejezeteire. A Norton Commander helpje is Norton Guide formátumban íródott. A használatához el kell indítani az NG.EXE memóriarezidens programot és egy speciális, beállítható billentyűkombináció segítségével elő lehet hívni a háttérből.

Az ilyen formátumú információt egy text formába az NGR.EXE segítségével lehet visszafordítani. A módosított szöveget NGC.EXE segítségével egy köztes állapotba kell fordítani, majd az NGL.EXE előállítja a több köztes állapotú fájlból a végleges állományt. Ezt a fájltípust nem lehet kinyomtatni olvasható formában, csak a szöveget. Ma már nem használatos a DOS-os alkalmazások visszaszorulása miatt.

Windows Help rendszere

A Windows-os help fájlok előállításához kell egy olyan tetszőleges szövegszerkesztő, amelyik nem helyez el speciális vezérlőkaraktereket a szövegben, vagy ASCII TXT formátumban kell elmenteni a szöveget. A szintaktikusan megfelelő Help állományt, utána a Windows 3.1 esetén a HC31.EXE programmal lehet lefordítani. Win95 esetén a HCRTF.EXE, illetve a HCW.EXE programok fordítják le a forrásszöveget Win95-ös HELP állománnyá. A HLP fájlokat ki lehet nyomtatni a WINHELP.EXE program segítségével.

Adobe Acrobat formátum

Az Adobe cég régóta piacon van az Acrobat formátummal. Ezek a fájlok PDF kiterjesztésűek. A fájlok szerkesztéséhez az Acrobat Composer-t lehet használni, vagy más eszközöket. A PDF állomnyok előállításának elterjedt módszere, hogy a rendszerbe egy PDF nyomtatót telepítünk, ami a kimenetet PDF állományként generálja. Az Acrobat Reader (ACROREAD.EXE) program olvassa, jeleníti meg az ilyen fájlokat. A fájlokat

kinyomtatva, a papíron ugyanúgy jelenik meg a tartalom, mint a képernyőn. Az Acrobat Reader freeware-ként is hozzáférhető.

HTML formátum
Az Internet elterjedésével egyre gyakrabban adják a programok dokumentációjának egy részét HTML formátumban. Az ilyen fájlok szerkesztéséhez végső soron egy tetszőleges editor, no meg egy jó HTML kézikönyv is elég. Ha komolyabb oldalakat akarunk tervezni, akkor célszerű lehet valamilyen erre a témára kihegyezett programot használni, például a Netscape Composer, Microsoft Frontpage, Adobe PageMill, Macromedia Dreamweaver, stb. programokat vagy más az internetről letölthető free programokat is. A HTML oldalak megjelenítéséhez csak egy böngésző program kell, ami létezik akár DOS-os környezetben is.

11.1.2 A papír alapú dokumentáció

Nyilván a papír alapú dokumentáció elkészítése előtt létrejön egy elektronikus formátum is. Ehhez mindenképpen a korábbiakban felvázolt eszközök kellenek. A kinyomtatás során azonban vigyázni kell, hogy a nyomtatók alapértelmezésének beállított A/4-es (amerikai nyomtatóknál a Letter) papírmérettel kinyomtatott dokumentáció nem használható jól, túlságosan nagy. Elterjedt formátum a B/5-ös, amely az A/5-nél és az A/4 között van. Nem tartja meg az aranymetszés szabályait, annál kicsit szélesebb.

Az is probléma, hogy a nyomtatók általában csak a papír egyik oldalára nyomtatnak, ugyanakkor a könyvek lapjain a papír mind két oldalára kerül szöveg. Házi körülmények között ezek a problémák igen csak nehézkessé teszik a dokumentációk kérdését.

A dokumentációban lévő ábrákat, fényképeket megfelelő minőségben csakis lézernyomtatók képesek kinyomtatni, de csak fekete-fehérben. A színes festékszórós elven működő nyomtatók csak speciális papír felhasználásával produkálnak megfelelő minőséget. Az ilyen papír drága, esetenként egy oldal kinyomtatása akár 100 Ft költséget is jelenthet.

A dokumentáció sokszorosítása már nyomdai feladat. Természetesen fénymásolókkal is lehet sokszorosítani, de a fénymásolt példányoknál megint előjön a B/5 –A/4 -es kérdés, a színes fénymásolás drága, két oldalas fénymásolat sokkal körülményesebb, továbbá a fénymásoló minősége is sokat számít.

11.2 Felhasználói dokumentáció

A felhasználó a program használata során több különböző fázison megy keresztül. A program telepítése után csak ismerkedik a számára kialakított rendszerrel, ekkor sok alapvető ismeretre van szüksége. A használat során egyre több aspektusát ismeri a rendszernek. Ekkor már inkább rövid, emlékeztető segítségre van csak szüksége.

A felhasználói dokumentáció készítésének célja, hogy a programrendszer felhasználója el tudjon igazodni a telepítés, a használatbavétel, az üzemszerű használat és az esetlegesen felmerülő hibák során. Ennek megfelelően a felhasználói dokumentációnak is több részből kell állnia. Az egyes részeknek elektronikus vagy papír alapon célszerű létezni, esetleg mind a két formában.

A felhasználói dokumentációnak a következő részeket kell tartalmaznia:

- Általános leírás a rendszerről, amiben a rendszer célja, a képességei le vannak írva.
- A rendszer hardverfeltételei: (minimális, ajánlott) processzor, memória, megjelenítő fajtája, szükséges hely a háttértáron, nyomtató kell-e, egér kell-e, egyéb speciális hardver kell-e.
- A rendszer szoftverfeltételei: operációs rendszer fajtája, verziószáma, esetlegesen szükséges kiegészítő, együttműködő programok, mint pl. megjelenítők, szövegszerkesztők, stb…
- Hálózati alkalmazás esetén, a hálózati operációs rendszer fajtáját, egyéb ismérveit.
- A rendszer telepítésének módja, lehetőleg lépésről-lépésre leírva.
- A rendszer indítása
- A felhasználói interface általános leírása (menürendszerének, párbeszédablakok)
- Az üzemszerű működéshez szükséges részek leírása – pontonként.
- A karbantartási feladatok elvégzéséhez szükséges részek leírása – pontonként.
- A képernyőn megjelenő listák, beviteli helyek, nyomtatási listák leírása.
- Előforduló hibaüzenetek magyarázata, és azok javításának módja.

- GYFK – Gyakran Feltett Kérdések. A programok működése során a felhasználók általában ugyanazokba a problémákba botlanak bele, és ugyanazokat a kérdéseket teszik fel. A kérdéseket és a rájuk adott válaszokat is célszerű befoglalni a dokumentációba
- A felhasználói segítségkérés és a válasz módja.
- További fejlesztési tervek, irányok.

11.3 Fejlesztői dokumentáció

A programok készítése során a fejlesztő előbb-utóbb szembetalálja magát azzal a helyzettel, hogy a régebben írt programok működésére, programrészletekre már nem emlékszik tisztán. Gyakori az is, különösen hosszabb fejlesztés alatt, hogy a tervezés során már jól megtervezett részleteken nem igazodik ki. A későbbi továbbfejlesztéseket, hibajavításokat sem feltétlenül ugyanazok a személyek végzik, akik annak idején a rendszert fejlesztették. Mindezek az okok igazolják a program készítése során a fejlesztői dokumentáció létrehozását.

A fejlesztői dokumentáció célja, hogy a rendszer fejlesztése, a későbbi hibakeresés, illetve a továbbfejlesztések során a rendszerről részletes, bárki hozzáértő által felhasználható dokumentáció legyen. A dokumentációnak minden olyan szükséges információt tartalmaznia kell, ami alapján egy teljesen idegen fejlesztő is sikeresen elvégezhesse a szükséges beavatkozásokat.

A fejlesztői dokumentációt nem szokás átadni a megrendelőnek. Ennek két oka van. Általában a megrendelő nem is tudja használni a fejlesztői dokumentációt. A másik ok pedig az, hogy ha átadjuk a fejlesztői dokumentációt, akkor evvel szabad utat adunk másoknak is a rendszer továbbfejlesztésére.

A fejlesztői dokumentációnál nem elsőrendű fontosságú, hogy papíron is meglegyen a dokumentáció, de mindenképpen olyan formában legyen, hogy évekkel később is felhasználható legyen.

A fejlesztői dokumentációnak tartalmaznia kell a következőket:

- A rendszer tervezése során létrejött minden dokumentációt, azaz
  A rendszer célját, (specifikáció) a tervezésének alapelveit,
  Képernyőterveket,
  Nyomtatási listákat,
  Az adatszerkezeteket,
  Algoritmusokat (Az algoritmusokat magyarázatokkal kell ellátni)
- A fejlesztés hardverfeltételeit: (minimális, ajánlott) processzor, memória, megjelenítő fajtája, szükséges hely a háttértáron, nyomtató kell-e, egér kell-e, egyéb speciális hardver kell-e.
- A fejlesztés szoftverfeltételeit: operációs rendszer fajtája, verziószáma, esetlegesen szükséges kiegészítő, együttműködő programok, mint pl. képszerkesztők, megjelenítők, szövegszerkesztők, egyéb editorok stb…
- Hálózati alkalmazás esetén, a hálózati operációs rendszer fajtáját, egyéb ismérveit.
- A felhasznált fejlesztő eszközök leírását:
  A fejlesztő eszköz elnevezése, verziószáma,
  Esetleges kiegészítő library-k neve, verziószáma,
  Esetleg maga a kiegészítő rendszer elektronikusan.
- Az elkészült szoftver teljes forrásszövegét, megfelelő megjegyzésekkel ellátva
- Az esetleges javítások, fejlesztések verzióit
- A javítások és fejlesztések követésének állomásait, mit javítottak, hogyan javították, mit fejlesztettek
- A továbbfejlesztéssel kapcsolatos terveket, lehetőségeket.

Mivel egy nagyobb rendszer esetén a fejlesztések során együtt változik a felhasználói dokumentáció is a rendszerrel, ezért a felhasználói dokumentációt is tárolni kell a fejlesztői dokumentáció mellett és adott esetben visszamenőleg meg kell őrizni azt, több verzióban is.

12 Betanítás, oktatás

Egy nagyobb rendszer bevezetése esetén a felhasználóknak gyakran gondot okoz az áttérés a korábban használt rendszerekről az újonnan kifejlesztett rendszerre. A felhasználók - teljesen jogosan – a munkájukat akarják csak végezni. Őket alapvetően nem érdekli, hogy milyen módon végzik a munkájukat, csak hatékonyan végezhessék. Egy átállás a hatékony munkavégzésben mindenképpen gátolja őket, ugyanis az átállás során egy ideig együtt kell végezniük a munkát a régi módon és az újonnan bevezetett rendszerben is, hiszen nem biztosítja semmi azt, hogy az új rendszer hibamentes és tökéletes.

Gyakran a munka számítógépesítését csak a főnök akarja, mivel így nagyobb intenzitású munkára tudja az alkalmazottakat rávenni. Gyakran az alkalmazottak nem is értik, hogy mire jó nekik a számítógépes rendszer.

Még manapság is benne van a munkát végzőkben a félelem a számítógépes rendszerektől, hiszen a mai 40-es korosztály még nem használt fiatalabb korában gépet. Bár a helyzet változik, mivel a mai fiatalok az oktatásban kapcsolatba kerülnek gépekkel, de azt is figyelembe kell venni, hogy az emberek képességei nem egyformák, és adott esetben olyannak kell használni egy szoftvert, aki esetleg a betűvetéssel is hadilábon áll.

A felhasználókban sok előítélet él a számítógépes rendszerekkel kapcsolatban. Jobban megjegyzik azokat az eseteket, amikor egy számítógépes rendszer nem a megfelelő módon működött és bosszúságot okozott nekik. (Mellesleg sokszor a „számítógéppel segített ügyintézés” tovább tart, mint a kézzel végzett)

A fenti problémák megoldását a megfelelő teljesítményű hardver és a megfelelően elkészített szoftveren kívül csakis a betanítás és oktatás jelentheti. A korábbiakban beszéltünk arról, hogyan lehet megfelelő szoftvert készíteni, a hardver pedig általában pénzkérdés

Az oktatás során két féle oktatást lehet elképzelni. Általános jellegűt, ami során a gép kezelését sajátítják el a dolgozók, illetve a konkrét rendszerhez tartozó oktatást. A dolgozóknak rendelkezniük kell általános felhasználói ismereteikkel is egy szoftverrendszer használatakor, hiszen vannak olyan helyzetek, amikor nem csak az adott szoftverhez tartozó problémákat kell megoldani. Gyakran egy alkalmazott számítógépén több különböző fejlesztésből származó, különböző célú programok foglalnak helyet, az alkalmazott pedig felváltva vagy akár párhuzamosan is használja őket.

12.1 Általános informatikai jellegű oktatás

Az általános jellegű informatikai oktatást a cégek nem szeretik, hiszen álláspontjuk szerint ne ők fizessék meg az alkalmazottak oktatását, hanem azok már okosan jöjjenek hozzájuk dolgozni. Sajnos egy munkahelyen az alkalmazottak tudásszintje korántsem egységes a számítógépek használata terén. Azok az alkalmazottak, akik valamilyen képzésben korábban részt vettek, gyakran nem használván a tanultakat, el is felejtik azokat.

Egy programrendszer készítőjének meg kell becsülnie a majdani felhasználók oktatásának optimális mértékét. Egy ilyen oktatásnak az a célja, hogy a fejlesztett rendszer használói a jövőben nagyjából egy kötelezően minimális szinten álljanak. Ennek a szintnek elegendőnek kell lennie az általános számítógép-kezelési eljárások elvégzéséhez, az operációs rendszer működtetéséhez, fájlkezelési műveletek végzéséhez, jogosultságok, belépések és kilépések, nyomtatási feladatok elvégzéséhez. Ha multitaszk operációs rendszert használnak, akkor az ebből adódó sajátosságokat is ismerni kell a felhasználóknak. Célszerű oktatni őket a dokumentumok és szoftverek elválasztásának szükségességére. Célszerű olyan gyakorlati tudnivalókat átadni, amelyeket később mankóknak használhatnak. Természetesen nem cél, hogy professzionális felhasználók legyenek az általunk betanítottak.

Tudatosítani kell a majdani diákokban, hogy ha a tanfolyam után hamarosan nem hasznosítják a friss ismereteiket, akkor azok holt ismeretek lesznek, majd hamarosan el is felejtődnek.

Néhány szempont az oktatáshoz:

- A felhasználók többnyire munka mellett vagy munka után részesülnek az oktatásban, ennek megfelelően nem mindig állnak szellemi képességeik legmagasabb fokán.
- A tanítandó anyagot úgy kell megtervezni, hogy minden szükséges gyakorlati tudnivaló benne legyen, amit a felhasználónak a későbbiekben használnia kell. Inkább tervezzünk kicsivel több anyagot a felhasználóknak, mint a feltétlenül szükséges. A túlzásoktól tartózkodjunk.
- Ha nem kell, ne akarjunk elméleti ismereteket átadni, hiszen általában ilyenkor bonyolult fogalmakat, elveket kell magyaráznunk, amiket az átlagos felhasználó elsőre nem is ért meg.

- A tananyag tervezésénél kövessünk egyfajta gondolatmenetet. Arra fűzzük fel mondanivalónk lényegét. Ez lehet a gép használatbavételének sorrendje (bekapcsolás, az elinduló operációs rendszer kezelőfelülete, segédprogramok, elindítása stb…), vagy lehet feladatközpontú is (Hogyan lehet egy feladatot elvégeznem?). Célszerűen lehet keverni a két módszert is.
- Az anyagban kell kellő időt tartalékolni a gyakorlásra. Legalább annyit kell gyakorlással tölteni, mint amennyit az új ismeret átadására fordítottunk.
- Nem szabad az anyagnak nagyon „sűrűnek" lenni. Ha oktatás közben észrevesszük, hogy a kedves felhasználó nem ért egy kukkot sem, akkor meg kell állni, és újra el kell magyarázni a meg nem értett részeket. Ennek megfelelően esetleg az időből kifutunk, de az nem olyan nagy baj.
- Az időbeosztást gondosan meg kell tervezni.
  1. Nem célszerű egy hét alatt lezavarni napi négy órában egy tanfolyamot, mert a résztvevőkre zúduló hatalmas információmennyiséget nem tudják feldolgozni.
  2. A heti egy alkalom azt eredményezi, hogy az eltelt idő miatt a résztvevők többsége nem emlékszik a korábban elhangzottakra. Célszerű legalább heti két alkalmat adni az oktatásra, lehetőleg nem egymás utáni napokon.
  3. A napi óraszám a 2-4 tanítási órában jelölhető meg, két tanítási órát egyben tartva. 1,5 óra után mindenképpen szünetet kell tartani.
- Hagyni kell a tervezés során egy vagy több alkalmat a felmerülő kérdések megválaszolására.
- Célszerű valamilyen vizsgát szervezni az utolsó alkalomra. Mivel a kezdeti tudás sok féle lehet, ezért a tanfolyam hatékonyságát úgy lehet megmérni, hogy ugyanazokból a kérdésekből egy felmérést végzünk a tanfolyam elején és a végén rendezett felmérés eredményét összevetjük az elején elvégzett teszttel.
- Egy számítógépes tanfolyam célszerűen úgy működik, hogy minden hallgató egyedül ül egy megfelelő gép előtt – ez alapfeltétel. A tanfolyam anyagának rövid tömör jegyzetéről is célszerű gondoskodni, akár témavázlat formájában is.

12.2 Rendszer betanításához szükséges oktatás

Az általunk fejlesztett rendszer betanítása csak az után képzelhető el, hogy meggyőződtünk, hogy általában a jövendő felhasználók a megfelelő minimális szinten képesek kezelni a számítógépet. Az oktatás során azt a sorrendet célszerű követni, ahogy a dolgozók a rendszert használni fogják. Az anyag nagyjából a következő legyen.

- A rendszer célja, tudása. A rendszer képességei és a jelenlegi valóság kapcsolata.
- A rendszer telepítése (azoknak, akiket ez érint)
- A rendszer indítása, hardver és szoftverfeltételei
- A program menüpontjainak, áttekintése először, általános tudnivalók a program működéséről, kapcsolatok más programokkal.
- A program áttekintése a munkafolyamatok sorrendjében. A képernyők magyarázata, listák megtekintése, nyomtatási képek megtekintése.
- Egyes feladatok begyakoroltatása.
- Mire kell vigyázni
- Mit kell tenni, ha hiba van.

Az időbeosztásra, a keretfeltételekre (gépek száma stb…) vonatkozó korábban elmondottak most is érvényesek. Nyilván lesz a felhasználók között, akik a fejlesztés során együttműködtek a fejlesztővel, azaz többet tudnak a rendszerről. Hagyni kell őket az oktatás során érvényesülni, hogy ezáltal is elmélyítsék tudásukat.

Meg kell kívánni azt, hogy az első tanítási szakasz után a felhasználóknak legyen lehetőségük a tanultak begyakorlására, azaz legyen rá idő és hely.

13 Garancia, az elkészült rendszerek további gondozása

A jelenlegi magyarországi törvények szerint minden eladott új termékre kötelező egy évig jótállást adni az eladónak. A szoftverek terén a jótállás kissé összetett probléma. Ha valaki bemegy a boltba és megvesz egy dobozos terméket, akkor tulajdonképpen nem a szoftvert veszi meg, hanem a szoftver felhasználási jogát! A szoftver tulajdonosa továbbra is az eredeti fejlesztő. **Ebből következően nem köteles semmiféle garanciát adni az eladónak a szoftver működőképességére vonatkozóan!**
Az úgynevezett „dobozos termékeken" vagy a belsejükben általában található egy licensz szerződésnek nevezett papír, amely azt mondja ki többek között, hogy a program használati jogát abban az állapotban vásárolták meg, ahogyan azt a dobozba csomagolták. Ha a program nem működik az elvárásoknak megfelelően, akkor sincsen semmiféle jogi következménye az eladóra nézve. Az eladó nem vállal garanciát a program működéséért. Ugyanígy a fejlesztő se vállal garanciát. Ha a szoftver kárt okoz, akkor egyes szerződések a dobozos termék áráig vállalnak garanciát, de ez alapvetően nem jellemző.

Más a helyzet akkor, amikor egy konkrét megrendelésre kell konkrét programot fejleszteni. Ekkor elvárható, hogy a program ne működjön hibásan. Azért nem hibátlant írok, mert a korábban megtárgyaltuk, hogy hibátlan program nem létezik.

Általában a Megrendelő és a Fejlesztő közötti alku tárgya, hogy a program működésére vonatkozó garancia milyen és hogyan lehet érvényesíteni, illetve mire terjed ki.

A fejlesztőnek kötelezően elő kell írni olyan procedúrákat, amelyek megvédik a program által használt adatokat a megsemmisüléstől. Ezek lehetnek a programba beépített **mentési**, illetve **backup eljárások**, de lehetnek a rendszergazda számára előírt napi, heti, havi mentések. Ha ezeket nem futtatják rendszeresen, akkor természetesen nem lehetnek biztosak abban, hogy egy áramszünet vagy egy hardver meghibásodása nem teszi –e tönkre hosszú idő munkáját.

A program működésének helyességének a tesztelési időszak alatt kell kiderülnie. A tesztelést pedig a majdani felhasználónak is el kell végeznie.

Ezek ellenére a józan Fejlesztői döntés az, hogy a végleges változat átadása után még legalább egy évig minden javítást, a szoftver tudását nem jelentősen növelő módosítást ingyen és bérmentve vagy valamilyen átalánydíjat felszámítva kell elvégezni. El kell dönteni minden hibajavításnál, hogy kinek a hibájából fordult elő, mi a hiba oka. Ha a fejlesztett szoftver a ludas, akkor a javításért nem szabad kérni semmit. Ha a futtató hardver vagy operációs rendszer hibás, akkor vis maior esete van, bölcs döntés, ha a hibát kijavítja a fejlesztő, és semmiféle ellenszolgáltatást nem kér, ha azonban a felhasználó hibája vagy vírusfertőzés okozta a program hibás működését, akkor megfontolás tárgya egyfajta kiszállási díj kérése.

Az egy éven túli gondozás, illetve a továbbfejlesztés lehetősége külön megállapodás tárgya a megrendelő és a fejlesztő között.

Más a helyzet akkor, ha egy szoftvert több tucat megrendelőnek is értékesítettünk. Feltehetően ez a szoftver kellően stabil ahhoz, hogy ő maga programhibákat ne okozzon. Ilyen helyzetben a fenti garancia – kiszállunk és megjavítjuk – nem működik megfelelően. Ekkor meg kell szervezni egy helpdesk lehetőséget, egy telefonszámot, levél vagy E-mail címet, ahol feltehetik a hibával kapcsolatos kérdéseiket a felhasználók és arra rövid időn belül választ is kapnak. Ha nem voltunk kellően körültekintőek és a programunk hibáktól hemzseg, akkor kötelességünk a hibák kijavítása után mindenkinek elküldeni a javított programváltozatot vagy értesíteni őket arról, hogy hogyan kaphatják meg a javítást. Ez felvet még egy kérdést. Mi van azokkal, akik a programot jogtalanul használják. Ők is kérhetek javítást, supportot? Természetesen nekik nem jár, de a jogosságot nekünk egy konkrét hívás esetén le kell ellenőriznünk. Ha kis eladott darabszámról van szó, akkor könnyű ezt ellenőrizni, de dobozos termékek esetén a visszaküldött regisztrációs kártya a jogosság elismerésének az alapja. A nagy fejlesztő cégek általában átadják a supportot a kereskedőknek, mondván ők vannak közelebb a vásárlóhoz, legyen ez az ő költségük.

Azt szokták mondani, hogy egy program életciklusa általában nem hosszabb 3-5 évnél. Ennek megfelelően nekünk szoftverfejlesztőknek csak maximum 5 évre kell munkát szereznünk, mert legkésőbb 5 év múlva úgyis megkeresnek a korábbi megrendelők és kérik a továbbfejlesztést.

14 Zárószó

A fenti jegyzet természetesen nem lehet teljes, hiszen az informatika és azon belül a programozás annyira szakosodott, hogy a teljes palettát egy könyvbe nem is lehetne összefoglalni. Azok számára, akik további elméleti jellegű tanulmányokat szeretnének önállóan elvégezni vagy a jegyzetben lévő egyes részek iránt érdeklődnek, az alábbi irodalmat ajánlom:

Wirth	Adatstruktúrák + Algoritmus = Programok	
Knuth	Programozás felülnézetből	
	Módszeres programozás	
	Számítástechnika középfokon	OMIKK

Az egyes programozási nyelvek leírása már sokféle formában megjelent, az egyes kiadványok változó, de általában jó színvonalon írják le a választott programozási nyelv szabályait és az ott előforduló lehetőségeket. Az alábbiakban leírunk néhány kiadványt, amelyek az adott programozási nyelvekben elmélyülni kívánóknak nagy segítséget jelenthetnek. Természetesen ez az irodalom lista nem lehet teljes, és csakis az 1998 évi állapotot jelezheti:

Benkő	Programozzunk C nyelven	ComputerBooks, 1997
Benkő	Programozzunk Pascal nyelven	ComputerBooks, 1997
Kernighan - Richie	A C programozási nyelv	Műszaki Könyvkiadó, 1985
Angster - Kertész	Turbo Pascal 6.0 feladatgyűjtemény I - II	
Hargitai - Kaszanyicki	Visual Basic 3.0	Számalk, 19

15 Szervezési ismeretek

15.1 Rendszerelméleti alapok

Rendszer
A rendszer egy több alkotóelemből álló, egymással kapcsolatban álló és egymással együttműködő elemek halmaza.

Részrendszer
Egy rendszer jól elkülöníthető része, amely a többi részrendszerrel jól definiálható kapcsolatokon keresztül kommunikál.

Alrendszer
Egy rendszer jól definiálható feladatokat végző része. Jól definiálható felületeken kapcsolódik a többi részrendszerrel.

Elem
Egy rendszer alkotórésze

Környezet
Egy rendszerrel kapcsolatot tartó viszonyok összessége. A környezet határozza meg a rendszer bemenő paramétereit, az kapja vissza a kimenő eredményeket, az határozza meg a működési körülményeket.

Input
A rendszerbe bemenő adatok halmaza
Output a rendszer által visszaadott eredmények halmaza. Egy rendszer outputja lehet más rendszer inputja is.

15.1.1 Elemzés

A

15.1.2 Modellezés

15.1.3 Szervezet elemzés

Cél – Folyamat
Minden szervezet valamilyen céllal jön létre és a működésének menete a követendő cél elérését szolgálják. Céljának elérését egymással kapcsolatban álló, egymással párhuzamosan működő vagy esetleg egymással konkuráló folyamatok alkotják. A folyamatok lehetnek egymás mellérendelt vagy alárendelt viszonyban egymással.

Szervezet kapcsolati rendszere
Semmiféle szervezet nem működhet önmagában. Mindegyik szervezetnek vannak külső kapcsolatai, amelyek szintén összefügghetnek egymással, illetve lehetnek függetlenek is egymástól. A kapcsolatok lehetnek alárendelt, mellérendelt kapcsolatok a szervezettel.

Feladatkör
A szervezetnek minden esetben a céljaiból levezethető feladati vannak. Ezeknek a feladatoknak a halmaza a feladatkör. A feladatok elvégzésével a szervezet céljai is megvalósulhatnak.

Hatáskör
Azon dolgok halmaza, amelyeke a szervezet működése során befolyását tudja érvényesíteni.

Felelősségi kör
Egy szervezet működése során tevékenysége során hat a környezetére és ezeknek a hatásoknak következményei lehetnek. Azok a jelenségek, amelyek kizárólag a szervezet működésének eredményei, a szervezet helyes, célnak megfelelő működése során előre tervezetten keletkezhetnek. Ebben az esetben a szervezet felelősséget vállalhat a tevékenységéért, amelyben azt vállalja, hogy a működése során csak bizoynos események az előre megadott módon következnek be.

15.1.4 Szervezet - szervezeti felépítés

szervezeti felépítési formák és működésük főbb jellemzői

- lineáris, funkcionális mátrix szervezeti formák
-

15.1.5 Gazdasági rendszerszervezés

- szervezés fogalma, fajtái, szakterületei (cél, irányultság)
- alap vagy fejlesztő
- folyamat, szervezet, munka információ

15.1.6 Ismeretelméleti alapfogalmak

Adat, információ, hír
Ezekről a témákról volt szó az adatbázis-kezelés című jegyzetben, továbbá az Informatika kezdőknek jegyzetben.

információ mértéke és hasznossága
A Shannon féle elméletben az információ mértéke a bit, byte, kbyte, stb...
Hasznosságát nem tudjuk mérni, azaz szemantikai szempontból nem mérhető az információ egzakt módon.

hír
Olyan adat, amely információ és amely valódi döntéseket indukálhat.

hírforrás
A hírforrás az adatszolgáltató, de nem az adat fizikai előállításával foglalkozik, hanem az információvá alakításval

Adó
Az amely az adatot fizikai valójában szolgáltatja

Kódoló
Az adatokat egyik megjelenési formájából átalakítja más formába.

Csatorna
Olyan adatátviteli szabványok és eljárások sorozata, amely az adó és a vevő között felépülve alkalmas az adatok folyamatos, de legalábbis hosszú távú átvitelére. Az **adatátviteli csatorna működése** során zajok lehetnek, amelyek az átvitt adatok deformálódását, hibáját okozhatják. Az adatátviteli csatornák egyik értékmérője a sebességük, míg másik értékmérője a hibátlanságuk, zajtalanságuk. Megjegyzendő, hogy a zaj nem minden esetben káros az átvitt adatokra. Főleg abban az esetben nem, ha kellő redundanciával rendelkezik az átviendő adat.
A vevőhöz megérkezve az adatot visszaalakítjuk eredeti formájára, akkor **dekódoló egységről** beszélünk.

Az informatika fogalma
Az informatika az ismeretek megismerésének, azok célszerű elrendezésének és kezelésének tudománya. Az informatikus az a szakember, aki ebben a tudományban jártas.

A informatika tárgyköre
Az informatika a valósággal, a róla alkotott képpel, a rendelkezésre álló technikai erőforrásokkal foglalkozik.

Az informatika kapcsolata a szervezéssel
Az informatika a megoldandó feladatok és a rendelkezésre álló erőforrások megszervezését végzi, azaz a szervezés az informatikai folyamatok egy része.

Irányítás
A folyamatok irányítása egy visszacsatolásos folyamat. A folyamat működését irányítja, megfelelő pontokon visszajelzéseket kap az irányító folyamat, és a visszajelzések alapján az irányító eszköz (irányító folyamat, team stb..) módosítja a folyamat paramétereit.

15.1.7 Rendszerfejlesztési projekt

A projekt fogalma
A projekt hosszabb időn keresztül zajló, több ember összehangolt munkáját igénylő fejlesztési feladat! (Nem programozási, vagy szervezési és programozási)

Az információrendszer (IR) fejlesztési projekt feladata
Az ilyen projektek azt a célt szolgálják, hogy egy információs rendszert alkosson a csapat, amelyben a világ valamelyik részének megkönnyítik az életét.

Egy cégen belül egyszerre több párhuzamosan futó és esetleg egymásnak nem teljesen megfelelő projekt is haladhat, ami kellemetlen ellentmondásokhoz, erőforrás pazarláshoz és egyéb gondokhoz vezethet, ezért ilyen esetben meg kell alkotni a projektek koordináló vagy kormányzó bizottságát. Ennek a bizottságnak a feladat, hogy a felmerülő ellentmondásokat elsimítsa. E bizottság tagjainak nem adminisztratív vagy felső vezetőknek kell lenniük, hanem esetleg a projekt teamek vezetőinek, hiszen az erőforrások elosztása és az ellentétek elsimítása, az együttműködés megszervezésének feladata amúgy is rájuk hárul..

Ezzel el is mondtuk, hogy minden **projktnek szükséges egy vezetőjének** lennie. Ez az ember az, aki összefogja a csapatot és amennyiben bármiféle külső kapcsolatot is kell találni, a projektvezető az, aki a külső kapcsolatokat megszervezi, stb...

A projektek általában nem lerögzített és bebetonozott fejlesztői csapatok, hanem kétféle módon is változik a csapat összetétele:
- az egyes projektek között van átjárás és egyik projektben egyik ember programozó, míg a másik projektben vezető lehet és fordítva.
- A projektben résztvevők a munka egyes fázisaiban többen vagy kevesebben vannak, attól függően, hogy mennyi és milyen fajta munkára van szükségre a projektnek.

Ezt a munkamegosztás nevezzük **projekt szervezésnek.**
A helyes megközelítésben egy fejlesztő cégnél például az emberek bér, munkaügyi stb.. szempontból tartozhatnak valahová, de a projektekkel kapcsolatban mindig máshol találhatók meg. Azt a folyamatot, amikor megtervezzük, hogy a dolgozók milyen esetekben hol és mit dolgozzanak **mátrix szervezésnek** hívják.

A projektvezetés feltételei
A projetek megfelelő szintű vezetéséhez szükséges, a megfelelő szervezeti keretek biztosítása, a humán erőforrások biztosítása és a megfelelő adminisztrációs erőforrás biztosítása.

A projekt vezetője átlátja a folyamat minden részletét és ő vezeti le a projektet a kialakulástól a végső átadásig. A projektvezetés folyamata során a vezetőség feladata
- a tervezés (durva becslés, projekt szintű szabályok) kialakítása,
- A projekt ütemezése, azaz az egyes lépcsőfokok elérésének az ellenőrzése és teszteltetése!
- A projekt ütemezésének mindenkori korrigálása,
- A projektben résztvevő személyekre kiosztandó feladat kiosztása
- A projekt végső fázisában az előre megállapított pontokon és módszerekkel a teljesítmények figyelése, értékelése

15.2 Az információrendszer fejlesztés életciklusa

15.2.1 Rendszerelemzés

Előzetes helyzetfelmérés
Az információs rendszer életének első fázisa a rendszerelemzés. Meg kell vizsgálni, hogy a jelenlegi rendszernek melyek a korlátai, milyen egységekből (egyed, egyedtípusból épülnek fel) és miért szükséges a változtatás. A rendszer felmérése során nem törekedünk az azonnali, átfogó rendszerelemzésre, hanem inkább iteratív módon fokozatosan közelítjük meg a megoldandó feladatot.

Rendszertanulmány készítése
Az elemzések eredménye egy rendszertanulmány, aminek segítségével hozzá lehet fogni a konkrét feladatok megoldásához. Ez a tanulmány kiindulópontja a későbbi munkának. A rendszerterv elsősorban a laikusok, illetve a rendszer jelenlegi üzemeltetői számára készített tanulmány.

15.2.2 Rendszertervezés

Átfogó helyzetfelmérés
A rendszertanulmány alapján, az informatikus elvégez egy átfogó felmérést, amelyben a folymatoka minden aspektusból megvizsgál és fokozatosan felderíti a követelményeket, a részleteket, stb.. Ehhez interjúkon vesz részt, amelyekben

a megrendelő oldaláról szakértők vesznek részt, míg a rendszer tervezője a szakértők segítségével pontosítja a feladatokat. A helyzetfelmérés alapján a szükségeshez egyre jobban közelítő rendszertervet készít az informatikus.

A **rendszerterv** tartalmazza a szükséges adatszerkezeteket, a használandó eszközöket, a szükséges erőforrásokat, a kritikus algoritmusokat és általában szakmai szempontból előkészíti a rendszerfejlesztés további lépéseit,

15.2.3 Kivitelezés

Programtervezés
A következő lépcső a program megtervezése. A programtervezés során már a figyelembe vett nyelvi elemekkel együtt a rendszertervben lévő adatszerkezeteket és eljárási szabályokat figyelembe véve tervezzük meg a szoftvert. Ez az a szint, ahol még az alap algoritmusok segítségével elindulhat a Down-Top, vagy a Top Down módszer segítségével a rendszer részletezése.

Programozás
A következő lépés a a programozás maga. A programtervezés során részletekre bontott programot a programozók modulonként leprogramozzák, majd az elkészült modulokat összeépítik.

15.2.4 Tesztelés, a rendszer bevezetése

A tesztelés folyamatát tervezni kell. A tesztelésnek az alábbi elvek szerint és módszerek szerint kell lezajlania. (A „Módszeres programozás” c. jegyzet megfelelő részei elolvasandók)

15.2.4.1 A programok tesztelésének célja

A programok tesztelésének célja, hogy a program vajon a bemenetekre a specifikáció alapján megfelelő kimenetet szolgáltatja-e.
A specifikációnak megfelelő programot **helyes programnak** hívják. A programok tesztelése és a hibakeresés során arra törekszünk, hogy az eredeti specifikációnak minél jobban megfelelő, illetve megfelelő programot állítsunk elő. Olyan tesztmódszereket kell használni és olyan hibakereső eszközöket, amelyek a hibák nagy részét kiszűrik. Néhány tapasztalati ténybe, azonban bele kell nyugodni:

- A program hibáinak száma és súlyossága exponenciálisan nő a mérettel
- A hibajavítás után az összes tesztelést célszerű lefolytatni
- A hibát megszüntető okokat kell megtalálni és kijavítani
- Gyakran egy hiba megszüntetése több másik hiba megjelenését vonja maga után
- A program készítője a legrosszabb tesztelő. A fejlesztőn kívül mással is teszteltetni kell a programot.

A fenti tényeken kívül még egy továbbiról is kell szót ejteni. Nagyobb méretű programok esetén 100%-osan hibátlan programról nem lehet beszélni.

15.2.4.2 A tesztelés kritériumai

- A jó tesztelés nagy valószínűséggel felfedi a hibákat
- A jó tesztelési eljárásoknak megismételhetőknek kell lenni
- Érvényes és érvénytelen adatokra is kell tesztelni
- Minden tesztesetet maximálisan ki kell használni, azaz a legtöbb hibát fel kell deríteni
- Fel kell tenni a kérdést, hogy **miért nem azt teszi** a program, amit kellene volna és **miért azt teszi**, amit nem kellett volna.

15.2.4.3 Statikus tesztelési módszerek

A statikus tesztelési módszerek a programkód vizsgálatán alapulnak. Ekkor nem futtatjuk a programot.

Kódellenőrzés
A legegyszerűbb lehetőség. Kinyomtatjuk, vagy a képernyőn átnézzük a kódot, miután begépeltük. Célszerű olyan editort használni, amely kiemeli az adott nyelv kulcsszavait, esetleg színnel vagy más módon elkülöníti az adatokat, az értékadó utasításokat. Ha lehet az program bevitelekor használni kell a strukturált írásmódot, ha akkor nem tettük meg, akkor utólag javítani kell a kódon.

Szintaktikai ellenőrzés
A legtöbb fejlesztő eszköz ma már szintaktikailag ellenőrzi a program kódját és a megfelelő sorban ki is írja a hibaüzeneteket. Az interpreteres nyelvek gyakran már a programsor bevitelekor elvégzik az ellenőrzést, míg a compileres nyelvek csak a fordítás során.

Szemantikai ellenőrzés
Az interpreteres nyelvek esetén csak a programozó tudja végiggondolni, hogy programja valóban logikailag megfelelő, az alkalmazott algoritmusok valóban a kellő végeredményt adják, és a kódolás megfelel az algoritmusnak.

A compileres rendszerek esetén előfordul, hogy a fordító figyelmeztet bizonyos utasítások szemantikai problémáira. Gyakran találnak az ilyen rendszerek felesleges változókat, olyan kódrészleteket, amelyek sohasem futnak le, mindig biztosan azonos értéket felvevő változókat, stb. Azok a compileres rendszerek, amelyek kódoptimalizálást végeznek, gyakran olyan kódot hoznak létre az optimalizálás során, amely logikailag nem felelnek meg az algoritmusnak. Ekkor ki kell kapcsolni az optimalizálást.

Inicializálatlan változók
Meg kell keresni a kódban az inicializálatlan változókat, és kezdőértéket kell nekik adni.

Felesleges utasítások kiszűrése
Gyakran kódoláskor az algoritmusnak megfelelő kódot írunk, holott az adott nyelv ugyanazt a funkciót esetleg gyorsabban is meg tudja oldani.

Keresztreferencia táblázat
Ha a programunkban lévő változók értékeinek változását nem tudjuk követni, akkor célszerű keresztreferencia táblázatot készíteni. Erre a legtöbb fordító képes. Ez egy olyan táblázat, amely felsorolja, hogy az adott változó hol kap értéket, illetve hol történik hivatkozás rá a program során. Ennek alapján megállapíthatjuk, hogy mely változókat használjuk a leggyakrabban.

Típuskeveredés
Egyes interpreteres nyelvek a bevitelkor nem ellenőrzik, hogy az értékadó utasítások két oldalán ugyanolyan típusú értékek szerepelnek-e.

15.2.4.4 Dinamikus tesztelési módszerek

A programok hibáinak egy részét a statikus tesztelési módszerekkel ki lehet szűrni, de vannak olyan helyzetek, hogy csak a futás közbeni ellenőrzés segít. Hogy egy-egy teszt minél több tulajdonságot áruljon el a programról az alábbi módszereket lehet alkalmazni:

15.2.4.5 Fehér doboz módszerek

Az utasítások lefedésének elve
A program minden utasítását legalább egyszer végre kell hajtani.

Döntések lefedésének elve
A programban lévő döntések minden következményét végig kell próbálni. A döntéseket igaz és hamis esetben is végig kell próbálni.

A feltételek lefedésének elve
A programban lévő feltételes elágazásokat minden feltételre ki kell próbálni, illetve az logikai összekötő műveleteket minden lehetséges helyzetre ki kell próbálni.

15.2.5 Fekete doboz módszerek

Ekvivalencia osztályok készítése
A lehetséges bemenő adatokat oly módon kell csoportosítani, hogy milyen kimenő adatot várunk tőlük. Ezeket ekvivalencia osztályoknak hívjuk. Minden ekvivalencia osztályra tesztelni kell a programot.

Határeset analízis
Ha a lehetséges bemenő adatok ekvivalencia osztályait helyesen is állapítottuk meg, és úgy találjuk, hogy az osztályokra megfelelő választ ad a program, még mindig meg kell vizsgálni, hogy az ekvivalencia osztályok

határeseteit hogyan kezeli le a program. Gyakran az ilyen helyzetben adott hibás eredmény helytelen algoritmusra, gondolatmenetre vagy túlzott egyszerűsítésre vezethető vissza

Stressz teszt
A programokat biztosan rossz bemenő adatokkal is tesztelni kell. A programok fejlesztése során a fejlesztő általában feltételezi, hogy a felhasználó csak helyes dolgokat művel, pedig ez nem így van. A felhasználó sokkal gyakrabban téved, hibázik, mint azt a legtöbb fejlesztő képzelné.

15.2.6 Speciális tesztek

Hatékonysági tesztek
A programok tesztelésének utolsó fázisa, annak megállapítása, hogy milyen hatékony a program, illetve menynyire jól felhasználható. Ha nem fut, vagy a sebessége nem megfelelő, akkor meg kell keresni azokat az okokat, amelyek a megfelelő futást megakadályozzák, és annak megfelelően kell módosítani a programot, akár az algoritmusok szintjére is visszamenve.

Biztonsági teszt
A programoknak stabilaknak kellene lenniük, nem szabadna előre látható okok miatt lefagyniuk. Ezekre valók a biztonsági tesztek.

15.2.6.1 Tesztállapotok

Az elkészült program a tesztelés fázisain végigmenve különböző állapotokba kerül.

- A fejlesztők belső tesztelését **alfa tesztnek hívjuk**. Az ilyen állapotban lévő programra azt mondjuk, hogy a teszt változat.
- Ha a programot már a jövendő felhasználók egy kisebb csoportja tesztelheti, akkor ezt az állapotot b állapotnak hívjuk, a tesztelőket béta tesztelőknek.

Gyakori, hogy egy program elkészültének fokát a következő vagy ehhez hasonló módon jelezzük: 0.01, 0. 11, stb...
Ekkor az elkészült, letesztelt program verziószámának az 1.0-át szokás írni.

A tesztelés folyamatát a fenti módszerek figyelembevételével meg kell tervezni és a tesztek tapasztalatait, a bemenő adatokat és az eredményeket jegyzőkönyvben rögzíteni kell. Ezt hívják tesztelési dokumentációnak, amely a fejlesztői dokumentáció része.

15.2.7 Program dokumentálása

15.2.8 Rendszer felhasználói kézikönyve

Egy rendszer felhasználói kézikönyve (dokumentációja) az alábbiakat kell hogy tartalmazza:
A felhasználói dokumentációnak a következő részeket kell tartalmaznia:

- Általános leírás a rendszerről, amiben a rendszer célja, a képességei le vannak írva.
- A rendszer hardverfeltételei: (minimális, ajánlott) processzor, memória, megjelenítő fajtája, szükséges hely a háttértáron, nyomtató kell-e, egér kell-e, egyéb speciális hardver kell-e.
- A rendszer szoftverfeltételei: operációs rendszer fajtája, verziószáma, esetlegesen szükséges kiegészítő, együttműködő programok, mint pl. megjelenítők, szövegszerkesztők, stb…
- Hálózati alkalmazás esetén, a hálózati operációs rendszer fajtáját, egyéb ismérveit.
- A rendszer telepítésének módja, lehetőleg lépésről-lépésre leírva.
- A rendszer indítása
- A felhasználói interface általános leírása (menürendszerének, párbeszédablakok)
- Az üzemszerű működéshez szükséges részek leírása – pontonként.
- A karbantartási feladatok elvégzéséhez szükséges részek leírása – pontonként.
- A képernyőn megjelenő listák, beviteli helyek, nyomtatási listák leírása.
- Előforduló hibaüzenetek magyarázata, és azok javításának módja.

- GYFK – Gyakran Feltett Kérdések. A programok működése során a felhasználók általában ugyanazokba a problémákba botlanak bele, és ugyanazokat a kérdéseket teszik fel. A kérdéseket és a rájuk adott válaszokat is célszerű befoglalni a dokumentációba
- A felhasználói segítségkérés és a válasz módja.
- További fejlesztési tervek, irányok.

15.3 Unified Modelling Language

A korábbi fejezetekben a kódolási technikáktól eljutottunk egy rendszer tervezésének elvi alapjaiig. A továbbiakban megvizsgálunk egy viszonylag új módszert, amellyel gyakorlatban is megnézhetjük a rendszerszervezésben használt eszközöket.

15.4 CASE eszközök szerepe a programozásban

CASE eszköz definiálása
Computer Adided system Engineering – Számítógéppel segített rendszer tervezés
Olyan eszközök, amelyek segítségével informatikai rendszerek egyes tervezési és megvalósítási lépéseit számítógéppel végezhetünk el.

CASE szoftverek csoportosítása
A CASE eszközök nem helyettesítik a megfelelő tervezést, hanem csak segítik azt. Milyen módon segítenek a CASE eszközök és milyen CASE eszközük léteznek?

- Adatszótárak definiálásának lehetősége
- Űrlapok definiálása és azok alapján az adatstruktúrák meghatározásának eszközei
- A logikai tervezés eszközei (adatszerkezetek, logikai kapcsolatok, stbb...)
- A dokumentálás eszközei (a modellekből automatikusan állít elő megfelelő módon dokumentációt)
- Alkalmazási vázlat, amellyel gyorsítani lehet a fejlesztési folymatot.
- Adatbeviteli és kimeneti formátumok gyors tervezése és megvalósítása (Riport writer, dialogus ablak készítő)
- Tesztelési lehetőségek beépítése
- Hibakereső rendszer beintegrálva

A CASE eszközök az információfejlesztési folymat majdnem minden lépésében szerepelhetnek, de nem tipikusan nem szerepelhetnek a rendszer elemző fázisokban. Azt semmiféle CASE eszköz nem tudja helyettesíteni.

A **CASE környezet jellemzői** közé tartozik a grafikus felület, látványos, célszerűen kialakított menürendszerek, ablakozó felületek, diagrammok.

CASE eszközök és 4GL nyelvek kapcsolata
A 4GL rendszerek alapvetően szorosan összefüggnek a CASE eszközökkel. A 4GL rendszerek az alapjai sokszor egy CASE eszköznek, illetve egy CASE eszköz alkalmas valamilyen 4GL rendszer nyelvi elemeit generálni.
Pl. C-Buildr, Delphi, Visual Basic, Access, Visual DBASE, CA-Visual Object, stb...

15.5 A programozó, szervező és felhasználó informális kapcsolata

Az Informatikai rendszer fejlesztése során az együttműködő partnerek szerei az alábbiak.

A **rendszerszervező**, vagy más néven szervező az, aki a rendszert mintegy felülről tudja szemlélni, annak látja összes fontos tulajdonságát, szakmai és informatikai szempontból is. Ismeri az alkalmazott módszereket, a rendszer alkotóelemeit, de nem feladata a program legelemibb részeinek ismerete, a hibalehetőségek minden részének ismerete.

A **programozó** az a személy, aki a rendszerterv utasításai alapján elkészíti az egyes programmodulokat, majd azt összeépíti magasabb szintű modulokká. Ha a rendszer fejlesztésén többen is dolgoznak programozóként, akkor kell közöttük lenni egy „vezetőprogramozónak”, akinek szerepe az, hogy vitás helyzetekben eldöntse, hogy az adott program megfelel-e a rendszertervnek, tanáccsal lássa el a modulok programozóit, segítsen az egyes modulokon dolgozó programozók munkájának összehangolásában, továbbá ellenőrizze azt, hogy az

egyes modulok valóban megfelelnek-e minden szempontból a rendszerterv előírásainak, beleértve a tesztelést és a teszteredmények dokumentálását is.

A **felhasználó** az a személy, aki végső soron használja majd az információs rendszert. Ezek közül a fejlesztés során célszerű kinevezni egy un. Szakértőt, aki a megrendelő, azaz a majdani felhasználók oldaláról közreműködik a rendszer fejlesztésében. A vitás kérdések során ennek a személynek a dolga, hogy különösen a rendszerelőkészítés, majd a beüzemelés, tesztelési fázisban eldöntse, hogy egy adott megoldás vajon megfelel-e, mit kell módosítani az informatikai rendszeren és mi felel meg.
Érdekes e hármas kapcsolata. Ugyanis a hivatalos – megrendelő szállító – kapcsolaton kívül e személyek gyakran a közös munka miatt olyan viszonyba kerülnek, ami során több információ áramlik a fejlesztőhöz, mint amennyi minimálisan szükséges lenne. Ez nem baj, sőt jó dolog, ugyanis ennek nyomán használhatóbb rendszertervek és használhatóbb rendszerek készülhetnek el, mint ha csak formális kapcsolat lenne a személyek között. (Gyakran egy rendszer minőségét nem az általánosított szabályok határozzák meg, hanem a kivételkezelések egyszerű és a szokásokhoz igazodó volta)

Tartalomjegyzék