Szoftverfejlesztés I.

Tömösközi Péter

MÉDIAINFORMATIKAI KIADVÁNYOK

Szoftverfejlesztés I.

Tömösközi Péter

Eger, 2013

Korszerű információtechnológiai szakok magyarországi adaptációja

TÁMOP-4.1.2-A/1-11/1-2011-0021

Lektorálta:
Nyugat-magyarországi Egyetem Regionális Pedagógiai Szolgáltató és Kutató Központ

Felelős kiadó: dr. Kis-Tóth Lajos
Készült: az Eszterházy Károly Főiskola nyomdájában, Egerben
Vezető: Kérészy László
Műszaki szerkesztő: Nagy Sándorné

Tartalom

1. BEVEZETÉS

1.1 CÉLKITŰZÉSEK, KOMPETENCIÁK A TANTÁRGY TELJESÍTÉSÉNEK FELTÉTELEI

1.1.1 Célkitűzés

A közhiedelemmel ellentétben a szoftverfejlesztés nem a programozók feladata. A programozók feladata az, hogy a kész rendszertervek alapján elkészítsék a program futtatható változatát egy adott számítógépes környezetben. A számítógépes környezetet azt határozza meg, hogy a felhasználó milyen hardvereszközöket használ, a számítógépén milyen operációs rendszer fut, és a munkája során milyen egyéb alkalmazói szoftvereket használ.

A programozás, azaz a szoftver terveinek implementálása az adott környezetbe a szoftverfejlesztésnek csak egy részét képezi. A végeredmény létrejöttét illetően ez a leglátványosabb rész, hiszen ennek során készül el a programnak az a változata, amelyet a felhasználó alkalmazni tud a munkája során. Az azonban egyáltalán nem biztos, hogy ez egyben a leghosszadalmasabb munka is a fejlesztés során. Egy jó szoftver készítésekor a programozást egy időigényes tervezési folyamat előzi meg, és remélhetőleg a felhasználó már jóval azelőtt megismeri a majdani szoftver funkcióit, mielőtt az első tesztváltozatot kipróbálhatja.

A szoftverfejlesztés tehát nem a programozó, és főként nem egyetlen programozó feladata. A magányos programozók kora lejárt, hiszen a piaci igények annyira gyorsan változnak, hogy egyetlen ember, egymaga képtelen lenne azokat követni.

Ez a kurzus azt mutatja be, hogy milyen lépések során juthatunk el a szoftverrel szembeni igények megfogalmazásától a tényleges rendszer használatba vételéig, sőt, a lépések sora még itt sem ér véget, hiszen egy szoftverrendszer fejlesztése nem fejeződik be az átadáskor. Az átadás a szoftver evolúciójának az első lépése, de egy jó fejlesztő az átadás után is karbantartja a termékét, és ha szükséges, újabb és újabb funkciókat épít be a már kész termékbe. A szoftverek evolúciója tehát olyan, mint az élővilág evolúciója: örökké tart.

1.1.2 Kompetenciák

Tudás: a kurzus teljesítése után a hallgatók képesek lesznek összetett szoftverrendszerek koncepcionális tervezésére. Mivel a jegyzet elsősorban nem

a programozással szakmaszerűen foglalkozó hallgatók számára készült, a kurzus végén a hallgatók képesek lesznek felmérni azt, hogy az ő munkájuk hol kaphat helyet egy szoftverrendszer elkészítésében. A szoftverek fejlesztése ma már minden esetben csoportmunka, nem kizárólag a magas szintű programozási ismeretekkel rendelkező szakemberek privilégiuma. A kurzus elvégzése azért hasznos a nem programozó hallgatók számára, mert annak elvégzése után akár szoftverfejlesztőkkel együtt dolgozva, akár egy egyedi szoftver megrendelőjeként hatékonyan részt tud venni a szoftverfejlesztési folyamatban.

Attitűdök/nézetek: a kurzus fejleszti az algoritmikus gondolkodást, így a kurzus elvégzése után a hallgatók képesek lesznek a problémák megoldásának algoritmikus leírására, megfogalmazására. A szoftverfolyamat sok lépésből áll, és a kezdeti lépések sikeres teljesítése előfeltétele a magasabb szintű műveletek elvégzésének. A problémák pontos leírásával, illetve azok algoritmikus megoldásával képesek lesznek a komplex feladatok részekre bontására. Az összetett feladatok megoldása a programozók magasfokú együttműködését követeli meg. A kurzus elvégzése után a hallgatók képesek lesznek a szoftverfejlesztésben szükséges együttműködés megvalósítására. Az együttes problémamegoldáshoz szükséges a pontos fogalmazás készsége, a segítőkészség, az együttműködésre való készség. Az összetett problémák megoldásához nem elegendő a gépies gondolkodás, a feladatok gyakran magas fokú kreativitást is igényelnek.

Képességek: a kurzus a következő képességeket fejleszti közvetlenül: áttekintő képesség, következtetési képesség, tervezési képesség, lényegfelismerés, rendszerben való gondolkodás, absztrakt gondolkodás, önállóság, stressztűrő képesség.

1.1.3 A tantárgy teljesítésének feltételei

A tanegység teljesítéséhez a hallgatók a félév során egy zárthelyi dolgozatot készítenek, és önállóan vagy projektmunkában elkészítik egy szoftverrendszer specifikációját.

1.2 A KURZUS TARTALMA

1. A szoftverkészítés folyamata, a szoftverekkel szembeni követelmények
2. Rendszermodellek a szoftverfejlesztésben
3. Szoftvertervezés
4. Objektumorientált tervezés
5. Felhasználói felületek tervezése
6. Gyors szoftverfejlesztés

7. Szoftver-újrafelhasználás
8. Komponensalapú szoftverfejlesztés
9. Szoftverevolúció, szoftvermódosítási lehetőségek, refaktorálás
10. Szoftvertesztelési folyamatok, technikák, szoftvermenedzsment, szoftverminőség, költségek
11. Összefoglalás

1.3 TANULÁSI TANÁCSOK, TUDNIVALÓK

A szoftverfejlesztés nem azonos a programozással, a programozás a fejlesztésnek csak az egyik lépése. Egy szoftverfejlesztő csapatban szükség van olyan szakemberekre, akik fel tudják mérni a megrendelők igényeit, az igényekből modelleket és terveket tudnak készíteni, és ezeket a programozók rendelkezésére tudják bocsátani. A programozók gyakran már a legelső megbeszélések alkalmával utasításokban, algoritmusokban, eljárásokban és objektumokban gondolkodnak, pedig a tervezésnek nem ez a lényege. Amikor egy szoftver terveit készítjük, kizárólag azt kell tisztáznunk, hogy *mit* fog csinálni a szoftver, és nem azt, hogy *hogyan*.

Mint ahogyan a megrendelőnek sem kell ismernie egy programozási nyelv eszközrendszerét, a programozónak sem kell ismernie a megrendelő szakterületét. A szoftverfejlesztésben szükség van olyan szakemberekre, akik hajlandók megismerni egy megrendelő szakterületének alapvető összefüggéseit azért, hogy majd olyan rendszertervet készíthessenek, amely alapján a programozók elkészíthetik a megfelelő forráskódokat, a megrendelő pedig elégedetten nyugtázza, hogy a megrendelt szoftver azt csinálja és úgy, ahogyan azt ő szerette volna.

A jegyzet leckéinek feldolgozása során képzelje magát ennek a középen álló szakembernek a helyébe, és próbálja meg egy szoftver fejlesztésének lépéseit ennek a szakembernek a szemszögéből figyelni!

2. LECKE: A SZOFTVERKÉSZÍTÉS FOLYAMATA, A SZOFTVEREKKEL SZEMBENI KÖVETELMÉNYEK

2.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A számítógépek fejlődése magában foglalja a hardver és a szoftver fejlődését is. Hiába jelennek meg újabb és újabb technikai újítások, egyre gyorsabb processzorok, vagy egyre nagyobb kapacitású háttértárak, ezek képességeit csak akkor tudjuk kihasználni, ha megfelelő minőségű szoftvereket tudunk a számítógépen futtatni. A *megfelelő minőség* minden felhasználó számára mást és mást jelent. Egy új számítógép vásárlásakor rendszerint operációs rendszert is kapunk a géphez, illetve néhány, a gyakori tevékenységek elvégzéséhez használható alkalmazói szoftvert is (böngészőprogram, levelezőprogram, irodai alkalmazások stb.).

Amikor a felhasználó speciális munkára akarja használni a számítógépet, a rendelkezésére álló, vagy a számára elérhető szoftverek nem minden esetben elégítik ki minden igényét. Ilyenkor dönthet egyedi szoftver fejlesztése mellett, és ehhez rendszerint szoftverfejlesztő cég segítségét kérheti. Ebben a leckében azt vizsgáljuk, hogy milyen lépéseken keresztül jutunk el a szoftverrel szemben felmerülő igény kialakulásától a tényleges szoftver használatba vételéig, sőt, azon túl is.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás képessége, áttekintő- és rendszerező képesség, figyelemösszpontosítás.

2.2 TANANYAG

2.2.1 Szoftvertechnológia, szoftverfolyamat

Mottó: „A szoftver olyan termék, amely nem készül el határidőre, többe kerül mint tervezték és – legalábbis részben – nem azt végzi, amit kellene.”

(csalódott felhasználó)

Bármelyik oldalon is állunk számítógép-használó szakemberként, a másik oldallal szemben szinte kibékíthetetlenek az ellentétek. Amikor az ember felhasználóként dolgozik egy szoftverrel, azért elégedetlen, mert nem találja a

megfelelő funkciót. Ha megtalálta, akkor azért elégedetlen, mert az a funkció nem úgy működik, ahogyan azt szeretné. Ha funkció a megfelelő helyen elérhető és azt is csinálja, amit elvárunk tőle, akkor túl bonyolult a megvalósítás: miért nem lehet ezt egyszerűbben megoldani?

A szoftverfejlesztők szemszögéből ugyanezek a problémák úgy jelennek meg, hogy vajon miért nem képes a megrendelő már az első alkalommal pontosan elmondani az elvárásait. Miért csak az első, második, harmadik átalakítás után derül ki, mik is a valós igények? Miért nem érti meg, hogy az a kérés, amit most írt meg, teljesen ellentétes az eddigi munkánkkal, és szóba sem került a szerződés megírásakor?

A számítógépek két alapvető erőforrása a hardver és a szoftver. Hardvernek nevezzük a számítógépet alkotó fizikai (elektronikai stb.) eszközöket, a kézzel fogható alkatrészeket, illetve az ezekből együttesen felépíthető berendezéseket. A szoftver ezzel szemben nem kézzelfogható, a szoftver minden esetben egy szellemi termék, amely a számítógép hardverét az ember számára használhatóvá teszi. Tévedés azonban azt gondolni, hogy a szoftver gyakorlatilag maga a program. A szoftver ennél több, ennél bővebb fogalom. Szoftvernek nevezünk minden olyan szellemi terméket, amely a számítógép hardverét az ember számára használhatóvá teszi, így a számítógépre írott program mellett a dokumentációt, illetve azokat a szakterületi ismereteket, amelyek alapján a számítógépes program elkészült.

1. ábra: Szoftver = szellemi termék

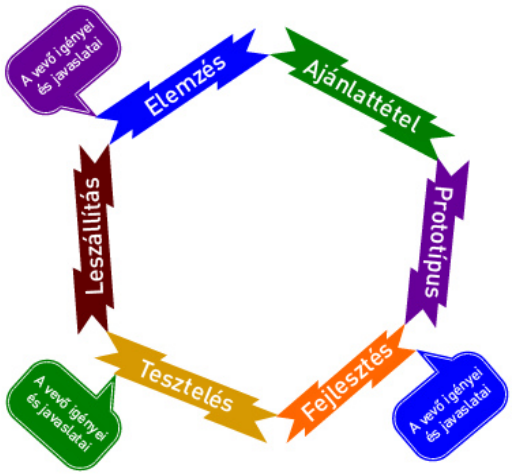
Ha az ember házat szeretne építtetni, alapvetően két féle lehetősége van. Vagy maga találja ki, hogy milyen elrendezésű legyen az épület, hány szoba legyen benne, mekkora konyhája legyen, kell-e terasz, pince stb.majd keres egy építészt, aki megtervezi, egy kivitelezőt aki felépíti a házat, , stb. Ha minden jól megy, az építkezés végén olyan háza lesz, amilyet szeretett volna.

A másik lehetőség az, hogy bemegy egy olyan építészeti irodába, amely kész házakat tervez, épít és forgalmaz. Itt már nem biztos, hogy lehetőséget kap arra, hogy beleszóljon, mekkorák legyenek a helyiségek és talán azt sem döntheti el, hogy fából vagy műanyagból legyenek-e a nyílászárók, esetleg azt sem, hogy milyen színű legyen a csempe a fürdőszobában.

Ehhez hasonló eset, amikor nem is mi építtetjük fel a házat, hanem megvásárolunk egy már álló épületet. Itt a legkevesebb a lehetőségünk arra, hogy a ház a saját igényeinkre legyen szabva. Ebben az esetben az a legjellemzőbb, hogy igyekszünk olyan házat keresni a piacon, amely leginkább megfelel az elvárásainknak.

A szoftverek fejlesztése, egyedi igényeink szerinti elkészíttetése, vagy készen kapható kereskedelmi szoftver vásárlása a fenti példához nagyon hasonló.

A szoftver készülhet általános céllal, ilyenkor sok felhasználó igényinek kielégítésére lehet alkalmas. Készülhet azonban egyedi megrendelésre is, ilyenkor kifejezetten egy megrendelő igényeit hivatott kielégíteni. Ha egy megrendelő egyedi szoftver készíttetése mellett dönt, akkor a szoftver fejlesztője – helyes esetben – az alábbi folyamatot fogja követni. Ez a folyamat a szoftverfolyamat.



2. ábra: A szoftverfolyamat részei

2.2.2 A szoftverfolyamat részei

Igények felmérése

Lehet, hogy meglepően hangzik, de a megrendelők általában nem tudják pontosan megfogalmazni az igényeiket. Van valamilyen elképzelésük, amelyet szeretnének megvalósítatni, de meglehetősen ritka, hogy ezt adekvát módon le is tudják írni. Az igényfelmérés azért fontos, mert a szoftver majdani fejlesztőjének pontosan kell tudnia, hogy mit várnak tőle, viszont éppen ezért már az igényfelmérés során lehetőséget kap nagyon sok olyan részlet tisztázására, amelyet a megrendelő esetleg nem tart fontosnak. A megrendelő kéréseit, elvárásait, ötleteit elemezni, rendszerezni kell.

Specifikáció

Egy számítógépes program feladata – nagyon leegyszerűsítve – az, hogy bizonyos bemenő adatokból (input) bizonyos kimenő adatokat (output) állítson elő. A beérkező adatokat tárolnia és feldolgoznia kell, ezért nagyon fontos, hogy már a tervezési szakaszt megelőzően specifikáljuk, hogy milyen adatok, adatcsoportok fognak megjelenni a szoftverben, ezekkel pontosan milyen műveleteket kell elvégezni. A specifikáció tehát meghatározza, hogy a program milyen adatokkal fog dolgozni, és milyen műveleteket fog rajtuk végrehajtani.

3. ábra: Specifikáció

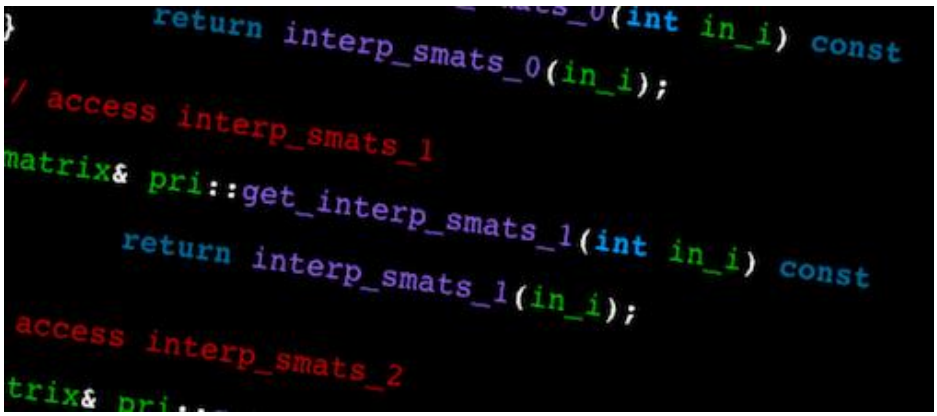
Tervezés

Amikor már tudjuk, hogy a készítendő program milyen funkcionalitással fog rendelkezni, és ezeket milyen adatokon fogja elvégezni, meg kell tervezznünk a szoftver részleteit. Egy nagy projekten rendszerint nem csak egy programozó dolgozik, így a szoftver fejlesztését részfeladatokra kell bonatni. Ez követheti a szoftver egyes egységeit (pl. az egyes menüpontokhoz tartozó funkciókat más-más programozó fejleszti), de rendszerint nem ez a jellemző, hiszen például a különböző menüpontokban elérhető funkciók használhatnak közös alprogramokat, közös folyamatokat. Az a legrosszabb eset, ha az azonos folyamatok programrészeit többször is megírjuk ugyanazon a szoftveren belül. Egyrészt, ha módosítani kell a folyamaton, akkor az összes alprogramot módosí-

tani kell, másrészt ez sokkal több hiba lehetőségét is magában hordoz. A szoftverfejlesztést tehát meg kell hogy előzze egy alapos tervezési folyamat, ahol pontosan tisztázni kell az elvégzendő feladatokat, illetve azt, hogy ezekért ki a felelős személy.

Implementáció

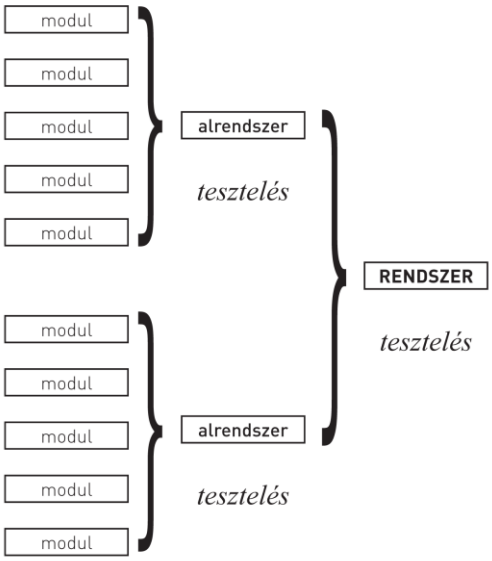
A specifikáció és a tervezés után, esetleg azzal párhuzamosan végezhető a tényleges programozási feladat, azaz a fejlesztés. Ha a szoftver több részből, több modulból áll, akkor ebben a fázisban az egyes modulok fejlesztése történik meg. A tervezés azért fontos, mert a program egyes részei kommunikálnak egymással, esetleg más, már létező szoftverekkel. A programozóknak úgy kell elvégezniük a munkájukat, hogy a programrészek később összeállíthatók legyenek egyetlen rendszerré.



4. ábra: Implementáció, azaz a tényleges kódolás

Integráció

Az elkészült részek összeállítása. Az integráció nem egyszerűen a programrészek (pl. forráskódok) összemásolását jelenti. Ebben a fázisban talán a tesztelés kapja a legnagyobb hangsúlyt: vajon képesek-e együttműködésre a programrészek, és ez az együttműködés összhangban van-e a specifikációban kitűzött célokkal, illetve a szoftver képes lesz-e a terveknek megfelelő működésre? Ebben a fázisban tehát már kell kezdeni a tesztelést, ami ekkor gyakran a hibák felkutatásában nyilvánul meg.



5. ábra: Integráció és integrációs tesztelés

Verifikáció, validáció

Ebben a fázisban kell megmutatni, bizonyítani, hogy a szoftver megfelel a követelményeknek és a felhasználó elvárásainak. Ez a két dolog gyakran nem ugyanazt jelenti. Például egy számlázó szoftverben minimális követelmény, hogy a számlák sorszámozása szigorúan monoton növekvő legyen, és ne legyen lehetőség a számlasorszámok utólagos manipulációjára, visszadátumozására. Még akkor sem, ha a megrendelők egy része ezt kifejezetten hasznos funkciónak tartaná. A szoftvernek meg kell felelnie a felhasználó igényeinek, de ezen felül meg kell felelnie a majdani futtató környezet követelményinek, törvényi előírásoknak stb.

Evolúció

Ez a fázis a szoftver megrendelőnek való átadását, leszállítását követően jön el. A szoftver evolúciója magában foglalja a karbantartást, pl. biztonsági mentések kezelése, adatok megsemmisítése a törvényi előírásoknak megfelelően. Ide tartozik a szoftver továbbfejlesztése a megrendelő igényeinek megfelelően.

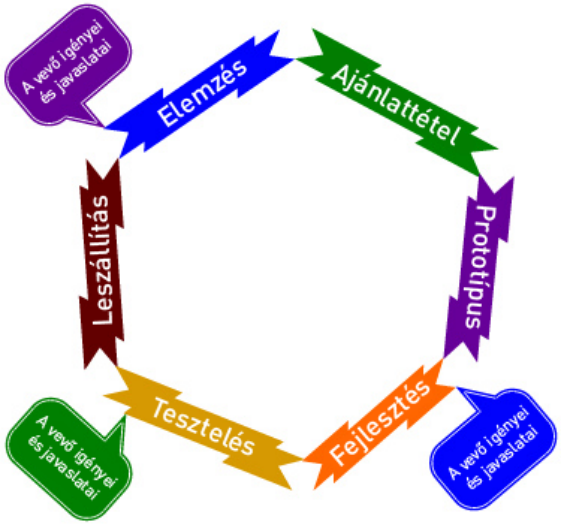
2.2.3 A szoftverrel szembeni igények felmérése

Amikor egy megrendelő egyedi szoftverfejlesztés mellett dönt, vagyis szoftvert készíttet valamilyen feladat ellátásához, számos előzetes elképzelése van arról, hogy mit vár az elkészítendő programtól. Hiába készítünk stabilan futó, gyors, esztétikus stb. programot, ha az nem azt csinálja, amit a megrendelő elvár tőle.

Ahhoz, hogy egy általunk fejlesztett szoftver az elvárásoknak megfelelően működjön, rendszerint az adott szakterület alapvető ismereteivel is rendelkeznünk kell.. Annak érdekében, hogy az általunk fejlesztett szoftver helyesen működjön, meg kell értenünk az adott szakterület szakmai feladatait. Egy raktárnyilvántartó program fejlesztőjének rendelkeznie kell legalább minimális logisztikai ismeretekkel. Ha pénzügyi szoftvert készítünk, ismernünk kell az adózás jogszabályainak legalább az alapjait. Ne kezdjen kottagrafikai szoftver fejlesztésébe, aki nem tud kottát olvasni, és ne fejlesszen nyelvoktató multimédiás alkalmazást az, aki csak a saját anyanyelvén tud kommunikálni, stb.

A szoftverfejlesztés felsorolt fázisai nem függetlenek egymástól, és nem feltétlenül követik szigorúan a fenti sorrendet. Minimális követelmény a megrendelővel való folyamatos kapcsolattartás, hogy ne fordulhasson elő az, hogy a rendszer bizonyos hiányosságai csak az átadáskor derülnek ki. Főleg akkor, ha ezek a hiányosságok abból erednek, hogy nem értettük meg pontosan, mit vár tőlünk a megrendelő.

Szerencsés esetben a megrendelő a szoftverfolyamat minden fázisába be tud kapcsolódni. Ha a megrendelő egy cég, akkor még célravezetőbb, ha a cég alkalmazottait, azaz a későbbi felhasználókat is be tudjuk vonni a tervezési, tesztelési folyamatba: készítsünk prototípusokat, amelyeket a majdani felhasználók ki tudnak próbálni.



6. ábra: A megrendelőt a lehető legtöbb folyamatba be kell vonni

Ne feledjük, hogy a majdani felhasználók nem feltétlenül rendelkeznek informatikai ismeretekkel, kompetenciákkal. Lehet, hogy korábban már használtak valamilyen szoftvert az adott feladat elvégzésére, de az is lehet, hogy eddig csak kockás papírt és golyóstollat használtak az adott tevékenységhez. Ha az adatrögzítést eddig papíralapú űrlapokon végezték, nagyon hálásak lesznek azért, ha az általunk fejlesztett szoftver elektronikus űrlapja ugyanolyan nevű mezőket fog tartalmazni, amilyeneket ők a papíron használtak. Ha az elektronikus űrlapunkat adatrögzítésre fogják használni, az adatok forrásai pedig az eddig használt papíralapú űrlapok, akkor nem igényel különösebb magyarázatot az a követelmény, hogy a két űrlapon az adatok sorrendje egyezzen meg.

2.2.4 Specifikáció

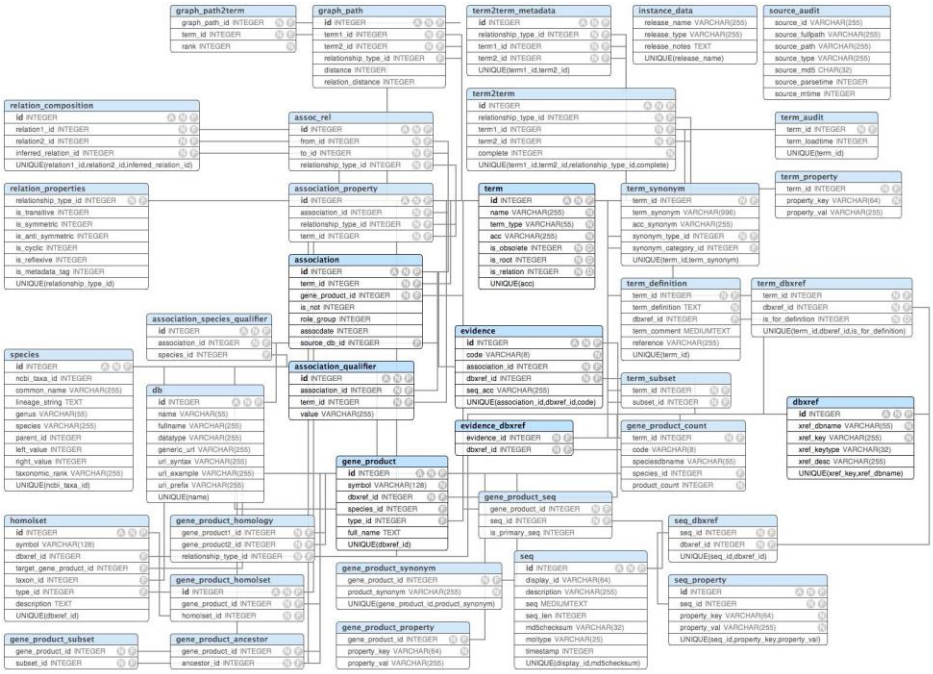
Ebben a fázisban kell következetesen megfogalmaznunk és leírnunk a követelményeket. A hangsúly a következetességen és a leíráson van. Ha ezt a fázist nem kellő gondossággal végezzük el, illetve nem dokumentáljuk kellő mértékben, akkor az eredmény minősége biztosan nem lesz megfelelő.

Rendszerbe kell foglalnunk a szoftver által megvalósítandó feladatokat. Ebben a fázisban azt kell leírnunk, hogy mit fog csinálni a szoftver, nem pedig azt, hogy hogyan.

Fel kell készülnünk arra, hogy a követelmények folyamatosan változ(hat)nak, ezért a specifikációnak is változtathatónak kell lennie. (Gondoljunk csak az adózást szabályozó törvényekre és jogszabályokra.)

Kezelni kell tudnunk az olyan helyzeteket is, amikor a megrendelő a munka olyan fázisában nyújt be új, vagy megváltozott igényeket, amikor az már nem kivitelezhető, vagy csak jelentős költségnövekedés árán valósítható meg. Szoftverekkel kapcsolatosan ez sokkal gyakrabban előfordul, mint például egy ház építése során, ahol a megrendelő rendszerint könnyen belátja, hogy az elfogadott alaprajz és építési terv akkor már nem módosítható, amikor már állnak a falak és készen van a tető.

Prototípus már a specifikáció során is készülhet. A majdani felhasználó sokkal jobban eligazodik egy látható, kipróbálható űrlapon, mint egy, a készítendő adatbázis tábláit és kapcsolatait bemutató ábrán.



7. ábra: A megrendelő bizonyosan nem ilyen ábrákat szeretne nézegetni (forrás: http://www.geneontology.org/GO.database.shtml)

2.2.5 Tervezés

Ebben a fázisban már konkrét programozói feladataink is vannak. A szoftver tervezésekor meghatározzuk:

- a szoftver és a benne használt adatok struktúráját,
- a program komponensei közötti kommunikációt, illetve az ehhez szükséges felületeket (interfészeket),
- magukat a komponenseket,
- az adatok tárolásához használandó adatszerkezeteket (pl. rekordok leírása)
- az elvégzendő műveletek, funkciók algoritmusait.

Kihívások a tervezés során

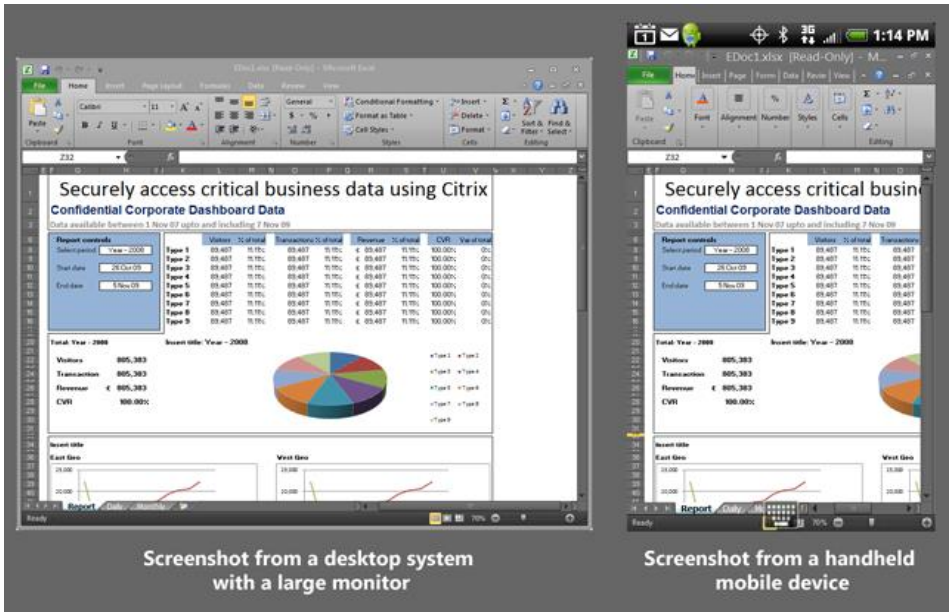
Ma már viszonylag ritka az olyan feladat, amelynek elvégzéséhez szoftvert készíttet egy megrendelő, és amelyet – legalább valamilyen módon – ne számítógéppel végeznének már a megrendelés pillanatában is, vagy az elvégzésében ne vennék igénybe számítógép használatát. A legtöbb esetben jellemző, hogy amikor egy megrendelő szoftvert készíttet, az elvégzendő feladat megoldásához legalább részben már most is számítógépet, azaz egy másik, meglévő szoftvert használ.

A ma használt rendszerek nagy többsége évekkel ezelőtt készült. Valószínű, hogy bármennyire is jól karbantartott rendszerről van szó, vannak olyan gyenge pontjai, amelyeket – ha ma kellene megírnunk a szoftvert – már egészen másként csinálnánk. Ennek oka gyakran abban keresendő, hogy a régi szoftverek egykor még régebbiek voltak, vagyis a ma használt verziók is egy még régebbi program karbantartásának, vagy fejlesztésének eredményei. Hosszú távon biztosan jobb megoldás ezeket a rendszereket teljesen leépíteni, azonban rövid távon ezek olyan költséggel bírnak, amelyet nem minden megrendelő akar vagy tud vállalni.

Ha szoftvert kell fejlesztenünk egy olyan feladat megoldására, amelyhez a megrendelő jelenleg egy meglévő rendszert használ, a tervezéskor figyelembe kell vennünk az ezzel a régi rendszerrel való kompatibilitást.

Figyelembe kell vennünk azt is, hogy a megrendelők nagy valószínűséggel eltérő képességekkel és erőforrásokkal rendelkező munkaállomásokon szeretnék használni a termékünket. Lehet, hogy ugyanazt a szoftvert (adatbázist) el kell érnie a felhasználóknak különböző operációs rendszert futtató asztali számítógépekről, de éppen PDA-ról vagy egyéb mobil eszközről is. A szoftvernek tehát lehetőséget kell biztosítania arra, hogy eltérő környezetben is használha-

tó legyen, ehhez valószínűleg többféle interfész (felhasználói felület) fejlesztésére is szükség lehet.



8. ábra: Ugyanaz az alkalmazás különböző platformokon (forrás: http://community.citrix.com/display/xa/Tuning+the+Apps+for+the+Demos)

A megrendelők általában nagyvonalúan kezelik a rájuk vonatkozó időkorlátokat a feladataikat illetően, de rendszerint elvárják, hogy a szoftver határidőre készüljön el. Ezek a határidők a szoftverek terén folyamatosan rövidülnek. Ennek oka a megrendelői igények mellett a konkurencia jelenléte is: ha a vetélytárs hamarabb elkészül, akkor ő kapja a megrendelést.

2.2.6 Implementáció

Ez a fázis a konkrét programozást foglalja magában. Az elkészült modulokat, komponenseket, egységeket tesztelni kell, a működő komponenseket integrálni. A tervezés és az implementáció párhuzamos munkamódszerekkel történhet, amennyiben egy komponens elkészítése nem feltételezi egy másik komponens meglétét.

2.2.7 Integráció, verifikáció és validáció

Ebben a fázisban zajlik a szoftver tesztelése. A szoftverek tesztelése önálló tudományág, számos módszer és stratégia létezik a hibák felkutatására, feltárására, illetve az okok felderítésére. A verifikáció során azt ellenőrizzük, hogy a kész szoftver a specifikációnak megfelelően működik-e? A validáció célja pedig annak a megállapítása, hogy a szoftver megfelel-e a minőségi és technológiai előírásoknak.

2.2.8 Szoftverevolúció

Egy kész szoftver átadás utáni módosításának többféle oka lehet. Az üzemszerű karbantartás részét képezheti a szoftver működésének ellenőrzése például a logfájlokon keresztül. Szükségessé válhat a tárolt adatok olyan karbantartása, amely a szoftverbe épített funkciókon keresztül nem érhető el.

Gyakori eset, hogy a felhasználói igények, esetleg törvényi, jogszabályi változások miatt szükségessé válhat a szoftver egyes részeinek átalakítása, átírása, illetve újabb igények felmerülésekor új komponensek készítése és integrálása.

Előfordulhat az is, hogy a szoftvert kiszolgáló hardver, vagy operációs rendszer frissítése, cseréje miatt válik szükségessé az általunk készített szoftver karbantartása.

Ahhoz, hogy ezeket a változtatásokat minél egyszerűbb legyen végrehajtani, már a szoftver tervezésekor fel kell készíteni azt a majdani változtatásokra, továbbfejlesztésre. Többek között itt derül ki, hogy a specifikáció, a tervezés és az implementáció (stb.) során mennyire volt körültekintő és teljes körű a dokumentáció. A szoftver jövője nem függhet attól, hogy a szoftvercég alkalmazottai mennyire emlékeznek vissza egy évekkel korábban kitalált rekordszerkezetre vagy éppen egy keresőeljárásra, nem beszélve arról, ha a szoftvercég egykori alkalmazottai esetleg azóta más munkahelyen dolgoznak, vagy már maga a szoftvercég sem létezik.

2.2.9 Szoftverekkel szemben támasztott követelmények

Amikor egy megrendelő szoftver készíttet velünk, általában a következő igényeket fogalmazza meg az elkészítendő szoftverrel kapcsolatban:

- A szoftver rendelkezzen a kívánt funkcionalitással.
- Legyen megfelelő a teljesítménye (számítási gyorsaság, pontosság, több felhasználó esetén a felhasználók hozzáférésének gyorsasága, stabilitás).

- Készüljön el a megbeszélt határidőre.
- Legyen üzembiztos és karbantartható. Alkalmazkodjon az újabb igényekhez.
- Legyen kellően kipróbált, tesztelt a váratlan hibákkal szemben. A szoftver hibája nem okozhat anyagi kárt a cég gazdálkodásában.
- Legyen jól dokumentált, és a felhasználók tudják egyszerűen kezelni.
- Optimálisan használja a hardver és az operációs rendszer adta lehetőségeket.

A szoftverpiac, illetve a felhasználók igénye a szoftverfolyamat felgyorsítása. A felhasználók minél hamarabb szeretnének hozzájutni a számukra legmegfelelőbb szoftverhez, a konkurencia pedig igyekszik ennek minél hamarabb meg is felelni. A szoftvercégek felelőssége az, hogy a verseny, a gyorsaság ne befolyásolja károsan a szoftverminőséget.

Egy szoftver minőségét az határozza meg, hogy mennyire felel meg az általános szabályoknak (törvények, jogszabályok, szabványok stb.), illetve a saját specifikációjának. Ez elsősorban minőségbiztosítási kérdés, ezért a szoftver minőségét és a gyártó céget minőségbiztosítási szempontból is folyamatosan tesztelni kell.

2.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

2.3.1 Összefoglalás

A szoftverfejlesztés csoportmunka, projektmunka. Egy programozó önmagában aligha tud megfelelni a piaci igényeknek, amennyiben a fejlesztés túlmutat egy néhány oldalból álló, reklámcélú weboldal összeállításának keretein. A szoftvercégek nem csak programozókból állnak, hiszen a szoftverfejlesztés nem egyenlő a programozással, ahogyan a szoftver sem csak magát a konkrét számítógépes programot jelenti.

A szoftverfejlesztés mindig egy folyamat eredménye. A folyamat az igények felmérésénél kezdődik, és nem ér véget a szoftver átadásával. A szoftverevolúció (karbantartás, frissítés, javítások, új komponensek integrálása) a szoftver leszállítását követően is ad munkát a fejlesztőknek.

A leckében leírtak alapján mindenkiben felmerülhet a kérdés: ezek szerint nem a programozás a legfontosabb, leghangsúlyosabb feladat egy szoftver elkészítése során? Majdnem biztosan állíthatjuk, hogy nem. Hiába a remek programozó, aki hiba nélkül, határidőre készít el minden feladatot, ha nem fordít energiát a dokumentáció elkészítésére. Hiába a remek tervezés, ha a specifiká-

ció nem készülhetett el megfelelő minőségben, mert nem fordítottunk kellő gondot a megrendelő igényeinek felmérésére. A sort még folytathatnánk, de talán így is belátható, hogy a szoftverfejlesztés hosszadalmas, aprólékos, szervezett munkát igénylő feladat. A piaci folyamatok azonban nem segítik az ilyen munkákat, hiszen a piac a lehető leggyorsabb fejlesztést várja, a lehető legrövidebb időn belül szeretné megszerezni a tökéletes szoftvert. A piaci igények gyakran a minőségi munka ellen dolgoznak.

2.3.2 Önellenőrző kérdések

1. Mi a szoftverfolyamat, vagy szoftvertechnológia?
2. Milyen fázisai vannak?
3. Ismertesse az egyes fázisokban végzett tevékenységeket!
4. Mi a szoftverevolúció?
5. Milyen követelményeket támasztunk a szoftverekkel szemben?

3. LECKE: RENDSZERMODELLEK A SZOFTVERFEJLESZTÉSBEN

3.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A jól felépített tervezési folyamatban a kezdetektől részt vesznek a majdani felhasználók. Az ő számukra természetes nyelven írjuk le a rendszerkövetelményeket, hiszen ők nem (feltétlenül) informatikai szakemberek. Ez azonban a fejlesztők számára nem minden esetben megfelelő módszer, hiszen a természetes nyelven történő leírás sokszor nem kellően pontos vagy szakszerű. A modelleket sokszor nem is szövegesen, sokkal inkább grafikusan adjuk meg, hogy a lehető legkevesebb félreértés vagy tisztázatlan részlet okozzon problémát a tervezési folyamatban.

A leckében néhány, gyakran használt rendszermodellel ismerkedünk meg. A modellezés legfontosabb jellemzője, hogy a komplex probléma megértéséhez, leírásához elhagyja a szükségtelen vagy kevéssé szükséges elemeket, hogy csak a megoldás szempontjából legfontosabb kérdéseket lehessen vizsgálni, azokat viszont a lehető legalaposabban. A különböző rendszermodellek abban különböznek egymástól, hogy mit tekintenek lényegi kérdésnek a probléma vizsgálata során. Ebben a tekintetben megkülönböztethetjük a következő modelleket:

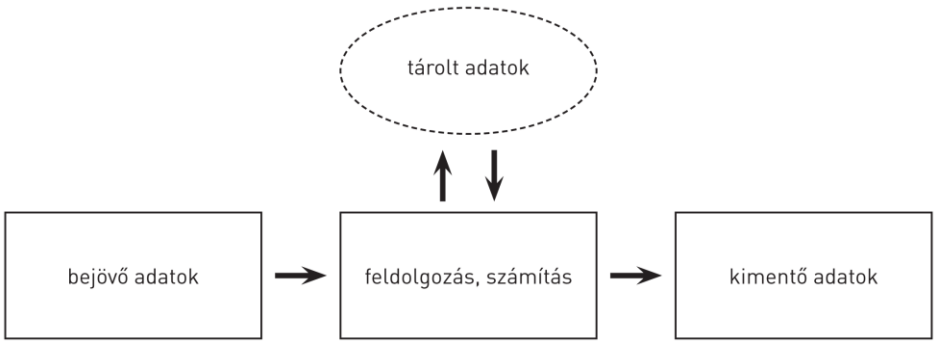
- adatfolyammodell, amely az adatok feldolgozását vizsgálja a szoftver működésének különböző szakaszaiban,
- kompozíciós modell: amely azt vizsgálja, hogyan származtathatók a rendszer egyedei más egyedekből,
- architekturális modell: amelyben a fő szempont a rendszert felépítő fő alkotóelemek vizsgálata,
- osztálymodell: amely kifejezetten az objektumorientált paradigma szemszögéből vizsgálja a rendszert (osztályok, objektumok és ezek kapcsolata).

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező képesség, figyelem-összpontosítás.

3.2 TANANYAG

3.2.1 Adatfolyammodell

Az adatfolyammodellek arra használhatók, hogy szemléltessük, hogyan áramlanak az adatok a feldolgozási folyamat egyes lépései során. Az adatok minden egyes lépés során átalakításon mennek keresztül, ezek a transzformációk folyamatokat, vagy akár konkrét függvényeket jelentenek. Az adatfolyammodellek tehát funkcionális szemszögből mutatják be a rendszert: az elemzők azt láthatják, hogy a folyamatokba érkező egyes inputok hogyan alakulnak át outputokká.



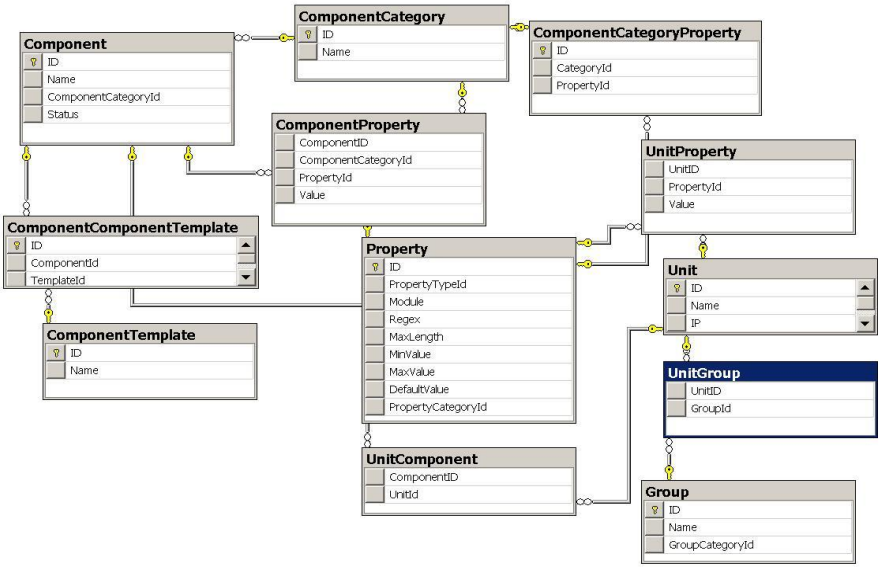
9. ábra: Az adatfolyammodell diagramja

3.2.2 Adatmodellek

A nagy szoftverrendszerek rendszerint jelentős mennyiségű adatot tárolnak és dolgoznak fel, ezért ezen rendszerek mögött nagy méretű adatbázisok állnak. Az adatbázisok tervezési modelljei közül az egyik legismertebb az egyedtulajdonság-kapcsolat (röviden: ETK) modell, amely megmutatja, hogy az adatbázis milyen egyedeket és azoknak milyen tulajdonságait tárolja, illetve leírja az egyes egyedek között fennálló kapcsolatokat. Ennek a modellnek az az egyik előnye, hogy az így leírt sémák azonnal 3NF-ben (3. normálformában) vannak, vagyis az adatbázis modellje mentes a részleges és tranzitív függésektől, illetve minden tábla rendelkezik elsődleges kulccsal.

- Emlékeztetőül: a relációs adatmodellben elsődleges kulcsnak nevezzük az egyedtípus (tábla) azon tulajdonságát (mezőjét), amely minden egyes egyedelőfordulás (rekord) esetében különböző értéket vesz fel. Részleges függésről akkor beszélünk, ha egy egyedtípuson belül az egyedek nemcsak az elsődleges kulccsal állnak funkcionális függésben, hanem annak egy valódi részhalmazával is. Tranzitív függésről akkor

beszélünk, ha egy egyedtípus valamilyen A tulajdonsága funkcionálisan függ egy B tulajdonságtól, ami funkcionálisan teljesen függ a C kulcstól.

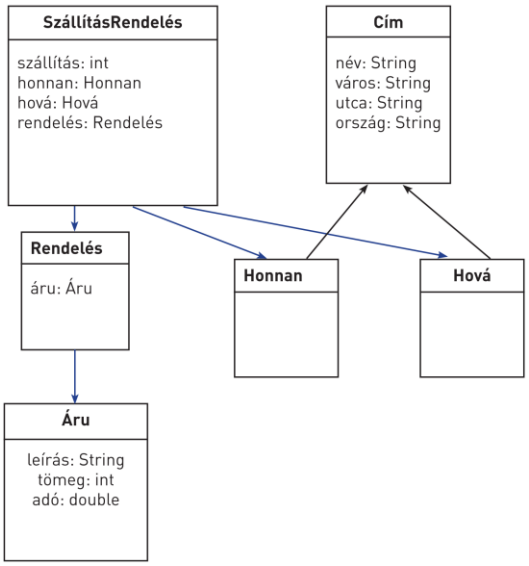


10. ábra: Egy adatbázis ETK-modellje

3.2.3 Objektummodell

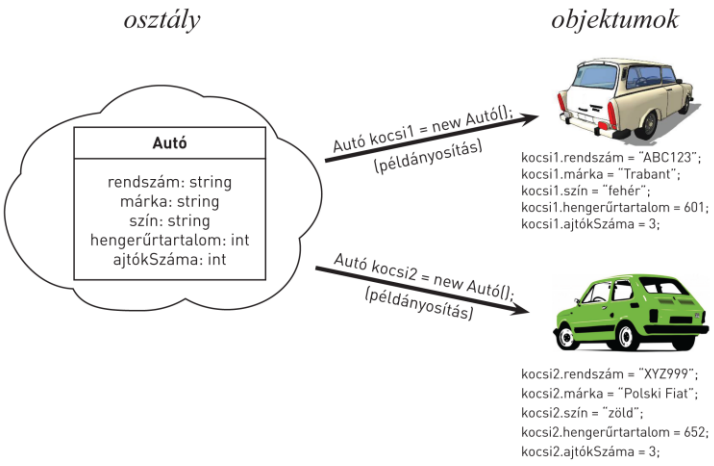
Az 5. leckében az objektumorientált tervezéssel foglalkozunk részletesen. Az objektummodell elsősorban olyan feladatok megoldásában használható fel, amelyek implementálása objektumorientált programozási nyelven (Java, C#, C++) történik. Az objektumorientált szemlélet a valós világ dolgait modellezi. Az objektumokat osztályokba soroljuk, egy osztály egyedei azonos jellemzőkkel és viselkedéssel rendelkeznek. Az objektumok az osztályok konkrét példányai. A program a konkrét objektumok halmazaként fogható fel, az objektumok a program futása során kommunikálnak egymással, ezáltal együttműködnek. A tervezés során az objektumok osztályait kell felismernünk és felhasználnunk a modellezési folyamatban.

Az objektumok modellezéséhez rendszerint az ún. UML (Unified Modeling Language, egységes modellezőnyelv) használatos. Ebben a rendszer objektumosztályait, azok jellemzőit és metódusait (ld. 5. lecke) írjuk le, ahogyan az alábbi ábrán is látható.



11. ábra: Egy UML-diagram

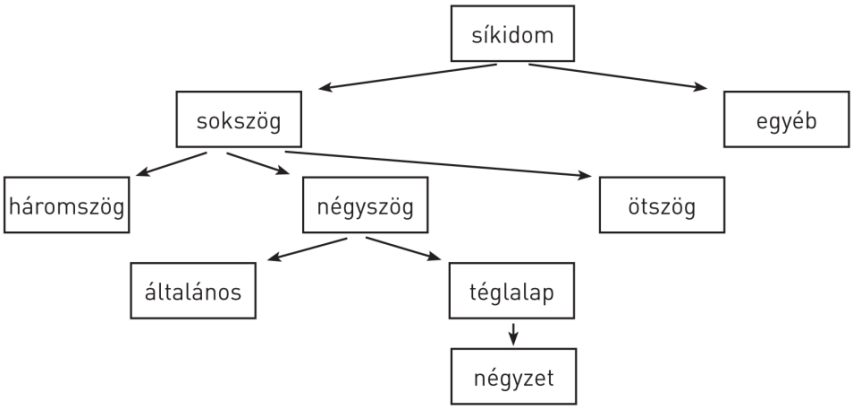
Az objektumorientált paradigma három alapelvre épül. Az egyik az egységbe zárás (encapsulation), amely azt jelenti, hogy az objektumok egy egységként kezelik az adatokat és a rajtuk végezhető műveleteket. Az objektumok belső állapottal rendelkeznek, amelyet elrejtenek a külvilág elől. Egy objektum úgy működik, hogy a létrehozásakor (példányosítás) egy speciális függvény, a konstruktor beállítja az objektum kezdőállapotát, majd amint egy másik objektum üzenetet küld neki, arra reagál valahogyan.



12. ábra: Osztály – példányosítás – objektum

A reakció lehet pl. az, hogy ez az objektum is üzenetet küld egy másik objektumnak, vagy éppen létrehoz egy új objektumot, vagy megváltoztatja a belső állapotát. Az objektumok nem mutatják meg a külvilágnak, hogy hogyan működnek: a kapott inputokból outputokat állítanak elő, de azt nem árulják el más objektumoknak, hogy ezt hogyan teszik. Így elkerülhető, hogy a külvilág objektumai beavatkozhassanak egy objektum működésébe, és megváltoztathassák egy objektum belső állapotát.

Az objektumorientált paradigma másik építőköve az öröklődés (inheritance). Amikor egy új objektumosztályt úgy akarunk leírni, hogy a definícióhoz felhasználjuk egy másik, már létező osztály definícióját, akkor az új osztály származtatható, örököltethető a régiből. Az így létrehozott osztályt alosztálynak, míg az eredeti, a régi osztályt ősosztálynak nevezzük. A származtatott alosztály mindenben úgy viselkedik, mint az ősosztály, hacsak felül nem bíráljuk bizonyos pontokon a definícióját. Ez rendszerint azt szokta jelenteni, hogy egy származtatott osztály definíciójában pontosítjuk az ősosztály definícióját. Az alosztály ilyenkor mindig speciálisabb, az ősosztály pedig általánosabb.



13. ábra: Öröklődés

A harmadik építőelem az öröklődésből következik, és többalakúságnak (polymorphism) nevezzük. A többalakúság vagy polimorfizmus azt jelenti, hogy egy származtatott osztályban egy felüldefiniált metódus (ld. 5. lecke) ugyanúgy képes működni, mint az ősosztályban definiált eredeti metódus.

Az OOP-ről (objektumorientált programozás) részletesen írunk az 5. leckében. Az objektummodell használatakor a majdani rendszer objektumait, pontosabban azok típusait, az objektumosztályokat, illetve a közöttük fennálló kapcsolatokat adjuk meg.

3.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

3.3.1 Összefoglalás

A rendszertervezés és -fejlesztés sikeréhez feltétlenül hozzátartozik, hogy alapos és pontos tervezési folyamaton essen át a készülő szoftver. Ahhoz, hogy egy problémát meg tudjunk oldani, azt először meg kell értenünk és pontosan le kell tudnunk írni. A leíráshoz modelleket készítünk, amelyek jellemzője, hogy egyszerűsítik a problémát: egy modellben mindig elhagyjuk a számunkra kevésbé fontos jellemzőket, és csak a számunkra fontos kérdésekkel foglalkozunk.

A leckében be mutattunk néhány gyakori rendszermodellt is.

3.3.2 Önellenőrző kérdések

1. Hol helyezkedik el a modellezés a szoftverfolyamatban?
2. Mik a jellemzői az adatfolyammodellnek?
3. Mik a jellemzői az ETK-modellnek?
4. Milyen jellemzői vannak az objektummodellnek?

4. LECKE: SZOFTVERTERVEZÉS

4.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A nagy rendszerek fejlesztésekor – és így természetesen tervezéskor is – kisebb alrendszerekre kell bontani a komplex rendszert. Az alrendszerek felismerésekor meg kell állapítani, hogy azok hogyan képesek együttműködésre, és hogyan építik fel a komplex rendszert. Ezt a tervezést architekturális tervezésnek nevezzük. A fő komponenseket és a közöttük zajló kommunikációt egy keretrendszerbe foglaljuk. Ennek az az előnye, hogy a rendszer tervezésének már a korai szakaszában is a kulcsfontosságú komponensek lesznek a fókuszban. Az architekturális tervezés így lehetősége biztosít arra, hogy a majdani felhasználókkal a kezdeti tervezési szakaszban is együtt tudjanak működni a fejlesztők, illetve ez a keretrendszer a tervezés terveként is funkcionálhat.

Az alrendszerek tervezése lényegében a rendszer majdani komponenseinek vázlatos megadása. Az alrendszerek kapcsolatát blokkdiagramokkal, folyamatábrákkal tudjuk leírni. Ennek az a hátránya, hogy a blokkdiagramokban található dobozokat (mint az alrendszerek megfelelőit) összekötő nyilak nem adnak semmilyen információt az alrendszerek közötti kapcsolat jellegéről. Azonban mégis jól használhatók a tervezésben, mert segítségükkel egyszerűen szemléltethetők egy rendszer legalapvetőbb részei és a közöttük értelmezhető függőségi viszonyok.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező képesség, figyelem-összpontosítás.

4.2 TANANYAG

4.2.1 Architekturális tervezés

Egy összetett rendszer alrendszerekre bontása, az egyes alrendszerek felismerése bonyolult feladat. Az architekturális tervezés során ilyen és ehhez hasonló kérdésekre kell választ adnunk:

- Létezik-e olyan, már kész architektúra, amely felhasználható a jelen rendszer tervezéséhez is?
- Hogyan kell felbontanunk a rendszert folyamatokra? Szóba jöhet-e elosztott, vagy többprocesszoros rendszerek használata?
- Milyen alapvető megoldások használhatók a rendszer strukturálására?

- Hogyan lehet a rendszer szerkezeti egységeit modulokra bontani?
- Hogyan lehet az egyes alrendszerek közötti vezérlést megvalósítani?
- Hogyan, milyen módszerrel fogják majd az architekturális tervet kiértékelni, és hogyan lehet azt dokumentálni?

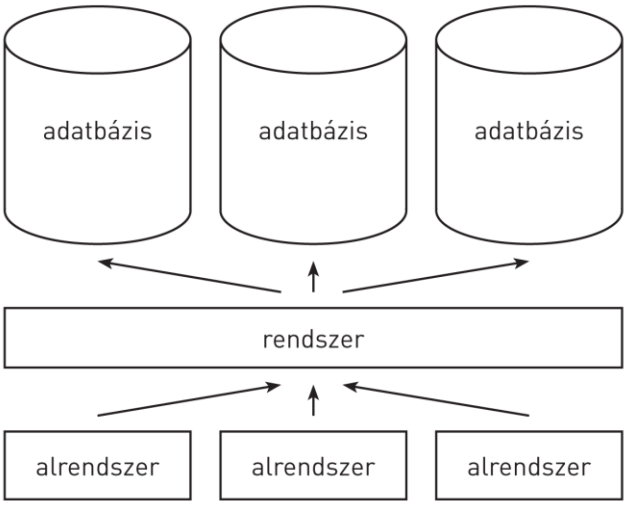
Ha olyan szoftvert fejlesztünk, amely egy bizonyos szakterülethez tartozik, és ehhez a szakterülethez már léteznek működő rendszerek, akkor azok architektúrái kiindulópontként szolgálhatnak. Meg kell vizsgálnunk, hogy az általunk fejlesztett rendszerben mik azok a pontok, amelyek azonosak lehetnek egy már működő rendszerben található elemekkel, és melyek azok az általános jellemzők, információk az adott szakterületen, amelyek felhasználhatók egy új rendszer kifejlesztésében.

4.2.2 A rendszer felépítése

Tárolási modell

A nagy rendszerek általában nagy mennyiségű adatot tárolnak és kezelnek. Ezeket az adatokat elosztott adatbázisokban tárolják, így az egyes alrendszerek közvetlenül hozzáférnek a szükséges adatokhoz. A megosztott tárolás előnyei között említhetjük, hogy az adatok megosztása elkerülhetővé teszi azt, hogy az alrendszerek között tényleges adatmozgatást kelljen megvalósítani a kommunikáció során. Az alrendszereknek ehhez azonos tárolási modellt kell használniuk, ami szükségszerűen azt igényli, hogy az alrendszerek lemondjanak a nagyon specifikus igényeikről az adatmodell tekintetében. Ha egy olyan új alrendszert kell a már működők mellé integrálni, amely nem követi a már működők által használt adatmodellt, akkor az integrálás nagyon nehéz feladat is lehet.

Minden alrendszernek figyelnie kell arra, hogy az általa szolgáltatott adatokat a többi alrendszer hogyan fogja felhasználni. Ahogyan az adatok mennyisége nő, a szoftver evolúciója során esetleg szükségessé váló új adatmodell bevezetése bonyolult lehet. Előny viszont, hogy ha minden alrendszer azonos adatmodellt követ, akkor az adatok karbantartását nem kell minden alrendszerben implementálni, elegendő egyetlen, a karbantartási funkciókat (biztonsági mentés, archiválás stb.) ellátó alrendszert készíteni.

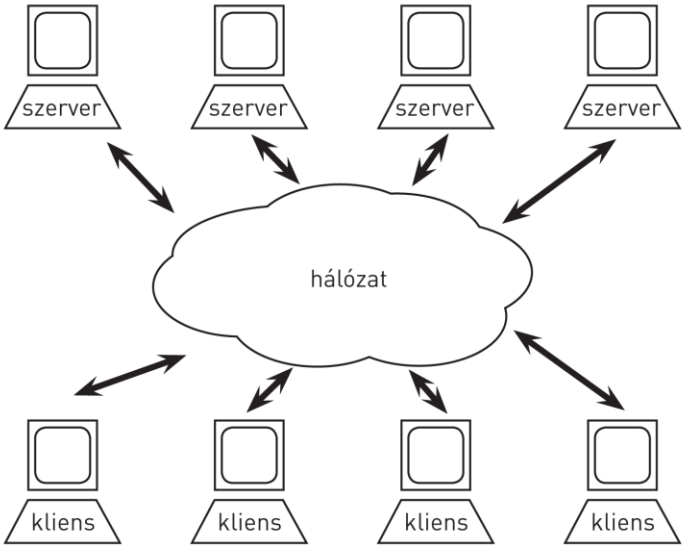


14. ábra: Tárolási modell

Kliens-szerver modell

A hálózatok, és főként az internet fejlődésének következménye, hogy egyre több olyan rendszer jelenik meg, amelyik ezt az architektúrát követi. A kliens-szerver modell lényege, hogy a rendszerben szervereket helyeznek el?, amelyek különféle szolgáltatásokat nyújtanak. Ezeket a szolgáltatásokat a kliensek veszik igénybe. A kliensek tudják, hogy a szerverek milyen szolgáltatásokat nyújtanak, és azokat hogyan lehet igénybe venni (ld. protokollok), de a szerverek nem tudnak semmit sem a kliensekről. Amikor egy kliens szeretne valamilyen szolgáltatást igénybe venni, kérést küld a szervernek. A szerver ezt nyugtázza, majd végrehajtja a kért szolgáltatást, és ha ennek van valamilyen eredménye, akkor azt elküldi a kliensnek. A szerverek tehát nem keresik a klienseket, hanem várnak, amíg egy kliens meg nem szólítja őket. Ekkor elvégzik a kért tevékenységet, majd ismét várakozni kezdenek.

Ennek a modellnek a legnagyobb előnye az erőforrások eloszthatósága. A különböző szerverek jelenthetnek különböző számítógépeket is, de a szerver kifejezés alapvetően olyan folyamatot (tehát szoftvert) jelent, amely várja a kliensek kéréseit és teljesíti azokat. Egy rendszerben integrált szerver lehet például egy nyomtatószerver, ami a kliensektől érkező nyomtatási igényeket teljesíti, vagy egy e-mail szerver, amely a kliensektől érkező elektronikus leveleket továbbítja, illetve tárolja a beérkező üzeneteket addig, amíg a kliensek le nem töltik azokat.

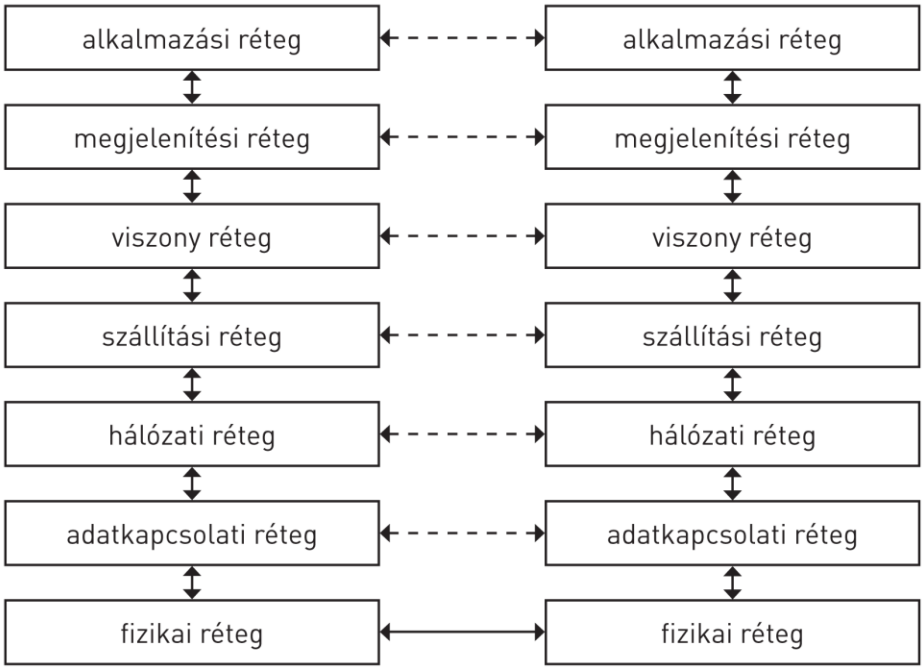


15. ábra: Kliens-szerver modell

4.2.3 A rétegezett modell

A hálózatok működésének példájánál maradva, a különböző típusú hálózati kommunikációs folyamatok leírása meglehetősen bonyolult feladat. Az interneten zajló kommunikációk jellemzően kliens-szerver típusú folyamatok. Ahhoz, hogy a kommunikáció mindkét fél számára feldolgozható legyen, részletesen tisztázni kell a kommunikációs folyamat egyes részeit. Ne felejtsük el, hogy az internet egy nyílt hálózat, vagyis bármilyen hardver részt vehet az internetes kommunikációban, ha megfelel a kommunikációs folyamat szabályainak.

Nyílt rendszerek összekapcsolását írja le az ún. OSI-modell (Open System Interconnection), mint az egyik legismertebb rétegezett modell. Ebben az internetet használó számítógépek közötti kommunikációt 7 rétegben írjuk le. Mind a hét rétegnek részletes specifikációja van. Az egyes rétegek közvetlenül az alattuk/fölöttük lévő rétegekkel kommunikálnak, de a logikai kommunikáció a két résztvevő azonos szinten lévő rétegei között jön létre. Fizikai kommunikáció csak a legalsó rétegben, a fizikai rétegben valósul meg. Ezen a szinten gyakorlatilag a magasabb szinteken előállított adatok fizikai jelekké alakított megfelelőinek továbbítása a feladat, a kommunikáció résztvevői között csak ezen a szinten jön létre valódi összeköttetés. A magasabb rétegek feladata, hogy a kommunikáció során küldendő információkat becsomagolják és kódolják olyan formában, hogy a „túloldalon” lévő szereplő azokat dekódolhassa és kicsomagolhassa úgy, hogy a küldött információ feldolgozhatóvá váljon.

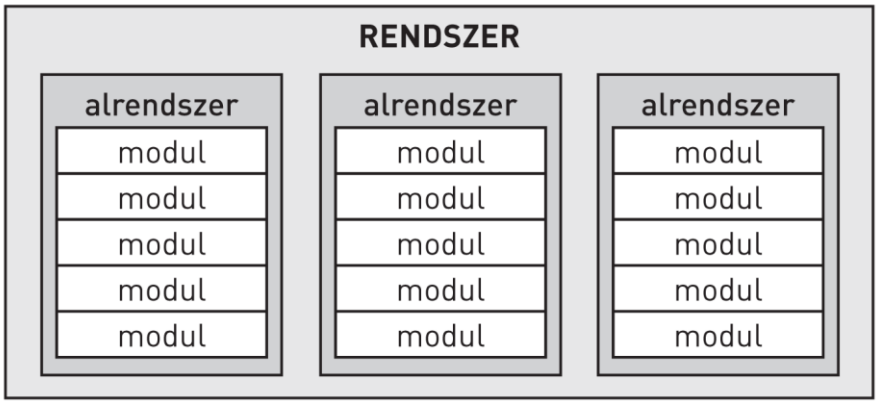


16. ábra: Az OSI referenciamodell

A rétegezett modell hátránya, hogy az egyes alrendszerek strukturálása igen bonyolult is lehet. Az OSI referenciamodell is jó példa arra, hogy egy a tervezésben jól használható módszer nem minden esetben használható módosítás nélkül az implementációhoz is. Az OSI modell gyakorlati felhasználására nem találunk példát, a különböző operációs rendszerek az OSI referenciamodelltől többé-kevésbé mindig eltérnek a konkrét megvalósítás során.

4.2.4 Alrendszerek modulokra bontása

Nem lehet egyértelműen meghatározni, milyen rendszeregységet tekinthetünk még alrendszernek és milyen egységet már modulnak, vagy fordítva. Általánosságban elmondhatjuk, hogy egy rendszer alrendszerei egymástól függetlenül működnek, és modulokból állnak. Az alrendszerek egymással az interfészeiken keresztül kommunikálnak (ld. 5. lecke).



17. ábra: Rendszer – alrendszer – modul

A modulok olyan egységek, amelyek más modulokkal kommunikálnak, és más modulok számára biztosítanak szolgáltatásokat.

Az alrendszerek modulokra bontásának egyik lehetősége az objektumorientált megközelítés, amelyről az 5. leckében részletesen olvashatunk. Az objektumorientált megközelítés előnye, hogy az objektumok egymástól függetlenek, egymáshoz csak lazán kapcsolódnak, így egy objektum kicserélése a teljes rendszer működését nem befolyásolja hátrányosan (rendszerint sehogyan sem befolyásolja). Az egyes objektumok gyakran más-más rendszerekben is használhatók, így egy már kész rendszerben elkészített objektumok egy új rendszer implementálásakor újrafelhasználhatók.

Az alrendszerek modulokra bontásának másik lehetősége a korábban már ismertetett adatfolyammodellre épül. Ebben az adatok egy meghatározott irányban haladnak a rendszerben, az egyes állomások pedig átalakítják a hozzájuk beérkező inputokat és outputként továbbítják azokat. Ezek az állomások képviselik az egyes feldolgozási lépéseket. A transzformációk sorosan és párhuzamosan is végrehajthatók.

Ebben a modellben is előnyként említhető, hogy az egyes transzformációk újrafelhasználhatók. A transzformációs lépések megértése könnyű, mert az emberek munkáját is vizsgálhatjuk abból a szemszögből, hogy milyen inputokat kapnak és azokat hogyan és milyen outputokká alakítják át. Ha egy már működő rendszert egy új transzformációs lépéssel kell bővíteni, az viszonylag egyszerűen végrehajtható.

A modell hátránya az, hogy a transzformációs lépéseknek közös interpretációt kell használniuk, tehát vagy minden transzformációs lépés azonos adatátviteli formátumot használ, vagy a szomszédos transzformációknak egyeztetniük

kell az adatok formátumát. Egy lehetséges megoldás, ha minden adatot karaktersorozatnak tekintünk, és minden bemenetet és kimenetet is ekként kezelünk, de ez ronthatja a feldolgozás hatékonyságát.

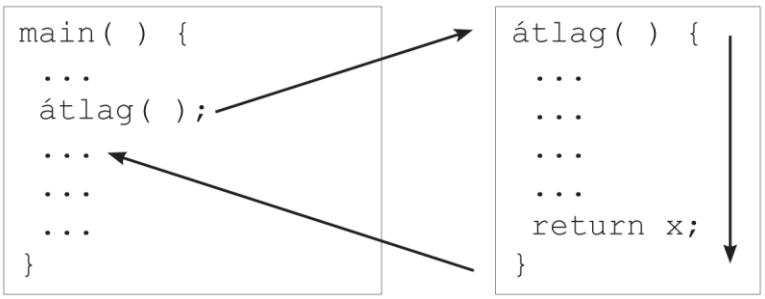
4.2.5 Alrendszerek közötti vezérlés

Egy összetett rendszer alrendszereit úgy kell vezérelni, hogy az általuk nyújtott szolgáltatások a megfelelő időben a megfelelő helyre juthassanak el. A strukturális modellek nem tartalmaznak információkat az alrendszerek vezérlésével kapcsolatosan, ezeket külön vezérlési modellek írják le.

Központosított vezérlés esetén minden vezérlési funkciót egyetlen alrendszer lát el. Ez indítja be és állítja le a többi alrendszert, ez vállal felelősséget minden alrendszer működéséért.

Eseményalapú vezérlés esetén minden alrendszer reagálhat a vele kapcsolatosan bekövetkező eseményekre. Események a rendszer környezetéből (pl. az operációs rendszertől), vagy más alrendszerektől érkezhetnek.

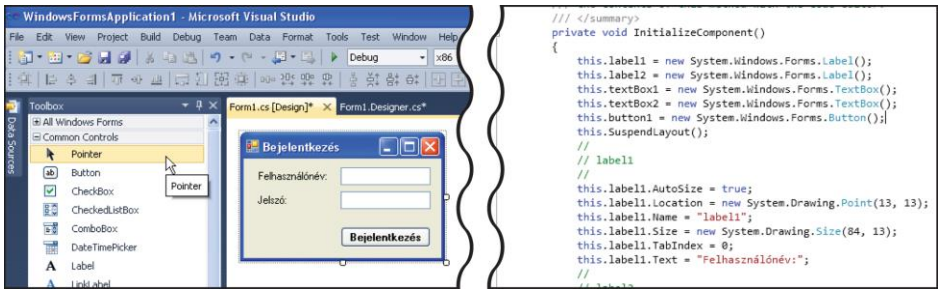
A központosított vezérlést vallja a legtöbb procedurális nyelv (C, Ada, Pascal), de objektumorientált nyelvekben is létezik. Ezekben a programozási nyelvekben alprogramokat hoz létre a programozó, a program indítása a fő alprogramon (főprogramon) kezdődik. Ez hívja meg az egyes alprogramokat, amelyek már alprogramokat indítanak stb. Amikor egy alprogram lefut, a vezérlés visszakerül a hívó programrész következő utasítására.



18. ábra: Alprogramok hívása

Eseményalapú vezérléssel találkozhatunk például a vizuális programozási nyelvekben. A vizuális programozási nyelvekben a programozó számos eszközt kap, amelynek segítésével gyorsan el tudja készíteni egy alkalmazás (rendszerint grafikus) felhasználói felületét. A felületen vezérlőkomponenseket (nyomógombokat, menüket, listamezőket stb.) helyez el, amelyeken a felhasználói interakció eredményeként következhetnek be események (pl. a felhasználó

rákattint egy gombra vagy kiválaszt egy menüpontot). Ezek a rendszerek magas szinten támogatják az eseményeknek a kezelését, a programozó feladata – némileg leegyszerűsítve – csak annyi, hogy elkészíti a felhasználói felületet, majd megírja az egyes eseményekhez kapcsolódó programrészeket. Az ilyen programozási módszer hasonlít a szkriptnyelveken történő programozáshoz: a programozó nem egy komplett programot készít, hanem sok, az egyes vezérlőkomponenseken bekövetkező eseményeket kezelni képes programrészletet.



19. ábra: Vizuális programozási nyelvben készült felület és egy komponenséhez tartozó kód

4.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

4.3.1 Összefoglalás

A bonyolult rendszereket alrendszerekre kell bontani annak érdekében, hogy az összetett feladatok hatékonyan megoldhatók legyenek. Az alrendszerek strukturálásának többféle megközelítése létezik. A rendszerek felépítését vizsgálhatjuk abból a szemszögből, hogy milyen (közös) adatokat használnak az egyes alrendszerek, de tekinthetünk úgy is a problémára, hogy hogyan lehet elosztani az egyes alrendszerek között a megoldandó feladatokat.

A tervezés során itt is modellekkel dolgozunk. Az alrendszerek tervezésekor azt kell vizsgálnunk, hogy az egyes alrendszerek hogyan kapcsolódnak egymáshoz, és közöttük milyen kommunikáció zajlik. A rendszerek architekturális tervezésekor nem kell megadni az alrendszerek közötti vezérlés sajátosságait, de természetesen ez a kérdés sem hagyható figyelmen kívül. A vezérlés leírásához külön modellt használhatunk.

4.3.2 Önellenőrző kérdések

1. Milyen jellemzői vannak a tárolási modellnek?
2. Mi jellemzi a kliens-szerver modellt?

3. Mik a jellemzői a rétegezett modellnek?
4. Mi a különbség alrendszer és modul között?
5. Hogyan írható le az alrendszerek közötti vezérlés?

5. LECKE: AZ OBJEKTUMORIENTÁLT TERVEZÉS

5.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A programozási nyelvek fejlődése során a hetvenes években jelent meg az az irányzat, vagy inkább paradigma, amely ma a legnagyobb hatással van a szoftverek implementációjára. Az objektumorientált programozás (OOP) a ma leginkább használt technológia a programozók körében. Egy szoftverfejlesztéssel foglalkozó cégnél éppen ezért nemcsak a programozóknak kell ismerniük az OOP-t. Az alapvető ismeretekkel minden, a fejlesztésben részt vevő szakembernek rendelkeznie kell.

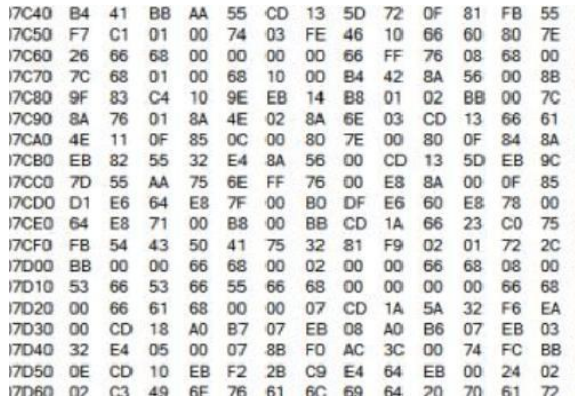
Ez a lecke az objektumorientált tervezéssel foglalkozik. Ahhoz, hogy ezt a metódust jobban megismerhessük, először is vizsgáljuk meg az OOP az alapjait!

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező képesség, figyelem-összpontosítás.

5.2 TANANYAG

5.2.1 Programozás régen és ma

A programok utasításait a számítógép processzora (CPU) hajtja végre. Minden processzornak saját utasításkészlete van. Ahhoz, hogy a processzor számára közvetlenül végrehajtható programot írhassunk, ezt az adott processzor utasításkészletének felhasználásával kell megtennünk. Az ilyen programokat nevezzük gépi kódnak. Tudjuk, hogy az éppen futó programokat, illetve a futásukhoz szükséges adatokat a memória tárolja. A memóriában minden adat, így a programok utasításai is binárisan (kettes számrendszerbeli) vannak kódolva, tárolva. Ha tehát olyan gépi kódot akarunk készíteni, amelyet a processzor közvetlenül végre tud hajtani, gyakorlatilag mindent számok formájában, mégpedig bináris számok formájában kell kódolnunk.



20. ábra: Gépi kód (itt hexadecimálisan)

Ez meglehetősen kényelmetlen, de a magas szintű programozási nyelvek megjelenése előtt sokáig ez volt az egyetlen megoldás a programozók számára. A munkát később valamelyest könnyítette az assembly nyelv(ek) megjelenése. Az assembly lényegében a gépi kód egy áttekinthetőbb, az ember számára kicsit könnyebben írható/olvasható jelölésmódja. Az utasításokat néhány betűs rövidítések (ún. mnemonikok) jelölik, ezek mögött állnak az utasításhoz tartozó adatok, az operandusok.

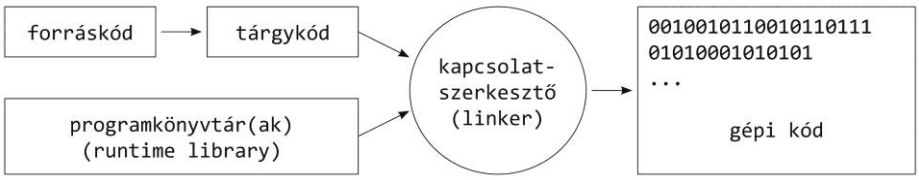
```
START: JUMP   LOOP       # jump past sensor and constant locations
RSV:   0000              # read right sensor value here
LSV:   0000              # read left sensor value here
RMP:   0000              # write right motor power level here
LMP:   0000              # write left motor power level here
OFF:   0000              # store motor-off constant here
ON:    0100              # store motor-on constant here
LOOP:  LOAD   1     RSV  # load right sensor value into register 1
       LOAD   2     LSV  # load left sensor value into register 2
       SUB    1  2  3    # subtract 1 from 2 and store result in 3
       LOAD   1     OFF  # load motor-off constant into register 1
       LOAD   2     ON   # load motor-on constant into register 2
       BRANCH 3     RGT  # if the left sensor is greater than the
LFT:   STORE  2     RMP  # right then turn the right motor on
       STORE  1     LMP  # and turn the left motor off
       JUMP   LOOP       # and then jump to beginning of the loop
RGT:   STORE  2     LMP  # else turn the left motor on
       STORE  1     RMP  # and turn the right motor off
       JUMP   LOOP       # and then jump to beginning of the loop
```

21. ábra: Assembly nyelvű kód megjegyzésekkel

Az assembly ugyanúgy hardverfüggő, mint a gépi kód, hiszen lényegében az adott processzor utasításkészletének felhasználásával készült gépi kódról van

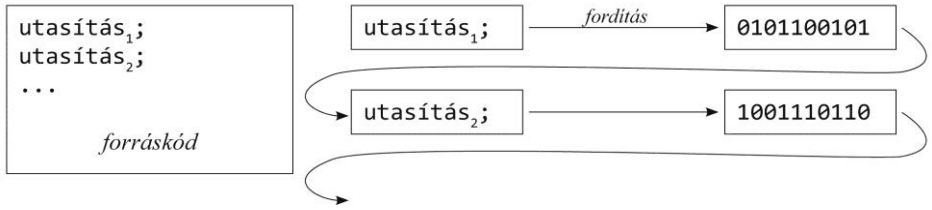
szó, az ember számára áttekinthetőbb, olvashatóbb jelölésmóddal. Azonban a mnemonikokat a processzor közvetlenül nem tudja értelmezni és végrehajtani, valamilyen eszköznek (egy másik programnak) az assembly kódot le kell fordítania gépi kódra. Az ilyen fordítóprogramokat nevezik assemblereknek.

Az 1950-es évek végén megjelentek az ún. magas szintű programozási nyelvek. Ezek általános jellemzője az, hogy a programozó ezeknek a nyelveknek a használatával úgy írhat programot, hogy a kódoláshoz a beszélt emberi nyelv (jellemzően az angol nyelv) szavait, kifejezésit, vagy ahhoz nagyon hasonló nyelvi szerkezeteket használhat. A magas szintű programozási nyelvekben parancsokat, utasításokat használhatunk, közvetlenül kezelhetünk összetett adatokat anélkül, hogy tudnunk kellene azt, hogy ezek az adatok hogyan (és hol) jelennek meg a memóriában. A magas szintű programozási nyelveken írott programot a program forráskódjának nevezzük. Ezt a processzor természetesen nem tudja közvetlenül végrehajtani, tehát ugyanúgy, mint az assemblyk esetében, a magas szintű nyelveken írott forráskódokat is gépi kóddá kell transzformálni. A transzformáció történhet fordítással, vagy értelmezéssel. A fordítás során a forráskódot teljes egészében gépi kóddá (esetleg egy másik magas szintű nyelvű kóddá) alakítja a fordítóprogram. Ha a forráskód hibás, akkor a fordító által felfedhető hibákról egy hibalistát állít össze a program és a fordítás nem történik meg.



22. ábra: A fordítás elvi vázlata

Az értelmezés során nem a teljes forráskódból készül gépi kód, csak a soron következő utasításból vagy parancsból. Az értelmező programok szorosan együttműködnek a futtató rendszerrel, így az értelmezés során a forráskód egy-egy utasítását transzformálják gépi kóddá, majd az elkészült gépi kódot a futtató rendszer azonnal végre is hatja. Ha a forráskódban hiba van, az csak akkor derül ki, ha a vezérlés erre a hibás utasításra kerül.

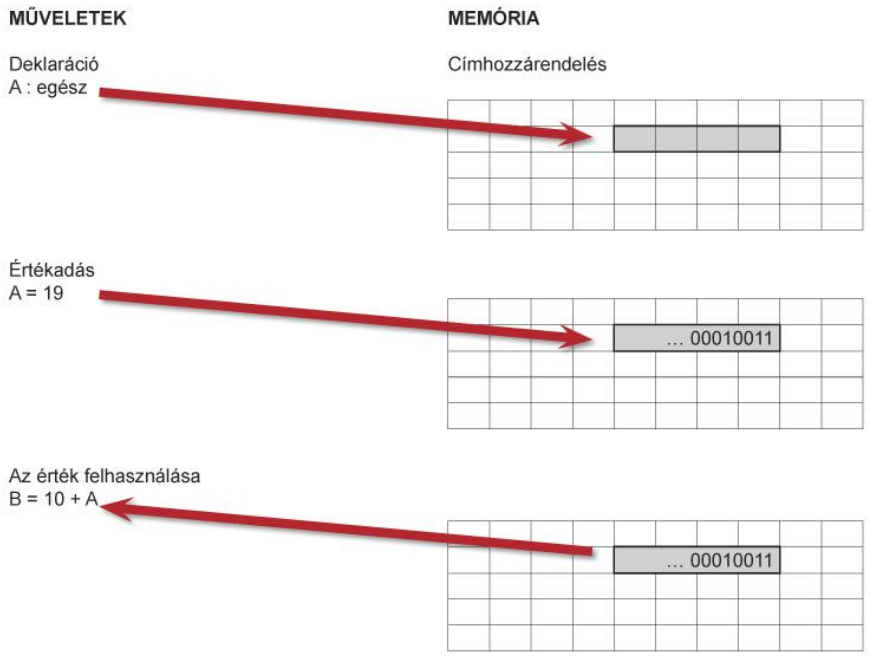


23. ábra: Az értelmezés elvi vázlata

5.2.2 Magas szintű programozási nyelvek csoportosítása

Imperatív, procedurális nyelvek

A magas szintű programozási nyelvek evolúciója során elsőként az imperatív, más néven procedurális nyelvek jelentek meg. Ezek használatakor a programozó utasításokat, parancsokat használ, a memóriában tárolt adatokat pedig ún. változókon keresztül éri el. A változók olyan programozási eszközök, amelyek a memória egyes területeihez nevet rendelnek. Amikor a programozó szeretne valamilyen adatot menteni a memóriában, az adott memóriaterülethez rendelt változónak adja értékül ezt az adatot. Amikor valamilyen adatot szeretne kiolvasni a memóriából, akkor a memóriaterülethez rendelt változó értékét kérdezi le. A címhozzárendelés történhet explicit módon, de jellemzőbb, amikor maga a fordító vagy futtató rendszer rendeli hozzá a memóriacímeket a változóhoz.

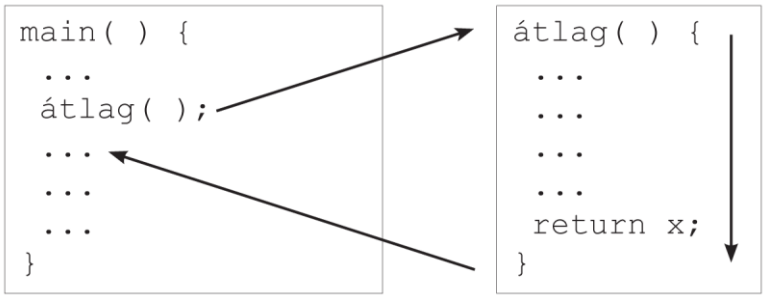


24. ábra: Változó

A programozónak így nem kell közvetlenül hozzáférnie a memóriához és nem kell foglalkoznia olyan kérdésekkel sem, mint például az adatkódolás, számábrázolás stb. A magas szintű programozási nyelveket használó programozóknak rendszerint nem kell törődniük egy adott típusú adat tárbeli reprezentációjának problémájával.

A procedurális nyelvek használata mellett a programozó feladata az, hogy a program számára bejövő adatokból (input) kimenő adatokat (output) állítson elő, és pontosan, algoritmikusan leírja azt, hogy ez hogyan történjen meg. Az imperatív nyelvek tehát algoritmikus nyelvek, a probléma megoldásának algoritmusát az adott nyelv nyelvi szabályainak (szintaxisának) betartásával kell leírnunk.

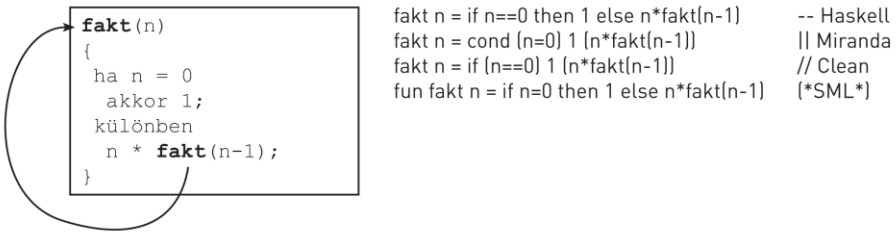
A procedurális nyelvek fejlődése során alakult ki az ún. strukturált programozás. Ennek alkalmazása azt jelenti, hogy a programozó a bonyolult feladatokat részekre tudja bontani, és az összetett problémánál jobban kezelhető méretű részfeladatokhoz tud algoritmust implementálni. Az ismétlődő műveleteket kiemelheti a programból, ezekből alprogramokat hozhat létre, és amikor egy ilyen gyakran ismétlődő tevékenység elvégzésére van szükség, akkor nem kell leírnia az utasításokat, csak hivatkoznia kell a kiemelt alprogramra.



25. ábra: Alprogram

Funkcionális, applikatív nyelvek

Ezek a nyelvek szintasxisukban a formális matematika jelölésmódját követik. Ezt az alapelvet követve a funkcionális nyelvet tisztán függvényekre alapozzák. Az ilyen nyelvekből hiányoznak az imperatív nyelvekben megszokott vezérlési szerkezetek (iteráció, szelekció), műveletként csak a függvények összehasonlítása, feltételes kifejezés és rekurzió áll a programozó rendelkezésére.



26. ábra: A rekurzió elve és a faktoriális függvény megvalósítása néhány funkcionális nyelvben

Logikai nyelvek (szabály alapú nyelvek)

Ezek a programozási nyelvek a formális logika és a halmazelmélet alaptörvényeire épülnek. A logikai programban formális szabályokat definiálhatunk, a program feladata olyan eset keresése, amely az adott kérdésre a programból logikailag következik.

27. ábra: Programrészlet Prologban

Objektumorientált nyelvek

Az objektumorientált program egymással kölcsönhatásban álló objektumokból áll. Az objektumok a valós világot modellezik. Az objektumok a világban található dolgokat írják le, minden objektum ismeri a saját tulajdonságait (jellemzőit), és az adott környezet, szituáció által meghatározott módon viselkedik. Az objektumok típusosak, tehát minden objektumnak vannak általános jellemzői, az egyes objektumok a konkrét jellemzőikben és viselkedésükben térnek el egymástól.

Az objektumorientált nyelvek a procedurális nyelvekből alakultak ki. A programozási nyelvek fejlődése során a nyelvek újabb és újabb kihívásoknak kezdtek megfelelni, történetüket tekintve először objektum alapú, majd az osztály alapú, végül az objektumorientált nyelvek alakultak ki.

Az objektumalapú nyelvek támogatják az objektumokat, de nem létezik az osztály fogalom. (erről a későbbiekben még lesz szó).

Az osztályalapú nyelvekben megjelenik az osztály, de nincs osztályhierarchia.

A ma legfontosabb nyelvek közé tartozó objektumorientált nyelvekben létezik az osztály és az objektum, az osztályok pedig hierarchiába sorolhatók az osztályokon értelmezett öröklés, öröklődés révén.

5.2.3 Az OOP paradigma

A valós világ tárgyait, dolgait az ember leegyszerűsíti, megkülönbözteti és rendszerezi annak érdekében, hogy megértse a körülötte lévő világot. A végső cél tehát a bonyolult világ megismerése, az ehhez felhasznált eszköz pedig a modellezés – ahogyan erről egy korábbi fejezetben már esett is szó.

A modellezés során olyan algoritmust használunk, amelynek segítségével elvonatkoztatunk, megkülönböztetünk, osztályozunk, általánosítunk vagy leszűkítünk, részekre bontunk, kapcsolatokat építünk fel stb. Absztrakciónak nevezzük azt a tevékenységet, amelynek során a cél elérése érdekében csak a feltétlenül szükséges részek megértésére, felderítésére törekszünk. Elvonatkoztatunk a számunkra pillanatnyilag nem fontos információktól, és kiemeljük az elengedhetetlenül fontos dolgokat.

Az objektumorientált szemléletben az objektumok a modellezendő világ egy-egy önálló egységét jelölik. Az objektumokat meghatározzák a saját jellemzőik, belső állapotuk, illetve a külvilágtól érkező üzenetekre való válaszaik, reakcióik. Egy objektum reakciója egy üzentre lehet az, hogy megváltoztatja saját belső állapotát, vagy ő is üzenetet küld egy másik objektumnak, vagy létrehoz, esetleg töröl vagy megváltoztat más objektumokat.

Az objektumok megkülönböztethetők a leglényegesebb tulajdonságaik, vagy viselkedésmódjuk szerint. A hasonló objektumok egy osztályba, a különböző objektumok más-más osztályokba kerülhetnek. Az osztályozás az általánosítás és a specializálás révén valósul meg. Az osztályozás nem állandó állapot. Az objektumok között állandóan különbségeket és hasonlóságokat keresünk és tárunk fel, így szűkebb vagy tágabb osztályokba soroljuk őket.

Az objektumosztályok hordozzák a hozzájuk tartozó objektumok jellemzőit. Az objektumok egy bizonyos osztálynak a konkrét képviselői, példányai. Az objektum rendelkezik a saját osztálya tulajdonságaival és viselkedési módjaival. Az objektum információt tárol és feladatokat hajt végre, vagyis az objektum nem más, mint adatok (jellemzők) és műveletek (módszerek, más szóval metódusok).

A objektumorientált programozás (OOP) alappillérei következők:

- egységbe zárás (encapsulation)
- öröklés (inheritance)
- többalakúság (polymorphism).

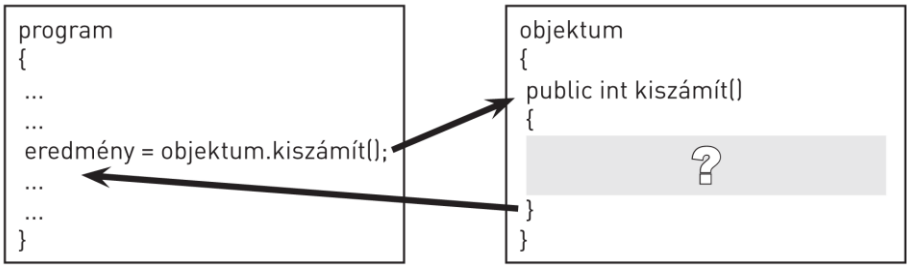
Egységbe zárás

A strukturális programozás kialakulása magával hozta a program strukturálhatóságának lehetőségét, ami azt jelentette, hogy a programból kiemelhetővé váltak az ismétlődő részek, és a sokszori, ismételt leírás helyett ezekre a programrészekre elegendő lett csak hivatkozni. Ez elsősorban algoritmizálási szempontból javította a kód minőségét.

A hagyományos procedurálási nyelvek az adatokat és a rajtuk végzett műveleteket (az implementált algoritmusokat) lényegében különálló egységekként kezelték. A program bizonyos egységeiben leírtuk a programban előforduló adatok, adatszerkezetek jellemzőit, a program egy másik részében pedig megadtuk az adatokat kezelő algoritmusokat.

Az OOP egyik döntő változása ezen a téren jelentkezett. Az objektumok egységbe zárják az őket jellemző adatokat és a rajtuk végezhető műveleteket, ezáltal az adat és a művelet zárt egységet alkothat. Ezen felül az OOP lehetőséget biztosít arra is, hogy egy objektum elrejtse a saját belső állapotát a külvilág elől. Egy objektumtól általában azt várjuk, hogy egy üzenet hatására a megfelelő reakciót adja. Az üzenetküldő szemszögéből az a fontos, hogy a helyes reakció szülessen meg, de hogy ez miként történik meg, az rendszerint nem érdekli az üzenet küldőjét. Az objektumok belső állapota a saját belügyük, nem is lenne jó, ha abba a külvilág (más objektumok) beleszólhatnának.

Az egységbe zárás tehát lehetőséget ad az adatelrejtésre is.



28. ábra: A hívó programegység nem „lát bele” a hívott programegység belsejébe

Öröklődés

Az objektumorientált megközelítésben a valós világ dolgait próbáljuk modellezni, ehhez az objektumokat osztályokba soroljuk. Az osztályozás történhet úgy, hogy először a legalapvetőbb különbségek alapján alakítunk ki osztályokat, majd az osztályozást finomítjuk. Az egyes osztályokban alosztályokat definiá-

lunk, amelyek mind szigorúbb szabályokat mondanak ki a belőlük származó objektumokra, pl.:

Zárt geometriai alakzatokat vizsgálva kialakíthatjuk a síkidomok és a testek osztályait. Minden síkidomnak van kerülete és területe, ahogyan a testeknek is van felszíne és térfogata. Egyelőre ez a két-két biztos ismeret van a birtokunkban, ennél pontosabbat nem tudunk az általunk vizsgált dolgokról.

A síkidomok osztályában kialakítható további két osztály, az egyikben olyan síkidomok lehetnek, amelyeknek minden oldala egy-egy egyenes szakasz (sokszögek osztálya). Ebben az osztályban az objektumok már jellemzőket is kaphatnak, például az oldalaik és/vagy csúcsaik száma, belső szögeik összege stb.

A másik osztályba kerülhetnek azok a síkidomok, amelynek nem csak egyenes szakaszokból (hanem pl. ívekből is) állnak.

A sokszögek osztályából létrehozhatunk olyan alosztályokat, amelyek például az oldalak száma alapján foglalják csoportba az egyedeket: háromszögek, négyszögek, ötszögek stb. osztályai. Ha az osztályokban található objektumok pontosan ismerik a saját oldalaik hosszát, akkor a kerületüket már ebben az absztrakciós szintben meg tudják határozni, hiszen a kerület kiszámítására létezik általános képlet: az oldalak hosszának összege. Természetesen speciális sokszögek, pl. szabályos sokszögek esetében van ennél konkrétabb, egyszerűbb képlet is.

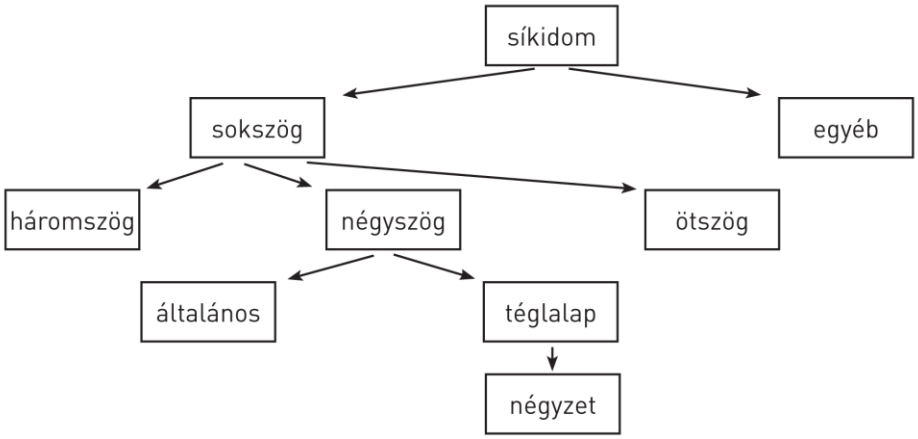
Ha a négyszögek osztályát tekintjük, abból létrehozhatjuk a deltoid, trapéz, paralelogramma, rombusz, téglalap és négyzet alosztályokat. A háromszögek osztályán belül kialakítható az általános háromszög, az egyenlő szárú háromszög, a szabályos háromszög és a derékszögű háromszög alosztály.

Talán érezhető, hogy itt már nem annyira nyilvánvaló az ősosztály-alosztály kérdés. A téglalap egyben trapéz, hiszen van két párhuzamos oldala (a téglalap ennél több, de megfelel a trapéz kritériumainak). A négyzet egyben deltoid, hiszen az átlói derékszöget zárnak be, de paralelogramma, hiszen két-két párhuzamos oldala van, ráadásul egyenlő oldalú paralelogramma, vagyis rombusz. Továbbá téglalap, mivel mind a négy szöge derékszög, ezen felül szabályos sokszög is.

Ebben a példában nem esett szó a sokszögek konvexitásáról, és számos további olyan jellemzőről sem, amely alapján az egyes síkidomok osztályokba sorolhatók lennének. A modellalkotásnak éppen ez az egyik lényege: tekintsük csak azokat a kérdéseket, amelyek a probléma szempontjából fontosak nekünk, a pillanatnyilag nem fontos kérdéseket hagyjuk figyelmen kívül.

A öröklődésre visszatérve: induljunk ki a síkidomok osztályából (legbővebb halmaz), és jussunk el valamilyen úton a négyzetig (legszűkebb halmazok egyike).

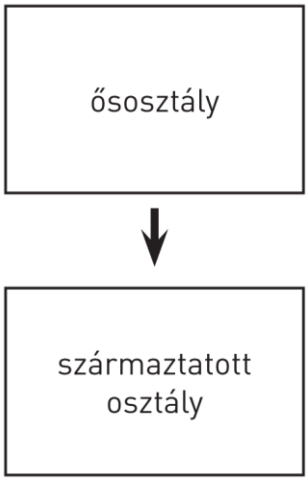
- A síkidomok lehetnek sokszögek. Minden sokszög egyben síkidom.
- A sokszögek között vannak négyszögek. Minden négyszög egyben sokszög, de mivel sokszög, egyben síkidom is.
- Speciális négyszög a téglalap, mert minden belső szöge derékszög. Minden téglalap négyszög, egyben sokszög, egyben síkidom is.
- A négyzet egy olyan speciális téglalap, amelynek minden oldala egyenlő. A négyzetek egyben téglalapok, egyben négyszögek, egyben sokszögek és egyben síkidomok is.



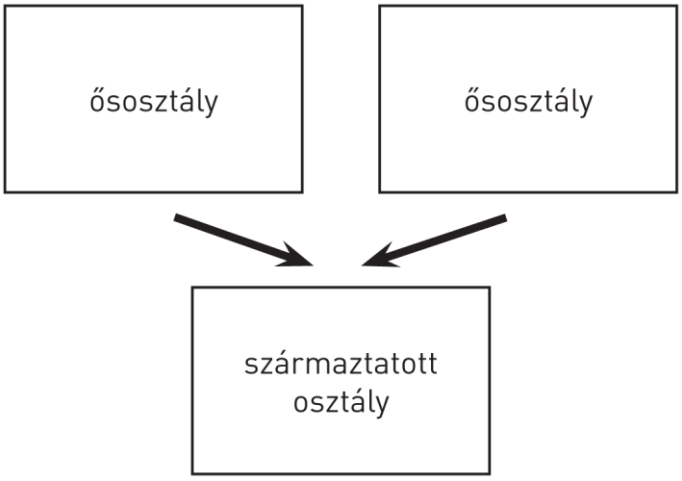
29. ábra: Osztályhierarchia

Az OOP lehetővé teszi, hogy ha egy osztályt már definiáltunk, akkor úgy definiálhassunk az osztályból származtatott alosztályt, hogy az alosztály örökölje az ősosztály minden jellemzőjét. Ha az alosztály specifikusabb, speciálisabb, mint az ősosztály, akkor lehetőség van arra is, hogy az ősosztályban definiált tulajdonságokat, jellemzőket az alosztály felülírhassa

Ezt a folyamatot nevezzük az OOP-ban öröklődésnek. Az objektumorientált programozási nyelvekben két típusú öröklés létezik: az egyszeres és a többszörös öröklés. Egyszeres öröklésről beszélünk, ha minden származtatott alosztálynak pontosan egy ősosztálya lehet. A többszörös öröklés jellemzője az, hogy egy származtatott alosztály több ősosztályból is öröklődhet.



30. ábra: Egyszeres öröklés



31. ábra: Többszörös öröklés

Többalakúság

Az öröklődés miatt gyakran előfordul, hogy egy objektum több típushoz (osztályhoz) is tartozhat. Vannak olyan metódusok, amelyek az ősosztályban vannak leírva, míg mások az alosztály definiálásakor kerültek be az osztályba. Ha például a négyzet osztályt a téglalap osztályból származtatjuk, amely a négyszögek osztály leszármazottja, akkor a kerület, terület kiszámítása attól függően történhet, hogy az adott metódust módosítottuk-e a származtatott osztályban vagy sem. Lehet, hogy a mi megoldásunkban a négyzet osztályban már speciali-

záltuk a terület kiszámítását (a^2), azonban ha a kerület kiszámítását még nem „igazítottuk” a négyzethez, attól a kerület kiszámítható lesz ($2(a+b)$ vagy $a+b+a+b$), annak ellenére, hogy a négyzetek esetében a $b = a$ egyenlőség is fennáll. A kerület kiszámítása tehát több, a négyszögek osztályból származtatott osztályon is igaz (mindegyiken igaz), de a speciális négyszögek felül is írhatják ezt a kiszámítási módot, amennyiben az adott négyszög esetében van másik (pl. egyszerűbb) kiszámítási mód.

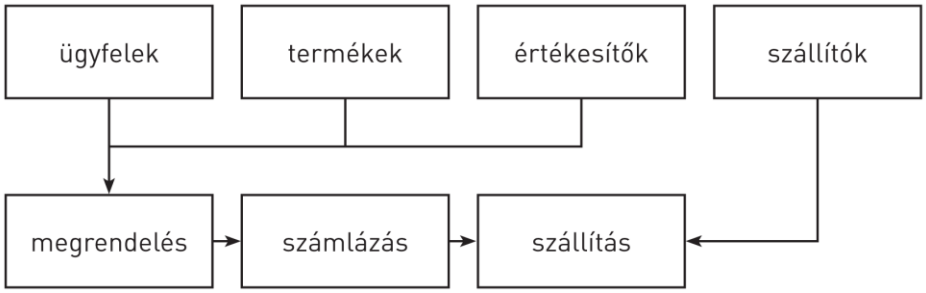
A polimorfizmus – némileg leegyszerűsítve – azt jelenti az OOP esetében, hogy ha egy ősosztályból származtatott alosztályban egy, az őstől örökölt metódust módosítunk, a származtatott alosztályban a módosítatlanul maradt többi, (az ősosztálytól öröklődött) metódus ezt az új változatot automatikusan tudja használni.

5.2.4 Objektumorientált tervezés

Egy objektumorientált rendszer egymással együttműködő objektumokból áll. Az objektumok ismerik saját belső állapotukat, képesek a hozzájuk címzett üzeneteket fogadni és feldolgozni, és az üzenetek hatására műveleteket végezni. Ezek a műveletek lehetnek olyanok, amelyek megváltoztatják az objektum belső állapotát, az üzenet hatására az objektum létrehozhat újabb objektumokat, vagy ő maga is üzenetet küldhet egy másik objektum számára. Leegyszerűsítve: az objektum is úgy működik, hogy a megfelelő inputokat átalakítja megfelelő outputokká. A transzformáció módját (illetve a saját belső állapotát) az objektum elrejti a külvilág elől.

Az osztályok az OOP-ban az objektumok típusát írják le. A konkrét működést az objektumok végzik. Ahhoz, hogy egy objektum működni tudjon, először létre kell hozni. A létrehozás műveletét példányosításnak hívjuk. A példányosítás során az adott osztály alapján létre hozunk egy konkrét objektumpéldányt, egyedet, és beállítjuk annak kezdőértékeit, kezdőállapotát. A kezdőállapotot egy speciális metódus, a konstruktor állítja be. Lényegében a konstruktor feladata az, hogy létrehozza az objektumot.

Az objektumorientált tervezés az osztályok és a közöttük lévő kapcsolatok, illetve interakciók megtervezéséből áll.



32. ábra: Objektumorientált tervezés

Az objektumorientált fejlesztés az objektumorientált tervezést követően kezdődhet meg. Ehhez objektumorientált elemzésre van szükség, amely a vizsgált probléma objektumorientált modelljét alakítja ki. Az objektumorientált programozás a specifikáció, az elemzés és a tervezés során létrejött objektumorientált modell implementációját jelenti, egy (vagy több) konkrét programozási nyelv eszközrendszerének felhasználásával.

Ahogyan a programozási nyelvekről szóló részben leírtuk, a nyelvek evolúciója során többféle szintű OO-támogatottságú nyelv alakult ki. Az objektumorientált tervezés feltételezi, hogy az implementáció is objektumorientált programozási környezetben fog történni.

A tervezés során ugyanazokat a finomítási, specializálási eljárásokat érdemes végiggondolni, amelyekre a fentiekben már láttunk példát. Az osztályok hierarchiájának kialakítása történhet implementáció-független módon is, hiszen egy jól kialakított osztályhierarchiának bármilyen OOP nyelven implementálhatónak kell lennie.

Az egymással együttműködő objektumokból álló programot könnyebb módosítani, mint egy más elven fejlesztett rendszert, mert az objektumok egymástól függetlenül működnek.

Objektumok azonosítása

A tervezésnek ebben a fázisában valójában nem az objektumokat, hanem az őket leíró osztályokat kell azonosítanunk egy megtervezendő rendszerben. A rendszerben található objektumok (osztályok) felismerésére többféle módszer is létezik.

- Érdekességként megjegyezhetjük, hogy ezeket a módszereket többékevésbé a konkrét programozási nyelvekhez ajánlott névadási szabályok is javasolják. A konkrét programozási nyelvekben szigorú szabályok léteznek a különböző programelemek elnevezésére, úgy mint: osztályok,

objektumok, változók, metódusok stb. Az alább bemutatott módszerek a konkrét nyelvekhez (pl. Java) tett elnevezési szabályokban is visszaköszönnek.

1. módszer: Írjuk le a rendszert természetes nyelven, majd elemezzük a leírást. A leírásban szereplő főnevek legyenek a modellünk objektumai és az objektumok attribútumai (tulajdonságai). Az igék legyenek a műveletek és szolgáltatások (metódusok) nevei. (Például a Java névadási ajánlásai ezt a módszert javasolják.)

2. módszer: vizsgáljuk meg a szakterület konkrét elemeit (egyetem), szerepköreit (oktató), eseményeit (zh), interakcióit (vizsga), helyszíneit (számítógépes labor), szervezeti egységeit (tanszék) a modellalkotáshoz.

3. módszer: viselkedési megközelítés, amelynek során a tervező először megérti a rendszer teljes viselkedését. Megfigyeli, hogy a konkrét alrendszerek hogyan viselkednek egymással: ki kezdeményezett egy adott műveletet és abban ki vett részt. A legfontosabb szerepkörrel rendelkező rendszerek lesznek az objektumok.

4. módszer: a forgatókönyv-alapú módszer. A rendszerhasználat különböző forgatókönyveit kell elemezni, az elemzők a forgatókönyvekben ismerik fel és azonosítják az objektumokat.

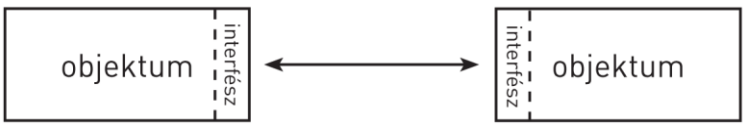
Az objektumok azonosítása iteratív folyamat, és nem feltétlenül írható le hozzá absztrakt eljárás. Először fel kell ismerni az általános összefüggéseket az elkészítendő rendszerhez, majd a kezdetben azonosított objektumok jellemzőit folyamatosan finomítani kell. Egy újabb objektum feltárása a már létezők újraértelmezését teheti szükségessé.

Az OOP paradigmában ismeretesek az ún. absztrakt osztályok. Ezek olyan osztályok, amelyekben csak specifikáljuk az elvégzendő műveleteket, de azokat még nem írjuk le. Korábbi példánkban szerepelt a síkidomok általános osztálya. Ennek a példában két leszármazottját képzeltük el, a csak egyenes szakaszokból álló síkidomokét (sokszögek), illetve azon síkidomokét, amelyeket nem, vagy nem csak egyenes szakaszok alkotnak (pl. kör, ellipszis). Az általános, *absztakt* síkidom osztályban specifikálhatjuk, jelezhetjük, hogy minden síkidomnak van kerülete és területe, de ezek kiszámítási módjára nincs általános képlet vagy eljárás (tekintsünk el most a határozott integrál alkalmazási lehetőségétől, amellyel egy általános síkidom területét is meg tudjuk határozni). A síkidom osztályt tekinthetjük absztrakt osztálynak, mert jelezzük benne, hogy a síkidomoknak van területe és kerülete (ezek az osztály egy-egy metódusai), de ebben az osztályban a konkrét kiszámítási módot nem adjuk meg. A konkrét

kiszámítási módszert majd a síkidom osztályból származtatott alosztályokban fogjuk pontosan leírni, implementálni.

Absztrakt osztálynak nevezünk minden olyan osztályt, amelyben akár csak egy absztrakt metódus is szerepel. Speciális absztrakt osztály az ún. interfész, amelyben nincsenek objektumváltozók (az osztályok jellemzőit leíró változók), csak metódusdeklarációkat tartalmaz. Az interfészek alkalmazásával olyan osztályhierarchia alakítható ki, amelyben a tervezés során egy bizonyos pontig el lehet térni a konkrét megvalósítástól.

Az interfész szó általánosságban a felület szó megfelelője. Olyan felületet jelent, amelyen keresztül két folyamat (vagy két objektum, vagy ember és gép stb.) kommunikálni tud. Ahhoz, hogy a tervezési folyamatban a komponensek tervezése párhuzamosan is történhessen, az egyes objektumok tervezőinek közzé kell tennie az általuk vizsgált, tervezett objektumok kommunikációs felületét, interfészét. Amikor a párhuzamosan zajló fejlesztési folyamat résztvevői találkoznak egy másik csoport által tervezett objektum interfészével, akkor azt feltételezhetik, hogy maga az objektum is implementálva van vagy lesz. Az objektum megvalósítását nem kell (nem szabad) látniuk, a különböző objektumok belső világa rejtve marad a külvilág szeme elől.



33. ábra: Objektumok és interfészeik

Amikor elkészül egy objektumorientált rendszer objektumorientált tervezése, szinte biztos, hogy a terveken utólag módosítani kell, vagy legalábbis erre kaphatunk javaslatokat. Az objektumorientált tervezés nagy előnye, hogy bár az objektumok szorosan együttműködnek, de egymástól mégis függetlenek, önállóak. Ha a síkidom osztályhierarchiát arra használjuk, hogy a segítségével kiszámítsuk egy ház alapterületének nagyságát, maga a rendszer biztosan működőképes marad akkor is, ha a kész tervekben megváltoztatjuk az ellipszisek kerületét kiszámító metódust az ellipszisek osztályban.

5.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

5.3.1 Összefoglalás

Ebben a leckében megismertük a programozási nyelvek fejlődésének történetét és a programozási nyelvek csoportosítását egy bizonyos szempontrend-

szer alapján. Erre azért volt szükség, hogy megismerjük az objektumorientált programozás (OOP) jellegzetességeit, hiszen a lecke az objektumorientált tervezést mutatta be. Habár szoftvereket nemcsak objektumorientált programozási nyelveken fejlesztenek, a különböző rendszerek implementálásában nagyon fontos szerepet kaptak az elmúlt évtizedekben ezek a nyelvek. Az OOP alapjainak ismerete azért is fontos, mert a következő két lecke (felhasználói felületek tervezése és komponensalapú tervezés) feltételezi az itt leírt ismeretek biztos tudását.

5.3.2 Önellenőrző kérdések

1. Mi a különbség a gépi kód és a magas szintű programozási nyelvek használata között?
2. Hogyan csoportosíthatjuk a magas szintű programozási nyelveket? Mik az egyes csoportok jellemzői?
3. Melyek az objektumorientált programozás alappillérei? Ismertesse ezek értelmezését!
4. Hogyan lehet tervezési fázisban egy rendszer objektumait felismerni?

6. LECKE: FELHASZNÁLÓI FELÜLETEK TERVEZÉSE

6.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

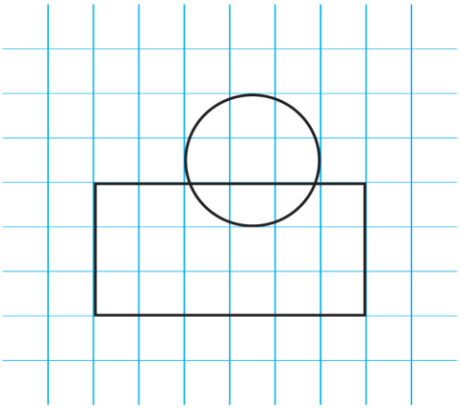
A felhasználók rendszerint nem kíváncsiak arra, hogyan működik beleül a program, ahogyan az átlagos autóvezetőt sem érdekli a motor működésének minden apró részlete. A felhasználó számára az a fontos, hogy tudja használni a szoftvert: tudjon bemenő adatokat küldeni a programnak, és a program azokat az ő igényeinek megfelelően dolgozza fel, majd jelenítse meg. A felhasználó tehát nem közvetlenül a programmal, a szoftver számításokat végző részével áll kapcsolatban, hanem azzal a felülettel, amely a felhasználó és a program között helyezkedik el. Ez a felület a felhasználói felület, ennek tervezési kérdéseivel foglalkozik ez a lecke.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező képesség, figyelem-összpontosítás.

6.2 TANANYAG

6.2.1 Felhasználók elvárásai

Készülő rendszerünket felhasználók fogják használni. Lehet, hogy a felhasználók informatikai előképzettségei hasonlók a fejlesztők képzettségéhez, de valószínűbb, hogy nem. Lehet, hogy a most készülő rendszer egy már működő másik szoftvert fog leváltani, ha elkészülünk vele, de az is lehet, hogy most még minden feladatot papíron végeznek el a megrendelő alkalmazottai. Egy biztos: a szoftvert emberek készítik emberek számára. Végezzünk el egy egyszerű kísérletet! Kérjen meg valakit a környezetében, hogy rajzolja le ezt az ábrát úgy, hogy ő nem láthatja ezt az oldalt, csak az ön útmutatása alapján dolgozhat. Az ön feladata az, hogy mondja el pontosan a rajzolónak, hogy mit, hogyan, mekkorában rajzoljon le (a rajzoló is dolgozhat négyzetrácsos papíron)!



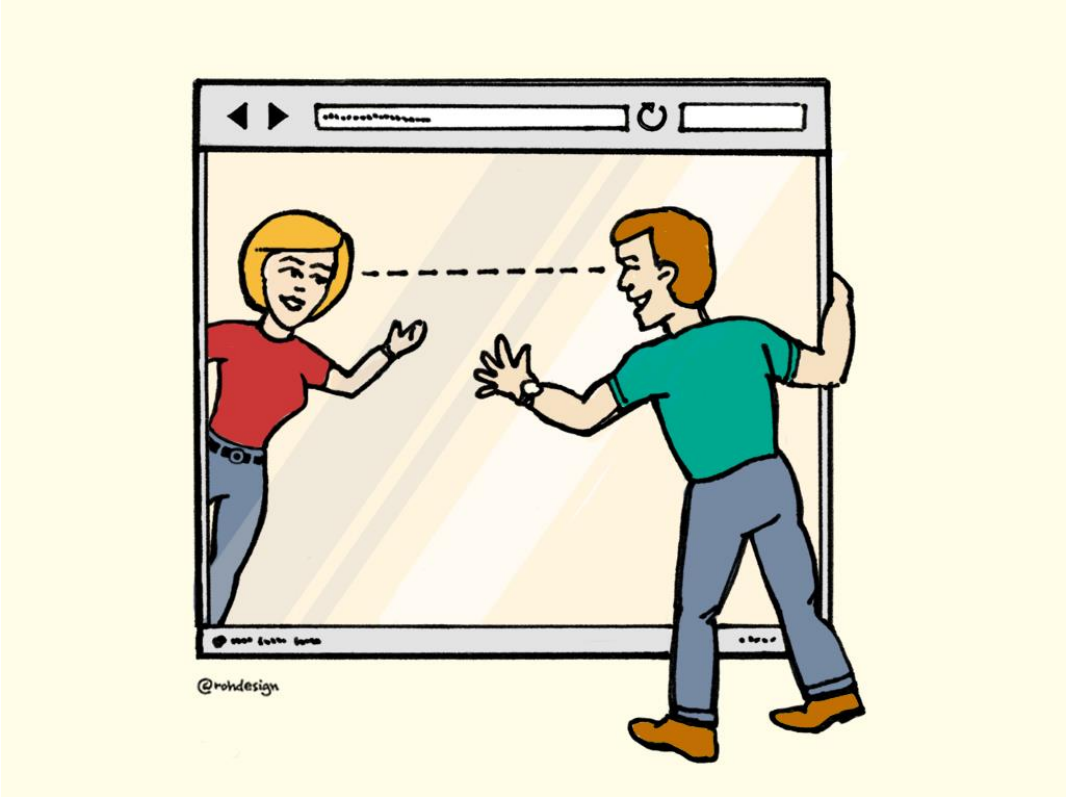
34. ábra: Magyarázza el pontosan, mit lát az ábrán!

Nem tűnik túl bonyolultnak a feladat, mégis, biztosra vehető, hogy nehezebb lesz végrehajtani, mint azt első ránézésre gondolná!

Tekintsük most ezt a feladatot úgy, hogy ön a szoftver, és a megfelelő inputra van szüksége, ami esetünkben ez az ábra. Ön – a szoftver – csak akkor tud helyesen működni, ha ezt az inputot kapja. A szoftver felhasználói felületének – esetünkben most ez is ön – pontos útmutatást kell adnia a felhasználónak –az ön segítőtársának, aki most rajzol –, hogy a megfelelő inputot szolgáltathassa.

Miből adódhatnak a félreértések? Ez a feladat nem is olyan könnyű: alakzatokat kell rajzolni az adott méretben, az adott pozícióban. Hányat, milyet, mekkorát és hová? Ha ezeket a kérdéseket úgy tudják tisztázni egymás között, hogy mindkét fél pontosan megértse a másikat, akkor a feladat nem okozhat problémát.

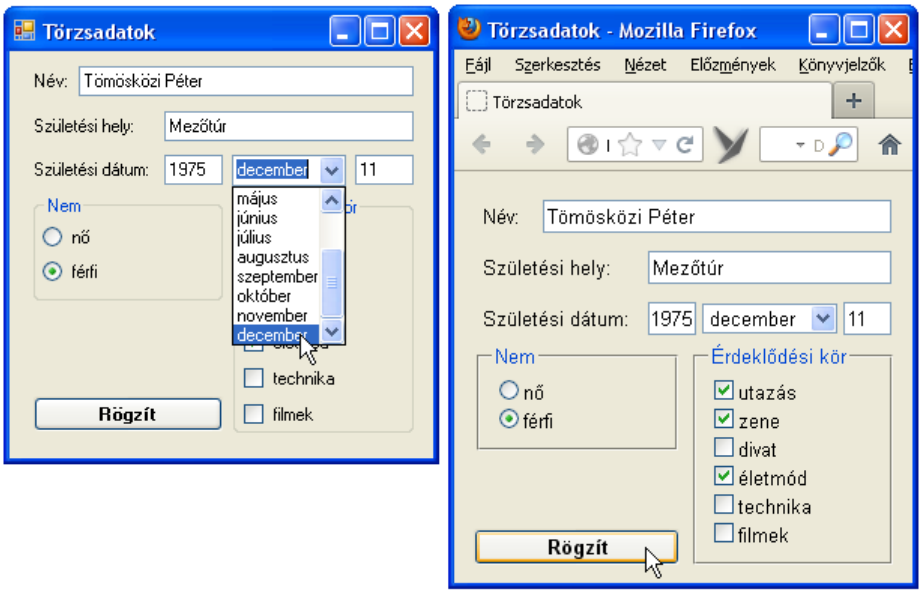
A felhasználói felület a szoftvernek az a része, amelyet a felhasználó használ arra, hogy inputokat adjon a programnak, illetve outputokat kapjon attól. A fenti példa tükrében – de talán anélkül is – nem szorul magyarázatra az az állítás, hogy a felhasználói felületnek a felhasználó számára kell érthetőnek, megfelelőnek, használhatónak, barátságosnak lennie, nem pedig a programozó számára.



35. ábra: Az interfész kapcsolatot teremt

(A kép forrása: http://www.flickr.com/photos/rohdesign/5580730214/)

Minden ma használatos programozási nyelv támogatja a grafikus felhasználói felületek készítését. Objektumorientált programozási nyelvekben ehhez előre definiált osztályok, csomagok (pl. a Javában a Swing és AWT csomagok, Visual C#-ban a *Windows Forms* névtér osztályai stb.) állnak rendelkezésünkre. Ezek használatával az alapprobléma megoldása egyszerűnek tűnik: helyezzük el a felhasználói felületre az adatok be- és kiviteléhez szükséges komponenseket, töltsük meg ezeket a szükséges adatokkal, állítsuk be a működésüket és kész. Ezzel azonban valószínűleg nem fogjuk elérni a kívánt hatást.



36. ábra: Egy űrlap többféle programozási környezetben

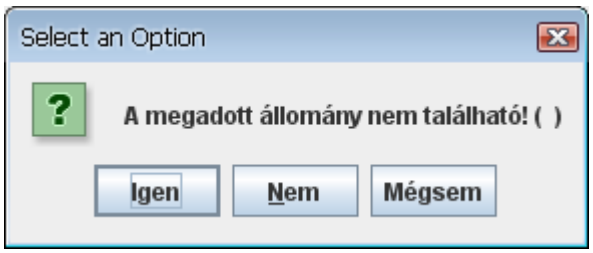
A felhasználói felületeket gyakran azok a fejlesztők készítik, akik a szoftver fejlesztése során a többi programozási feladatot is. Ezt természetesnek is véljük, pedig ha arra gondolunk, hogy a hardver összerakásához rendszerint szakembert hívunk vagy alkalmazunk, vajon miért nem érdemel külön szakembert a felhasználói felületek elkészítése is?

A felhasználói felület minősége jelentősen befolyásolja a szoftver működésének hatásfokát. Tekintsünk vissza a lecke elején vizsgált ábrához! A feladat az volt, hogy magyarázzuk el a velünk szemben ülőnek, hogy mit és hogyan rajzoljon le. Az, hogy mit, hogyan próbáltunk elmagyarázni, jelentős mértékben függött a segítőnk életkorától, iskolai végzettségétől, humán/reál beállítottságától, hangulatától stb. A felhasználói felület használói között is sokféle ember található, akik számára nem feltétlenül jelent azonos szintű feladatot az egér használata, vagy egy billentyűkombináció megjegyzése. Ami számunkra könnyű, az másnak talán megoldhatatlan is lehet.

Egy rosszul megtervezett felhasználói felület azt eredményezheti, hogy a felhasználó nem fér hozzá a szoftver szolgáltatásaihoz, így nem tudja a szoftvert arra használni, amire azt fejlesztették. A rossz felhasználói felület mögött ülő felhasználó inkább érzi azt, hogy a rendszer gátolja őt a munkában, mint hogy segítené.

Lássunk néhány, a felhasználói felületek kialakítását befolyásoló tényezőt!

- Az emberek különböző módon képesek a figyelmük megosztására, eltérőek a motorikus képességeik, lehetnek fogyatékkal élők. Ne az legyen a képernyőfelbontás kiválasztásakor a szempont, hogy mi mit szoktunk meg, illetve hogy mi fér el egy ablakban. Van, aki rosszabbul lát, van, akinek gondot jelent a dupla-tripla kattintás stb.
- Ha a felhasználó lépten-nyomon hibaüzeneteket kap, pláne olyanokat, amelyeket nem is ért, vagy nem is hibaüzeneteket kap, csak annak hiszi őket, frusztrálttá válik, és még többet fog hibázni, vagy bosszankodni.

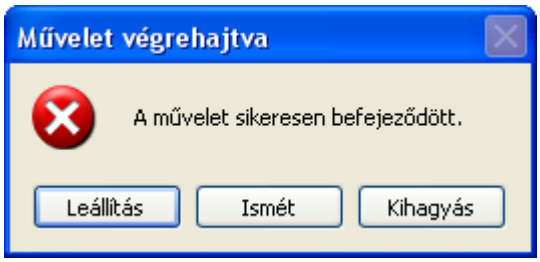


37. ábra: Nehezen értelmezhető hibaüzenet

- Vannak, akik jobban szeretnek menüpontokra, gombokra, listamezőkre stb. kattintani az egérrel, mások a billentyűkombinációkat kedvelik. Vannak, akik szeretik az ikonokat, ábrákat, képeket, mások inkább a bebillentyűzött parancsokat részesítik előnyben.
- Az emberek memóriája véges. Ha egyszerre túl sok információt kell valakinek megjegyeznie vagy megértenie, akkor nem lesz képes azok befogadására.

6.2.2 Tervezési alapelvek

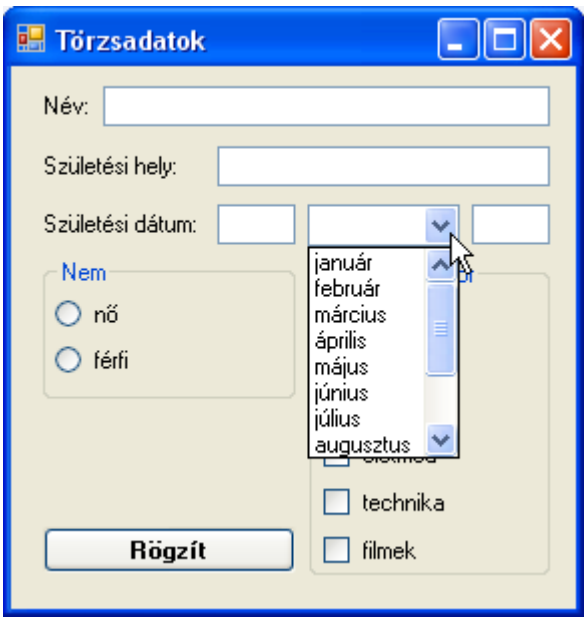
Ne készítsünk el egy felhasználói felületet rosszul csak azért, mert úgy implementálható könnyen. Ne csak az adatbázis felépítése, illetve egy tábla mezőinek sorrendje határozza meg azt, hogy mit lát maga előtt a képernyőn a felhasználó. A felületen jelenjenek meg olyan szavak és kifejezések, amelyeket a felhasználó megszokott és ismer. Ez vonatkozik a fejlesztett rendszer szakterületéhez kapcsolódó szakkifejezésekre, és a jól megszokott informatikai szakzsargonra is.



38. ábra: Most mi a teendő?

Ügyeljünk arra, hogy a felhasználói felület funkcionális elemei (nyomógombok, listamezők, rádiógomb- és választónégyzet-csoportok stb.) megjelenése hasonló legyen, a párbeszédablakokon a gombok felirata legyen minden esetben azonos stb. A digitális tipográfia szabályainak betartása mellett ügyeljünk az ergonómiára is.

Ahol csak tudjuk, segítsük a felhasználó munkáját. Ha egy rögzítendő adat egy állandó lista valamilyen eleme, akkor a felhasználó időt és enegriát spórol, ha nem begépeltetjük vele, hanem lehetőséget kap a kiválasztásra.



39. ábra: A hónapok neveit ne gépeltessük be, hanem tegyük lehetővé a kiválasztást

Designerek szemében ez a szabály csorbíthatja a kreativitás lehetőségét, de erről nincs szó. Egy felületen használhatunk egyedi megjelenésű gombokat, a Windowsban megszokott menük helyett saját magunk által fejlesztett megjelenésűt stb. Ez az alapelv csak azt javasolja, hogy a felület komponenseinek megjelenése legyen következetes. Ez vonatkozik a billentyűkombinációkra is: a felhasználók biztosan hálásak lesznek, ha az általuk használt és ismert billentyűkombinációk az általunk fejlesztett szoftverben is elérhetők lesznek.

A szoftverek tervezése azért is fontos dolog, mert átgondolt tervezéssel csökkenthető az olyan szituációk száma, amellyel meglepjük, megijesztjük a felhasználót. Amikor valamilyen váratlan esemény történik, vagy a felhasználó valamit nem talál meg ott, ahol keresni szokta, vagy ahol eddig elérte, frusztrálttá válik és csak kapkodni fog. Éppen ezért fontos, hogy a szoftver hasonló célú funkciói hasonlóképpen is működjenek. Amikor valamilyen modulban a felhasználó létrehoz egy új bejegyzést (pl. rögzíti egy új ügyfél törzsadatait), akkor ennek eredménye a felhasználói felületen legyen hasonló, mint amikor a forgóeszközök között rögzít egy új rekordot. A felhasználó számára küldött megerősítés, vagy vizuális visszajelzés legyen hasonló minden olyan esetben, amikor hasonló műveletet végez el. Legyen kiszámítható a szoftver válasza egy adott tevékenység elvégzése esetén!

Az operációs rendszerek általában biztosítanak olyan lehetőséget, amely segítségével a véletlen, meggondolatlan vagy nem szándékos adatvesztés hatásai visszavonhatók. Fontos, hogy az általunk fejlesztett szoftverben is legyenek olyan biztonsági pontok, amelyekkel a felhasználó visszavonhat, vagy semmissé tehet egy nem szándékos műveletet. Természetesen vannak olyan esetek, amikor egy művelet célja az, hogy valamit visszavonhatatlanul végrehajtson. Tipikusan ilyen művelet az adatok végleges törlése, vagy egy banki tranzakció végrehajtása stb. Fontos, hogy az ilyen műveletek elvégzése előtt a felhasználó minden esetben kapjon tájékoztatást a várható eredményről, hogy tisztában lehessen a tevékenységének következményeivel. Ez lehet egy figyelmeztető üzenet, vagy megerősítés kérése stb. A cél, hogy elkerülhetőek legyenek a rendszer pontatlan tájékoztatásaiból, vagy a felhasználó figyelmetlenségéből adódó véletlen adatvesztések. A felhasználók támogatása közé tartoznak a súgók, amelyek az adott szituációban azonnali tájékoztatást tudnak adni egy tevékenység hatásairól.

Bizonyos típusú súgóként foghatók fel a varázslók (tündérek), vagy sablonok. A felhasználók egy része csak alkalmanként kerül kapcsolatba egy szoftverrel, és nem is akarja annak minden szolgáltatását teljes körűen megismerni. Amikor a szoftver jellege ezt lehetővé teszi, a felhasználók jó néven veszik, ha van olyan beépített segítő szolgáltatás a programban, amellyel percek alatt meg tudják oldani a kívánt feladatot. A varázslók használata mellett nem elvárás,

hogy a kész produkció a szoftver minden lehetőségét kihasználja, de a felhasználók egy részének ez is kielégítő minőségű eredményt fog szolgáltatni.

A haladó felhasználók biztosan nem fognak varázslókat használni, közülük egyeseket kifejezetten zavarhatnak is ezek a szolgáltatások. Természetesen lehetőséget kell biztosítania a szoftverünknek a varázslók mellőzéséhez is.

6.2.3 Felhasználói felületeken végezhető műveletek

A grafikus felhasználói felületek (Graphical User Interface – GUI) megjelenése előtt a felhasználók viszonylag kevés lehetőséget kaptak a feladataik elvégzésére. Az operációs rendszerek esetében parancsokat kellett begépelnük a parancssorba, az alkalmazói szoftverek esetében pedig parancsok, billentyűkombinációk, esetleg legördülő menük álltak rendelkezésre, esetenként egér használata nélkül.

A grafikus felhasználói felületek megjelenése nem csupán esztétikai értelemben jelentett áttörést. Gondoljunk a nyílt végű és a zárt végű tesztek közötti különbségre: ha egy kérdésre úgy kell választ adnunk, hogy megadott válaszlehetőségek közül kell kiválasztanunk a helyes választ, az rendszerint könnyebb feladat, mint magunktól megadni a jó megoldást, még akkor is, ha az ember valóban jól felkészült a vizsgára. Ha a számítógép egy listát ad a számunkra a választható lehetőségeink felsorolásával, akkor abból mindig könnyebb kiválasztani a helyes, vagy nekünk megfelelő opciót, mint begépelni azt – nem beszélve a gépelési hibák elkerüléséről. Ez főként a kezdő, vagy az alacsonyabb IKT-kompetenciával rendelkező felhasználókra igaz. A haladó felhasználók számára nem okoz gondot egy-egy nemgrafikus felület parancssorának használata, sőt, vannak olyan felhasználók, akik kifejezetten ezt részesítik előnyben.

A felhasználói felület megtervezésekor tehát meglehetősen sok szempontot figyelembe kell venni:

- a leendő felhasználók életkori sajátosságai (gyerekek – felnőttek),
- a leendő felhasználók kompetenciái, elsősorban IKT-kompetenciái,
- a szoftvert futtató munkaállomások paraméterei,
- a megjelenítőeszköz paraméterei,
- a szoftver felhasználási célja,
- és nem utolsó sorban, de semmiképpen sem első helyen: annak a rendszernek a lehetőségei, amelyben a fejlesztést végezzük.

A felhasználói felületek minősége és egy adott funkcionalitás egymástól függetlenül is kezelhető, vizsgálható. Készülhet a szoftverünk úgy is, hogy a különböző felhasználói csoportok különböző felhasználói felületeket használ-

hatnak. Aki kedveli a grafikus felületet, az menüből, ikonok segítségével navigálhat a programban. Aki jobban kedveli a parancssori vezérlést, az ezt is választhatja.

6.2.4 Felhasználói felületek az adatkivitel szemszögéből

A felhasználói felületek teszik lehetővé a szoftver használatát, az adatbevitelt és a feldolgozott adatok megjelenítését. Az outputok megjelenítésének módja sok tényezőtől függhet, ezért célszerű a számított adatokat és azok megjelenítését külön kezelni. Nem jó megoldás, ha a számítást végző metódusok egyben a megjelenítést is elvégzik, mert nehezebbé válik egy esetleges későbbi módosítás implementálása. A jó megoldás az, ha külön objektum számítja ki vagy őrzi meg a kiszámított adatokat, mert ehhez akár több megjelenítő objektum is kapcsolódhat. Ennek a módszernek előnye az is, hogy így egy bizonyos output akár többféle módon is egyszerűen megjeleníthető: a megjelenítő objektumokhoz különféle nézetek társíthatók a programban.

Ha az eredmény egy numerikus adatsor, esetenként az a jó, ha ez az adatsor számszerűen leolvasható a képernyőről. Máskor azt várják a fejlesztőktől, hogy a kiszámított adatsor egy diagramon jelenjen meg. A megjelenítés módja függhet attól is, hogy a kiszámított eredményeket fel fogja-e használni más programegység további számításokhoz, illetve milyen exportálási funkciókat vár el a megrendelő a fejlesztőtől.

A számsorok hátránya, hogy nehéz őket ránézésre értékelni, a diagramok viszont túlságosan helyigényesek. Ha a megjelenítő egy PDA vagy egyéb mobil eszköz, akkor ezzel a tényezővel is számolnunk kell.

A digitális tipográfia szabályai a szoftverek felhasználói felületein hasonló szereppel bírnak, mint a tipográfia szabályai a nyomtatott dokumentumok esetében. További tényezőként jelentkeznek a megjelenítő eszköz sajátosságai:

- képes-e grafikus megjelenítésre
- mekkora a fizikai mérete
- milyen felbontást támogat
- támogatja-e a színes megjelenítést, és ha igen, akkor milyen mértékben

A színek alkalmazása különösen fontos kérdés. A különböző operációs rendszerekben a képernyőkön megjelenített információkat rendszerint különféle sémák befolyásolják. Ezek a sémák a színösszeállítást, a betűtípusokat és betűméreteket, a grafikus komponensek megjelenését stb. határozzák meg. Ezeknek a sémáknak az az előnye, hogy használatuk mellett a fejlesztőnek nem

kell konkrét döntéseket hoznia egy-egy képernyőobjektum megjelenésével kapcsolatosan, hanem hivatkozhat a jellemző sémabeli elnevezésére. Például: nem kell azt mondanunk, hogy az üzenetablak fejlécének színe legyen kék, helyette mondhatjuk azt, hogy legyen olyan színű, amilyen az üzenetablakok fejlécének színe szokott lenni a sémában. Ezzel biztosíthatjuk, hogy a szoftverünk felhasználói felületének megjelenése az adott környezet szabványait fogja követni, lemondunk viszont az egyéni ötleteink, a saját design megvalósításáról, vagy annak egy részéről.

A színek használata különös gondosságot igényel. A felhasználók számára egy felhasználói felület az első ránézés pillanatában barátságos vagy ellenszenves lehet a kiválasztott színek alapján: ez az első benyomásokat befolyásoló tényezők egyike. A feleslegesen tarka felület komolytalanságot, kuszaságot sugározhat, nem beszélve arról, hogy bizonyos színpárosítások rontják, míg mások javítják az olvashatóságot.

A szoftverek üzenetek segítségével kommunikálnak a felhasználókkal. Ezeknek az üzeneteknek egy része információt ad, más részük információt kér, harmadik csoportjuk hibákat jelez stb. A felhasználók sokszor éppen az ezekkel az üzenetekkel való találkozások során ismerik ki a szoftver működésének részleteit, ezért nagyon fontos az üzenetek pontos megtervezése és következetes használata.

Ahol csak lehet, a szoftver olyan üzeneteket adjon, amelyek a felhasználó által végzett aktuális műveletekhez kapcsolódnak. Lehetőség szerint konkrét módon kell a felhasználó tudtára hozni a teendőit. Az üzenetek inkább pozitívak legyenek, mint negatívak. Sértő, bántó, vicces üzenetek nem fogják a kellő eredményt elérni.

Ha nem magyar felhasználók számára készítünk alkalmazást, akkor természetes, hogy a szoftver nyelvének a megrendelő kérésének megfelelőnek kell lennie. A fordítást nem feltétlenül olyan emberre kell bízni, aki jól beszéli az adott ország nyelvét és helyesen is tud írni az adott nyelven. Talán még fontosabb, hogy az adott ország kultúrájával, udvariassági szokásaival is pontosan tisztában legyen. A szoftver üzeneteinek nemcsak pontosnak és érthetőnek, de udvariasnak is kell lenniük.

6.2.5 Akadálymentes felhasználói felületek

Az akadálymentesített felhasználói felület kifejezés hallatára az emberek általában a látássérültek számára készült, nagy kontrasztú (fekete-sárga) képernyőképekre gondolnak. Az akadálymentesítés célcsoportját részben valóban a fogyatékkal élők alkotják, de nem kizárólag. A fogyatékkal élők a számítógép használatához speciális berendezéseket használhatnak, azonban – meglepő

módon – ők alkotják a kisebb csoportot. A nagyobbik csoportba a nem fogyatékkal élők csoportjai tartoznak. Az idősek számára is gondot jelenthet az olvasás, ahogyan az olvasni még nem tudó gyerekek számára is. Akadályoztatott lehet egy egészségügyi értelemben ép, aktív korú felnőtt is, ha éppen lassú sávszélességet, régi számítógépet vagy kis kijelzőjű mobilkészüléket használ.

A felületek akadálymentesítése azt jelenti, hogy a felhasználók minél szélesebb rétege számára elérhetővé tesszük a szoftver használatát. Ugyanakkor nem szabad jelentéktelen feladatnak tekinteni a fogyatékkal élő személyek munkájának megkönnyítését. Nem biztos, hogy egy fejlesztő birtokában van minden olyan ismeretnek, amely alapján a különböző fogyatékkal élő, különböző fogyatékosságú emberek munkáját valóban megkönnyítő felületet tud tervezni. A legegyszerűbb megoldás megkérdezni magukat az érintetteket: Magyarországon többféle szervezet is hathatós segítséget tud nyújtani ilyen kérdésekben. Ilyen szervezet például az „Informatika a látássérültekért” Alapítvány (www.infoalap.hu).

40. ábra: Akadálymentességet igénylő célcsoportok (a kép forrása: http://www.hfe.co.uk/info/policies/accessibility/)

A lecke korábbi részein már szót ejtettünk a vizuális megjelenés fontosságáról, a színek szerepéről, illetve a grafikus felhasználói felület előnyeiről: pl. az egérrel kiválasztás egyszerűségéről a billentyűzetről való begépelés helyett stb. Jól látó, jól halló, ép és egészséges végtagokkal rendelkező felnőtt, de nem idős felhasználók számára ezek valóban fontos kérdések, és ha figyelembe vesszük ezeket a szabályokat, az ő munkájukat segíthetjük velük. De ami előny vagy haszon egy ép felhasználónak, leküzdhetetlen akadály lehet egy fogyatékkal élőnek vagy más ok miatt hátrányba került felhasználónak. Ha fogyatékkal élő

felhasználóra kell gondolni, szinte automatikusan mindenki a vak emberekre gondol, akik monoton, gépi hangon beszélő felolvasó programokkal használják a számítógépet. Azonban távolról sem csak a vak emberek lehetnek fogyatékkal élő, speciális igényű felhasználók, de vizsgáljuk meg elsőként az ő helyzetüket!

Vak emberek felhasználói felületekkel szembeni speciális igényei

Gondoljunk bele, milyen feladat elé állítja a grafikus felhasználói felület a vak felhasználókat: szükségük van egy programra, amely helyettük lát. De mit lát? A grafikus felületen minden operációs rendszerben természetes az ablakozó technika: a különféle szoftverek ablakokban futnak. Ezeknek az ablakoknak menüik vannak, különféle képernyőüzeneteket közölnek a felhasználóval, aztán az operációs rendszer maga is fel-feldob üzenteket a képernyő legkülönböző helyein… mit olvas fel a felolvasóprogram? Minden egyszerre képtelenség, ahogyan a látó felhasználó sem olvas el a képernyőn egyszerre mindent.

Ha vak emberek számára szeretnénk akadálymentes felületű szoftvert fejleszteni, feltétlenül érdemes (szükséges), hogy magunk is kipróbáljunk egy felolvasóprogramot. Ezt megtehetjük ingyen is, a korábban már említett www.infoalap.hu weboldalon ingyenesen elérhető egy közismert képernyőolvasó szoftver próbaverziója. Próbáljuk ki: próbáljunk meg elvégezni egy egészen egyszerűnek tűnő feladatsort – kikapcsolt képernyővel:

❁ Feladat:

1. Indítsuk el az operációs rendszerhez kapott egyszerű szövegeditort (pl. Jegyzettömb)!
2. Gépeljük be a mai dátumot, írjuk oda azt is, hogy milyen névnap van ma!
3. Mentsük el a dokumentumaink közé úgy ezt a fájlt, hogy a dokumentumok között létrehozunk egy új mappát, melynek a Névnap nevet adjuk. A fájl neve is legyen Névnap!
4. Zárjuk be az editort, majd indítsunk el valamilyen fájlkezelő alkalmazást.
5. Csatlakoztassunk egy USB-háttértárat (pendrive vagy mobil merevlemez) a számítógéphez, majd másoljuk át a létrehozott fájlt (mappa nélkül) a csatlakoztatott eszköz valamely mappájába, amely a gyökérből nyílik.

Hamar tapasztalni fogjuk, hogy ez a feladat egy látó ember számára elsőre gyakorlatilag megoldhatatlan. Azonnal felmerülhet bennünk az ötlet: a vak embereknek másik felületet kell készíteni, hogy használhassák a számítógépet. Ez az ötlet azonban kivitelezhetetlen.

- A vak felhasználók ugyanazt az operációs rendszert használják, melyet a látók.
- A vakok ugyanazokat az alkalmazói szoftvereket használják, mint a látók.
- A látóknak készült szoftverek felülete általában magának a szoftvernek a jellegéből adódóan bonyolult (ha az). Az egyszerű, áttekinthető felületnek a látók is örülnek. Megfordítva: egy bonyolult tevékenységet végző szoftver felületét általában nem lehet leegyszerűsíteni anélkül, hogy a funkcionalitásból is vesztene a szoftver.

Megoldás: úgy kell elkészíteni a szoftver felhasználói felületét, hogy azt mind látó, mind vak felhasználók képesek legyenek használni. Ehhez ismerni kell a felolvasóprogramok (esetleg más, vakoknak készült eszközök, pl. Braille-megjelenítő) működését, és a szoftver felületét úgy kell kialakítani, hogy az maximálisan képes legyen együttműködni a látássérült felhasználók munkáját könnyítő eszközökkel.

Néhány javaslat:

- A vak felhasználók a billentyűzetet használják, nem pedig az egeret. A szoftver minden funkcióját elérhetővé kell tenni billentyűzetről.
- A felolvasóprogram a szöveget tudja felolvasni, a képeket nem látja. Éppen ezért képként semmilyen szöveges információt nem szabad reprezentálni, illetve a csak képként megjeleníthető információk esetében elérhetővé kell tenni szöveges leírást (alternatív szöveg).
- Az űrlapok esetében a felolvasóprogramnak (és a vak felhasználónak) pontosan kell tudnia, hogy a szoftver milyen típusú adatot vár tőle az adott űrlapmezőben. Csak a színekkel való megkülönböztetés, vagy különféle beviteli maszkok önmagukban nem szolgáltatnak kellő információt.
- Segítség a vak felhasználóknak, ha van mód arra, hogy a szoftverben megkönnyítjük a navigációt a számukra. A látó felhasználók olvasáskor egyszerűen átugorják a számukra nem érdekes vagy nem fontos képernyőrészeket, a vakok viszont arra kényszerülnek, hogy végighallgassák a számukra felesleges részeket is, ha nincs mód arra, hogy ezeket átugorhassák.
- Nem kell külön felületet készíteni a vakoknak, hanem a látóknak készült felületet kell vakbaráttá tenni. Ez azonban véletlenül sem a – teljesen téves elképzeléseken alapuló – fekete-sárga, nagykontrasztú képernyőt jelenti. Vakbarát az a szoftver, amelynek a felületét csupán felolvasóprogram használatával, képernyő nélkül is 100%-osan lehet használni.

Színtévesztő, színvak felhasználók felhasználói felületekkel szembeni speciális igényei

Ennek a csoportnak a tagjai a színek érzékelésével kapcsolatosan kerülnek hátrányba az ép felhasználókkal szemben. Az ő esetükben az akadálymentesítés azt jelenti, hogy a szoftver semmilyen információt nem közölhet a felhasználóval úgy, hogy az információt kizárólag a szín hordozza. Tipikusan ilyen eset az, amikor egy űrlapon a kötelezően kitöltendő űrlapmezők piros, míg a többi mező zöld vagy fehér színű. A színtévesztők körében éppen a vörös-zöld színtévesztés számít az egyik leggyakoribbnak, egy ilyen felhasználó számára semmilyen információt nem nyújt egy vörös vagy zöld hátterű űrlapmező.

Bármelyikünk megtapasztalhatja ezt a problémát, ha egy olyan űrlapot kell kitöltenie, amelyen a vörös és a zöld telítettsége és világossága nagyjából megegyezik, viszont a kijelző – amelyet a kitöltéskor használhatunk – monokróm. Egy szürkeárnyalatos kijelzőn elvész a színinformáció, és a tájékozódást csak a kontraszt segítheti. Minden szoftver felhasználói felülete esetében biztosítani kell a megfelelő kontrasztarányt.

Siket, nagyothalló emberek felhasználói felületekkel szembeni speciális igényei

Siket vagy nagyothalló emberek számára természetesen azok az információk nem jutnak el (vagy csak részben), amelyek a halláson alapulnak (hang, beszéd, zene stb.) Alkalmazói szoftverek esetében – a legáltalánosabb esetben – ez azt jelenti, hogy egy szoftver akkor tekinthető akadálymentesnek az ilyen felhasználók számára, ha nem közöl úgy információt, amely kizárólag hallás útján juthat el a felhasználóhoz. Ha egy hibajelzést csak egy sípszó jelez, arról egy siket felhasználó nem értesül. Az ilyen fogyatékkal élő felhasználók számára mindig szükséges valamilyen vizuális alternatíva. Hibák esetében ez kézenfekvően valamilyen üzenetablak lehet.

Egy látó felhasználó általában nem használja a számítógépet, ha annak monitora vagy megjelenítője nem működik, viszont elég gyakori eset, hogy egy ép hallású ember olyan eszközt használ, amely nem képes hangjelzést adni (nincs benne hangkártya vagy ki van kapcsolva a hangszóró). Az akadálymentes felhasználói felületek tervezésekor tehát nemcsak a fogyatékkal élő személyekre, de a technikai okok miatt hátrányos helyzetű felhasználókra is gondolnunk kell.

Mozgásukban korlátozott emberek felhasználói felületekkel szembeni speciális igényei

Mozgásukban korlátozott felhasználók esetében elsősorban a felső végtagok, vagyok azok részeinek betegségéből, esetleg hiányából adódó nehézsé-

geket kell megvizsgálnunk akadálymentességi szempontból. Az ilyen felhasználók számára mind a billentyűzet, mind az egér, tehát a két legáltalánosabb beviteli eszköz használata okozhat problémát. Ez megnyilvánulhat abban, hogy a felhasználó nem tud vagy csak nehezen képes billentyűkombinációkat lenyomni, esetleg egyáltalán nem képes billentyűzetet használni, vagy éppen egeret.

Az operációs rendszerek mindegyike biztosít kisegítő lehetőségeket a fogyatékkal élő felhasználóknak, és a billentyűzet, illetve az egér használatának megkönnyítése minden operációs rendszerben megtalálható. Emellett speciális szoftverek is segíthetik a felhasználók munkáját, ilyen például a fejegér, amely használatához egy webkamerára és egy speciális (de ingyenesen elérhető) szoftverre van szükség. Ilyen pl. a MouSense nevű szoftver, amely elérhető az alábbi címről: http://mousense.com/hu/

Ahogyan a (csak) hallás útján megszerezhető információk eljutása is korlátozott lehet egészen hétköznapi esetekben is, bármelyikünk válhat ideiglenesen a mozgásában korlátozott felhasználóvá. Ha a számítógéphez nincs egér csatlakoztatva, akkor csak a billentyűzetet használhatjuk. Ha valaki eltöri a karját vagy ínhüvelygyulladást kap a sok gépeléstől, máris hátrányba kerülhet, ha sok és bonyolult billentyűkombinációt kell használnia stb.

A mozgáskoordináció az életkor függvényében haranggörbéhez hasonló képet mutat. A kisgyermekek mozgáskoordinációja még nem fejlődik ki, a finommotorikát igénylő mozgások az ő számukra éppen olyan nehézséget okoznak, mint az idős embereknek. Ha egy bonyolult menüszerkezeten belül úgy kell elnavigálni az egeret, hogy annak pályája egészen keskeny, azt egy remegő kezű idős ember éppúgy nem tudja megcsinálni, ahogyan az a kisgyermek sem, aki még csak tanulja használni a kezét. Mindketten ugyanolyan hamar feladják a feladatot, mint amilyen hamar mérges lesz az az ép és fiatal felhasználó, akinek piszkos az egerében a golyó, vagy össze-vissza ugrál a szeme előtt az egérkurzor, mert megfelelő alátét hiányában nem jól működik az optikai egere.

Jogosan érezhetjük, hogy ez az utolsó példa nem feltétlenül életszerű. Valóban ritkán van szükség arra, hogy olyan felhasználói felületet tervezzünk, amelyet olvasni még nem is tudó kisgyermekek, valamint életük végéhez közelítő idős emberek egyaránt használnak. De nem reménytelen vállalkozás ilyen szoftvert keresni. Hipermarketekben, postán, köztereken stb. álló információs eszközök esetében az érintőképernyő előtt gyerekek és öregek is megállhatnak. Egy információs weblapnak lehetnek kicsik és nagyok, látók és vakok, épek és végtaghiányos felhasználók, vagy egyszerűen csak balkezesek is a látogatói. Nem beszélve arról, hogy a weben futó alkalmazásokat különböző operációs rendszereken, különféle böngészőprogramokkal, más-más eszközökön nézik, használják a felhasználók. Tulajdonképpen oda jutottunk, hogy bármelyikünk bármelyik pillanatban válhat ideiglenesen fogyatékos vagy hátrányos helyzetű

felhasználóvá, tehát igen fontos kérdés az, hogy az általunk fejlesztett szoftverek felhasználói felülete mennyire képes kiszolgálni a megváltozott képességű emberek igényeit.

Ehhez kapcsolódóan mindenképpen érdemes tanulmányozni a W3C által készített webes akadálymentesítési útmutatót (Web Content Accessibility Guidelines, WCAG 2.0). Az útmutató megtalálható a W3C weboldalán, de magyar fordítás is készült róla, melyet a http://www.w3c.hu/forditasok/WCAG20/ címen érhetünk el. Az útmutató elsősorban a weboldalak akadálymentesítésének kérdéseit járja körül, de számos olyan tanács található benne, amelyek asztali alkalmazások fejlesztésénél is megfontolandók.

6.2.6 Felhasználói felületek tervezési folyamata

Amikor egy weboldalt kezdenek tervezni, gyakran egy designer elkészíti a weboldal terveit egy képszerkesztő programmal, majd ennek elemeit bocsátja rendelkezésre a fejlesztőknek. Az egyéb szoftverek esetében is hasonló módon születik meg a felhasználói felület: a tervezés alapja itt is a modellezés. A majdani felhasználók a fejlesztőkkel együttműködve prototípusokat készítenek, ezek a prototípusok a szoftver többi komponensének fejlődése során szintén változhatnak. Az alapos tervezéshez itt is hozzátartozik a modellek alkotása (ezek lehetnek akár papír alapúak, vagy képszerkesztő programban alkotott vázlatok), és lehetőség szerint célszerű ezek előzetes kipróbálása is.

A tervezés előtt meg kell vizsgálnunk a különböző modulok felhasználóinak a munkakörét. Egy nagy rendszert többféle felhasználói csoport használ, ők a szoftvernek többféle nézetével találkozhatnak: az adatrögzítést végző csoport munkatársainak jellemzően az adatrögzítés felületét kell használniuk, a bérszámfejtők felhasználó felületét valószínűleg nem kell ismerniük. Meg kell vizsgálnunk, hogy az egyes csoportok milyen típusú munkát végeznek és milyen egyéb rendszerekkel dolgoznak munkájuk során.

A modellezés után prototípus(oka)t kell készítenünk, amelyeket a leendő felhasználók értékelhetnek, lehetőség szerint ki is próbálhatnak. Nehéz véleményt alkotni egy olyan dologról, amit csak elképzelni lehet, de látni nem. Az értékelés során kapott javaslatokat, ötleteket fel kell használni a felület terveinek finomításában, így a majdani felhasználók aktív részesei lehetnek a tervezésnek.

Ahogyan a szoftvertervezés során, a felhasználói felületek tervezésénél is fel kell készülnünk – már a modellalkotás folyamatában – a szoftver evolúciójára: szükséges lehet az elkészült szoftver átadás utáni karbantartása. Ha új funkciókat építünk a szoftverbe, akkor az új funkcionalitás újabb komponenseket,

akár újabb felületeket igényelhet, illetve a használat során feltárt hibák javítása is befolyásolhatja a felületek komponenseinek működését.

6.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

6.3.1 Összefoglalás

A program helyes működése szűk keresztmetszetben azt jelenti, hogy a szoftver a beérkező adatokat helyesen dolgozza fel és helyes kimenő adatokat, outputokat állít elő. A felhasználó szemszögéből a helyesség azonban kisebb jelentőségű, hiszen a felhasználó ugyanis eleve azt feltételezi, hogy a program helyesen működik.

A felhasználó számára a helyesség sokkal inkább gyakorlati kérdés: hozzáfér-e a szoftver minden funkciójához? Ki tudja-e használni a szoftver minden lehetőségét a lehető legmagasabb szinten? Megérti-e a szoftver utasításait, üzeneteit? Képes-e magát a felhasználó megértetni a szoftverrel?

Mivel a szoftverek használata rendszerint interaktív folyamat, a felhasználó minimálisan elvárhatja, hogy az a felület, amelyen kommunikál, az ő számára készüljön. A jó felhasználói felület a felhasználó számára szinte átlátszó: úgy tudja használni, hogy nem kell észrevennie, hogy a szoftver felületét használja. Ha a felülettel kell hadakoznia, akkor biztosan nem tud majd a programmal együttműködni, és viszont.

6.3.2 Önellenőrző kérdések

1. Milyen elvárásai vannak a felhasználóknak a felhasználói felületekkel szemben?
2. Miért nem előnyös, ha a felhasználói felületet a programozók készítik?
3. Milyen tényezők befolyásolják a felhasználói felületek kialakítását?
4. Melyek a felhasználói felületek tervezésének alapelvei?
5. Milyen műveleteket végeznek a felhasználók a felhasználói felületeken?

7. LECKE: GYORS SZOFTVER-FEJLESZTÉS

7.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Az előző leckék során, de főleg a 2. leckében leírtuk a szoftvertechnológia optimális folyamatait. Ennek során láthattuk, hogy a tényleges implementációt akár több hónapos, de esetenként több éves tervezési folyamat is megelőzheti. Ez a módszer nem minden szoftverrendszer fejlesztése esetén megvalósítható. Ebben a leckében azt vizsgáljuk, hogy hogyan lehet a szoftvertechnológia egyik legfontosabb tényezőjét optimalizálni: a szoftverfejlesztés időtartamát lerövidíteni.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező képesség, figyelem-összpontosítás.

7.2 TANANYAG

7.2.1 Szoftverfejlesztés a valós piaci igények tükrében

A cégek állandóan változó piaci körülmények között működnek és igyekeznek követni a változásokat. Az információtechnológia a cégek minden tevékenységében megjelenik, ezért fontos, hogy minden feladat elvégzéséhez a megfelelő szoftvert választhassák a vállalatok. Mivel a piac nagyon gyorsan változik, a szoftverekkel szemben támasztott követelmények közül talán a legfontosabb a gyorsaság, vagyis hogy a szoftver fejlesztője a lehető leggyorsabban átadhassa a kész rendszert a megrendelőnek. A megrendelők ennek fejében hajlandók kompromisszumokra, engednek az elvárásaikból, hogy minél hamarabb használhassanak a szoftvert. Másfelől egy jól modellezett, megtervezett szoftver esetében is természetes, hogy bizonyos elvárások csak a szoftver evolúciója során fogalmazódnak meg a felhasználókban, akkor, amikor kipróbálhatják a rendszert, megismerhetik annak együttműködését más rendszerekkel stb. Ezt megelőzően a megrendelő sem tud feltétlenül pontos igénylistát benyújtani a készítendő rendszerrel kapcsolatban.

Ennek megfelelően a korábban leírt modellezési, tervezési, fejlesztési folyamat gyakran nem kivitelezhető. A jó szoftver készítésének feltétele a jó rendszerterv, azonban ennek kialakítása esetenként túlságosan hosszú időt vesz igénybe. Egy gyorsan változó piaci környezetben előfordulhat, hogy mire elké-

szül egy rendszer terve és kezdődhetne annak implementációja, a piaci folyamatok annyira megváltoznak, hogy a munkát ismét a tervezésnél kell kezdeni, vagy a meglévő terveket teljesen át kell alakítani.

A gyors szoftverfejlesztési folyamatokat arra tervezték, hogy segítségükkel valóban gyorsan készíthessünk szoftvereket. Általánosságban elmondható, hogy az ilyen tevékenység során a modellezési, tervezési és implementációs munka párhuzamosan folyik.

A modellezés és a specifikáció csak a legszükségesebb komponensekre, illetve azok legfontosabb jellemzőire korlátozódik. A rendszert lépésről lépésre tervezik, és a tervezés, illetve fejlesztés minden mozzanatában aktívan részt vesznek a felhasználnók, így azonnali visszajelzést tudnak adni minden, a rendszerbe újonnan bekerülő komponensről. A fejlesztés során a felhasználói felület implementálása bizonyos rendszerekben gyorsabb, másutt lassabb művelet. A gyors szoftverfejlesztés eszközéül olyan fejlesztői környezetet érdemes választani, ahol a felhasználói felületek fejlesztését a rendszer beépített eszközei közvetlenül támogatják (vizuális programozási nyelvek).

Ennél a technológiánál a felhasználóknak nem kell heteket, hónapokat várnia az első prototípusokra, mivel a tervezési és specifikálási lépések az implementálással párhuzamosan folynak. Így már a fejlesztés legkorábbi szakaszában is működő prototípusokkal találkozhatnak, esetleg a legfontosabb funkciók már a végleges formájukban működhetnek a rendszer legelső változataiban is. Így a felmerülő igényeket is azonnal jelezhetik a felhasználók, amelyek implementálása a fejlesztés további irányát is kijelölheti, vagy megváltoztathatja.

A gyors szoftverfejlesztés ezen módja hordoz magában bizonyos kockázatokat is. A fentebb leírtak nemcsak lehetőséget biztosítanak a felhasználók számára az azonnali kipróbálásra, de ez mint követelmény is megfogalmazódik velük szemben. Nem biztos, hogy a majdani felhasználók birtokában vannak azoknak a kompetenciáknak, amelyek szükségesek ahhoz, hogy egy fejlesztés alatt álló rendszert véleményezni tudjanak. Nem biztos, hogy van a fejlesztési folyamatban olyan résztvevő (akár a megrendelő, akár a fejlesztő oldalán), aki vállalni tudná az újabb és újabb részegységek betanítását és lehet, hogy ez a felhasználók számára egyébként is csak többletfeladatokat jelentene, amelyet nem szívesen vállalnak.

Egy vállalat általában szigorú szabályok betartása mellett működik, a folyamatokat rendszerint pontosan és alaposan dokumentálni kell. Egy fejlesztés alatt álló szoftverkomponens által szolgáltatott kimeneteknek illeszkednie kell a jelenleg működő rendszer szabványaihoz, ami lassíthatja a munkát. Ha – akár csak ideiglenesen – nem illeszkednek a meglévő szabványokhoz, az a vállalatot

állíthatja nehéz döntés elé. A dokumentációk pontos és szabványos elkészítése szintén követelmény lehet, ugyanakkor nagyon időigényes tevékenység.

A gyors szoftverfejlesztés anyagi és szerződésbeli kérdéseket is felvet. A hagyományos szoftvertechnológia során a szerződések megkötése a specifikáción alapszik. Amikor elkészül egy rendszer specifikációja, a felek szerződést kötnek az elvégzendő feladatokra, a komponensek kifejlesztésére és a vállalt határidőkre. Ha a specifikáció a fejlesztéssel párhuzamosan zajlik, akkor mindkét fél számára nehézséget jelenthet a kedvező szerződési feltételek kialakítása. A megrendelő nem akarja a vételárat kizárólag az alapján elfogadni, hogy a fejlesztő milyen határidőre vállalja a munkát. A fejlesztő pedig nem akar fix összegű megrendelést kötni egy olyan projektre, amelyben bizonyos komponensek iránti igény csak a fejlesztés egy későbbi szakaszában fog jelentkezni.

A hagyományos technológia alkalmazása során egy fejlesztés alatt álló rendszer validációja és verifikációja már a specifikáció alatt megkezdődhet. Ha a specifikáció átfedi az implementációt, akkor ez a majdani kész rendszer ellenőrzését, tesztelését is befolyásolja. Ugyanez a probléma adódik a későbbi módosítás, karbantartás során is. Mivel a kész rendszer nem feltétlenül egy jól átgondolt tervezési munka után készül el, bizonyos funkciók megvalósítása nem az optimális módon történhet. Ezen lehet azzal segíteni, ha az újabb és újabb komponenseket állandóan tökéletesítjük, például refaktorálással, amelyről a 11. leckében esik szó.

Belátható, hogy vannak olyan rendszerek, amelyek fejlesztéséhez nem alkalmazható a gyors szoftvertechnológia. Ha nagy rendszert kell fejleszteni, amelyen egymástól térben és időben távol lévő csoportok dolgoznak, a rendszer dokumentációja szigorú szabályokat követ és a szerződések a specifikáción alapulnak, akkor csak a hagyományos szoftverfejlesztési eljárások jöhetnek szóba. Ugyanakkor azt is meg kell jegyeznünk, hogy az ilyen nagy méretű projektek rendszerint nem annyira érzékenyek a piac napi változásaira, hogy az időtényező mindennél fontosabb lenne a fejlesztésben.

A gyors szoftverfejlesztés szempontjából a legfontosabb tényező az idő: a megrendelő minél hamarabb szeretné használatba venni az új szoftvert. Ez azonban nem jelentheti azt, hogy megelégszik egy közepes vagy rossz minőségű, instabil, megbízhatatlan szoftverrel, legfeljebb azt, hogy a fejlesztés gyorsaságának növelése érdekében hajlandó bizonyos kompromisszumokat kötni a funkcionalitás kárára.

A valódi megoldások rendszerint valahol a két szélsőség között helyezkednek el.

7.2.2 Gyors szoftverfejlesztést támogató eszközök

Egy raktárkészlet-nyilvántartó szoftver esetében a következő komponensek megtervezése lehet szükséges:

- megfelelő relációs adatbázis kialakítása (táblák, mezők, kapcsolatok),
- űrlapok tervezése és készítése az adatok karbantartásához,
- felhasználók jogosultságainak kialakítása,
- lekérdezések megvalósítása a felhasználó igényeinek megfelelően,
- jelentések készítése a számított adatok felhasználásával,
- biztonsági mentések, adatarchiválás megvalósítása,
- megfelelő felhasználói felület kialakítása a funkciók eléréséhez.

A fejlesztéshez választott eszközt az alábbiak befolyásolhatják:

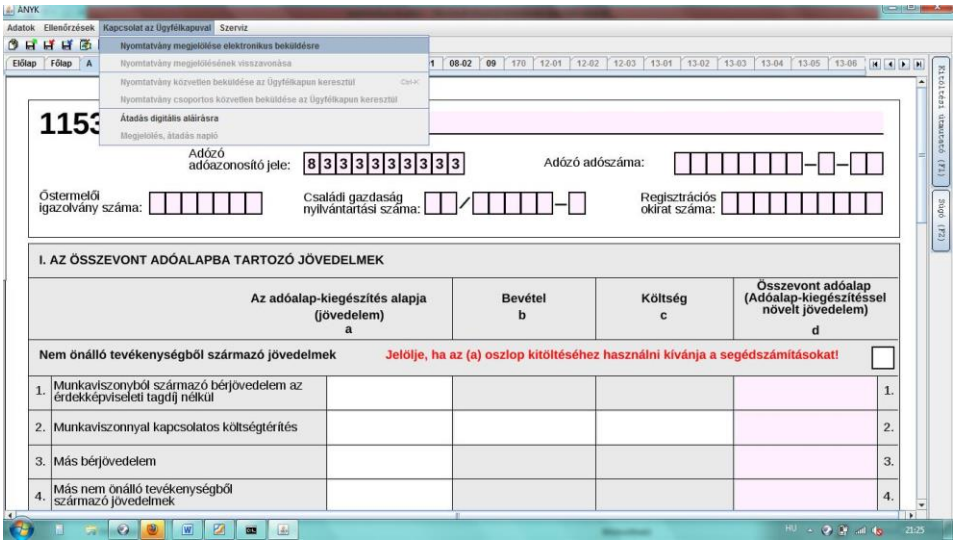
- Jelenleg milyen rendszert használ az ügyfél?
- Hányan használják a szoftvert?
- Honnan, milyen munkaállomásokról használják a szoftvert?
- Szükséges-e a szoftver által szolgáltatott kimenetek (jelentések) további feldolgozása más rendszerekben?
- Mennyi pénzt és mennyi időt szán a megrendelő a fejlesztésre? Hajlandó-e újabb beruházásokat tenni a szoftver használatához?

Ha a szoftvert sokféle felhasználó, egymástól földrajzilag nagy távolságra lévő, különböző operációs rendszert használó, különböző típusú munkaállomáson veszi igénybe, célszerűnek látszik valamilyen internetes (pl. webes) felületű szoftver kialakítása. Ha a karbantartott rekordok száma, illetve a konkurens felhasználók száma nem extrém méretekkel jellemezhető, és nem szigorúan titkos, bizalmas információk (pl. államtitkok) karbantartása a feladat, szóba kerülhetnek ingyenes adatbázis-kezelő rendszerek. Az implementációt ilyenkor az befolyásolja, hogy a munkaállomásokon a felhasználói felületek megjelenítésére célszerűbb-e a böngészőprogramok szolgáltatásait kihasználni, vagy a megjelenítéssel szemben támasztott speciális követelmények miatt saját eszközt kell készítenünk. (Kis és közepes méretű webes adatbázis-kezelő alkalmazások fejlesztésénél a MySQL és a PHP alkalmazása szinte de facto szabvány.

41. ábra: Webes adatbázis-kezelő alkalmazások legkedveltebb fejlesztőeszközei

Ugyanakkor pl. az AbevJava program fejlesztésekor biztosan szigorú követelmény volt, hogy a megjelenített űrlapok vizuálisan is az adóbevallásban használható papír alapú űrlapokkal egyezzenek meg. HTML-űrlapelemekkel ez a megjelenítés nem valósítható meg. A platformfüggetlenség biztosítása miatt itt a Java nyelvre és a Swing csomagra esett a fejlesztők választása.)

42. ábra: A Java és a Swing logója



43. ábra: Az általános nyomtatványkitöltő (ÁNYK, AbevJava) szoftver egy űrlapja

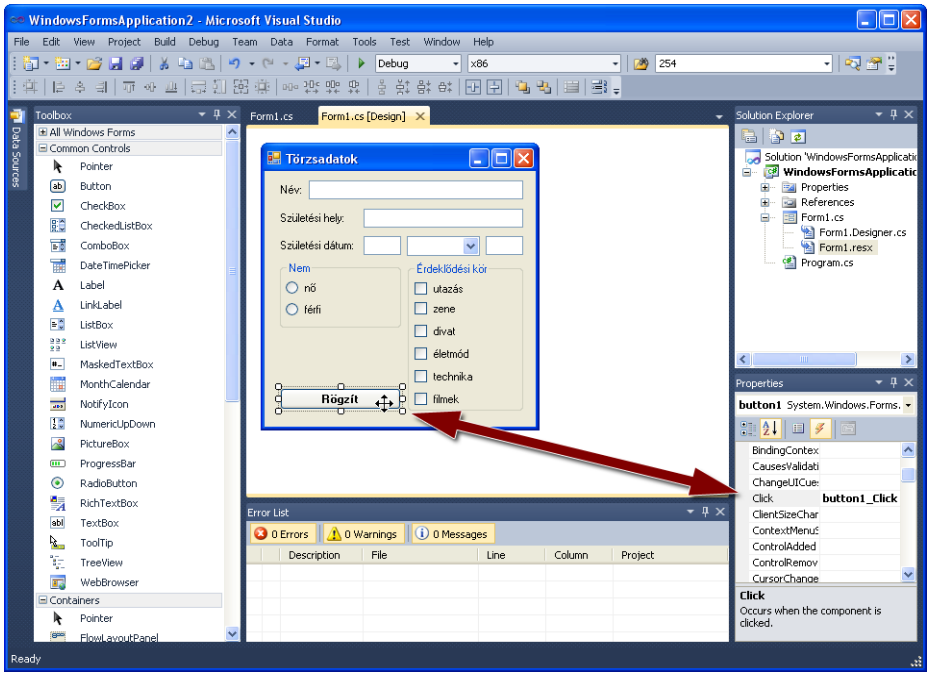
Ha a megrendelő már használ olyan rendszert, amelyhez meg kellett vásárolnia az Microsoft SQL-t, akkor érdemes olyan szoftvert választani a fejlesztéshez, amely jól együtt tud működni ezzel az adatbázis-kezelővel. Vélhetőleg a megrendelt alkalmazást ilyenkor valamilyen vizuális programozási nyelv segítségével célszerű fejleszteni (Microsoft Visual Studio, Visual Basic, Visual C# stb.). Ezek a fejlesztői környezetek beépített eszközöket és szolgáltatásokat tartalmaznak a felhasználói felület kialakításához is. Az MS SQL természetes módon támogatja a konkurens, akár távoli hozzáférést az adatbázisokhoz, de hátránya, hogy közvetlenül csak Windows operációs rendszerekkel képes együttműködésre.

Ha a fejlesztendő szoftvert jellemzően csak egy munkaállomáson, esetleg egy helyi hálózat néhány munkaállomásán fogják használni, a munkaállomásokon Windows fut és a megrendelő nem kíván különösebb beruházásokat tenni a szoftver használata érdekében, akkor az Office csomagban található Access adatbázis-kezelő rendszer tűnik a jó megoldásnak. A fejlesztő ebben a rendszerben szinte programozás nélkül készíthet olyan alkalmazást, amely a legáltalánosabb adatbázis-karbantartási funkciókat maradéktalanul képes ellátni. Hátránya, hogy az alkalmazás csak Windows környezetben működik, és bár a kifejleszthető felhasználói felület egészen széles határok között skálázható, a szoftver felhasználói felületei mégis felismerhetően ehhez az alkalmazáshoz fognak kötődni. Ennek a módszernek az előnyei között kell említenünk, hogy az adatok importálása és exportálása más irodai szoftverekkel készített fájlokból

nagyon magas szinten támogatott. Ha a megrendelő Windows környezetben dolgozik, akkor szinte biztos, hogy az általa használt adatoknak legalább egy részét ilyen szoftverekkel kezeli. Ez is szólhat az Access választása mellett.

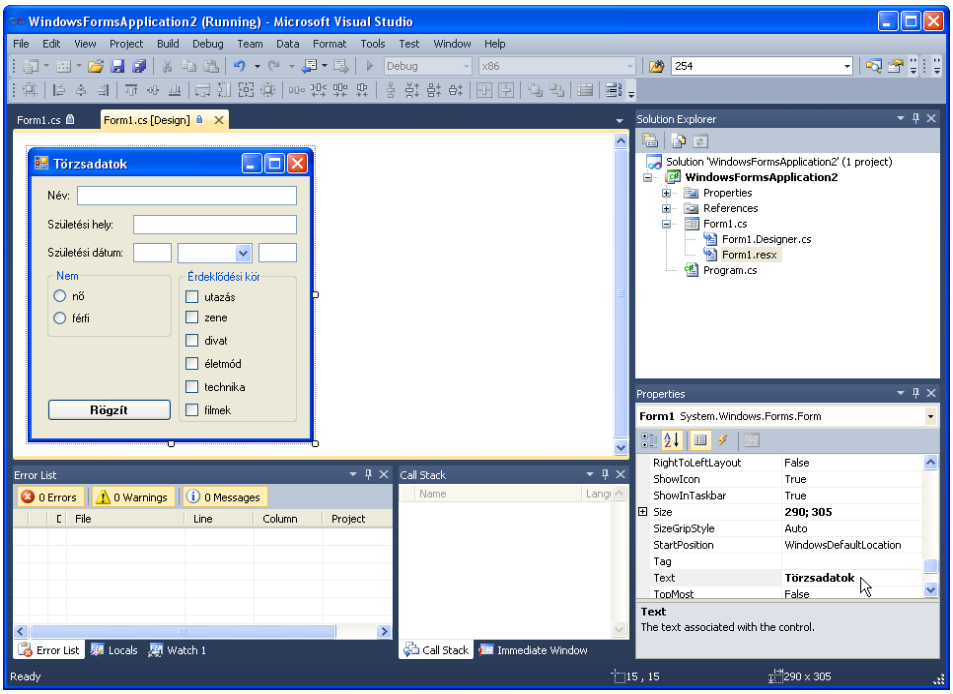
7.2.3 Vizuális programozási nyelvek

A vizuális programozási nyelvek beépített felhasználói felületgenerátorának a használatával szintén nagyon gyors alkalmazásfejlesztés valósítható meg. Ez akár a prototípusok szintjén, akár a kész szoftver tekintetében nagyon gyors fejlesztést tesz lehetővé. A programozó gyakorlatilag megrajzolja a felületet, elhelyezi a szükséges űrlapelemeket, majd beállítja az interakciós lehetőségeket. Az ilyen fejlesztést eseményorientált fejlesztésnek is nevezik. Az eseményorintált programozás lényege, hogy a futó program lényegében egy végtelen, várakozó ciklust valósít meg. A program megrajzolja a felület elemeit, majd várakozik a bekövetkező eseményekre. Az események egy része a felhasználói interakciók eredményei (a felhasználó rákattint egy nyomógombra, kiválaszt egy elemet egy legördülő listából, bezár egy párbeszédablakot stb.), míg mások érkezhetnek az operciós rendszertől, esetleg más alkalmazásoktól.



44. ábra: Vezérlőelem eseménye és a hozzárendelt művelet

Az eseményorientált programozás hasonlít a szkriptnyelveken való programozáshoz. A programozó nem egyetlen, összefüggő programot fejleszt, hanem a futás során bekövetkező eseményeket kezeli kvázi eseménykezelők készítésével. Magát az eseménykezelő alapprogramot és a futtató rendszert a vizuális programozási nyelv állítja össze, a programozó feladata nagyrészt az események kezelése, kisebb részben pedig a rendszer működéséhez szükséges többi komponens fejlesztése (pl. adatbázis kialakítása, karbantartása).



45. ábra: A Microsoft Visual Studio felhasználói felülete

7.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

7.3.1 Összefoglalás

A piaci folyamatok rendszerint nem kedveznek a hosszas szoftvertervezésnek. A nem jól tervezett és specifikált szoftver később nem fog jól működni. A megrendelők ezáltal nehéz helyzetbe hozzák a fejlesztőket: helyesen működő szoftvereket várnak el a lehető legrövidebb szállítási idővel. Ez az ellentét esetenként nehezen oldható fel, ez a lecke a feloldás egyik lehetőségét vizsgálta.

7.3.2 Önellenőrző kérdések

1. Milyen ellentmondás van a piaci igények és a jó szoftver igénye között?
2. Milyen kompromisszumokat várhatunk el a megrendelőtől a szoftverfolyamat felgyorsítása érdekében?
3. Hogyan befolyásolják a megrendelő igényei a fejlesztő eszköz kiválasztását?
4. Milyen jellemzői vannak a vizuális programozási nyelveknek?

8. LECKE: SZOFTVER-ÚJRAFEL-HASZNÁLÁS

8.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Gondoljuk végig a következőket! A nyomtatógyártó cégek egyre újabb és újabb nyomtatókat dobnak piacra. Ahhoz, hogy ezeket a szövegszerkesztő programok használni tudják, a szövegszerkesztőket újabb és újabb tudással kell felvértezni, az új nyomtatók működését és használatát meg kell nekik tanítani. Amikor a piacon megjelennek a táblázatkezelő szoftverek is, akkor két lehetőség van:

- a szövegszerkesztőkben a nyomtatókkal kapcsolatosan már megismert tudást beírjuk a táblázatkezelő szoftverekbe is,
- vagy kiemeljük azokat a szövegszerkesztőkből, és olyan helyen tesszük őket elérhetővé, ahonnan a szövegszerkesztők és a táblázatkezelők is hozzáférhetnek.

Az operációs rendszerek alkalmazásának okai között az egyik ok éppen ez. Nem kell minden alkalmazói szoftvernek megtanítani a számítógép különböző hardvereinek működését, hanem ki kell őket emelni egy olyan helyre (az operációs rendszerbe), ahonnan minden szoftver elérheti őket.

A példa kicsit esetlen, de a lényeg talán érthető. Ha egyszer elkészítettünk egy olyan programrészt, amelyet máshol is tudunk használni, akkor felesleges azt mindig újra és újra megírni. Ha olyan programrészeket tudunk újrafelhasználni egy fejlesztés során, amelyet korábban már elkészítettünk, akkor a fejlesztés nem csak gyorsabb lesz, de a majdani szoftver működése valószínűleg megbízhatóbbá is válik.

Ez a lecke a szoftver-újrafelhasználás kérdéseit vizsgálja.

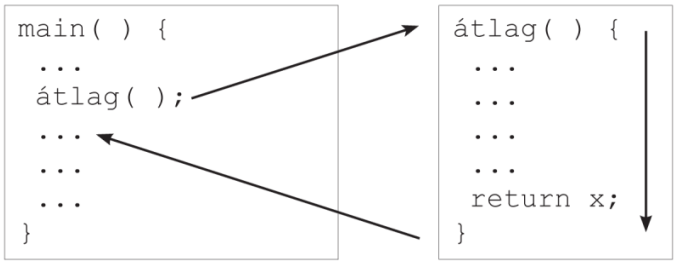
A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerezők épesség, figyelem-összpontosítás.

8.2 TANANYAG

8.2.1 Programelemek újrafelhasználása

Már az 1970-es években felismerték a kód-újrafelhasználás előnyeit, és a strukturált programozás eszközeivel lehetővé is tették a megvalósítását. Ha egy

programon belül egy feladatot többször is el kell végezni azonos, vagy hasonló módon, akkor az ismétlődő kódrészt érdemes kiemelni, és névvel ellátni. A későbbiekben, amikor erre a kódrészletre van szükségünk, akkor már csak hivatkoznunk kell rá a nevével. A strukturált programozás ehhez az alprogramok használatát biztosította.



46. ábra: Alprogram (függvény) használata

Egy névvel ellátott utasítássorozatot eljárásnak, míg egy számítás elvégző, visszatérési értéket eredményező alprogramot függvénynek neveztek. Az alprogramokat és függvényeket a legtöbb ebben az időben megszülető programozási nyelv programkönyvtárakba vagy egységekbe (unit) engedte szervezni. Így a programozók lehetőséget kaptak arra, hogy a munkájuk során gyakran előforduló problémákhoz önálló alprogramgyűjteményeket hozzanak létre, és ezzel jelentős időt takaríthattak meg az újabb szoftverek kifejlesztése során. Bizonyos programkönyvtárakat maguk a fejlesztői környezetek is a programozók rendelkezésére bocsátottak, ilyenek például az ANSI C standard header-állományai, vagy a Turbo Pascal unitjai, amelyeket a gyártó cég szállított a fordítóprogrammal együtt.

A szoftverrendszerek – mint az az előzőekben láttuk – nemcsak a program állományaiból állnak. A szoftverrendszer részeit képezik az adatbázisok, a felhasználói felületeket leíró komponensek, mint ahogyan a dokumentáció is. Az újrafelhasználás ezért természetesen nem csak a forráskódok újrafelhasználását jelenti. Az újrafelhasználás okai között szerepelhet, hogy olyan szoftvert kell kifejlesztenünk, amely a korábban használt rendszer adatait, adatbázisát használja. Ha követelmény, hogy az adatbázis definíciójának változatlan formában meg kell maradnia az új rendszerben is, előfordulhat, hogy a legegyszerűbb az adatbázis megtartása. Hasonló eset áll elő akkor is, ha az új rendszernek integrálnia kell a teljes eddigi rendszert.

Az újrafelhasználás alapja lehet az eddig működő rendszer egy-egy programelemének újból felhasználása. Egy autókereskedő cég újabb hitelintézetekkel köt szerződést, esetleg újabb márkák értékesítésére is szerződést köt más

gyártókkal. Ez szükségessé teszi a most működő rendszer teljes átalakítását, és új rendszer készíttetése mellett döntenek. Az új rendszer ismerni fogja az új bankok hitelügyintézési jellegzetességeit, az új autómárkák típusainak sajátosságait stb. Ezeket a komponenseket valószínűleg meg kell tervezni, specifikációt kell készíteni hozzájuk és implementálni is kell őket. Az új rendszerben viszont felhasználható marad például a régi rendszer marketing moduljából az ügyfelek számára hírlevelet küldő komponens, ez változtatás nélkül bekerülhet az új rendszerbe is.

Az újrafelhasználás legalacsonyabb szintje, amikor eljárásokat, vagy függvényeket, esetleg osztályokat, objektumokat örökítünk át egy régi rendszerből az újba.

Az újrafelhasználásnak egy speciális módjával a legtöbb kezdő programozó előbb-utóbb találkozik. Amikor egy kezdő webfejlesztő teljesíti első megrendelését, és elkészít egy reklámcélú weboldalt, a következő munka során igyekszik felhasználni a már kész weboldalból a lehető legtöbb komponenst. Szembesülni fog azzal, hogy nagyon sok olyan komponens van a weboldalakban, amelyeket általánosan kellene elkészítenie, és a konkrét megrendeléseknél kellene azokat konkrétan megvalósítania. Előbb-utóbb minden programozó készíteni fog magának egy olyan sablonrendszert, egy kvázi-CMS-t (CMS – Content Management System, tartalomkezelő rendszer), amely a jövőben készítendő weboldalak mindegyikéhez felhasználható lesz, így az egyes megrendeléseknél elegendő lesz csak a speciális dolgokra koncentrálnia.

Ez a módszer éppen ellentétes irányú az eddig megszokottakkal. Rendszerint az általános felől haladunk a konkrét megvalósítás felé, ebben a speciális helyzetben viszont az *általános* megalkotását megelőzi a *konkrét* elkészítése. Újrafelhasználás ez is, de ez inkább elméleti, koncepcionális újrafelhasználás, hiszen itt a megszerzett tapasztalatok felhasználásáról van szó annak érdekében, hogy egy általános rendszert lehessen kifejleszteni a már kész, konkrét rendszerek fejlesztésének tapasztalatai alapján.

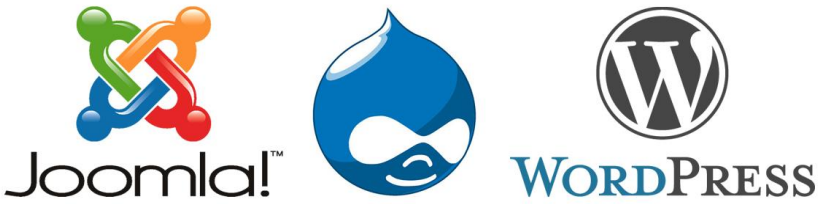
8.2.2 Az újrafelhasználás előnyei és hátrányai

Nagyon sok érv szól a szoftverkomponensek újrafelhasználása mellett, a teljesség igénye nélkül álljon itt most néhány ezek közül:

- Ha egy komponens egy korábbi rendszerben már működött, akkor annak működése bizonyítottan megfelelő, hiszen a korábbi rendszerek már átestek a validáción és verifikáción, illetve a szoftverevolúciójuk során a működésük finomhangolása is megtörténhetett. Ez azt jelenti, hogy egy ilyen komponens működése valószínűleg megbízhatóbb, mint

egy újonnan kifejlesztendő, hiszen ez már átesett a szoftvertechnológia különféle fázisain.

- Ha nagyobb alrendszereket újra fel tudunk használni, akkor azzal csökkenteni lehet az új rendszer bizonytalansági mutatóit, mind költségek, mind a megbízhatóság (és a határidők) szempontjából.
- Ha a CMS-példát tekintjük, előbb-utóbb minden webprogramozó megpróbálkozik olyan általános eszköz kifejlesztésével, amelyet hatékonyan fel tud használni későbbi munkája során. Ilyen CMS-rendszereket azonban nagyobb cégek is fejlesztenek, tapasztaltabb programozók munkájának felhasználásával. Ha a kezdő programozó ilyen, mások által fejlesztett és már bizonyítottan jól működő CMS-eket használ fel munkája során, azzal lényegében tapasztalt szakemberek szaktudását használja fel, és az általa fejlesztett szoftver sok tekintetben megbízhatóbb működésű lesz, mintha a saját rendszerét használta volna. Jegyezzük meg azonban, hogy ez az állítás csak nagy általánosságban igaz, viszont nemcsak CMS-rendszerek esetében.



47. ábra: Közismert CMS-ek: Joomla!,Drupal, WordPress

- A vizuális programozási nyelvekben, vagy akár a Java nyelv Swing csomagjában található űrlapvezérlő komponensek szabványosan működnek. Egyrészt a programozónak nem kell időt töltenie azzal, hogy a megszokott módon működő legördülő listamezőket fejlesszen, másrészt a felhasználóknak nem kell elmagyarázni a komponensek használatát, hiszen más rendszerekből ezeket már ismeri. A szabványos felhasználói felületek használata növeli a megbízhatóságot, mert a használók kevesebbet hibáznak, ha olyan felületeken dolgoznak, amelyeknek az elemeit már rutinszerűen tudják használni.
- A komponensek újrafelhasználása minden esetben csökkenti a fejlesztés időtartamát, így a megrendelő hamarabb juthat hozzá a kész szoftverhez.

Az előnyök mellett számolnunk kell bizonyos hátrányokkal is. Lássunk ezek közül néhányat:

- Ha olyan komponenst használunk fel saját fejlesztésünkben, amelynek az eredetijét nem mi magunk készítettük, akkor két eset lehetséges: az egyik, hogy az általunk használt komponens forráskódja nem elérhető. Ebben az esetben a módosítás, illetve az esetleges rejtett hibák javítását mi magunk valószínűleg nem tudjuk elvégezni. Ha a forráskód rendelkezésre áll, akkor viszont annak elemzése és megértése sokszor nagyobb energiát követelhet, mint magának a komponensnek az elkészítése. Különösen igaz ez abban az esetben, ha más programjában kell rejtett hibákat keresnünk, feltárnunk azok okait és megoldást találni rájuk.
- CMS-rendszerekkel kapcsolatban éppen ezt szokták az egyik hibaként említeni. Ameddig megelégszünk a más által fejlesztett CMS adta szolgáltatásokkal, addig a fejlesztés nagyon gyors és egyszerű. Amint azonban a megrendelő speciális igényeket fogalmaz meg, ezek megvalósítása egy működő CMS-ben nagyon nehézkes, de akár lehetetlen feladat is lehet.
- A programozókban mindig is létezett bizonyos fokú ellenérzés más programozókkal szemben. Előfordul, hogy egy programozó egyszerűen nem bízik meg más fejlesztők munkájában, és akkor is újraír egy egyébként létező és működő komponenst, ha az ő megoldása sem lesz jobb, esetleg még rosszabb is lesz.

8.2.3 Újrafelhasználás vagy új összetevő fejlesztése?

A fentiek alapján látható, hogy az újrafelhasználás nagyon sok esetben kívánatos, követendő eljárás, vannak azonban ellenérvek is vele szemben. Milyen esetben érdemes, és milyen esetben nem érdemes az újrafelhasználás lehetőségével élni? A válaszhoz az alábbiakat kell figyelembe vennünk:

- A rendelkezésünkre álló idő: ha nagyon szűk határidővel kell dolgoznunk, az újrafelhasználást, és lehetőség szerint minél nagyobb alrendszerek újbóli felhasználását akkor is célszerű megvalósítanunk, ha a végeredmény nem fog tökéletesen illeszkedni a megrendelő által leírt követelményekhez.
- A szoftver felhasználásának várható időtartama: ha a szoftvert várhatóan relatíve rövid ideig (napok, hetek, néhány hónap) fogják használni, például azért, mert ezt követően egy másik, új rendszert fognak alkalmazni, megint az újrafelhasználás mellett érdemes dönteni. Ha a szoftver várható élettartama több év, akkor mindenképpen számolnunk kell az evolúció, illetve a karbantartás nehézségeivel egy újrafelhasznált rendszer esetében. Ilyenkor megfontolandó az új komponens tervezése, specifikálása és implementálása.

- Emberi erőforrások: egy nagyméretű, komplex rendszer fejlesztése hónapokig, akár évekig is tarthat. Ennek megfelelően várhatóan hosszabb ideig biztos munkát ad a fejlesztőknek. Ilyen körülmények között a fejlesztésben részt vevő szakemberek szívesebben vállalják a szükséges továbbképzéseket, tréningeken való részvételt, mint egy rövid ideig tartó, anyagi szempontból kisebb eredményt ígérő projekt esetében. Egy néhány hetes fejlesztési folyamat kedvéért valószínűleg egy fejlesztő sem vehető rá arra, hogy egy általa addig nem ismert programozási nyelvvel, környezettel ismerkedjen meg, főleg, ha ez anyagilag sem éri meg neki a várható bevétel ismeretében. Ezért egy fejlesztő cég igazgatója sem jár el megfelelőképpen, ha a feladathoz esetleg jobban illeszkedő programozási környezet használatát erőlteti rá az alkalmazottjaira. Ha az alkalmazottak nem ismerik ezt a környezetet, a fejlesztés időtartama elhúzódhat, és a végeredmény minősége is alul fogja múlni a várakozásokat.
- Alkalmazási terület: bizonyos szakterületeken (egészségügy, atomenergia felhasználása, űrtechnológia stb.) léteznek olyan általános termékek, amelyek sokkal egyszerűbben használhatók fel a megfelelő problémához igazított konfigurálással, mint ezek újraírásának elvégzése. Ilyen esetekben mindenképpen az újrafelhasználást kell mérlegelni, hiszen megtakarítható a szakismereti képzésre fordítható idő, vagy képzett szakemberek bevonásának költségei a fejlesztési folyamatban.
- A felhasználók által használt platform. Lehet, hogy elegánsabb egy probléma megoldásához egy „komoly” programozási nyelv és környezet használata, de ha a felhasználó a fejlesztendő alkalmazást, pl. egy raktárnyilvántartó kisalkalmazást csak egy számítógépen és csak egymaga fogja használni, ne habozzunk a Microsoft Accesst választani a fejlesztéshez. Fogalmazhatnánk így is: „Verébre ne lőjünk ágyúval.”

8.2.4 Az újrafelhasználás speciális esetei: alkalmazáscsaládok

Amikor a megrendelő igényei speciálisak, az a fejlesztést bonyolultabbá, vagy éppen könnyebbé is teheti. Speciális igény lehet, hogy a kész alkalmazás csak Windows alatt működjön és csak egy munkaállomáson, vagy egy helyi hálózatba kötött néhány munkaállomáson. Ha ennek az alkalmazásnak tárolt adatokat kell karbantartania, a fejlesztés Microsoft Accessben elvégezhető.

A speciális igények másik véglete, amikor a kliensoldalon többféle platform és többféle eszköz is lehet, és ezekhez mind felhasználói felületet kell készítenünk. Ha az alkalmazás kliens-szerver típusú, akkor a kliensoldali felületek elké-

szítéséhez kínálja magát a HTML, PHP, JavaScript és MySQL, vagy ezeket kiegészítő/helyettesítő egyéb webes eszközök használata. Ha viszont a megrendelő olyan szoftvert kér, amely többféle platformon is fut, de nem kliens-szerver típusú alkalmazásról van szó (például speciális nyomtatványkitöltő és nyomtatványszerkesztő szoftver), esetleg a majdani munkaállomások nem is feltétlenül érik el a hálózatot, csak az a lehetőség marad, hogy többféle platformon is elvégezzük a fejlesztést.

Ebben az esetben az újrafelhasználás nyilvánvalóan nem a forráskódok, vagy a létező objektumok újrafelhasználását jelenti. Újrafelhasználhatók viszont a modellek, specifikációk, prototípusok. Ebben az esetben lényegében a funkcionalitás az, ami újrafelhasználható.

A funkcionális specializáció egyik esete, amikor egy alkalmazást különböző méretű, de hasonló funkcióval működő egységek részére kell elkészítenünk. Egy könyvtári rendszer funkciói alapvetően azonosak a könyvtár méretétől függetlenül, mégis indokolt lehet a rendszer különböző funkcióinak megfelelően közkönyvtári, kézikönyvtári vagy egyetemi könyvtári verziót is készíteni a szoftverből.

A különböző platformokon futó, de azonos funkcionalitású, vagy a különböző méretekre skálázott alkalmazások verzióinak fejlesztésekor beszélünk alkalmazáscsalád-fejlesztésről. Ezeknél mindig szükséges az adott rendszer adott környezetben való konfigurálása. A konfigurálás történhet tervezési időben, vagy kihelyezési időben. Tervezési időben a fejlesztők végzik a konfigurálást. Az alkalmazáscsalád azon részeit, amely minden platformon azonos működésű (platform független) kiegészítik az adott platformra jellemző specifikációkkal. A kihelyezési időben végzett specializációt telepítéskor akár maga a felhasználó is elvégezheti. Ennek során állítja be a rendszer jellemzőit, kifejezetten az aktuális futtató környezet tulajdonságaihoz igazítva.

Az alkalmazáscsaládok fejlesztése hasonlít ahhoz a példához, amelyet a CMS-rendszerkkel kapcsolatosan mutattunk be. Sokszor előfordul, hogy egy platformon valamilyen újonnan megjelenő szoftver annyira sikeres lesz, hogy felmerül más platformokon való megjelentetése is. Ekkor a fejlesztés, az evolúció során a fejlesztő cég előbb-utóbb azt a megoldást fogja választani, hogy a következő verzió készítéséhez teljesen újraírja az eddigi kódot. Az eddig elkészült konkrét alkalmazások tapasztalatait felhasználva így egy olyan általános verziót készít, amelyet sokkal könnyebb lesz ezután a különböző platformokon karbantartani.

Alkalmazáscsaládok kialakítása során a következő folyamatot érdemes követni:

- A legfontosabb követelmények meghatározása. Ebben a lépésben azt kell megállapítani, hogy melyek azok a követelmények, amelyeket a szoftvercsalád minden tagjától minimálisan elvárnak, és a különböző platformokon ezek hogyan valósíthatók meg.
- A jelenleg létező családtagok összehasonlítása. Ki kell választani azt a családtagot, amelyik jelenleg a legjobban megvalósítja a követelményeket, és az újratervezést ennek elemzésével kell kezdeni.
- A követelmények újbóli megvitatása. Ahhoz, hogy az újonnan kifejlesztendő, általános változat a leghatékonyabban legyen tervezhető, szükségessé válhat a korábbi verziókban elvárt követelmények módosítása. Ne feledjük: a korábbi verziók úgy készültek, hogy egy már meglévő alkalmazást írtak meg egy másik platformon is. Ez az átirat az eredeti platform alkalmazásának követelményeit követte, nem pedig az új platform lehetőségeit.
- Adaptálás. Miután a követelményeket újra megvizsgáltuk, a legjobban működő családtag alapján elkészült változatot alapnak tekintve új komponenseket fejlesztünk. A legjobb változat rendszermoduljait átdolgozzuk a többi platformon való futás biztosításához.

8.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

8.3.1 Összefoglalás

Ebben a leckében megvizsgáltuk a különböző programegységek újrafelhasználásának lehetőségeit. Újrafelhasználhatók egyszerű algoritmusok, eljárások vagy függvények, de nagyobb programegységek is, vagy akár teljes alrendszerek, rendszerek. Az újrafelhasználástól a fejlesztési idő lerövidülését, és a fejlesztett rendszer megbízhatóságának növekedését (de legalábbis nem csökkenését) várjuk. A leckében megismertük a szoftver-újrafelhasználás előnyeit és hátrányait, illetve speciális eseteit.

8.3.2 Önellenőrző kérdések

1. A programok milyen egységeit lehet újrafelhasználni?
2. Melyek az újrafelhasználás előnyei?
3. Melyek az újrafelhasználás hátrányai?
4. Milyen esetben nem célszerű az újrafelhasználás?
5. Mik azok az alkalmazáscsaládok?
6. Mi az alkalmazáscsaládok kialakításának folyamata?

9. LECKE: KOMPONENSALAPÚ SZOFTVERFEJLESZTÉS

9.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A szoftver-újrafelhasználásról szóló leckében láttuk, hogy az újrafelhasználás csökkentheti a fejlesztési költségeket, és növelheti a szoftver hatékonyságát, megbízhatóságát.

Ebben a leckében azt vizsgáljuk meg, hogy milyen programrészek újrafelhasználása a legkifizetődőbb, honnan juthatunk mások által kifejlesztett komponensekhez, mennyire tekinthetjük ezeket megbízhatónak, és milyen nem várt következményei lehetnek a komponensek felhasználásának.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív gondolkodás képessége, deduktív gondolkodás képessége, absztrakt (elméleti) gondolkodás, áttekintő képesség, rendszerező képesség és figyelemösszpontosítás.

9.2 TANANYAG

9.2.1 Komponensek újrafelhasználása

Az előző leckében megmutattuk, hogy az újrafelhasználás megtörténhet a legapróbb eljárások, függvények újbóli alkalmazásától a teljes alrendszerek, akár rendszerek újrafelhasználásig bármilyen dimenzióban. Az eljárások, függvények, vagyis az alapvetően programozásinyelv-specifikus elemek felhasználása rendszerint nem kellően hatékony, a nagy rendszerek újrafelhasználása pedig a későbbi karbantartási munkákat befolyásolhatja. Ezekből kifolyólag az újrafelhasználás leginkább a komponensek újrafelhasználásában mutatkozik meg.

A komponensalapú szoftverfejlesztés (component-based software engineering) során a fejlesztők kiválasztják, rendszerbe foglalják azokat a már létező szoftverkomponenseket, amelyek az adott feladat elvégzésére a leginkább alkalmasak, majd az új rendszert az így összekapcsolt komponensek segítségével implementálják. Ennek a módszernek az előnyeiről korábban már esett szó:

- a fejlesztési idő lerövidíthető,

- a fejlesztett rendszer megbízhatósága növelhető, hiszen az integrált komponensek bizonyítottan jól működnek,
- hatékonyan használhatunk fel magas szintű szakmai tudást úgy, hogy azzal mi magunk nem is feltétlenül rendelkezünk.

Az előre elkészített komponensek egymástól függetlenek, így egymás működését nem befolyásolják. A komponensek – mivel erősen kötődnek az OOP stratégiáihoz – implementációja el van rejtve, az egyes összetevők csak a komponensek interfészein keresztül kommunikálnak. Ennek az az előnye, hogy egy komponens implementációja anélkül változtatható meg utólag, hogy az őt tartalmazó rendszert át kellene alakítani. Mivel a komponensek csak az interfészeken keresztül kommunikálnak, egy komponens kicserélhető egy azonos funkcionalitású, esetleg hatékonyabb működésű komponensre, ha az interfészeik megegyeznek.

Természetesen ezek az előnyök csak optimális esetben érhetők el. A komponensalapú fejlesztés során szembesülnünk kell esetleges hátrányokkal is:

- Nem lehetünk teljesen biztosak a komponens megbízhatóságában. A komponensek elrejtik a működésüket a külvilág elől, csak az interfészeiken keresztül tudunk velük kommunikálni. A bemenő adatokból kimenő adatokat állítanak elő, de a számítási módot nem ismerhetjük. Ha nem rendelkezünk a komponens forráskódjával, csak a működéséből, kizárólag próbálgatással nem feltétlenül tudjuk visszafejteni a teljes funkcionalitást. Lehet, hogy a komponensben vannak olyan kódrészek is, amelyek előlünk rejtett módon befolyásolják a teljes rendszer működését. Szélsőséges esetben az is előfordulhat, hogy egy nyilvánosan közzétett, szabadon felhasználható komponens akár ártalmas kódot (vírus, trójai stb.) is tartalmazhat.
- A komponensek rendelkezhetnek tanúsítványokkal, pl. digitális aláírással. Esetenként az is elegendő lehet, ha biztosak vagyunk a komponens készítőjének személyében, és maximálisan megbízunk benne. Az interneten elérhető nyilvános komponensek többsége esetében a tanúsítványok vagy teljesen hiányoznak, vagy nem ismerjük a tanúsítvány kiállítóját. Nem sokkal nagyobb a garancia egy programegység megbízhatóságával kapcsolatosan, ha egy általunk nem ismert gyártó termékét egy általunk szintén ismeretlen minősítő cég látja el tanúsítvánnyal.
- A komponensek külön-külön eltérő funkcionalitásúak lehetnek, mint azonos rendszerbe integrálva. Lehet, hogy két komponenst külön-külön felhasználva egy rendszerben azt találjuk, hogy tökéletesen működnek, ugyanakkor őket egyszerre felhasználva egyazon rendszerben már váratlan jelenségeket, vagy hibákat produkálnak.

- Valószínűsíthető, hogy bármennyire sok előre elkészített komponenst használhatunk a fejlesztés során, pontosan olyan komponenst nem fogunk találni, amelyik minden, az általunk fejlesztett rendszer specifikációjában szereplő követelménynek meg fog felelni. Emiatt kompromisszumokat kell kötnünk: melyek azok a specifikációban megfogalmazott követelmények, amelyekről hajlandóak vagyunk lemondani. Ha ez nem vezet megoldásra, a komponensről kell lemondanunk, és saját magunknak kell fejlesztenünk a programegységet.

9.2.2 Komponensek jellemzői

A szakirodalom nem ad teljesen egységes képet a komponensekkel szemben támasztott követelményeket illetően. Általános megállapodás szerint a komponens olyan önálló szoftver-alkotóelem, amely más komponensekkel összeilleszthető egy szoftverrendszerben. Ezt kiegészíthetjük a korábban már említett jellemzőkkel: a komponens elrejti a külvilág elől a belső állapotát, a működését, és csak az interfészein keresztül lehet vele kommunikálni. A komponensek előre lefordított programegységek, a forráskódjuk nem mindig elérhető, ezért az általunk fejlesztett szoftverrendszerbe építéskor ezeket nem kell külön lefordítani.

Ahhoz, hogy egy komponenst használhassunk, először is hozzá kell férnünk. A ma használatos programozási nyelvek és fejlesztői környezetek eleve nagyon sok kész komponenst adnak a programozó kezébe, de az interneten is rengeteg, jól működő és megbízható szoftver-alkotóelem elérhető. A használathoz szükséges, hogy rendelkezzünk a komponens dokumentációjával. Ebben le vannak írva azok a kezelőmetódusok, amelyekkel a komponens használható.

- Az objektumorientált szemlélet szerint az objektumok egymással együttműködő programegységek. Ha a komponenseket is objektumként, objektumok egy csoportjaként tekintjük, akkor ezekről is elmondhatjuk azt, hogy a velük folytatott kommunikáció lényege az, hogy üzeneteket küldünk nekik, amelyeket azok feldolgoznak és valamilyen reakciót adnak rájuk. Az üzeneteket rendszerint metódushívásokkal valósíthatjuk meg. Ahhoz, hogy egy nem általunk fejlesztett komponenst használni tudjunk, üzeneteket kell tudnunk küldeni a számára, vagyis ismernünk kell azokat a metódusait, amelyeket a külvilág számára elérhetővé tesz. Ehhez van szükségünk a komponens dokumentációjára.

Sajnos ezek a tulajdonságok nem kellően részletezettek, nem kellően konkrétak. Az elérhető komponensek is különböző mértében valósítják meg őket. A legfontosabb kritériumok az alábbiak, ezeket minden komponenstől elvárhatjuk:

- Önálló, egymástól független egyedek. A forráskódjukat nem ismerjük. Képesek más komponensekkel való együttműködésre.
- A kommunikáció interfészeken keresztül valósul meg. A komponens belső működése rejtett a külvilág elől.

Habár a fentiek alapján úgy tűnhet, hogy a komponensek szükség szerűen egyben objektumok is és a komponensek fejlesztése nagyrészt valóban OOP-környezetben történik, néhány dologban különböznek az obejtumoktól:

- Nem kötődnek konkrét programozási nyelvhez.
- A komponens konkrét, míg az objektumok osztályai absztraktak. A komponens maga a példány, ebből további példányok nem hozhatók létre.
- A komponens le van fordítva gépi kódra, tehát a szoftverrendszer fejlesztésekor a fordítóprogram nem fordítja le (újra). Ehelyett a futtatási környezetben kell telepíteni, és az általunk fejlesztett szoftver lefordított változata tudja használni (komponenshívás).
- Szerencsés esetben szabványosak. Míg egy általunk fejlesztett objektumosztály tetszésünk szerint implementálható, a komponensek szerencsés esetben szabványokat követnek. A komponensmodellek implementációja így rendszerint bizonyos megkötések alapján történik.

9.2.3 Szoftverfolyamat a komponensalapú fejlesztés mellett

A 2. leckében bemutattuk a szoftverfolyamat általános lépéseit. Abban a leckében azt feltételeztük, hogy az implementációt a fejlesztő cég kizárólag saját programozóival végzi. Az implementációt megelőző specifikációban így teljes mértékben alapul lehet venni a megrendelő féllel a tervezés, modellezés során kialakított igényeket, ötleteket. Az implementáció pedig nemcsak követni tudja, de követnie is kell a specifikációban leírottakat.

Komponensalapú fejlesztés során a szoftverfolyamat kissé módosul. Szem előtt kell tartanunk, hogy bármennyire is nagy a komponensek szabad piaca, majdnem biztos, hogy csak a legáltalánosabb feladatok megoldásához találunk olyan komponenst, amely szász százalékosan megfelel az elvárásainknak. Sokkal valószínűbb, hogy sok olyan komponenst fogunk találni, amelyek szóba jöhetnek a fejlesztésben való felhasználáshoz, és ezek közül fogjuk kiválasztani bizonyos szempontok alapján a legmegfelelőbbet. Ez már a specifikációt megelőző tervezéskos, modellalkotáskor is befolyásolni fogja a szoftverfolyamatot.

A komponensalapú szoftverfejlesztés folyamata vázlatosan a következő:

- Kezdetben csak a legalapvetőbb, legfontosabb követelményeket határozzuk meg. A tervezésben, modellezésben résztvevőket (pl. majdani felhasználók) arra bíztatjuk, hogy lehetőség szerint ebben a fázisban minél kevesebb specifikus igényt fogalmazzanak meg, és a hangsúlyt inkább a globális tervek kialakítására fektessék.
- Próbáljunk a globális tervekhez minél jobban illeszkedő komponenseket keresni. Mivel szinte biztos, hogy nem fogunk az igényeknek száz százalékosan megfelelő egyedeket találni, a megrendelővel tekintsük át, melyek azok a követelmények, amelyek nem érhetők el az általunk javasolt komponensben. A megrendelők nagy valószínűséggel hajlandók lesznek kompromisszumokat kötni annak érdekében, hogy a készülő szoftver megbízható működésű legyen, valamint kisebb költségvetésből és hamarabb készülhessen el.
- Ahogyan arról már korábban szó esett, az egyazon rendszerbe integrált komponensek működése nem feltétlenül egyezik meg a komponensek külön-külön működésének összegével. Sajnos az a ritkább eset, hogy több komponens egyazon rendszerbe integrálásakor nem várt, pozitív jelenségekkel találkozzunk, vagyis hogy az együtt működő komponensek eddig dokumentálatlan többletfunkcionalitást eredményeznek. Sokkal gyakoribb az az eset, hogy a komponensek egy rendszerbe való integrálása a komponensek közötti diszfunkciókat eredményez. A rendszer nem várt működést produkál, esetleg hibajelenségek lépnek fel, amelyekről nem találunk dokumentációt, mert más fejlesztők korábban nem használták éppen e két komponenst egyszerre. Mivel a komponensek belső működése rejtett, valószínűleg nem tudjuk megoldani a problémát, bármilyen egyszerű lenne is az a forráskódok ismeretében. Ilyenkor csak az a megoldás marad, hogy egy vagy több komponensről lemondunk, és ismét kérjük a megrendelőt a tervek finomítására, a követelmények csökkentésére.
- Olyan szoftverrendszert sohasem fogunk tudni fejleszteni, amelyhez kizárólag csak előre gyártott komponenseket kell összeilleszteni. Ez hasonló lenne ahhoz az esethez, amikor sok különböző puzzle-játékból véletlenszerűen kiválasztunk egy vagy több darabot, és ezekből nemhogy össze tudnánk rakni egy téglalap alakú képet, de az a kép ráadásul még mutatna is valamit, méghozzá valami minőségileg új dolgot. (Pl. sok-sok kiscicás, kiskutyás stb. képet ábrázoló puzzle-játékból kivett kirakódarabokból egy havas tájképet tudnánk összerakni.)
  Biztos, hogy szükséges lesz a komponensek összeillesztéséhez saját kódot is implementálnunk, illetve azokat a funkciókat, amelyekhez nem találtunk kész komponenst, min magunk fogjuk leprogramozni.

9.2.4 Komponensek keresése és kiválasztása

Honnan juthatunk megbízhatóan működő komponensekhez? A válasz csak látszólag egyszerű: megbízható komponensfejlesztőktől és -forgalmazóktól. Vajon kit tekinthetünk megbízhatónak? Ez nem csupán bizalom kérdése. Étteremvezetőként hiába a legjobb barátunk gyerekkorunk óta egy sajtkészítéssel, tejtermékek előállításával és forgalmazásával foglalkozó gazda, ha ő nem rendelkezik a szükséges tanúsítványokkal, az éttermünkben nem használhatunk fel tőle származó alapanyagokat. Hiába vannak egy ismerőseink egy szoftvercégnél, akik évek óta megbízható partnereink, ha nem rendelkeznek a megfelelő tanúsítványokkal, nem használhatjuk fel az általuk gyártott komponenseket például egy kritikus rendszer fejlesztése során.

A komponensek piaca bár kialakulóban van, még távolról sem úgy működik, mint pl. a zeneipar termékeit áruló e-boltok. A rendelkezésünkre álló, vagy egyáltalán szóba jöhető komponenseket elsőként célszerű az általunk használt programozási nyelv vagy fejlesztői környezet weboldalán keresnünk. Ha egy programozási nyelv, illetve a hozzá kapcsolódó fejlesztői környezet gyártója egy komponenst elérhetővé tesz a weboldalán, akkor abban többé-kevésbé akkor is megbízhatunk, ha azt nem az adott cég fejlesztette.

Feltétlenül óvatosnak kell lennünk, de talán nem túlzás, hogy óvakodnunk kell az interneten talált ingyenes komponensek feltétel nélküli felhasználásától. Bizalmunkat növelheti, hogy más programozók is találkozhattak már hasonló problémával, és esetleg ők maguk már ki is fejlesztették a megoldáshoz szükséges komponenst, vagyis egykor ők is hasonló cipőben jártak. Ha a komponens mellé dokumentációt és esetleg referenciákat is mellékeltek, illetve megtalálható a fejlesztő elérhetősége, esetleg fórumbejegyzésekben a további felhasználók pozitív bejegyzéseit találjuk, akkor megfontolandó a komponens használata. De nem lehetünk elég óvatosak, hiszen rejtett hibája még egy megbízható forrásból származó, jól dokumentált és alaposan tesztelt komponensnek is lehet, amely esetleg csak más komponensekkel együtt jelentkezik. Ha egy komponens hibája menthetetlen adatvesztést eredményez, a megrendelő aligha lesz megértő.

Azzal is számolnunk kell, hogy az általunk választott komponens funkcionalitása bővebb, mint amire nekünk szükségünk lesz. Ilyenkor a számunkra szükségtelen funkciókat egyszerűen nem hívjuk meg, vagy ha módunk van rá, esetleg el is távolíthatjuk a komponensből. Ez viszont esetleg nem várt (talán katasztrofális) eredményekkel járhat, ha a törölt részekben olyan programkódok is szerepeltek, amelyekre nekünk közvetlenül nem volt szükségünk, viszont az általunk használt funkciók közül valamelyiknek vagy valamelyeknek igen.

9.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

9.3.1 Összefoglalás

A komponensalapú fejlesztés ma a legnépszerűbb szoftverfejlesztési eljárások közé tartozik. Előnye, hogy a szoftverköltségek csökkenthetők, a fejlesztési idő lerövidíthető, és az általunk fejlesztett szoftver hatékonysága nőhet. Ennek többek között az is az oka, hogy mások, esetleg nálunk tapasztaltabb szakemberek által kifejlesztett komponensek olyan szaktudást is hordozhatnak, amelyeknek mi magunk nem is vagyunk a birtokában.

A komponensalapú fejlesztésnek azonban lehetnek hátulütői, buktatói is.

Ebben a leckében a fejlesztésnek e módját, a lehetséges előnyöket és hátrányokat vizsgáltuk meg.

9.3.2 Önellenőrző kérdések

1. Milyen érvek szólnak a komponensalapú fejlesztés mellett? Milyen előnyei vannak ennek a módszernek?
2. Mit nevezünk komponensnek?
3. Milyen jellemzői vannak a komponenseknek?
4. Hogyan változik a szoftverfolyamat a komponensalapú fejlesztés során?
5. Hogyan tudunk komponensekhez jutni? Hogyan választható ki a legalkalmasabb komponens?

10. LECKE: SZOFTVEREVOLÚCIÓ, SZOFTVERMÓDOSÍTÁSI LEHETŐSÉGEK, REFAKTORÁLÁS

10.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A szoftverfejlesztés nem ér véget a rendszernek a felhasználóhoz történő szállításával és telepítésével. Az átadás pillanata lezárja a fejlesztés egyik szakaszát, de megkezdődik az újabb szakasz. A szoftverevolúció magába foglalja a kész rendszer karbantartását, a használat során felmerülő igények szerinti továbbfejlesztést, az esetleg kiderülő hibák javítását.

A használat során felmerülő újabb igények nem feltétlenül a megrendelő hanyagságából következnek, hiszen joggal gondolhatnánk, hogy azért vett részt már a modellalkotás és a specifikáció készítésének folyamatában, hogy olyan rendszert készíthessünk, amely valóban az ő igényeire van szabva. Azzal nem számolhatott sem a megrendelő, sem a fejlesztő, hogy a kész szoftver használata során megváltozhatnak a piaci folyamatok, esetleg törvényi vagy más jogszabályi változások következnek be. Előfordul az is, hogy a megrendelő új operációs rendszert vásárol, és szeretné a régi rendszert az új platformon is használni. Ezeket a változásokat kész szoftvernek is követnie kell.

A 10. lecke a szoftverevolúcióval, a szoftver utólagos módosításának lehetőségeivel, illetve egy magas szintű szoftvermódosítási eljárással, a refaktorálással foglalkozik.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező képesség, figyelem-összpontosítás.

10.2 TANANYAG

10.2.1 Szoftverevolúció

Egy kész szoftver átadás utáni módosításának többféle oka lehet. Az üzemszerű karbantartás részét képezheti a szoftver működésének rendszeres ellenőrzése. Szükségessé válhat a tárolt adatok olyan karbantartása, amely a szoftverbe épített funkciókon keresztül nem érhető el, például – biztonsági okok miatt – a szoftver nem képes a szükségtelenné vált adatok garantáltan végleges törlésére. Gyakori eset, hogy felhasználói igények, esetleg törvényi, jogszabályi

változások miatt szükségessé válhat a szoftver egyes részeinek átalakítása, átírása, illetve újabb igények felmerülésekor új komponensek készítése és integrálása. Előfordulhat az is, hogy a szoftvert kiszolgáló hardver, vagy operációs rendszer frissítése, cseréje miatt válik szükségessé az általunk készített szoftver karbantartása.

Ahhoz, hogy ezeket a változtatásokat lehetőség szerint minél egyszerűbb legyen végrehajtani, már a szoftver tervezésekor fel kell készíteni azt a majdani változtatásokra, továbbfejlesztésre. Többek között itt derül ki, hogy a specifikáció, a tervezés és az implementáció stb. során mennyire volt körültekintő és teljes körű a dokumentáció. A szoftver jövője nem függhet attól, hogy a szoftvercég alkalmazottai mennyire emlékeznek vissza egy évekkel korábban kitalált rekordszerkezetre vagy éppen egy kereső eljárásra. Nem beszélve arról, ha a szoftvercég egykori alkalmazottai esetleg azóta más munkahelyen dolgoznak, vagy már a szoftvercég sem létezik.

A szoftverevolúció ezért a szoftver átadásakor nem fejeződik be: végigkíséri a teljes életciklusát.

A szoftver telepítés utáni karbantartása nem egyszerűen a későn felfedezett hibák javítását jelenti. Sokkal nagyobb mértékben válik szükségessé a az új követelmények megjelenése miatti módosítás. A leszállított változatra úgy is tekinthetünk, mint egy új szoftver tervét képező jelenlegi rendszerre. A megrendelő új szoftvert szeretne készíttetni, mert a jelenleg rendelkezésére álló rendszer már nem kielégítő működésű. A megrendelő változtatásokat javasol, és elkezdődhet a következő verzió specifikálása, megtervezése. Ez a módszer természetesen azt feltételezi, hogy a megrendelő elégedett volt az eddigi munkánkkal, és megbízik bennünk a jövőben is, vagyis az új verziót is mi fogjuk készíteni, vagy a másik oldalról: a jelenlegi rendszert is mi fejlesztettük.

Ha nem ez a kedvező állapot áll fenn, akkor vagy nekünk lesz szükségünk az előző verzió dokumentációira, vagy mi fogjuk azt rendelkezésre bocsátani.

A szoftverfolyamat alapvetően azt feltételezi, hogy egy egyedi szoftver leszállításakor a megrendelő megkapja a dokumentációt is, amelyben megtalálható a specifikáció utolsó, elfogadott változata is. Ha a rendszer jelenlegi változatát mi gyártottuk, de a következő verzióra felkért gyártó nem kapta meg a specifikációt, akkor vagy a megrendelő volt hanyag és elvesztette, vagy oda sem adtuk neki.

10.2.2 A karbantartást befolyásoló tényezők

A rendszerkarbantartás nélkülözhetetlen folyamat. A karbantartás a környezeti változásokat követi, és az átalakított szoftverrel az evolúció újrakezdő-

dik. Ha egy rendszer megváltozik, akkor a struktúrája romlani fog. Ez ellen úgy tehetünk, ha már a specifikáció során felkészülünk az evolúció során bekövetkező, vagy valószínűsíthető változásokra. Ennek a specifikáció és a tervezés során többletköltségei vannak, azonban ezek a költségek az evolúció ideje alatt megtérülhetnek.

A rendszerek karbantartását, karbantarthatóságát maguk a rendszerek is befolyásolják. Minél összetettebb a rendszer, annál bonyolultabb a megértése. Egy bonyolult rendszer fejlesztése, karbantartása során a programozók többet hibázhatnak, ezért kívánatos, hogy egyszerre csak kicsiket változtassunk a rendszeren. Az egy lépésben véghezvitt nagy változtatás sokkal több hibalehetőséget rejt magában.

Szintén hibalehetőséget rejtenek az újonnan beépített funkciók. Minél több új funkcionalitást adunk egy működő rendszer új verziójához, annál több a potenciális hibalehetőség. A sok új funkciót integráló új verzió megjelenése utáni következő verzióban vélhetőleg ennek a verziónak a hibáit fogjuk kijavítani, és ez az utolsó változat kevés (kevesebb) új funkciót fog a rendszerhez adni.

10.2.3 Karbantartás

A szoftverkarbantartás műveletei alapvetően három csoportba sorolhatók:

Szoftverhibák javítása

Ez jelentheti a programozók által vétett, vagy a tervezési hibák javítását, rendszerint az előbbi kategória hibáinak javítása kevésbé költséges és gyorsabban elvégezhető.

Az implementáció során a programozók szintaktikai és szemantikai hibákat vétethetnek. A szintaktikai hibák a nyelv grammatikáját, nyelvtani szabályait sértő hibák, a szemantikai hibák inkább a nyelvhelyességi hibákhoz hasonlíthatók. Egy adott programozási nyelven írott forráskódban szintaktikai hiba ritkán maradhat rejtve, mert a programozási nyelvek fordítóprogramjai ezeket rendszerint felismerik, és a fordítás meg sem történik a hibák kijavításáig. Szemantikai hiba lehet például, ha egy sorozat nem létező elemére hivatkozunk, vagy megengedjük egy műveletben a nullával való osztást. Az ilyen hibák nagy részét a fejlett fordítási technológiák miatt a fordítóprogramok egyre inkább ki tudják szűrni, de előfordulhatnak olyan esetek, amikor a hiba nem jelezhető előre, és csak futásidőben jelentkezik.

A tervezési hibák már nem a programozók figyelmetlenségének számlájára írhatók. A tervezési hibák több, akár nagy alrendszerek módosítását is igényel-

hetik, és szükségszerűen költségesebbek (pénzben, időben, humán erőforrásban), mint a programozási hibák javítása.

A környezet változása miatt szükségessé vált karbantartások

Ha a megrendelő új operációs rendszert, új hardvert vásárolt, vagy egyéb, a környezetet befolyásoló tényező változott meg, a rendszert hozzá kell igazítani az új környezethez. Szerencsés esetben a beruházásról nemcsak utólag értesülünk, így van időnk felkészülni a bekövetkező változásokra.

Funkcionalitás kibővítése

Ilyen karbantartásra akkor van szükség, ha a rendszerrel szemben támasztott követelmények megváltoztak.

A gyakorlatban e három kategória nem mindig válik el élesen. Ha egy új operációs rendszer alá kell hangolni a szoftvert, előfordulhat, hogy új funkciókkal kell bővíteni a működését.

10.2.4 Hibakezelés

Habár nem tartozik szorosan a szoftverevolúció témaköréhez, érdemes megemlítenünk, hogy a szoftver átadása után jelentkező hibák nagy része abból adódik, hogy a felhasználók rosszul, vagy előre nem látható módon kezelik a programot.

Egy rendszertől akkor várunk el helyes működést, ha a megfelelő adatokkal, a megfelelő körülmények között tud dolgozni. Sajnos nem élhetünk azzal a feltételezéssel, hogy a felhasználók mindig biztosítani tudják a helyes adatokat és a megfelelő körülményeket. Éppen ezért a programokat lehetőségeinkhez mérten fel kell készítenünk a nem megfelelő körülmények közötti működésre is, illetve a hibák legszélesebb körű kezelésére.

A felhasználói interakciókból, illetve a helytelen adatbevitelből származó hibákat kétféle módon kezelhetjük. Az egyik módszer, hogy eleve el sem fogadjuk a helytelen adatokat (ellenőrzött adatbevitel). Ehhez a fejlett, magas szintű programozási nyelvek és fejlesztői környezetek többféle támogatást is adnak, például a beviteli mezőkhöz beviteli maszkokat definiálhatunk, és csak olyan adatokat fogadunk el, amelyekre illeszkedik a maszk (hová például egy négyjegyű egész számot, pl. irányítószámot várunk, oda nem lehet szöveges adatot beírni).

Sajnos nem minden adatbevitel maszkolható. A bonyolultabb kritériumok maszkkal nem határozhatók meg, ilyenkor az lehet a megoldás, hogy az adatbevitelt egy végfeltételes ciklusba írjuk. A végfeltételes ciklusok jellemzője, hogy a ciklus belsejébe írott utasítások, a ciklus magja egyszer biztosan lefut, és ha kell – ha a ciklus végén elhelyezett feltétel azt előírja –, akkor ismétlés történhet. Előfeltételes ciklusok esetében a ciklus feltétele a ciklusfejben van, így ha az a ciklusba lépéskor eleve hamis, a ciklusmag egyszer sem fut le, vagyis a ciklus üresciklus lesz.

Amikor így próbálunk a hibák ellen védekezni, a potenciális hibák kezelése esetenként több energiát emészt fel, mint maga megoldandó feladat kódolása. Az ilyen programozási stílust robusztus programozásnak hívják. A robusztus programok hibatűrőek, azonban mind a fejlesztés, mind pedig egy ilyen forráskód visszafejtése, megértése nehézkes lehet.

Az ezzel a szemlélettel szemben álló technikát kivételkezelésnek nevezzük. A kivétel – programozási szempontból – egy nem várt esemény. Kivételes esemény, amikor nem olyan típusú adat érkezik a billentyűzetről, amilyet vártunk, vagy megpróbálunk olvasni egy olyan hálózati meghajtóról, amelyhez előzetesen nem is csatlakoztunk.

A kivételkezelést megvalósító program fejlesztője abból a feltételezésből indul ki, hogy a program az esetek többségében helyes adatokkal és megfelelő környezetben fog futni. Nem tölti az idejét azzal, hogy megpróbálja kitalálni, milyen problémák léphetnek fel a futás ideje alatt, és csak a megoldandó problémára koncentrál: nem készít ellenőrzött adatbevitelt, nem ellenőrzi minden olvasás előtt a fájlok meglétét stb.

Azokat a programrészeket, amelyekben potenciális hibalehetőségektől tart, védett részekké teszi (z így védett programrészeket a legtöbb programnyelvben a `try` kulcsszó vezeti be). Amikor készít egy védett részt, abban leírja, hogy milyen hibák történhetnek az adott részben. Nem próbálja őket megakadályozni, vagy elkerülni, csak jelzi, hogy milyen hiba történhet. Amikor valóban be is következik egy ilyen hiba, akkor a futtató rendszer ezt a hibát mint *kivételes eseményt* eldobja. A programozónak gondoskodnia kell arról, hogy az eldobott kivételeket egy bizonyos programrész, a kivételkezelő elkapja. A kivételkezelő feladata, hogy megpróbáljon kezdeni valamit a kialakult helyzettel. Lehet, hogy a kivételkezelő nem tud semmit sem kezdeni ezzel a helyzettel, de még akkor is jobb dolog történt, mintha a program futása hibaüzenettel leállt volna.

10.2.5 Karbantartások előrejelzése

A rendszerek karbantartásával párhuzamosan a fejlesztők más rendszerek kifejlesztésén is dolgozhatnak, ezért fontos, hogy előzetes becsléseket tudjanak

mondani a várható karbantartási költségekről (humánerőforrás-igény, karbantartási munkaórák száma stb.). Mindig szem előtt kell tartani, hogy egy karbantartás valószínűleg további karbantartásokat fog generálni, hiszen egy működő rendszer megváltoztatása megváltoztatja (rontja) a rendszer struktúráját.

A karbantarthatóság megbecsülhető az alábbi mutatók figyelésével:

- Hibák javítására adott kérések száma: ha ez a szám növekszik, ahogy egyre több hiba kerül a programba, akkor? a karbantarthatóság mértéke csökken.
- A változtatás által érintett programrészek számának növekedéséből szintén arra következtethetünk, hogy a karbantarthatóság mértéke csökken, mivel egyre több programelem karbantartása válik szükségessé.
- Elintézetlen változtatási kérelmek számának növekedése: szintén arra utal, hogy a karbantarthatóság csökken.

Idővel szükségszerűen felmerül a rendszer újratervezésének igénye.

10.2.6 Rendszerek újratervezése

Egy olyan régi rendszer karbantartása, amelyet nem mi fejlesztettünk, és nem érhető el hozzá a dokumentáció, meglehetősen költséges folyamat. Ilyen esetekben célszerűnek látszik a rendszer újratervezése. Ennek hatására javulni fog a program struktúrája, és könnyebben megoldhatóvá válik a későbbi karbantartás. Az újratervezés során megtörténhet a hiányos vagy elveszett dokumentáció pótlása (újradokumentálás), a rendszer újraszervezése, vagy lefordítása egy modernebb programozási nyelvre, amely jobban támogatja az aktuális környezetet. Lényegében minden megváltozhat, de a rendszer funkcionalitása nem romolhat.

Az újratervezés magában hordozza a rendszer megbízhatóbbá tételét, ugyanakkor veszélyekkel is járhat, hiszen egy újra elkészített rendszertervben is lehetnek rejtett hibák. Egy működő rendszer újratervezése ugyanakkor mindig jelentősen olcsóbb, mint egy új szoftver kifejlesztése.

A legnagyobb különbség egy rendszer újratervezése és új rendszer kifejlesztése között abban mutatkozik meg, hogy az új rendszer fejlesztésekor mindig új specifikáció készül, míg egy régi rendszer újratervezésekor a régi specifikáció a kiindulási pont. Ezért nagyon fontos, hogy már a specifikációban felkészüljünk a szoftver evolúciójára, és a specifikáció akár az újratervezéshez is tartalmazzon útmutatást.

Az újratervezés hátrányai annak függvényében jelentkezhetnek, hogy milyen a jelenlegi rendszer. Amennyiben a jelenlegi rendszer nagyon régi, nagyon sok módosítást hajtottak végre rajta, és emiatt a struktúrája nagyon leromlott, az újratervezés után a karbantartás költsége ugyan csökkenhet, de a karbantartás hatékonysága elmaradhat egy újra kifejlesztett rendszer karbantartásának hatékonysága mögött.

Ebben az esetben az alábbi stratégiák mellett dönthet a megrendelő:

- A rendszer lecserélése: ez akkor válhat szükségessé, ha a régi rendszer nem tud az új környezetben (hardver, operációs rendszer) működni.
- A rendszer valamilyen szintű módosítása: a karbantarthatóság javításának érdekében lehet erre szükség, ha a jelenlegi rendszer karbantarthatósága már nem költséghatékony, de nincs egyéb ok a rendszer lecserélésére.
- A rendszer további fenntartása: ha a rendszer működése egyébként stabil, és a rendszerrel szembeni módosítási kérések száma nem kimagaslóan nagy, akkor érdemes a rendszer működését továbbra is fenntartani.
- A teljes rendszer leselejtezése: jellemzően akkor következik be, amikor a rendszer már annyira elavult, hogy semmilyen gazdasági érdek nem indokolja a további fenntartását. Például, ha egy vállalat egy SAP-megoldást vásárol, és ehhez pályázati forrásokból hardverberuházás is társul, a régi PC-n futó DOS-os raktárkészlet-nyilvántartást ki kell selejtezni.

10.2.7 Refaktorálás

Ez a rész némileg kívül esik a lecke tárgykörén, ugyanakkor mindegyik leckével kapcsolatban áll. A refaktorálás (refactoring) a forráskód átalakítását, optimalizálását jelenti, ezért elsősorban a szoftver implementációjakor kaphat hangsúlyos szerepet. Mivel azonban a szoftverevolúció során is szükségessé válhat a kód egyes részeinek újraimplementálása, illetve új funkciók beépítése, a kódoptimalizálás egyáltalán nem távoli kérdés a szoftvermódosítás szemszögéből sem.

Az objektumorientált tervezésről szóló leckében (5. lecke) vázlatosan leírtuk a magas szintű programozási nyelvek használata melletti programozás elvét. A programozó a választott programozási nyelven elkészíti a program forráskódját (a program szövegét), amelyet a fordítóprogram gépi kódra, esetleg egy másik nyelvre fordít le. Ott azonban nem esett szó arról, hogy már egy kis méretű projekt esetében sem egyetlen forráskód készül. A programot funkcio-

nális egységekre bontottuk a tervezési folyamatban, és az egyes egységeket különböző csoportok fejleszthetik, egymástól térben és időben is nagy távolságokra.

Egy programegység is felbontható kisebb részekre, amelyeket azután, vagy a fordítás során, vagy később integrálunk egységbe. A refaktorálás a forráskód(ok) manipulációját jelenti. A legtöbb magas szintű programozási nyelvhez kapcsolódó integrált fejlesztői környezet már nyújt refaktorálási szolgáltatásokat. Ezek közül a leggyakoribbak:

- Egy kódrészlet többször is szerepel a forráskódban,
- egy metódus túlságosan nagyra nőtt,
- túlságosan nagyméretű a ciklus magja, vagy túlságosan bonyolultan egymásba ágyazott ciklusok vannak a programban,
- egy osztályban szereplő kód valójában nem is az osztály viselkedését írja le (nem illik oda a kód)
- egy metódushívás paraméterlistájában túl sok a paraméter
- egy osztály túlságosan nagyra nőtt, többféle, jól elkülöníthető feladatot is ellát
- egy metódus csak adatot közvetít egy másik metódus számára, de önmaga lényegében nem csinál semmit
- semmitmondóak a változónevek a programban
- egy osztály belső adattagjai túlságosan nyilvánosak a külvilág számára
- egy származtatott osztály csak nagyon csekély mértékben csinálja azt, amit az az ősosztály csinál, amelyből származtattuk.

Az itt felsorolt anomáliák nem sorolhatók be egyértelműen a hibák kategóriájába, hiszen ha hibák lennének, akkor a fordítóprogram kiszűrné őket, vagy futáskor jelentkezne a hatásuk (kivételek generálódnának, vagy a program leállna futásidejű hibával). Ezek a problémák inkább a forráskód hatékonyságát rontják, ezért a későbbi karbantartást is befolyásolják. A refaktorálás lényege a fenti hibák kiszűrése és kiküszöbölése a forráskódból. Ne higgyük, hogy ezek megoldása egyszerű és költségmentes feladat! Ha egy program 200 sorból áll, és abban kell átnevezni egy változó vagy egy metódus nevét, azt a szövegszerkesztő keresés és csere funkciójával is meg lehet valósítani. Ha a kód 2 millió soros és 17 fordítási egységben van leírva, amelyek bármelyikében szerepelhet ennek a metódusnak a neve, akkor ez már nem annyira egyértelmű feladat, csupán szövegszerkesztési eszközök használatával.

A refaktoráló eszközök elemzik a forráskódot, és nem egyszerűen csak szövegszerkesztési műveleteket hajtanak végre. Gondoljunk például arra a feladat-

ra, ha egy ismétlődő utasítássorozatot szeretnénk kiemelni a kódból és függvénnyel helyettesíteni minden előfordulását. A műveletsorozat lehet azonos, de az aktuális paraméterlista változhat is. Kereséssel és cserével ne próbálkozzunk ennek a megoldásához!

A refaktorálás költséges és az eredmény nem garantált. Ugyanakkor meg kell jegyeznünk, hogy egy hatékonyan megtervezett szoftver kódolásakor nem születnek olyan kódok, amelyeket azonnal refaktorálni kell. A refaktorálás sokkal gyakrabban merül fel igényként akkor, ha egy működő rendszert utólag már sokszor módosítottak. Ha a refaktorálástól remélhető a kód olvashatóságának, illetve a program struktúrájának javulása, akkor érdemes élni a lehetőséggel.

10.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

10.3.1 Összefoglalás

A szoftverek fejlesztése nem ér véget a megrendelőnek történő leszállításkor. A karbantartás, az új funkciók integrálása, illetve a használat során felfedezett hibák javítása, valamint a megváltozott környezethez való igazítás mind a szoftverfejlesztéshez tartozó feladat.

Optimális esetben a rendszer karbantartását az a cég végzi, amely a fejlesztésért is felelős volt. Ha ez nem így van, akkor a fejlesztés első fázisában elkészített specifikáció döntő szerepet kap a karbantartás során. A problémák egyik forrása lehet éppen a specifikáció hiánya, vagy hiányosságai.

Egy rendszer karbantartása mindig ront a szoftver struktúráján, ezért idővel szükségessé válhat a rendszer újratervezése, újraimplementálása. Ilyenkor mérlegelni kell, hogy az újratervezés vagy az új fejlesztés mellett döntsön-e a megrendelő. Az újratervezés költségei alacsonyabbak, a várható eredmény, hatékonyság viszont minden esetben elmarad egy új fejlesztés eredményeihez képest.

10.3.2 Önellenőrző kérdések

1. Mi az a szoftverevolúció?
2. Milyen tényezők befolyásolják a szoftverkarbantartást?
3. Milyen műveleteket foglal magában a szoftverkarbantartás?
4. Milyen hibákkal szembesülhetünk szoftverkarbantartáskor?
5. Mi az a robusztus programozás?
6. Mi a kivételkezelés lényege?
7. Hogyan lehet előre jelezni a szükséges karbantartásokat?

8. Mikor válhat szükségessé a rendszerek újratervezése?
9. Mire való a refaktorálás?
10. Milyen esetekben hívhatók segítségül a refaktorálás eszközei?

11. LECKE: SZOFTVERTESZTELÉSI FOLYAMATOK, TECHNIKÁK, SZOFTVERMENEDZSMENT, SZOFTVERMINŐSÉG, KÖLTSÉGEK

11.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A szoftverfejlesztés – mint arról többször szóltunk már – nem csak a tervezést és az impementációt foglalja magában. Egy elkészült szoftvert csak akkor szállít le a megrendelőnek a fejlesztő, ha azt kellően tesztelték. A megrendelő minőségi munkát vár tőlünk, de mit is jelent pontosan az a kifejezés, hogy szoftverminőség, illetve mi jellemzi a minőségi szoftvert?

A fejlesztés kezdő fázisában a felek szerződést kötnek, amelyben megállapodnak a szoftver elkészítésének árában is. A fentieken túl jegyzetünk utolsó leckéjében megvizsgáljuk azt is, hogy mi befolyásolja a szoftverek költségeit.

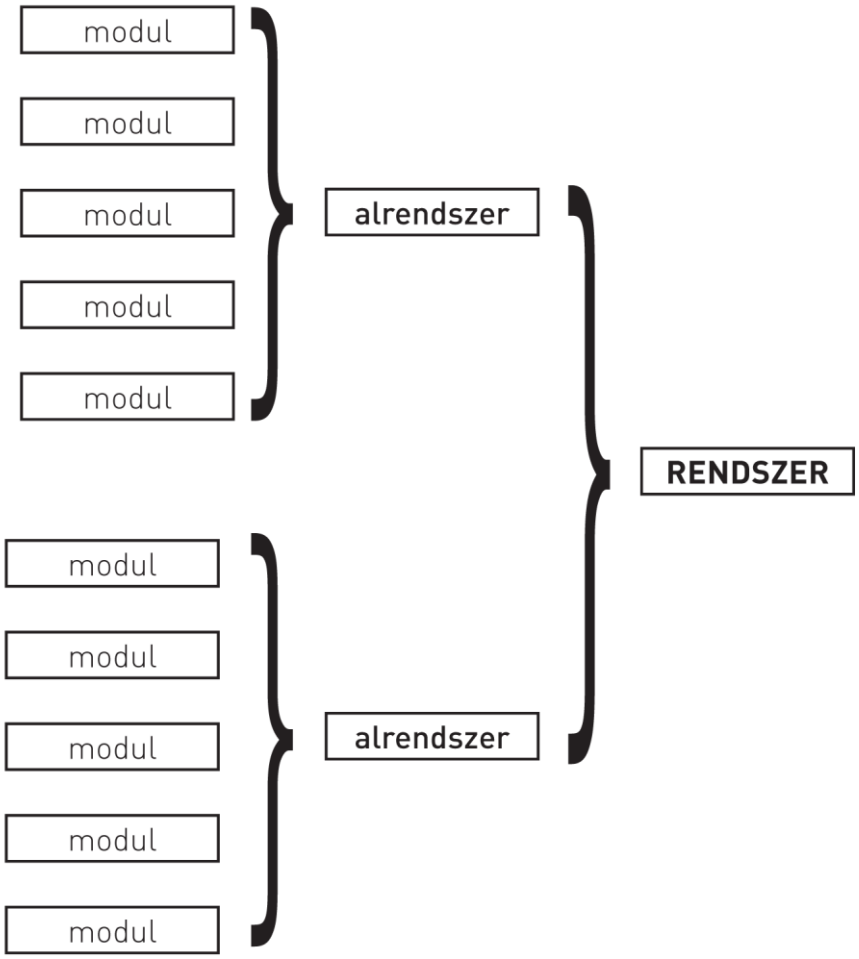
A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező képesség, figyelem-összpontosítás.

11.2 TANANYAG

11.2.1 Tesztelési folyamatok, technikák

Amikor elkészül a rendszer egy-egy eleme, azt a fejlesztők tesztelik: vajon a programrész a specifikációban leírtaknak megfelelően működik? Az inputokra a megfelelő outputokkal válaszol? Tud-e a felhasználó olyan tevékenységet végezni, amely nem várt eseményt, kivételt okoz? Kezeli a programrész az összes várható és váratlan kivételt?

A komponensekből összeállítják az alrendszereket, az alrendszereket integrálva pedig elkészül a rendszer. Hiába működik egy komponens megfelelően, előfordulhat, hogy bizonyos hibák csak más alrendszerekkel való együttműködéskor jelentkeznek, esetleg bizonyos komponensek nem is képesek másokkal való együttműködésre. Egy rendszer tesztelési folyamatában ezeket lépéseket mindig végig kell járni. A kész rendszer elkészültekor kezdődhet a teljes rendszer tesztelése, amelyet rendszerint már nem (csak) a fejlesztők végeznek, hanem bevonnak külső szakértőket is.



48. ábra: Rendszerek kialakítása

A szoftvertesztelésnek alapvetően két célja van. Az egyik, hogy bizonyítani kell mind a fejlesztő, mind a megrendelő számára, hogy a szoftver a specifikációban leírottaknak megfelelően működik, tehát alapvetően megfelel a vele szemben támasztott követelményeknek. Ez a validációs teszt feladata. A validációs teszt akkor eredményes, ha a teszt során bebizonyosodik, hogy a rendszer megfelel a tőle elvárt követelményeknek.

A másik cél, hogy feltárjunk minden olyan hibát, rendellenességet, amely a rendszer működését, vagy a működés biztonságát veszélyezteti. Ez a hiányos-

ságtesztelés feladata. A hiányosságtesztelés akkor eredményes, ha feltár egy olyan hibát, amely a rendszer helytelen működését okozza.

Egyik teszttől sem várjuk azt, hogy megmutassa, hogy a rendszerben nem maradtak feltáratlan hibák, vagyis hogy bebizonyosodjon az, hogy a rendszer minden körülmények között képes a helyes működésre. A teszteléstől legfeljebb elvárható eredmény az, hogy megmutassa mind a fejlesztőknek, mind pedig a megrendelőnek, hogy a rendszer elég jó ahhoz, hogy üzemszerűen lehessen használni.

49. ábra: A tesztelés általános modellje

A tesztelést tesztesetekkel végezzük. A teszteset az adott inputokra adott, elvárt outputok specifikációja, illetve annak a leírása, hogy a teszteset pontosan mit tesztel a rendszerben.

A tesztelés tehát az egyes programegységeken, illetve az azok integrálásával létrejött mind nagyobb és nagyobb egységeken is folyamatosan meg kell, hogy történjen. Az egyes programegységek fejlesztőinek munkája rendszerint magában foglalja a tesztelést is, amelyet természetesen dokumentálni kell (milyen teszteseteket vizsgáltak, milyen eredménnyel).

11.2.2 Komponensek tesztelése

A komponensteszt jellemzően hiányossági teszteket foglal magában, hiszen egy rendszer egyetlen – a rendszer méretéhez mérve viszonylag kicsi – elemétől nem várható, hogy a teljes rendszertől elvárt helyes működést produkálja. Tesztelhető komponens lehet egy függvény vagy egy objektum metódusa, egy objektumosztály vagy osztályok olyan halmaza, amelyben az egyes osztályokból létrehozott példányok az interfészeken keresztül kommunikálhatnak.

Ha egy függvényt kell tesztelnünk, akkor a híváskori aktuális paraméterlista tartalmazza a különböző tesztesetek adatait. A függvény működése viszonylag egyszerűen tesztelhető, hiszen ha egy adott inputra a megfelelő outputot adja, akkor a függvény az adott tesztesetet teljesítette.

Objektumok, osztályok tesztelésekor meg kell vizsgálnunk az osztályban leírt összes metódus működését. Vizsgálnunk kell az objektum belső állapotait, amelyet úgy tehetünk meg, hogy kipróbálunk minden olyan tesztesetet, amely az objektum belső állapotának megváltozásához vezethet. Osztályok tesztelésénél különös figyelmet igényel, ha az osztályhierarchia egy részét akarjuk tesztelni, vagyis a tesztelésbe bevonunk egy vagy több ősosztályt és egy vagy több származtatott osztályt is.

Egy komplex rendszerben az objektumok, osztályok egymás interfészein keresztül kommunikálnak. Osztályok tesztelésekor vizsgálni kell az interfészeik helyes működését is, illetve olyan eseteket is, amikor egy interfész nem a megfelelő adatokat kapja.

11.2.3 Rendszertesztelés

Amikor egy rendszert összeállítunk a kész komponensekből, az addigi hiányossági teszteléseket felváltja, illetve kiegészíti a validációs tesztek sora is. A rendszertesztek általában két fázisból állnak. Az első fázis az integrációs tesztelés, ahol a tesztelést végző szakemberek – akik lehetnek külső cég szakemberei is – a forráskód elemzésével tárják fel az egyes hibák hátterét és azonosítják azokat a komponenseket, amelyek érintettek az adott hiba létrejöttében vagy maguk is szenvedői a hibának. A tesztelés másik fázisában a felhasználók egy csoportja kapja meg a rendszer lefordított, futtatható változatát. Ez a tesztelési módszer hasonlít a fekete doboz módszerhez, ahol a felhasználó nem tudja, hogy hogyan működik a szoftver, csak azt tudja vizsgálni, hogy az adott inputokra a megfelelő outputokkal válaszol-e a program.

11.2.4 Teljesítménytesztelés

Ennek a tesztelési eljárásnak az a célja, hogy megmutassa, hogy a rendszer képes működni a tervezett terhelés mellett (konkurens felhasználók száma, adatmozgás mennyisége egységnyi idő alatt stb.). Ebben a tesztelési fázisban a teszteseteknek a majdani rendszer működésének határain kell működnie. A teszteseteknek az a feladata, hogy a rendszert szélsőséges körülmények közötti működésében vizsgálja, és az ilyenkor jelentkező hibákat is feltárja. Az ilyen tesztek akkor tekinthetők eredményesnek, ha olyan hibát tárnak fel, amely normális körülmények között nem jelentkezne.

11.2.5 Szoftverminőség

A minőség a követelményeknek való megfelelést jelenti. Ez szoftverekre vetítve és szűken értelmezve azt jelenti, hogy minőségi szoftver az, ami megfelel a specifikációban leírtaknak. Ez azonban nem minden esetben szerencsés megközelítés. A specifikációban nem lehet mindent pontosan, explicit módon leírni (például a karbantarthatóság kérdése előzetesen, egy rendszer specifikálásakor még nem feltétlenül látható tisztán), illetve egy specifikáció sohasem lehet teljes. Ezért hiába felel meg egy szoftver a specifikációjának, ha a felhasználóknak nem tetszik, vagy nem elégedettek vele, akkor nem fogják jó minőségűnek tartani.

A szoftverminőség megvalósítását leíró folyamat a minőségbiztosítás. A minőségbiztosítás során meghatározhatók a szoftvertermékre vonatkozó szabványok. A termékszabványok a fejlesztett szoftverre vonatkozó szabványok, amelyek leírják például a szoftver dokumentációjára vonatkozó szabályokat, vagy azt, hogy a fejlesztés során milyen programozási nyelv(ek)et kell használni.

A folyamatszabványok a szoftver fejlesztése alatt követendő folyamatokat határozzák meg.

11.2.6 Költségek

Költség szó alatt rendszerint pénzben kifejezett értéket értünk. Rendszerek esetében a költség nemcsak pénzben kifejezett értéket jelenthet, a költség szó ennél általánosabb jelentésű: időigény, bonyolultság, humánerőforrás-igény stb. A karbantartási költségek magasabbak, ha egy új funkcionalitást egy már működő rendszerhez adunk, mintha már az implementáció során felkészülünk a karbantartásra. Ennek okai a következőkben keresendők:

- A szoftvercégeknél is jelen van a fluktuáció, vagyis a munkahelyi elvándorlás. A fejlesztőcsapatok felbomlanak, a régi fejlesztők helyére újak érkeznek. Az új szakemberek nem ismerik a régi rendszert, ahhoz tehát,

hogy elvégezhessék a szükséges módosítást, először meg kell ismerniük, meg kell érteniük és meg kell tanulniuk a régi rendszert.

- Ha a karbantartásra nem ugyanazzal a céggel kötnek szerződést, amelyik a fejlesztést is végezte, akkor a fejlesztő cég nem érdekelt abban, hogy megkönnyítse a rendszer karbantartását. Ennek lehet hatása a dokumentáció teljességét illetően, vagy megnyilvánulhat abban, hogy egy általuk is felismert hibát inkább kikerülnek, még ha ezzel a majdani karbantartás költségeit meg is növelik.
- A karbantartás nem túl népszerű munka, így rendszerint a tapasztalatlanabb szakemberek kapják. Ha egy régi rendszer esetleg még fejletlenebb környezetben (programozási nyelven) is készült, akkor a tapasztalatokkal egyébként sem rendelkező szakembereknek először a programozási nyelvet, aztán a régi rendszert kell megtanulniuk, és csak ekkor kezdhetnek a karbantartáshoz.

50. ábra: Más programozó által írt kód karbantartása nagyon nehéz munka

- A programok kora is befolyásoló tényező. Ha egy régi rendszert sokszor kellett már módosítani, akkor a struktúrája bizonyosan romlott. Ennek következtében nehéz a program megértése egy olyan szakembernek, aki most találkozik vele először. A régi programozási nyelvek régi környezetekben való futáshoz voltak alkalmazhatók, lehet, hogy a jelenlegi környezet már nem támogatja az ilyen nyelveken készült alkalmazáso-

kat. Magyarországon pl. még mindig megdöbbentően sok DOS-os, pl. Clipperben készített alkalmazás fut, pedig az újabb és újabb Windows-verziók egyre kevésbé támogatják az ilyen szoftverek használatát.

Az előző fejezetben szó esett a rendszerek újratervezésének folyamatáról. Ennek költségeit a következő tényezők befolyásolják:

- A jelenlegi (vagyis az újratervezendő) rendszer minősége: minél gyengébb minőségű a jelenlegi rendszer, annál nagyobb költséggel jár az újratervezése.
- A szükséges adatkonverziók mennyisége: ha az újratervezés során arról döntünk, hogy a jelenlegi rendszerben tárolt adatok típusát, struktúráit is megváltoztatjuk, a régi adatokat konvertálni kell az új struktúrába. Ha a jelenlegi rendszer mögött egy adatbázis áll, amelyben rekordok milliárdjait tároljuk, az adatkonverzió költsége kiugróan magas is lehet.
- Szakemberek hozzáértése: ha a jelenlegi rendszer kezelőszemélyzetét nem lehet bevonni az újratervezésbe, akkor az újratervező team szakembereinek meg kell tanulnia a jelenlegi rendszert, amely szintén növeli a költségeket.

Ha a költségeket szigorúan az anyagiak oldaláról vizsgáljuk, akkor három tényezőt kell tekintenünk: a fejlesztők bérköltségét, az utazási és esetleges képzési költségeket, illetve a fejlesztéshez szükséges hardver és szoftver költségét.

Ezek közül fajlagosan a fejlesztők bérköltsége a legmagasabb, amelyhez hozzáadódik még néhány, járulékos tétel is: az irodahelyiségek rezsije (fűtés, világítás), a kisegítő személyek (adminisztráció, takarítás stb.) költségei, társadalombiztosítási költségek (biztosítások, egészségpénztár, nyugdíj stb.).

Egy szoftver árát ezek tükrében nem könnyű meghatározni. A legnagyobb gondosság mellett sem tervezhető meg egy szoftver fejlesztésének időtartama napra pontosan. Ha a fejlesztők nem munkára szerződnek, hanem alkalmazottak, akkor a napi bérükkel kell számolni, ezt viszont a megrendelőnek nem kell tudnia. Nem tervezhető meg az sem, hogy egyidejűleg hány projekten dolgoznak majd a fejlesztők, hiszen a piac telítettsége miatt egyetlen cég sem engedheti meg magának, hogy lemondjon egy projektről csak azért, mert egy másikon éppen most dolgoznak. A cégek profitorientáltak, de más-más stratégiát folytatnak a profitmaximalizálás során, ha új szereplői a piacnak, vagy ha már régi, komoly referenciákkal rendelkező vállalkozások. Egy új szereplő hajlandó kezdetben kisebb profitért dolgozni, így később nagyobb nyereséget könyvelhet el. A kezdeti, kisebb profitot eredményező termékeinek fejlesztése során szerzett tapasztalatok később megtérülnek.

Ha egy szervezet nem biztos abban, hogy mekkora összeget becsüljön a fejlesztés költségeinek, akkor valószínűleg némileg felül fogja becsülni az árat az előre nem látható tényezők miatt.

Nem létezik olyan módszer, amellyel pontosan megbecsülhető lenne az előre, hogy egy rendszer kifejlesztése mennyi időt fog igénybe venni. A rendszerek fejlesztésének korai szakaszában rendszerint csak becslésekre lehet alapozni a költségek kiszámítását. Igaz ugyanakkor az is, hogy mivel a szerződést a fejlesztés elején vagy annak korai szakaszában írják alá, a fejlesztést később úgy időzítik, hogy a ráfordított munkaórák száma végül igazolja a becslést.

A fejlesztésre szánt időt a cégek egyre rövidebbre szabják, hiszen a piaci folyamatok ezt várják el a fejlesztőktől. Ez egyfelől eredményezhetné a termékek költségeinek csökkenését is, de ahogyan arról korábban írtunk, a fejlesztésre (tervezés, specifikáció, implementáció) szánt idő csökkentésével valószínűsíthető, hogy nőni fog az evolúció során szükséges karbantartás időtartama, amely viszont növeli a költségeket.

11.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

11.3.1 Összefoglalás

Egy kész szoftvert akkor lehet leszállítani a megrendelőnek, ha az átesett a megfelelő tesztelésen. A tesztelés többféle szinten és többféle módszerrel történhet: tesztelhetők a komponensek, az alrendszerek és maga a kész rendszer is. A tesztek lehetnek hiányossági vagy validációs tesztek.

A szoftverek minőségét számos tényező befolyásolja. A minőség általános definíciója általánosságban igaz a szoftverrendszerek esetében is, de a szoftverek esetében speciális tényezők is meghatározhatják a minőséget.

A szoftverek fejlesztésének korai szakaszában a szerződő felek megállapodnak a szoftverfejlesztés költségeiben, amely ebben a szakaszban rendszerint még csak becsléseken alapulhat. A pontos becslés mindkét fél számára fontos, ugyanakkor nem minden esetben várható igazán pontos eredmény, hiszen a szoftver fejlesztésének időtartamát előzetesen csak hozzávetőlegesen lehet meghatározni..

11.3.2 Önellenőrző kérdések

1. Milyen programegységeket kell tesztelni a fejlesztés során?
2. Mi az a hiányosságteszt? Mikor nevezhető eredményesnek egy hiányosságteszt?

3. Mi az a validációs teszt? Mikor nevezhető eredményesnek?
4. Mi a feladata a teljesítménytesztnek?
5. Hogyan biztosítható a szoftverminőség?
6. Milyen tényezők befolyásolják a költségeket?

12. ÖSSZEFOGLALÁS

12.1 TARTALMI ÖSSZEFOGLALÁS

A szoftverfejlesztés nem csupán a programozók feladata. A programozók a kész rendszertervek alapján elkészítik a program futtatható változatát egy adott számítógépes környezetben: adott hardvereszközök mellett az adott operációs rendszer alá, a megrendelő munkája során használt egyéb alkalmazói szoftverek mellé.

A programozás, azaz a szoftver terveinek implementálása az adott környezetbe a szoftverfejlesztésnek csak egy részét képezi. A végeredmény létrejöttét illetően ez a leglátványosabb része, hiszen ennek során készül el a programnak az a változata, amelyet a felhasználó alkalmazni tud a munkája során. Az azonban egyáltalán nem biztos, hogy ez egyben a leghosszadalmasabb munka is a fejlesztés során. Egy jó szoftver készítésekor a programozást egy hosszadalmas tervezési folyamat előzi meg, és remélhetőleg a felhasználó már jóval azelőtt megismeri a majdani szoftver lehetőségeit, mielőtt az első tesztváltozatot kipróbálhatja.

A szoftverfejlesztés tehát nem a programozó, és főként nem egyetlen programozó feladata.

Ebben a jegyzetben bemutattuk a szoftverfejlesztés folyamatát, a szoftverfolyamatot, más szóval szoftvertechnológiát.

A jegyzetben az alábbi témaköröket tekintettük át:

1. A szoftverkészítés folyamata, a szoftverekkel szembeni követelmények
3. Rendszermodellek a szoftverfejlesztésben
4. Szoftvertervezés
5. Objektumorientált tervezés
6. Felhasználói felületek tervezése
7. Gyors szoftverfejlesztés
8. Szoftver-újrafelhasználás
9. Komponensalapú szoftverfejlesztés
10. Szoftverevolúció, szoftvermódosítási lehetőségek, refaktorálás
11. Szoftvertesztelési folyamatok, technikák, szoftvermenedzsment, szoftverminőség, költségek

12.2 ZÁRÁS

Reméljük, hogy a jegyzet hasznos ismereteket adott minden olyan olvasónak, aki jelenlegi vagy későbbi munkája során közelebbi kapcsolatba kerül a szoftverfejlesztéssel.