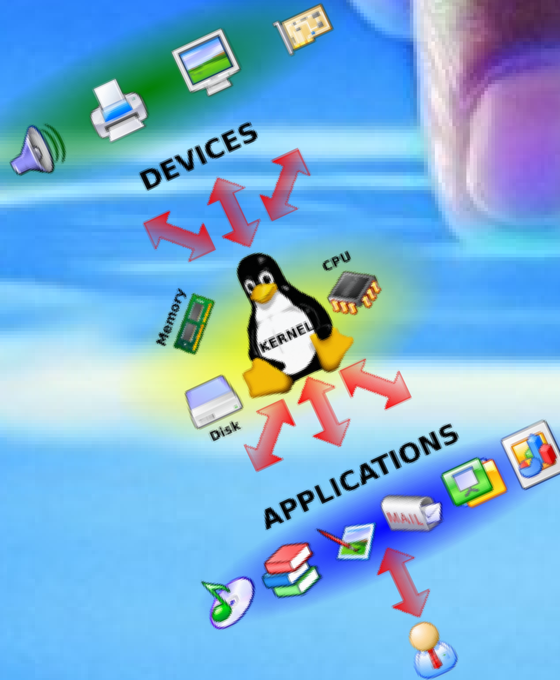
2010. január

Informatikai Navigátor

Gondolatok a szoftverek használatáról és fejlesztéséről

CreedSoft

2. szám (1)



There's a difference between knowing the path and walking the path - Morpheus

Tartalomjegyzék

1. GRETL – Számítógépes közgazdasági adatelemzés 3
1.1. Az adatok megszerzése 3
1.2. A korreláció 4
1.3. Egyváltozós regresszió 5
1.4. Többváltozós regresszió 6
1.5. Összefoglalás 7

2. Az eseményvezérelt programozás - Az Observer minta 9
2.1. Alapfogalmak és a hagyományos elnevezési konvenciók 9
2.2. Egy eseményvezérelt komponens elkészítése Java környezetben 10
2.3. Egy eseményvezérelt komponens elkészítése C# környezetben 18
2.4. SWT 22
2.5. Google Web Toolkit (GWT) 23
2.6. A .NET Form 26
2.7. Qt 27
2.8. Gtk 29
2.9. A Java beépített observer minta támogatása 30

3. SAP függvény készítése JAVA környezetben 33
3.1. Az RFC function (BAPI) felületének megtervezése és létrehozása 33
3.2. A JAVA szerver megtervezése és létrehozása 34
3.3. A JAVA szolgáltatás meghívásának tesztelése 40

4. Informatikai szilánkok - tippek és trükkök 43
4.1. LDAP Query parancssorból 43
4.2. File elmentése böngészőből HTTP-én keresztül 44
4.3. JavaScript class 44
4.4. Struktogramok készítése LaTeXsegítségével 45

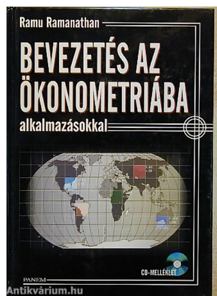
5. Elemzői sarok 48
5.1. A clearing fogalma 48
5.2. A Microsoft Office SharePoint Server 2007 50

Főszerkesztő: Nyiri Imre (imre.nyiri@gmail.com)

1. GRETL – Számítógépes közgazdasági adatelemzés

Ebben a cikkben GRETL (Gnu Regression, Econometrics and Time-series Library) program lehetőségeit szeretnénk bemutatni, ami egy elterjedten használt, szabad és nyílt forráskódú ökonometriai, azaz közgazdasági adatelemzést támogató programcsomag.
Webhelye: `http://gretl.sourceforge.net/`

A szoftver annak is köszönheti a jó hírét, hogy a világhírű, magyar nyelvre is lefordított „*Bevezetés az ökonometriába alkalmazásokkal*" című egyetemi tankönyv is ezt használja a példák szemléltetésére, megoldására (1. ábra).



1. ábra. Ökonometria tankönyv

Az ökonometria[1] a társadalomban és gazdaságban (bár módszerei a természettudományokban is alkalmazhatóak) mért adatok elemzésével foglalkozik. A megfigyelt adatfajtákat 3 csoportba osztja ez a tudomány:

- idősoros adatok → a mért adatok idő szerint rendezhetőek (példa: évenkénti GDP egy országban 2005-től 2009-ig)
- keresztmetszeti adatok → például a dolgozók fizetésének listája. Itt az idő nem játszik szerepet, illetve minden mért adat egyetlen időszakba esése kötelező elvárás a felméréssel szemben.
- panel adatok → egyesíti az idősoros és keresztmetszeti adatok sajátosságait.

Ezen cikknek nem lehet feladata, hogy egy komplett ökonometria tankönyv legyen, azaz a különféle statisztikai, elemzési módszereket (indexszámok, leíró statisztikák, korreláció, regresszió, idősorok elemzése, stb.) mutasson be, ellenben a *GRETL* program használatához szükséges alapvető ismereteket mégis elmagyarázza úgy, hogy azt még a teljesen kezdő is alkalmazni tudja majd a munkájához. A példákhoz Gary Koop: Közgazdasági adatok elemzése című könyvéhez csatolt adathalmazt használjuk, miközben – a talán 2 legfontosabb eszköz – a korreláció és regresszió számítását és értelmezését mutatjuk be.

1.1. Az adatok megszerzése

A *GRETL* képes nagyon sok forrásból importálni (File→Open Data→Import menü) az elemzendő adathalmazokat, ezeket az analízis során együtt is tudja használni, mert minden megszerzett adatot vagy adatsort a belső adatmemóriájában tárol. Természetesen ismeri az excel file-ok formátumát is. Amennyiben egy excel munkalap egyik oszlopa egy számsort tartalmaz oly módon, hogy az első sorban az oszlop neve (például: fizetés) van, akkor az importálás eredménye egy olyan *GRETL* által is látott adatsor, aminek a

[1] Az ökonometria a közgazdaságtan – azon belül is a matematikai közgazdaságtan – önálló tudománnyá fejlődött részterülete, amelynek célja a gazdasági jelenségek matematikai jellegű elemzése, továbbá a közgazdasági elméletek és modellek tapasztalati adatok alapján történő igazolása, illetve megcáfolása. Eszközeit elsősorban a matematika, azon belül is főként a valószínűség-számítás, továbbá a statisztika eszköztárából meríti.

változó neve: fizetés. Egyszerre több adatoszlopot is importálhatunk.

Az ismertetett példákban olyan adatoszlopokat (változókat) fogunk használni, ami a következő adatsorokat tartalmazza majd: *lotsize* (a telek alapterülete), *saleprice* (a ház ára), *bedroom* (a hálószobák száma), *bath* (a fürdőszobák száma), stb... (azaz az xls lap első sorának oszlopnevei ezek lesznek majd).

1.2. A korreláció

A korreláció megértéséhez és használatához használjuk fel azt a felmérést, amit valamely kanadai városrészen végeztek. Rendelkezésünkre áll 546 darab mérés arra vonatkozóan, hogy mekkora a telekméret (négyzetlábban) és a rajta lévő ház ára ($-ban). Ezt egy 2 oszlopos excel táblázat tartalmazza, aminek első 2 sorának oszlopnevei: *lotsize* és *saleprice*. Az 1.1 pontban leírtak szerint importáljuk be ezt az xls táblázatot a GRETL saját tárolójába, így létrejön 2 belső változó: *lotsize* és *saleprice*. Amikor csak általánosságban beszélünk az adatsorok, mint számoszlopok neveiről, akkor az X, Y, Z, ... (változó)neveket szoktuk használni. Amikor az X adatsor i. elemére gondolunk, akkor ezt X_i-vel jelöljük. A korrelációs elemzés azt vizsgálja, hogy 2 együtt mért adathalmaz között van-e valamilyen kapcsolat, összefüggés, illetve ha igen, akkor az milyen erős és milyen irányú (pozitív vagy negatív). A statisztikai elméleti részletek megértése nélkül is kiszámítható az X és Y adathalmazok közötti korreláció, amit r-rel jelölünk és indexben szokás feltüntetni annak a két változónak a nevét, amire számítjuk:

$$r_{x,y} = \frac{\sum_{i=1}^{N}(Y_i - \bar{Y})(X_i - \bar{X})}{\sqrt{\sum_{i=1}^{N}(Y_i - \bar{Y})^2}\sqrt{\sum_{i=1}^{N}(X_i - \bar{X})^2}}$$

Esetünkben az $r_{lotsize,\, saleprice}$ értéket keressük, azaz arra vagyunk kíváncsiak, hogy a telekméret és a ház ára között milyen erős összefüggés van. A képletben $\bar{X}$ és $\bar{Y}$ az X, Y változók számtani átlagát jelölik. Mielőtt a *GRETL* kiszámolja nekünk ezt az értéket, gyorsan nézzük át az r korreláció tulajdonságait, hogy az eredményt értelmezni tudjuk majd!

1. *Az r* értékei ilyenek lehetnek: $-1 \leqq r \leqq 1$
2. Az $r_{X,Y} = 0$ azt jelenti, hogy X és Y értékei nem korrelálnak egymással, közöttük semmi összefüggés sincs. Az $r > 0$ pozitív, az $r < 0$ pedig negatív kapcsolatra utal az X és Y között. Az r abszolútértékének nagysága pedig ezen kapcsolat erősségére utal. Ennek megfelelően az $r = 1$ egy pozitív és függvényszerű kapcsolat, az $r = -1$ pedig a legerősebb negatív kapcsolat.
3. Egy trivialitás: $r_{X,Y} = r_{Y,X}$
4. Az $r_{X,X} = 1$, azaz bármely adathalmaz saját magával a legerősebben pozitívan összefügg, ami szintén triviális.

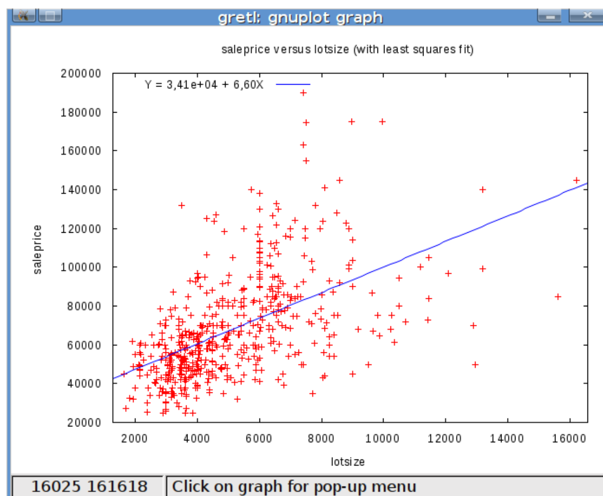
Ennyi bevezető után számítsuk ki a $r_{lotsize,\, saleprice}$ értéket! Az adatokat már beimportáltuk, nézzük meg a telekméret és a házár kapcsolatát. Ezt a feladatot egy *GRETL* konzol nyitással, és a *corr()* függvény hívásával oldhatjuk meg a legkönnyebben. A 2. ábra mutatja a *corr(saleprice, lotsize)* eredményét, ami ~+0,54. Elmondhatjuk tehát, hogy a teleknagyság és a ház eladási ára között határozott pozitív kapcsolat van, azaz a nagyobb telken lévő házak általában többet is érnek.

```
gretl console
gretl console: type 'help' for a list of commands
? corr(saleprice,lotsize)

corr(saleprice, lotsize) = 0,53579567
Under the null hypothesis of no correlation:
 t(544) = 14,8005, with two-tailed p-value 0,0000

?
```

2. ábra. A korreláció kiszámítása

1.3. Egyváltozós regresszió

A sokaságok közötti kapcsolatot vizsgáló korreláció fogalmánál is fontosabb, a közgazdászok által használt egyik legalapvetőbb matematikai eszköz a *regresszió* számítás. Ezen belül is a gyakorlatban a lineáris regresszió a leginkább használatos, azaz 2 együtt mért adathalmazt egy egyenessel próbálunk reprezentálni [2]. Ehhez nem kell sokkal többet tudnunk a matematikából, mint az egyenes egyenlete: $Y = \alpha + \beta X$. A GRETL még mindig tartalmazza a betöltött *lotsize* és *saleprice* adatsort, ezért most a „View→Graph Specified vars" menüpont használatával generáljunk le egy diagramot, ahol ezek a pontpárok (546 darab) vannak feltüntetve (3. ábra). A *GRETL* behúzza a regressziós egyenest is és feltünteti a képletét: $Y = 34100 + 6,6X$. Pontosabban a mi változóinkkal felírva: $saleprice = 34100 + 6,6 \cdot lotsize$. Az egyenes α és β értékei itt a beállításnak megfelelően kerekítve vannak, azonban a pontos számokat és a regresszió egyéb fontos adatait a „Modell→OLS" (OLS=legkisebb négyzetek módszere) menüpont futtatásával a 4. ábra mutatja, azaz a nagyon pontos egyenes egyenlet: $saleprice = 34136,2 + 6,59877 \cdot lotsize$.



3. ábra. A ponthalmaz grafikonja

A 3. ábra kinézete jól konfigurálható, feliratai, színei és betűtípusai megváltoztathatóak, azonban ezzel most nem foglalkozunk.

```
gretl: model 1
File Edit Tests Save Graphs Analysis LaTeX

Model 1: OLS estimates using the 546 observations 1-546
Dependent variable: saleprice

              coefficient   std. error    t-ratio   p-value
  -----------------------------------------------------------
  const       34136,2       2491,06        13,70    6,28e-37 ***
  lotsize         6,59877      0,445847    14,80    6,77e-42 ***

Mean dependent var    68121,60   S.D. dependent var   26702,67
Sum squared resid     2,77e+11   S.E. of regression   22567,05
R-squared             0,287077   Adjusted R-squared   0,285766
F(1, 544)             219,0558   P-value(F)           6,77e-42
Log-likelihood       -6246,977   Akaike criterion     12497,95
Schwarz criterion     12506,56   Hannan-Quinn         12501,32
```

4. ábra. Pontos regresszió számítás

A *lotsize* itt most a független (közgazdasági szakszóval: magyarázó), míg a *saleprice* pedig a független változó. Ez szavakkal elmondva azt is jelenti, hogy a telek méretének egységnyi növekedése a ház árának ~6,6$-os növekedését jelenti, azaz ennyit magyaráz meg a ház árának kialakulásakor. Ez matematikai értelemben az egyenes meredeksége. A statisztikai részletekben való elmélyedés helyett csak megjegyezzük, hogy ez a *6,59877*-es érték csak egy pontbecslés, igazából pontosabbak vagyunk, ha azt mondjuk, hogy tudunk egy intervallumot mondani (neve: *konfidencia*-intervallum), amibe valamilyen valószínűséggel beleesik a keresett $\hat{\beta}$, ami a becslése a regressziós egyenes meredekségének. A 4. ábra ablakában lévő „Analysis→Confidence intervals for coefficients" menüpont kiválasztásával kapunk egy táblázatot:

Változó	coefficient	Alsó	Felső
const	34136,2	29242,9	39029,5
lotsize	6,59877	5,72298	7,47456

Az utolsó 2 oszlop az egyenes konstansának (Az egyenes egyenleténekαértéke) és a *lotsize* együtthatónak a megbízhatósági intervalluma 95%-os valószínűségen. Ez azt jelenti, hogy

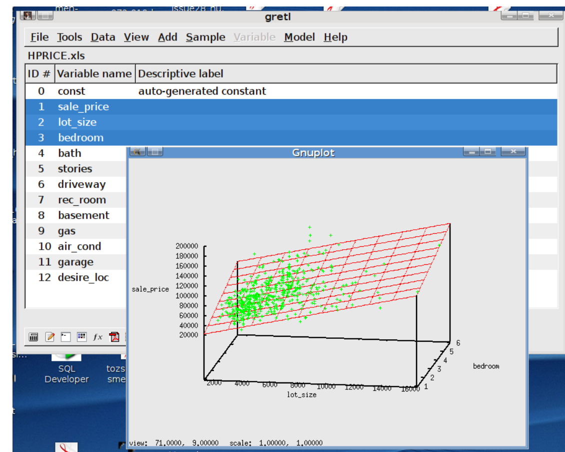
[2] A nem lineáris kapcsolat is gyakran visszavezethető lineárisra.

a *lotsize* együtthatója 95%-os biztonsággal esik a [5,72298; 7,47456] intervallumba, amibe a számított 6,59877 elég jól „beleszorul”, azaz ez az egyenes meredekség becslés nagyon jónak tekinthető. Az ökonometria (és ezzel együtt a *GRETL*) még sok dolgot ki tudna számítani, amit ritkábban használunk, de egy fontos dolgot még érdemes megtanulni. Ez a $\beta = 0$ hipotézis, azaz annak a vizsgálata, hogy X-től egyáltalán függ-e az Y, azaz példánkban a telekméretet érdemes-e egyáltalán vizsgálni a házárak elemzésénél (ez majd a többváltozós regressziónál is fontos lesz, ugyanis nem célszerű jelentéktelen hatású tényezővel megterhelni a vizsgálandó rendszerünket). Ezt a feladatot már megoldottuk, hiszen a konfidencia-intervallum nem tartalmazza a 0 értéket. Amennyiben nem számítunk regressziót, úgy a 4. ábráról leolvasható P érték is választ ad erre a kérdésre. Van egy ökölszabály: amennyiben N nagy és $P \leqq 0,05$, úgy elvethetjük a $\beta = 0$ hipotézist, azaz a telek méretétől is függenie kell nagy valószínűséggel az árnak. Ez az érték a 4. ábráról leolvasva most $6,77 \cdot 10^{-42}$, azaz radikálisan kisebb szám, mint *0,05*, így a $\beta \neq 0$ bizonyítottan nagy valószínűséggel igaz.

1.4. Többváltozós regresszió

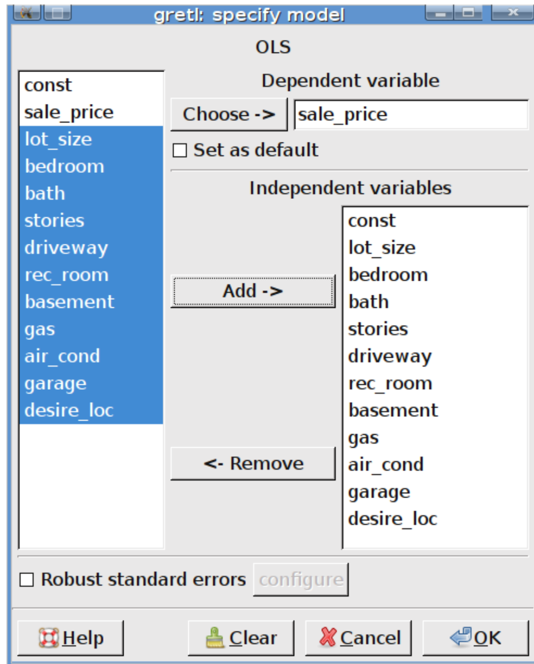
A valóság természetesen sokkal összetettebb, mint az a feltételezés, hogy a ház ára csak a telekmérettől függene. Általában tudományos vizsgálat tárgya azt eldönteni, hogy mely tényezők játszanak szerepet és az mekkora mértékben valamely jelenség (például a ház ára) alakításában. Amennyiben egy nem jelentős tényezőt vonunk be a vizsgálódásba, úgy modellünk túlságosan elbonyolódik, míg egy-egy jelentős hatás kihagyása rontja a modell erejét. Többváltozós regresszió esetén több független (azaz a jelenséget magyarázó) változónk van és egy többdimenziós illeszkedő egyenest igyekszünk meghatározni: $Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \ldots \beta_k X_k$. Itt most egy k változós egyenes képletét írtuk fel.

Az 5. ábra 2 dolgot mutat. Egyrészt a bővített felmérés szempontjait, azaz a telekmérettel (*lot_size*) együtt 11 magyarázó változó bevonásával próbáljuk elemezni a házárak alakulását, illetve az azokra gyakorolt hatások mértékét. Az érezhető, hogy ezek a szempontok nem egyforma erősen hatnak a házak árára. A másik érdekesség egy 3 dimenziós pontgrafikon, ahol a telekméret (*lot_size*) és hálószobák száma (*bedroom*) szerinti házárakat (*sale_price*) tüntettük fel. Persze ilyen 3 dimenziós ábrával csak max. 2 független változó esetén tudunk elemezni. A *GRETL* képes ezt a „Gnuplot”-os ábrát dinamikusan forgatni, különféle nézetekből időnként érdekes benyomásokat is szerezhetünk, de első ránézésre is látszik, hogy a hálószobák száma is növelő hatással van az árra.



5. ábra. A házárak 2 változós vizsgálata

Most elemezzük általánosságban is a 11 magyarázó változónk hatását! Jó hír, hogy a módszer ugyanaz, mint egyváltozós esetben, azaz válasszuk ki a „Modell→OLS” menüpontot és bejön a már ismerős dialógusablak, amit a 6. ábra mutat. A függő (dependent) változó természetesen továbbra is a ház ára (sale_price), a 11 többi változó pedig a magyarázó (independent) változónak lesz kiválasztva, majd nyomjuk meg az OK gombot!



6. ábra. Dialógus

A *GRETL* kiszámolja és megjeleníti (7. ábra) mindegyik változó együtthatóját (*coefficient* oszlop), ez alapján fel tudjuk íri a keresett 11 változós lineáris regressziós egyenest, amiből egy részlet: $sale_price = -4038.35 + 3,5430 \cdot lot_size + 1832 \cdot bedroom + \ldots + 9369,51 \cdot desire_loc$

gretl: model 1

File Edit Tests Save Graphs Analysis LaTeX

Model 1: OLS estimates using the 546 observations 1-546
Dependent variable: sale_price

	coefficient	std. error	t-ratio	p-value	
const	−4038,35	3409,47	−1,184	0,2368	
lot_size	3,54630	0,350300	10,12	3,73e-22	***
bedroom	1832,00	1047,00	1,750	0,0807	*
bath	14335,6	1489,92	9,622	2,57e-20	***
stories	6556,95	925,290	7,086	4,37e-12	***
driveway	6687,78	2045,25	3,270	0,0011	***
rec_room	4511,28	1899,96	2,374	0,0179	**
basement	5452,39	1588,02	3,433	0,0006	***
gas	12831,4	3217,60	3,988	7,60e-05	***
air_cond	12632,9	1555,02	8,124	3,15e-15	***
garage	4244,83	840,544	5,050	6,07e-07	***
desire_loc	9369,51	1669,09	5,614	3,19e-08	***

Mean dependent var	68121,60	S.D. dependent var	26702,67
Sum squared resid	1,27e+11	S.E. of regression	15423,19
R-squared	0,673124	Adjusted R-squared	0,666390
F(11, 534)	99,96774	P-value(F)	6,2e-122
Log-likelihood	−6034,094	Akaike criterion	12092,19
Schwarz criterion	12143,82	Hannan-Quinn	12112,37

7. ábra. A többváltozós egyenes együtthatói

A jobb oldalon lévő *-ok azt jelzik, hogy az adott változóra mennyire jelenthető ki, hogy 95%-os valószínűségi szinten magyarázza a házárat. A „***” jelenti azt, hogy biztosan magyarázza, a „*”, hogy lehet. Ezt mutatja számokkal a 8. ábrán látható, egyes változók együtthatóira számított konfidencia-intervallumok (megbízhatósági intervallumok) is. Például az 1 „*”-os bedroom-hoz tartozó intervallumba a 0 is beleesik, így a 0-hipotézist sem vethetjük. A többi változó megbízhatósági intervalluma nem tartalmazza a 0-át, így ezek az áralakulás szempontjából határozottan magyarázó erővel bírnak. A szöveges elemzésben tehát ilyen mondatokat írhatunk: a garázs határozottan alakítja a ház árának alakulását, azaz amennyiben egy ház rendelkezik ilyennel, úgy ˜4244 $-ral növeli az árat. Ilyen biztos hatása a hálószobák számának nincs, akár ki is hagyhatnánk a további vizsgálódásból, megtartva a maradék 10 darab, erős hatású tényezőt.

gretl: coefficient confidence intervals

t(534, 0,025) = 1,964

VARIABLE	COEFFICIENT	95% CONFIDENCE INTERVAL	
const	-4038,35	-10736,0	2659,27
lot_size	3,54630	2,85817	4,23444
bedroom	1832,00	-224,741	3888,75
bath	14335,6	11408,7	17262,4
stories	6556,95	4739,29	8374,60
driveway	6687,78	2670,06	10705,5
rec_room	4511,28	778,976	8243,59
basement	5452,39	2332,85	8571,93
gas	12831,4	6510,71	19152,1
air_cond	12632,9	9578,18	15687,6
garage	4244,83	2593,65	5896,01
desire_loc	9369,51	6090,72	12648,3

8. ábra. Konfidencia intervallumok

1.5. Összefoglalás

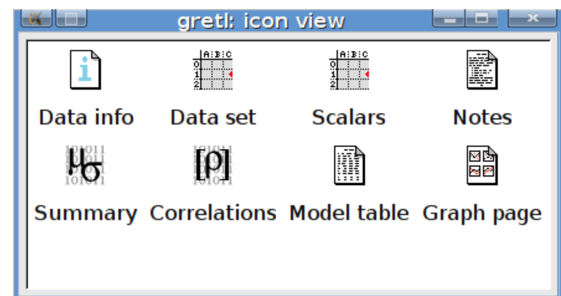
Az előzőekben gyakorlatiasan és tömören igyekeztünk bemutatni a *GRETL* regressziós lehetőségeit, azonban ez a szoftver ennél lényegesen többet tud. Lehetővé teszi (File→Databases almenü) az Interneten elérhető online adatbázisok közvetlen használatát. Az eddigi példákban

mindig csak 1 excel file-ból importáltuk az adatokat, de lehetőség van a különféle forrásokból származó adatok összefésülésére is. Eközben – amennyiben szeretnénk – az importált adatokat aggregálhatjuk is, azaz ha csak negyedéves adatok vannak a forrásban, de mi éveset szeretnénk használni, ezt ilyenre összevontan is importálhatjuk.

Nagyon lényeges, hogy az elemzésnél nem csak az importálás során keletkező változókat használhatjuk, hanem mi magunk is készíthetünk újat. Ez a lineáris regresszió pontosabb használata érdekében is elengedhetetlen. Miért? Képzeljük el, hogy van egy jelenségünk, aminek a mért pontpárjait (X független és Y függő változók) gyorsan ábrázoljuk egy grafikonon és „látjuk", hogy az valami $Y = 3X^2$ vagy $Y = \log_n X$ kinézetű, erre pedig nyilvánvalóan pontatlan lenne regressziós egyenest illeszteni. Itt egy ötlet segíthet! Tekintsük ehelyett például az Y, Z közötti regressziót (Y és $Z = X^2$), azaz Y és X^2 viszonyát vizsgáljuk. A *GRETL*-ben a „Define new variable" menüpont segítségével így írhatjuk ezt be a megjelenő soreditorba: $Z = X\hat{}2$, majd a Z és Y közötti regressziót kell futtatnunk. Ezzel a trükkel szinte minden gyakorlati vizsgálat visszavezethető a lineáris egyenes esetére.

Vannak jelenségek, amik binárisak. Példánkban például a ház légkondicionált állítás igaz vagy hamis. Amennyiben a 0 és 1 értéket rendeljük ezekhez, úgy máris bevonhatjuk a vizsgálódásunkba ezt a magyarázó változót is (azaz a légkondicionálás milyen hatással van a ház árára).

A *GRETL* természetesen tudja az egyszerű leíró statisztikákat is, a fontosabbak: mean (számtani átlag), medián, szélsőértékek, standard deviation (szórás), relatív szórás (CV=Coefficient of Variance), ferdeség (skewness), lapultság (Ex.Kurtosis). A *GRETL* ikon nézetében (9. ábra) az összes változó leíró statisztikája lekérdezhető.



9. ábra. Ikonnézet

A *GRETL* rendelkezik egy parancskonzollal, ahonnan minden funkció elérhető. A grafikonokat a *GNU Plot* program használatával jeleníti meg.

A regresszió analízisnél a leggyakrabban alkalmazott becslést, az *OLS*-t (legkisebb négyzetek módszere) használtuk, azonban a program sok más lineáris és nem lineáris módszert (a model menüpontnál ezek megtekinthetőek, itt nem soroljuk fel) is ismer. Tekintve, hogy a *GRETL* nyílt szoftver, amit sokan és intenzíven használnak, valószínűsíthető, hogy a jövőben is gyorsan része lesz minden olyan új matematikai és közgazdasági eredmény, ami jelenleg csak kutatási fázisban van.

2. Az eseményvezérelt programozás - Az Observer minta

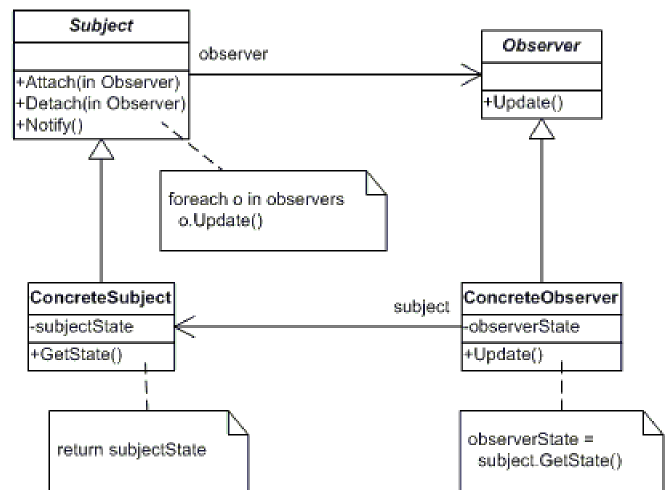
Az összetettebb programozási feladatok gyakran igénylik az aszinkron, párhuzamos működési módot. Ezen nem szabad csodálkozni, hiszen a való életben is így van. Egy ember csak 1 szálon (thread-en) végzi a dolgait, hiszen nem tud több példányban előállni, mint Mr. Smith ügynök a Mátrix c. film 2. részében.

Arra azonban mindenkinek van lehetősége, hogy aszinkron módon több feladattal foglalkozzon, bár egyszerre mindig csak eggyel. Példaképpen gondoljunk arra, hogy olvasunk, azaz ez a fő tevékenységünk, mindeközben fő a tea. Egyszercsak a teafőző elkezd sípolni (egy SIGNAL-t vagy EVENT-et bocsát ki), amire abbahagyjuk az olvasást, elzárjuk a teafőzőt, majd folytatjuk az olvasást. Az egész forgatókönyv azt tételezi fel, hogy az olvasó ember rendelkezik egy olyan képességgel (CALLBACK function), ami még olvasás közben is észleli a teafőző sípolását (INTERRUPT) és rendelkezik azzal a tudással, hogy megtegye olyankor a fentebb már említett teendőket (interrupt handler-re van, ami maga a callback function). Ez a fajta lehetőség összességében a feladatok sokkal hatékonyabb elvégzését eredményezi, hiszen enélkül csak teafőzés előtt és után olvashatnánk, közben nem. Helyette folyamatosan a teafőzést kéne figyelni (POOL-ozni).

A fentiek szellemében készített programokat eseményvezérelt programoknak nevezzük és elkészítésük főleg az *observer* (megfigyelő) tervezési mintára épül. Ezen módszer használatát bemutatjuk néhány ismertebb fejlesztői környezetre.

2.1. Alapfogalmak és a hagyományos elnevezési konvenciók

Az eseményvezérelt programozás a megfigyelő–megfigyelt tervezési mintát követi, aminek osztálydiagramját az 10. ábra mutatja.



10. ábra. Observer design pattern

Mit tanít nekünk ez az UML ábra? Először próbáljuk megérteni a konkrét „teafőzős” példánkon keresztül!

Kell egy *Subject* (gyakran használt rokonértelmű kifejezései a programokban: *Alany*, *Observable*, *EventSource*, *Component*, *Widget*, *Control*, *Gadget*), azaz egy olyan objektum, ami „sípolni képes”. Az ilyen tudással bíró dolgoknak olyan interface-szel kell rendelkezniük, ami arra tanítja meg őket, hogy kik felé kell „sípolni”, azaz az *Attach()* metódusával bejegyezteti magának ezeket (és a *Detach()* metódussal pedig törli, ha valaki le akar iratkozni a sípolásról). De kinek is sípol a teafőző? A megfigyelőknek, akik az említett *Attach()* metódussal tudnak erre feliratkozni. Az ilyen megfigyelő tudással bíró objektumokat (esetünkben az embert, aki olvasott) *Observer*-eknek hívjuk (gyakori rokonértelmű kifejezései: *Listener*, *Handler*, *Call-*

back). Egy observer képességgel bíró sípolást érzékelő és arra reakciót ismerő felületű ember képes kezelni a „sípolás" eseményt. Az *Observer* és a *Subject* csak a játékban résztvevő felületeket (a szükséges képességeket) határozzák meg, ugyanis a szép programozási gyakorlatban mindig az ezeket implementáló *ConcreteSubject* és *ConcreteObserver* objektumok a tényleges játékosok. Ezt a szemléletet a felületre való programozásnak nevezzük, az egyik legfontosabb jógyakorlat az OOP-ben. Az *Attach(Observer observer)* metódust gyakran *addListener(Observer observer)* metódusnévvel használjuk (hasonlóan a *Detach()* neve: *removeListener()*).

A további példáinkban a talán leggyakrabban elterjedt *EventSource* és *Listener* neveket fogjuk a legtöbbször megadni a megfigyelt alanyra, illetve a megfigyelő observer-re. Az UML ábrán lévő *Listener* (*Observer*) felület a visszahívható (callback) metódusokat adja meg, amennyiben azoknak van paraméterük, akkor azok között gyakran az események kísérő információi közlekednek. A legjobban elterjedt elnevezési konvenció szerint ezeket a callback metódusokat *onMethodName()* formátummal nevezzük el, de gyakori a *handleMethod()* elnevezési szabály is. Nem ritka, hogy ezeket a callback metódusokat valamilyen esemény típusú adattal paraméterezzük, így az eseményvezérelt környezet megtervezésekor az $E_1, E_2, \ldots E_n$ eseményeket is meg kell alkotnunk és egyes visszahívható metódusok ilyenkor *onMethod(Event event)* vagy *handle(Event event)* paraméterezésű alakot fognak ölteni.

A Subject felületű objektum tehát képes jelenségeket érzékelni vagy más helyről eseményeket befogadni, amikről esetleg jól definiált Event-eket generál, majd a szituációnak megfelelő callback metódusok használatával az összes beregisztrált, Observer felületű megfigyelőt akcióra készteti. Elszakadva a teafőzőtől, képzeljük el, hogy az állapotterünk, azaz a létező működési keretünk egy böngésző. Legyen benne egy *gomb* objektum (*Button* típusú Subject, ami megvalósítja többek között az *addListener()* felületet). Legyen 2 megfigyelőnk: *txt1* és *txt2*. Mindkettő *TextBox* típusú és rendelkezik *onClick()* callback metódussal, megvalósítva ezzel azt az egyszerű observer felületet, ami csak a click-re tud reagálni, azaz kizárólag az *onClick()* akció a tudása. A gombot természetesen egy ember nyomja meg, a böngésző érzékeli ezt, majd - amennyiben úgy rendelkeztünk - ez az információ eljuthat a *gomb* objektumhoz is, amely kiad egy *fire()* (trigger hívó) metódust, amely a leggyakoribb elnevezése annak a metódusnak, amely ténylegesen visszahívja az összes beregisztrált megfigyelő megfelelő metódusát. Példánkban ez a gomb belsejében megbújó metódus most valójában ezt fogja csinálni: *fire()* → *{txt1.onClick(); txt2.onClick();}*

Az observer mintára gyakran úgy tekintenek, mint egy speciális *Publisher/Subcriber* (közzététel és arra való feliratkozás) megvalósításra. Ilyenkor a *Subject* a *Publisher*, míg a *Listener* pedig a *Subscriber* szerepkört tölti be. Ebben a szituációban az események kerülnek sok esetben a fókuszba. Definiálnak egy E ős eseményt, aminek a $E_1, E_2, \ldots E_n$ események a leszármazottjai és a callback metódus tipikusan *receive(E e)* alakú, azaz az események aszinkron közvetítése és feldolgozása játsza a főszerepet.

2.2. Egy eseményvezérelt komponens elkészítése Java környezetben

A 2.1 pont elméleti áttekintése után írjuk meg az első igazi eseményvezérelt megközelítésű példánkat, amihez most a *Java*, a következő pontban pedig a *C#* nyelvet hívjuk segítségül.

Elkészítünk egy *NumberPairComponent* osztályt, ami képes egy számpár fogadására, tárolására. Minden egyes számpárfogadás esetén erről a tényről értesíti a megfigyelőit, átadva nekik a számpárt, amit már egy *NumberPairEvent* for-

mátumú osztály reprezentál. Komponensünk intelligens, azaz a listener megfelelő callback metódusát hívja vissza. Persze ez nem lenne szükségszerű, de mi azt is meg szeretnénk most mutatni, hogy általában egy listener több visszahívási ponttal rendelkezhet.

A forráskód magyarázatát az observer mintát bemutató UML ábra mentén tesszük meg. Ennek megfelelően az első a „Subject” interface (1. programlista), aminek most a neve: *NumberPairEventSource* (természetesen ilyesmi is lehetne az interface neve: *NumberPairSubject*, *NumberPairObservable*). Az *Attach()*, listener beregisztráló metódus neve most *addNumberPairListener(NumberPairListener l)*. A *removeAllListener()* kiregisztrálja az összes előzetesen beregisztrált megfigyelőt. Ezzel megterveztük azt a felületet, ahogy a *NumberPairComponent* osztály használni szeretnénk az observerek be és kiregisztrálásánál.

```
1  // 1. programlista: A Subject interface
2
3  package cs.test.dp.events;
4  /**
5   *
6   * @author inyiri
7   */
8
9  public interface NumberPairEventSource
10 {
11     public void addNumberPairListener(NumberPairListener l);
12     public void removeAllListener();
13 }
```

Menjünk tovább az UML diagram mentén és tervezzük meg az egyetlen event típusunkat! A neve *NumberPairEvent* és nagyon egyszerű (2. programlista). A szerepe mindössze annyi, hogy a callback metódusok ezt a típusú objektumot kapják meg egyetlen argumentumként. Tárolja a számpárt, illetve formázva vissza tudja adni saját maga *String* reprezentációját.

```
1  // 2. programlista: NumberPairEvent class
2
3  package cs.test.dp.events;
4  /**
5   *
6   * @author inyiri
7   */
8  public class NumberPairEvent
9  {
10     public int n1;
11     public int n2;
12
```

```
13      public NumberPairEvent(int n1, int n2)
14      {
15          this.n1 = n1;
16          this.n2 = n2;
17      }
18
19      @Override
20      public String toString()
21      {
22          return "Első: " + n1 + " Második: " + n2 + "\n";
23      }
24  }
```

Harmadik lépésként tervezzük meg azt, hogy mit várunk el egy observer, azaz listener objektumtól, amit az ilyen irányú interface-ének (3. programlista) specifikációjával tehetünk meg. Legyen olyan a megfigyelő objektumunk, hogy 3 fajta akcióra tud reagálni: amikor a számpár első száma, amikor a második száma kisebb, illetve, amikor a 2 szám egyenlő. Ezekben az esetekben mást és mást csináljon, különben nem is volna értelme a 3 metódusnak. Mindegyik callback metódus ajándékul megkap egy *NumberPairEvent* adatot is. Vegyük észre, hogy a callback metódusok szépen követik a de facto szabványt és *onXXX()* alakúak.

```
1  // 3. programlista: NumberPairListener interface
2
3  package cs.test.dp.events;
4  /**
5   *
6   * @author inyiri
7   */
8  public interface NumberPairListener
9  {
10     public void onFirstLess(NumberPairEvent e);
11     public void onSecondLess(NumberPairEvent e);
12     public void onEquals(NumberPairEvent e);
13 }
```

A negyedik lépés az egyik legizgalmasabb, mert itt valósítjuk meg az újrahasznosítható *NumberPairComponent* class-t, aminek a – fentiek alapján – természetesen implementálnia kell a Subject-et, azaz a *NumberPairEventSource* interface-t (4. programlista). Komponensünk alapszolgáltatása a számpár eltárolása, a 14-15 sorok ezért szükségesek. A 17. sor a *listeners* objektuma egy java *List* class, itt tároljuk a beregisztrál listener-eket. A 19-28 sorok a be és kiregisztrálások implementálása (az UML ábra *Attach()*, *Detach()* funkcionalitása). Álljunk meg egy pillanatra a 30-36 sorok *receiveNumberPair()* metódusánál, mert nagyon lényeges a

szerepe! Fogadja és eltárolja a kapott számpárt, késztít belőle egy – a rendszerünk számára ismert – *NumberPairEvent* eseményobjektumot, majd a 35. sorban befejezésképpen „tüzel” egyet, azaz visszahívja a beregisztrált listener-eket. Ennek kódja a 38-59 sor között van. A működés lényege, hogy a *fireEvent()* metódus beépített tudásként a megfelelő callback metódusokat hívja meg az összes beregisztrált listener-re, azaz végigmegy a listener-ek listáján.

```
1  // 4. programlista: A konkrét Subject: NumberPairComponent
2
3  package cs.test.dp.events;
4
5  import java.util.ArrayList;
6  import java.util.List;
7
8  /**
9   *
10  * @author inyiri
11  */
12 public class NumberPairComponent implements NumberPairEventSource
13 {
14     public int n1;
15     public int n2;
16
17     List<NumberPairListener> listeners = new ArrayList<NumberPairListener>();
18
19     public void addNumberPairListener(NumberPairListener l)
20     {
21         listeners.add(l);
22     }
23
24     public void removeAllListener()
25     {
26         listeners.clear();
27
28     }
29
30     public void receiveNumberPair(int n1, int n2)
31     {
32         this.n1 = n1;
33         this.n2 = n2;
34         NumberPairEvent event = new NumberPairEvent(n1, n2);
35         fireEvent(event);
36     }
37
```

```
38      public void fireEvent(NumberPairEvent event)
39      {
40          if (event.n1 < event.n2)
41          {
42              for (int i = 0; i < listeners.size(); i++)
43              {
44                  listeners.get(i).onFirstLess(event);
45              }
46          } else if (event.n2 < event.n1)
47          {
48              for (int i = 0; i < listeners.size(); i++)
49              {
50                  listeners.get(i).onSecondLess(event);
51              }
52          } else
53          {
54              for (int i = 0; i < listeners.size(); i++)
55              {
56                  listeners.get(i).onEquals(event);
57              }
58          }
59      }
60
61  } // end class
```

Ezzel elkészültünk az eseményvezérelt működésre képes *NumberPairComponent* komponensünkkel, azonban a fentiek teljes megértéséhez még a használatának bemutatása van hátra (5. programlista). Készítsünk ehhez egy *Test* osztályt! Az első feladat az, hogy 2 Listener-t csinálunk (akármennyi lehetne): *EgyikNumberPairListener* és *MasikNumberPairListener*. Mindkettő megvalósítja a *NumberPairListener* interface-t. A megvalósított callback metódusok annyira egyszerűek, hogy azokhoz nem fűzünk magyarázatot. Miután van 2 listener fajtánk is, próbáljuk ki a komponensünk működését! Az 55. sorban kezdődő *working()* metódus pont ezt csinálja. A számpárokat jobb híján véletlenszám generátorral fogjuk előállítani, amihez az előkészületeket az 57. sor tartalmazza. Az 58. sorban végre létrehozzuk a *NumberPairComponent* típusú *component* objektumot. A 60-61 sorokban mindkét listener fajtából létrehozunk egy-egy objektumot: *listener1* és *listener2*, amiket a 63-64 sorokban beregisztrálunk a *component* objektumhoz. A 67-72 sorok között generáljuk a számpárokat, majd elküldjük azokat a *component*-nek, ahogy az a 71. sorban látható. Itt történik meg a beregisztrált megfigyelők visszahívása is, amit a program futtatása során láthatunk is, hiszen azok a képernyőre írják működésüket. A 78-82 sorok a tesztprogram elindítását szolgálják.

```
1  // 5. programlista: Eseménykezelés
2
3  package cs.test.dp.events;
4
5  import java.util.ArrayList;
6  import java.util.List;
7  import java.util.Random;
8
9  /**
10  *
11  * @author inyiri
12  */
13 public class Test
14 {
15    // Egy konkrét Listener class
16    public static class EgyikNumberPairListener implements NumberPairListener
17     {
18         public void onFirstLess(NumberPairEvent e)
19         {
20             System.out.println("A kisebb szám:" + e.n1 + "\n");
21         }
22
23         public void onSecondLess(NumberPairEvent e)
24         {
25             System.out.println("A kisebb szám:" + e.n2 + "\n");
26         }
27
28         public void onEquals(NumberPairEvent e)
29         {
30             System.out.println("A kisebb szám:" + e.n1 + "\n");
31         }
32     }
33
34    // Egy másik konkrét Listener class
35    public static class MasikNumberPairListener implements NumberPairListener
36     {
37         public void onFirstLess(NumberPairEvent e)
38         {
39             System.out.println("Kisebb az első számjegy! " + e);
40         }
41
42         public void onSecondLess(NumberPairEvent e)
43         {
```

```
44                 System.out.println("Kisebb a második számjegy!" + e);
45             }
46 
47             public void onEquals(NumberPairEvent e)
48             {
49                 System.out.println("Egyenlő a 2 számjegy!" + e);
50             }
51 
52         }
53 
54         // A komponens (azaz Subject) használata
55         public void working()
56         {
57             Random generator = new Random();
58             NumberPairComponent component = new NumberPairComponent();
59 
60             NumberPairListener listener1 = new EgyikNumberPairListener();
61             NumberPairListener listener2 = new MasikNumberPairListener();
62 
63             component.addNumberPairListener( listener1 );
64             component.addNumberPairListener( listener2 );
65 
66 
67             for (int i = 1; i < 20; i++)
68             {
69                 int n1 = generator.nextInt(10) + 1;
70                 int n2 = generator.nextInt(10) + 1;
71                 component.receiveNumberPair(n1, n2);
72             }
73             component.removeAllListener();
74 
75         }
76 
77         // A tesztprogram belépési (indulási) pontja
78         public static void main(String[] args)
79         {
80            Test test = new Test();
81            test.working();
82         }
83 
84     } // end class
```

Néhány megjegyzés. A gyakorlati munkában felbukkan még néhány olyan fogalom, amit érdemes megérteni, hogy teljesebbek legyenek az ismereteink az eseményvezérelt környezetek programozásáról. A következő néhány megjegyzés ezt foglalja össze nagyon tömören.

1. Gyakori, hogy nem akarjuk az összes callback metódust implementálni, mert az üzleti igény ezt nem követeli meg. Példánknál maradva, lehetséges, hogy csak azokat a számpárokat akarjuk kiírni a képernyőre, ahol az első szám a kisebb, azaz ekkor valódi implementációt csak az *onFirstLess()* metódusnak akarunk készíteni, a másik kettő pedig ilyen alakú: *onSecondLess(NumberPairEvent e) {}* és *onEquals(NumberPairEvent e) {}*, azaz mindkettő üres törzzsel implementált, hogy ne csináljon semmit. Ez kötelező, különben a *NumberPairListener* interface nem lenne implementálva. Ilyenkor kényelmi szempontból szoktak készíteni egy *Adapter* class-t, esetünkben például ilyen neve lehetne: *NumberPairListenerAdapter*. Ez az osztály implementálja a Listener-t, mindegyik metódust üres törzzsel. Ez az osztály kifejezetten öröklési célra készül ilyenkor és a feladata csak annyi, hogy az utódosztály felülírhassa azon callback metódusait, amit az üzleti igény megkövetel. Az elnevezésben használt *Adapter* postfix használata elterjedt, használjuk mi is!

2. Az események lekezelése sokszor hosszú idét igényelhet, amennyiben a callback metódus (például az *onFirstLess()*) nem olyan egyszerű. Ilyenkor érdemes megfontolni, hogy az algoritmust egy külön végrehajtási szálon végezzük el, azaz az *onXXX()* metódus futása az új *Thread* létrehozására és elindítására korlátozódik, így gyorsan vissza fog térni, ezzel nem blokkolja le a komponens működését. Ez a szempont sokszor a GUI felületen szokott felmerülni, de természetesen mindenütt fontos, ahol egy eseményvezérelt komponens elevenségi jellemzőjét megköveteli az üzleti logika.

3. Gyakori, hogy az eseménykezelő megkapja a sender objektum referenciáját is. Ekkor például az *onFirstLess()* callback metódus ilyen lenne pontosan:
public void onFirstLess(Object sender, NumberPairEvent e)
Ebben az esetben a *fireEvent()* metódusban a
listeners.get(i).onFirstLess(event) hívás ilyenné változik:
listeners.get(i).onFirstLess(this, event), azaz megkapja a kérdéses *NumberPairComponent* objektum referenciáját. Gyakori technika, hogy az 1 db *Event* paraméter mégis marad a callback metódusban, de ekkor az *Event* class egészül ki egy például *Object* típusú és *sender* vagy *eventSource* nevű adattaggal, így ezen az úton az eseménykezelő hozzáfér ahhoz az információhoz, hogy ki volt az üzenet küldője.

4. Érdemes észrevenni, hogy egy Listener is lehet Subject szerepkörben, amennyiben megvalósítja az *EventSource* interface-t. Ilyenkor egy eseménykezelő láncot alakíthatunk ki.

5. Amikor pusblish/subscribe modell alapján szervezzük az eseménykezelést, akkor a Listener általában csak 1 *receive(Event event)* metódussal rendelkezik.

6. A *fireEvent(event)* funkcionalitású metódusban a *fire* szó helyett gyakran a *notify()* használatos.

2.3. Egy eseményvezérelt komponens elkészítése C# környezetben

Készítsük el az előző pontban elkészített *NumberPairComponent* C#-ban megvalósított verzióját, amivel áttekintjük azt is, hogy ebben a környezetben milyen módon valósul meg az eseménykezelés. A C# a Listener interface és annak implementációja helyett a *metódusreferenciát* használja callback metódusként. A 6. programlistán látható forráskód 22-38 soraiban a már megismert *NumberPairEvent* class-t implementáljuk, szerepe pont ugyanaz, mint a Java-beli testvérének. A 8. sor így néz ki: *public delegate void NumberPairListener(NumberPairEvent e).* Itt egy metódusreferenciát definiáltunk a *delegate* kulcsszó használatával. A *NumberPairListener* név itt egy típus (class) szemantikával bíró név, azaz ilyen típusú objektumokat hozhatunk létre, amik hivatkozhatnak majd olyan metódusokra, amik pont ugyanilyen szignatúrájúak. A háttérben a *delegate* mechanizmus valójában egy teljes class-t generál le (ezért típus értékű), aminek egyetlen üzleti metódusa van, viszont ezt *szinkron* és *aszinkron* módon is meg lehet hívni. Természetesen ez az üzleti metódus az a metódus, amire a metódusreferencia éppen „mutat”. A 8. sorban létrehozott metódusreferenciára tekintve felismerhetjük, hogy az pont olyan, mint a Javas példák *onXXX()* callback metódusa. A 11. sort csak érdekességként tüntettük fel, gondolva az előző pont 3. megjegyzésére. A példában azonban ezt nem fogjuk használni. A 14-20 sorok az *NumberPairEventSource* interface-t definiálják, azonban itt az *Attach()* jellegű metódus minden callback-re külön van megadva, ezért példánkban 3 beregisztráló metódus szerepel. A 41-90 sorok a *NumberPairComponent* megvalósítását adják, ugyanazzal a funkcionalitással, mint ahogy azt a Java-s példában tettük. A 46-48 sorok újdonságot jelentenek a Java-s példához képest, ugyanis itt 3 darab metódusreferencia objektumot hozunk létre a *listeners List*-beli objektum helyett. Az elnevezési konvencióra azonban itt is nagyon vigyáztunk, például: *public NumberPairListener onFirstLess* a változó neve. Szeretném kiemelni, hogy a megfigyelő beregisztráló metódusok mögött nincs most egy *List* adatszerkezet, ezt a tudást a metódusreferencia C# nyelvi konstrukció „gyárilag” tudja. Ez azt jelenti, hogy a += (például 52. sorban) operátorral akárhány metódust hozzáfűzhetünk a metódusreferencia változóhoz. Persze az egyszerű =, értékadó operátor is használható. A *NumberPairComponent* implementációjának többi része könnyen érthető és a Java-s esettel teljesen egybevág. A komponens elkészülte után próbáljuk is ki (7. programlista), legyen itt is a *Test* class a 93. sorban kezdődő tesztosztály. A 95-123 sorokban létrehozunk pár eseménykezelő metódust, ezeket adjuk majd értékül az *onXXX* metódusreferencia objektumunknak. A 126. sorban kezdődő *working()* metódus szerepét is ismerjük már az előző pontból. A működése is pont ugyanaz, véletlen számpárokból csinálunk eseményeket, amiket „kilövünk”. Előzetesen szeretném felhívni a figyelmet egy érdekességre, amit a megjegyzésbe tett 138-145 sorok mutatnak, ugyanis ez volt a program előző verziója, ekkor hiányoztak a 131-136 sorok. A comment-be tett sorok hibátlanok, de elég bonyolult így a Java-hoz hasonlító beregisztráló módszer. Szerencsére a C# rendelkezik azzal a lehetőséggel, hogy ne kelljen beregisztráló metódusokat készíteni egy komponenshez, így a 131-136 sorok egyből használhatók. Szép szintaktika, igazán tetszik! Ugyanakkor itt kicsit szétesik a Listener felület, ami 1 callback metódus per Listener esetén nem zavaró, sőt előny. Több metódus esetén viszont inkább hátránynak tűnik, de valószínűleg ez az általános eset a ritkábban is fordul elő. A 147-154 sorok közötti for ciklust, illetve az utána lévő kiregisztrálást (csak a példa kedvéért van) már jól ismerjük az előző pontból.

```
1  // 6. programlista: A komponens elkészítése C# környezetben
2
3  using System;
4
5  namespace Test
6  {
7          //
8          public delegate void NumberPairListener(NumberPairEvent e);
9
10         //
11         public delegate void NumberPairListener2(object sender, NumberPairEvent e);
12
13         //
14         public interface NumberPairEventSource
15         {
16                 void addOnFirstLessListener(NumberPairListener listener);
17                 void addOnSecondLessListener(NumberPairListener listener);
18                 void addOnEqualsListener(NumberPairListener listener);
19                 void removeAllNumberPairListener();
20         }
21
22         //
23         public class NumberPairEvent
24         {
25                 public int n1;
26                 public int n2;
27
28                 public NumberPairEvent(int n1, int n2)
29                 {
30                         this.n1 = n1;
31                         this.n2 = n2;
32                 }
33
34                 public String toString()
35                 {
36                         return "Első: "+ n1 + " Második: " + n2;
37                 }
38         }
39
40         //
41         public class NumberPairComponent : NumberPairEventSource
42         {
43                 public int n1;
44                 public int n2;
45
46                 public NumberPairListener onFirstLess;
47                 public NumberPairListener onSecondLess;
48                 public NumberPairListener onEquals;
49
50                 public void addOnFirstLessListener(NumberPairListener listener)
51                 {
52                         onFirstLess += listener;
53                 }
54
55                 public void addOnSecondLessListener(NumberPairListener listener)
56                 {
57                         onSecondLess += listener;
58                 }
59
60                 public void addOnEqualsListener(NumberPairListener listener)
61                 {
62                         onEquals += listener;
63                 }
64
65         //
```

```
66                  public void removeAllNumberPairListener()
67                  {
68                          onFirstLess = null;
69                          onSecondLess = null;
70                          onEquals = null;
71                  }
72
73          //
74                  public void fireEvent(NumberPairEvent e)
75                  {
76                          if (e.n1 < e.n2) onFirstLess( e );
77                          else if (e.n1 > e.n2) onSecondLess( e );
78                          else onEquals( e );
79                  }
80
81          //
82                  public void receiveNumberPair(int n1, int n2)
83                  {
84                          this.n1 = n1;
85                          this.n2 = n2;
86                          NumberPairEvent e = new NumberPairEvent(n1, n2);
87                          fireEvent(e);
88                  }
89
90          }
91
92          //
93          public class Test
94          {
95                  public void Egy_FirstLess(NumberPairEvent e)
96                  {
97                          Console.WriteLine("Első a kisebb: " + e.toString());
98                  }
99
100                 public void FFF_FirstLess(NumberPairEvent e)
101                 {
102                         Console.WriteLine("FFF :-): " + e.toString());
103                 }
104
105                 public void Egy_SecondLess(NumberPairEvent e)
106                 {
107                         Console.WriteLine("Második a kisebb: " + e.toString());
108                 }
109
110                 public void SSS_SecondLess(NumberPairEvent e)
111                 {
112                         Console.WriteLine("SSS :-): " + e.n2);
113                 }
114
115                 public void Egy_Equals(NumberPairEvent e)
116                 {
117                         Console.WriteLine("Egyenlőek: " + e.n1);
118                 }
119
120                 public void EEE_Equals(NumberPairEvent e)
121                 {
122                         Console.WriteLine("EEE :-): " + e.n1);
123                 }
124
125         //1-10 közötti
126     public void working()
127         {
128                 Random RandomClass = new Random();
129                 NumberPairComponent component = new NumberPairComponent();
130
```

```
131             component.onFirstLess += Egy_FirstLess;
132             component.onFirstLess += FFF_FirstLess;
133             component.onSecondLess += Egy_SecondLess;
134             component.onSecondLess += SSS_SecondLess;
135             component.onEquals += Egy_Equals;
136             component.onEquals += EEE_Equals;
137
138 //          component.addOnFirstLessListener( new NumberPairListener( Egy_FirstLess ) );
139 //          component.addOnFirstLessListener( new NumberPairListener( FFF_FirstLess ) );
140 //
141 //          component.addOnSecondLessListener( new NumberPairListener( Egy_SecondLess ) );
142 //          component.addOnSecondLessListener( new NumberPairListener( SSS_SecondLess ) );
143 //
144 //          component.addOnEqualsListener( new NumberPairListener( Egy_Equals ) );
145 //          component.addOnEqualsListener( new NumberPairListener( EEE_Equals ) );
146
147             for (int i=1; i < 20; i++)
148             {
149                 int n1 = RandomClass.Next(1, 11);
150                 int n2 = RandomClass.Next(1, 11);
151
152                 component.receiveNumberPair(n1, n2);
153
154             }
155
156             component.removeAllNumberPairListener();
157         }
158         }
159 }
```

Végül nézzük meg a főprogramot, azaz a *Test* osztály használatát!

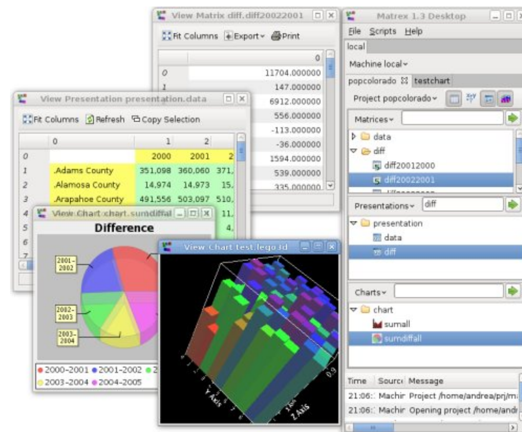
```
1  // 7. programlista: A C# komponens tesztelése
2
3  using System;
4
5  namespace Test
6  {
7      class MainClass
8      {
9          public static void Main(string[] args)
10         {
11             Test test = new Test();
12             test.working();
13         }
14     }
15 }
```

Látható, hogy C#-ban ugyanolyan elv alapján valósul meg az eseménykezelés, mint Javaban, azaz nincs lényegi különbség. A már említett megjegyzések természetesen itt is érvényesek. A fenti 2. megjegyzéshez itt egy kis kiegészítés jár, ugyanis a *delegate* kulcsszóval megalkotott metódusreferenciák már beépítve lehetővé teszik az aszinkron hívást, ahol az üzleti metódus(ok) egy külön szálon futnak. Ennek részleteire ebben a cikkben nem térünk ki, de egyszerűen megérthető mechanizmusról van szó.
A továbbiakban röviden áttekintünk néhány ismertebb gyári C# és Java eseménykezelő megoldást abban a reményben, hogy a fentiekben kialakított jó szemléletmód kiváló alapot ad a megértéshez.

2.4. SWT

Ezt a Java GUI keretrendszert az IBM készítette és itt lehet többet megtudni róla, illetve letölteni azt a könyvtárat, amit a programjainkhoz szerkesztve használhatjuk azt: `http://www.eclipse.org/swt/`. A jól ismert *eclipse* környezet is ennek a felhasználásával készül és kiváló grafikus programokat készíthetünk ezzel az eszközzel.

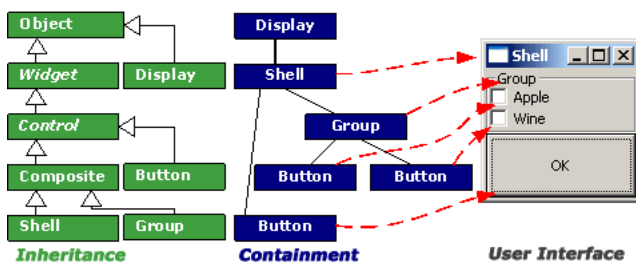
Az *SWT* a Standard Widget Toolkit rövidítése. Példaképen nézzünk meg egy ebben a frameworkben készített alkalmazást a 11. ábrán. A továbbiakban (8. programlista) tekintsünk egy példát, azaz elemezzük egy kicsit az *AL* class-t, ami egy futtatható *SWT* alkalmazás, így az ott leprogramozott komponenseknek „csak” a használatát nézzük meg most. Az *AL* class maga implementálja a *WindowListener* és *ActionListener* interface-eket. A 2. interface egyedüli callback metódusa az *actionPerformed(ActionEvent e)*. A 34-44 metódusok a *WindowListener* metódusai, de a *windowClosing()* (ami elvégezteti a még hátralévő események feldolgozását és kilépteti a programot) kivételével mindegyik üres törzzsel van implementálva, itt akár Adapter-t is használhatott volna a kód írója. A program egyébként nagyon egyszerű dolgot csinál. Van neki egy nyomógombja és egy text mezője. Minden gombnyomásra 1-gyel nő a kijelzett szám. A futtatható *AL* class 18-27 sorok között lévő konstruktora az örökölt *Frame*-be (ez a GUI gyökér widget konténere) regisztrálja a gombot (24. sor) és a text mezőt (25. sor). Az *addWindowListener(this)* azt jelenti, hogy regisztrálja azt a WindowListener-t, amit az *AL* (azaz saját magunk) objektum implementál. A regisztráció közvetlenül az *AL* class-ba történik, azaz Ő játsza most a Subject szerepet, ami logikus is, hiszen a felhasználó rá kattint kilépéskor, azaz Ő tudja kibocsátani az eseményt is (azaz ő az *EventSource* más szavakkal). A 26. sor *b.addActionListener(this)* hívása a *b* nyomógombhoz regisztrálja a 29-32 sorokban megírt *actionPerformed()* callback metódust. Itt más szavakkal arról van szó, hogy egy olyan objektumot kell regisztrálni, ami az *ActionListener* interface-t valósítja meg, ez pedig maga az *AL*-beli *this* („saját magam” objektum), ami rendelkezik a megfelelő *actionPerformed()* metódussal. A 12. sortól indul a program.



11. ábra. Matrex SWT alkalmazás

```
1  // 8. programlista: Java SWT példa
2
3  import java.awt.*;
4  import java.awt.event.*;
5
6  public class AL extends Frame
7  implements WindowListener, ActionListener {
8          TextField text = new TextField(20);
9          Button b;
10         private int numClicks = 0;
11
```

```
12          public static void main(String[] args) {
13                  AL myWindow = new AL("My first window");
14                  myWindow.setSize(350,100);
15                  myWindow.setVisible(true);
16          }
17
18          public AL(String title) {
19
20                  super(title);
21                  setLayout(new FlowLayout());
22                  addWindowListener(this);
23                  b = new Button("Click me");
24                  add(b);
25                  add(text);
26                  b.addActionListener(this);
27          }
28
29          public void actionPerformed(ActionEvent e) {
30                  numClicks++;
31                  text.setText("Button Clicked " + numClicks + " times");
32          }
33
34          public void windowClosing(WindowEvent e) {
35                  dispose();
36                  System.exit(0);
37          }
38
39          public void windowOpened(WindowEvent e) {}
40          public void windowActivated(WindowEvent e) {}
41          public void windowIconified(WindowEvent e) {}
42          public void windowDeiconified(WindowEvent e) {}
43          public void windowDeactivated(WindowEvent e) {}
44          public void windowClosed(WindowEvent e) {}
45
46  }
```



12. ábra. Egy SWT alkalmazás szerkezete

2.5. Google Web Toolkit (GWT)

A GWT azért izgalmas technológia, mert tisztán Java nyelven megírható a teljes web alapú alkalmazás, amiből a gwt Java→Javascript fordítóprogramja készíti el a „futtatható” targetet (alapvetően html és javascript file-ok halmaza). Nézzük meg a lenti *Hello* class által megvalósított alkalmazást (9. programlista)! Itt egyetlen nyomógomb van Subject szerepben. A 19-23 sorokban regisztrálunk egy névtelen *ClickHandler* interface-t megvalósító callback metódust, ami csak egy dialógus ablakban a „Hello, AJAX” szöveget jeleníti meg. Látható tehát, hogy miért jó a GWT. Lehetővé teszi a design patternek, például az observer minta használatát, amihez egy igazi OO nyelvet használhatunk Javascript helyett.

```
 1 // 9. programlista: A GWT eseménykezelése
 2
 3 package com.google.gwt.sample.hello.client;
 4
 5 import com.google.gwt.core.client.EntryPoint;
 6 import com.google.gwt.event.dom.client.ClickEvent;
 7 import com.google.gwt.event.dom.client.ClickHandler;
 8 import com.google.gwt.user.client.Window;
 9 import com.google.gwt.user.client.ui.Button;
10 import com.google.gwt.user.client.ui.RootPanel;
11 import com.google.gwt.user.client.ui.Widget;
12
13 /**
14  * Hello World application.
15  */
16 public class Hello implements EntryPoint {
17
18   public void onModuleLoad() {
19     Button b = new Button("Click me", new ClickHandler() {
20       public void onClick(ClickEvent event) {
21         Window.alert("Hello, AJAX");
22       }
23     });
24
25     RootPanel.get().add(b);
26   }
27 }
```

És most egy újabb izgalmas példa jön, ugyanis a *NumberPairComponent* mellett megtervezzük a 2. igazi eseményvezérelt, a GWT-ben használható új komponensünket. A neve *LinkWidget* lesz (10. programlista). Amikor teljesen új *Widget*-et készítünk (azaz a böngésző DOM szintjén implementáljuk a működést), akkor a GWT *Widget* ősosztály kell használni. Itt a linkre kattintás lesz az egyetlen esemény, amit a *LinkWidget* kibocsát – mint Subject – ezért implementálni kell a *HasClickHandlers* interface-t, ami az *addClickHandler(ClickHandler handler)* beregisztráló (a mintabeli *Attach()*) metódus megvalósítását jelenti a 44-48 sorokban. Az implementáció nem nehéz, mert az *addDomHandler()* – valódi munkát végző – metódust a *Widget* ajándékba adja nekünk, a formája pedig kötött, mindig így kell megírni. Nézzük meg egy kicsit a 25-30 sorokban lévő konstruktort! A 27. sorban létrehozunk egy új DOM node elemet, ami egy link típusú, majd a 28. sorban az ős *Widget sinkEvents()* metódusával rendelkezünk arról, hogy milyen eseményeket adjon a browser tovább ennek a *LinkWidget* vezérlőnek. Gondolkodjunk el egy pillanatra! Mi is van itt? Minden eseményt a böngésző fog el, ő osztja azokat szét, így valahonnan ki kell derülni számára, hogy egy

click esetén továbbítsa-e a *LinkWidget*-nek. A 29. sorban csak azt intéztük el, hogy az alapértelmezett böngészőműködés ne érvényesüljön, azaz ne akarjon egy új lapot betölteni. Ehhez a 33-36 sorokban implementált *setUrl()*-t használjuk, ami az újdonsült nodunknak egy „href” attribútumot ad. A 39-42 kódrészlet egyértelmű, innen is látható, hogy a teljes Widget implementáció a DOM használatával történik. Ezek a sorok szinte egy az egyben fordulnak a megfelelő Javascript sorokra, ahol mindezt hasonlóan tennénk. Az eseménykezelés böngészőben a DOM+Javascript párost jelenti, ezért a beregisztráló metóduson nem jelent semmi újat.

És most nézzük meg a használatát a *LinkWidget* gadget-nek, ezt az 54-64 közötti forrás mutatja. Az eddigiek alapján a működése teljesen egyértelmű kell, hogy legyen, amikor a "A kedvenc helyem" link-re kattintunk megnyílik egy ablak, amiben tartalomként a `http://e-matematika.lap.hu/`hely tartalma töltődik.

```
1  // 10. programlista: Egy új komponens és annak használata
2
3  package org.cs.udportal.client.desktoplib;
4
5  import com.extjs.gxt.ui.client.widget.Window;
6  import com.google.gwt.event.dom.client.ClickEvent;
7  import com.google.gwt.event.dom.client.ClickHandler;
8  import com.google.gwt.event.dom.client.HasClickHandlers;
9  import com.google.gwt.event.shared.EventHandler;
10 import com.google.gwt.event.shared.GwtEvent;
11 import com.google.gwt.event.shared.GwtEvent.Type;
12 import com.google.gwt.event.shared.HandlerManager;
13 import com.google.gwt.event.shared.HandlerRegistration;
14 import com.google.gwt.user.client.DOM;
15 import com.google.gwt.user.client.Event;
16 import com.google.gwt.user.client.ui.Widget;
17
18
19 public class LinkWidget extends Widget
20                         implements HasClickHandlers
21 {
22     ClickHandler clickHandler;
23
24     // Konstruktor
25     public LinkWidget()
26     {
27         this.setElement(DOM.createAnchor());
28         this.sinkEvents(Event.ONCLICK | Event.MOUSEEVENTS);
29         setUrl( "#" );
30     }
31
```

```
32     // A link URL-je
33     public void setUrl(String url)
34      {
35         DOM.setElementAttribute(this.getElement(), "href", url);
36      }
37
38      // A link szövege
39      public void setText(String text)
40      {
41          DOM.setInnerText(this.getElement(), text);
42      }
43
44      // A beregisztráló metódus
45      public HandlerRegistration addClickHandler(ClickHandler handler)
46      {
47          return addDomHandler( handler, ClickEvent.getType() );
48      }
49 } // end class
50
51 //
52 // Használata
53 //
54 LinkWidget lw = new LinkWidget();
55 lw.setText("A kedvenc helyem");
56 ClickHandler ch = new ClickHandler() {
57       public void onClick(ClickEvent event)
58       {
59           Window w = new Window();
60           w.setUrl( "http://e-matematika.lap.hu/" );
61           w.show();
62       }
63 };
64 lw.addClickHandler(ch);
```

2.6. A .NET Form

Az egyes környezetek közötti kalandozásunkat a .NET Form-ra pillantással folytatjuk. Már tudjuk, hogy a C# környezetben az eseménykezelők olyan metódusok, amiket a metódusreferencia típusú objektumokhoz rendelünk (az *Attach()* művelet gyanánt). Tekintsük a *HelloWorld* GUI példát (11. programlista)! Érdemes tudnunk, hogy a .NET környezet tartalmaz egy előre definiált metódusreferencia típust, ami a következő: *public delegate void EventHandler (object sender, EventArgs e)*
Ilyen típusú a legtöbb Form alapú eseménykezelő, így a mi *Button_Click()* metódusunkat is eképpen kell készíteni, ugyanis az eseményvisszahívó a *Button* komponens ilyenre számít.

A konstruktorból az is kikövetkeztethető, hogy a *Click* változó tárolja (ami emiatt *EventHandler* típusú) el a *Button_Click* callback metódus referenciáját. Szeretném ezen név kapcsán felhívni a figyelmet egy érdekes callback metódus elnevezési konvencióra! Itt az eseménykezelő neve a *Button* class és a *Click* esemény összetapasztásából képzett. Gyakoribb – különösen a designer eszközök használata során – a *b_click* név, amikor a Subject szerepkörben lévő objektum (esetünkben *b*) neve a névképzés első tagja. Gondoljunk arra, hogy több gomb is lehet, eltérő eseménykezelésekkel!

```
1  // 11. programlista: A .NET Form eseménykezelése
2
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  public class HelloWorld : Form
8  {
9          static public void Main ()
10         {
11                 Application.Run (new HelloWorld ());
12         }
13
14         public HelloWorld ()
15         {
16                 Button b = new Button ();
17                 b.Text = "Click Me!";
18                 b.Click += new EventHandler (Button_Click);
19                 Controls.Add (b);
20         }
21
22         private void Button_Click (object sender, EventArgs e)
23         {
24                 MessageBox.Show ("Button Clicked!");
25         }
26 }
```

2.7. Qt

A Qt elemkészlet C++ nyelven készült, sok híres program használja, mi csak egyet említünk meg, a KDE Destop környezetet. A *Qyoto* project lehetővé teszi, hogy használjuk a Qt-t C# nyelvből, így ezzel megörököljük a teljes .NET hátteret is. A Qt kicsit érdekes szóhasználatot és technikát alkalmaz az eseménykezelésre, mert *SIGNAL*-ok és *SLOT*-ok vannak. Egy objektum tartalmazhat signal deklarációkat és slot metódus implementációkat is. A signal úgy néz ki, mint egy metódus, lehetnek paraméterei, de sosem kell implementálni. A feladata mindössze az, hogy az eseménykezelő a signal (jel) nevét

az esemény neveként jegyezze fel, a paraméterei pedig a már említett *Event* jellegű class megfelelője, azaz a kibocsátott jelet kísérő adatok deklarációja. A slot (csatlakozó) egy valódi metódus, de a paraméterezése olyan kell legyen, mint annak a jelnek, amire csatlakozni képes és amit fogad, különben az abban lévő adatokat nem tudná átvenni. Gondoljunk bele! A SIGNAL a Subject szerepkörhöz tartozik, a SLOT pedig a callback metódus álneve. A jeleket és a csatlakozókat mindig a 15. sorban is látható *Connect()* metódussal kötjük össze (12. programlista), ahol azt kell megmondanunk, hogy melyik objektum, milyen jelére, melyik objektum csatlakozó (SLOT) metódusát kell meghívni. A program egyébként csak annyit csinál, hogy megjelenít egy ablakot, amiben 1 db nyomógomb van "Quit button" felirattal. Amikor megnyomjuk, akkor a *quit* nevű gomb kiad egy paraméter nélküli *click()* SIGNAL-t, amihez egyedül az alkalmazást reprezentáló *qApp* objektum *quit()* slot-ja van kötve, ez kilépteti az alkalmazást.

```
1  // 12. programlista: A Qt könyvtár eseménykezelése
2
3  using System;
4  using Qyoto;
5
6  public class QyotoApp : QWidget {
7      public QyotoApp() {
8          SetWindowTitle("Quit button");
9          InitUI();
10         Resize(250, 150);
11         Move(300, 300);
12         Show();
13     }
14
15     public void InitUI() {
16         QPushButton quit = new QPushButton("Quit", this);
17         Connect(quit, SIGNAL("clicked()"), qApp, SLOT("quit()"));
18         quit.SetGeometry(50, 40, 80, 30);
19     }
20
21     public static int Main(String[] args) {
22         new QApplication(args);
23         new QyotoApp();
24         return QApplication.Exec();
25     }
26 }
```

2.8. Gtk

A Gtk a GIMP toolkit elemkészlete, de ma már inkább az ismert GNOME desktop jut róla az eszünkbe. A lenti példa az eddigiek alapján triviális, csak a szemléltetés kedvéért tettük be ebbe az írásba.

```
// 13. programlista: A Gtk könyvtár eseménykezelése

using Gtk;
using System;

class SharpApp : Window {

    public SharpApp() : base ("Button")
    {
        SetDefaultSize(250, 200);
        SetPosition(WindowPosition.Center);

        DeleteEvent += delegate { Application.Quit(); };

        Fixed fix = new Fixed();

        Button quit = new Button("Quit");
        quit.Clicked += OnClick;
        quit.SetSizeRequest(80, 35);

        fix.Put(quit, 50, 50);
        Add(fix);
        ShowAll();
    }

    void OnClick(object sender, EventArgs args)
    {
        Application.Quit();
    }

    public static void Main()
    {
        Application.Init();
        new SharpApp();
        Application.Run();
    }
}
```

2.9. A Java beépített observer minta támogatása

A Java már a kezdeti verziók óta támogatja a megfigyelő tervezési minta szerint épített kód könnyű lehetőségét. Ehhez 2 előregyárott eszközt ad: az *Observer* interface-t és az *Observable* ősosztályt. A lenti (14. programlista) *EventSource* class – ahogy a neve is utal rá – a Subject szerepkört tölti majd be. A kódja egy külön szálon képes futni, így ebben a példában azt is be tudjuk mutatni, hogy milyen az, amikor a főszál „kívülről", azaz egy másik száltól kapja a „lekezelendő" eseményeket. Ez a külön végrehajtható szál egyébként nagyon egyszerű, végtelen ciklusban kéri a billentyűzetről a szövegsorokat, majd az „alanyunk" a *setChanged()* beállításával magát olyan állapotba hozza, hogy most valami esemény történt, így a *notifyObservers(response)* callback metódus visszahívja az összes beregisztrált megfigyelőt. Ennek a metódusnak *Object* típusú a paramétere, ezért a *String* helyett bármi más *Event* jellegű message osztály is lehetne az elküldött üzenet.

```
1  // 14. programlista: A Java beépített observer minta támogatása
2
3  package cs.test.dp.observer;
4
5  import java.io.BufferedReader;
6  import java.io.IOException;
7  import java.io.InputStreamReader;
8  import java.util.Observable;
9
10 //
11 // Az Observable, azaz Megfigyelhető osztály tartalmaz olyan metódusokat, amik
12 // jelenteni tudnak magukról, ha be beállítjuk, hogy változtak.
13 // Ők a megfigyeltek (alanyok, subject-ek, stb...), azaz eseményforrások
14 //
15 // Ez a példa egy külön szálon futó algoritmus, ami minden <enter> után
16 // értesíti a megfigyelőit (akik utána persze futtatják az update()-jeiket
17 //
18 public class EventSource extends Observable implements Runnable
19 {
20     public void run()
21     {
22         try
23         {
24             final InputStreamReader isr = new InputStreamReader(System.in);
25             final BufferedReader br = new BufferedReader(isr);
26             while (true)
27             {
28                 final String response = br.readLine();
29                 setChanged();
30                 notifyObservers(response);
```

```
31                }
32            } catch (IOException e)
33            {
34                e.printStackTrace();
35            }
36
37        }
38 } // end class
```

A következő lépésben készítsük el a megfigyelőt, amiből most csak 1 fajtát csináltunk (15. programlista). A *KiiroObserver* class a megfigyelő neve – azaz a Listener osztályunk – ami a már említett *Observer* interface-t valósítja meg, aminek – ahogy látható – egyetlen, a visszahívást szolgáló *update()* metódusa van. Amit csinál nagyon egyszerű. Egyik paraméterében átveszi az üzenetküldő komponens referenciáját (sender), azaz az *Ovservable* objektumot. A másik paraméterével pedig az üzenet referenciáját kapja meg, amit kiír a képernyőre.

```
1  // 15. programlista: A megfigyelő Java beépített támogatással
2
3  package cs.test.dp.observer;
4
5  //
6  import java.util.Observable;
7  import java.util.Observer;
8
9  //
10 // Ő egy megfigyelő. Amikor kap egy jelzést az Megfigyelttől, akkor az
11 // update() metódussal reagál erre
12 //
13 public class KiiroObserver implements Observer
14 {
15
16     private String resp;
17
18     public void update(Observable o, Object arg)
19     {
20         if (arg instanceof String)
21         {
22             resp = (String) arg;
23             System.out.println("\nReceived Response: " + resp);
24         }
25     }
26 }
```

Ennyi előkészítés után elkészíthető a *Test* tesztosztály (16. programlista). Létrehozza az *evSrc* alanyt (subject), utána a *kiiroMegfigyelo* megfigyelőt, amit a *evSrc.addObserver(kiiroMegfigyelo)* sorban azonnal be is regisztrál az alanyhoz. A *main()* metódus végén a subject elindul egy külön szálon és amennyiben a felhasználó ENTER gombot üt (esetleg előtte néhány más karaktert, mint üzenetet), azt megjeleníti a képernyőn.

```
1  // 16. programlista: Eseménykezelés
2
3  package cs.test.dp.observer;
4
5  public class Test
6  {
7
8      public static void main(String args[])
9      {
10         System.out.println("Enter Text >");
11
12         // create an event source - reads from stdin - Egy MEGFIGYELT objektum
13         final EventSource evSrc = new EventSource();
14
15         // create an observer - Egy MEGFIGYELŐ objektum
16         final KiiroObserver kiiroMegfigyelo = new KiiroObserver();
17
18         // A MEGFIGYELT-be regisztrál a MEGFIGYELŐ
19         evSrc.addObserver( kiiroMegfigyelo );
20
21         // starts the event thread
22         Thread thread = new Thread( evSrc ); // ennek a run()-ját futtatja
23         thread.start();
24     }
25 }
```

Természetesen még sok környezetből lehetett volna példákat adni az eseménykezelés megvalósításának filozófiájára, de azok felderítését a kedves olvasóra bízzuk. Előre szeretnénk felhívni a figyelmet, hogy a következő cikk-ben is találunk részleteket az eseménykezelésre.

3. SAP függvény készítése JAVA környezetben

Képzeljünk el egy olyan szerver alkalmazást, ami hálózaton keresztül lehetővé teszi egy SAP instancia számára, hogy az valamilyen szolgáltatáshalmazt igénybe vegyen. Ilyen megoldások léteznek, a legtriviálisabb példa egy másik SAP példány használata. Egy vállalatnál vagy az Internetről rengeteg hasznos létező szolgáltatás érhető el. Nem kell feltétlenül a legbonyolultabb dolgokra gondolni, lehet az is egy szolgáltatás, hogy lekérdezzük és továbbszolgáltatjuk a deviza árfolyamokat.
Összetettebb szolgáltatásra lehet példa egy valamilyen mesterséges intelligencia rendszer, amit az SAP-os üzleti folyamatba be szeretnénk építeni, de ott nincs meg és nem tervezik ezt a funkcionalitást megvalósítani. Erre példa egy vevőtörzs ellenőrző, ami egy új vevő esetén megpróbálja eldönteni a bevitt cím és a vevő neve alapján, hogy vajon ez a vevő kissé más néven nem lett-e már rögzítve.
Az SAP oldali példakód elkészítését *Bajusz Péter* kollégám vállalta magára, amit itt külön szeretném megköszönni.

Ebben a cikkben áttekintjük egy példa segítségével, hogy egy SAP környezet számára hogyan tudunk külső JAVA programokat – mint elérhető szolgáltatásokat – biztosítani. A felhasznált technológia a *SAP JCo* (Java Connector). A példa neve *BapiServer* lesz, ami hasonló viselkedést fog mutatni más külső SAP site-okhoz, azaz egy SAP-on belüli ABAP programozó nem is sejti majd, hogy az általa meghívott távoli függvény valójában nem is egy másik SAP instancián fut. A Java-ban megírt szolgáltatás persze lehet egy delegáció is egy másik meglévő szolgáltatáshoz, ami különösen fontos tulajdonság, mert a Java egyik fő ereje éppen abban van, hogy rengeteg technológiát és szolgáltatást el tud érni, amit esetünkben a SAP rendszer felé annak SAP RFC-nek[3] nevezett „anyanyelvén” tud prezentálni.

3.1. Az RFC function (BAPI) felületének megtervezése és létrehozása

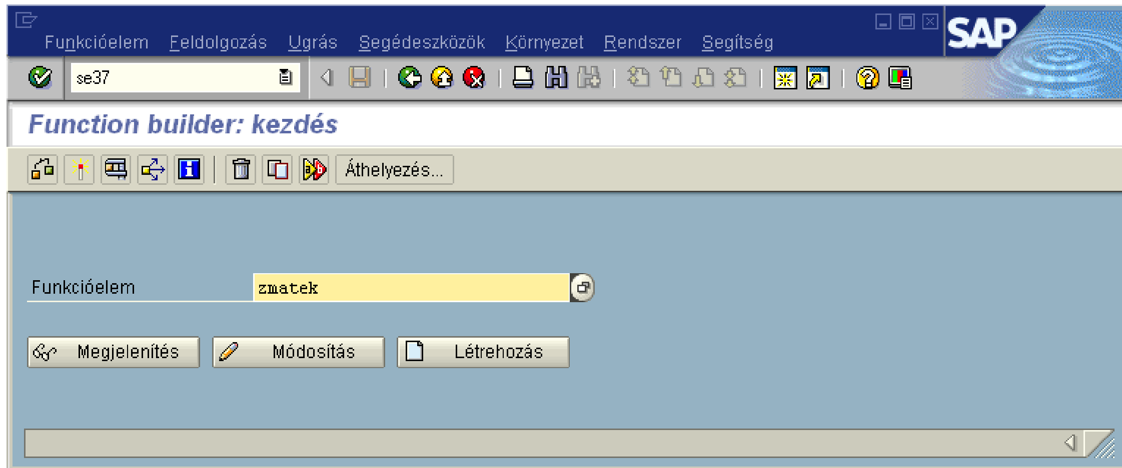
Az új szolgáltatás fejlesztése természetesen annak specifikálásával és felületének (interface-ének) megtervezésével indul. Az SAP rendszerek általában tartalmaznak egy DEV (fejlesztői), QA (minőségbiztosító) és PROD (éles) környezetet, fejleszteni mindig a DEV-ben szokás. Nevezzük ezt az SAP példányt DEV010-nek, minden további feladatot ebben végzünk SAP oldalon. A QA és PROD rendszerekre majd automatikus transzporttal telepítődik a szolgáltatás igénybevételi lehetősége.

A szolgáltatás felületét az SAP-ban tervezzük meg, ehhez készítünk egy új „Function modul”-t. A szolgáltatás üzleti specifikációja a lehető legegyszerűbb lesz: 2 input egész számra visszaadja azok összegét. A szolgáltatás SAP oldali neve legyen *ZMATEK*!

Egy új Bapi függvény írása úgy kezdődik, hogy definiálni kell a függvény szignatúráját, azaz a *nevét*, *import* (in típusú), *export* (out típusú) és *tábla* (inout típusú) paramétereit. Ezt a feladatot az SAP *se37* tranzakciójával tudjuk elvégezni. A példánkban tehát mi most egy *ZMATEK* függvényt hozunk létre. Adjuk meg a nevét és click a *Módosítás*-ra, ahogy azt a 13. ábra is mutatja.

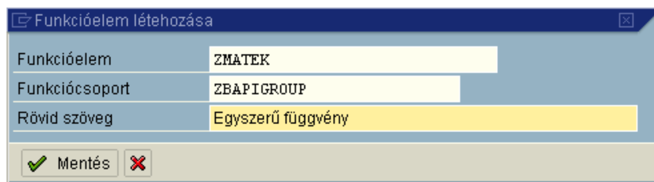
Minden függvényt egy „csomagba” kell tenni,

[3]RFC=Remote Function Call



13. ábra. Egy SAP RFC function fejlesztése (SE37 tranzakció)

amit az SAP *function group*-nak hív. Mi ezt a csomagot választottuk: *ZBAPIGROUP* (14. ábra)



14. ábra. Egy új ZMATEK nevű SAP RFC function készítése

Fontos, hogy a BAPI „Remote képes” kell legyen (15. ábra), emiatt csak érték (VALUE típusú) paraméterei lehetnek. Az EXPORT (17. ábra), IMPORT (16. ábra) és TABLE paramétereket a megfelelő füleken vehetjük fel.

Import Paraméterek: A 6. oszlopban a pipa jelzi az érték paramétereket!

Export paraméterek: A 4. oszlopban a pipa jelzi az érték paramétereket!

A 18. ábra mutatja a legenerált üres, törzsnélküli függvény szignatúrát.

A *ZMATEK* törzsét természetesen nem kell megírni, hiszen azt Java-ban a *BapiServer* fogja távolról biztosítani. Ez csupán egy helyi hívási felület (stub), amit például egy *ABAP* report-ból lehet majd használni.

3.2. A JAVA szerver megtervezése és létrehozása

A *BapiServer* implementációja az SAP által biztosított Java könyvtárra (*JCo*) épül, amit egy SAP-os customer kód birtokában mindenki letölthet.

Próbáljuk meg előbb azt megérteni, hogy mi történik, amikor egy ABAP program egy távoli függvényt hív, esetünkben például így:

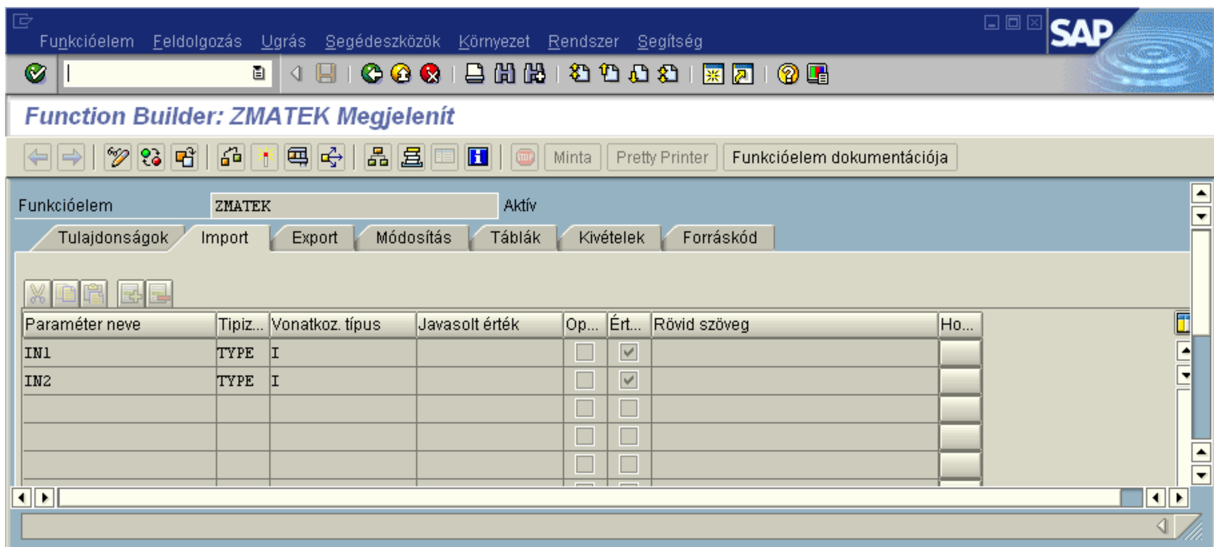
```
CALL FUNCTION 'ZMATEK' DESTINATION 'V_COTAS020ACK'
EXPORTING IN1 = I1 IN2 = I2 IMPORTING OUT = O1.
```

A meghívott függvény neve *ZMATEK*, a paraméterei pedig: *IN1*, *IN2* és *OUT*, ahogy azt már tudjuk. A hívásban az egyenlőségek csak a konkrét paraméterek átadásának a szintaxisa. A *DESTINATION* az RFC protokoll sajátja, 3 dolgot fog össze 1 név alatt, amit a SAP admin felületén kell bekonfigurálni: az SAP gateway host nevét, a gateway TCP portját és egy un. program ID-ét. A mi szerverünk is ilyen paraméterekkel fogja beregisztrálni magát az SAP infrastruktúrába, ugyanakkor a hívás is ezen az úton oldódik majd fel. Ezt nevezzük a hívás fizikai, adatkapcsolati szintjének.

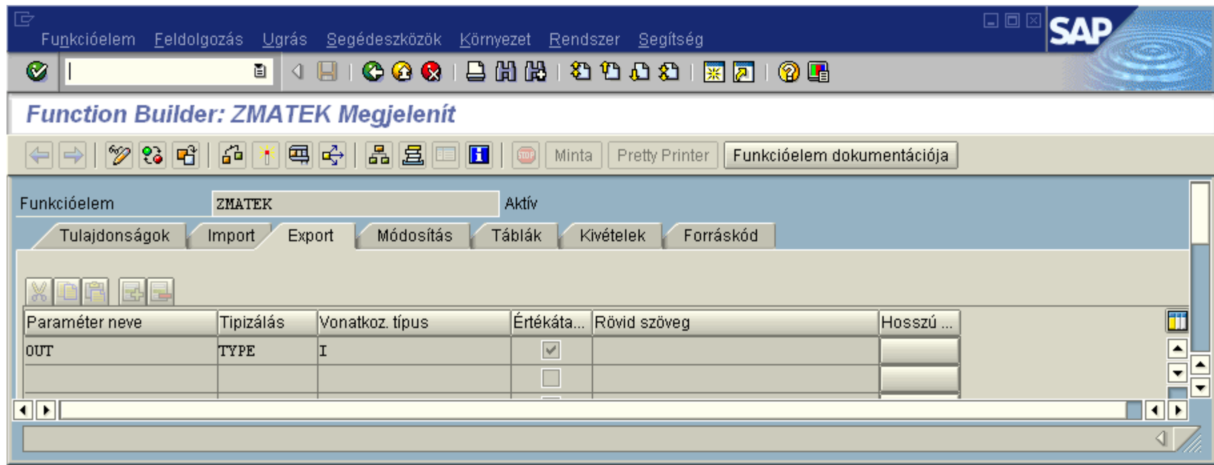
Az igazán érdekes az, hogy amikor egy

Funkcióelem Feldolgozás Ugrás Segédeszközök Környezet Rendszer Segítség
SAP
Function Builder: ZMATEK Megjelenít
Minta Pretty Printer Funkcióelem dokumentációja
Funkcióelem ZMATEK Aktív
Tulajdonságok Import Export Módosítás Táblák Kivételek Forráskód
Osztályozás
Funkciócsoport ZBAPIGROUP TesztBAPI
Rövid szöveg The ZMATEK local stub
Lefutási mód
Normál fukcióelem
Remote-képes elem
AB-módosítási elem
Azonnali indít.
Azonnali ind.-nincs utólagos ABA
Késlelt.indítás
Gyűjtő futás
RFC-célgép
Általános adatok
Felelős I09328038
Utolsó módosító I09531881
Módosítási dátum 2008.05.31
Fejlesztési osztály $TMP
Programnév SAPLZBAPIGROUP
Include neve LZBAPIGROUPU02
Eredeti nyelv HU
Nincs engedélyezve
Szerkeszt. zárlat
Globális

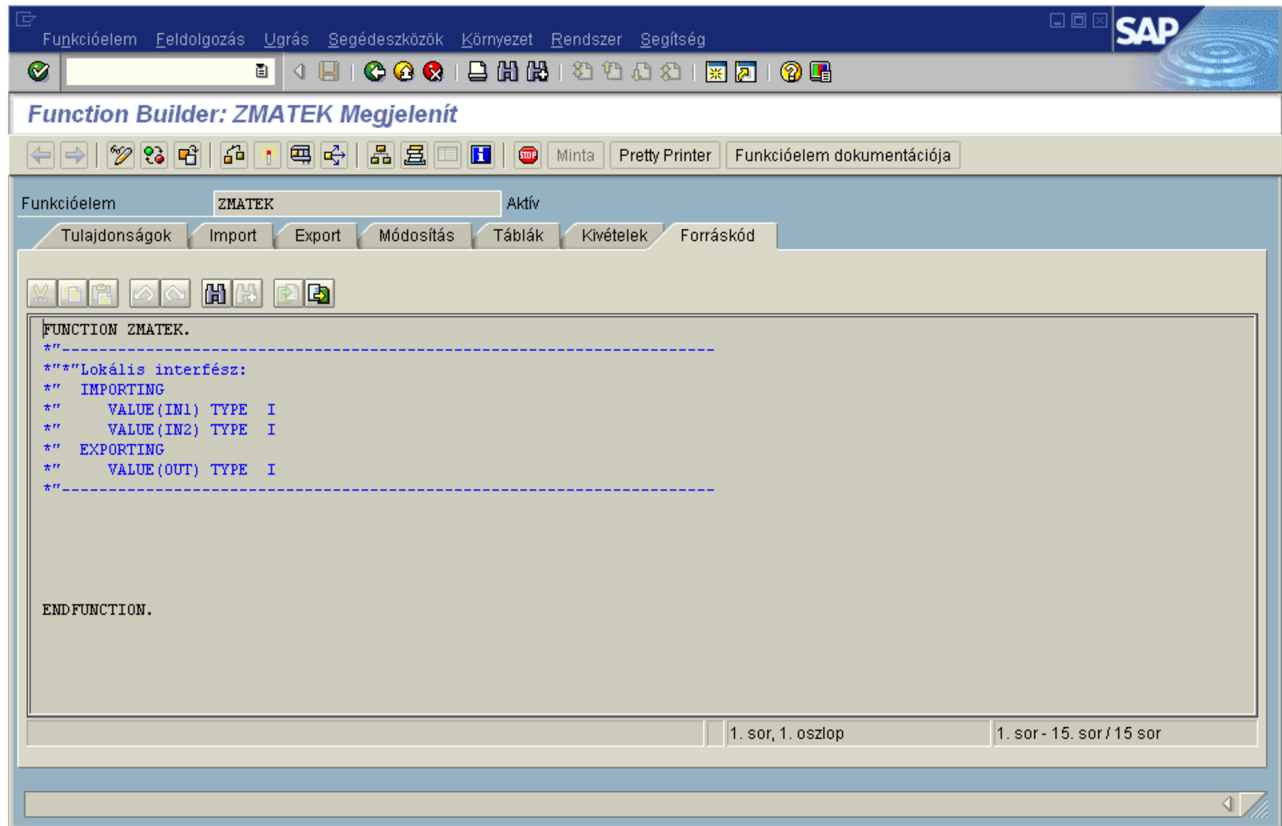
15. ábra. A ZMATEK function konfigurálása - alaptulajdonságok



16. ábra. A ZMATEK function konfigurálása - input (import) paraméterek



17. ábra. A ZMATEK function konfigurálása - output (export) paraméterek



18. ábra. A ZMATEK function konfigurálása - A generált forráskód

```
1  // 17. programlista: A Bapi függvények ősosztálya
2
3  package hu.mol.bapiserver;
4
5  import com.sap.mw.jco.JCO;
6
7  public abstract class BapiFunction
8  {
9      protected JCO.ParameterList input  = null;
10     protected JCO.ParameterList output = null;
11     protected JCO.ParameterList tables = null;
12
13     public void bapiService( JCO.Function function )
14     {
15         input  = function.getImportParameterList();
16             output = function.getExportParameterList();
17             tables = function.getTableParameterList();
18         return;
19     }
20 } // end class
```

adatkapcsolati szintű konfiguráció (ezt röviden CHANNEL-nek nevezzük) lehetővé teszi a hívásokat – ami mögött egy beregisztrált szolgáltatás (más néven szerver) hallgatózik –, milyen adatok jönnek a hívás kérésében és a válaszban, azaz a távoli hívásnak milyen az adatcsere protokollja. A kérés (*RFC request*) során a függvény neve (*ZMATEK*), az *in* típusú (nevezik még import paraméternek) paramétereinek, és az *inout* típusú table paramétereinek láncolt listája megy a hívást kiszolgáló szerver felé. A szerver – esetünkben majd a BapiServer – átveszi ezeket az információkat és kiszámítja a választ (*RFC response*) visszaküldve az *out* típusú (nevezik még export paraméternek), és az *inout* típusú (azaz 2 irányú kommunikációra képes) table paramétereinek láncolt listáját. Ezek miatt az adottságok miatt a 17. programlistán szereplő *BapiFunction* class egy olyan ősosztály, ami egy bapi-t reprezentál a megoldásunkban. Beépítve tartalmazza az említett 3 paraméterlistát, kifejezetten azért, hogy egy új függvény implementációja ezeket majd használja. A *bapiService()* metódus azt az egyetlen értelmesen megfogalmazható dolgot csinálja, hogy az infrastruktúrától megkapott bapi function paraméter alapján elmenti a paramétereket egy egy-egy lokális listába. A *BapiFunction* class-nak azonban ennél sokkal hasznosabb az a feladata, hogy ő az ősosztálya minden konkrét bapi szerviz függvény megvalósításnak, ugyanis a példamegoldásunk filozófiája az, hogy minden egyes ABAP-ból meghívható függvényt a *BapiServer* oldalon egy *BapiFunction* utódosztály reprezentál, használva közben a gyártómetódus tervezési mintát.

Elérkeztünk oda, hogy implementálhatjuk a *ZMATEK* szolgáltatásunkat, ahogy azt a 18. programlista mutatja. Mivel ez a szolgáltatás – a fentiek szerint – egyben egy *BapiFunction*, ezért ez lesz az ősosztály. Fontos névkonvenció a keretrendszerünkben, hogy a megvalósított class neve mindig az ABAP oldalon megadott csonk függvény neve legyen. Mivel ez a fentiek alapján a *ZMATEK*, így az osztályunknak is ez lett a neve. Ez a megegyezés a *factory method design pattern*-hez kell majd. A szolgáltatás feladata és implementációja a lehető legegyszerűbb. Meghívja az ősosztály *bapiService()* metódusát, beállítva ezzel az örökölt *input*, *output* és *tables* paraméterlisták helyes belső hivatkozásait a paraméterekre. Ezután a

String i1 = input.getInt("IN1") és *int i2 = input.getInt("IN2")* sorok lekérik a SAP oldal által elküldött 2 input paramétert, amiknek emlékezzünk rá *IN1* és *IN2* volt a neve. Az összeadás után a *output.setValue("OUT", o)* sor gondoskodik a hívó ABAP program felé response tartalomról, amit az OUT változó beállításával tesz meg. A többi sor csak a program működésének konzolon való követését szolgálja majd (a szolgáltatásunk minden hívását ezzel monitorozhatjuk).

```
// 18. programlista: A ZMATEK függvény, mint szolgáltatás implementáció

package hu.mol.bapiserver.functions;

import com.sap.mw.jco.JCO;
import hu.mol.bapiserver.BapiFunction;

public class ZMATEK extends BapiFunction
{
    public void bapiService( JCO.Function function )
    {
        super.bapiService( function );
        System.out.println( "F= " + function.getName()  );
        String i1 = input.getInt("IN1");
        System.out.println( i1 );
        int i2 =  input.getInt("IN2");
        System.out.println( i2 );
        int o = i1 + i2;
        output.setValue("OUT", o);
        return;
    }
} // end class
```

Utolsó lépésben magát a szerver környezetet hozzuk létre, itt fog összeállni a teljes kép. A 19. programlista mutatja, hogy egy *JCO.Server* típusú *JCOBapiServer* class-t készítünk a megvalósított *BapiFunction* osztályok közös konténereként. A *JCOBapiServer* konstruktor a már említett SAP gateway beregisztrálást csinálja, ehhez kapja meg a szükséges paramétereket. Ezt a *main()* metódusban indítjuk el, ugyanis – ahogy látható is – ott hozunk létre egy *server* nevű, a *BapiServer*-t reprezentáló változót. A *main()* metódus *server.start()* hívása egy eseménykezelő ciklusba teszi a szerver alkalmazásunkat, ahonnak a *server.stop()* billentené ki, de ez most nincs implementálva még, így ezt a szervert a kill paranccsal kell leállítani. Programozói szempontból a legfontosabb az örökölt *handleRequest(JCO.Function function)* callback metódus, amit felül kell írni az igényeink szerint (egy eseménykezelő rendszerre ez is jó példa lenne a 2. cikkhez). Amikor egy ABAP program a mi CHANNEL-ünkön keresztül hív egy távoli függvényt, akkor ebben az infrastruktúrában a fenti programszervezés mindig ennek a metódusnak (ami ingyen megkapja a *function* paramétert) a visszahívását fogja eredményezni. A kód többi része már triviális. A kapott *JCO.Function* objektumtól megkérdezzük a bapi nevét (a ZMATEK hívásakor ez ZMATEK lesz), ez alapján megkonstruáljuk a megfelelő *BapiFunction* utódosztály teljes nevét, majd csinálunk ebből egy *BapiFunction* objektumot (itt van a gyártómetódus), aminek a *bapiService()* metódusát kényelmesen meghívjuk. Fontos észrevenni, hogy ezzel a keretmódszerrel akárhány szervizt beiktathatunk a szerverünkbe, azaz könnyen és rugalmasan bővíthető.

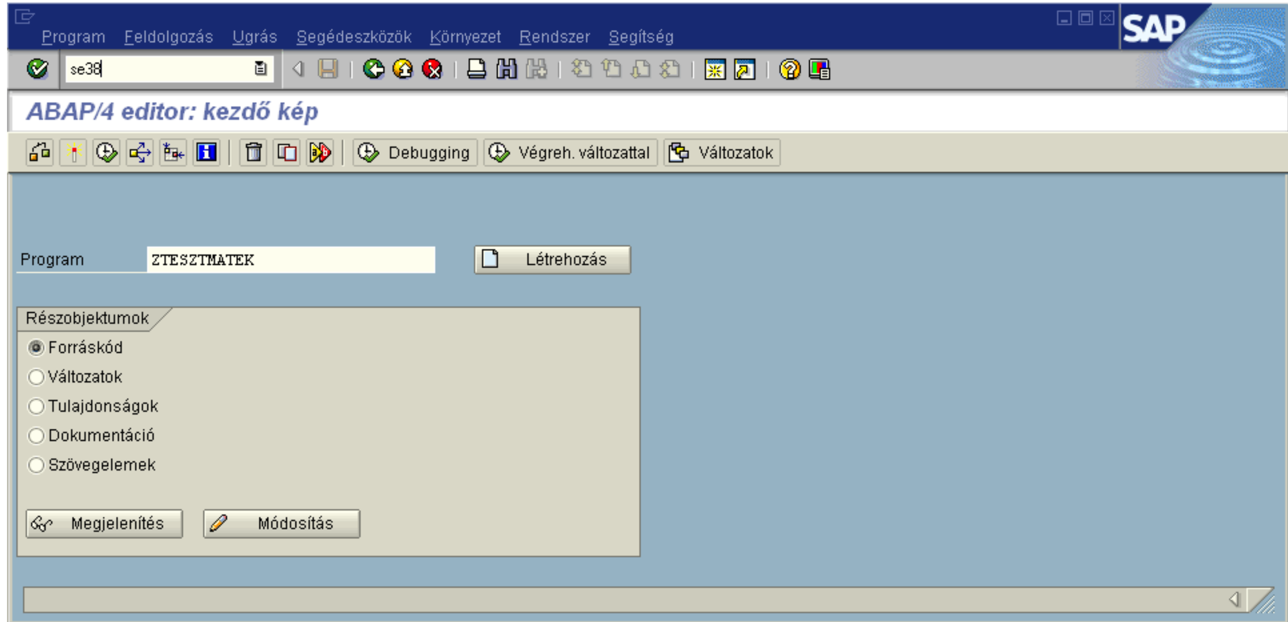
```
// 19. programlista: A BapiServer

package hu.mol.bapiserver;

import com.sap.mw.jco.*;

public class JCOBapiServer extends JCO.Server
{
    //

    public JCOBapiServer(String gwhost, // gateway host
            String gwserv, // gateway service number
            String progid, // program ID
            IRepository repository)  // Az én kis repository-m

    {
        super(gwhost, gwserv, progid, repository);
    }

    //
    protected void handleRequest(JCO.Function function)
    {
        BapiFunction bapiFunction;
        String functionName = "hu.mol.bapiserver.functions." + function.getName();
        System.out.println("Service Class = " + functionName );
        try
        {
            System.out.println( function.getName() );
           bapiFunction = (BapiFunction) Class.forName( functionName ).newInstance();
            //bapiFunction = new hu.mol.bapiserver.functions.ZMATEK();
            bapiFunction.bapiService( function );
        } catch (Exception e)
        {
             System.out.println( e.toString() );
        }
    }

    // Start server
    public static void main(String[] args)
    {
        System.out.println("Start BapiServer...");

        JCOBapiServer server = new JCOBapiServer("molsapmod",
          "3303", "V_COTAS020ACK", BapiServerRepository.getRepo() );

        server.start();
        //System.out.println("Stop...");
        //server.stop(); // egy másik szálról jöhet!
    }
} // end class
```

Még adósak vagyunk a 20. programlista tartalmával, ami csak egy Registry-t kér le a SAP-tól. Ez a registry a gateway-be való beregisztráláskor érdekes, de a háttérben folyamatosan támogatja a szerver munkáját, különben honnan is tudná a kommunikációban résztvevő adatszerkezeteket és neveket. Programozási szempontból nem túl érdekes, ez csupán egy kötelező kellék az RFC protokoll használatához.

```
// 20. programlista: Kell egy Repository

package hu.mol.bapiserver;

import com.sap.mw.jco.IRepository;
import com.sap.mw.jco.JCO;

public class BapiServerRepository
{

    public static IRepository repository;
    public static final String MYREPO = "BapiServerRepo";

    public static JCO.Client aClient = null;

    public static IRepository getRepo()
    {
        if (JCO.PoolManager.singleton().getPool("MOD") == null)
        {
            JCO.addClientPool("MOD", // Alias for this pool
                    10, // Max. number of connections
                    "020", // SAP client
                    "moleai", // userid
                    "xxxxxxx", // password
                    "EN", // language
                    "molsapmod.mol.hu", // host name
                    "03");                                  // system number
        }


        aClient = JCO.getClient("MOD");
        repository = JCO.createRepository(MYREPO, aClient);



        return repository;
    }

    public static IRepository getRepository()
    {
        return repository;
    }
} // end class
```

3.3. A JAVA szolgáltatás meghívásának tesztelése

A 3.1 pontban elkészítettük a szolgáltatás helyi, SAP-on belüli hívási felületét, míg a 3.2 pontban bemutattuk annak Java implementációját. Ezzel az SAP rendszer rendelkezésére áll az új, esetünkben *ZMATEK* szolgáltatás. Próbáljuk ki! Ehhez az *se38* tranzakció használatával írjunk egy programot (reportot), ami leteszteli, használja a külső Java függvényünket. Legyen a report neve: *ZMATEKTESZT* (19. ábra).



19. ábra. Egy ABAP program (report) a ZMATEK teszteléséhez - Létrehozás

A teszt ABAP program a következő, azaz ezt a programot gépeljük be a 20. ábrán látható módon!

```
1 REPORT ZTESZTMATEK                                  .
2 DATA: I1 TYPE I. DATA: I2 TYPE I. DATA: O1 TYPE I.
3 I1 = 10. I2 = 3.
4 CALL FUNCTION 'ZMATEK' DESTINATION 'V_COTAS020ACK' EXPORTING IN1 = I1 IN2 = I2 IMPORTING OUT = O1.
5 WRITE 'Eredmény: '. WRITE O1.
```

Egy program a *se80* vagy *se38* tranzakcióval szerkeszthető, *F8*-cal futtatható. A program módosítása esetén a „floppyval” kell lementeni, 2-vel aktiválni, 3-mal futtatni (21. ábra).

A program futása eredményeképpen kiírja a SAP GUI felületén az „Eredmény: 13”-at, ugyanis most 10 és 3 volt a megadott input.

Program Feldolgozás Ugrás Segédeszközök Környezet Rendszer Segítség

ABAP Editor: Riport ZTESZTMATEK megjelenítés

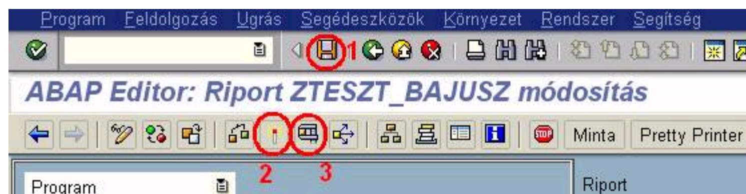
Minta Pretty Printer

Riport ZTESZTMATEK Aktív

```
*&---------------------------------------------------------------------*
*& Report  ZTESZTMATEK                                                 *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&Report: RFC teszt. 2008.05.27                                        *
*&---------------------------------------------------------------------*

REPORT  ZTESZTMATEK                    .

DATA:  I1 TYPE I.
DATA:  I2 TYPE I.
DATA:  O1 TYPE I.

I1 = 10.
I2 = 3.

CALL FUNCTION 'ZMATEK' DESTINATION 'V_COTAS020ACK'
EXPORTING
 IN1 = I1
 IN2 = I2
IMPORTING
 OUT = O1.

WRITE 'Eredmény: '.
WRITE O1.
```

1. sor, 1. oszlop 1. sor - 26. sor / 26 sor

20. ábra. Egy ABAP program (report) a ZMATEK teszteléséhez - A forráskód



21. ábra. Egy ABAP program (report) a ZMATEK teszteléséhez - Futtatás

4. Informatikai szilánkok - tippek és trükkök

Az *Informatikai Navigátor* állandó rovata lesz a szilánkok, aminek célja az, hogy minden érdekes, apró vagy elgondolkodtató dolgot feljegyezzen, leírjon. A mindennapok során sűrűn találkozunk egy-egy érdekes, ötletes megoldással, amiket a továbbiakban igyekszünk ebben a rovatban mindenki számára közkinccsé tenni. Minden kedves olvasó saját érdekes feljegyzését is várjuk abban a reményben, hogy egyszer a Te írásod is megjelenik itt. Az írásokat erre az e-mail címre várjuk: *creedsoft.org@gmail.com*.

4.1. LDAP Query parancssorból

Adott egy címtár, esetünkben egy MS AD. A feladat az, hogy hozzájussunk a címtár egyes bejegyzéseihez és azokat egy egyszerű textfile-ba mentsük. Létezik egy LDAP utils nevű szoftver, amiben – többek között – van egy *ldapsearch* parancs is.

ldapsearch -Hldap://myADHost -b dc=mol,dc=sys,dc=corp -s sub -x -W -D inyiri@mol.hu (mail=inyiri*)

Ez a parancs lekéri az *AD* címtárból annak a bejegyzésnek az adatait, akinek az e-mail címe *inyiri* szóval kezdődik. Az eredményformátum egy un. *LDIF* formátum, amit az LDAP csomag egy másik parancsával lehet egy – esetleg másik – címtárba beszúrni. Az output néhány első és közbülső sora így néz ki (21. programlista), a többit lehagytuk a nagy méret miatt:

```
# 21. programlista
#
# extended LDIF
#
# LDAPv3
# base <dc=mol,dc=sys,dc=corp> with scope subtree
# filter: (mail=inyiri*)
# requesting: ALL
#

# Nyiri Imre, MOLUsers, mol.sys.corp
dn: CN=Nyiri Imre,OU=MOLUsers,DC=mol,DC=sys,DC=corp
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: user
cn: Nyiri Imre
sn: Nyiri
l: Budapest
title:: QWxrYWxtYXrDoXMgSW50ZWdyw6FjacOzIGZlamwuIHN6YWvDqXJ0xZE=
description:: QWxrYWxtYXrDoXMgSW50ZWdyw6FjacOzIEZlamxlc3p0w6lz
postalCode: 1117
physicalDeliveryOfficeName:: RmVow6lyaMOheiBJLiBlbS4gMTA2IHN6b2Jh
telephoneNumber: 20220
facsimileTelephoneNumber:: IA==
userCertificate:: MIIEDDCCA3WgAwIBAgIBADANBgkqhkiG9w0BAQQFADCBuzELMAkGA1UEBhMC
 LS0xEjAQBgNVBAgTCVNvbWVTdGF0ZTERMA8GA1UEBxMIU29tZUNpdHkxGTAXBgNVBAoTEFNvbWVPc
 mdhbml6YXRpb24xHzAdBgNVBAsTFlNvbWVPcmdhbml6YXRpb25hbFVuaXQxHjAcBgNVBAMTFWxvY2
 FsaG9zdC5sb2NhbGRvbWFpbjEpMCcGCSqGSIb3DQEJARYacm9vdEBsb2NhbGhvc3QubG9jYWxkb21
 haW4wHhcNMDYwMTAzMTQzNTQ0WhcNMDcwMTAzMTQzNTQ0WjCBuzELMAkGA1UEBhMCLS0xEjAQBgNV
 BAgTCVNvbWVTdGF0ZTERMA8GA1UEBxMIU29tZUNpdHkxGTAXBgNVBAoTEFNvbWVPcmdhbml6YXRpb
...
givenName: Imre
distinguishedName: CN=Nyiri Imre,OU=MOLUsers,DC=mol,DC=sys,DC=corp
instanceType: 4
```

```
whenCreated: 20050602155836.0Z
whenChanged: 20090924081347.0Z
displayName: Nyiri Imre
otherTelephone: 20-220
otherTelephone:: IA==
uSNCreated: 11663017
memberOf: CN=INA Study MOL TLs,OU=Distribution,OU=MOLGroups,DC=mol,DC=sys,DC=c
 orp
memberOf: CN=FCBC Project Leaders,OU=Distribution,OU=MOLGroups,DC=mol,DC=sys,D
 C=corp
...
```

4.2. File elmentése böngészőből HTTP-én keresztül

Több alkalommal előfordul, hogy egy Java servlet legyárt egy eredményt, de azt nem akarjuk a böngészőben megjeleníteni, helyette inkább az operációs rendszer adjon egy ablakot a böngészőn keresztül, hogy a servlet által visszaküldött tartalmat – ami lehet excel, text vagy bármilyen file – el lehessen menteni a lokális filerendszerbe. A lenti kódrészlet pont ennek a feladatnak adja meg a lehetséges megoldását. A lementendő tartalom típusa és a 2 header érték beállítása fontos!

```
// 22. programlista: File mentése böngészőből (Zilahy Zoltán megoldása)

getResponse().setContentType("text/plain");
getResponse().setHeader("Cache-Control","max-age=0"); getResponse().setHeader("Content-Disposition",
                              "attachment;filename=\"order.csv\";");
OutputStreamWriter w = new OutputStreamWriter(getResponse().getOutputStream(), "ISO-8859-2");
w.write(sb.toString());
w.flush();
w.close();
```

4.3. JavaScript class

Az AJAX programok miatt a Javascript a reneszánszát éli. Azok a programozók, akik valamilyen „tisztességes” OO nyelven szocializálódtak, sokszor fintorogva nézik a Javascript nyelvet (én is). Pedig ott is lehet szépen osztályokat és objektumokat csinálni! A kedvenc kódmintámat a 23. programlista mutatja:

```
// 23. programlista: Új class Javascriptben
function MyClass(kp1, kp2)
{
    this.kp1 = kp1;
    this.kp2 = kp2;
}

MyClass.prototype =
{
        mezo1 : 5,
        mezo2: "alma",
        mezo3: 0,

        metodus1: function(p1, p2, ..., pn)
        {

        },
        metodus2: function(p1, p2, ..., pn)
        {

        },
        metodus3: function(p1, p2, ..., pn)
        {

                return x;
        }

} // end class
```

Ezzel létrehoztunk egy *MyClass* nevű osztályt, aminek példányosítását és használatát a 24. programlista mutatja.

```
// 24. programlista
var obj = new MyClass("Imre", 12);
obj.metodus2(...);
```

4.4. Struktogramok készítése LaTeXsegítségével

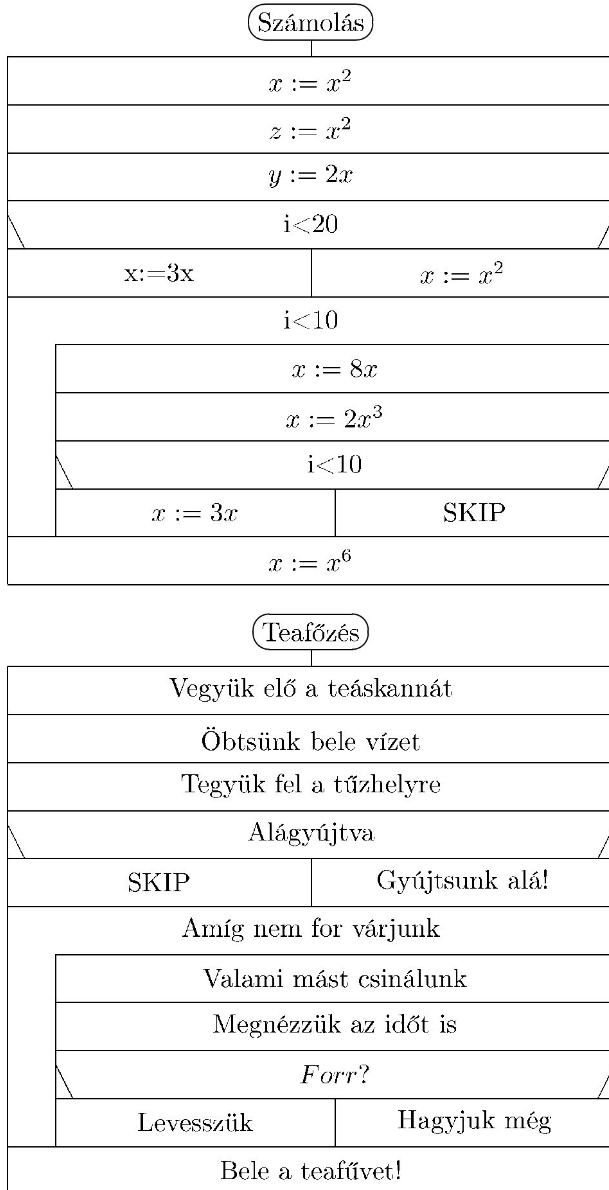
Emlékszik még valaki a struktogramokra? Amikor a strukturált programozást tanultuk, akkor egy alapvető ábrázolási eszköze volt az algoritmusoknak. Bizonyos célokra még ma is jól jöhet, nem is biztos, hogy csak a programok megtervezéséhez.

Itt csak az érdeklődést szeretnénk felkelteni egy rövid példával, a részleteket itt tanulhatjuk meg, illetve innen tölthető le a stuky.sty stíluslap:`http://lorentey.hu/project/stuki.html.hu`

Aki ismeri a LaTeX környezetet, annak a lenti listát egyszerű megérteni, használni kell a dokumentum preambulumban a *\usepackage{stuki}* deklarációt. Aki megnézi ezt a kódot és látja a 22. ábrán az eredményt, minden magyarázat nélkül megértheti a példánkat.

```
%% LyX 1.6.2 created this file.  For more info, see http://www.lyx.org/.
%% Do not edit unless you really know what you are doing.
\documentclass[magyar]{article}
\usepackage[T1]{fontenc}
\usepackage[latin2]{inputenc}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% User specified LaTeX commands.
\usepackage{stuki}

\usepackage{babel}

\begin{document}

\begin{stuki*}[8cm]{Számolás}

  \stm*{ $x:=x^2$ }
  \stm*{ $z:=x^2$ }
  \stm*{ $y:=2x$ }

  \begin{IF}{1}{\stm{$i<20$}}
    \stm{ $x:=3x$ }
  \ELSE
    \stm*{ $x:=x^2$ }
  \end{IF}

  \begin{WHILE}{4}{\stm{$i<10$}}
    \stm*{ $x:=8x$ }
    \stm*{ $x:=2x^3$ }
  \begin{IF}{1}{\stm{$i<10$}}
    \stm*{ $x:=3x$ }
  \ELSE
    \stm*{ SKIP }
```

```
  \end{IF}


  \end{WHILE}

  \stm*{ $x:=x^6$ }
\end{stuki*}

\begin{stuki*}[8cm]{Teafőzés}

  \stm*{ Vegyük elő a teáskannát }
  \stm*{ Öbtsünk bele vízet }
  \stm*{ Tegyük fel a tűzhelyre }

  \begin{IF}{1}{\stm*{Alágyújtva}}
    \stm*{ SKIP }
  \ELSE
    \stm*{ Gyújtsunk alá! }
  \end{IF}

  \begin{WHILE}{4}{\stm*{Amíg nem for várjunk}}
    \stm*{ Valami mást csinálunk }
    \stm*{ Megnézzük az időt is }
  \begin{IF}{1}{\stm{Forr?}}
    \stm*{ Levesszük }
  \ELSE
    \stm*{ Hagyjuk még }
  \end{IF}


  \end{WHILE}

  \stm*{ Bele a teafűvet! }
\end{stuki*}

\end{document}
```



22. ábra. Struktogram készítés

A fenti LaTeXkódot most png képfile-ra fordítjuk a következő paranccsal, aminek az eredményét a 22. ábra mutatja: *latex2png -d 300 mystuki.tex*. A *-d* paraméter a legyártott eredmény felbontását adja meg.

A *stuki* csomag arra is példát mutat (Lőrentey Károly munkája), hogy LaTeXsegítségével milyen kiváló dolgok készíthetőek. A strukturált programok 3 programkonstrukcióból építkeznek: szekvencia, elágazás és ciklus. Ennek megfelelően a stuki renderelő környezet használja a stuki csomagban implementált az IF és WHILE környezeteket. Néhány szó a struktogramról: Az egymás után végrehajtandó tevékenységeket egy oldallal érintkező, egymás alatti téglalapokba írjuk, ez a szekvencia. A tevékenységeket felülről lefelé, egymás után kell végrehajtani: T1, T2, ..., Tn. A szelekció valamilyen feltételtől függő tevékenységvégrehajtást jelent. A ciklusnál egy belépési feltételtől függően ismétlünk meg egy tevékenységet, vagy tevékenységsorozatot. Az ismétlésre kerülő tevékenységek alkotják a ciklus magját. Az első példánkban a $ jelek közötti kifejezés a matematikai módot jelenti, ha megfigyeljük ettől lesz olyan stílusos a képlet. Ahhoz hogy az így szedett dokumentumokat lefordíthassuk valamilyen formátumra (pdf, png, stb.) használni kell a *stuky.sty* stíluslapot, amit innen lehet letölteni: `http://lorentey.hu/downloads/stuki/stuki.sty`. Más feltétel nem szükséges egy struktogram elkészítéséhez. A felhasználói dokumentáció innen elérhető: `http://lorentey.hu/downloads/stuki/stuki.ps.gz`

5. Elemzői sarok

Ebben a számban az elemzői sarokban 2 témáról írunk. Az első a szerző egyik projectje során felmerült clearing témakörben született gondolatait foglalja össze. A másik – MOS-ról szóló – írás pedig *Orbán Gábor* tollából származik, amit ehelyütt is szeretnénk megköszönni neki.

5.1. A clearing fogalma

Kártya kibocsátó = card issuer
Kártya elfogadó = card acceptor
$Issuer = \{hu, at, ro, slo, srb, tifon, \ldots\}$
$Acceptor = \{hu, at, ro, slo, srb, tifon, \ldots\}$
$Vevők = \{v_1, v_2, \ldots, v_n\}$

Az Issuer egy tetszőleges elemét I, az Acceptorét A , a vevőét pedig V jelöli. Ennek megfelelően egy kártyás tranzakció indexelése: $T_{I,A,V}$ azaz, ebben a tranzakcióban (kártyás vásárlás a kúton) a V vevő az I kibocsátó kártyájával fizetett az A kártya-elfogadó helyen.

Példa egy konkrét üzleti eseményre: T_{hu,at,v_8}, azaz a *hu* kibocsátó kártyájával vásárolt a v_8 vevő az *at* elfogadó helyen. Mit jelent ez? Két dolgot:

1. A v_8 vevő tartozik aT_{hu,at,v_8}összegével (jele a továbbiakban: MT_{hu,at,v_8}) *hu* felé.

2. A *hu* tartozik MT_{hu,at,v_8} összeggel *at* felé.

Ez azt jelenti, hogy egy tartozás jön létre 2 vállalat között abban az esetben, amikor az $I \neq A$. Egy adott időszakra (1 nap, 1 hét, 1 hónap, ?), idő-sorosan vezethetőek ezek a tartozások, amiket az időszak végén egymásnak ki kell fizetni. A clearing lényege, hogy nem a teljes összeget fizetik egymásnak a felek, hanem csak a *settlement*-et, azaz az egymással szemben elszámolt összegek egyenlegét. Az egész rendszer a készpénz mozgások csökkentésére, azaz a cash flow javítására lett kitalálva.

Mi történik, ha több felet is bevonunk ebbe a *clearing* rendszerbe? Ekkor egy A acceptornak több $I_1, I_2, \ldots$ issuer fog tartozni az adott időszakra nézve (persze itt az az eset, amikor $I = A$ nem jelent tartozás növekedést). Ugyanakkor ez az A acceptor issuer szerepkörben több helyre is tartozni fog. A clearing műveletet lehet páronként is végezni, azonban sokkal nagyobb a pénzkimélés, amennyiben több (lehetőleg mindenki) szereplő részt vesz benne. Egy példa, ahol ezek a tranzakciók vannak:

1. $MT_{hu,at,v_8} = 1000\, pénz$

2. $MT_{at,ro,v_8} = 500\, pénz$

3. $MT_{hu,ro,v_8} = 100\, pénz$

4. $MT_{at,hu,v_8} = 900\, pénz$

5. $MT_{ro,at,v_8} = 200\, pénz$

Követelések:

1. Ekkor az *at* felé való össztartozás (ahol Ő az acceptor): $1000 + 200 = 1200$

2. A *ro* felé: $500 + 100 = 600$

3. A *hu* felé: 900

Tartozások:

1. Az *at*-tól várt pénz (ahol Ő az issuer): $500 + 900 = 1400$

2. A *ro*-tól: 200

3. A *hu*-tól: $1000 + 100 = 1100$

A Clearing House-ba való befizetések a tartozások alapján: $1400 + 200 + 1100 = 2700$

A Clearing House-tól kapott pénzek: $1200 + 600 + 900 = 2700$

Ezek azonban csak virtuális fizetések, valójában csak a Clearing House clearing feladatai és nem kerülnek befizetésre. Valójában ez történik. Vesszük az at-ét: Követel →1200, Tartozik →1400, azaz az at egyenlege: 200 befizetés a Clearing House-ba.

Nézzük meg mindhárom felet, egy szép táblázatba rendezve:

Ügyfél	Tartozik	Követel	Egyenleg (pénzmozgás)
at	1400	1200	-200
ro	200	600	+400
hu	1100	900	-200

Érdekességként nézzük meg, hogy a páronként való pénzkimélő clearing-nek mekkora a hatékonysága, például a hu és at viszonylatban:

1. *at* követel *hu*-tól: 1000,
2. *hu* követel *at*-tól: 900. Egyenleg: *at* kap 100-at *hu*-tól.

És hu, ro peer to peer viszonyban:

1. *ro* követel *hu*-tól: 100
2. *hu* követel *ro*-tól: 0. Egyenleg: *ro* kap 100-at *hu*-tól.

Itt most hu, at, ro felek vettek rész a kártyás fizetésben, az (hu, ro), (hu, at), (ro, at) peer to peer elszámolás lehet, nézzük a 3. (at, ro) esetet is:

1. *ro* követel *at*-tól: 500
2. *at* követel *ro*-tól: 200. Egyenleg: *ro* kap 300-at *at*-tól.

A fentiekből látszik, hogy csak az utolsó esetben at-ank 300 egység készpénzt kell fizetnie, ugyanakkor a nagy közös elszámolás szerint csak 200 pénzegységet. Itt látszik, hogy bár a clearing nélküli elszámolásoknál a P2P is kedvezőbb (kevesebb) pénzmozgást jelent, ugyanakkor a Clearing kör több résztvevős kiterjesztése egyre hatékonyabbnak tűnik a cash flow kímélés szempontjából.

A fenti példában a Clearing House-ba hu és at a befizető, mindketten 200 – 200 pénzt, ro pedig a kivételező: 400 pénzzel.

Megjegyzés: A fentiek nem igényli, hogy az összes fél, azaz hu, ro, at, slo, srb, ... részt vegyen a clearing folyamatban (azaz tagja legyen a Clearing House-nak), azonban ezzel csökken a pénzkimélés hatékonysága.

Érdekességként, az Internet-es lexikonból a clearing-ről:. (ejtsd: kliring hauz). A C. bankárok egyesülete azon célból, hogy a köztük fennálló követelések és tartozások bizonyos napokon esetleg naponként a készpénzfizetés kerülésével kiegyenlíttessenek. Állítólag ez intézmény első eszméje londoni kereskedősegédektől származott, kik főnöküktől számlatartozások behajtásával voltak megbízva és a sok szaladgálás kikerülésére egymásnak a kölcsönös számlákat megmutatták és kiegyenlítették. A C. intézmény magán- és társadalom-gazdaságilag nagy előnyökkel jár, amennyiben a pénzforgalmat kevés pénzzel engedi eszközöltetni, egyúttal pedig a fizetések behajtásával járó egyéb veszélyeket és kellemetlenségeket megszünteti. A C. a modern hitelforgalom betetőzésének tekinthető. A legrégibb C. a londoni, mely már 1773. létezett, de valószínűleg már régebben alapíttatott. Azonkívül vannak Manchesterben, Liverpoolban, Newcastle upon Tyneban, Edinburghben, Glasgowban, Dublinban. Még nagyobb terjedelmű mint a londoni Clearing forgalma a newyorkié. É.-Amerikában, mely 1853. alapíttatott; kivüle van az észak-amerikai köztársaságban még közel 40 hasonló intézet. Anglián és É.-Amerikán kívül a clearing-intézmény nem emelkedett hasonló jelentőségre, mivel többnyire az előzmények is hiányoznak, nevezetesen a közön-

ségnek azon szokása, hogy a bankárral állandó összeköttetésben áll és ezzel együtt a szokás, chéque-kel fizetni. Franciaországban Párizsnak van kiegyenlítő intézete (chambre de compensation), mely 1872. alapíttatott. Olaszországban a valutarendezés alkalmával a fizetési mechanizmus tökéletesítésére honosították meg a stanze di compensazione-t. Ausztriában 1872. alapították a bécsi intézetet (Saldierungsverein); hazánkban 1880. a budapestit. Ausztráliában is létezik már 1867 óta egy Clearing Melbourneban. Legkésőbben történt ez intézmény meghonosítása Németországban, hol 1883. alapíttatott az első kiegyenlítő intézet Berlinben, és rövid idővel reá Hamburgban, majnai Frankfurtban, Lipcsében, Boroszlóban, Kölnben Brémában, Stuttgartban, Drezdában.

A clearing-intézetek forgalmára vonatkozólag ide írhatjuk a következő adatokat: a londoni C. forgalma az utolsó években 140 000 000 000 koronára emelkedett; még nagyobb a newyorki intézet forgalma, mely egyes években 200 000 000 000 koronára emelkedett. A többi intézetek forgalma jóval kisebb; a párizsié 2000, a hamburgié 6500, a berlinié 3600, a frankfurtié 2700, a milanóié 1300, a bécsié 556, a budapestié 215 millió korona. A készpénzhasználatban elért megtakarítás nagyságát mutatja azon tény, hogy Londonban nagy bankházakban a készpénzzel történt fizetések az egész forgalom 1%-ára sem emelkednek. A new-yorki Clearingben a forgalom mintegy 95%-a kiegyenlítés útján eszközöltetik, mintegy 5%-ot tesz a készpénzzel és átírással eszközölt lebonyolítás. A clearingrendszer alkalmazást talált más téren is: a vasúti forgalom terén a vasutak kölcsönös követeléseinek kiegyenlítésére, a tőzsdeügyleteknél, postakarékpénztárnál, a liverpooli gyapot üzletben stb.

A C. alatt azt a helyet is értik, ahol ez intézmény létezik.

5.2. A Microsoft Office SharePoint Server 2007

Az MS Office System (MOS) honlapja: `http://office.microsoft.com/hu-hu/sharepointserver/default.aspx`

A statisztikák szerint sok vállalat alkalmazza a MOS-t, ami elsősorban egy csoportmunkát és kollaborációt támogató szoftver. A szerző vállalatánál is több mint 8000 felhasználó használja rendszeresen munkájához, a napi látogatószám pedig 3000 körül mozog.

De mire való a MOS? A MOS létrehozásának elsődleges célja a csoportszintű munka és kollaboráció támogatása volt a Microsoft SharePoint megoldásának segítségével. A központi portálon az egyes szervezetek, projektek webhelycsoportot igényelhetnek, az így létrejött weboldalakon dokumentumokat-, feladatokat-, adatokat-, információkat oszthatnak meg, fórumokban kommunikálhatnak egymással, találkozókat szervezhetnek. Nagy előnye az eszköz használatának a verziókezelés, hiszen így adott időpontban minden felhasználó a dokumentum legfrissebb állapotát látja, és egyszerre csak egyikük végezhet módosításokat, így elkerülhető a változatok keveredése. A MOS azonban nem csak dokumentumkezelésre alkalmas, az elmúlt évek jellemző alkalmazásai között találjuk a szervezeti- és projekt webhelyeket, tudásbázis és problémakezelő oldalakat, különféle munkafolyamatok webes támogatását, feladatkezelést, különböző felméréseket és kérdőíveket, valamint üzleti riport készítő alkalmazásokat.

Webhely vagy webhelycsoport? A szervezetek, projektek nyitó weboldalait nevezzük a MOS-on webhelycsoportnak, hiszen a kezdő webhely alá a webhelygazda további webhelyeket hozhat létre, így egy szervezet a saját weboldalán belül elkülönített munkafelületet biztosíthat az egyes csoportoknak. A MOS nyitóoldalán az „I want to..." menü segítségével igényelhetünk új webhelycsoportot, az alwebhelyek viszont közvetlenül létrehozhatók.

Szolgáltatások a MOS-on. Az alapszolgáltatások közé tartoznak a verziózható dokumentumtár, a feladatkezelő, a felmérések, a naptár és a fórum, hogy csak néhányat említsünk. Vállalatunknál 2009 során vezettük be az SPS 2003 helyett a MOSS 2007-et, az eredeti adatok migrálásával, illetve a funkcionalitás bővítésével. A funkciók tárháza többek között a rendszer adatvédelmi lehetőségeit kiterjesztő RMS-el (Right Management System) és a feladatkezelő funkciót csoportos feladatkiadással, valamint központi feladatkezelő oldallal erősítő TMT-vel (Task Management Template) bővült. Előbbi a dokumentumok titkosítását biztosítja, míg utóbbival hatékonyabb feladatkezelés valósítható meg. A felhasználók saját webhelyein (mysite) megjelenítjük a MOS portál bármely webhelyén létrehozott, a felhasználóhoz kapcsolódó összes feladatot, így a különböző projektekben kapott, vagy adott feladatainkat egy helyen kezelhetjük. Ezen kívül a MOS portál keretet biztosít több döntés előkészítési- és üzleti alkalmazásnak.

A SharePoint Portal Server (SPS) elnevezéssel 2000-ben találkozhattunk először, ekkor jelentette be a Microsoft az Exchange kiegészítőjeként fejlesztett dokumentumkezelő platformot. Ezzel párhuzamosan piacra került az Office 2000-hez is egy ingyenes kiegészítő, amely a SharePoint Team Services (STS) nevet kapta, és az irodai alkalmazásokhoz nyújtott webes felületű kollaborációs lehetőségeket.

2003-ban az SPS és az STS összeolvadásával jelent meg az elsősorban kollaborációs és portál alapokat nyújtó Windows SharePoint Services (WSS) és az erre épülő, ezt adminisztrációs-, testreszabási- és keresési funkciókkal kiegészítő SharePoint Portal Server 2003. A 2006 novemberében az Office 2007 részeként megjelent harmadik verzióban azonban az SPS új nevet kapott, és Microsoft Office SharePoint Server (MOSS) néven ez a rendszer képezi az alapját az új csoportmunka és kollaborációs portálunknak.

Orbán Gábor