

1	Alkalmazásfejlesztés és a C# .....	4
1.1	A forráskód feldolgozása .....	4
1.2	Fordítók .....	4
1.2.1	értelmezők (interpreter):.....	4
1.2.2	Köztes nyelvek (intermediate languages) .....	5
1.2.3	Köztes rendszerek a gyakorlatban: Java és .NET .....	5
1.3	Az Objektumorientált programozás (OOP) elve.....	7
2	A Visual Studio 2005. fejlesztőkörnyezet.....	9
2.1	A Visual Studio 2005. letöltése és telepítése .....	9
2.1.1	Letöltés .....	9
2.1.2	Telepítés .....	9
2.2	A Visual Studio 2005. fejlesztőkörnyezet használata .....	10
2.2.1	Új projekt létrehozása.....	10
2.2.2	Az űrlapszerkesztő és a toolbox .....	11
2.2.3	A Properties ablak .....	11
2.2.4	Kódszerkesztő .....	13
2.2.5	Töréspontok és nyomkövetés .....	14
2.3	Alapvető komponensek használata és speciális vezérlők .....	15
2.3.1	A gomb (Button) .....	18
2.3.2	Szövegdoz (TextBox).....	18
2.3.3	Jelölőnégyzet (CheckBox) .....	18
2.3.4	Rádiógomb (RadioButton) .....	19
2.3.5	Legördülő doboz (ComboBox) .....	19
2.3.6	Jelölőnégyzet lista .....	20
2.3.7	Listadoz (ListBox).....	20
2.3.8	Menü építés a ToolStrip komponenssel .....	21
2.3.9	Tárolók (Containers) .....	22
2.3.10	Fájlkezelő párbeszédablakok (FileOpenDialog, FileSaveDialog) .....	23
2.3.11	Időzítő (Timer) .....	24
3	Nyelvi elemek (kulcsszavak, változók, literálok, konstansok, névterek).....	26
3.1	Kulcsszavak.....	26
3.2	Változók, azonosítók .....	26
3.3	Literálok, konstansok .....	28
3.3.1	Szám literálok.....	28
3.3.2	Logikai literálok .....	29
3.3.3	Karakter, karakterlánc literálok.....	29
3.3.4	Konstansok .....	29
3.4	Névterek .....	29
3.5	A kifejezések .....	31
3.6	Műveletek (operandusok, operátorok) .....	32
3.6.1	Értékadó műveletek.....	32
3.6.2	Matematikai műveletek .....	33
3.6.3	Összetett matematikai műveletek.....	33
3.6.4	Egyoperandusú művelet .....	34
3.6.5	Kétooperandusú műveletek .....	34
3.6.6	Háromoperandusú művelet .....	34
4	Feltételes utasítások - szelekció .....	35
4.1	Relációs operátorok.....	35
4.2	Feltételes műveletek.....	35
4.3	Feltételes utasítások.....	37

4.3.1	Egyágú szelekció.....	37
4.3.2	Kétágú szelekció .....	37
4.3.3	Többágú szelekció.....	38
4.4	Példa: Feltételes elágazásokat tartalmazó alkalmazás készítése.....	39
5	Stringek kezelése.....	42
5.1	Deklarálás.....	42
5.2	A "string"-ek fontosabb metódusai.....	42
5.2.1	A "string"-ek mint vektorok.....	44
5.3	Példa: String műveletek gyakorlása .....	45
6	Iteráció.....	49
6.1	Előírt lépésszámú ciklusok.....	49
6.2	Feltételes ciklusok.....	50
6.3	Példa: Prímszám vizsgálat.....	54
7	Saját metódusok létrehozása .....	56
7.1	Példa: Két szám legnagyobb közös osztójának a meghatározása .....	56
8	Objektumosztályok és objektumok létrehozása .....	59
8.1	A programok szerkezete (osztály, adattag, metódus).....	59
8.1.1	A program összetevői.....	59
8.1.2	Az osztály deklarációja.....	59
8.1.3	Adattagok deklarációja .....	59
8.1.4	Metódusok deklarációja.....	60
8.1.5	Metódusok paraméterezése .....	61
8.1.6	Adatok és metódusok láthatósága .....	62
8.1.7	A property-k .....	63
9	Grafika.....	65
9.1	A Graphics osztály .....	65
9.2	A színek (Color).....	66
9.3	A toll (Pen).....	66
9.4	Az ecset (Brush).....	67
9.4.1	Mintás ecset és toll (TexturedPen).....	67
9.5	Szöveg rajzolása.....	68
9.6	Állandó grafikák létrehozása az űrlapon.....	68
10	Többablakos alkalmazások készítése .....	70
10.1	Üzenetablakok.....	70
10.2	Példa: Kétablakos alkalmazás .....	72
11	Kivételkezelés .....	76
11.1	Hibakezelés .....	76
11.2	A try és catch.....	76
11.3	A finally blokk .....	78
11.4	Kivételek feldobása.....	78
11.5	Checked és unchecked .....	79
11.6	Példa: Hibakezelés a másodfokú egyismeretlenes egyenlet kapcsán.....	80
12	Állománykezelés .....	83
12.1	Szöveges állományok kezelése .....	83
12.1.1	Szöveges és bináris fájlok írása, bővítése, olvasása.....	83
12.1.2	Bináris fájlok írása, olvasása .....	84
12.2	Példa: Véletlen számok generálása lottószámokhoz, szövegfájlba.....	86
12.3	Könyvtárkezelési utasítások.....	88
12.3.1	Könyvtár műveletek .....	88
12.3.2	Példa: Könyvtár műveletek használata.....	89

12.3.3	Állományműveletek .....	91
13	XML alapok és XML fájlok kezelése .....	94
13.1	Az XML nyelv .....	94
13.1.1	Az XML dokumentum szintaktikája .....	95
13.1.2	Az XML dokumentum érvényessége .....	96
13.1.3	Az XML előnyei .....	96
13.2	XML fájlok feldolgozása C# környezetben .....	97
13.2.1	XML fájl betöltése .....	97
13.2.2	A fa ágai, műveletek node-okkal .....	97
13.2.3	Új node beszúrása és a fájl mentése .....	98
13.2.4	Node törlése a fából .....	98
13.3	Példa XML fájl feldolgozására .....	99
14	Az UML nyelv és fontosabb diagramtípusai .....	104
14.1	Használati esetek .....	108
14.2	Tevékenységi diagram .....	112
14.3	Objektumok és osztályok .....	113
14.3.1	Kapcsolatok .....	115
14.3.2	Szekvenciadiagram .....	122
14.3.3	Kommunikációs diagram .....	124
15	Az alkalmazásfejlesztés folyamata, szoftver-életciklus modellek .....	128
15.1	A rendszerfejlesztés folyamatmodelljei .....	133
15.1.1	A vízésés modell .....	134
15.1.2	A V modell .....	136
15.1.3	Prototípusok alkalmazása .....	137
15.1.4	A spirál modell .....	140
16	Függelék .....	148
16.1	A C# kulcsszavai .....	148

# 1 Alkalmazásfejlesztés és a C#

## 1.1 A forráskód feldolgozása

A forráskód feldolgozásának szempontjából a szoftverfejlesztő rendszereket három csoportba sorolhatjuk:

- fordítók
- értelmezők
- köztes nyelvek

## 1.2 Fordítók

Az adott nyelven (forrásnyelv) megírt programot egy fordítóprogram (compiler) lefordítja egy másik nyelvre, a célnyelvre. A célnyelvi program futtatásához ezután sem a forrásnyelvű programra, sem a fordítóprogramra nincs szükség. A célnyelv általában a processzor gépi nyelve.

Előny:

- gyorsan futó programot ad
- A fordítás egyirányú utca, azaz a lefordított programból nem állítható vissza a forráskód. Hiába tesszük közzé a lefordított programot, az a forráskód nélkül gyakorlatilag nem módosítható.

Hátrányok:

- A lefordított program nem hordozható – nem futtatható más processzoron illetve más operációs rendszer alatt.

### 1.2.1 értelmezők (interpreter):

A forrásnyelven megírt programot ebben az esetben nem fordítjuk le. A program futtatásához egy interpreter (értelmezőprogram) szükséges, mely utasításonként olvassa, majd értelmezi a forrásnyelven megírt programot. Az értelmezett utasítást azonnal végre is hajtja, majd megkeresi a végrehajtásban következő utasítást.

Előnyök:

- Ha más operációs rendszerre, illetve más processzorra is létezik értelmező, programunkat ott is futtathatjuk. (Megvalósul a platformfüggetlenség elve.)
- Ha a program, futás közben hibával leáll, a futtatás a hiba kijavítása után folytatható.

Hátrányok:

- Értelmezővel a futtatás lényegesen lassabb, mint ha fordítót használnánk, hiszen minden utasítást futtatás közben értelmezünk. Minden utasítást minden egyes végrehajtásnál értelmezni kell.
- Ha programunkat közzé tesszük, a forráskódot tesszük közzé, így azt mások is fel tudják használni.

### 1.2.2 Köztes nyelvek (intermediate languages)

A köztes nyelveket használó rendszerek a fordítás és a lépésenkénti értelmezés előnyeit próbálják egyesíteni. A program futtatása két lépésből áll:

1. Egy fordítóprogram a forráskódot egy köztes nyelvre fordítja le. A köztes nyelvre lefordított programot tesszük közzé.
2. A köztes nyelvre lefordított programot egy értelmező program hajtja végre lépésenként.

A köztes nyelvre lefordított program futtatásához szükség van a futtató környezetre, vagyis az értelmező programra.

Előnyök:

- Megvalósítható a platformfüggetlenség. Amelyik rendszerhez létezik futtató környezet, képes futtatni a köztes nyelvre lefordított programot.
- Ha programunknak a köztes nyelvre fordított kódját tesszük közzé, megőrizhetjük a forráskódot.

Hátrányok:

- Sajnos a platformfüggetlenségért nagy árat kell fizetnünk. A köztes nyelvre lefordított program futtatása jelentősen lassúbb, mint a gépi kódra fordított program futtatása.
- Az operációs rendszerek általában nem tartalmazzák a szükséges futtatókörnyezetet, ezért azt külön kell telepíteni.

### 1.2.3 Köztes rendszerek a gyakorlatban: Java és .NET

Java

A Java technológiát a Sun Microsystems fejlesztette ki. A cél egy olyan rendszer megalkotása volt, melyre érvényes az, hogy az alá készült programok bármely környezetben platform-függetlenül futtathatók.

A megoldás köztes kód használatán alapul.

A Java fordító a forráskódot egy köztes nyelvre, az úgynevezett byte-kód-ra fordítja, melyet egy képzeletbeli számítógép, a Java Virtual Machine tud futtatni. A képzeletbeli Java gépet emuláló programnak, a Java Runtime Enviroment-nek jelen kell lennie gépünkön a Java alkalmazások futtatásához.

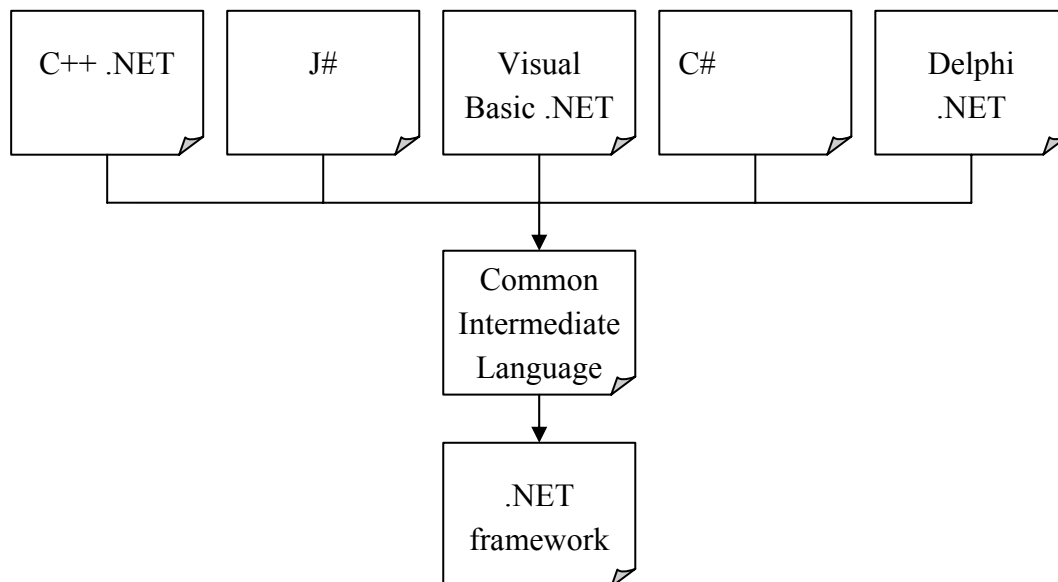
## .NET framework

A .NET keretrendszer (framework) szintén köztes nyelv használatán alapul. A .NET rendszer köztes nyelvét CIL-nek hívják (Common Intermediate Language), melyet a CLR (Common Language Runtime) névre keresztelt virtuális gép képes futtatni.

A .NET programok futtatásához szükséges környezet 2-es verziója 2005 októberében jelent meg, így nem része a Windows telepítőkészleteknek. A 22.5 MB-os telepítőkészlet ingyenesen letölthető a Microsoft honlapjáról. A .NET framework-nek létezik egy compact változata is, mely mobil eszközökön teszi lehetővé alkalmazásaink futtatását.

CIL-t értelmező futtató környezetek megjelentek Linux alá is. Két fejlesztés is fut párhuzamosan: a Portable.NET és a Mono.

A .NET technológia a Microsoft Java verziójából alakult ki. A két technológia közti egyik alapvető különbség, hogy Java környezethez csak Java nyelven írhatunk programokat, .NET környezet alá többféle nyelvből létezik fordító.



### 1.3 Az Objektumorientált programozás (OOP) elve

A programozási paradigma (vagy elv) a számítógépes programok tervezésének és programozásának módszerét jelenti. Ezen elvek célja az, hogy a programozók a programokat minél hatékonyabban tudják elkészíteni, azokkal az adott problémát minél egyszerűbben tudják megoldani. Ebben a fejezetben az objektumorientáció néhány alapfogalmával ismerkedünk meg. A C# programok az OOP paradigma szerint épülnek fel.

**Osztály (Class)** – Az osztály az objektum-orientált program alapegysége. Egy “dolog” tulajdonságait és viselkedését általánosságban írja le.

Az osztálynak lehetnek:

- **Adattagjai (data members-s)**, melyek leírják az adott osztály tulajdonságait.
- **Metódusai (methods)**, melyek meghatározzák az osztály viselkedését.

**Objektum (Object)** – Az objektumosztály egy konkrét példánya. Például a kutyákat általában leíró objektumosztály egy konkrét példánya “Lassie”, a kutya. Egy objektumosztály tetszőleges számú példányát létrehozhatjuk.

**Öröklődés (Inheritance)** - Egy osztály öröklöheti egy már meglévő osztály tulajdonságait és metódusait.

**Ősosztály (superclass)** - Azt az osztályt, amelytől öröklünk, őszosztálynak nevezzük.

**Származtatott osztály (subclass)** - Az az osztály, amely örökl.

A származtatott osztály további tulajdonságokkal és metódusokkal egészítheti ki az őszosztálytól örökölt adattagokat és metódusokat. Lehetőség van az örökölt adattagok, illetve metódusok felülírására is. (lásd: polimorfizmus később.)

A származtatás mindig egyfajta **specializáció**. A “Labrador” osztály a nála általánosabb “Kutya” osztály leszármazottja.

Az öröklődést meg kell különböztetni a **tartalmazás** viszonylattól: a kutyánk - adattagjai között - lehet egy anyja (egy másik kutya) és egy apja.

**Betokozottság (Encapsulation)** – Az objektum tulajdonságai és működése el van zárva a külvilág elől. A tulajdonságokat és metódusokat csak meghatározott csatornákon keresztül lehet elérni, a hozzáférést korlátozhatjuk. Az objektum tulajdonságait és metódusait az alábbi csoportok valamelyikében adhatjuk meg:

- private :** Az osztály példányai és az osztályból származtatott osztályok nem férhetnek hozzá az itt felsorolt tulajdonságokhoz, illetve metódusokhoz.
- protected :** Az ide soroltakhoz a származtatott osztályok hozzáférnek, de a konkrét objektumok nem.
- public:** Ezekhez a tulajdonságokhoz és metódusokhoz mindenki hozzáférhet.

Hozzáférés	Az adott osztály metódusai	Az osztályból származtatott osztály metódusai	Az osztály példánya
private	Igen	Nem	Nem
protected	Igen	Igen	Nem
public	Igen	Igen	Igen

**Polimorfizmus** (többretűség, vagy sokalakúság, sokoldalúság) — Egy kört megadhatunk középpontjának koordinátaival és sugarával vagy a kört befoglaló négyzet koordinátaival. Az első esetben a kör megadásához három, a második esetben négy paraméter szükséges. A “kör” osztály a kör létrehozásához tartalmazhat két azonos nevű metódust, az egyiket három, a másikat négy paraméterrel. Azt az esetet, amikor egy osztály több azonos nevű, de a paraméterek számában eltérő metódust tartalmaz, túlterhelésnek (overloading) nevezzük.

Az osztályok többé-kevésbé önálló egységeket alkotnak a programon belül, ezért gyakorlati megvalósításuk is elkülönülő feladatokként történik. Összetettebb program tervezése a szükséges osztályok meghatározásával kezdődik. Az egyes osztályok megvalósítását párhuzamosan több programozó is végezheti.



## 2 A Visual Studio 2005. fejlesztőkörnyezet

### 2.1 A Visual Studio 2005. letöltése és telepítése

#### 2.1.1 Letöltés

A Visual Studio 2005 telepítő csomag a Campus Agreement keretében a felsőoktatásban résztvevő hallgatók és oktatók számára ingyenesen hozzáférhető a

<http://www.tisztaszoftver.hu/>

honlapon. A honlap csak az egyetemek regisztrált címtartományából érhető el, otthon felesleges próbálkozni a letöléssel. Aki nem tudja megoldani a letöltést és az image fájlok CD-re égetését, 5000 Ft. költségtérítés ellenében postai úton juthat a telepítőlemezekhez. A rendelés részleteiről szintén a fenti honlapon olvashatunk tájékoztatást.

A telepítőkészlet öt CD-ből áll. Az első kettő tartalmazza magát a fejlesztőkörnyezetet, míg az utolsó három az elektronikus dokumentációt. A dokumentáció Interneten keresztül is elérhető.

#### 2.1.2 Telepítés

Sajnos a Visual Studio 2005 telepítőprogramjába hiba került. Az első CD telepítése közben a gép egy rejtélyes **ENU 1** nevű **CD**-t kér - ilyen lemez nincs a telepítőkészletben. A hibát az okozza, hogy az „\_15780\_RTL\_x86\_enu\_NETCF\_v2.0.cab” fájlt a telepítő az első CD-n keresi, pedig a második CD-n van. A sikeres telepítéshez egy apró trükköt kell alkalmaznunk:

1. A letöltött ISO fájlokat égessük CD-re.
2. Kezdjük el a telepítést - futtassuk a setup.exe-t az első CD-ről.
3. Folytassuk a telepítést addig, amíg a képernyőn meg nem jelenik az előre kitöltött termékkulcs. A termékkulcsot jegyezzük fel egy darab papírra, majd lépünk ki a telepítőből.
4. Hozzunk létre egy könyvtárat a merevlemezre, majd másoljuk ide az első két CD tartalmát. Fontos: ne másoljuk az egyes CD-ket külön alkönyvtárakba. A Windows hat fájlnál kérdezi meg, hogy felülírhatja-e a már meglévőt. Nyugodtan válaszolhatunk *igen*-t.
5. Kezdjük újra a telepítést a merevlemezre másolt setup.exe-vel.
6. A termékkulcs már megvan, folytathatjuk a telepítést az instrukcióknak megfelelően.

## 2.2 A Visual Studio 2005. fejlesztőkörnyezet használata

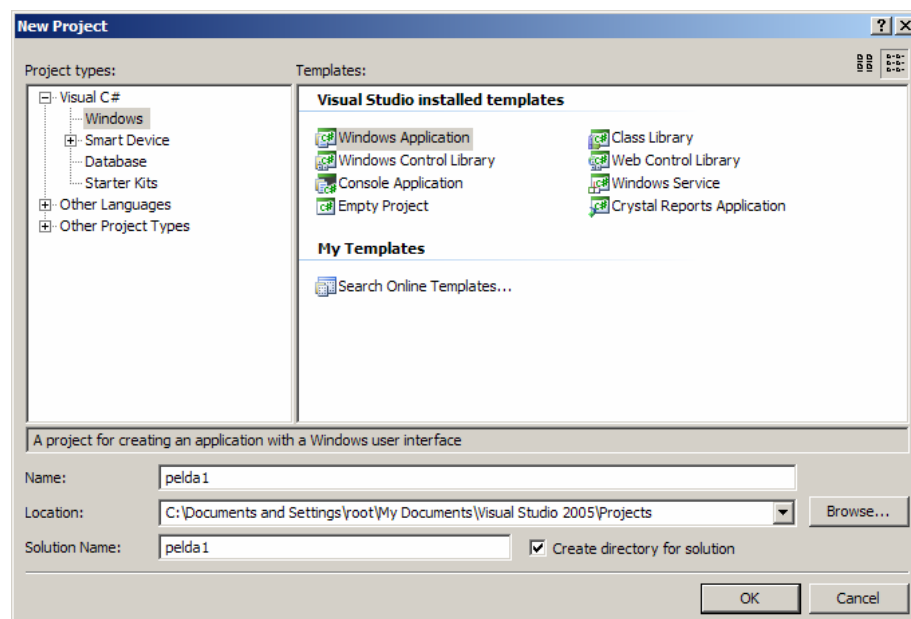
Az IDE (integrated development environment – integrált fejlesztőkörnyezet) egyfajta szoftver, mely segíti a programozót a programírásban és tesztelésben. A következő részekből áll:

- Forráskód szerkesztő
- Grafikus felhasználói felület tervező
- Fordító vagy értelmező program
- Debugger (Hibakereső)


### 2.2.1 Új projekt létrehozása

A fejlesztőkörnyezet használatát egy klasszikus példán keresztül mutatjuk be: készítünk egy programot, mely gombnyomásra megjeleníti a „Hello” feliratot egy szövegdobozban.

1. Indítsuk el a fejlesztőkörnyezetet!
2. A „Recent projects” ablakban válasszuk a „Create Project” lehetőséget. Később innen tudjuk megnyitni a már létező projekteket.
3. A Visual Studio által támogatott programozási nyelvek közül válasszuk a C#-ot, a felkínált sablonok (templates) közül a „Windows Application”-t! Nevezzük el a projektet és állítsuk be a mentés helyét. A „Windows Application” sablon választásával olyan projektet hozunk létre, mely már tartalmaz egy ablakot.

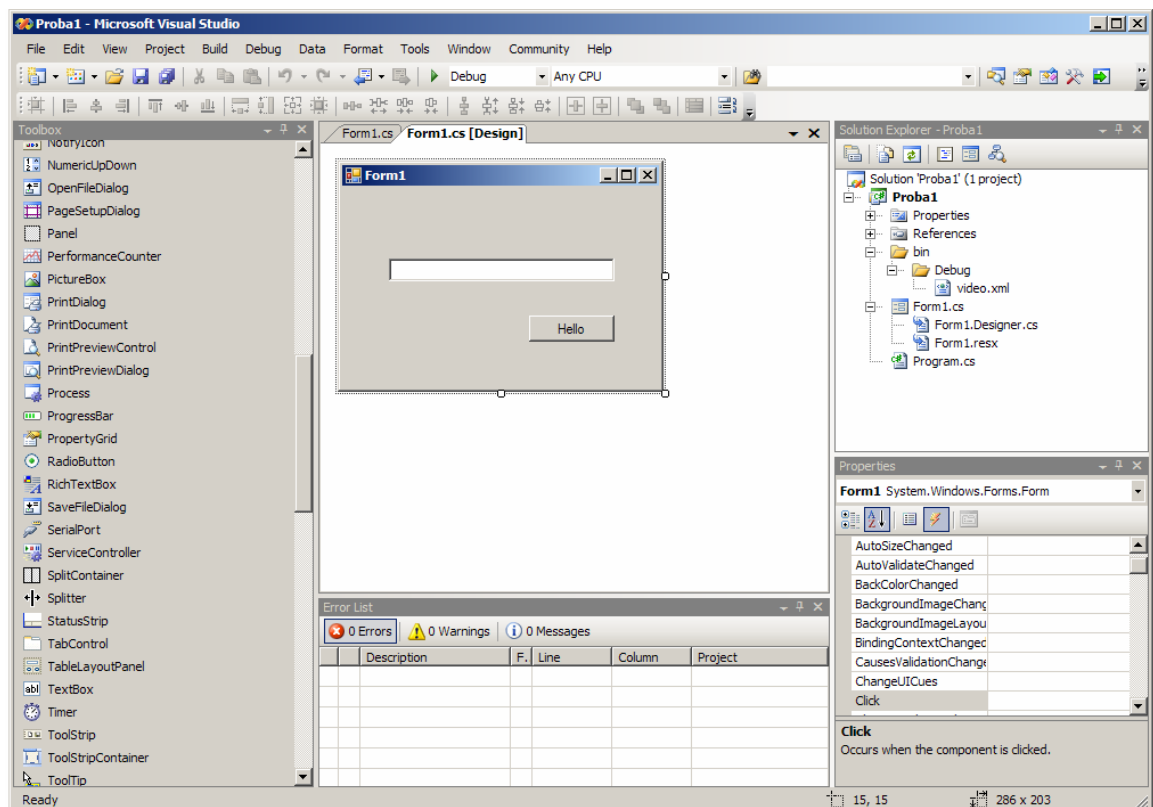


### 2.2.2 Az űrlapszerkesztő és a toolbox

A toolbox-ot az ablak bal oldalán található fülre kattintva nyithatjuk fel. Ha szeretnénk az ablakba dokkolni, kattintsunk az ablakfejléc gombostű ikonjára (  )!

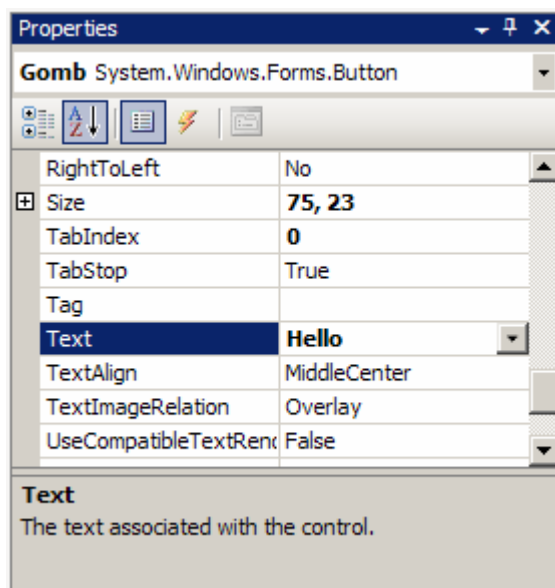
Az toolbox-on látható komponenseket helyezhetjük el az űrlapon. Itt találjuk a gombot, a szövegdobozt, a menüt, stb.

4. Helyezzünk el egy gombot (Button) és egy szövegdobozt (TextBox) az úrlapon (Form)



### 2.2.3 A Properites ablak

A „Properties” ablakot a View / Properties menüponttal jeleníthetjük meg.



### Tulajdonságok (properties) beállítása

Az űrlapokon használt komponensek tulajságait kétféleképpen állíthatjuk be:

- Design time: az űrlap tervezésekkor, a szerkesztőfelületen
- Runtime: a program futása közben, programkódból

A properties ablakban tervező nézetben állíthatjuk be a kiválasztott komponens tulajdonságait. A komponensre jellemző tulajdonságokat megtekinthetjük név szerinti sorrendben, vagy funkciójuk szerint csoportosítva.

Néhány gyakran előforduló tulajdonság:

Name	A komponens neve – ezen a néven tudunk majd programkódból hivatkozni a komponensre. Szigorúbb értelemben nem tulajdonság, mégis itt tudjuk beállítani.	Minden komponens
Text	Szöveg	Form, Button, TextBox
Enabled	Engedélyezettség. Ha értéke hamis (false) a komponens a program futtatásakor le lesz tiltva.	Button, TextBox
Anchor	(horgony) Átméretezhető ablakoknál hasznos. Beállíthatjuk, hogy a komponens mely oldalainak távolsága legyen az ablak oldalaihoz rögzítve.	Button, TextBox

Gyakorlat:

5. Válasszuk ki a gombot és nevezzük el „Gomb”-nak. Feliratát (Text) írjuk át „Hello”-ra!

6. Válasszuk ki a szövegdobozt és nevezzük el „Szoveg”-nek!

### Események (events)

Grafikus felületek programozásánál az eseményvezéreltség elvét követjük. Ez azt jelenti, hogy mindig egy adott eseményhez rendelünk programkódot. Példánkban meghatározzuk, mi történjen akkor, ha a felhasználó a gombra kattintott.

A kiválasztott komponenshez kapcsolódó eseményeket az „Events” gombra kattintva tekinthetjük meg. Ha az esemény neve mellé duplán kattintunk, a kódszerkesztőbe jutunk, ahol megírhatjuk az eseményt kiszolgáló függvényt. (Azt a függvényt, ami az esemény bekövetkeztekor kerül futtatásra.)

A leggyakrabban használt esemény az egérekattintás, a „Click”.

Gyakorlat:

7. Rendeljünk függvényt a gombra kattintás eseményhez.
8. Válasszuk ki a gombot!
9. A „Properties” ablakban jelenítsük meg az eseményeket!
10. A „Click” esemény mellett kattintsunk duplát. Ha mindent jól csináltunk, a kódszerkesztőben találjuk magunkat.
11. Futtassuk projektünket az F5 billentyűvel!

### 2.2.4 Kódszerkesztő

A modern IDE-k forráskód-szerkesztői több funkcióban eltérnek az egyszerű szövegszerkesztőktől:

- Syntax highlighting (szintaxis kiemelés): a szerkesztő a nyelv foglalt szavait más színnel jelöli, így azok elkülönülnek az általunk adott elnevezésektől.
- Auto Code Completion (automatikus kódkiegészítés): a forráskód szerkesztése közben a Ctrl + Space billentyűk lenyomására lista jelenik meg a képernyőn, mely tartalmazza a szintaktikailag odaillő nyelvi elemeket. Így elég egy nyelvi elem első néhány karakterét begépelni, a befejezést választhatjuk a listából. Ezzel a módszerrel nem csak a gépelési idő rövidül le, hanem csökken az elütésekből valamint a kis és nagybetűk helytelen használatából adódó hibák száma. Ahhoz, hogy a felkínált lista mindig aktuális legyen a szerkesztőnek folyamatosan elemeznie kell a teljes forráskódot, ami nagyobb projekteknél jelentős gépkapacitást igényel.
- Code collapse (Kód „összeomlasztás”): Bizonyos nyelvi struktúrák esetén, a sor elején kis négyzetben + vagy - jel jelenik meg. Ezzel elrejtethető a kódnak az a része, amin épp

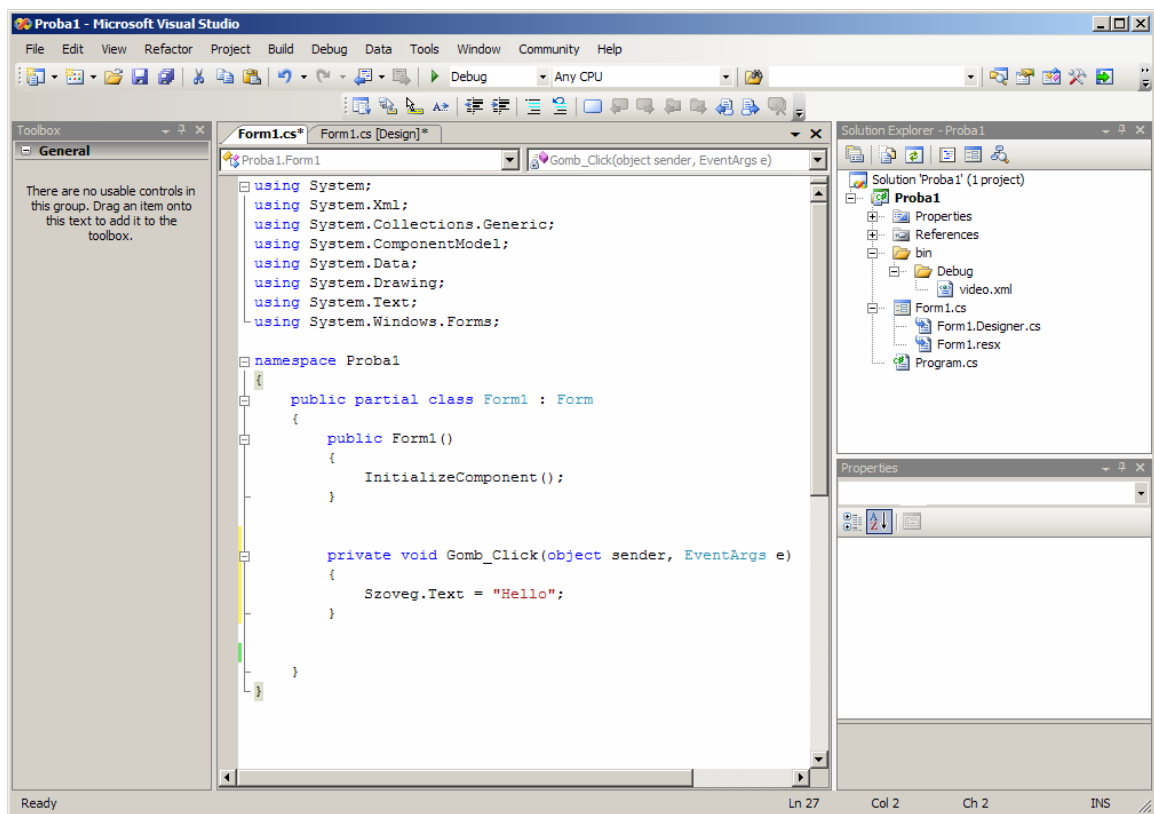
nem dolgozunk. Az összetartozó kódrészleteket vonal köti össze, mely segít az eligazodásban.

- Auto Code Formating (Automatikus kódformázás): Az egymásba ágyazott programstruktúrákat automatikusan egymás alá igazítja. (Edit / Advanced / Format document )

Gyakorlat:

12. A kódszerkesztőben az eseményhez rendelt gomb\_click függvényben állítsuk be a szövegdoz szövegét!

```
Szoveg.Text = "Hello";
```



## 2.2.5 Töréspontok és nyomkövetés

Összetettebb algoritmusoknál gyakran előfordul, hogy a program szintaktikailag helyes, futtatáskor mégsem a kívánt eredményt adja. Ilyen hibák felderítésére használhatjuk a nyomkövetést. A programsorok előtti szürke sávra kattintva töréspontokat helyezhetünk el a forráskódban. Ezeket piros körrel jelöli a szerkesztő. Ha a program futás közben törésponthoz ér, a futtatás megszakad, és a rendszer „debug” módba kerül. Innen programunkat soronként hajthatjuk végre, közben pedig nyomon követhetjük változóink értékének alakulását.

Változók értékének követése:

- Ha az egeret a forráskódban egy változó fölé helyezzük, a képernyőn megjelenik a változó értéke.
- Debug módban a képernyő alján megjelenik egy „Locals” ablak, melyben a modulban érvényes változók listáját találjuk. Ha véletlenül kikapcsoltuk a „Locals” ablakot, a Debug / Windows menüpontban állíthatjuk vissza.
- A Debug / Windows menüpont alatt négy Watch ablakot is találunk. Ezekben a jobb egér gombbal tudjuk összeállítani a vizsgálni kívánt változók listáját. Ugyanitt az „Edit value” lehetőséget ad változóink értékének módosítására is.

### Program futtatása „debug” módban:

Debug módban a fejlesztőkörnyezet újabb eszköztárral bővül:



- F5 Folytatja a program futtatását a következő töréspontig.
- Szünetelteti a program futását.
- Megszakítja a program futtatását.
- Újraindítja a programot.
- F11 „Step into” a következő programsorra ugrik. Ha függvényhívás történik, a függvénybe is beleugrunk, és azt is soronként hajtjuk végre.
- F10 „Step over” a következő programsorra ugrik. Függvényhívásoknál nem ugrik bele a függvényekbe.
- „Step Out” befejezi a függvény futtatását, és a hívási pontnál áll meg.

A fenti funkciókat a Debug menün keresztül is elérhetjük.

## 2.3 Alapvető komponensek használata és speciális vezérlők

A Visual Studio fejlesztőkörnyezet tartalmaz egy komponenskönyvtárat. Ezeket a komponenseket szabadon felhasználhatjuk saját programunkban. A komponenskönyvtárat bővíthetjük más szoftvercégektől vásárolt komponensekkel is - a komponensgyártás új üzletág a szoftverpiacon. (Pl: vonalkód rajzoló, diagram megjelenítő, vagy éppen FTP kezelő.)

Egy gyakorlott programozó tudásának egyre kisebb részét teszi ki a programozási nyelv ismerete. Egyre nagyobb súlyt kap a fejlesztőeszköz által kínált, előre elkészített komponensek ismerete. Jegyzetünk nem referenciakönyv, nem tartalmazhatja az összes

komponens leírását. Egy ilyen kiadvány terjedelme több-ezer oldal lenne, ami egyrészt megfizethetetlen, másrészt kezelhetetlen. A mostani fejlesztőkörnyezetek referenciakönyvei már csak elektronikus formában érhetők el.

Ebben a fejezetben egy nagyon rövid összefoglalót adunk a leggyakrabban előforduló komponensekről.

Azokat a komponenseket, melyeken keresztül a felhasználó kezelni tudja az alkalmazást, vezérlőknek (controls) nevezzük. Ilyenek a gomb, a menü, a szövegdoboz, kép, stb. Léteznek olyan komponensek is, mint például az időzítő (Timer), melyek futtatás közben nem jelennek meg a képernyőn. Ezek nem tartoznak a vezérlők közé.

Azokat a tulajdonságokat, melyekkel a legtöbb vezérlő (control) rendelkezik, az alábbi táblázatban foglaljuk össze, és nem említjük külön-külön az egyes vezérlők bemutatásánál.

Name	A komponens neve – ezen a néven tudunk majd programkódból hivatkozni a komponensre. Szigorúbb értelemben nem tulajdonság, mégis itt tudjuk beállítani.	Minden komponens
Text	Szöveg	Form, Button, TextBox
Enabled	Engedélyezettség. Ha értéke hamis (false) a komponens a program futtatásakor le lesz tiltva.	Button, TextBox
Anchor	(horgony) Átméretezhető ablakoknál hasznos. Beállíthatjuk, hogy a komponens mely oldalainak távolsága legyen az ablak oldalaihoz rögzítve.	Button, TextBox
TabIndex	Windows-os alkalmazásokat általában egér nélkül, billentyűzetről is kezelhetünk: a Tab illetve Shift + Tab billentyűkkel léphedhetünk előre és hátra az ablak elemei között. A TabIndex tulajdonság vezérlőnk helyét adja meg ebben a sorban.	
TabStop	Jelzi, hogy a Tab billentyűvel adhatunk-e fókust az elemnek.	
Location / x,y	A vezérlő koordinátáit adja meg az őt tartalmazó elem (pl. űrlap) bal-felső sarkától mérten.	
Size / Width, Height	A vezérlő szélessége és magassága pixelekből megadva.	



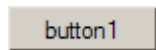
Text	A vezérlőhöz rendelt szöveg	
TextAlign	A szöveg helyzetét állítja a vezérlőn belül.	
Font	A vezérlő szövegének betűtípusa.	
ForeColor	A vezérlő előtérszínét állítja.	
BackColor	A vezérlő háttérszínét állítja.	
Cursor	Az egérkurzor alakja a vezérlő felett.	
AutoSize	A vezérlő méretét automatikusan úgy állítja be, hogy a vezérlő szövege elférjen.	
UseMnemonic	Mnemonic-ok (emlékeztetők) használatának engedélyezése. Mnemonic: Ha gombok vagy menüpontok feliratában „&” karaktert helyezünk el, az „&” karakter utáni betű aláhúzva jelenik meg. Az aláhúzott betű Alt billentyűvel együtt leütve kattintás eseményt vált ki. Pl: &Ok -> <u>O</u> k	

#### Események

Click	Kattintáskor következik be. A kattintás történhet egérrel, vagy bizonyos vezérlőknél az Enter billentyűvel.	
TextChanged	Akkor következik be, ha a vezérlő Text tulajdonsága (szövege) megváltozik.	
Enter	Akkor következik be, amikor a vezérlő megkapja a fókuszt. (Kiválasztjuk egérrel vagy a Tab billentyűvel)	
Leave	Akkor következik be, ha a vezérlő elveszti a fókuszt.	
MouseClicked	Egérkattintáskor következik be.	
MouseDown	Akkor következik be, ha a komponens fölött lenyomjuk az egérgombot.	
MouseUp	Akkor következik be, ha a komponens fölött felengedjük az egérgombot.	
MouseEnter	Akkor következik be, amikor az egér a komponens látható része fölé ér.	
MouseLeave	Akkor következik be, amikor az egér elhagyja a komponens látható részét.	
MouseMove	A komponens fölött megmozduló egér váltja ki.	
MouseHover	Akkor következik be, amikor az egér már	

	eltöltött egy bizonyos időt a komponens fölé.	
--	---	--

### 2.3.1 A gomb (Button)



#### Tulajdonságok (Properties)

Text	A gomb felirata.
Image	A gombon levő kép.
ImageAlign	A kép elhelyezkedése a gombon belül.

#### Események (Events)

Click	A gomb megnyomásakor következik be.
-------	-------------------------------------

### 2.3.2 Szövegdoz (TextBox)



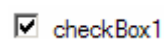
#### Tulajdonságok (Properties)

Text	A szövegdoz szövege ezen a tulajdonságon keresztül állítható be és olvasható ki.
Multiline	Ha értéke igaz, a szövegdozba több sornyi szöveget is írhatunk.
UseSystemPasswordChar	Ha értéke igaz (true), a begépet szöveg helyén egy meghatározott karakter jelenik meg, így a jelszavakat nem lehet leolvasni a képernyőről.

#### Események (Events)

TextChanged	Akkor következik be, ha a szövegdoz szövege megváltozik.
-------------	--

### 2.3.3 Jelölőnégyzet (CheckBox)



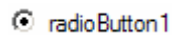
#### Tulajdonságok (Properties)

Checked	Jelzi, hogy a doboz bejelölt állapotban van-e.
CheckedState	A jelölőnégyzet három állapotban lehet: - bejelölt (checked) <input checked="" type="checkbox"/> - jelöletlen (unchecked) <input type="checkbox"/> - köztes (intermediate) <input checked="" type="checkbox"/>

#### Események (Events)

CheckStateChanged	Jelzi, ha a jelölőnégyzet állapota megváltozott
-------------------	---

### 2.3.4 Rádiógomb (RadioButton)



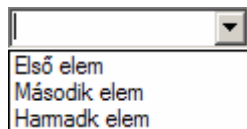
#### Tulajdonságok (Properties)

Checked	Jelzi, hogy a gomb benyomott állapotban van-e.
---------	--

#### Események (Events)

CheckedChanged	Jelzi, hogy a gomb állapota megváltozott.
----------------	---

### 2.3.5 Legördülő doboz (ComboBox)



#### Tulajdonságok (Properties)

Items	Az items gyűjtemény tartalmazza a leördülő lista elemeit.
Sorted	Jelöli, hogy a lista elemei abc sorrendben jelenjenek-e meg.

#### Események (Events)

TextUpdate	Akkor következik be, ha a legördülő lista szövege megváltozik.
SelectedIndexChanged	Akkor következik be, ha a kiválasztott elem sorszáma megváltozik.

A legördülő doboz elemeinek feltöltése történhet programból is:

```
comboBox1.Items.Add("Első elem");  
comboBox1.Items.Add("Második elem");  
comboBox1.Items.Add("Harmadik elem");
```

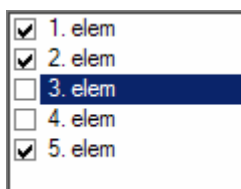
A kiválasztott elem sorszámát a `ComboBox.SelectedIndex` tulajdonságán keresztül tudjuk kiolvasni. Ez a tulajdonság csak programkódból érhető el, tervező nézetben nem.

```
int valasztott = ComboBox1.SelectedIndex;
```

Az elemeket programkódból is törölhetjük:

```
ComboBox1.Items.RemoveAt(valasztott);
```

### 2.3.6 Jelölőnégyzet lista



Ez a komponens jelölőnégyzetekből álló listát jelenít meg. Tulajdonságai hasonlítanak a legördülő dobozéhoz (`ComboBox`). A fő különbség az, hogy itt egyszerre több elemet is ki tudunk választani.

Tulajdonságok (Properties)

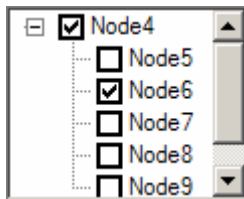
Items	Az items gyűjtemény tartalmazza a leördülő lista elemeit.
CheckedItems	Ez a gyűjtemény tartalmazza azokat az elemeket, melyeket a felhasználó kiválasztott.
Sorted	Jelöli, hogy a lista elemei abc sorrendben jelenjenek-e meg.

Események (Events)

SelectedIndexChanged	Akkor következik be, ha a kiválasztott elem sorszáma megváltozik.
----------------------	---

### 2.3.7 Listadoboz (ListBox)

Egy fát jelenít meg, melyben a felhasználó több elemet is bejelölhet.



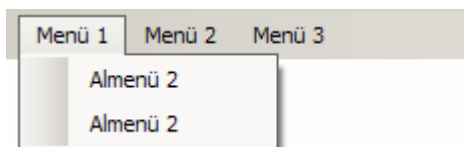
#### Tulajdonságok (Properties)

Nodes (collection)	A nodes gyűjtemény tartalmazza a fa elemeit.
CheckBoxes	Ha értékét igaz-ra állítjuk, a fa minden eleme előtt jelölőnégyzet jelenik meg.
FullPath	Tervezőnézetben nem elérhető tulajdonság. A felhasználó által kiválasztott elemek útvonalát adja vissza a fában.

#### Események (Events)

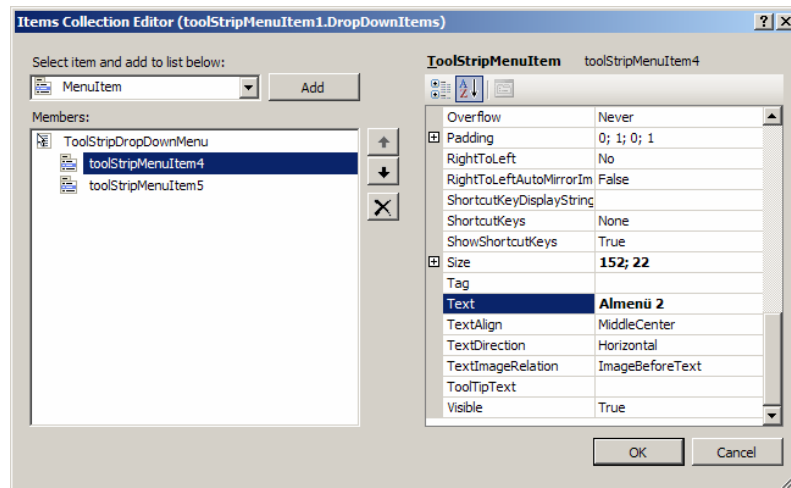
AfterCheck	Akkor következik be, ha egy jelölőnégyzet állapotát megváltoztatja a felhasználó
------------	--

### 2.3.8 Menü építés a ToolStrip komponenssel



A menü felépítése a ToolStrip komponens úrlapra helyezésével kezdődik. A ToolStrip DropDownItems tulajdonsága mellé kattintva egy párbeszédablak jelenik meg, melyben létrehozhatjuk a menüpontokat. Itt van lehetőségünk a menüpontok sorrendjének és tulajdonságaik beállítására is. Minden menüpont rendelkezik DropDownItems tulajdonsággal. Ez a tulajdonság egy gyűjtemény, mely tartalmazza a menüpont almenüpontjait. Ha a Toolbox-on a DropDownItems tulajdonság mellé kattintunk, megjelenik egy párbeszédablak, ahol létrehozhatjuk a szükséges almenüket.

Minden menüponthoz tartozik Click esemény, mely az adott menüpont kiválasztásakor következik be.

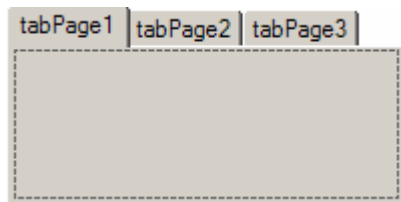


## 2.3.9 Tárolók (Containers)

A tárolók olyan vezérlők, melyek más vezérlők elrendezésére szolgálnak. A tárolókba további vezérlőket helyezhetünk.

A tárolókon keresztül adatbevitel nem történik, eseményeiket is csak ritkán kezeljük.

### 2.3.9.1 Lapozható kartoték (TabPage)



A kartotékhoz új fület legegyszerűbben a jobb egér / Add tabs menüponttal adhatunk legegyszerűbben. A lapok sorrendjének, illetve az egyes lapok tulajdonságainak beállításához kattintsunk a TabPages tulajdonság mellé. A TabPages tulajdonság gyűjtemény, mely a kartoték lapjait tartalmazza. Itt tudjuk külön állítani az egyes lapok tulajdonságait is.

### 2.3.9.2 Panel



A panel vezérlőre is helyezhetünk további vezérlőket. Ha ezek túlnyúlnak a panel területén görgetősávok jelennek meg a panel szélein. Nagyméretű képek megjelenítéséhez is használhatunk panelt: Helyezzük a képet (PictureBox) a panelbe. Ha a kép nem fér be a panel területére, a megjelenő görgetősávok segítségével mozgathatjuk.

### 2.3.9.3 Osztott tároló (SplitContainer)



Két panelből és a közük levő elválasztó vonalból áll. Ha a BorderStyle tulajdonságát „Single”-re vagy „3D”-re állítjuk, a felhasználó egerrel állíthatja a két oldal arányát.

#### Tulajdonságok (Properties)

BorderStyle	A két panel keretének stílusa.
Orientation	Értéke lehet „horizontal” vagy „vertical”. A paneleket elválasztó sáv irányát állítja be.
Panel1Collapsed	Ha értéke igaz, az 1-es panel eltűnik, a vezérlő teljes területét a 2-es panel tölti ki.
Panel1MinSize	Az 1-es panel minimális mérete.
Panel2Collapsed	Ha értéke igaz, a 2-es panel tűnik el.
Panel2MinSize	A 2-es panel minimális mérete.
SplitterWidth	A középső elválasztó sáv szélessége.

### 2.3.10 Fájlkezelő párbeszédablakok (FileOpenDialog, FileSaveDialog)

A Windows-os alkalmazásokból jól ismert fájlmegnyitás és mentés párbeszédablakok megjelenítésére szolgáló komponensek.

Működésüket egyszerű példával mutatjuk be. Helyezzünk el az űrlapon egy gombot (Button) és egy OpenFileDialog komponenst. A gomb kattintásához rendeljünk eseményt:

```

private void button1_Click(object sender, EventArgs e)
{
    openFileDialog1.InitialDirectory = "c:\\";

    openFileDialog1.Filter =
        "Szövegfájlok (*.txt)|*.txt|Mindenki (*.*)|*.*";

    openFileDialog1.FileName = "";

    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        MessageBox.Show(openFileDialog1.FileName);
    }
    else
    {
        MessageBox.Show("Nem választott fájlt");
    }
}
}

```

- Az InitialDirectory tulajdonsággal állíthatjuk be a kiinduló könyvtárat. Ne felejtsük el, hogy a C# bizonyos nem látható karakterek könnyű kezelésére úgynevezett karakter konstansokat vezettek be, ezeknek sajátossága, hogy mind a "\" nyitó karakterrel kezdődnek és egy betű, vagy egy számjegy követi a nyitó karaktert. A karakter konstansok a következők.
- Az openFileDialog1.ShowDialog() metódus visszatérési értékéből tudjuk meg, hogy a felhasználó választott-e fájlt.
- A Multiselect tulajdonsággal engedélyezhetjük több fájl egyidejű kiválasztását. Ilyenkor a FileNames tömbből olvashatjuk ki a kiválasztott fájlok neveit.
- A FileSaveDialog hasonló elven működik.

### 2.3.11 Időzítő (Timer)

Meghatározott időközönként eseményt generál.

Tulajdonságok (Properties)

Enabled	Ezzel a tulajdonsággal tudjuk engedélyezni illetve letiltani az időzítőt.
Interval	A két esemény között eltelt idő ezredmásodpercben megadva.

Események (Events)

Tick	Az esemény akkor következik be, ha a számláló lejárt.
------	---

Példa:



Helyezzünk el egy szövegdobozt (TextBox) az űrlapon, Text tulajdonságához írjunk egy „1”-est, nevét hagyjuk változatlanul. Helyezzünk el egy időzítőt (Timer) is az űrlapon. A számlálót engedélyezzük, az események között eltelt időt (Interval) állítsuk 1000 ms-ra. Kezeljük le a Tick eseményt:

```
private void timer1_Tick(object sender, EventArgs e)
{
    int x;
    x = int.Parse(textBox1.Text);
    x++;
    textBox1.Text = x.ToString();
}
```

Időzítőnk minden másodpercben kiolvassa a szövegdoboz értékét, számmá alakítja, eggyel növeli, majd visszaírja a szövegdobozba.

A Timer pontos időmérésre nem alkalmas. Ha két esemény között eltelt időre vagyunk kíváncsiak, használjuk a rendszerórát.

### **3 Nyelvi elemek (kulcsszavak, változók, literálok, konstansok, névterek)**

#### **3.1 Kulcsszavak**

Valamennyi programozási nyelvben a kulcsszavak különleges kifejezéseknek számítanak, amelyeknek egyedi jelentése van, hiszen ezek alkotják a definíciók és műveletek alapszavait. Az adott nyelv szempontjából minden kulcsszónak speciális jelentése van, ezért nem használhatjuk őket saját céljainkra (fenntartott szavak).

Az alábbi táblázatban a C# nyelv kulcsszavait mutatjuk be jelentésével együtt.

#### **3.2 Változók, azonosítók**

Bármilyen programot írunk is minden esetben szükség van arra, hogy a felhasználótól a program futása során ún. input értéke(ke)t kérjünk be, illetve a számításaink során keletkező eredményeket (output) meg tudjuk jeleníteni a képernyőn. Az alapvető input- output, műveletek tárgyalásához mindenképpen meg kell ismerkednünk a C# nyelvben használatos változó típusokkal. Az olyan tárhelyet, amelynek tartalma változhat a program futása közben, változónak nevezzük.

A változók és a változó típusok ismerete nagyon fontos kérdés legyen az bármely programozási nyelv, hiszen a programunkban használt adatokat változókban esetleg konstansokban szoktuk tárolni. A program futása közben ezekkel számol a memóriában, és segítségükkel kommunikál a felhasználóval. A kommunikáció a programozási nyelvek esetén azt jelenti, hogy adatokat olvasunk be a billentyűzetről, és a munka végeztével a kapott eredményt kiírjuk a képernyőre.

Minden változónak rendelkeznie kell névvel vagy más néven azonosítóval, ezen kívül típussal, és tartalommal, vagyis aktuális értékkel. Az előzőekben említetteken kívül rendelkeznie kell élettartammal és hatáskörrel. Az élettartam azt jelenti, hogy az adott változó a program futása során mikor és meddig, a hatókör pedig azt adja meg, hogy a program mely részeiben használható. A változók nevének, azonosítóinak kiválasztása során ügyelnünk kell a nyelv szintaktikai szabályainak betartására.

A változók neveit a fordító azonosítóknak tekinti, amelyeket a Unicode 3.0 szabvány ajánlásának megfelelően kell írni. Az azonosítók képzésénél használható még az aláhúzás karakter (\_) is. Ügyelni kell arra is, hogy több más nyelvvel ellentétben a C# nyelv megkülönbözteti a kis- és nagybetűket.

Az azonosítók csak betűvel vagy aláhúzás jellel kezdődhetnek és nem tartalmazhatnak szóközt, speciális karaktereket (pl. #) valamint kulcsszavakat. Az első karakter után további

betűk, számok vagy aláhúzás jelek következhetnek. Itt kell megjegyeznünk, hogy más nyelvektől eltérően a C# nyelvben elméletileg az ékezetes betűket is használhatjuk névadásra. Bár ez a lehetőség adott mégsem javasoljuk, hogy a programozók éljenek ezzel a lehetőséggel!

A programban használt kulcsszavakat nem használhatjuk azonosítóként. Érdekességként megemlíthetjük viszont, hogy a kulcsszavak a @ bevezető jellel már használhatók akár azonosítóként is (például: @int). Ezt a lehetőséget a programok hordozhatósága illetve átjárhatósága miatt vezették be. A @ prefixű szimbólumok neve valójában @ nélkül kerülnek fordításra. Így attól függetlenül, hogy egy másik nyelv nem tartalmazza például a **”sealed”** kulcsszót, és egy programban azonosítóként használják, a C# nyelvben lehetőség van az azonosító közvetlen használatára.

A kulcsszavakból, változókból, literálokból és azonosítókból kifejezéseket, utasításokat állítunk össze, amelyek összességéből áll össze a forrásprogram. A C# nyelvben az utasításokat pontosvesszővel zárjuk.

A változókat a programban be kell vezetni, azaz közölni kell a fordítóprogrammal, hogy milyen névvel és milyen típusú változót kívánunk használni a programban. Ezt nevezzük deklarációnak.

```
típus változónév;
```

A típus meghatározza a változó lehetséges értékeit, értéktartományait, valamint hogy, milyen műveleteket lehet végezni velük, és milyen más típusokkal kompatibilis. A névvel a változóra tudunk hivatkozni a program bármely részéből. A kompatibilitás akkor fontos, amikor az egyik változót értékül szeretnénk adni egy másik változónak. Abban az esetben ha a két változó inkompatibilis, akkor a .NET fordító hibát jelez.

A változók a program futása közben a memóriában tárolódnak, vagyis minden azonosítóhoz hozzárendeljük a memória egy darabját (címét), amelyet a rendszer deklarációkor lefoglal és a változó teljes életciklusa alatt létezik. Néhány változótípus esetén (pl. pointerek - mutatók) valamivel bonyolultabb a helyzet, de a .NET keretrendszerben nem kell törődnünk a memória kezelésével, mert a .NET felügyeli, lefoglalja és felszabadítja a memóriát, amikor szükséges.

A változóinkat akár már a deklaráció során elláthatjuk kezdőértékkel is. A kezdőérték adás azért is fontos, mert az érték nélkül használt változók kifejezésekben való szerepeltetése esetén a fordító hibát fog jelezni.

```
típus változónév=kezdőérték;
```

Gyakran előforduló hiba, hogy a deklaráció során nem adunk nevet vagy típust a változónak, vagy egyáltalán nem deklaráljuk, de a programban mégis megpróbálunk hivatkozni rá. Ekkor a .NET fordító a futtatáskor hibát jelez. Az is előfordulhat, hogy nem megfelelő típusú kezdőértékkel látjuk el a változónkat. Ebben az esetben különböző hibaüzenetek jelenhetnek meg a képernyőn.

Azonos változónevek használata esetén is hibaüzenetet kapunk. Gyakori hiba az is, hogy az osztály, vagyis a **"class"** neve egyezik meg az egyik változó nevével.

Bizonyos esetekben, amikor nem hiba (Error), hanem figyelmeztető (Warning) típusú hibaüzenetet kapunk, a fordító olyan hibát talál, amitől a programunk még működőképes, de hatékonyságát csökkenti. Ilyen hiba lehet például, ha egy változót deklarálunk, de nem használjuk fel programunkban.

A változók értékadásának természetesen jól meghatározott szintaktikája van. Az értékadó utasítás bal oldalán a változó azonosítója, középen az egyenlőségjel, a jobb oldalon pedig az érték, vagy kifejezés áll, amelynek az aktuális értékét a változóban tárolni kívánjuk.

```
int  adat1=10;
int  adat2=(adat1+5)*4;
```

### 3.3 Literálok, konstansok

A változók egy érdekes típusa a literál, akkor használjuk őket, amikor programunkban egy konkrét értéket szeretnénk használni. A literálok tehát egyszerű, rögzített értékek, tulajdonképpen konstansok. Jelentésük mindenesetben az aktuális tartalmukkal egyezik meg.

#### 3.3.1 Szám literálok

A literálok alapértelmezés szerint **"int"** típusúak, ha egészek, és **"double"** típusúak, ha valósak. Abban az esetben, ha programunkban például **"float"** típust szeretnénk használni, az érték után kell írni az **"f"** karaktert, vagy például **"long"** típus esetén az **"l"** karaktert, illetve **"ulong"** típus esetén az **"ul"** karakterpárt.

Ha magunk szeretnénk megadni a literál típusát, tegyünk hozzá egy utótagot. Ha például a 25 értéket **"long"** típusként szeretnénk használni, írjuk így: 25L. Ha ki szeretnénk kötni az előjel nélküliséget is, akkor: 25ul.

Ha az L utótagot önmagában használjuk, nagybetűt kell alkalmaznunk; más esetben mindegy. Ennek oka az, hogy a l és L (egy) könnyen összetéveszthető, és a fordító ezt meg kívánja előzni.

Lebegőpontos számok esetén a *float* típus jele a f és F, a *decimal* típusé a m és M.

```
Float_literal=16 235 000f
```

### 3.3.2 Logikai literálok

A logikai literálok esetében a *true* és *false* kulcsszavakkal adható meg az érték.

### 3.3.3 Karakter, karakterlánc literálok

Karakter vagy karaktersorozat literált a ” ” jelek között adunk meg.

### 3.3.4 Konstansok

A C# programokban állandókat, vagy más néven konstansokat is definiálhatunk. A konstansok a program futása alatt nem változtatják értéküket, s nem lehet felüldefiniálni őket, vagy értékadó utasítással megváltoztatni értéküket (ilyenkor ugyanis a fordító hibát jelez). Több más nyelvtől eltérően, a C# nyelvben a konstansnak is van típusa.

```
const byte szazalek=100;  
const string="Ez egy konstans string";
```

## 3.4 Névterek

A névterekkel az osztályok valamilyen logika szerinti csoportosítását lehet megoldani. A névterek egy adott osztály-csoport számára egy, a külvilág felé zárt világot biztosítanak, amelyben a programozó döntheti el, hogy mely osztályok láthatóak és használhatóak kívülről, és melyek nem. A névterek segítségével nagyobb újrafelhasználható kódrészeket készíthetünk, valamint modularizálhatjuk programjainkat. Az objektumorientált programozás legmagasabb szintű modulja az osztály. Nagyon fontos megjegyeznünk, hogy a névterek nem az objektumorientáltság fogalmkörébe tartoznak, használatukat inkább a fájlszervezés körébe sorolhatjuk.

A .NET keretrendszer több előre gyártott névteret tartalmaz, amelyek osztályait a programozás során felhasználhatunk. A .NET rendszer biztosítja számunkra például a

- **”System”** névteret, amely az általánosan használt osztályokat (pl. Console) tartalmazza.

A **”System”** több alnévteret tartalmaz, ezekre a **”System.Alnévter”**-ként hivatkozhatunk. Az alnévtereken kívül a **”System”** névtér közvetlenül tartalmazza a gyakori adattípusok (pl. String) definícióját is.

- A **”System.Net”** névtér adja a hálózati programozáshoz szükséges osztályokat.
- A **”System.Windows.Forms”** névtér tartalmazza a Windows programozáshoz szükséges elemeket, komponenseket.
- A **”System.IO”** névtér biztosítja a fájl- és mappa műveleteket.

- A **"System.Data"** névtér szolgáltatja az adatbázis-kezeléshez szükséges komponenseket.

Névteret az alábbiakban látható deklarációval hozhatjuk létre:

```
namespace névtérNeve
{
    Osztályok deklarációja
    namespace
    {
        Osztályok deklarációja
    }
}
```

A névterekben levő osztályokra a következő módon hivatkozhatunk:  
névtérNeve.[AlnévtérNeve...].OsztályNeve  
például: System.Console.WriteLine...

Amint már korábban említettük programjainknak fontos része a felhasználóval való kommunikáció. A program futása alatt adatokat kell bekérni, vagy éppen a program futásának eredményét szeretnénk közölni vele. A változók tartalmának beolvasásához vagy megjelenítéséhez a képernyőn, a .NET rendszerben igénybe vehetjük a C# nyelv alapvető I/O szolgáltatásait, a **"System"** névtérben található **"Console"** osztály ide vonatkozó metódusait (függvényeit és eljárásait).

System.Console.Read(); - A felhasználói inputra használható

System.Console.Write(); - A képernyő konzol alkalmazás ablakába ír

System.Console.ReadLine(); - A konzol ablak ne tűnjön el a program futás végén

System.Console.WriteLine(); - A képernyő konzol alkalmazás ablakába ír

A Console.Write() és a Console.WriteLine() a kiírásra, míg a Console.Read() és a Console.ReadLine() a beolvasásra használható. A beolvasás azt jelenti, hogy alapértelmezés szerint a standard inputeszközökről várjuk az adatokat.

Abban az esetben ha a Read() beolvasó utasítást használjuk, **"int"** típusú adatot kapunk, a ReadLine() metódus esetén viszont **"string"**-et.

```
public static string ReadLine();
```

```
public static int Read();
```

Adat beolvasásakor azonban nem csak erre a két típusra lehet szükségünk, ezért az input adatokat konvertálnunk kell a megfelelő konverziós eljárásokkal.

A **"System"** hivatkozás elhagyható a metódusok hívásakor, amennyiben azt a program elején, a **"using"** bejegyzés után felvesszük. Ezt a műveletet névtér importálásnak nevezzük. Abban az esetben, ha eltekintünk a névtér importálástól, akkor az adott metódus teljes vagy más néven minősített.

```
using System;
```

Mivel konzolról való beolvasás során az adatainkat **”string”** típusú változóba olvassuk be (Console.ReadLine()), így szükség van az adat például számmá történő konvertálására. Ezeket a konverziós lehetőségeket a C# nyelvben a **”Convert”** osztály metódusai (pl..ToInt32) kínálják.

Néhány fontosabb konverziós metódus, korántsem a teljesség igényével. Akit a részletek is érdekelnek, tanulmányozza át a **”Convert”** osztály további metódusait.

- Convert.ToInt32
- Convert.ToChar
- Convert.ToString
- Convert.ToBoolean
- Convert.ToDateTime

### 3.5 A kifejezések

A kifejezéseket konstans értékekből, változókból tárolt adatokból, függvényekből és műveletekből állíthatjuk össze. A kifejezéseket a számítógép kiértékeli, kiszámítja és ennek eredményét felhasználhatjuk a későbbiekben. A kifejezések eredményének típusa a bennük szereplő adatok és műveletek típusától függ, kiszámításuk módját pedig a precedencia szabályok írják elő.

A precedencia szabályok határozzák meg a kifejezésekben a műveleti sorrendet, vagyis azt, hogy az egyes műveleteket a számítógép milyen sorrendben végezze el. Elsőként mindig a magasabb precedenciájú művelet kerül végrehajtásra, majd utána az alacsonyabb precedenciájúak. Azonos precedenciájú műveletek esetén általában a balról jobbra szabály érvényesül. A fontosabb műveleteket a következő táblázatban foglaljuk össze. A táblázatban a precedencia felülről lefelé csökken.

2. táblázat Műveletek precedenciája felülről lefelé

Precedenciaszint	Művelettípus	Operátorok
1.	Elsődleges	(), [], x++, x--, new, typeof, sizeof, checked, unchecked
2.	Egyoperandusú	+, -, !, ~, ++x, --x
3.	Multiplikatív	*, /, %
4.	Additív	+, -
5.	Eltolási	<<, >>
6.	Relációs	<, >, <=, >=, is
7.	Egyenlőségi	==, !=
8.	AND	&

9.	XOR	$\wedge$
10.	OR	$ $
11.	Feltételes AND	$\&\&$
12.	Feltételes OR	$  $
13.	Feltételes művelet	$? :$
14.	Értékadás	$=, *=, /=, \%=, +=, -=, <<=, >>=, \&=, \^=,  =$

Itt kell megjegyeznünk, hogy a műveletek sorrendje zárójelezéssel megváltoztatható, de a zárójelen belül a táblázatban felsoroltaknak megfelelően végzi el a műveleteket.

### 3.6 Műveletek (operandusok, operátorok)

A változókkal kapcsolatosan többféle típusú műveletet tudunk végezni, amelyeket az alábbi kategóriákba sorolhatjuk:

- értékadó műveletek
- matematikai műveletek
- relációs műveletek
- feltételes műveletek
- egyéb műveletek

Ha a műveleti jelek oldaláról nézzük, akkor az alábbi kategóriákat állíthatjuk fel:

- Egyváltozós műveletek
- Kétváltozós műveletek
- Háromváltozós műveletek

#### 3.6.1 Értékadó műveletek

Programjaink írása során az értékadó műveletek segítségével értékeket tudunk adni a változóinkhoz.

```
a=kifejezés;
```

A művelet jobb oldalán csak olyan kifejezés szerepelhet, melynek eredménye közvetlenül megfelel az "a" változó típusának vagy közvetlenül konvertálható az "a" változó típusára.

A C# nyelv lehetőségei adottak arra, hogy egyszerre több azonos típusú változónak adhassunk értéket.

```
a=b=kifejezés;
```



Először kiszámításra kerül a kifejezés értéke, majd az bekerül a "b" változóba és ezt felveszi az "a" változó is.

### 3.6.2 Matematikai műveletek

A matematikai műveletek során az adatokkal számításokat végzünk, amelyek operandusai számok vagy számokat eredményező kifejezések, függvények. A matematikai kifejezésekben használható operátorokat az alábbi táblázatban foglaljuk össze.

3. táblázat Matematikai kifejezésekben használható operátorok

OPERÁTOR	MŰVELET	OPERANDUSOK TÍPUSA	EREDMÉNY TÍPUSA
+	összeadás	egész, valós	egész, valós
-	kivonás	egész, valós	egész, valós
*	szorzás	egész, valós	egész, valós
/	osztás	egész, valós	valós
%	maradékképzés	egész, egész	egész
+	előjelezés	egész, valós	egész, valós
-	előjelezés	egész, valós	egész, valós

### 3.6.3 Összetett matematikai műveletek

Az összetett műveletek lehetővé teszik számunkra, hogy egyszerre végezzük el a számítást és az értékadást. A C# nyelvben lehetőség van arra, hogy az előző pontban ismertetett matematikai műveleteket tömörebb formában fejezzük ki. Ezeket a következő táblázatban tüntettük fel.

4. táblázat Összetett matematikai műveletek tömörebb alakjai

Műveleti jel	Használati alak	Kifejtve
+=	x += kifejezés	x = x+kifejezés
-=	x -= kifejezés	x = x-kifejezés
*=	x *= kifejezés	x = x*kifejezés
/=	x /= kifejezés	x = x/kifejezés
%=	x %= kifejezés	x = x%kifejezés

### 3.6.4 Egyoperandusú művelet

A fentebbi fejezetekben megismert műveletek általában kétoperandusúak voltak. Létezik azonban két olyan művelet, konkrétan a növelés és a csökkentés, amelyek eggyel növelik illetve csökkentik a változó értékét.

`++x;` ugyanaz, mint `x=x+1;`      Prefixes alak  
`--x;` ugyanaz, mint `x=x-1;`      Prefixes alak

Az egyoperandusú műveleteknél a műveleti jelek a változó mindkét oldalán elhelyezhetők:

`x++;` ugyanaz, mint `x=x+1;`      Postfixes alak  
`x--;` ugyanaz, mint `x=x-1;`      Postfixes alak

A növelés és csökkentés viszont nem ugyanúgy zajlik le az egyik illetve a másik esetben. A prefixes alak esetében a növelés vagy csökkentés előzetes műveletként hajtodik végre és ezt követően történnek a további műveletek (pl. értékadás). A postfixes alak esetében a növelés vagy csökkentés utólagos műveletként hajtodik végre és ezt megelőzően történnek a további műveletek (pl. értékadás). Az eredmény szempontjából tehát abszolút nem mindegy melyik alakot használjuk.

### 3.6.5 Kétoperandusú műveletek

A matematikai műveletek során megismert összeadás, kivonás, szorzás, osztás maradékképzés tartozik ebbe a kategóriába, de mivel ott már ezek ismertetésre kerültek, így itt nem térünk ki külön rá.

### 3.6.6 Háromoperandusú művelet

A C# nyelvben egyetlen olyan művelet létezik, amely háromoperandusú, ez pedig egy feltételes művelet. Szintaxisa:

`(feltétel) ? utasítás1 : utasítás2`

A kifejezés három részre oszlik, mely részeket a `"?"` és `":"` operátor választ el egymástól. Amennyiben a kifejezés elején álló feltétel igaz, úgy az utasítás1 kerül végrehajtásra, ha viszont a feltétel hamis, akkor az utasítás2. Ez a háromoperandusú művelet, tulajdonképpen az `"if"` utasítás tömörebb formája.

## 4 Feltételes utasítások - szelekció

A strukturált utasítások más egyszerű és strukturált utasításokból épülnek fel. Az utasítások által definiált tevékenységek sorban (összetett), feltételtől függően (if és switch) vagy ciklikusan ismétlődve (for, while, do-while) hajtódnak végre.

### 4.1 Relációs operátorok

A programjaink írása közben gyakran szükségünk lehet arra, hogy értékeket összehasonlítsunk, amelyhez szükségünk lesz a relációs operátorokra. A relációs műveletek eredménye minden esetben egy logikai érték, ami vagy true(igaz) vagy false(hamis). A relációs műveleteket a leginkább a feltételes utasításoknál és a feltételes ciklusoknál használjuk.

A relációs műveletek lehetséges operátorait a következő táblázatban mutatjuk be:

4. táblázat **Relációs műveletek operátorai**

Műveleti jel	Jelentés
>	nagyobb, mint
<	kisebb, mint
==	egyenlő
!=	nem egyenlő
>=	nagyobb vagy egyenlő, mint
<=	kisebb vagy egyenlő, mint

### 4.2 Feltételes műveletek

Logikai műveletek alatt a szokásos NOT, AND, OR és XOR műveleteket értjük. A NOT művelet operátora a "!" jel, a XOR műveleté a "^" jel. A többi művelet esetében azonban megkülönböztetünk feltételes és feltétel nélküli AND vagy OR műveletet.

5. táblázat **Feltételes műveletek**

Művelet	Műveleti jel	Használat	Magyarázat
Feltételes AND	&&	A && B	Ha A hamis, B már nem értékelődik ki
Feltételes OR		A    B	Ha A igaz, B már nem értékelődik ki
Feltételnélküli AND	&	A & B	B mindig kiértékelődik
Feltételnélküli OR		A   B	B mindig kiértékelődik

A feltételes műveletek és kiértékelésük:

- AND (csak akkor igaz, ha mindegyik részfeltétel igaz)
- OR (csak akkor igaz, ha legalább egy részfeltétel igaz)
- NOT (a megadott logikai kifejezés értékét az ellenkezőjére változtatja)
- XOR (ha két részfeltétel ellentétes értékű – pl.TRUE és FALSE -, akkor TRUE)

A következőkben a teljesség és a gyakorlati használhatóság kedvéért nézzük meg a logikai műveletek igazságtábláit.

*6. táblázat Az AND operátor igazságtáblája*

1. logikai kifejezés	művelet	2. logikai kifejezés	Végeredmény
TRUE	AND	TRUE	TRUE
TRUE	AND	FALSE	FALSE
FALSE	AND	TRUE	FALSE
FALSE	AND	FALSE	FALSE

*7. táblázat Az OR operátor igazságtáblája*

1. logikai kifejezés	művelet	2. logikai kifejezés	Végeredmény
TRUE	OR	TRUE	TRUE
TRUE	OR	FALSE	TRUE
FALSE	OR	TRUE	TRUE
FALSE	OR	FALSE	FALSE

*8. táblázat A XOR operátor igazságtáblája*

1. logikai kifejezés	művelet	2. logikai kifejezés	Végeredmény
TRUE	XOR	TRUE	FALSE
TRUE	XOR	FALSE	TRUE
FALSE	XOR	TRUE	TRUE
FALSE	XOR	FALSE	FALSE

*9. táblázat A NOT operátor igazságtáblája*

1. logikai kifejezés	művelet	Végeredmény
TRUE	NOT	FALSE
FALSE	NOT	TRUE

### 4.3 Feltételes utasítások

A feltételes utasításokat akkor használjuk, ha programunkat érzékennyé kívánjuk tenni a program futása közben valamely kifejezés vagy változó értékének változására. Attól függően milyen értéket vesz fel a változó vagy kifejezés a programunk más-más programrészt hajt végre, azaz elágazik. Az elágazások lehetnek egyágúak, kétágúak vagy többágúak.

#### 4.3.1 Egyágú szelekció

Az **"if"** utasítás egyágú elágazást valósít meg. Szintaxisa a következő:

```
if (feltétel) utasítás;
```

természetesen az utasítás helyén utasításblokk is állhat. *A feltételt mindig zárójelbe kell tenni!*

Amennyiben az **"if"** után álló feltétel igaz, a program az utasítást vagy az utasításblokkot hajtja végre, ha **"hamis"**, akkor nem tesz semmit.

#### 4.3.2 Kétágú szelekció

A **"if else"** kétágú elágazásnál a feltétel hamis esetére is adunk meg végrehajtandó kódot, amit az **else** kulcsszó után adunk meg. Szintaxisa a következő:

```
if (feltétel)
    utasítás1;
else
    utasítás2;
```

ebben az esetben is az utasítás helyén utasításblokk is állhat.

Az **"if"** után álló feltételtől függően a program **"igaz"** esetben az **"utasítás1"**-et hajtja végre, **"hamis"** esetben, pedig az **"utasítás2"**-öt. Fontos szintaktikai elem, hogy az **"else"** kulcsszó előtt is kell a pontosvessző (;), azt főleg a korábbi Delphi programozók figyelmébe ajánljuk. Az elágazások egymásba is ágyazhatók, így gyakorlatilag tetszőleges mélységben építhetjük fel a feltételes utasítások láncolatát.

Előfordulhat, hogy egy **"if"** utasítás valamelyik ágán belül egy másik feltétel alapján újabb elágazást építhetünk be.

Az egymásba ágyazott feltételes utasítások helyett több esetben összetett logikai kifejezést is alkalmazhatunk. A következő utasítás csak akkor hajtódik végre ha mindkét feltétel egyidejűleg teljesül.

```
if (feltétel1 && feltétel2) utasítás;
&& = Logikai ÉS
```

A feltételek egymásba ágyazásánál arra érdemes odafigyelnünk, hogy ezzel ne rontsuk kódunk átláthatóságát illetve hatékonyságát. Ilyen esetben használjunk inkább többágú szelekciót.

### 4.3.3 Többágú szelekció

A **"switch"** *utasítás* segítségével többágú szelekciót tudunk programunkban megvalósítani. Szintaxisa:

```
switch kifejezés of
{
    case      érték1:
                utasítás1;
                break;
    case      érték2:
        utasítás2;
        break;
    case      érték3:
        utasítás3;
        break;
    ...
    default:
        utasításN;
        break;
}
```

Az egyes **"case"** ágak az adott esetben végrehajtandó utasítást esetleg utasításokat tartalmaznak. Az egyes ágak utolsó utasítása a **"break"**, amely elmaradása esetén a fordító hibát jelez. A **"default"** ág a **"case"**-ben felsorolt eseteken kívül eső egyéb lehetőségek bekövetkezése esetén hajtódik végre.

A gyakorlatban előfordulhat olyan eset is, amikor több érték esetén is ugyanazt az utasítást vagy utasításblokkot kell végrehajtani. Mivel a **"case"** kulcsszó után nem lehet több értéket felsorolni, így több **"case"** utasításhoz rendelhetjük ugyanazt az utasítást vagy utasítás blokkot.

```
switch kifejezés of
{
    case      érték1:
    case      érték2:
    case      érték3:
                utasítás1;
                break;
    case      érték2:
        utasítás2;
        break;
    default:
        utasításN;
        break;
}
```

A **"switch"** utasítások **"case"** vezérlőinél az sbyte, byte, short, ushort, int, uint, long, ulong, char, string, enum típusú kifejezéseket használhatjuk.

#### 4.4 Példa: Feltételes elágazásokat tartalmazó alkalmazás készítése

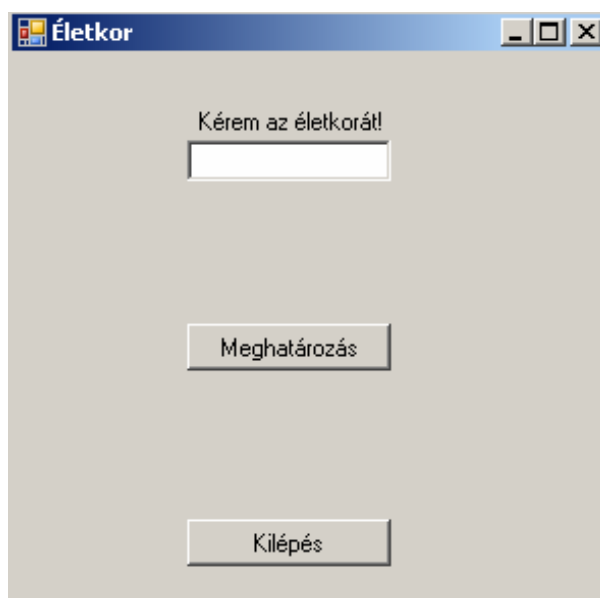
Példánkban szerepeljen egy szövegdoboz (TextBox), amelyben az illető személy életkorát adhatja meg a felhasználó. Ha a „Meghatározás” gombra kattintunk, ugorjon fel egy üzenetablak, melyben a program kiírja a következőket:

Ha az illető életkora

- 1 - 3 év: csecsemő
- 4 - 6 év: óvodás
- 7 - 14 év: általános iskolás
- 15 -18 év: középiskolás
- 19 - 23 év: egyetemista
- egyéb esetben: tanulmányait befejezte

A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását!

1. Hozzunk létre új projektet Eletkor néven!
2. Tervezzük meg az űrlapot az alábbiak szerint:



3. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Form	Életkor	frmEletkor	
Label	Kérem az életkorát!	lblKor	
TextBox		txtKor	
Button	Meghatározás	btnMeghatároz	
Button	Kilépés	btnKilepes	

Eletkor.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Eletkor
{
    public partial class frmEletkor : Form
    {
        public frmEletkor()
        {
            InitializeComponent();
        }

        private void btnMeghatároz_Click(object sender, EventArgs e)
        {
            int kor;
            kor = int.Parse(txtKor.Text);
            if (kor < 0)
                MessageBox.Show("Negatív életkort adott meg!!!");
            else
            {
                switch (kor)
                {
                    case 1:
                    case 2:
                    case 3:
                        MessageBox.Show("Csecsemő");
                        break;
                    case 4:
                    case 5:
                    case 6:
                        MessageBox.Show("Óvodás");
                        break;
                    case 7:
                    case 8:
                    case 9:
                    case 10:
                    case 11:
                    case 12:
                    case 13:
                    case 14:
                        MessageBox.Show("Általános iskolás");
                        break;
                    case 15:
                    case 16:
                    case 17:
                    case 18:
                        MessageBox.Show("Középiskolás");
                        break;
                    case 19:
                    case 20:
                    case 21:
                    case 22:
                    case 23:
                        MessageBox.Show("Egyetemista");
                        break;
                }
            }
        }
    }
}
```



```

                                default:
                                    MessageBox.Show("Tanulmányait
befejezte!");
                                break;
                            }
                        }
                    }

    private void btnKilepes_Click(object sender,
EventArgs e)
    {
        Close();
    }
}

```

## 5 Stringek kezelése

### 5.1 Deklarálás

A C# nyelvben is, mint minden magas szintű nyelv esetében létezik a "string" típus. Általánosságban azt mondhatjuk, hogy az objektumorientált nyelvek esetén a "stringeket" osztályként kezeljük. Tehetjük ezt azért, mert vannak módszereik, tulajdonságaik valamint paramétereik, ahogy ezt már az osztályoknál megszoktuk. Persze a gyakorlatban úgy dolgozunk velük, mint a többi programozási nyelv "string" típusával. Mivel a C# is típusos nyelv, így a többi változóhoz hasonlóan deklarálni kell őket, sőt ezzel egyidőben kezdőértéket is adhatunk nekik.

```
string szoveg="Hello";
```

Amennyiben úgy deklarálunk egy "stringet" hogy nem adunk neki kezdőértéket, akkor kezdetben "null" értéke van.

```
string szoveg;
```

A "string" változó értékét bármikor lekérdezhetjük programunkban és kiírathatjuk.

```
lblString.Text=szoveg;
```

A "string" értéke minden esetben karakterlánc literálként kerül tárolásra.

A "string"-eket használhatjuk konstansként is, csakúgy mint más típusok esetén.

```
const szoveg_string="konstans";
```

### 5.2 A "string"-ek fontosabb metódusai

A relációk használata tulajdonképpen a "string"-eknél is megengedett, persze vannak bizonyos megkötések. A "string"-ek egyes elemeit relációba tudjuk hozni egymással `szoveg[1]<szoveg[2]`, a teljes "string"-re azonban csak a "!=" és a "==" relációkat alkalmazhatjuk. Ha két "string"-et kívánunk összehasonlítani, abban az esetben a következő megoldást használhatjuk:

```
string1.CompareTo(string2)
```

Amennyiben a két "string" egyenlő, akkor a visszatérési érték 0. Ha a kapott érték kisebb, mint 0, akkor viszont a `string1` ABC sorrendben előbb helyezkedik el mint a `string2`, ha viszont nagyobb mint 0, akkor a `string2` van előbb.

`TrimStart('karakter')` metódus eltávolítja a szöveg elejéről a paraméter listájában megadott karakter összes előfordulását.

```
string szoveg="bbbbbele";  
szoveg.TrimStart('b') → bele
```

`TrimEnd('karakter')` metódus eltávolítja a szöveg végéről a paraméter listájában megadott karakter összes előfordulását.

```
string szoveg="öveggggg";  
szoveg.TrimEnd('g') → öve
```

A `TrimStart()`, `TrimEnd()` és a `Trim()` metódusok paraméter nélküli alakban a "string" elejéről, végéről illetve elejéről és végéről is levágják a szóközöket, azaz tulajdonképpen a paraméter nélküli alakban a "-karakter az alapértelmezett.

Itt érdemes felhívni a figyelmet arra, hogy a "string"-eket macskakörmök ("..."), míg a `TrimStart()` és `TrimEnd()` metódusok paramétereit aposztrófok ('...') közé tettük, mivel az első esetben "string"-eket, a második esetben pedig karaktert definiáltunk.

A `ToUpper()` és a `ToLower()` metódusok segítségével a "string"-ek szükség esetén nagybetűssé illetve kisbetűssé alakíthatók. Íme a klasszikus példa:

```
string szoveg="Nemecsek Ernő";  
szoveg.ToUpper() → NEMECSEK ERNŐ  
szoveg.ToLower() → nemecsek ernő
```

A `Substring(kezdoindex,karakterszam)` metódussal, a "string" egy darabját tudjuk kinyerni. A paraméterlistában meg kell adni a kezdő indexet, ahonnan a másolást kezdjük, majd pedig a másolandó karakterek számát. Az alábbi programrészlet a `Substring()` metódus használatát mutatja be.

```
string szoveg="Nemecsek Ernő";  
szoveg.Substring(9,4) →ERNŐ
```

Ha a `Substring()` metódusnak csak egy paramétert adunk és a paraméter értéke a "string" indextartományán belül van, akkor az adott indextől a "string" végéig valamennyi karaktert visszaadja.

A paraméterezéskor ügyelni kell arra is, hogy mindig csak akkora indexet adjunk meg, amennyi eleme van a "string"-nek. A "string" első eleme a 0. indexű, míg az utolsó pedig a karakterszám-1.

A metódusok között fontos szerepet tölt be a keresés. Ennek a metódusnak `IndexOf()` a neve. A. függvény -1-et ad vissza, abban az esetben ha a paraméter listájában megadott karakter,

vagy karaktersorozat nem szerepel a "string"-ben, ha szerepel 0-át vagy pozitív értéket. Azt már korábban láttuk, hogy a "string"-ek 0- tól kezdve vannak indexelve, így találat esetén 0-tól a "string" hosszáig kaphatunk visszatérési értéket, ami a találat helyét, azaz a kezdő indexét jelenti a keresett karakternek, vagy karakterláncnak.

```
string mondat="Az ünnepeken mindig nagy a kavalkád.";
string szo="kád";
if (mondat.IndexOf(szo)>-1)
    lblNev.Text="A "+szo+" szerepel a "+ mondat+" ban.";
```

Igaz hogy a "+" jel nem metódus, de mint művelet értelmezett a "string"-ekre, összefűzést jelent. Az alábbi példában egy vezetéknév és egy keresztnév fűzünk össze, ügyelve arra, hogy a két rész közé elválasztó "SPACE" karaktert is tegyünk, majd kiíratjuk egy címke felirataként.

```
string vezeteknev="Nemecsek";
string keresztnév="Erő";
nev=vezeteknev+" "+keresztnév;
lblNev.Text=nev;
```

### 5.2.1 A "string"-ek mint vektorok

A "string"-ek kezelhetők vektorként is, ami azzal az előnnyel jár, hogy hivatkozhatunk rá elemenként, mint a karaktereket tartalmazó tömbökre.

```
string szoveg1=txtSzoveg1.Text;
int darab=0;
for (int i = 0; i < szoveg1.Length; i++)
{
    darab++;
}
MessageBox.Show(txtSzoveg1.Text+" szó "+ Convert.ToString(darab)+"
karakterből áll");
```

A "string" Length tulajdonságával a hosszára hivatkoztunk, majd az előírt lépésszámú ciklus magjában mindig az aktuális karaktert írtuk ki. címkére. A fenti programrészletben azt használtuk ki, hogy a "string" tulajdonképpen egy karaktertömb.

A "string"-ekbe lehetőségünk van rész "string"-eket beszúrni, melyet az Insert() metódus segítségével valósíthatunk meg. A metódus paraméterlistájának tartalmaznia kell azt az indexet, ahová be szeretnénk szúrni, és a karaktert vagy a karakterláncot. A következő kis programrészletben az Insert() metódus használatát mutatjuk be.

```
string nev="Nem Erő";
nev=nev.Insert(3,"ecsek");
```

A "string" típusról nyugodtan kijelenthetjük, hogy nélküle nem létezik komolyabb program. A leggyakrabban előfordulása talán a fájlkezelő programokhoz köthető. A "string"-eket felhasználhatjuk még például számrendszerek közötti átváltások eredményeinek tárolásakor,

vagy olyan nagy számok összeadására, szorzására, amelyekhez nem találunk megfelelő méretű változókat.

Sajnos nem minden programozási nyelv tartalmazza a string típust, és viszonylag kevés olyan nyelv van, amelyben osztályként használhatjuk őket, de ahol igen és szerencsére a C# nyelv ilyen, éljünk a lehetőséggel, mert sok esetben nagyban megkönnyíti illetve meggyorsítja a programozási munkánkat.

A "string"-ek használatának a felsoroltakon kívül még nagyon sok metódusa létezik, nem beszélve a tulajdonságokról. Ezt a részt tekintjük ízelítőként a "string"-ek kezeléséhez. Akit részletesebben is érdekel a "string"-kezelés és a metódusok, tulajdonságok, az nyugodtan lapozzon fel egy referenciakönyvet, amely tekintsen bele az MSDN-Help-be (érdeemes). Ezekben a "string"-kezelés további finomságairól is továbbiakat olvashat.

### **5.3 Példa: String műveletek gyakorlása**

Példánkban szerepeljen két szövegdoboz (TextBox), amelybe a szövegeket a felhasználó gépelheti be. A felhasználónak lehetősége van összehasonlítani a két TextBoxba írt szöveget vagy nagybetűssé illetve kisbetűssé konvertálni az első TextBoxba írt szöveget. A szövegrész gomb hatására kinyerhetünk az első TextBoxba írt szövegből a megadott karaktertől számított valahány karaktert. A karakter keresés gomb hatására az első TextBoxba írt szövegben megkeresi a második TextBoxba írt szöveg előfordulását és az eredményről üzenetablakban tájékoztat a program. A szöveghossz gomb hatására az első TextBoxba írt szöveg karaktereinek a számát írja ki a program szintén üzenetablakban. Az utolsó lehetőség a szövegbe beszúrás gomb, amellyel valahanyadik karaktertől szűrhatunk be egy a felhasználó által TextBoxba megadott szövegrészt.

A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását!

1. Hozzunk létre új projektet String néven!
2. Tervezzük meg az űrlapot az alábbiak szerint:

3. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Form	String	frmString	
Label	Gépelje be a szöveget!	lblSzoveg1	
Label	Gépelje be a másik szöveget!	lblSzoveg2	
Label	Hanyadik karaktertől?	lblHanyadik	
Label	Hány karakter?	lblMennyi	
Label	Hanyadik karaktertől?	lblMelyiktol	
Label	Mit szűrjön be?	lblMit	
TextBox		txtSzoveg1	
TextBox		txtSzoveg2	
TextBox	2	txtHanyadik	
TextBox	1	txtMennyi	
TextBox	2	txtMelyiktol	

TextBox	beszúr	txtMit	
Button	Számol	btnSzamol	
Button	Összehasonlít	btnHasonlit	
Button	Nagybetűsre alakít	btnNagybetus	
Button	Kisbetűsre alakít	btnKisbetus	
Button	Szövegrész	btnResz	
Button	Karakter keresés	btnKeres	
Button	Szöveghossz	btnHossz	
Button	Szövegbe beszúrás	btnBeszur	
Button	Kilépés	btnKilepes	

#### 4. String.cs:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace String
{
    public partial class frmString : Form
    {
        public frmString()
        {
            InitializeComponent();
        }

        private void btnKilepes_Click(object sender, EventArgs e)
        {
            Close();
        }

        private void btnHasonlit_Click(object sender, EventArgs e)
        {
            int azonose;
            string szoveg1 = txtSzoveg1.Text;
            string szoveg2 = txtSzoveg2.Text;
            azonose = szoveg1.CompareTo(szoveg2);
            if (azonose == 0)
                MessageBox.Show("A két beírt szöveg azonos!");
            else
                MessageBox.Show("A két beírt szöveg nem azonos!");
        }

        private void btnNagybetus_Click(object sender, EventArgs e)
        {
            if (txtSzoveg1.Text != "")

```

```

        MessageBox.Show("Első szöveg nagybetűs alakja=" +
txtSzoveg1.Text.ToUpper());
    else
        MessageBox.Show("Üres a szöveg mező!!!");
    }

    private void btnKisbetus_Click(object sender, EventArgs e)
    {
        if (txtSzoveg1.Text != "")
            MessageBox.Show("Első szöveg kisbetűs alakja=" +
txtSzoveg1.Text.ToLower());
        else
            MessageBox.Show("Üres a szöveg mező!!!");
    }

    private void btnResz_Click(object sender, EventArgs e)
    {
        string szoveg1=txtSzoveg1.Text;
        string
        resz=szoveg1.Substring(Convert.ToInt32(txtHanyadik.Text)+1,
Convert.ToInt32(txtMennyi.Text));
        MessageBox.Show(resz);
    }

    private void btnKeres_Click(object sender, EventArgs e)
    {
        string mondat=txtSzoveg1.Text;
        string szo = txtSzoveg2.Text;
        if (mondat.IndexOf(szo)>-1)
            MessageBox.Show("A "+szo+" szó szerepel a "+ mondat+"
mondatban.");
        else
            MessageBox.Show("A "+szo+" szó nem szerepel a "+
mondat+" mondatban.");
    }

    private void btnHossz_Click(object sender, EventArgs e)
    {
        string szoveg1=txtSzoveg1.Text;
        int darab=0;
        for (int i = 0; i < szoveg1.Length; i++)
        {
            darab++;
        }
        MessageBox.Show(txtSzoveg1.Text+" szó "+
Convert.ToString(darab)+" karakterből áll");
    }

    private void btnBeszur_Click(object sender, EventArgs e)
    {
        string szoveg1=txtSzoveg1.Text;

        szoveg1=szoveg1.Insert(Convert.ToInt32(txtMelyiktol.Text),txtMit.Text);
        MessageBox.Show("Az új szöveg a beszúrást után="+szoveg1);
    }
}

```



## 6 Iteráció

A programunk írása közben gyakran találkozunk olyan dologgal, hogy ugyanazt a tevékenységsort többször kell végrehajtanunk, ilyenkor használunk ciklusokat. Általában minden programozási nyelv, így a C# is az utasítások végrehajtására több lehetőséget kínál. Ezek az előírt lépésszámú és feltételes ciklusok. A feltételes ciklusok kétfélék lehetnek – előltesztelő vagy hátultesztelő - a feltétel helyétől függően.

### 6.1 Előírt lépésszámú ciklusok

A **”for” ciklust**, akkor használjuk, ha pontosan meg tudjuk mondani, hogy egy adott tevékenységet hányszor kell ismételni. Szintaxisa:

```
for (ciklusváltozó=kezdőérték; végérték vizsgálat; léptetés)  
    utasítás;
```

A **”for”** utasítás a ciklus feje, ahol megadjuk a ciklusváltozót, annak kezdőértékét, a végérték vizsgálatát illetve a léptetés mértékét.

A ciklusváltozó, a kezdőérték és a végérték csak sorszámozott típusú lehet. Itt kell megjegyeznünk, hogy az **”utasítás”** helyén összetett utasítás is állhat, ebben az esetben használjuk a korábban megismert **”{...}”** utasítás párt.

Az alábbi példában 1 és 3 közötti egész számokat jelenítünk meg szövegdobozokban egymásután.

```
for (int i = 1; i < 4; i++)  
    MessageBox.Show(Convert.ToString(i));
```

A ciklus fejében létrehozott változók csak a ciklus blokkjában érvényesek, azaz azon kívül nem érhetők el. Abban az esetben, ha ezeket a változókat illetve azok értékeit a cikluson kívül is fel akarjuk használni, akkor a cikluson kívüli változóban kell eltárolni az értékeiket vagy pedig másik típusú ciklust kell használni.

A **”foreach”** ciklus a **”for”** ciklus egy speciális esetre módosított változata. A gyakorlatban akkor használjuk, ha valamilyen összetett adattípus (pl. tömb) elemeit egytől egyig fel akarjuk dolgozni.

```
string[] nevek;  
nevek = new string[3];  
nevek[0] = "ELSŐ";  
nevek[1] = "MÁSODIK";  
nevek[2] = "HARMADIK";  
foreach (string nev in nevek)  
    MessageBox.Show(nev.ToUpper());
```

A fenti példában egy neveket tartalmazó tömb (nevek) elemeit nagybetűs formában kívánjuk megjeleníteni, külön-külön üzenetablakokban.

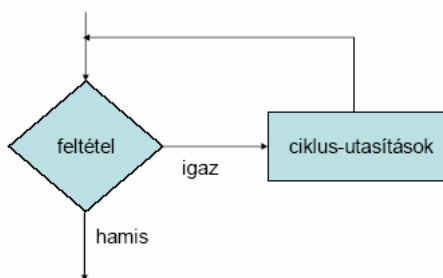
## 6.2 Feltételes ciklusok

A feltételes ciklust akkor használjuk programunkban, ha nem ismerjük az ismétlések számát, ebben az esetben egy feltételtől tehetjük függővé az ismétlések számát. Attól függően, hogy a feltétel a ciklusba való belépéskor vagy kilépéskor vizsgáljuk, beszélhetünk előltesztelő illetve hátultesztelő ciklusról.

Az előltesztelő ciklus szintaxisa a következő:

```
while (feltétel)
{
    utasítás(ok);
}
```

A feltételt itt is csakúgy, mint a szelekciómál zárójelbe kell tenni.



3. ábra  
*Az előltesztelő ciklus működése*

A **”while”** utasításban szereplő belépési feltétel vezérli a ciklus végrehajtását. A ciklusmag (utasítás vagy utasításblokk) csak akkor hajtódik végre, ha a feltétel igaz. Nagyon fontos megjegyezni, hogy a ciklusmagban kell gondoskodnunk arról, hogy a feltételt előbb-utóbb hamisra állítsuk, mert egyébként végtelen ciklusba jut programunk (soha nem tud kilépni a feltétel igaz volta miatt a ciklusból).

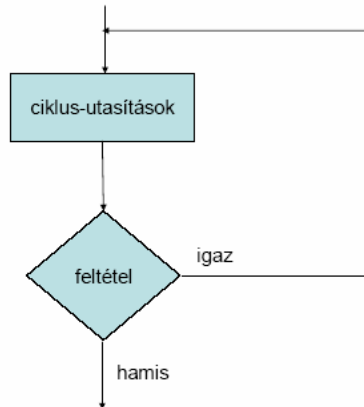
Ha a feltétel a ciklusba való belépés előtt hamissá válik, akkor a ciklus magja egyszer sem hajtódik végre.

Nézzük meg a **”for”** ciklusnál bemutatott példát előltesztelő ciklussal.

```
int i = 1;
while (i < 4)
{
    MessageBox.Show(Convert.ToString(i));
    i++;
}
```

A hátultesztelő ciklus szintaxisa a következő:

```
do
{
    utasítás(ok);
}while (feltétel)
```



4. ábra  
*A hátultesztelő ciklus működése*

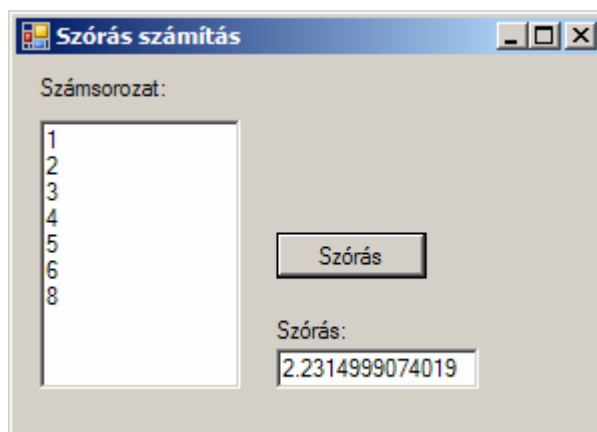
A hátultesztelő ciklus esetén a ciklusmag egyszer mindenképpen lefut, hiszen a feltétel vizsgálata és kiértékelése csak a ciklus végén történik meg. A ciklusmag mindaddig végrehajtott, amíg a feltétel igaz. A ciklusmagban itt is gondoskodnunk kell arról, hogy a feltételt előbb-utóbb hamisra állítsuk, mert egyébként végtelen ciklusba jut programunk. Nézzük meg az előbbi példát hátultesztelő ciklussal:

```
int i = 1;
do
{
    MessageBox.Show(Convert.ToString(i));
    i++;
} while (i < 4);
```

A gyakorlatban előfordulhat az az eset is, amikor egy iterációból hamarabb kell kilépni, minthogy az végigfutott volna vagy a végrehajtást valamilyen ok miatt a ciklus belsejéből a ciklusfejre kell léptetni azonnal. Ezekre a **"break"** és a **"continue"** utasítás szolgál a C# programozási nyelvben. A **"break"** hatására a program az iterációt követő utasításon folytatódik, míg a **"continue"** hatására nem lépünk ki a ciklusból, a végrehajtás a ciklusfejre kerül vissza.

Példa: Szórás számítása

Készítsünk programot, mely meghatározza egy számsorozat szórását!



Az adatok szóródásáról a legpontosabb képet úgy kaphatjuk, ha az összes adatot – és azoknak az átlagtól való eltérését – figyelembe vesszük. Ennek leggyakrabban használt módszere szórásnégyzet ( $s^2$ ) meghatározása. Ennek négyzetgyöke a szórás ( $s$ ):

$$s^2 = \frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_n - \bar{x})^2}{n}$$

$$s = \sqrt{\frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_n - \bar{x})^2}{n}}$$

A szórás meghatározásának lépései:

- Adatok kiolvasása a szövegdobozból
- Átlag kiszámítása.

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

- Az átlagtól való eltérések négyzetének kiszámítása, valamint négyzetösszeg képzése.

$$eno = \sum_{i=1}^n (x_i - \bar{x})^2$$

- Az átlaguk kiszámolása, majd ezen átlag négyzetgyökének kiszámolása.

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}} = \sqrt{\frac{eno}{n}}$$

- A kiszámolt szórás megjelenítése a másik szövegdobozban.

1. Hozzuk létre az űrlapot a megfelelő vezérlőkkel!

Komponens	Name	Text	Properties
Form	Form1	Szórás számítás	

Button	btSzamitas	Szórás	
TextBox	tbSzamsor		Multiline=True
TextBox	tbSzoras		
Label	label1	Számsorozat:	
Label	label2	Szórás:	

## 2. A gomb kattintás eseményéhez írjuk meg a számítást elvégző kódot!

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace szoras
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btSzamitas_Click(object sender, EventArgs e)
        {
            //Határozzuk meg a szövegdoboz sorainak számát!
            int n = tbSzamsor.Lines.GetLength(0);

            //Hozzunk létre egy n darab double elemből álló tömböt!
            double[] szamsor;
            szamsor = new double[n];

            //A szövegdoboz sorait számmá alakítva olvassuk a tömbbe!
            for (int i = 0; i < n; i++)
            {
                szamsor[i] = int.Parse(tbSzamsor.Lines[i]);
            }

            //Számítsuk ki az összeget és az átlagot!
            double osszeg=0;
            for (int i = 0; i < n; i++)
            {
                osszeg += szamsor[i];
            }
            double atlag = osszeg / n;

            //Számítsuk ki az átlagtól való négyzetes
            //eltérések összegét!
            //A System.Math.Pow(alap,kitevő) függvénnyel tudunk
            //hatványozni.
            double eno=0;
            for (int i = 0; i < n; i++)
            {
                eno += System.Math.Pow((szamsor[i] - atlag),2);
            }
        }
    }
}
```

```
        //Az az átlagtól való négyzetes eltérések
        //átlagából gyököt vonva megkapjuk a szórást.
        //System.Math.Sqrt() függvény segítségével vonunk gyököt.
        double szoras = System.Math.Sqrt(eno / n);

        tbSzoras.Text = szoras.ToString();
    }
}
```

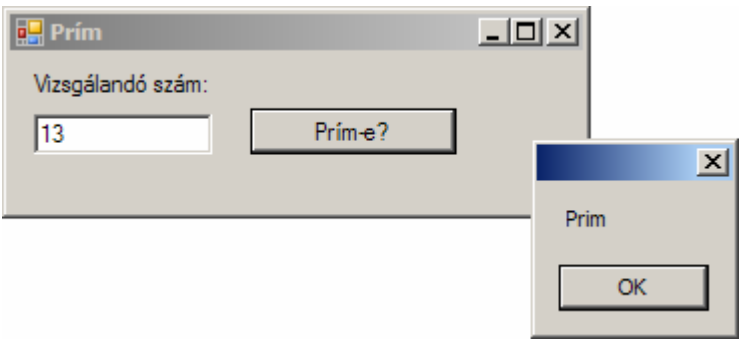
6.3 Példa: Prímszám vizsgálat

Készítsünk programot, mely egy számról eldönti, hogy prímszám-e. A vizsgálat eredményét felugró ablakban jelenítsük meg.

Annak eldöntésére, hogy egy szám prímszám-e, több algoritmus is létezik. Válasszuk az egyik legegyszerűbb módszert:

- Feltételezzük a számról, hogy prím
- [2 ; szám/2[ tartományba eső összes egész számmal megkíséreljük az osztást. Ha találunk olyan számot, melynél az osztási maradék 0, a szám nem prím. (A vizsgálandó szám felénél nagyobb számokkal felesleges próbálkozni, biztos, hogy közöttünk nem találunk egész osztót.)
- Ha a fenti tartományból egyetlen szám sem ad 0-t osztási maradékként, a szám prím.
- A C# nyelvben osztási maradék képzésére a % operátor szolgál.

1. Készítsük el az űrlapot



Komponens	Name	Text	Property
Form	Form1	Prím	
TextBox	tbSzam		
Button	btPrime	Prím-e?	
Label	Label1	Vizsgálandó szám:	

2. Rendeljük eseményt a gomb kattintásához, és végezzük el a vizsgálatot

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace primszam
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btPrime_Click(object sender, EventArgs e)
        {
            //Hozzunk létre int típusú változót, melybe beolvassuk
            //a szövegdobozból a vizsgálandó értéket.
            //A szövegdoboz string típusú tartalmát a int.Parse(szöveg)
            //függvénnyel alakíthatjuk számmá.
            int szam = int.Parse(tbSzam.Text);

            //prim-e logikai változóban kezdetben feltételezzük,
            //hogyan a vizsgált szám prim.
            bool prim_e = true;

            //Ciklussal végigjárjuk a lehetséges osztókat.
            for (int i = 2; i < szam / 2; i++)
            {
                //Ha az osztási maradék 0, a szám nem prim.
                if (szam % i == 0)
                {
                    prim_e = false;

                    //Ha találtunk osztót, felesleges tovább
                    //folytatnunk a keresést, a break; paranccsal
                    //kiugrunk a ciklusból.
                    break;
                }
            }

            //Az eredménytől függően megjelenítjük az üzenetet.
            if (prim_e == true)
            {
                MessageBox.Show("Prim");
            }
            else
            {
                MessageBox.Show("Nem prim");
            }
        }
    }
}

```

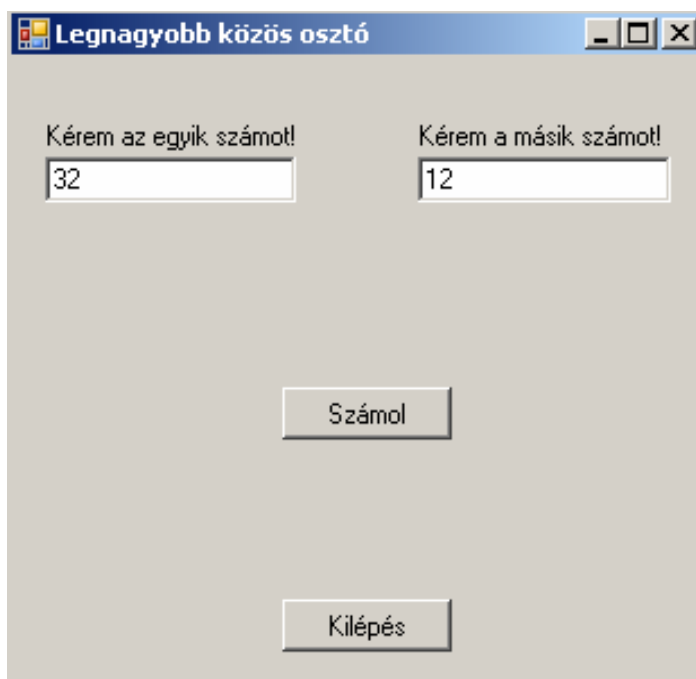
## 7 Saját metódusok létrehozása

### 7.1 Példa: Két szám legnagyobb közös osztójának a meghatározása

Példánkban szerepeljen két szövegdoboz (TextBox), amelyben az egyik illetve másik számot adhatja meg a felhasználó, amelyeknek a legnagyobb közös osztójára kíváncsi. Ha a „Számol” gombra kattintunk, ugorjon fel egy üzenetablak, melyben a program kiírja a felhasználó által megadott két szám legnagyobb közös osztóját.

A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását!

3. Hozzunk létre új projektet Lnko néven!
4. Tervezzük meg az űrlapot az alábbiak szerint:



5. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Form	Legnagyobb közös osztó	frmLnko	
Label	Kérem az egyik számot!	lblEgyik	
Label	Kérem a másik számot!	lblMasik	
TextBox	32	txtEgyik	



TextBox	12	txtMasik	
Button	Számol	btnSzamol	
Button	Kilépés	btnKilepes	

A programunkban létrehozunk Lnko néven (`int Lnko(int a, int b)`) egy függvény típusú metódust, amely két egész számot vár paraméterként és a legnagyobb közös osztójukat adja vissza eredményként. A metóduson belül ellenőrzést végzünk, ha a felhasználó negatív értéket ad meg, akkor veszi annak abszolút értékét és azzal számol. A két megadott számot összehasonlítja és amennyiben az első kisebb, úgy felcseréli őket, a könnyebb számolhatóság kedvéért. A `”%”` operátor az osztás maradékát adja vissza.

## 6. Lnko.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Lnko
{
    public partial class frmLnko : Form
    {
        public frmLnko()
        {
            InitializeComponent();
        }

        int Lnko(int a, int b)
        {
            int s;
            if (a < 0)
                a = -a;
            if (b < 0)
                b = -b;
            if (a < b)
            {
                s = b; b = a; a = s;
            }
            s = a % b;
            while (s != 0)
            {
                a = b; b = s; s = a % b;
            }
            return b;
        }

        private void btnSzamol_Click(object sender, EventArgs
e)
        {
            int egyik = int.Parse(txtEgyik.Text);
            int masik = int.Parse(txtMasik.Text);
            int oszt = Lnko(egyik, masik);
            string stroszt = oszt.ToString();
        }
    }
}
```

```
        MessageBox.Show("A két szám legnagyobb közös  
osztója: "+stroszto);  
    }  
  
    private void btnKilepes_Click(object sender, EventArgs  
e)  
    {  
        Close();  
    }  
}
```

## 8 Objektumosztályok és objektumok létrehozása

### 8.1 A programok szerkezete (osztály, adattag, metódus)

A C# programozási nyelv segítségével objektumorientált alkalmazásokat készíthetünk, sőt: a nyelv szigorúan objektumorientált - ami azt jelenti, hogy a programok készítése közben objektumokban kell gondolkodnunk, ebből pedig az következik, hogy sem adatok, sem metódusok (műveletek) nem létezhetnek objektumokon kívül.

#### 8.1.1 A program összetevői

Egy objektumorientált programot együttműködő objektumok halmaza alkot. Az objektumok adattagokból és metódusokból állnak. A gyakorlatban ahhoz, hogy egy működő program objektumokkal rendelkezzen, azokat a forrásprogramban az adott nyelv által előírt módon definiálnunk kell. Az objektumok definíciója (típusának leírása) az osztály. Az objektumokat az osztálydefiníciók alapján futási időben hozzuk létre. Az objektumokat az osztályok példányainak nevezzük.

Az osztálydefiníció minden esetben az adattagok és a metódusok leírását tartalmazza. Az adatok és a függvényjellegű metódusok azonosítójuk előtt típus-megjelölést is kapnak.

#### 8.1.2 Az osztály deklarációja

Az osztályokat a következő általános deklarációs formában szokás megadni:

```
class Osztály
{
    adattok deklarációja;
    metódusok deklarációja;
}
```

Az előbbi példából jól látható, hogy az osztálydefiníció egy blokkot alkot, a blokk kezdetét és végét a {} zárójelpár jelzi. Az osztály minden adattagját és metódusát a blokkon, azaz a zárójel-páron belül kell megadni.

#### 8.1.3 Adattagok deklarációja

Az objektum műveletei az objektumban deklarált adattagokon dolgoznak. Az adatok változóknak helyezhetők el. A változók deklarálásakor el kell döntenünk, hogy milyen értékeket vesznek majd fel és ezt a típusmegjelöléssel definiálni kell. A változók deklarálásának általános módja, amint ezt már a korábbiakban láttuk:

```
típus adattag_azonosító = kezdőérték;
```

Az adattagok deklarálásakor kétféle módosítót lehet használni, a **"static"** és a **"const"** kulcsszavakat. A **"static"** módosítóval az osztályszintű adattagot hozhatjuk létre. Az így deklarált statikus adattag értéke nem az osztály alapján létrehozott objektumban, hanem az osztályban kerül tárolásra, így beállítható illetve lekérdezhető az objektum létrehozása nélkül is.

Nézzünk meg egy példát az osztályszintű adattag deklarálására:

```
static double oVáltozó = 211.12;
```

A **"const"** módosítóval olyan adattagokat deklarálhatunk, amelyeknek értéke később már nem módosítható.

Nézzünk meg egy példát az állandó adattag deklarálására:

```
const név = "Visual C#";
```

Az adattagokra általában a programjainkban az **"osztálynév.adattagnév"**, illetve az **"objektumváltozó.adattagnév"** formában tudunk hivatkozni.

#### 8.1.4 Metódusok deklarálása

Az objektumok adatainak dolgozó eljárásokat és függvényeket (metódusokat) is az osztályokon belül deklaráljuk, ennek általános formája:

Eljárásjellegű metódus esetén:

```
void Eljárásjellegű_metódusnév (paraméterlista)
{
    eljárás belső változóinak deklarálása;
    utasítások;
}
```

Függvény-metódus esetén

```
típus Függvényjellegű_metódusnév (paraméterlista)
{
    függvény belső változóinak deklarálása;
    utasítások;
}
return érték;
}
```

A függvényjellegű metódus azonosítója előtti típus megadásakor kell meghatározni, a függvény által eredményként visszaadott adat típusát. Az eredmény visszaadását a **"return"** utasítás biztosítja, amelyet a függvény utasításrészében kell elhelyezni. Ezen utasítás hatására a függvény végrehajtása véget ér és az érték visszaadódik a függvény hívásának helyére, mint eredmény. Fontos felhívni arra a programozó figyelmét, hogy a függvényjellegű metódusok írása során, ha szelekciót alkalmazunk, akkor valamennyi ágon el kell helyezni a **"return"**

utasítást, azaz egyetlen feltétel esetén sem léphetünk ki úgy a metódusból, hogy ne adna vissza értéket.

A paraméterlista, amelyet mindig zárójelek között adunk meg adatokat küldhetünk a metódusunkba, sőt akár vissza is kaphatunk onnan eredményeket (de csak függvényjellegű metódusok esetén).

A metódusok deklarálásakor alkalmazhatjuk az adattagoknál már megismert **"static"** módosítót. Használatával osztályszintű metódust hozhatunk létre. Az ilyen metódus az osztályból létrehozott objektumokban nem használható, csak osztályszinten létezik.

A metódusokra az **"osztálynév.metódusnév"** vagy az **"objektumváltozó.metódusnév"** formában hivatkozhatunk.

### 8.1.5 Metódusok paraméterezése

A metódusok nem csak osztályaik vagy saját belső adattagjaikkal dolgoznak, hanem kívülről is kaphatnak adatokat. Minden metódus rendelkezhet paraméterekkel, amelyeket a fejlécében definiálunk, mint formális paraméterlista.

A paraméterek az adatok átadása szempontjából lehetnek érték vagy cím szerinti paraméterek. Érték szerinti paraméterátadásnál a bejövő adat csak felhasználható. Az alábbi példában egy metódus a kapott adatokkal számol.

```
class Param1
{
    public static void Main()
    {
        Account haszon = new Account();
        haszon.Count("Alma",152,3.5,15);
    }
}
```

Ha a metódustól eredményt szeretnénk visszakapni, akkor függvényjellegű metódust alkalmazunk. A függvény **"return"** parancsa azonban csak egy értéket képes visszaadni. A gyakorlatban az az eset is előfordulhat, hogy több értéket is szeretnénk visszakapni. Ekkor használjuk a címszerinti paraméterátadást.

```
class Param2
{
    public static void Main()
    {
        double netto = 0;
        double brutto = 0;
        Account haszon = new Account();
        haszon.Count(152,3.5,15,ref netto, ref brutto);
    }
}
```

A fenti példából jól látható, hogy a címszerinti paraméterátadásnál a "ref" direktívát alkalmaztuk mind az aktuális, mind pedig a formális paramétereknél. Észre kell venni, hogy bár a "netto" és "brutto" változókat adatvisszaadásra használjuk, a metódushívás előtt valamilyen értéket kell kapniuk, ami jelen esetben 0.

A most bemutatandó másik példában a "netto" és "brutto" értékek esetében deklaráljuk, hogy ezek a változók kifejezetten csak visszatérő adatok ("out").

```
class Param3
{
    public static void Main()
    {
        double netto;
        double brutto;
        Account haszon = new Account();
        haszon.Count(152, 3.5, 15, out netto, out brutto);
    }
}
```

Az "out" jelölés azt is szükségtelessé teszi, hogy a formális paraméternek a metódushívás előtt valamilyen korábban kapott értéke legyen. A metódusok paraméterezésénél alkalmazhatunk előre nem definiált számú paramétert is.

### 8.1.6 Adatok és metódusok láthatósága

Az objektumorientált programozás egyik alapelve az, hogy az objektumok zárt rendszerben működnek. Ez azt jelenti, hogy egy objektumot csak az ún. interfészén keresztül lehet megszólítani; az interfész pedig gondosan megválogatott metódusokat tartalmaz. Az objektumok adatait és metódusait ezért különböző láthatósági megszorításokkal látjuk el.

Az osztálydefiníciókban az alábbi láthatósági szinteket használhatjuk:

- **private**, csak az osztályban, vagy a belőle közvetlenül létrehozott objektumokban elérhető illetve használható adattag vagy metódus. **(alapértelmezett)**
- **protected**, csak az osztályban, vagy a belőle származtatott osztályokban, és az ezekből létrehozott objektumokban elérhető illetve használható adattag vagy metódus.
- **public**, bárhonnan elérhető illetve használható adattag vagy metódus.

A láthatósági szintek alkalmazásával az objektumok adattagjait és az objektumon belül használt metódusait elrejthetjük a külvilág elől. Az adattagok másik objektum általi lekérdezését vagy beállítását olyan metódusokkal biztosítjuk, melyek kódja ellenőrzött hozzáférést biztosít az objektum belsejéhez.

Ha az osztály valamely tagjaira nem adunk meg láthatósági megszorítást, akkor azokra az alapértelmezett "private" láthatóság vonatkozik.

Nézzünk meg egy példát az adattagok és metódusok láthatóságának beállítására:

```
public class Osztály
{
    private int i = 5;
```

```

        protected c = "Karakterlanc"
        private void Atlag()
        .....
    }

```

Amint az a fenti példából látható láthatósági megszorítás magára az osztályra is beállítható, ennek az objektumok öröklődésénél lehet fontos szerepe.

### 8.1.7 A property-k

Az előző fejezetekben már volt szó az objektumok zártságáról, a belső adattagok metódusokon keresztül való eléréséről. Ebben az esetben azonban az adat-hivatkozás helyett metódusokat kell meghívunk. A property-k alkalmazása viszont, úgy teszi lehetővé az objektum belső adattagjaira való hivatkozást, hogy közben megtarthatjuk a hivatkozás adat-jellegét is.

```

private int ev;           //kívülről nem elérhető adat
public int Ev            //kívülről elérhető property
{
    get
    {
        return ev;
    }
    set
    {
        ev = value;
    }
}

```

A fenti kódrészlet szerint egy osztály belső adattagként az "ev" változót tartalmazza. Az "Ev" property-n keresztül adathivatkozás-jelleggel írhatjuk és olvashatjuk is:

```
Osztaly.Ev = 2004;
```

Ebben az esetben, a háttérben végrehajtásra kerül az "Ev" property "set" metódusa, és megtörténik az értékadás az "ev" belső változóba. A metódus természetesen bármilyen további, esetleg ellenőrzést szolgáló kódot is tartalmazhat még ezen felül.

Az év változó tartalmát például a következő utasítással is lekérdezhethetjük:

```
x = Osztaly.Ev;
```

Ez utóbbi esetben a "get" metódus hajtódik végre, visszaadva (esetleg átalakítva, feldolgozva) az év értékét. Ha szöveges formában is (kettőezernégy) meg szeretnénk kapni az évszámot, készíthetünk még egy property-t:

```

public string SzovegesEv
{
    get
    {

```

```
        return Szoveg(ev);  
    }  
}
```

A SzovegesEv property-n keresztül csak lekérdezhetjük az évet, mivel nincs **”set”** metódus. Az eredményt viszont most szöveggént kapjuk meg, abban az esetben, ha létezik a **”Szoveg”** függvény, amely az évszámot szöveggé konvertálja.

A property-k azért hasznosak, mert használatukkal

- beállíthatjuk, hogy a belső adatok írhatók és olvashatók, csak olvashatók vagy csak írhatók legyenek,
- a **”get”** és **”set”** metódusok ellenőrző, átalakító, feldolgozó kódok beiktatását is lehetővé teszik,
- adatként lekérdezhetünk olyan adattagokat, amelyek belső adattagként esetleg nem is léteznek,
- a **”get”** metódus végül is bármilyen számítási, feldolgozási művelet eredményét visszaadhatja.



## 9 Grafika

A C# környezet hatékony eszközöket kínál grafikák készítésére. A rajzolás első ránézésre nem tűnik egyszerűnek, de néhány példa után könnyen áttekinthetővé válik.

Windows rendszerben mindenért, amit a képernyőre rajzolunk a GDI (Graphics Device Interface) felelős. A windows-os programok, és maga a Windows is ezen a kódon keresztül rajzolja a szövegeket, vonalakat, grafikákat, stb. a képernyőre. A GDI közvetlen kapcsolatban áll a grafikus kártya meghajtó programjával, és eldönti, hogy a kívánt rajz hogyan jeleníthető meg legjobban a rendelkezésre álló grafikus eszközön. Például, ha a grafikus kártyán beállított színmélység kisebb a megjelenítendő grafika színmélységénél, a GDI gondoskodik az optimális helyettesítő színek megválasztásáról. A programozónak ezért nem kell azzal törődnie, hogy milyen a rendelkezésre álló grafikus megjelenítő eszköz.

A C# kód a `Graphics` osztályon keresztül kommunikál a GDI-vel.

### 9.1 A Graphics osztály

Mielőtt elkezdenénk rajzolni, létre kell hoznunk egy `Graphics` objektumot. Szerencsére minden grafikus windows vezérlő automatikusan öröklí a `CreateGraphics()` metódust, melyen keresztül hozzá tudunk férni a vezérlő grafikus részéhez.

Példának hozzuk létre az űrlapon egy gombot, és kezeljük le a kattintás eseményét! Az eseményhez rendeljük a következő kódot:

```
private void button1_Click(object sender, EventArgs e)
{
    Bitmap kep;
    kep = new Bitmap(@"c:\kép.jpg");

    Graphics grafika;
    grafika = this.CreateGraphics();
    grafika.DrawImage(kep, 0, 0);
}
```

1. Az első két sor létrehozza a `Bitmap` osztály egy példányát `kep` néven, majd a konstruktornak paraméterként átadjuk a betöltendő kép elérési útvonalát.
2. Ezután hozzuk létre a `Graphics` osztály egy példányát `grafika` néven. `Grafika` nevű objektumunkhoz most rendeljük hozzá az aktuális űrlap (`this`) területét. Itt az űrlap helyett választhatnánk bármely vezérlőt az űrlapon pl:

```
grafika = this.TextBox1.CreateGraphics();
```

3. Ezután már csak a kép felrajzolása marad hátra a `DrawImage()` metóduson keresztül a (0,0) koordinátájú ponttól. A koordinátarendszer origója a bal felső sarokban van, a tengelyek jobbra, illetve lefelé mutatnak.

A példa szemléletes, de nem működik tökéletesen. Ha a Windows-nak újra kell rajzoltatnia ablakunkat, az általunk elhelyezett grafikát nem fogja újrarajzolni. Ha letesszük az ablakot a tálcára, majd vissza, a grafika eltűnik. Ugyanez történik, ha egy másik ablak egy időre fedi a miénket. Ha a fenti programsorokat az űrlap `Paint` eseményéhez rendeljük, a rajzolás megtörténik minden alkalommal, amikor az ablak többi elemét újra kell rajzolni.

## 9.2 A színek (Color)

Színválasztásra három lehetőségünk kínálkozik:

1. Választunk a `System.Drawing.Color` felsorolás előre definiált színei közül. Ezek a színek angol nevei: `pl`, `Red`, `Green`, `DarkBlue`. A teljes felsorolás értelmetlen lenne, a komplett lista megjelenítéséhez használjuk az automatikus kódkiegészítőt! (A „color.” legépelése után `ctrl+betűköz` billentyűk lenyomásával megjelenő lista.)

Pl: `Color.Orange`

2. A rendszerszínek közül választunk. Az asztal beállításai között a felhasználó át tudja állítani a Windows színeit. Ha azt szeretnénk, hogy grafikánk kövesse a felhasználó beállításait, a `System.Drawing.SystemColors` felsorolás színeire hivatkozunk! (Pl: `Window` – alak színe ; `Highlight` – szövegkijelölés háttérszíne ; `HighlightText` – szövegkijelölés szövegszíne ; `Desktop` – asztal színe , stb.)

Pl: `SystemColors.WindowText`

3. Saját színt definiálunk, melynek megadjuk a piros, kék és zöld összetevőit. A `Color` osztály `FromArgb()` metódusa három byte típusú paraméterrel rendelkezik, melyeken keresztül beállíthatjuk a három összetevőt. A piros színt a következőképpen adhatjuk meg:

Pl: `Color.FromArgb(255, 0, 0);`

## 9.3 A toll (Pen)

Míg `graphics` osztály biztosítja számunkra a felületet, amire rajzolhatunk, szükségünk van egy eszközre, amivel rajzolunk. Hozzuk létre a `Pen` osztály egy példányát, és állítsuk be tollunk jellemzőit: adjuk meg a vonal színét, vastagságát és stílusát! A `Pen` konstruktorának első paramétere a vonal színe, ezután következik a vonal vastagsága.

```

Pen toll;
toll = new Pen(Color.DarkBlue, 3);
toll.DashStyle
System.Drawing.Drawing2D.DashStyle.Dot;
=

```

A `System.Drawing.Drawing2D.DashStyle` felsorolás tartalmazza a rendelkezésünkre álló vonalstílusokat, ezek közül választhatunk. Ha nem adunk meg külön vonalstílust, a vonal folytonos lesz.

Érték	Leírás
Dash	-----
DashDot	.-.-.-.-
DashDotDot	...-.-.-
Dot	.....
Solid	Folytonos
Custom	Egyedi. A <code>Pen</code> osztály rendelkezik olyan tulajdonságokkal, melyeken keresztül egyedi vonalstílus is kialakítható. A részletekre itt nem térünk ki.

A rajzoljunk egy vonalat, ellipszist és egy téglalapot! A `DrawEllipse()` metódus utolsó négy paramétere az ellipszist befoglaló téglalap bal-felső sarkának x és y koordinátája, valamint a téglalap szélessége és magassága:

```

Pen toll;
toll = new Pen(Color.DarkBlue, 3);
toll.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;

grafika.DrawLine(toll, 10, 10, 100, 100);

grafika.DrawEllipse(toll, 10, 10, 100, 100);

grafika.DrawRectangle(toll, 10, 10, 100, 100);

```

## 9.4 Az ecset (Brush)

Míg a toll a vonalak jellemzőit tartalmazza, az ecset a kitöltendő alakzatok kitöltési tulajdonságait írja le. Hozzunk létre egy saját ecsetet és rajzoljunk vele:

```

Brush ecset;
ecset = new System.Drawing.SolidBrush(Color.DarkBlue);

grafika.FillEllipse(ecset, 10, 10, 100, 100);

```

### 9.4.1 Mintás ecset és toll (TexturedPen)

Létrehozhatunk olyan ecsetet, illetve tollat is, mely egy képből nyert mintával rajzol:

```

Bitmap kep;
kep = new Bitmap(@"c:\kép.jpg");

Brush mintasEcset;
mintasEcset = new System.Drawing.TextureBrush(kep);

Pen mintasToll = new Pen(MintasEcset, 30);
grafika.DrawRectangle(mintasToll, 10, 10, 100, 100);

```

## 9.5 Szöveg rajzolása

Szöveg rajzolásához két objektumot is létre kell hozni: egy betűtípust és egy ecsetet:

```

Font betu;
betu = new System.Drawing.Font("Arial", 30);

Brush ecset;
ecset = new System.Drawing.SolidBrush(Color.Black);

grafika.DrawString("Hello", betu, ecset, 10, 10);

```

## 9.6 Állandó grafikák létrehozása az űrlapon

Egyszerűbb alakzatok rajzolásánál járható út, hogy a rajzolást az űrlap objektum `Paint` eseményéhez kötve minden alkalommal elvégezzük, amikor az űrlap újrajzolja magát.

Számításigényes grafikák esetén a fenti megoldás nagyon lelassítja a programot. Ilyenkor létrehozhatunk egy `Bitmap` objektumot a memóriában, amire egyszer elég megrajzolni a grafikát. A `Paint` metódusban ezt az objektumot másoljuk az űrlapra.

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication5
{
    public partial class Form1 : Form
    {
        Bitmap rajzFelulet;

        public Form1()
        {
            InitializeComponent();

            //Létrehozzuk a bittérképet a konstruktorban
            //átadott méretben
            rajzFelulet = new Bitmap(200,200);

            //Létrehozzunk a Graphics osztály egy példányát,

```

```

        //melyen keresztül rajzolhatunk a rajzfelületre
        Graphics grafika;
        grafika = Graphics.FromImage(rajzFelulet);

        Brush ecset;
        ecset = new System.Drawing.SolidBrush(Color.DarkBlue);

        grafika.FillEllipse(ecset, 10, 10, 100, 100);
    }

    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        Graphics grafika;
        //A grafika objektum űrlapunk-ra (this) mutat
        grafika = this.CreateGraphics();

        //A DrawImage metódussal felrajzoljuk a rajzFelulet
        //az űrlapra
        grafika.DrawImage(rajzFelulet, 0, 0);

        //Megszüntetjük a grafika objektumot
        grafika.Dispose();
    }
}

```

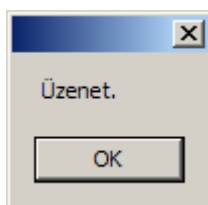
## 10 Többablakos alkalmazások készítése

### 10.1 Üzenetablakok

A MessageBox osztály segítségével üzenetablakot jeleníthetünk meg, mely szöveget, gombokat és egy ikont tartalmazhat. A MessageBox Osztály új példányát nem hozhatjuk létre, az üzenetet a statikus MessageBox.Show() metódus segítségével jeleníthetjük meg. A metódust több paraméterezéssel is hívhatjuk.

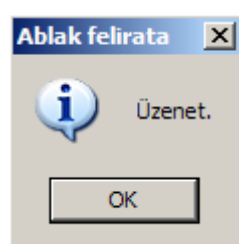
Ha csak egyetlen szöveges paramétert adunk át, a megadott szöveg és egy „Ok” gomb jelenik meg:

```
MessageBox.Show("Üzenet.");
```



Paraméterezéssel megadhatjuk az ablak nevét, a szükséges gombokat és az üzenet típusát jelképező ikont is:

```
MessageBox.Show("Üzenet.", "Ablak felirata",  
    MessageBoxButtons.OK, MessageBoxIcon.Information);
```











A MessageBoxButtons felsorolás tagjaival adhatjuk meg, milyen gombok jelenjenek meg a felugró ablakon. A gombok felirata a Windows nyelvi beállításaitól függ. A lehetséges kombinációkat az alábbi táblázatban foglaljuk össze:

Tag neve	Megjelenő gombok
AbortRetryIgnore	
OK	Ok
OKCancel	Ok és Mégse
RetryCancel	Ismét és Mégse

YesNo	Igen és Nem
YesNoCancel	Igen, Nem, Mégse

A MessageBoxIcon felsorolás tagjaival az ablakon megjelenő ikont állíthatjuk be:

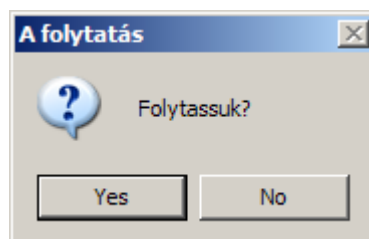
Tag neve	Ikon
Asterisk	
Error	
Exclamation	
Hand	
Information	
Question	
Stop	
Warning	

A MessageBox.Show() metódus visszatérési értékéből megtudjuk, hogy a felhasználó melyik gombbal zárta be az ablakot. A DialogResult felsorolás tagjai azonosítják a lehetséges gombokat (Abort, Cancel, Ignore, No, OK, Retry, Yes).

```

DialogResult valasz;
valasz = MessageBox.Show("Folytassuk?", "A folytatás",
    MessageBoxButtons.YesNo, MessageBoxIcon.Question);
if (valasz == DialogResult.Yes)
{
    MessageBox.Show("Jó, folytatjuk");
}
else
{
    MessageBox.Show("Akkor abbahagyjuk...");
}

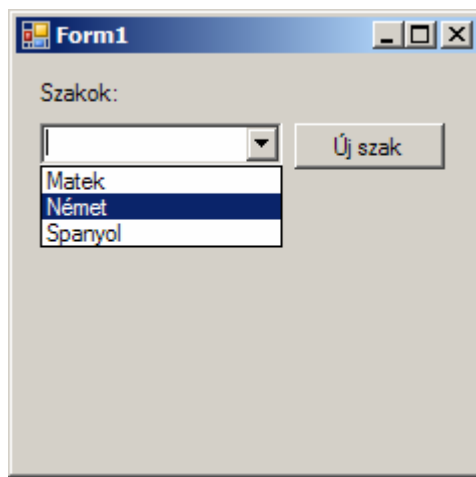
```



## 10.2 Példa: Kétablakos alkalmazás

Példánkban szerepeljen egy legördülő doboz (ComboBox), melynek elemeit az „Új szak” gombra kattintva bővíthetjük. Ha a gombra kattintunk, ugorjon fel egy ablak, melyben megadhatjuk az új szak nevét. A legördülő doboz csak akkor bővüljön az új elemmel, ha az ablakot OK gombbal zártuk be.

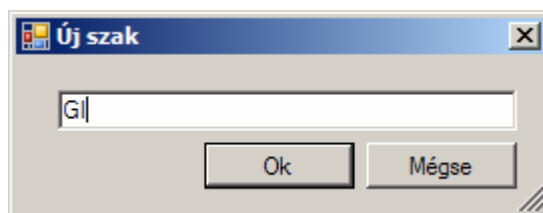
1. Hozzunk létre új projektet szaklista néven!
2. Tervezzük meg a főürlapot az alábbiak szerint:



3. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:
- 4.

Komponens	Text	Name	Properties
Label	Szakok:		
ComboBox		cbSzakok	
Button	Új szak	btUjSzak	

5. Készítsük el a felugró űrlapot, melyben megadhatjuk az új szak nevét:



6. Válasszuk a project / Add Windows Form menüpontot
7. A megjelenő sablonok közül a Windows Formra lesz szükségünk. Nevezzük az új űrlapot „UjSzak”-nak, majd kattintsunk az „Add” gombra!
8. Helyezzünk el egy szövegdobozt (TextBox) és két gombot (Button) az űrlapon!
9. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:



Komponens	Text	Name	Properties
Label	Új szak neve		
TextBox			Modifiers: public
Button	Ok	btnOk	DialogResult: OK
Button	Mégsem	btnMegsem	DialogResult: Cancel
Form	Új szak	UjSzak	AcceptButton: btnOk CancelButton: btnMegsem FormBorderStyle: FixedDialog MaximizeBox: False MinimizeBox: False ShowInTaskbar: False

- Ha szövegdozoz modifiers tulajdonságát public-ra állítjuk, kívülről is láthatóvá válnak a szövegdozoz tulajdonságai, így ki tudjuk olvasni annak tartalmát.
- Nem mindegy, hogy a felhasználó „Ok” vagy „Mégse” gombbal zárja be az ablakot. Ezt az információt a gombok DialogResult, illetve a Form AcceptButton és CancelButton tulajdonságainak értelem szerű beállításával tudjuk visszaadni a főúrlapnak. Ezeket lásd később.
- Rendeljük eseményeket az „Ok” és a „Mégse” feliratú gombokhoz!  
Mindkét gomb zárja is be az ablakot.

UjSzak.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication2
{
    public partial class UjSzak : Form
    {
        public UjSzak()
        {
            InitializeComponent();
        }

        private void btMegse_Click(object sender, EventArgs e)
        {
            Close();
        }

        private void btOk_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
}
```

```
}  
}
```

10. A szaklista űrlap gombjához rendeljünk eseményt, mely megnyitja az UjSzak űrlapot. Ha a felhasználó az Ok gombbal zárta be a felugró ablakot, adjuk hozzá a legördülő doboz tartalmához az új szakot!

```
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Text;  
using System.Windows.Forms;  
using System.Xml;  
  
namespace WindowsApplication2  
{  
    public partial class Form1 : Form  
    {  
        public Szaklista()  
        {  
            InitializeComponent();  
  
            private void btUjSzak_Click(object sender, EventArgs e)  
            {  
                //frmUjSzak néven létrehozuk az UjSzak  
                //az UjSzak osztály egy példányát  
                UjSzak frmUjSzak = new UjSzak();  
  
                //a ShowDialog metódussal megjelenítjük az új  
                //ablakot  
                if (frmUjSzak.ShowDialog() == DialogResult.OK)  
                {  
                    cboSzakok.Items.Add(frmUjSzak.tbSzak.Text);  
                }  
            }  
        }  
    }  
}
```

A `ShowDialog()` metódus hívásával az űrlapot úgynevezett „modal” ablakként jelenítjük meg. Ez azt jelenti, hogy amíg a felhasználó ezt az ablakot be nem csukja, az alkalmazás többi ablaka inaktív. A `ShowDialog()` metódus csak akkor tér vissza, amikor a felugró ablakot a felhasználó bezárta. (A `ShowDialog()` metódushívás után szereplő utasítások csak a felugró ablak bezárása után kerülnek feldolgozásra.) Visszatérési értékéből tudjuk meg, hogy a felhasználó melyik gombbal csukta be az ablakot. A felugró ablakban megadott értéket csak akkor adjuk hozzá a `cboSzakok` legördülő doboz elemeihez, ha a felhasználó Ok-val zárta be az ablakot.

A Form osztálynak létezik egy `Show()` metódusa is, mely nem modal ablakként jeleníti meg az űrlapot, és azonnal visszatér. Ezért visszatérési értékéből nem derül ki, hogy melyik gombbal zártuk be az ablakot.

## 11 Kivételkezelés

A C# programozási nyelv a futásidejű hibák kiküszöbölésére alkalmas eszközt is tartalmaz. Ennek az eszköznek a segítségével a programjainkba hibakezelő (más néven kivételkezelő) kódrészleteket is beépíthetünk. A kivételkezelő eljárásoknak programjainkban az a lényege, hogy elkerüljük a futásközben fellépő hibák során felbukkanó hibaüzeneteket, és megvédjük programjainkat a váratlan leállástól.

### 11.1 Hibakezelés

A programkészítés során még a gyakorlott programozóknál is előfordul, hogy valamilyen utasítást, operátort vagy egy műveleti jelet nem a C# programnyelv szabályai szerint írt le. A leggyakoribb hiba talán az, hogy az utasítás végéről lefelejtik a pontosvesszőt. Ilyenkor a program az előző utasítás folytatásaként kezdi el értelmezni a következő utasítást, amelynek eredménye fordítási hiba lesz. Ezek a hibák mivel a program fordítása során keletkeznek, így megakadályozzák a program futtatását.

A fordítóprogram által észlelt hibákat szintaktikai hibáknak nevezzük. Egy program addig nem futtatható, amíg szintaktikai hibát tartalmaz.

Előfordulhat olyan eset is, amikor a fordító csak valamilyen figyelmeztetést küld a programíró számára. A figyelmeztetések olyan hibára hívják fel a programíró figyelmét, amelyek nem akadályozzák meg a program futtatását, de esetleg rontják programunk hatékonyságát vagy esetleg csak rontják a forrásprogram áttekinthetőségét (pl. deklarálunk egy változót, de azt sehol nem használjuk a programunkban, azaz felesleges tárhelyet foglalunk!).

A szintaktikai hibák mellett a programunk logikai hibákat is tartalmazhat, de ez nincs befolyással a program futtatására (a program fut csak bizonyos input adatokra nem azt az outputot adja, amit kellene).

A program futása során fellépő (nem külső okból származó) hibákat szemantikai hibának nevezzük. Megelőzésükről a programkódban kell megfelelő ellenőrző, hibakezelő utasításokkal gondoskodni.

### 11.2 A try és catch

A "try" parancs segítségével a programok egyes részeihez tudunk hibakezelő eljárásokat rendelni, amelyek a futásközbeni hiba esetén a vezérlést az általunk megírt hibakezelő utasításokra adják át, ezzel kiküszöbölve a program leállítását.

A **”catch”** parancs segítségével azokat a kivételeket tudjuk elfogni, amelyek a **”try”** blokkban keletkeznek. Itt lehet meghatározni, milyen utasítások kerüljenek végrehajtásra, és hogyan kezeljük a felmerülő hibákat. A **”catch”** segítségével különböző hibák egyidejű kezelését is meg lehet valósítani.

Abban az esetben, ha nem használjuk a hibakezelő eljárásokat, a programunk leáll, és a hibakezelést vagy az operációs rendszer vagy pedig a .NET rendszer veszi át. Ennek általában az a következménye, hogy különböző (esetleg angol nyelvű) hibaüzeneteket ír a programunk a képernyőre, amely **”éles”** programok esetén elfogadhatatlan (nem várható el a felhasználatól, hogy a programozó hanyagsága miatt megtanuljon legalább alapszinten angolul). Fokozottan érvényes ez azokra a programokra, ahol a felhasználó adatbevitelt valósít meg, amint ez a következő példában történik.

```
int oszto=0;
double hanyados;
try
{
    hanyados=10/oszto;
}
catch (ArithmeticException ar)
{
    MessageBox.Show(Convert.ToString(ar));
}
```

Amennyiben az **”oszto”** változó értéke 0 a végrehajtás pillanatában, akkor a kivételkezelés működésbe lép, a **”catch”** blokkban elkapjuk a hibát és kiíratjuk üzenetablakban az okát.

A fenti programrészletben a **”try”** után kapcsos zárójelek közt adjuk meg a hibát okozható programrészletet. A hiba abban az esetben lép fel, ha az **”oszto”** értéke 0. A **”catch”** hibakezelő utasításait is kapcsos zárójelek közé tesszük, ezzel jelezve a fordítónak, hogy hol kezdődik és hol végződik a hibakezelés.

A **”catch”** blokkjában lehetőségünk van a különböző okok miatt keletkezett hibák szétválasztására, valamint a hiba típusának meghatározására is. A **”catch”** parancs a kivételt akár paraméterként is fogadhatja **catch(Exception e)**. A paraméterként megadott változó **”System.Exception”** típusú, amelyből ki lehet olvasni a hiba okát és a megfelelő hibakezelőt indíthatjuk el.

A képernyőn megjelenő hibaüzenet az esetek többségében nem túl beszédes, vagy ha igen akkor gyakran túlságosan sok, nehezen érthető információt tartalmaz. Természetesen a hibakezelés során nem az a célunk, hogy a felhasználót hosszú, számára értelmezhetetlen üzenetekkel terheljük, mert így gyakorlatilag ugyanazt tesszük, mint az eredeti hibaablak (amelyben megjelenik az eredeti hibaüzenet). Azt sem szabad elfelejteni, hogy a felhasználó tulajdonképpen nem is nagyon tudja kezelni a program futása közben fellépő hibákat, még akkor sem, ha kiíratjuk a képernyőre a hiba valamennyi paraméterét. A gyakorlatban sokkal

jobb megoldásnak számít, ha megállapítjuk a hiba okát, majd pedig a megfelelő hibakezelés aktivizálásával meg is szüntetjük azt, anélkül hogy a felhasználót nehéz helyzetbe hoznánk.

A gyakorlatban általában a **”catch”** blokkok sorrendje sem mindegy. A helyes megközelítés az, ha azokat a hibákat az általános hibák elé helyezzük, melyekre jó eséllyel számítani lehet a programban. Ilyen lehet például a felhasználói adatbekérésnél fellépő hiba vagy a matematikai műveleteknél előforduló nullával való osztás esete.

A kivétel típusokat a **”System”** névtérben (namespace) találhatjuk meg.

### 11.3 A finally blokk

A programunk írása során előfordulhat olyan eset is, hogy egy programrészlet hibás és hibátlan működés esetén is mindenképpen lefusson. A leggyakrabban a fájlkezelésnél találunk erre példát, mivel a megnyitott fájl hiba esetén is mindenképpen le kell zárni. Gondoljunk bele milyen hibát okozhatna, ha a fájl bezárása nélkül próbálnánk újra megnyitni a fájlt. Az ilyen típusú problémákra nyújthat megoldást a **”finally”** parancs.

A **”finally”** blokkban elhelyezett kód tehát minden esetben lefut – kivéve, ha a program végzetes hibával áll meg -, függetlenül az előtte keletkezett hibáktól. A következő példában a matematikai műveleteknél gyakran fellépő nullával való osztás esetét mutatjuk be a **”finally”** blokk használatával:

```
int osztó=0;
double hanyados;
try
{
    hanyados=10/osztó;
}
catch (ArithmeticException ar)
{
    MessageBox.Show(Convert.ToString(ar));
}
finally
{
    MessageBox.Show("A program ezen része mindenképpen lefut");
}
```

### 11.4 Kivételek feldobása

A C# programozási nyelv lehetőséget ad a programozó által definiált kivételek használatára is. A kivételek dobása a **”throw”** paranccsal történik.

```
throw (exception);
throw exception;
```

A program bármely szintjén dobhatunk a kivételt a **"throw"** parancs segítségével, és egy tetszőleges **"catch"** blokkal el is tudjuk kapni. Amennyiben nem kapjuk el sehol a programunkban, akkor az a Main() függvény szintjén is megjelenik, majd végül az operációs rendszer lekezeli a saját hibakezelő eljárásával, ami persze a legtöbbször a programunk leállításával jár.

## 11.5 Checked és unchecked

A C# programozási nyelv tartalmaz két további parancsot a kivételek kezelésére és a hibák javítására. Az egyik a **"checked"** a másik pedig az **"unchecked"** parancs. Amennyiben az értékadáskor a változóba olyan értéket kívánunk elhelyezni, amely az adott változó értéktartományában nem fér el, **"OverflowException"** hibával áll meg a programunk. Szerencsés esetben az ilyen jellegű hibákat el tudjuk kapni, a megfelelő `catch{}` blokk alkalmazásával, de az **"unchecked"** parancs használatával megakadályozhatjuk a kivétel keletkezését is, mivel ilyen esetben elmarad a hibaellenőrzés.

```
unchecked
{
    byte a=300;
}
```

A fenti példában a byte típusú változó maximum 255-ig tud értéket tárolni. Az értékadás ennek ellenére megtörténik, és a változóba bekerül a csonkított érték, persze csak akkora, amekkora még elfér benne.

A **"checked"** alkalmazásával pontosan az előbbi folyamat ellenkezőjét érhetjük el. Az ellenőrzés mindenképpen megtörténik, és kivétel keletkezik a hiba miatt.

A **"checked"** és **"unchecked"** parancsok nem csak blokként használhatóak, hanem egy kifejezés vagy értékadás vizsgálatánál is. Ekkor a következő alakban írhatjuk őket:

```
checked(kifejezés, művelet, értékadás);
unchecked(kifejezés, művelet, értékadás);
```

A bemutatott formában csak a zárójelek közé írt kifejezésekre, vagy egyéb műveletekre vonatkoznak. Alapértelmezés szerint a **"checked"** állapot érvényesül a programokban található minden értékadásra és műveletre. Csak nagyon indokolt esetekben használjuk ki az **"unchecked"** nyújtotta lehetőségeket, mert könnyen okozhatunk végzetes hibákat a programjainkban.

A kivételkezelés fejezetben leírtak alapján könnyen beláthatjuk, hogy a hibakezelés illetve a kivételek kezelése nagyon fontos és hasznos lehetőség a C# programozási nyelvben. Ezek nélkül nehéz lenne elképzelni hibátlanul működő programokat. Természetesen meg kell vallanunk, hogy a kivételkezelést alkalmazó programok sem tökéletesek, de jó esély van arra,

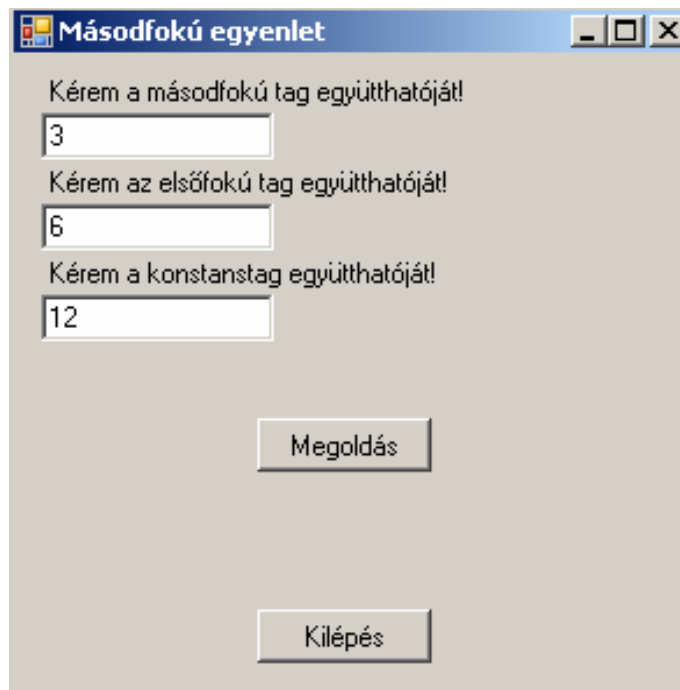
hogy programunk nem áll le végzetes hibával és nem keserítik sem a programozó, sem pedig a felhasználók életét.

### 11.6 Példa: Hibakezelés a másodfokú egyismeretlenes egyenlet kapcsán

Példánkban szerepeljen három szövegdoboz (TextBox), amelyben a felhasználó a másodfokú, majd az elsőfokú végül pedig a konstans tag együtthatóját adhatja meg. Ha a „Megoldás” gombra kattintunk, ugorjon fel egy üzenetablak, melyben a program kiírja a felhasználó által megadott másodfokú egyenlet gyökeit, amennyiben ez lehetséges. Ha 0-val való osztás történik, úgy erről tájékoztatjuk a felhasználót az üzenetablakban.

A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását!

1. Hozzunk létre új projektet Masodfoku néven!
2. Tervezzük meg az űrlapot az alábbiak szerint:



3. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Form	Másodfokú egyenlet	frmMasodfoku	
Label	Kérem a másodfokú tag együtthatóját!	lblMasodfoku	
Label	Kérem az elsőfokú tag	lblElsOfoku	



	együtthatóját!		
Label	Kérem a konstanstag együtthatóját!	lblKonstans	
TextBox	3	txtMasodfoku	
TextBox	6	txtElsofoku	
TextBox	12	txtKonstans	
Button	Megoldás	btnMegold	
Button	Kilépés	btnKilepes	

Amennyiben a gyökünk eredménye "végtelen", ami 0-val való osztás esetén lép fel, akkor kivételt dobunk `throw new DivideByZeroException()`, melyet a `catch()` részben kapunk el és kezelünk le. A felhasználót üzenetablakban tájékoztatjuk a nullával való osztás következtében fellépő hibáról!

#### 4. Masodfoku.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Masodfoku
{
    public partial class frmMasodfoku : Form
    {
        public frmMasodfoku()
        {
            InitializeComponent();
        }

        private void btnKilepes_Click(object sender, EventArgs e)
        {
            Close();
        }

        private void btnMegold_Click(object sender, EventArgs e)
        {
            int a = int.Parse(txtMasodfoku.Text);
            int b = int.Parse(txtElsofoku.Text);
            int c = int.Parse(txtKonstans.Text);
            double gyok1, gyok2;
            try
            {
                gyok1 = (-b + Math.Sqrt(b * b - 4 * a *
c) / (2 * a));
                gyok2 = (-b - Math.Sqrt(b * b - 4 * a *
c) / (2 * a));
                if (gyok1.ToString() == "végtelen")
                    throw new DivideByZeroException();
                else
                    MessageBox.Show("Gyök1=" + gyok1 + "
```

```

Gyök2=" + gyok2);
    }
    catch (DivideByZeroException divex)
    {
        MessageBox.Show("A másodfokú tag
együtthatója=0");
    }
}
}

```

## 12 Állománykezelés

A C# nyelv programozása során elsősorban a hagyományos C-nyelvből megismert fájlkezelési technikáit szokás használni. Abban az esetben, ha a fájl megnyitásakor valamilyen hiba lép fel, akkor a programunk többnyire valamilyen hibaüzenettel leáll. A C# programozási nyelv persze lehetővé teszi az állománykezelés során a be- és kiviteli műveleteknél fellépő műveletek hiba ellenőrzését. Ezzel a résszel itt nem kell foglalkoznunk, hiszen a kivételkezelésnél leírtak itt is érvényesek.

A programozási nyelvekben az állományokat tartalmuk alapján szöveges (text), típusos és típus nélküli fájlok csoportjára osztjuk. A szöveges fájlokat soros, míg a másik kettőt soros és közvetlen eléréssel egyaránt elérhetjük.

A számítógépen az operációs rendszer feladati közé tartozik a fájlrendszer támogatása és a fájlok elérésének biztosítása. Az állományok tartalmának eléréséhez a programozási nyelvtől és az operációs rendszer fajtájától függetlenül mindig ugyanazokat a főbb lépéseket kell végrehajtani. Nézzük meg melyek ezek!

- Előkészületek
- Az állomány megnyitása
- Az állomány tartalmának feldolgozása fájl műveletekkel (írás, olvasás, pozicionálás)
- A szükséges műveletek elvégzése után az állomány lezárása

Fájlkezelés, állomány fajták

- Szöveges
- Bináris
- Xml

### 12.1 Szöveges állományok kezelése

#### 12.1.1 Szöveges és bináris fájlok írása, bővítése, olvasása

A következő példában a "c:\temp" könyvtárban lévő "Teszt.txt" fájlt hozzuk létre a "File.CreateText" metódus segítségével, amennyiben még nincs ilyen nevű fájl a könyvtárban, ha igen, akkor nem történik semmi.

```
string utvonal = @"c:\teszt.txt";
if (!File.Exists(utvonal)) //ha nem létezik a fájl az adott könyvtárban,
csak akkor ír bele
{
    using (StreamWriter sw = File.CreateText(utvonal)) //Létrehoz egy
szöveg fájlt írásra
    {
```

```

        sw.WriteLine("Szia!");           //"Hello" szöveg írása a fájlba
        sw.WriteLine("Én vagyok az");    //"And" szöveg írása a fájlba
        sw.WriteLine("Megismersz?");    //"Welcome" szöveg írása a fájlba
    }
}

```

A következő részben már ellenőrzés nélkül írunk a fájlba a **"File.AppendText"** módszer segítségével. Amennyiben a fájl nem létezik akkor létrehozásra kerül, ezért nem szükséges az ellenőrzés.

```

using (StreamWriter sw = File.AppendText(utvonal))
{
    sw.WriteLine("Ezt már");
    sw.WriteLine("ellenőrzés nélkül");
    sw.WriteLine("írtam a fájlba!!!");
}

```

Ebben a programrészletben egy fájlt nyitunk meg olvasásra a **"File.OpenText"** módszer segítségével és addig olvasunk belőle, amíg **"null"** értéket nem olvasunk. A beolvasott adatokat kiírjuk üzenetablakban a képernyőre.

```

using (StreamReader sr = File.OpenText(utvonal))
{
    string utvonal = @"c:\teszt.txt";
    string s = "";
    while ((s = sr.ReadLine()) != null)
    {
        MessageBox.Show(s);
    }
}

```

### 12.1.2 Bináris fájlok írása, olvasása

Az előző részben már szóltunk a szöveges állományok kezeléséről. Vannak azonban olyan esetek, amikor egyszerűbb lenne a munkánk, ha nem kellene szöveges állományból illetve állományba konvertálni az adatainkat (pl. számok kezelése során). Előfordulhat olyan eset is, hogy nem praktikus a tároláshoz szöveges állományokat használnunk, mert egy külső egyszerű kis szövegszerkesztő programmal meg lehet nézni a tartalmát (pl. jelszavak tárolásánál, persze lehet kódolni őket, de ez már újabb programozási problémákat vet fel).

Az ilyen jellegű problémák feloldására találták ki a bináris állományokat, amelyek tartalmát nem lehet szövegszerkesztő programokkal megtekinteni illetve lehet bennük közvetlenül számokat tárolni, mindenféle konverzió nélkül. A bináris adatok jellemzője, hogy megtartják az adattípus eredeti tárolási formáját, azaz nem alakítják szöveggé.

A következő kis példaprogramunkban 10 darab egész számot tárolunk:

```

FileStream File = new FileStream(args[0], FileMode.CreateNew);

```

```

BinaryWriter bw = new BinaryWriter(File);
for (int i = 0; i < 10; i++)
{
    bw.Write(i);
}
bw.Close();
File.Close();

```

A 10. táblázatban a "FileMode" lehetséges értékeit mutatjuk be:

10. táblázat A FileMode értékei

Érték	Leírás
Append	Megnyit egy létező fájlt vagy újat készít.
Create	Új fájlt készít. Ha a fájl már létezik, a program törli és új fájlt hoz létre helyette.
CreateNew	Új fájlt hoz létre. Ha a fájl már létezik, kivételt vált ki, amit illik lekezelnie a programozónak.
Open	Megnyit egy létező fájlt.
OpenOrCreate	Megnyit egy fájlt, vagy ha nem létezik létrehozza.
Truncate	Megnyit egy létező fájlt és törli a tartalmát.

A "FileStream" objektumot létrehozása után azonnal fel kell készítenünk a bináris adatok fogadására. Ezt úgy tudjuk megoldani, hogy a BinaryWriter típust kapcsoljuk az objektumhoz.

```

FileStream File = new FileStream(FileMode.Create);
BinaryWriter bw = new BinaryWriter(File);
//A következő lépésben az objektumba közvetlenül tudunk írni
//adatokat, a Write tagfüggvénnyel
bw.Write(i);
//Ha befejeztük az írást a fájlba, akkor gondoskodnunk kell a
//megnyitott folyamat bezárásáról
bw.Close();
File.Close();
//Ha állományba kiírtunk valamit, akkor a fájlból olvasáskor
//előltesztelős ciklust használunk
FileStream File = new FileStream(FileMode.Open);
BinaryReader br = new BinaryReader(File);
lstLista.Items.Clear();
while (br.PeekChar() != -1)
{
    lstLista.Items.Add(br.ReadInt32());
}
br.Close();
File.Close();

```

while (br.PeekChar() != -1) sorban olvassuk be a BinaryReader osztály PeekChar() tagfüggvényével az adatfolyam következő karakterét. Ez a tagfüggvény mindig a következő karaktert adja vissza, egy kivételtől eltekintve, ha elértük az adatfolyam végét, ebben az esetben "-1"-et ad vissza. Az egész értékek beolvasásához a "ReadInt32()" tagfüggvényt

tudjuk felhasználni. A BinaryReader osztály rendelkezik a "ReadInt32"-höz hasonló tagfüggvényekkel a többi egyszerű adattípushoz is.

## 12.2 Példa: Véletlen számok generálása lottószámokhoz, szövegfájlba

Példánkban szerepeljen három rádiógomb (RadioButton) egy konténerben (GroupBox), amelyben a felhasználó kiválaszthatja, hogy ötös, hatos vagy skandináv lottóhoz kíván véletlen számokat generálni. Az első esetben 5 számot generálunk 90-ből, a másodikban 6-ot 45-ből, míg a harmadikban 7-et 35-ből. A generálást a generál gombra kattintással indítjuk, melyeket egy ComboBox-ban jelenítünk meg. A felhasználónak lehetősége van szöveges fájlba kiírni az adatokat igény szerint.

A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását!

5. Hozzunk létre új projektet Veletlenfajlba néven!
6. Tervezzük meg az űrlapot az alábbiak szerint:

7. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Form	Lottószámok	frmLotto	
GroupBox	Lottó	gbLotto	
RadioButton	Ötös lottó	rgOtos	
RadioButton	Hatos lottó	rgHatos	

RadioButton	Skandináv lottó	rgHetes	
ComboBox		cmbSzamok	
Button	Generál	btnGeneral	
Button	Fájlba ír	btnFile	
Button	Kilépés	btnKilepes	
saveFileDialog1		saveFileDialog1	

A feladat kódja tartalmaz egy eljárásszintű metódust, amellyel a különböző típusú lottószámokból tudunk véletlenszerűen generálni megadott darabszámot. Két bemenő paramétere van a tartomány felsőhatára, amelyből húzhatunk, valamint a kihúzható számok darabszáma (mindkettő egész típusú), `void lottoszamok(int szam, int darab)`.

Fel kell hívni a figyelmet még arra is, hogy fájlkezelés során az állomány elejére be kell szúrunk a következő sort, mert csak ezután lesz képes programunk a fájlok kezelésére.

```
using System.IO;
```

#### 8. Veletlenfajlba.cs:

```
using System;
using System.IO;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Veletlenfajlba
{
    public partial class frmLotto : Form
    {
        public frmLotto()
        {
            InitializeComponent();
        }

        private void btnFile_Click(object sender, EventArgs e)
        {
            saveFileDialog1.InitialDirectory = "c:\\\\";
            saveFileDialog1.Filter =
                "Szövegfájlok (*.txt) | *.txt | Mindenki
(*.*) | *.*";
            saveFileDialog1.FileName = "";
            if (saveFileDialog1.ShowDialog() ==
DialogResult.OK)
            {
                int i=0;
                TextWriter tw =
File.CreateText(saveFileDialog1.FileName);
                while (i < cmbSzamok.Items.Count)
                {
                    cmbSzamok.SelectedIndex = i;
                    tw.WriteLine(cmbSzamok.SelectedItem);
                    i++;
                }
                tw.Close();
            }
        }
    }
}
```

```

    }
}

void lottoszamok(int szam,int darab)
{
    byte veletlen;
    System.Random rnd = new System.Random();
    for (int i = 1; i <= darab; i++)
    {
        veletlen = (byte)rnd.Next(1, szam + 1);
        cmbSzamok.Items.Add(veletlen);
    }
}

private void btnGeneral_Click(object sender, EventArgs
e)
{
    cmbSzamok.Items.Clear();
    if (rgOtos.Checked == true)
        lottoszamok(90,5);
    if (rgHatos.Checked == true)
        lottoszamok(45,6);
    if (rgHetes.Checked == true)
        lottoszamok(35,7);
}

private void btnKilepes_Click(object sender, EventArgs
e)
{
    Close();
}
}
}

```

## 12.3 Könyvtárkezelési utasítások

### 12.3.1 Könyvtár műveletek

Új könyvtár létrehozása:

```
System.IO.Directory.CreateDirectory (string)
```

Könyvtár tartalmának mozgatása egyik (source) helyről a másikra (destination):

```
System.IO.Directory.Move(source, destination);
```

Üres könyvtár tartalmának törlése:

```
System.IO.Directory.Delete (string)
```

Alkönyvtár létrehozása:

```
System.IO.DirectoryInfo.CreateSubdirectory (string)
```

Visszaadja az adott könyvtárban lévő alkönyvtárak neveit:

```
System.IO.Directory.GetDirectories (string)
```

A paraméterében megadott könyvtár létezését vizsgálja



```
System.IO.Directory.Exists(string)
```

Visszaadja az aktuális alkönyvtárat

```
System.IO.Directory.GetCurrentDirectory()
```

A gyökérkönyvtárat adja vissza

```
System.IO.Directory.GetDirectoryRoot(string);
```

A gépen található logikai meghajtókat listázza ki

```
System.IO.Directory.GetLogicalDrives();
```

### 12.3.2 Példa: Könyvtár műveletek használata

Példánkban a könyvtárak kezelését mutatjuk be. A létezik gombra kattintva a felhasználó ellenőrizheti léteik-e már a "proba" nevű könyvtár a "c:\\" meghajtón. A könyvtár létrehozása gombbal létre tudjuk hozni a tervezett könyvtárakat. A következő gomb az aktuális könyvtárat adja eredményül. Az alkönyvtárak létrehozása gombbal az adott alkönyvtárba tudunk további könyvtárakat létrehozni. A következő gombra kattintva a mellette lévő listaablakban az adott könyvtár alkönyvtárait tudjuk listáztatni. A gyökérkönyvtár gombra kattintva az adott útvonal gyökérkönyvtárát adja vissza. A meghajtók gombra kattintva a mellette lévő listaablakban a logikai meghajtókat listázza ki.

A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását!

9. Hozzunk létre új projektet Konyvtarak néven!

10. Tervezzük meg az űrlapot az alábbiak szerint:

Könyvtárak kezelése

Létezik

Könyvtár létrehozása

Aktuális könyvtár

Alkönyvtárak létrehozása

Könyvtár alkönyvtárai

IstKonyvtar

Gyökérkönyvtár

Meghajtók

IstMeghajtok

Kilépés

11. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Form	Könyvtárak kezelése	frmKonyvutar	
ListBox	lstKonyvutar	lstKonyvutar	
ListBox	lstMeghajtok	lstMeghajtok	
Button	Könyvtár létrehozása	btnKonyvutar	
Button	Aktuális könyvtár	btnAktualis	
Button	Alkönyvtárak létrehozása	btnAlkonyvutar	
Button	Könyvtár alkönyvtárai	btnAllista	
Button	Gyökérkönyvtár	btnGyoker	
Button	Meghajtók	btnMeghajtok	
Button	Kilépés	btnKilepes	

Fel kell hívni a figyelmet még arra is, hogy állománykezelés során az állomány elejére be kell szúrunk a következő sort, mert csak ezután lesz képes programunk a fájlok kezelésére.

```
using System.IO;
```

12. Konyvtarak.cs:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Globalization;
using System.IO;

namespace DirectoryTest
{
    public class frmKonyvutar : System.Windows.Forms.Form
    {
        public frmKonyvutar()
        {
            InitializeComponent();
        }

        static void Main()
        {
            Application.Run(new frmKonyvutar());
        }

        private void btnLetezik_Click(object sender, EventArgs e)
        {
            //Könyvtár létezésének vizsgálata.
        }
    }
}
```

```

        if (Directory.Exists("c:\\proba"))
            MessageBox.Show("c:\\proba könyvtár már létezik");
        else
            MessageBox.Show("c:\\proba könyvtár még nem létezik");
    }

    private void btnKonyvtar_Click(object sender, EventArgs e)
    {
        //Könyvtár létrehozása.
        Directory.CreateDirectory("c:\\proba");
    }

    private void btnAktualis_Click(object sender, EventArgs e)
    {
        //Aktuális könyvtár lekérdezése.
        MessageBox.Show("Az aktuális könyvtár" +Directory.
GetCurrentDirectory());
    }

    private void btnAlkonyvtar_Click(object sender, EventArgs e)
    {
        //Alkonyvtárak létrehozása.
        Directory.CreateDirectory("c:\\proba\\a\\1\\2\\3");
        Directory.CreateDirectory("c:\\proba\\b");
        Directory.CreateDirectory("c:\\proba\\c");
    }

    private void btnGyoker_Click(object sender, EventArgs e)
    {
        // Gyöker könyvtár lekérdezése.
        MessageBox.Show(Directory.GetDirectoryRoot("c:\\proba\\aa\\11"));
    }

    private void btnMeghajto_Click(object sender, EventArgs e)
    {
        // A gépen található meghajtók lekérdezése.
        lstMeghajtok.Items.AddRange(Directory.GetLogicalDrives());
    }

    private void btnAlLista_Click(object sender, EventArgs e)
    {
        // Könyvtárban lévő alkönyvtárak listázása.
        lstKonyvtar.Items.Clear();
        foreach (string s in Directory.GetDirectories("c:\\proba"))
            lstKonyvtar.Items.Add(s);
    }

    private void btnKilepes_Click(object sender, EventArgs e)
    {
        Close();
    }
}
}

```

### 12.3.3 Állományműveletek

A következő metódus a fájl attribútumát olvassa be:

```
File.GetAttributes(path1)
```

A következő metódus a fájl attribútumát módosítja és a változásokat állítja be:

```
File.SetAttributes(path1, File.GetAttributes(path1) Or  
FileAttributes.Hidden)
```

A következő metódus a fájl teljes (abszolút) elérési útvonalát adja meg:

```
fullPath = Path.GetFullPath(path1)
```

A következő metódus a fájl nevét adja meg kiterjesztéssel együtt:

```
result = Path.GetFileName(path1);
```

A következő metódus a fájl kiterjesztését adja meg:

```
extension = Path.GetExtension(path1);
```

A következő metódus a fájl kiterjesztésének változtatási lehetőségét nyújtja:

```
result = Path.ChangeExtension(f1, ".old")
```

Nézzünk meg egy kódrészletet arra, hogyan lehet egy állományt egy másik könyvtárba mozgatni illetve másolni, közben az állomány törlése is bemutatásra kerül, abban azt esetben ha abban a könyvtárban létezik ilyen nevű állomány:

- `Move(f1, f2)` – az adott könyvtárban lévő **"f1"** fájl mozgatása **"f2"**-be
- `Delete(f1)` – az adott könyvtárban lévő **"f1"** fájl törlése
- `Copy(f1, f2)` – az adott könyvtárban lévő **"f1"** fájl másolása **"f2"**-be
- `CopyTo(f1, f2)` – az adott könyvtárban lévő **"f1"** fájl másolása **"f2"**-be, de nem engedi meg az ott lévő ugyanilyen nevű fájl felülírását
- `Exists(f1)` – megvizsgálja, hogy az adott könyvtárban létezik-e az **"f1"** fájl

```
string utvonall1 = @"c:\proba\testt.txt";  
string utvonall2 = @"c:\proba\teszt.txt";  
string utvonall3 = @"c:\proba\teszt.txt";  
string teljesutvonal, eredmeny, kiterjesztes;  
if (!File.Exists(utvonall1))  
{  
    using (FileStream fs = File.Create(utvonall1)) {}  
}  
  
//ha létezik ebben a könyvtárban ilyen néven fájl akkor törli  
if (File.Exists(utvonall2))  
    File.Delete(utvonall2);  
  
//ha létezik ebben a könyvtárban ilyen néven fájl akkor törli  
if (File.Exists(utvonall3))  
    File.Delete(utvonall3);  
  
// a fájl mozgatása egyik könyvtárból a másikba  
File.Move(utvonall1, utvonall2);  
MessageBox.Show("útvonal1="+utvonall1+" útvonal2="+utvonall2);  
  
// a fájl másolása egyik könyvtárból a másikba  
File.Copy("útvonal1="+utvonall1+" útvonal3="+utvonall3);  
MessageBox.Show(utvonall1, utvonall3);
```

A következő kódrészletben kiíratjuk egy adott könyvtárban található állományok neveit és méreteit:

- GetFiles() – az adott könyvtárban lévő fájlok listáját állítja elő
- Name – az adott könyvtárban lévő fájlok nevei
- Length - az adott könyvtárban lévő fájlok méretei

```
DirectoryInfo di = new DirectoryInfo("c:\\");  
lstFiles.Items.Clear();  
FileInfo[] fiArr = di.GetFiles();  
lstLista.Items.Add(di.Name);  
lstLista.Items.Clear();  
foreach (FileInfo f in fiArr)  
    lstLista.Items.Add(f.Name + " " + f.Length);
```

## 13 XML alapok és XML fájlok kezelése

### 13.1 Az XML nyelv

Az Extensible Markup Language (XML, kiterjeszthető leíró nyelv) általános célú leíró nyelv speciális célú leíró nyelvek létrehozására. A leíró nyelvek adatok strukturált leírására szolgálnak, nem programozási nyelvek. Strukturált adatok alatt olyan dolgokat értünk, mint táblázatok, címjegyzékek, 2D vagy 3D rajzok, stb.

Az XML bemutatására lássunk egy példát:

```
<?xml version="1.0" encoding="UTF-8"?>
<Recept név="kenyér" elk_idő="5 perc" sütés_idő="3 óra">
  <cím>Egyszerű kenyér</cím>
  <összetevő mennyiség="3" egység="csésze">Liszt</összetevő>
  <összetevő mennyiség="10" egység="dekagramm">Élesztő</összetevő>
  <összetevő mennyiség="1.5" egység="csésze">Meleg víz</összetevő>
  <összetevő mennyiség="1" egység="teáskanál">Só</összetevő>
  <Utasítások>
    <lépés>Keverj össze minden összetevőt, aztán jól gyúrd össze!</lépés>
    <lépés>Fedd le ruhával és hagyd pihenni egy óráig! </lépés>
    <lépés>Gyúrd össze újra, aztán süsd meg a sütőben!</lépés>
  </Utasítások>
</Recept>
```

Az XML nem határozza meg a nyelv jelölőelem készletét, ez teljes mértékben kiterjeszthető - innen ered a neve is. A jelölőelem készlet meghatározásával kialakíthatjuk saját adatstruktúránk leírására alkalmas XML alapú leíró nyelvünket.

Léteznek XML-re épülő szabványosított leírónyelvek is, például:

- XHTML: HTML-hez hasonló, XML alapú dokumentum leíró nyelv
- SVG (Scalable Vector Graphics): 2D vektorgrafikák leírására

```
<svg>
  <g style="fill-opacity:0.7;">
    <circle cx="1cm" cy="1cm" r="1cm" style="fill:red" />
    <circle cx="2cm" cy="1cm" r="1cm" style="fill:blue" />
    <circle cx="1.5cm" cy="2cm" r="1cm" style="fill:green"/>
  </g>
</svg>
```



Ahhoz, hogy egy XML dokumentum helyes legyen, két követelménynek kell megfelelnie:

1. Helyes szintaktika
2. Érvényesség

### 13.1.1 Az XML dokumentum szintaktikája

1. Az dokumentum első sora az XML deklaráció. Megadása nem kötelező. Információt tartalmazhat a használt XML verzióról (általában 1.0) és a karakterkódolásról.

```
<?xml version="1.0" encoding="UTF-8"?>
```

2. A dokumentum további része egymásba ágyazott **csomókból** (node) áll, melyeknek lehetnek **tulajdonságai** (attribute) és **tartalma** (content). Egy elem általában egy nyitó tag-ből (start tag) és egy záró tag-ből (end tag), valamint a közöttük lévő szövegből és más node-okból áll.
3. A **nyitó tag** egy név, melyet < > jelek közé írunk. Pl: <Recept>. A tag név nem tartalmazhat szóközt, a nyelv különbséget tesz a kis és a nagy betűk között. A **záró tag** egy / jelből és a nyitó tag-ben is szereplő névből áll, pl: </Recept>
4. A node **tulajdonságai** - ha vannak - a nyitó tag-ben kerülnek felsorolásra. Minden tulajdonság szimpla vagy dupla idézőjelek közé kerül.

```
<Recept név="kenyér" elk_idő="5 perc" sütés_idő="3 óra">
```

5. A nyitó és a záró tag-ek közé kerül az node tartalma (content).

```
<összetevő mennyiség="3" egység="csésze">Liszt</összetevő>
```

6. Ha egy node-nak nincs tartalma, üres node-nak nevezzük. Üres node-oknál használhatunk önzáró tag-eket:

```
<Szakács név="Szakács Gyula" />
```

Ez az írásmód ekvivalens a következővel:

```
<Szakács név="Szakács Gyula"></Szakács>
```

7. A dokumentum egy gyökérelemet kell tartalmazzon. Az XML deklaráció és a feldolgozásra vonatkozó utasítások megelőzhetik a gyökérelemet. Példánkban a gyökérelem a Recept.
8. Az XML dokumentum tartalmazhat megjegyzéseket is a következő formátumban:

```
<!-- a jelek közé írt szöveg nem kerül feldolgozásra -->
```

9. Ha a dokumentum nem helyesen formázott, az XML értelmezőnek meg kell tagadnia a feldolgozást.

A fentiekből következik, hogy minden nyitó tag-et le kell zárni egy záró tag-el. A tag-ek faszzerűen egymásba ágyazhatók, de átfedés nem lehet közöttük.

```
helytelen:
<Példa1><Példa2></Példa1><Példa2>

helyes:
<Példa1>
<Példa2>
</Példa2>
</Példa1>
```

### 13.1.2 Az XML dokumentum érvényessége

Azon túl hogy az XML dokumentum formailag helyes, meg kell feleljen egy adott sémának. Ez azt jelenti, hogy csak olyan tag-eket és attribútumokat tartalmazhat, melyeket az XML fájl feldolgozó program felismer. Ez lehet szabványosított XML alapú formátum, mint például a vektorgrafikák leírására alkalmas SVG (Scalable Vector Graphics) nyelv, de lehet általunk meghatározott formátum is.

Megjegyzés: Az XML dokumentumokhoz készíthetünk DTD (Document Type Definition) kódot, mely az XML fájl érvényességét ellenőrzi. A DTD kód kerülhet az XML fájl elejére, de külső fájlba is helyet foglalhat. A DTD-ben megadhatjuk, mely milyen node-oknak milyen gyermekei és tulajdonságai lehetnek. Előírhatunk kötelező és opcionális tulajdonságokat. Ha DTD-t használunk, sem az XML szerkesztő, sem a fát feldolgozó program nem enged a megadott sémának ellentmondó node-okat és tulajdonságokat létrehozni. A DTD ismertetésére terjedelmi okok miatt itt nem térünk ki.

Bővebb információért a következő honlapot érdemes felkeresni:

[http://en.wikipedia.org/wiki/Document\\_Type\\_Definition](http://en.wikipedia.org/wiki/Document_Type_Definition)

### 13.1.3 Az XML előnyei

Ember és gép számára egyaránt könnyen olvasható, szöveges formátum.

A legtöbb fejlesztőkörnyezethez rendelkezésre állnak objektumosztályok, melyek segítségével könnyedén tudjuk kezelni az XML formátumban leírt adatokat. Így XML használatával sok unalmas és felesleges programozási munkától kímélhetjük meg magunkat.



## 13.2 XML fájlok feldolgozása C# környezetben

XML fájlok feldolgozására többféle lehetőség van, terjedelmi okok miatt ezek közül egyet mutatunk be.

Dolgozzuk fel az alábbi egyszerű XML fájlt:

```
<?xml version="1.0" encoding="utf-8" ?>
<gyökér>
  <gyermek1>
    <unoka1> Én unoka1 vagyok, gyermek1 leszármazottja. </unoka1>
    <unoka2> Én unoka2 vagyok, gyermek1 leszármazottja. </unoka2>
  </gyermek1>
  <gyermek2>
    <unoka1> Én unoka1 vagyok, gyermek2 leszármazottja. </unoka1>
    <unoka2> Én unoka2 vagyok, gyermek2 leszármazottja. </unoka2>
  </gyermek2>
</gyökér>
```

A dokumentum tartalmaz egy gyökérelemet, melynek neve „gyökér”. (Ahogy azt már korábban láttuk, minden XML dokumentum csak egyetlen gyökérelemet tartalmazhat.)

A gyökérelemnek két gyermeke (childnode-ja) van: „gyermek1” és „gyermek2”.

„gyermek1”-nek is van két gyermeke (childnode-ja): „unoka1” és „unoka2”.

### 13.2.1 XML fájl betöltése

Az XML dokumentumok feldolgozására az XmlDocument objektumosztály szolgál. Hozzunk létre belőle egy példányt peldaXML néven. A peldaXML.Load() metódusával tölthetünk be egy XML fájlt.

```
XmlDocument peldaXML = new XmlDocument();
peldaXML.Load("pelda.xml");
```

### 13.2.2 A fa ágai, műveletek node-okkal

Az egyes node-ok elérésére nézzünk néhány egyszerű példát! (Az első elemre mindig nullás indexel hivatkozunk.)

```
textBox1.Text = peldaXML.DocumentElement.InnerXml;
```

```
textBox2.Text = peldaXML.DocumentElement.ChildNodes[0].InnerXml;
```

```
textBox3.Text =
peldaXML.DocumentElement.ChildNodes[0].ChildNodes[1].InnerXml;
```

A példa futtatása után a szövegdobozokban a következő jelenik meg:

TextBox1 – a gyökérelemen belül lévő XML kódot tartalmazza:

```
<gyermek1><unoka1> Én unoka1 vagyok, gyermek1 leszármazotta.  
</unoka1><unoka2> Én unoka2 vagyok, gyermek1 leszármazotta.  
</unoka2><unoka3 neve="Anna">Legkisebb unoka  
:)</unoka3></gyermek1><gyermek2><unoka1> Én unoka1 vagyok,  
gyermek2 leszármazotta. </unoka1><unoka2> Én unoka2 vagyok,  
gyermek2 leszármazotta. </unoka2></gyermek2>
```

TextBox2 – a gyökérelem első gyermekét tartalmazza:

```
<unoka1> Én unoka1 vagyok, gyermek1 leszármazotta.  
</unoka1><unoka2> Én unoka2 vagyok, gyermek1 leszármazotta.  
</unoka2><unoka3 neve="Anna">Legkisebb unoka :)</unoka3>
```

TextBox3 – a gyökérelem első gyermekének második gyermeke:

```
Én unoka2 vagyok, gyermek1 leszármazotta.
```

### 13.2.3 Új node beszúrása és a fájl mentése.

Bővítsük az XML fát egy új elemmel!

```
XmlElement ujElem;  
ujElem = peldaXML.CreateElement("unoka3");  
ujElem.InnerText = "Legkisebb unoka :)";  
ujElem.SetAttribute("neve", "Anna");  
peldaXML.DocumentElement.ChildNodes[0].AppendChild(ujElem);  
  
peldaXML.Save("pelda.xml");
```

A program futtatása után az XML fájl:

```
<?xml version="1.0" encoding="utf-8"?>  
<gyökér>  
  <gyermek1>  
    <unoka1> Én unoka1 vagyok, gyermek1 leszármazottja. </unoka1>  
    <unoka2> Én unoka2 vagyok, gyermek1 leszármazottja. </unoka2>  
    <unoka3 neve="Anna">Legkisebb unoka :)</unoka3>  
  </gyermek1>  
  <gyermek2>  
    <unoka1> Én unoka1 vagyok, gyermek2 leszármazottja. </unoka1>  
    <unoka2> Én unoka2 vagyok, gyermek2 leszármazottja. </unoka2>  
  </gyermek2>  
</gyökér>
```

### 13.2.4 Node törlése a fából

Ez a programsor törli „gyermek1”-et, és annak gyermekeit a fából:

```
peldaXML.DocumentElement.RemoveChild(peldaXML.DocumentElement.ChildNodes[0]);
```

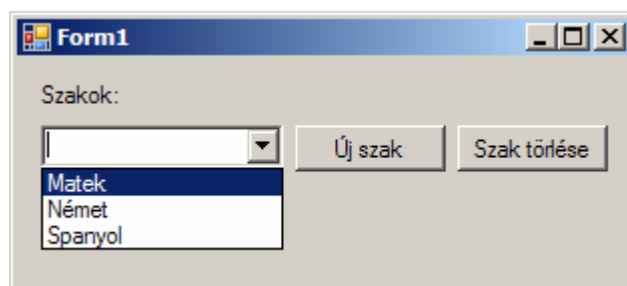
### 13.3 Példa XML fájl feldolgozására

Példánk hasonlít a *többablakos alkalmazások készítése* fejezet feladatához. A különbség annyi, hogy itt a legördülő doboz adatait XML fájlból töltjük be. Az XML fájl és a lista elemeit tudjuk bővíteni és törölni is.

A lista elemeit az „Új szak” gombra kattintva bővíthetjük. Ha a gombra kattintunk, ablak ugrik fel, melyben megadhatjuk az új szak nevét. A legördülő doboz csak akkor bővül az új elemmel, ha az ablakot OK gombbal zártuk be.

A szak törlése gomb mindig az éppen kiválasztott szakot törli. A változás azonnal mentésre kerül az XML fájlba.

1. Hozzunk létre egy új solutiont xml\_prg néven!
2. Tervezzük meg a főürlapot az alábbiak szerint:



- Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Label	Szakok:		
ComboBox		cbSzakok	
Button	Új szak	btUjSzak	
Button	Szak törlése	btSzakTorles	

3. Csatoljunk XML fájlt a Solution-hoz

- A File / New / File menüpont alatt hozzunk létre új XML dokumentumot!
- Az XML szerkesztő ellenőrzi dokumentumunk szintaktikai helyességét.  
Készítsünk el a következő dokumentumot:

```
<?xml version="1.0" encoding="utf-8"?>
<szakok>
  <szak>Matek</szak>
  <szak>Német</szak>
```

```
<szak>Spanyol</szak>
</szakok>
```

- Mentsük el XML dokumentumunkat szakok.xml néven. Ha a programban külön elérési útvonal nélkül hivatkozunk az XML fájlra, alapértelmezésben abban a könyvtárban fogja keresni, ahonnan magát a programot futtatjuk. Ez valószínűleg a

„My Documents\Visual Studio 2005\Projects\xml\_prg\xml\_prg\bin\Debug”

könyvtár lesz. Ha nem akarunk az elérési útvonal megadásával külön bajlódni, mentsük ide a szakok.xml-t.

- Ha a „szakok.xml” aktív a szerkesztőben, a File menüben a „Move szakok.xml into” menüponttal hozzáadhatjuk solutionunkhoz a fájlt. Ezután a „Solution explorer”-ben is megtaláljuk.

#### 4. Az alkalmazás indításakor automatikusan töltsük fel a legördülő dobozt az XML fájl adataival!

- Az XML fájlokat feldolgozó objektumosztályok külön osztálykönyvtárban szerepelnek. Használjuk a System.Xml könyvtárat.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Xml;
...
```

- Bővítsük az űrlapot leíró objektumosztályt XmlDocument osztály szakokXML nevű példányával:

```
...
namespace WindowsApplication2
{
    public partial class Form1 : Form
    {
        //Bővítsük az objektumosztályt szakokXML
        tulajdonsággal
        XmlDocument szakokXML;

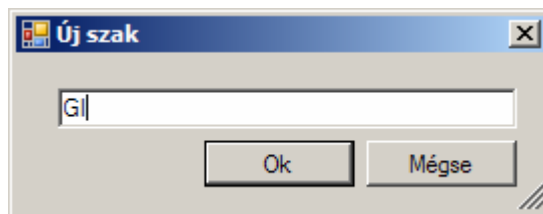
        public Form1 ()
        {
            InitializeComponent();
        }
    }
}
```

```
}
```

- Az űrlap konstruktorában nyissuk járjuk be az XML fájlt, és a „szak” nevű node-ok tartalmával bővítjük a legördülő dobozok elemeit!

```
...  
public Form1()  
{  
    InitializeComponent();  
  
    //Hozzuk létre a szakokXML-t,  
    //és töltsük be a szükséges fájlt!  
    szakokXML = new XmlDocument();  
    szakokXML.Load("szakok.xml");  
  
    XmlNodeList lstSzakok =  
        szakokXML.DocumentElement.GetElementsByTagName("szak");  
  
    for (int i = 0; i < lstSzakok.Count; i++)  
    {  
        cboSzakok.Items.Add(lstSzakok[i].InnerText);  
    }  
}  
...
```

5. Készítsük el a felugró űrlapot, melyben megadhatjuk az új szak nevét:



- Válasszuk a project / Add Windows Form menüpontot
- A megjelenő sablonok közül a Windows Formra lesz szükségünk. Nevezzük az új űrlapot „UjSzak”-nak, majd kattintsunk az „Add” gombra!
- Helyezzünk el egy szövegdobozt (TextBox) és két gombot (Button) az űrlapon!
- Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name	Properties
Label	Új szak neve		
TextBox			Modifiers: public
Button	Ok	btnOk	DialogResult: OK
Button	Mégsem	btnMegsem	DialogResult: Cancel
Form	Új szak	UjSzak	AcceptButton: btnOk

			CancelButton: btnMegsem FormBorderStyle: FixedDialog MaximizeBox: False MinimizeBox: False ShowInTaskbar: False
--	--	--	--

- Ha szövegdozoz modifiers tulajdonságát public-re állítjuk, kívülről is láthatóvá válnak a szövegdozoz tulajdonságai, így ki tudjuk olvasni annak tartalmát.
- Nem mindegy, hogy a felhasználó Ok vagy Mégsem gombbal zárja be az ablakot. Ezt az információt a gombok DialogResult, illetve a Form AcceptButton és CancelButton tulajdonságainak értelem szerű beállításával tudjuk visszaadni a főúrlapnak. Lásd később.
- Rendeljük eseményeket az „Ok” és a „Mégsem” feliratú gombokhoz!  
Mindkét gomb bezárja az ablakot.

#### 6. UjSzak.cs:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication2
{
    public partial class UjSzak : Form
    {
        public UjSzak()
        {
            InitializeComponent();
        }

        private void btMegse_Click(object sender, EventArgs e)
        {
            Close();
        }

        private void btOk_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
}

```

7. Szaklista űrlap gombjához rendeljük eseményt, mely megnyitja az UjSzak űrlapot.  
Ha a felhasználó az Ok gombbal zárta be a felugró ablakot, adjuk hozzá a legördülő doboz tartalmához az új szakot!

- A ShowDialog() metódus hívásával az űrlapot „modal” ablakként jelenítjük meg. Ha ShowDialog() metódus OK értékkel tér vissza, bővítjük a lista elemei és az XML fájlt.

```
private void btUjSzak_Click(object sender, EventArgs e)
{
    UjSzak frmUjSzak = new UjSzak();

    if (frmUjSzak.ShowDialog() == DialogResult.OK)
    {
        //Bővítjük a legördülő doboz elemeit az új szakkal!
        cboSzakok.Items.Add(frmUjSzak.tbSzak.Text);

        XmlElement ujElemXML =
            szakokXML.CreateElement("szak");

        ujElemXML.InnerText = frmUjSzak.tbSzak.Text;

        szakokXML.DocumentElement.AppendChild(ujElemXML);

        //Mentsük lemere a bővített XML dokumentumot!
        szakokXML.Save("szakok.xml");
    }
}
```

8. A törlés gomb törölje ki a lista éppen kiválasztott elemét!

```
private void btSzakTorles_Click(object sender, EventArgs e)
{
    //változóban tároljuk a legördülő dobozban
    //kiválasztott elem sorszámát
    int torlendo = cboSzakok.SelectedIndex;

    //A kiválasztott elemet töröljük a legördülő doboz
    //elemei közül
    cboSzakok.Items.RemoveAt(torlendo);

    //A megfelelő Node-ot töröljük az XML dokumentumból
    szakokXML.DocumentElement.RemoveChild(
        szakokXML.DocumentElement.ChildNodes[torlendo]);

    //A módosított dokumentumot mentüklemezre!
    szakokXML.Save("szakok.xml");
}
```

## 14 Az UML nyelv és fontosabb diagramtípusai

Az UML (Unified Modeling Language) egy igen elterjedt, szoftverrendszerek fejlesztésére szolgáló általános célú vizuális modellező nyelv. Az UML egy olyan egységesített grafikus jelölésrendszert használ, amely három korábbi technika, az OMT-2 (Object Modeling Technique), az OOAD (Object-Oriented Analysis and Design) és az OOSE (Object-Oriented Software Engineering) szintéziseként alakult ki 1995-től kezdődően, e három módszer alkotói (James Rumbaugh, Grady Booch és Ivar Jacobson) együttműködése eredményeként. A tudomány történetében általában a jelentős eredmények megszületését komoly vajúadás előzi meg, amely az UML esetében sok elméleti és gyakorlati szakember évekig tartó szisztematikus munkáját és felhalmozott alkalmazásfejlesztési tapasztalatát jelentette. 1989 és 1994 között a modellező nyelvek és módszerek száma kevesebb, mint 10-ről több mint 50-re nőtt. Az ún. első generációs módszerek közül feltétlenül meg kell említeni a Coad/Yourdon, a Shlaer/Mellor és a Fusion módszert. A 90-es évek első felében megjelent, szoftverfejlesztésről szóló tankönyvek általában az egyik vagy másik módszert propagálták. Ez az időszak az informatika történetébe mint a „módszerek háborúja” vonult be.

1996-ben jelent meg az UML 0.9, majd 1997-ben az 1.1 verzió. Ez utóbbi megszületésénél már egy széleskörű ipari konzorcium bábáskodott, amelynek tagjai között volt az IBM, a HP, a Microsoft, az Oracle, a DEC, a Texas Instruments és az Unisys, vagyis az informatikai piac hét legbefolyásosabb szereplője. Az első teljes változat, az UML 1.3 ténylegesen ipari szabvánnyá vált. Már évek óta az UML 2.0 verziót használják, de 2006-ban várható a 2.1 változat megjelenése. A 2.0 verzióban a diagramok száma 9-ről 13-ra nőtt, s más tekintetben is jelentős változások jelentek meg.

Az UML tehát egy általános célú vizuális modellező nyelv, s nem egy programozási nyelv. Nem tudunk programot írni UML-ben, bár sok CASE eszköz képes UML diagramokból Java, C++ vagy C# kódot generálni. Bár az UML maga nem egy szoftverfejlesztő folyamat, szorosan kapcsolódik az USDP-hez (Unified Software Development Process), amely viszont már alkalmas rendszerek fejlesztésére. Az USDP is a Booch - Jacobson – Rumbaugh trió munkája, s gyakran RUP –ként (Rational Unified Process) emlegetik. Az elnevezésben szereplő Rational egy informatikai cég neve, ahol éveken át együtt dolgozott az említett hármas. A Rational cég terméke a CASE eszközök között piacvezető szerepet betöltő Rational Rose is. Magát a céget 2003-ban felvásárolta az IBM, de a márkanév megmaradt. Az UML nem kötődik egyetlen meghatározott módszertanhoz, hanem bármelyik létező módszertannal (pl. OPEN) együtt használható. Egyebek mellett ez teszi az UML-t különösen népszerűvé a rendszerfejlesztők körében.

Az UML által használt diagramtípusokat két osztályba szokás sorolni. Beszélhetünk struktúradiagramokról, amelyek közé tartoznak az



- osztály (*class*)
- objektum (*object*)
- komponens (*component*)
- csomag (*package*) vagy alrendszer
- összetétel (*composite structure*) diagramok
- telepítés (*deployment*)

valamint viselkedési diagramokról (szám szerint hét diagramtípusról)

- tevékenység (*activity*)
- használati eset (*use case*)
- állapotgép (*state machine*, korábbi verziókban state transition vagy state chart)
- illetve
- kölcsönhatás (*interaction*) diagramok.

Ez utóbbi kategória az alábbi négy diagramtípus összefoglaló neve

- szekvencia
- kommunikáció
- kölcsönhatás-áttekintés (*interaction overview*)
- idő (*timing*).

A struktúradiagramok azt mutatják meg, hogy a modellelemek (objektumok) hogyan kapcsolódnak egymáshoz.

Lássuk, mire alkalmasak és használatosak ezek a diagramtípusok az UML-ben!

Az **osztálydiagramok** a modellben szereplő osztályok és kölcsönhatásaik (öröklés, aggregáció, társulás) leírására szolgálnak. Mivel az osztály az objektum-orientált technikák egyik legfontosabb alapfogalma, ezért az osztálydiagramok igen fontos szerepet játszanak a modellezésben. A legfontosabb jellemzője ennek a diagramtípusnak az, hogy a rendszer statikus struktúráját írja le.

Az **objektumdiagramokat** a vizsgált rendszer objektumai és a közöttük egy adott időpillanatban meglévő kapcsolatok ábrázolására használjuk, mint egy pillanatfelvételt. Ilyképpen tehát az osztálydiagramok speciális esetének tekinthetjük őket, ezért nem fogjuk az objektumdiagramokat külön tárgyalni. Egy objektumdiagram kevésbé absztrakt, mint a megfelelő osztálydiagram.

Az UML-ben a **komponensdiagramok** az újrahasználható (reusable), bizonyos értelemben autonóm és nagyobb méretű tervezési egységet jelentő komponenseket ábrázolják a biztosított és szükséges interfészeikkel együtt. Fontos megjegyezni, hogy a komponensdiagramok az

implementáció nézőpontjából írják le a rendszert. Különösen abból a szempontból hasznosak, hogy lehetővé teszik a tervezőnek annak végiggondolását, vajon az alkalmazás valóban rendelkezik-e a szükséges funkcionalitással, azaz végrehajtja-e mindazt, amit a megrendelő elvár tőle. A komponensdiagramokat előszeretettel használják a rendszer nagybani architektúráját egyeztető megbeszéléseken, ahol a projekt kulcsfigurái közötti tárgyalás legfontosabb segédeszközevé vál(hat)nak. A komponensdiagramok tulajdonképpen légi felvételként funkcionálnak, s a rendszer egészének szerkezetét mutatják. A felvétel felbontása alapvetően attól függ, pontosan mit is szeretnénk megmutatni.

Az **alrendszer-diagramok** vagy másképpen **csomagdiagramok** azt mutatják meg, hogyan lehet a modell elemeit nagyobb egységekbe, alrendszerekbe, csomagokba rendezni, illetve azt, milyen függőségi kapcsolat áll fenn az egyes csomagok között. Az elsődleges ok, ami miatt ezt a diagramtípust használjuk az az, hogy az UML ábrák meglehetősen nagyra képesek nőni, olyan méretűre, amit már nehezen lehet átlátni. Ilyenkor szokás a modell bizonyos elemeit (pl. osztályokat) valamilyen logikai szempont szerint csoportosítani. Eredményként egy áttekinthetőbb ábra keletkezik. Leegyszerűsítve azt mondhatjuk, hogy a csomagok fájlmappák, amelyek egy csoportban tartalmaznak logikailag összetartozó dolgokat. Akkor sikeres a „csomagolás”, ha a különböző alrendszerek közötti függőség minél kisebb. A tervezői cél minden esetben lazán kapcsolódó csomagok kialakítása.

Az **összetétel-diagram** a vizsgált rendszer belső struktúráját, felépítését, „összetételét” mutatja. Egészen pontosan azt, hogy a rendszer elemei hogyan kapcsolódnak egymáshoz a kívánt funkcionalitás elérése céljából. A diagramon lehetnek port-ok, amelyek kapuként szolgálnak a környezet felé. Konceptcionálisan az összetétel-diagramok az osztálydiagramokat és a komponensdiagramokat kötik össze, de nem mutatják az osztálydiagramok tervezési, illetve a komponensdiagramok implementációs részleteit.

A **telepítésdiagramok** készítésének az a célja, hogy megmutassa, hogyan lesz a rendszer fizikailag telepítve a hardver környezetbe, hol fognak az egyes komponensek működni, és hogyan kommunikálnak majd egymással. A telepítésdiagramok mind a hardverkomponenseket, mind az ún. köztes termékeket (middleware) ábrázolják. Ez utóbbiak „ragasztóként” szolgálnak hardver és szoftver között.

A **tevékenységdiagramok** üzleti folyamatok modellezésére szolgálnak, a vizsgált rendszer belső logikájának feltérképezésére. Azt mutatják meg, milyen elemi tevékenységekből épül fel egy komplex üzleti folyamat, mi hajtható végre párhuzamosan, s léteznek-e alternatív útvonalak az üzleti folyamat gráfjában. A tevékenységdiagramok sok hasonlóságot mutatnak a strukturált rendszerfejlesztés talán legismertebb technikájával, az adatfolyamábrákkal.

A **kölcsönhatás-áttekintés** diagramok a tevékenységdiagramok leegyszerűsített változataiként foghatók fel. Ezek is irányított gráfok, de a csúcsok nem elemi tevékenységek, hanem különféle kölcsönhatás-diagramok, leggyakrabban szekvenciadiagramok,

kommunikációdigrammok stb. A folyamatok áttekintéseként foghatók fel, s a részleteket csak elnagyoltan tartalmazzák.

A **használati eset diagram** a követelmények feltárásának egyik leghatékonyabb eszköze. Azonosítja az interakcióban részt vevő szereplőket, s megnevezi magukat a kölcsönhatásokat. Az aktorokat pálcikafigurák, az interakció osztályait pedig névvel ellátott ellipszisek jelölik.

Az **állapotgép-diagram** (state machine, az UML korábbi verzióiban state chart vagy state transition) egy rendszer lehetséges állapotait írja le, s az állapotok közötti átmeneteket. Ezt a diagramtípust már évtizedek óta használják az informatikában. Lényegében egy irányított gráfot jelentenek, kezdő és végponttal, s megengedjük, hogy folyamatok szétváljanak és újra egyesüljenek. Szoros az állapotgépek és a (Mealy és Moore) automaták kapcsolata.

A **szekvenciadiagramok** egy folyamat belső logikáját mutatják be. Az osztálydiagramok mellett a szekvenciadiagramok jelentik az egyik legfontosabb UML diagramtípust. A szekvenciadiagramok a dinamikus modellezés eszközei a tevékenységdiagramokkal, a kommunikációdigrammokkal stb. egyetemben. Különösen alkalmasak forgatókönyvek (interakciósorozatok) leírására.

A **kommunikációdigrammok** a rendszer különböző elemei, entitásai között lebonyolított kommunikációt ábrázolják. Hasonló módon készülnek, mint a szekvenciadiagramok, de ezekben az időbeli sorrend nem explicit módon jelenik meg.

Az UML **idődiagramja** entítások időbeli viselkedését írja le. Egészen pontosan azt, hogy a vizsgált entitás milyen hosszan marad egy adott állapotban. Az UML 2.0 verzióban jelent csak meg. Különösen valós-idejű rendszerek és beágyazott szoftverek tervezésére használják.

A felsorolt 13 diagramtípusból nem szükséges egyformán jól ismerni valamennyit ahhoz, hogy jó minőségű alkalmazást tudjunk fejleszteni. Az UML három alkotójának véleménye szerint is a nyelv 20%-ának ismerete elégséges a feladatok 80%-ának megoldásához. Gyakorlati tapasztalatok azt mutatják, hogy üzleti alkalmazások fejlesztése során a legfontosabbak a

- tevékenység-,
- szekvencia-,
- osztálydiagramok,

másodlagosan pedig a

- használati eset,
- állapot-,
- kommunikáció-,

- komponens-,
- telepítés

diagramtípusokat használják. A fennmaradó típusok viszonylag ritkábban kerülnek alkalmazásra, bár számos esetben nagyon jól használhatók a követelményelemzés és a tervezés fázisában. Terjedelmi korlátok miatt ebben a fejezetben csak a legfontosabb típusokat fogjuk részletesebben tárgyalni.

## 14.1 Használati esetek

Mint korábban említettük, a használati esetek azonosítják a kölcsönhatások szereplőit és magukat a kölcsönhatásokat. Egy szereplő (aktor) bármi vagy bárki lehet, aki valamilyen kapcsolatban, kölcsönhatásban áll a vizsgálandó rendszerrel, abból a célból, hogy azon tudatos tevékenységet hajtson végre, illetve valamilyen megfigyelhető eredményt érjen el a tevékenység eredményeként. A használati esetek szereplői lényegében azt modellezzik, ahogyan a rendszerek felhasználói működtetnek, kezelnek valós rendszereket. A használati esetekkel történő modellezés a követelmények feltárásának egyik, ha nem a legfontosabb technikája. Másképpen forgatókönyv alapú feltárásnak is nevezzük. A használati eset diagramoknak négy alkotó eleme van:

- szereplők – aktorok, amelyeket pálcikafigurák személyesítenek meg, illetve téglalapok jelölnek, bennük az <<actor>> szócskával, amennyiben ezek a szereplők a saját jogukon további rendszerek,
- használati esetek – tevékenységsorozatok, ellipszisekkel ábrázoljuk őket,
- kapcsolatok – szereplők és használati esetek között fennálló kölcsönhatások, vékony folytonos vonallal ábrázolva,
- rendszer határai – a használati esetek köré rajzolt téglalap, amely azt mutatja meg, mi tartozik a rendszerhez, s mi nem. Mi van a határokon belül, illetve mi esik azokon kívül.

Fontos megjegyezni, hogy a szereplők mindig kívül esnek a rendszer határain. Azonban az előfordulhat, hogy a külső szereplőknek létezik valamilyen reprezentációja a rendszeren belül is, például egy adatbázis, amely az ügyfelek adatait tartalmazza. Az Ügyfél maga egy külső aktor, de a lista már egy belső adattároló.

A használati esetek diagram elkészítésének egyik fontos lépése (a rendszer határainak kijelölése után), a szereplők és a használati esetek azonosítása. Egy bonyolult rendszer esetében ez nem biztos, hogy egy könnyű feladat. Azzal tudjuk megkönnyíteni a keresést, ha felteszünk pár kérdést, és megpróbáljuk megválaszolni azokat. Ilyen kérdések lehetnek:

Szereplők

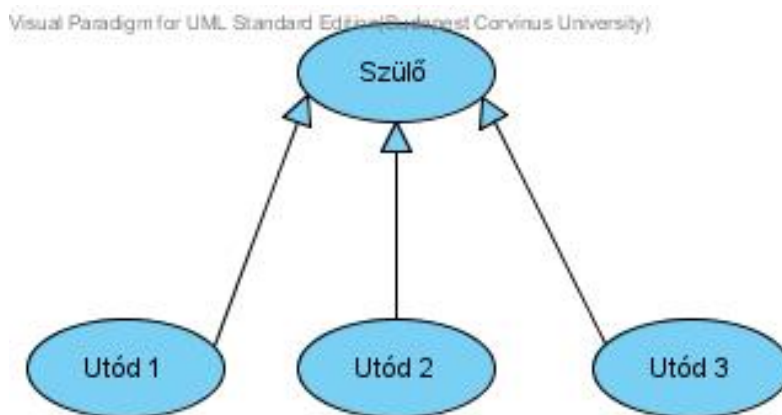
- Kik a rendszer használói?
- Ki szolgáltat adatot a rendszernek?
- Ki kap információt a rendszertől?
- Ki tartja karban a rendszert?
- Milyen más rendszerek állnak kapcsolatban a vizsgált rendszerrel?

#### Használati esetek

- Milyen konkrét szolgáltatást vár el egy szereplő a rendszertől?
- Lesz-e bármelyik szereplő „értésvetve”, ha változik a rendszer állapota?
- Mely szereplők tevékenysége eredményezi adatok tárolását, illetve visszakeresésüket?
- Léteznek-e olyan külső események, amelyek hatással vannak a rendszerre?

Sok esetben egy felhasználót, egy felhasználói viselkedést egynél több szereplővel, aktorral jellemezhetünk.

Használati esetek között háromféle kapcsolatot definiálhatunk. Az első egy öröklési vagy másképp generalizációs / specializációs kapcsolat. A használati esetek egyike az „utód”, a másik a „szülő” szerepét fogja játszani, s az őket összekötő (nem szaggatott, hanem folyamatos) vonal végén lévő nyíl a speciálistól (utód) az általános (szülő) felé mutat. E kapcsolat jelentése az, hogy az utód használati eset teljes egészében öröklí a szülő használati esetet, de azon módosíthat. Az általános használati esetet átírhatja, a résztvékenységeken egyszerűsíthet, vagy azokhoz újabb résztvékenységeket adhat hozzá. Egy szülő használati esetnek lehet több utóda, s elvben egy utódnak is lehet több szülője. Általában arra látunk példát, hogy felfelé a kapcsolat egyértelmű, azaz egy utódhoz csak egy szülő tartozik.

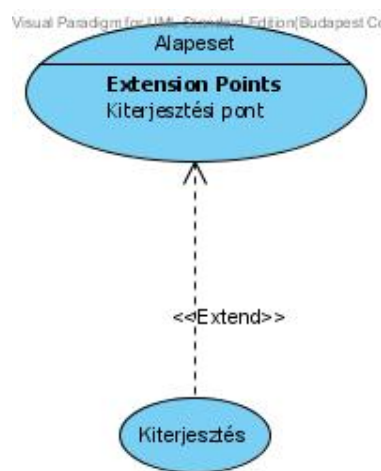


1.1. ábra: Öröklési (generalizációs / specializációs) kapcsolat

A generalizációs kapcsolat fordított olvasatban egy specializációs kapcsolatként fogható fel, s talán ez a nézőpont jobban is segíti a fogalom megértését.

A második lehetséges kapcsolati forma két használati eset között az <<extend>> (UML-ben foglalt) szócskával írható le. Itt is az egyik használati esetet alapnak, a másikat pedig kiterjesztésnek fogjuk hívni. A két használati esetet szaggatott nyíl köti össze, amely az alap használati eset felé mutat, s a nyílra írjuk rá az <<extend>> szócskát. Az alapeset a kiterjesztések nélkül is végrehajtható, azaz önmagában is teljes. A kiterjesztés néhány olyan résztevékenységet tartalmazhat, amely nem található meg az alapesetben. A kiterjesztések általában nem teljes értékű használati esetek.

Az <<extend>> kapcsolat tehát további tevékenységeket ad hozzá egy már létező használati eset tevékenységeihez. Szemléletesen szólva az alapeset előre definiált beszúrási (kiterjesztési) pontokat tartalmaz(hat), ahová a kiterjesztések beékelhetők.



1.2. ábra: <<Extend>> kapcsolat

Tegyük fel a példa kedvéért, hogy egy könyvtári rendszert modellezünk, ahol az egyik használati eset a könyvek visszaadásának folyamatát írja le. Ez lesz az alapeset, amely az alábbi tevékenységeket tartalmazza:

1. A könyvtáros beüti a kölcsönző személy azonosítóját
2. A rendszer kilistázza a kölcsönző által kölcsönvett könyvek (és a lejárat határidők) listáját.
3. A könyvtáros megkeresi a behozott könyvet a listában
4. Átveszi a behozott könyvet.

Ebben a használati esetben figyelmen kívül hagytuk annak megvizsgálását, hogy a könyvet vajon lejárat idő előtt vagy az után hozták vissza. Egy lehetséges kiterjesztés használati eset az lenne, hogy a késedelmes visszahozás miatt büntetést fizet a beiratkozott olvasó. Jelölje a kiterjesztési pontot <későKönyv>:

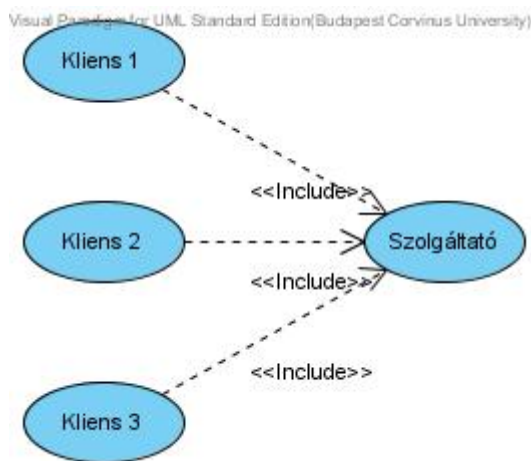
5. A könyvtáros megkeresi a behozott könyvet a listában
6. <későKönyv>

## 7. Átveszi a behozott könyvet.

A <későKönyv> azt a pontot jelöli, ahol a BüntetésFizetés használati eset tevékenységei hajtódnak majd végre.

Az <<extend>> kapcsolatot elsősorban arra használhatjuk, hogy a kivételeket kezeljük. Kivétel alatt azt értjük, amikor a folyamat lefolyása valamilyen értelemben eltér a „normálistól”.

Két használati eset között definiálható harmadik kapcsolati forma az <<include>> szócskával jellemezhető, s a „részeként tartalmaz” kifejezéssel írható le. A szaggatott nyíl, ráírva az <<include>> megnevezés, a tartalmazótól (kliens) a tartalmazott (szolgáltató) felé mutat. Ez egy olyan kapcsolatot jelöl, ahol a tartalmazott használati eset tevékenységeit egy (vagy több) terjedelmesebb használati eset is tartalmazza, mégpedig teljes egészében. Igazából akkor van értelme az <<include>> kapcsolat használatának, ha több használati eset is igényli a szolgáltatót.



1.3. ábra: <<Include>> kapcsolat

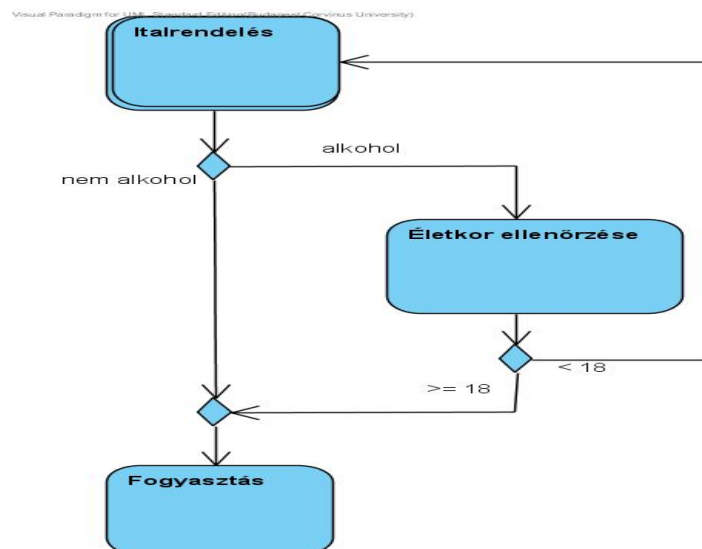
Általában azt mondhatjuk, hogy amennyiben több használati eset viselkedésében létezik valami közös, ezt a közös részt jeleníthetjük meg egy külön használati esetként. Az elmondottakból következően csak akkor célszerű az <<include>> kapcsolatot használni, ha értelmesen megadható ez az önmagában is megálló közös rész. Az <<include>> kapcsolat egy kicsit a függvényhívásra és a szubrutinok használatára emlékeztet. A kliens használati eset futása egy adott pontnál megáll, a vezérlés átadódik a szolgáltató használati esetnek, majd annak lefutása után ismét visszakérül a klienshez. Az <<include>> kapcsolat lehetővé teszi, hogy a szolgáltató használati eset tevékenységeit beékeljük a kliens munkafolyamába. Alapvető különbség az <<extend>> és az <<include>> kapcsolat között az, hogy míg az <<include>> kapcsolatban a kliens használati eset nem teljes a szolgáltató használati eset nélkül, addig az <<extend>> kapcsolatban az.

## 14.2 Tevékenységdiagram

A tevékenységdiagram, az UML diagramtípusai közül az egyik legfontosabb, egy vizsgált rendszer folyamatainak belső felépítését, szerkezetét mutatja be. Egészen pontosan azt, milyen lépések, taszkok történnek valójában egy munkafolyamatban, milyen tevékenységek végezhetők párhuzamosan, és léteznek-e alternatív útvonalak a folyamat gráfjában. Ha elkészítettük a használati eset diagramot, akkor ahhoz kapcsolódóan a tevékenység diagramok egy-egy használati eset részletesebb leírását adják.

Megjelenésükben nagyban emlékeztetnek az évtizedekkel korábban használt blokkdiagramokra. A jelölések is nagyon hasonlóak. Hasonlítanak továbbá az UML állapotgép-diagramjaira is. Az UML szerzői szándéka szerint a tevékenységdiagramok valójában az állapotgép-diagramok egy variánsát jelentik.

Tekintsük példaként a vendéglői italrendelés egy lehetséges tevékenységdiagramját!



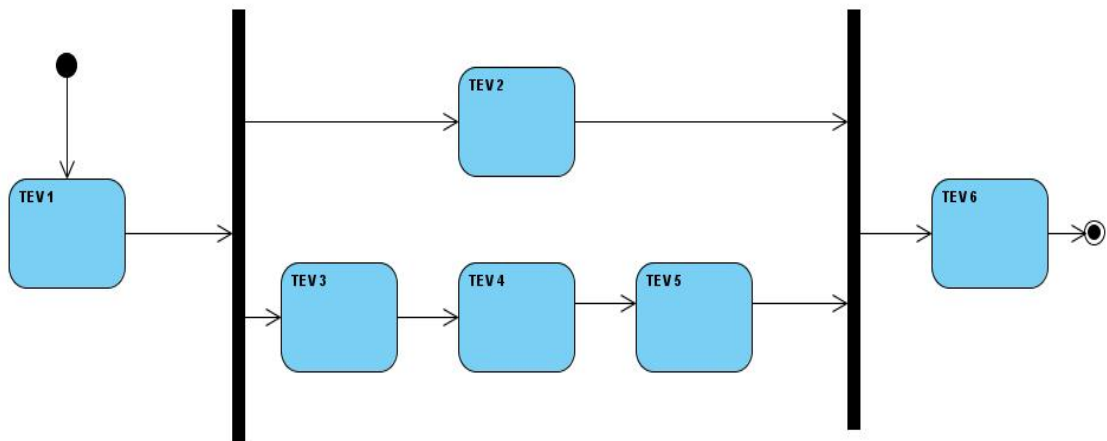
1.4. ábra: Az italrendelés tevékenységdiagramja

Az ábrán két döntési helyzet és egy egyesítési pont látható.

Általánosságban a tevékenységdiagramok az alábbi jelöléseket használják:

- Tevékenységi állapotok lekerekített téglalapokba, ún. kapszulákba írva
- Egy kezdeti és egy vagy több végállapot. A kezdeti állapotból csak egyetlen vonal indulhat ki, azaz egyetlen szálon indulhat el a tevékenységsorozat
- Feltételes döntési helyzetek, valamilyen őrfeltétellel, IF ... THEN ... ELSE
- Párhuzamosság kezelése, szétválás és egyesítés (fork/join) vastag vonallal megjelölve
- Partíciók vagy másképp pályák (swimlanes)





1.5. ábra: Szétválás és egyesítés a tevékenységdiagramon

Tevékenységi diagramokkal nagyon könnyen modellezhetünk párhuzamos munkafolyamatokat. A fenti ábrán a szétválási és az egyesítési vonal közötti tevékenységek párhuzamosan hajthatók végre. A szétválási vonalnak pontosan egy bemenete és kettő vagy több kimenete, az egyesítési vonalnak pedig kettő vagy több bemenete és pontosan egy kimenete van.

### 14.3 Objektumok és osztályok

Mint azt korábban tisztáztuk, az UML nyelv osztálydiagramjai az osztályokat és a köztük meglévő kapcsolatokat ábrázolják, azokat szemléltetik, az objektumdiagramok pedig ugyanezt teszik az objektumokkal és a köztük fennálló kapcsolatokkal. Amikor a körülöttünk levő világot, annak folyamatait szeretnénk modellezni, azt leggyakrabban objektumok segítségével tesszük. Objektumok lehetnek emberek, épületek, bankszámlák, lényegében bármi. Feltételezzük, hogy az objektumoknak létezik egyértelmű objektumazonosítójuk. Ez az azonosító különbözteti meg őket más objektumoktól. Minden objektumnak egyetlen azonosítója van, két különböző objektumnak pedig nem lehet ugyanaz az azonosítója. Az objektumokhoz hozzá tudunk rendelni bizonyos függvényeket, másképpen eljárásokat, műveleteket, metódusokat.

A vizsgálandó rendszer objektumait a hasonló tulajdonságú objektumokat tömörítő osztályokba csoportosíthatjuk. Az osztály fogalma tehát absztrahálja az objektumok valamilyen részhalmazát. A valós életben csak objektumokkal találkozhatunk (pl. Nagy Péter harmadéves hallgató vagy Nagy Péter, Oroszország cárja), s absztrakció útján jutunk el a Hallgató vagy az Uralkodó osztályhoz. Az objektumokat az osztályok példányainak fogjuk nevezni.

Ha az Olvasó valamennyire jártas az adatbázisok elméletében, akkor az alábbi párhuzam segít megérteni, mi a logikai kapcsolat az osztályok és az objektumok között. Tegyük fel, adott egy bank adatbázisában két tábla (másképpen reláció), az egyik az Ügyfelek tábla, s ebben tárolják az ügyfelek nevét, címét és személyi számát, a másik tábla pedig a Számlák, amely tábla tárolja az egyes számlák számlaszámát, egyenlegét és a számlatulajdonos személyi számát. Ha objektum-orientált szemléletben közelítünk a kérdéshez, akkor mindkét táblának egy-egy osztály felel meg, a táblákban szereplő rekordoknak pedig egy-egy objektum. Azaz úgy tekintünk az objektumokra, mint egyedi rekordokra. Az objektumoknak mezői vannak, amelyekben értékek szerepelnek. Egy rekordbeli adatelemet tulajdonságnak vagy attribútumnak nevezünk. Ezek együttesen jellemzik, karakterizálják az objektumot. Programozási nézőpontból az attribútum egy lokális változó, amely csak az adott objektumhoz tartozik.

Az osztályokat az UML nyelvben rekeszekre osztott téglalapok jelölik. Alapesetben három rekeszt tartalmaznak a téglalapok:

- Osztálynév
- Tulajdonságok, jellemzők, attribútumok felsorolása
- Műveletek, metódusok felsorolása

(Az UML megengedi további rekeszek használatát is, de ezekre most nem térünk ki.)

Számla
-számlaSzáma : int -számláló : int = 0
+létrehoz(szszaám : int) +kérSzáma : int -növel() +kérSzámláló() : int

A harmadik rekeszben szereplő műveleteket az egyik első objektum-orientált programozási nyelv, a Smalltalk metódusnak, a Java nyelv operációnak, a C++ és a C# nyelv osztály tagfüggvénynek (*class member function*) nevezi. Szokás még az eljárás és a szolgáltatás megnevezések használata is. Ami a terminológiát illeti, az elemzés fázisában inkább a műveletek, a tervezés fázisában pedig a metódusok megnevezést használjuk. Az elemzés fázisában néha elhagyjuk az attribútumok és műveletek felsorolását, s csak az osztálynév szerepel a diagramon.

A középső rekeszben szereplő attribútumok általános alakja:

láthatóság név számosság : típus = kezdetiÉrték

A harmadik rekeszben szereplő metódusok általános alakja

láthatóság metódusnév (paraméternév : paramétertípus, ...) : returnType

A láthatóság lehet publikus (+), privát (-) vagy védett (#). (Ezekről részletesebben a C# nyelv bemutatásakor lesz szó.)

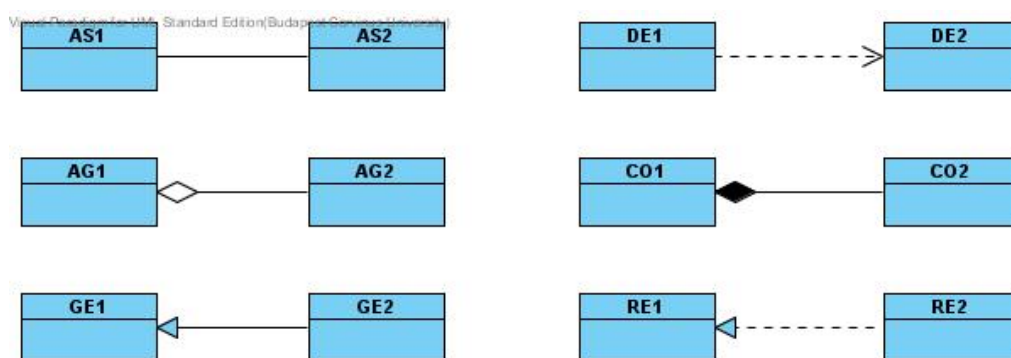
Világosan kell látnunk, hogy a műveletek, a metódusok biztosítják a rendszer funkcionalitását. A metódusoknak lehetnek paramétereik, s végrehajtás után visszaadhatnak valamilyen értéket vagy végrehajthatnak valamit, amire aktuálisan szükség van.

### 14.3.1 Kapcsolatok

UML-ben a modellelemek között meglehetősen sokféle kapcsolat létezhet. Ebben a részben az osztályok közötti kapcsolattípusokat fogjuk áttekinteni.

Ezek közül az alábbi típusokat használjuk elsősorban:

- Asszociáció vagy társítási kapcsolat, röviden társítás, s ennek speciális esetei az aggregáció, illetve a kompozíció
- Generalizáció vagy általánosító kapcsolat, röviden általánosítás
- Függőségi kapcsolat, röviden függőség, s ennek speciális esete az absztrakció (ezzel bővebben most nem foglalkozunk)



1.6. ábra: Kapcsolatok áttekintése

Az **asszociáció** egy strukturális kapcsolat két osztály (vagy használati eset) között. Ábrázolása vékony folytonos vonallal történik. (AS1 és AS2)

Az **aggregáció** az *egész — rész* típusú kapcsolatot modellezi. Szemléletes példa erre a kapcsolattípusra egy számítógép és a perifériális egységei közötti kapcsolat. Az „egész” és a „rész” entitások élettartama független egymástól. Ha eldobjuk a számítógépet, a perifériákat

egy új számítógéppel tovább használhatjuk. Kicserélhetjük a perifériákat, a számítógép ettől meg használható marad. Ábrázolása vékony folytonos vonallal történik, amelynek egyik végén egy nyíl, a másik végén pedig egy üres rombusz található, a logikailag bővebb („egész”) entitáshoz kapcsolódva. (AG1 és AG2)

A **kompozíció** az aggregáció erősebb fajtájának tekinthető. Kompozíció típusú kapcsolatban a rész élettartama az egész élettartamától függ. Ha megszűnik az „egész”, megszűnik a „rész” is. Kompozícióra példa a virág és a szirmai közötti kapcsolat. A kompozíció ábrázolása vékony folytonos vonallal történik, amelynek egyik végén egy nyíl, a másik végén pedig egy teli rombusz található, a logikailag bővebb entitáshoz kapcsolódva. (CO1 és CO2)

A **generalizáció** típusú kapcsolatban az egyik osztály az utód (alosztály), a másik a szülő (szuperosztály) szerepét játssza. Ez a kapcsolattípus nem csak osztálydiagramon, hanem komponens-, telepítés- és használati eset diagramon is megjelenhet. Ábrázolása vékony folytonos vonallal történik, s a zárt üres nyílvég az utód modellelemtől mutat a szülő modellelem felé. Általában egy szülő és több utód osztály kapcsolatát ábrázolja, bár elvileg egy utódhoz több szülő is tartozhat. (GE1 és GE2)

A **függőség** típusú kapcsolatban az egyik elemet szolgáltatónak, a másikat pedig kliensnek nevezzük, s feltételezzük azt, hogy bármilyen változás a szolgáltatóban kihat a kliensre. Függőségi kapcsolat nem csak osztálydiagramban, hanem komponens-, telepítés- és használati eset diagramban is létezhet. Ábrázolása szaggatott vonallal történik, s a nyílt nyílvég a kliens elemtől a szolgáltató elem felé mutat. (DE1 és DE2)

Az **absztrakció** típusú kapcsolat a függőség azon esete, amikor a kliens elem sokkal részletesebb, mint a szolgáltató, vagyis az absztrakció különböző szintjein találhatók. (RE1 és RE2)

#### 14.3.1.1 Asszociáció

Az asszociáció osztályok közötti szemantikus kapcsolat, amely ezen osztályokat bizonyos értelemben rokonítja egymással. Asszociáció lehetséges kettő vagy több osztály között.



1.7. ábra: Asszociációs kapcsolat, személy - gépkocsi

Formailag a két osztályt reprezentáló téglalapot egy egyenes vonallal kötjük össze. A vonal fölé egy ige (*birtokol*) kerülhet, alá pedig egy igei szókapcsolat (*birtokában van*). Az ige után tehetünk egy jobbra mutató nyílvéget (►), az igei szókapcsolat elé pedig balra mutató nyílvéget (◄), de ennek használata nem kötelező.

A téglalapokhoz illeszkedően szerepeltet(het)jük a kapcsolat számosságát. A fenti példában egy személynek lehet, hogy nincs gépkocsija, lehet, hogy egy van neki, s lehet, hogy több. Megfordítva, egy gépkocsinak lehet egy tulajdonosa, de lehet több is. Ezért szerepel a *Személy* osztály mellett 1..\*, és a *Gépkocsi* osztály mellett 0..\* számosság.

Általánosan, számosság lehet

Nulla – egy : 0..1

Nulla – sok: 0..\*

Egy – egy: 1..1 vagy 1

Egy – sok: 1..\*

M – N: M..N

Tetszőleges természetes számokból álló növekvő sorzat

Az asszociációs kapcsolat lehet reflexív, azaz ugyanazt az osztályt társítja saját magához. Egy számítógép tetszőleges számú számítógéppel köthető össze.

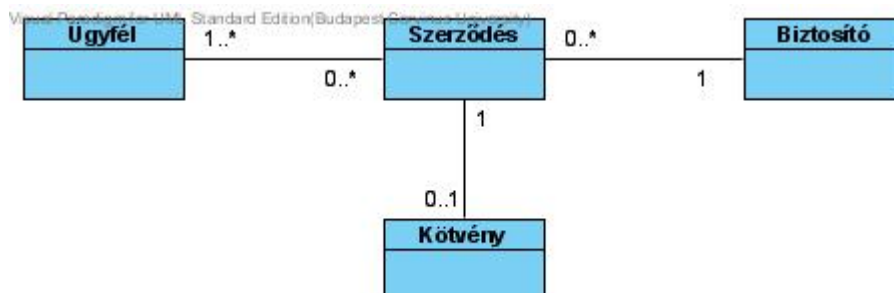


1.8. ábra: Reflexív asszociációs kapcsolat

Nézzük meg, hogyan lehet a biztosító társaság és az ügyfél kapcsolatát asszociációs kapcsolatként modellezni!

Tegyük fel az alábbiakat :

- A biztosító társaság szerződéseket köt az ügyfelekkel.
- A szerződések a biztosító társaság és egy vagy több ügyfél között jönnek létre.
- A biztosító társaságnak és bármelyik ügyfélnek tetszőleges számú szerződése lehet.
- Egy adott szerződést tetszőleges számú ügyféllel köthet meg a biztosító.
- Minden szerződéshez tartozik egy kötvény, de ez gyakran csak a szerződéskötés után jelentős késéssel készül el. (Ennek a ténynek akkor van jelentősége, ha a káresemény közvetlenül a szerződéskötés után következik be.)



### 14.3.1.2 Hármas asszociáció

Társulási kapcsolat nem csak két, hanem három vagy több osztály között is lehetséges.

Tegyük fel, hogy iparosok dolgoznak egy családi ház építkezésén. Az alábbi táblázat mutatja, hogy melyik iparos melyik helyiségben milyen munkát végez.

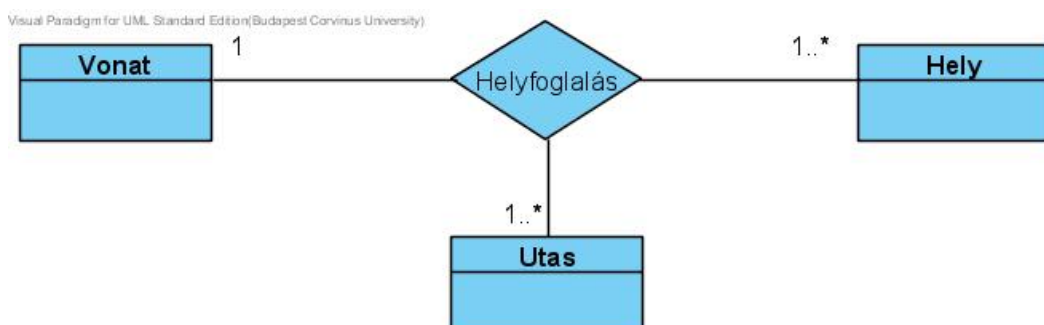
Iparos	Helyiség	Munka
András	konyha	festés
András	nappali	festés
András	fürdőszoba	festés
Béla	konyha	parkettázás
Béla	fürdőszoba	parkettázás
Csaba	nappali	fűtésszerelés
Csaba	konyha	fűtésszerelés

Az UML az ilyen típusú társulások modellezéséhez rombuszt használ, s nem háromszög alakzatban köti össze az osztályokat.



1.10. ábra: Három osztály asszociációja

További példát szolgáltat hármas társulásra a vonat – utas – ülőhely modell



### 1.11 ábra: Három osztály asszociációja

#### 14.3.1.3 Aggregáció és kompozíció

Az aggregáció az asszociáció speciális formája, ahol bizonyos értelemben „egész” és „rész” objektumokról beszélhetünk. (Az UML angol nyelvű leírásában ezt a kapcsolatot *whole—part*, *consists—of*, *is—part—of*, *has—a* kapcsolatként írják le.) Az aggregálásra példa egy személygépkocsi, amelynek egyebek mellett van motorja, futóműve, karosszériája és négy kereke. Egy további példa az aggregálásra a

Sportklub – Csapatok – Játékosok

kapcsolat, amit az alábbi ábra mutat:



1.12 ábra: Aggregáció és kompozíció

A diagramon látható nyílvégi kis rombusz lehet üres vagy teli. Az előbbi esetben gyenge vagy osztott aggregálásról, az utóbbiban erős vagy összetett aggregálásról beszélhetünk. Gyenge típusú aggregálás esetén az objektum több „egészhez” is tartozhat, míg erős típusú aggregáláskor csak egyhez. Az előbbi esetben ha az „egészt” töröljük, annak következményeként a részek nem törölődnek, míg erős típusú aggregálás esetén igen. Ha egy sportklub megszűnik (FC Barcelona), akkor megszűnnek a klub keretében működő csapatok (labdarúgás, kosárlabda, teke stb.) is, de ha a kosárlabda csapat megszűnik, attól egy játékos még játszhat tovább a tekecsapatban, hiszen egy játékos egyszerre több csapatnak is tagja lehet.

Kompozícióra (erős aggregálásra) példa:

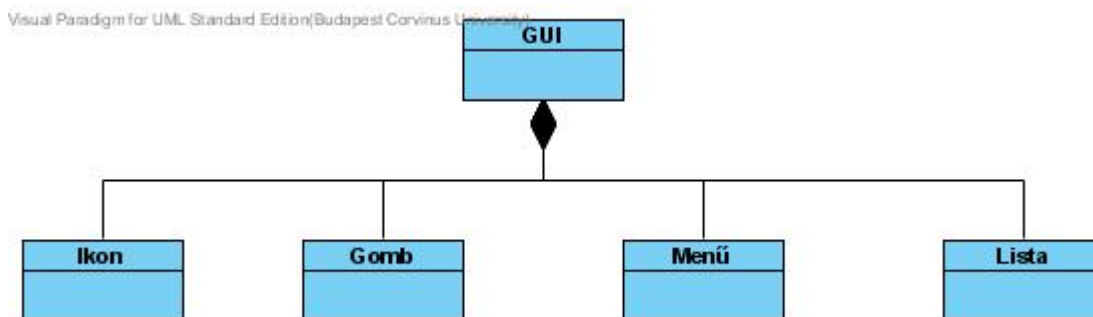


1.13 ábra: Kompozíció: virág és szirmai

A virág a szirmok kompozíciójaként fogható fel. Egy virágon lehet, hogy nincs szirm egyáltalán, s lehet, hogy több is van. Egy adott szirm azonban csak egyetlen egy virághoz tartozhat. Ezt másként úgy fogalmazhatjuk meg, hogy az összetett aggregálásban a részek az

egészben „élnek”. Az „egész” oldalon a számosság lehet 0 vagy 1, a „rész” oldalon viszont tetszőleges intervallum.

A kompozíció egy másik lehetséges ábrázolása a faszerkezet. Ezt akkor használjuk, ha az egésznek egynél több része van.

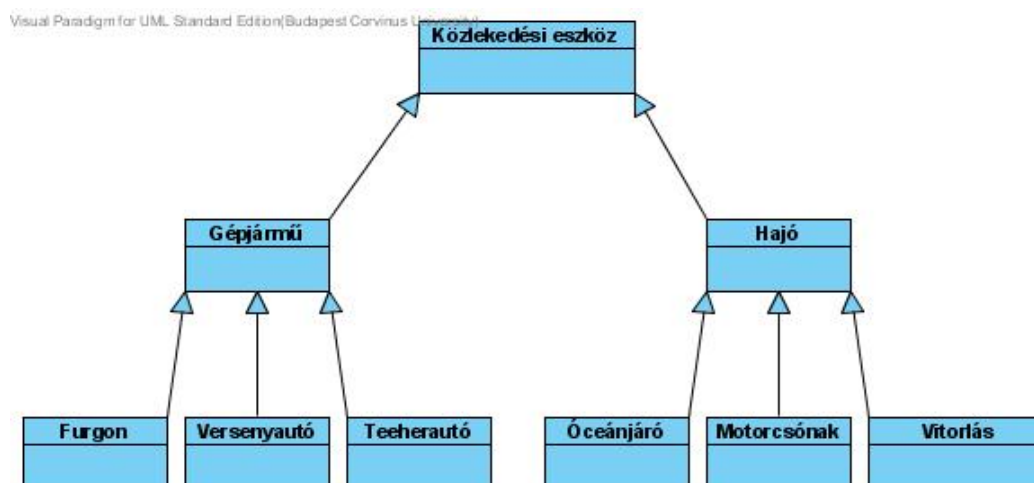


1.14. ábra: Kompozíciós faszerkezet

A grafikus felhasználói felület (GUI) sokféle menüt, gombot, ikont és listát tartalmazhat.

#### 14.3.1.4 Általánosítás

Az általánosítás (vagy másképp generalizáció) két osztály, egy általános és egy speciális entitás, az ún. szülő és utód osztály közötti különleges kapcsolatot modellezi. (Az UML angol nyelvű leírásában ezt *is—a—kind—of* kapcsolatként írják le.) A speciális osztály rendelkezik az általánosabb osztály tulajdonságaival.



1.15. ábra: Általánosítás / generalizáció



A példa azt mutatja, hogy a Gépjármű osztály a Közlekedési eszköz osztály alosztálya, s a Teherautó osztály szuperosztálya. Az általánosítás lehetővé teszi a hierarchikus osztályozást, ami az UML modellezés egyik lényeges eleme.

Gyakran létezik valamilyen hasonlóság egy modell különböző osztályai között, két vagy több osztálynak lehetnek közös attribútumai és/vagy műveletei, metódusai. Mivel felmerül az a természetes igény, hogy ne kelljen többször is ugyanazt a programrészletet megírni, ezért kihasználhatjuk az öröklés mechanizmusát. Amikor az A osztály örököl a B osztálytól, azt mondjuk, hogy az A osztály a B osztály egy alosztálya, megfordítva, B az A szuperosztálya. Az UML nyelvben az öröklés jelölésére a zárt végű üres nyílhegyet használjuk, s a nyíl az alosztálytól a szuperosztály felé mutat. Tegyük fel példaként, hogy a modellünkben van egy Hallgató osztály és egy Tanár osztály. Az előbbi attribútumai között szerepel a

- név
- telefonszám
- email cím
- hallgatói azonosító
- évfolyam
- utolsó félévi átlag

az utóbbi attribútumai között pedig a

- név
- telefonszám
- email cím
- beosztás
- iroda
- fogadóóra.

Vezessük be a Személy osztályt, melynek attribútumai az alábbiak lesznek

- név
- telefonszám
- email cím

Ekkor e három osztály egy öröklési hierarchiába (hierarchikus láncba) szervezhető. A gyökérosztálya a Személy lesz a megadott attribútumokkal, a Hallgató és a Tanár pedig alosztályok, amelyek öröklik a szuperosztály attribútumait, s emellett rendelkeznek saját attribútumokkal is. Paradox módon a gyökérosztály az öröklési hierarchia legfelső osztálya.

#### 14.3.1.5 Betokozás

Mint azt a korábbi fejezetben tárgyaltuk, a betokozás vagy bezárás (*encapsulation*) egy nagyon fontos tervezési elv az objektum-orientált alkalmazásfejlesztésben. A betokozás fogalma leegyszerűsítve azt jelenti: nem szükséges tudnunk, hogy valami miként van implementálva ahhoz, hogy képesek legyünk azt használni. Ha ezt az elvet követjük, akkor a fejlesztés során bármikor megváltoztathatjuk egy komponens implementációját anélkül, hogy ez a rendszer más komponenseit zavarólag érintené, feltéve, hogy a szükséges és a biztosított interfészek változatlanok maradnak. A részletek elrejtésének elve és gyakorlata a mindennapi élet számos területén megfigyelhető. Így nem minden bankkártya-tulajdonos tudja pontosan, hogyan is működik egy kártyaautomata, de bízik abban, hogy az kiszolgáltatja neki azt a pénzmennyiséget, amire aktuálisan szüksége van. Feltéve persze, hogy rendelkezik azzal az összeggel a számláján. A bank bármikor változtathat a kártyaautomata programján, átszervezheti az adatbázisait stb., de ez magát a szolgáltatást nem fogja, pontosabban nem szabad befolyásolnia.

Egy osztályt azért zárunk be, mert biztosítani akarjuk, hogy annak objektumai csak az osztállyal együtt definiált műveleteken keresztül változhassanak. Más szavakkal, a betokozás megakadályozza az objektumokhoz való külső hozzáférést, s azt, hogy az objektumokat meg lehessen változtatni az osztály megalkotójának szándéka ellenében. Csak az osztály metódusai módosíthatják az osztálybeli objektumokat közvetlenül.

A betokozás elve az egyik garanciája annak, hogy nagyméretű szoftverrendszereket lehessen létrehozni, amelyekben viszonylag kevés hiba van. Ha ugyanis ez az elv nem érvényesülne, s a szoftver különböző részei között nagyfokú kapcsolódás állna fenn, akkor egy kicsi változtatás valamelyik modulban megkövetelné a változtatások átvezetését sok másik modulban is, ami gyakorlatilag lehetetlenné tenné ezek nyomon követését.

#### 14.3.2 Szekvenciadiagram

A szekvenciadiagramok a vizsgált rendszer entitásai közötti kommunikációt, azaz az üzenetváltásokat ábrázolják, jól látható időbeli sorrendben. A szekvenciadiagramok a rendszer viselkedési nézetéhez tartoznak. A hangsúly azon van, hogyan küldenek és fogadnak üzeneteket a rendszer objektumai. A szekvenciadiagramok kétdimenziós ábrázolások, az egyik (vízszintes) tengely az objektumokat (és/vagy osztályokat), a másik (függőleges) tengely az időt reprezentálja. A vízszintesen felsorolt objektumok nevét kis téglalapokba írjuk. Az időt jelölő függőleges szaggatott vonalon, amit életvonalnak nevezünk, helyezkednek el az aktív periódusokat jelölő vékony kis álló téglalapok.

A diagramon telt végű (—►) nyilak mutatják az üzeneteket. Amikor egy üzenet megérkezik, a fogadó objektum aktív állapotba kerül, s vagy maga hajt végre bizonyos utasításokat, vagy válaszüzenetre vár egy harmadik objektumtól, amelynek ő küldött el egy üzenetet. A válaszüzeneteket telt végű szaggatott vonallal jelölik. Az üzeneteket szokás megszámozni, bár igazából erre nincs szükség, mivel a diagram sugallja a sorrendiséget.

További lehetőségek:

- Feltételek társulhatnak az üzenetekhez
- Amennyiben az üzenet neve előtt egy \* szimbólum szerepel, úgy az üzenet ismételt elküldésre kerül, azaz iterált üzenetről beszélhetünk.
- Lehetséges, hogy egy objektum sajátmagának küld üzenetet
- Léteznek szinkron és aszinkron üzenetek

Formálisan egy szekvenciadiagram elkészítéséhez az alábbi lépések megtétele szükséges:

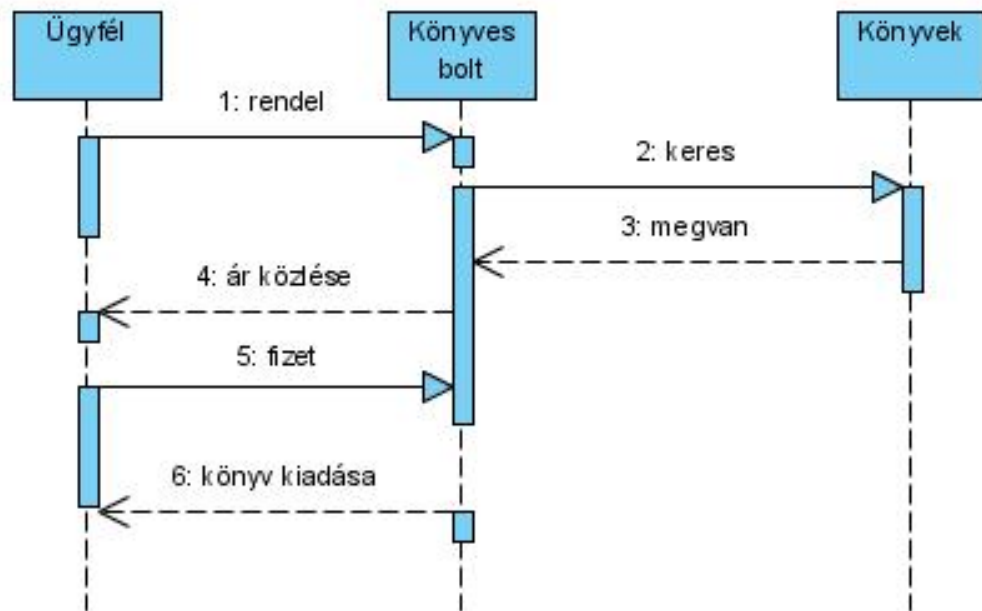
1. A rendszer objektumai közötti kommunikáció (üzenetváltások) forgatókönyvének leírása
2. Az objektumok (osztályszeretek) megadása
3. Az életvonal meghúzása valamennyi objektumra
4. Az aktív időszakok megrajzolása
5. Az üzeneteket (és a válaszokat) jelölő nyilak megrajzolása
6. Iteráció, a modell finomítása

Nézzük, pontosan mit is jelentenek a fenti fogalmak!

Az életvonal (lifeline) egy olyan grafikus jelölés, amely az objektumok (osztályszeretek) létezését mutatja, ahogy időben haladunk előre. Az életvonalat szaggatott vonallal jelöljük. Az első üzenet felett indul, és az időben utolsó üzenet alatt, a szekvenciadiagram alján ér véget.

Egy objektum további objektumokat képes üzenetek által létrehozni. Az életvonal elkezdődik, ha az objektum létrehozásra kerül (ezt egy üres hegyű (—>) szaggatott vonal jelöli a diagramon), s véget ér, ha az osztályszeret megsemmisül. A megsemmisülés tényét egy „X” szimbólum jelöli annál az üzenetnél, amely azt (azaz a megsemmisülést) eredményezte.

Példaként tekintsük egy könyvvendelés szekvenciadiagramját!



1.16 ábra: Szekvenciadiagram egy könyvrendelésre

Az 1.16 ábra a bolti könyvrendelés leegyszerűsített folyamatát mutatja. Három objektum szerepel az ábrán:

1. Vevő
2. Bolt
3. Keresett könyv.

Az ábrán megjelenített üzenetek:

1. A vevő (telefonon) megrendeli a könyvet a könyvesbolttól.
2. A bolti alkalmazott ellenőrzi az adatbázisban a könyv meglétét, illetve az árát.
3. Ha megvan a könyv, az eladó feljegyzi a könyv árát.
4. Ezt az árat közli a vevővel.
5. A vevő kifizeti a könyv árát.
6. S megkapja az óhajtott könyvet.

### 14.3.3 Kommunikációdigram

Az UML korábbi szabványaiban kollaborációs diagramként emlegetett ábra több szempontból is hasonlít a szekvenciadiagramra. Az üzeneteket kötelező megszámozni, míg a szekvenciadiagramban az időt expliciten ábrázoljuk, felülről lefelé haladva. Nyilvánvaló, hogy az időrendiséget a szekvenciadiagramból könnyebben tudjuk leolvasni. Tény, hogy egy szekvenciadiagram elkészítése után már könnyű feladat egy kommunikációdigram

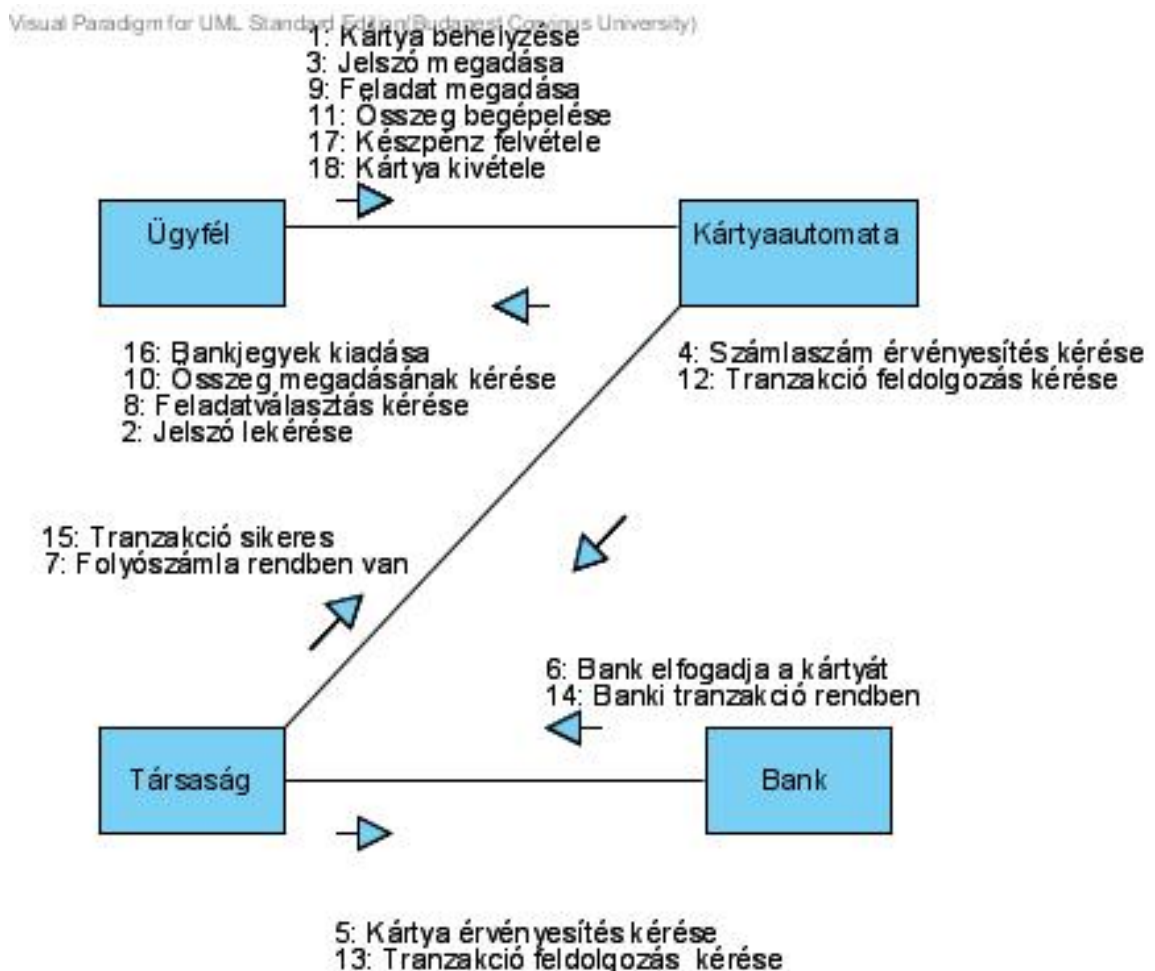
megszerkesztése, és fordítva is igaz ez az állítás. Arra a kérdésre, mikor érdemes szekvenciadiagramot, illetve kommunikációdigramot használni, azt válaszolhatjuk, hogy az előbbit akkor, ha csak az üzenetek sorrendje az érdekes, míg az utóbbit akkor, ha inkább a kommunikáció mibenlétét szeretnénk megérteni.

A kommunikációdigram használatát egy bevásárlóközpontban működtetett bankjegykiadó automata példájával illusztráljuk. (A bevásárlóközpontnak a történetben csak annyi a szerepe, hogy az automatát nem maga a bank üzemelteti.)

Az 1.17. ábrán látható kommunikációdigram entitásai az alábbiak:

Ügyfél – Automata – Társaság - Bank

A Bank bocsátotta ki a plastikkártyát, s a Társaság üzemelteti az automatát. Értelmszerűen minden tranzakció keresztülmegy a társaságon.



1.17 ábra: Kommunikációdigram egy bankjegykiadó automatára

Az 1.17. ábrán számokkal ellátott üzenetek a következők:

1. Kártya behelyezése az automatába
2. Jelszó kérés
3. Jelszó megadása
4. Számlaszám érvényesítés kérése
5. Kártya érvényesítés kérése
6. Bank elfogadja a kártyát
7. Folyószámla rendben van
8. Feladatválasztás kérése
9. Feladat megadása
10. Összeg megadásának kérése
11. Összeg begépelése
12. Tranzakció feldolgozásának kérése a Társaságtól
13. Tranzakció feldolgozásának kérése a Banktól
14. Banki tranzakció sikeres
15. Tranzakció sikeres
16. Bankjegyek kiadása
17. Készpénz felvétele
18. Kártya kivétele az automatából

Az objektumokat az ábrán nem szükségszerűen vízszintesen helyezük el, hanem abban a formában, ami legjobban segíti az objektumok közötti kölcsönhatások (kommunikáció) megértését. A kommunikáció irányát nem az objektumokat összekötő nyilakra írjuk rá, hanem mellettük helyezük el a kisebb nyilakat.

### **Állapotgép-diagramok**

Ha egy rendszer viselkedését kívánjuk leírni, a tevékenység- és kommunikációdigramok mellett gyakran szükség van az ún. állapotgép-diagramok használatára is, amelyek az elnevezésből következően egy objektum állapotait és azokat a belső és külső eseményeket modellezik, amelyek változásokat idéz(het)nek elő bennük. Ez a diagramtípus különösen hasznos valós idejű (real-time) és biztonság-kritikus rendszerek fejlesztésében, ahol a vizsgált rendszert a környezetéből érkező külső hatások, ingerek vezérlik. Az állapotgép-diagramok irányított gráfok használatával azokat a rendszerállapotokat és ingereket, eseményeket mutatják be, amelyek az egyik állapotból egy másikba történő átmenetet okozzák. Természetesen feltételezzük, hogy a lehetséges állapotok száma véges.

Illusztráció gyanánt tekintsünk egy igazán egyszerű rendszert, egy villanykapcsolót. Ennek két állapota létezik: a leoltott és a felgyújtott állapot, röviden LE és FEL. Azzal a logikus feltevéssel élünk, hogy a rendszer bármely időpillanatban e két lehetséges állapot egyikében található. Kétféle ingert, eseményt (esetünkben cselekvést) különböztethetünk meg, amelyek

állapotváltozást idéznek elő a rendszerben, ez a leoltás és a felgyújtás eseménye. Különösen bonyolultabb rendszerek esetén célszerű táblázatba rendezni a rendszerállapotokat, az ingereket (eseményeket) és az átmeneteket.

Állapot	Felgyújtás	Leoltás
LE	FEL	-
FEL	-	LE

Amennyiben nincs változás az állapotban vagy nem értelmezhető az átmenet, a „—” jel szerepel a táblázatban.

Az állapotgép-diagramokon a lekerekített téglalapok jelölik az állapotokat, közéjükbe írva az állapot nevét, a folytonos, nyíllal ellátott vonalak pedig magukat az átmeneteket. Rendszerállapot nem csak passzív lehet, mint a fenti példában, hanem aktív is. Ilyenkor az állapotonév alatt a „do” szócska és a tevékenység megnevezése szerepel. Az állapotgép-diagramokon két kitüntetett állapot szokott szerepelni, a kezdeti és a végállapot.

A diagramok másik eleme az átmenet, amely két rendszerállapot között létezik. Ezek egyikét forrás-, másikét célállapotnak nevezzük. Az átmenetet jelölő nyílra nem csak az eseményt, hanem annak logikai feltételét is ráírhatjuk. Így az átmenet csak akkor következik be, ha a logikai feltétel értéke igaz.

Tegyük fel a példa kedvéért, hogy egy DVD-lejátszó a modellezendő rendszer. Három állapotot különböztethetünk meg: lejátszás, szünet és leállítás. Az állapottáblázat az alábbi lesz:

Állapot	Lejátszás gomb megnyomása	Szünet gomb megnyomása	Stop gomb megnyomása
Lejátszás		Szünet	Leállítás
Szünet	Lejátszás		Leállítás
Leállítás	Lejátszás		

Könnyen belátható, hogy Leállítás → Lejátszás átmenet csak abban az esetben valósul meg, ha van lemez a lejátszóban.

Az állapotgép-diagramok kizárólag akkor használatosak, amikor olyan objektumokat kívánunk modellezni, amelyek eltérő módon viselkednek aszerint, hogy aktuálisan melyik rendszerállapotban vannak. Ha egy objektum nem változtatja az állapotát életciklusa során, akkor nem indokolt állapotgép-diagram készítése. Gyakorlati problémákban, amikor nagyméretű rendszereket modelleznek, mind a lehetséges állapotok, mind az ingerek száma meglehetősen nagy, akár több száz is lehet.

## **15 Az alkalmazásfejlesztés folyamata, szoftver-életciklus modellek**

A szoftverrendszerek tervezésében és megvalósításában kevésbé járatos emberek rendszerek fejlesztését gyakran azonosítják a számítógépek programozásával. Ebből következően úgy gondolják, hogy a rendszerfejlesztés lényegében semmi más, mint programírás. Ha közelről megvizsgálunk egy pár informatikai projektet, különösen a nagyobb méretűeket, akkor azt fogjuk tapasztalni, hogy az elvégzett munkának átlagosan 10%-a fordítódott ténylegesen programírássra, teszteléssel együtt kb. 30%-a, a fennmaradó 70% ugyanis egyéb tevékenységeket takar. Az említett arányok természetesen csak tájékoztató jellegűek. E számok projektről projektre változnak, mégpedig meglepően nagy szórással, másrészt a különböző számítások nem mindig veszik figyelembe az összes, rendszerfejlesztéssel kapcsolatos tevékenységet. Ez különösen a már elkészült szoftverrendszerek karbantartására, az ún. szoftverevolúcióra vonatkozik, amely fázis pedig nagyon fontos része a rendszerfejlesztésnek.

A szoftverrendszerek fejlesztésének folyamatát 7 alapvető fázisra bonthatjuk, tekintet nélkül arra, hogy maga a fejlesztés ténylegesen milyen módszerrel történik:

1. célok meghatározása, projektindítás,
2. elemzés, a rendszerrel szemben támasztott követelmények meghatározása,
3. tervezés,
4. kivitelezés, implementálás, integráció,
5. validáció,
6. telepítés, átadás
7. üzemeltetés, karbantartás, evolúció.

Ez a hét fázis alkotja együtt a alkalmazásfejlesztés életciklusát. (Természetesen ezek a fázisok összefonódhatnak, összefésülődhetnek, s nincs arról szó, hogy egy fázist csak és kizárólag akkor lehet elkezdni, ha a megelőző fázis már teljes egészében befejeződött.) Az életciklus-modell azokat a lépéseket és feladatokat tartalmazza, amelyek minden rendszerfejlesztési projektben közősek. Ugyanis, tekintet nélkül arra, hogy milyen típusú és mekkora méretű informatikai rendszert akarunk fejleszteni, mindig szükség van a jelenlegi rendszer tanulmányozására – legyen az manuális (kézi) vagy automatizált –, a megrendelői követelmények megfogalmazására, megvalósíthatósági tanulmány készítésére, szükség van az egész informatikai projekt megtervezésére, költségterv készítésére, dokumentálásra, validálásra, a felhasználók kiképezésére és betanítására, telepítésre stb.

Lássuk, hogy az életciklus egyes fázisai milyen tevékenységeket tartalmaznak!



A **projektindítás** fázisa a gyakorlatban jó pár lépést fog össze. Ide tartozik minden olyan tevékenység, amely előkészíti a *tényleges* informatikai fejlesztést, ami a rendszerelemzéssel indul. A projektek elindításának ötlete többféleképpen merülhet fel. Talán leggyakrabban az a motivációja egy informatikai fejlesztés kezdeményezésének, hogy a felhasználók elégedetlenek a működő rendszerrel – ami lehet teljesen vagy csak részben automatizált esetleg teljesen manuális –, s javasolják akár a régi rendszer lényeges módosítását, akár egy új rendszer kiépítését. A projektek elkezdésének másik gyakori oka új üzleti lehetőségek felmerülése, amelynek révén a vállalat versenyelőnyre képes szert tenni. Erre talán a legjobb példa az e-business területe.

Bármilyen legyen is a motiváció, mindenképp szükséges egy alapos és körültekintő vizsgálat a megvalósítás lehetőségeit és esélyeit illetően. Ennek a vizsgálatnak fontosabb megállapításait tartalmazza az ún. megvalósíthatósági tanulmány (*feasibility study*).

A szoftverprojekt menedzserének dolga az, hogy biztosítsa (megtervezze, ütemezze, értékelje, felügyelje stb.), a szoftverprojekt megfelel a költségvetési és ütemtervi megszorításoknak, s olyan rendszer kerül leszállításra és átadásra, amely jelentősen hozzájárul a szervezet, cég, közintézmény üzleti céljai eléréséhez. Szoftverprojektek menedzselése feltétlenül szükséges, mert a szoftverfejlesztés mindig kemény pénzügyi (költségvetési) és ütemezési (időbeli) korlátok között folyik. A projektmenedzser feladatai közül az alábbiak a legfontosabbak:

- Projektjavaslatok írása
- Projektek tervezése és ütemezése
- Projektkockázatok azonosítása, elemzése és figyelése
- Projektköltségek figyelemmel kísérése
- Projektek felülvizsgálása
- Részrtvevők kiválasztása és teljesítményük értékelése
- Beszámolójelentések írása és prezentációk készítése

A projektindítás egyik legfontosabb tevékenysége

- a szükséges emberi erőforrások (ember – nap, man – month),
- az időtartam (naptári hónapok),
- a költségek

lehetőség szerinti minél pontosabb megbecslése.

(Ha tegyük fel egy feladat elvégzésének emberi erőforrás igénye 2000 ember - nap, akkor ez azt jelenti, hogy például 20 főnek 5 hónapig, 100 főnek 4 hétig, 400 főnek pedig egy hétig kellene dolgoznia a megoldáson. Természetesen a valóságban a feladat természete, a részfeladatok egymásra épülése kizárhat bizonyos kombinációkat.)

Ezek a becslések leggyakrabban a korábbi projektek tapasztalataira építenek. Viszonylag egyszerű a helyzet, ha az új projekt – a rendeltetését és méretét illetően – hasonlít korábbi fejlesztésekre. Bár két egyforma projekt nincs, mégis valószínű, hogy az új vállalkozás nagyságrendileg ugyanannyi erőforrást fog igényelni, mint a korábbiak. A helyzet lényegesen bonyolultabb, ha olyan projekt beindítását tervezik, amelyhez hasonlót (legalábbis a projekt résztvevői) nem ismernek. Ilyenkor azt a taktikát szokták alkalmazni, hogy megkísérlik a fejlesztés várható folyamatát egészen apró részekre bontani, s e részekre külön-külön készítenek becslést múltbeli tapasztalatok felhasználásával és különféle szoftvermértékek alkalmazásával.

A projektindítás további fontos része a fejlesztés ütemtervének elkészítése. Ez technikájában megegyezik bármilyen más projekt ütemezésével. Elsőként a részfolyamatokat kell azonosítani, hozzájuk rendelve a végrehajtás időtartamigényét. Majd a részfolyamatok függőségi viszonyait kell feltérképezni, ami azt jelenti, hogy minden részfolyamatra meg kell mondani, mely tevékenységeknek kell azt megelőzniük; illetve mely tevékenységek elkezdésének feltétele, hogy a szóbanforgó tevékenység részben vagy teljesen befejeződjön. Fontos megjegyezni, hogy az ütemezésnek nem csak az időtartamra, hanem az emberi erőforrásokra is vonatkoznia kell. A rendszerfejlesztés első fázisának természetesen része a fejlesztői csapat összeállítása.

Próbáljuk megfogalmazni miben is különbözik az alkalmazásfejlesztés, a szoftverrendszerek fejlesztése (*software engineering*) más (mérnöki) tevékenységektől. (Az eltérések, a különbségek szem előtt tartása azért is kívánatos, mert nagyon sok, hagyományosan menedzselte szoftverprojekt bizonyult sikertelennek az informatika rövid, alig 60 éves történetében. A kudarc aránya lényegesen felülmúlja az ipari átlagot.)

Legalább négy okot tudunk megnevezni, ami miatt a szoftvertervezők munkája nem olyan jellegű, mint a hagyományos mérnököké:

1. A szoftver nem egy kézzelfogható termék. Ha egy toronyház építése csúszik, mind a kivitelezők, mind a megrendelők látják az elmaradást. A szoftverprojekt csúszását sokkal nehezebb felismerni. Ezért szoktak a projektek megrendelői ragaszkodni ahhoz, hogy időről-időre valamilyen közbenső termék (*deliverable*) átadásra kerüljön, bár ez nem oldja meg teljesen a problémát.
2. A szoftver egyedülállóan rugalmas termék. Ha egy épületet eredetileg kórháznak terveztek, menetközben nehéz azt mondjuk sportstadionná átépíteni. Szoftverprojekteknél ilyen méretű irányváltás gyakran előfordul.
3. A szoftverprojekteknél jelentősen különbözhetnek a korábbiaktól. Bár a mérnöki tudományok fejlődése is gyors, az még sem közelíti meg az informatika fejlődési ütemét. A gyakori technológiai váltások a korábbi projekttapasztalatokat rendkívül hamar elavulttá teszik.

4. Nehéz megjósolni mikor fog egy szoftverprojekt fejlesztési problémákba botlani. Az informatikában még nem értük el azt a szintet, hogy a terméktípusok és az őket előállító szoftverfolyamatok kapcsolatát tökéletesen megértsük.

Az **elemzés** fázisában a projekt résztvevői az üzleti problémát és annak környezetét tanulmányozzák, elemzik abból a célból, hogy egzakt módon meg tudják fogalmazni a leendő informatikai rendszerrel szemben támasztott elvárásokat és követelményeket, függetlenül attól, hogy létezik-e már valamilyen informatikai rendszer, vagy a semmiből kell azt megkonstruálni. Az elemzés fázisának fontosságát nem lehet eléggé hangsúlyozni, bár a gyakorlatban erről igen gyakran megfeledkeznek. A tapasztalatok szerint az elindított rendszerfejlesztési projekteknek csak töredéke fejeződik be a kitűzött határidőre és az eltervezett költségvetésen belül. Ennek az egyáltalán nem megnyugtató ténynek az az elsődleges magyarázata, hogy az elemzők az esetek jelentős részében nem a kellő körültekintéssel végezték munkájukat. Fontos megjegyezni, hogy az elemzés elsősorban nem informatikai feladat, hanem szakterületi (gazdasági, mezőgazdasági, orvosi stb.). A hangsúly a szakterületi problémán van, és nem a számítógépes programokon. A rendszerelemzőkkel kapcsolatban támasztott egyik legfontosabb elvárás a jó kommunikációs készség, hiszen a követelmények megfogalmazása főként a felhasználókkal folytatott megbeszélések és a velük készített interjúk révén lehetséges.

A **tervezés** fázisa az alternatív megoldási javaslatok értékelését és a véglegesített megoldás részletekbe menő megadását, az ún. specifikációk rögzítését tartalmazza. A tervezés során a hangsúly az üzleti célokról, problémákról és környezetről fokozatosan áttolódik a szigorúan vett informatikai kérdésekre, mint például képernyőtervek, riportformátumok elkészítése, adatbázisok, illesztések, programok tervezése stb. Megkülönböztetünk logikai és fizikai tervezést. A tervezés fázisának végterméke, a specifikációkat rögzítő dokumentum lesz a kiinduló pontja a programozók munkájának.

A **kivitelezés** vagy **implementálás** fázisa a megoldás fizikai megkonstruálását jelenti, s annak üzembe állítását a megrendelő telephelyén. Ehhez a fázishoz tartozik a programozás vagy kódolás.

A **validálás** fázisa mindazokat az ellenőrző folyamatokat tartalmazza, amelyek biztosítják, hogy a szoftverrendszer megfelel specifikációjának és kielégíti a felhasználók igényeit. A validálás legfontosabb tevékenysége a tesztelés, ami az egyes programrészek, azaz modulok ellenőrzésétől az együttműködést biztosító ún. integrációs teszteken keresztül a teljes rendszer (azaz szoftver, hardver, hálózatok és adatbázisok) megfelelő viselkedésének ellenőrzéséig terjed. Fontos megjegyezni, hogy a bár a validációt együtt szokás emlegetni a verifikációval, ez a két fogalom némileg mást jelent. Verifikáció alatt azt értjük, hogy vajon megfelel-e a szoftverrendszer a specifikációjának, illetve azt, hogy eleget tesz-e a funkcionális és nem-funkcionális követelményeknek, míg a validáció annak ellenőrzése, hogy megfelel-e a

szoftverrendszer a megrendelő, az ügyfél, a vevő valódi elvárásainak. A különbség tisztán kivehető.

A 20. századi menedzsmenttudomány legnagyobb hatású gondolkodója, Peter Drucker szerint az eltérés a következőképpen fejezhető ki közérthető formában:

Verification — *"Are we building the product right?"*

Validation — *"Are we building the right product?"*

A **telepítés** fázisában kerül sor egyebek mellett a rendszer átadására – amely lépés meglepően sokféleképpen történhet –, a végleges dokumentáció elkészítésére és a felhasználók betanítására.

Az **üzemeltetés** fázisa egyrészt felöleli a folyamatos napi működtetést, annak támogatását, másrészt az esetleges hibák kijavítását, a rendszer tökéletesítését, a megváltozott körülményekhez történő adaptálását. Ez a fázis, amit másképp rendszerevolúciónak is neveznek, általában hosszú ideig tart és költséges, s azzal ér véget, hogy a rendszer elavulttá válik, képletesen szólva „meghal”. Ekkor üzemén kívül helyezik, s ez jelenti az életciklus végét.

Informatikai rendszerek fejlesztésének több módja is lehetséges. A nagy rendszerek „semiből” való létrehozása mellett az utóbbi években egyre népszerűbb lett az újrafelhasználható komponens alapú fejlesztés, a kész alkalmazási programcsomagok adaptálása és használata, valamint kisebb projekteknél az önálló végfelhasználói fejlesztés. A 2.1. fejezetben részletesen megvizsgáljuk és összehasonlítjuk a rendszerfejlesztés különféle formáit.

Informatikai rendszerek értelmes fejlesztése, mint azt a fentiekben már elemeztük, elképzelhetetlen az üzleti folyamatok megértése nélkül, amelyek támogatására tervezik és fejlesztik a rendszereket. Ez a tevékenység a rendszerelemzés, vagy másképpen a követelmények meghatározása (*requirement engineering*) fázisában történik meg. Minden létező vagy tervezett üzleti folyamat és ezzel együtt a rendszerek sokféleképpen elemezhetők. Ezen megközelítések némelyike az alkalmazott technológiára, mások a szervezetre koncentrálnak, amelyben az üzleti folyamatok lejátszódnak. Létezik egy további megközelítés, amely az üzleti folyamatok során elvégzett üzleti tevékenységekre figyel, ezek alapján ír le, illetve elemez rendszereket. Ez a módszer az elmúlt években igen elterjedt.

## 15.1 A rendszerfejlesztés folyamatmodelljei

A rendszerfejlesztés folyamata bizonyos tevékenységeket, erőforrásokat, termékeket, technikákat és eszközöket foglal magában. Az elmúlt évtizedekben többféleképpen közelítettek e folyamathoz. Ezen megközelítések közül az alábbi öt használatos ma is:

- hagyományos vízésesmodell, s annak variánsai,
- prototípus készítése, evolúciós fejlesztés,
- újrafelhasználáson alapuló fejlesztés,
- kész alkalmazási programcsomagok használata,
- végfelhasználói fejlesztés.

Ebben a fejezetben ezeket a technikákat ismertetjük és hasonlítjuk össze, megemlítve előnyeiket és hátrányaikat. Fontos hangsúlyozni, hogy a rendszerfejlesztés bármelyik változatát választjuk is, valamennyi technika a már említett tevékenységekből áll:

- projektindítás,
- követelményelemzés,
- tervezés,
- megvalósítás, kivitelezés, integráció
- validálás,
- telepítés, üzemeltetés, rendszerevolúció.

Meg kell itt jegyeznünk, hogy a fázisok számát, illetve azok elnevezését illetően sokféle konvenció van forgalomban. Az életciklus tagolását, az alkotó fázisok számát illetően az informatikusok véleménye eltérő. Valójában nincs nagy jelentősége annak, hogy 5, 6 vagy 7 tevékenységet különböztetünk-e meg. Ezt a kérdést tehát rugalmasan kell kezelni. Időnként több fázist össze fogunk vonni, illetve egy-egy fázist további részfázisokra fogunk bontani.

A projektindítás szakaszában elsőként magát a problémát kell definiálni. Ez általában az alábbi két kérdés feltevését és megválaszolását jelenti:

- Miért van szükségünk egy új informatikai fejlesztésre, egy új projektre?
- Mit szeretnénk megvalósítani, elkészíteni, megoldani az új informatikai rendszerrel?

Ebben a fázisban kell a projekt elindításáról dönteni egy körültekintő megvalósíthatósági tanulmány alapján, költségbecslést és projekttervet készíteni, összeállítani a fejlesztői csapatot stb.

Az elemzés fázisa a rendszer tanulmányozását és modellezését jelenti. Az esetek tekintélyes százalékában egy létező (részben vagy teljesen automatizált, vagy akár teljesen manuális) rendszer problémáit elemzik a lehető legnagyobb részletességgel. Más esetekben új kihívásoknak igyekeznek innovatív módon megfelelni. Valamennyi esetben pontosan meg kell határozni a rendszerrel szemben támasztott követelményeket. Az elemzés célja tehát a probléma pontos és világos leírása.

A tervezés fázisában az előzőekben megfogalmazott követelményeket, problémákat kell hardver/szoftver „megoldássá” átalakítani. Az informatikában a terv a megoldás specifikálását, egzakt leírását jelenti, az implementálás pedig a rendszer fizikai megvalósítását, kivitelezését, tesztelését és átadását. A megvalósítás történhet a szükséges szoftver

- megírásával vagy
- megvásárlásával, esetleg
- lízingelésével.

Ebben a fázisban kell a hardvert beszerezni és installálni, amennyiben az még nem áll rendelkezésre. Szintén az implementálás része a felhasználók betanítása és a dokumentáció véglegesítése.

A validálás fázisa az ellenőrzést jelenti. Ez általában nem az implementálás befejezésekor kezdődik el, hanem lényegében végig kíséri a megelőző fázisokat. Is.

Az üzemeltetés fázisa akkor kezdődik, amikor a fejlesztői csapat átadja a rendszert a megrendelőnek, s azt folyamatos napi használatra üzembe állítják. Egyrészt a működés során felmerülő hibákat kell viszonylag gyorsan kijavítani, másrészt adaptálni a rendszert a megváltozott üzleti körülményekhez, továbbá tökéletesíteni, például a funkciók bővítésével.

A fentiek együttesen alkotják az informatikai rendszer életciklusát. Az elnevezés találó, hiszen egy rendszer „élete” valóban sok hasonlóságot mutat az emberi élettel. Mindkettő esetében beszélhetünk kezdetről és végről. Az informatikai rendszerek „halála”, azok üzemén kívül helyezése és lecserélése újabb rendszerekre.

A fejlesztés módszereit nem lehet rangsorolni, az eltérő helyzetek különböző megközelítéseket tesznek indokolttá. Ha létezne egy olyan módszer, amelyik minden tekintetben jobb a versenytársainál, szükségtelenné tenné a többiekkel való foglalkozást. Mivel ilyen nem létezik, ezért tisztában kell lennünk azzal, hogy milyen kritériumok alapján válasszunk az alternatívák közül.

### **15.1.1 A vízesés modell**

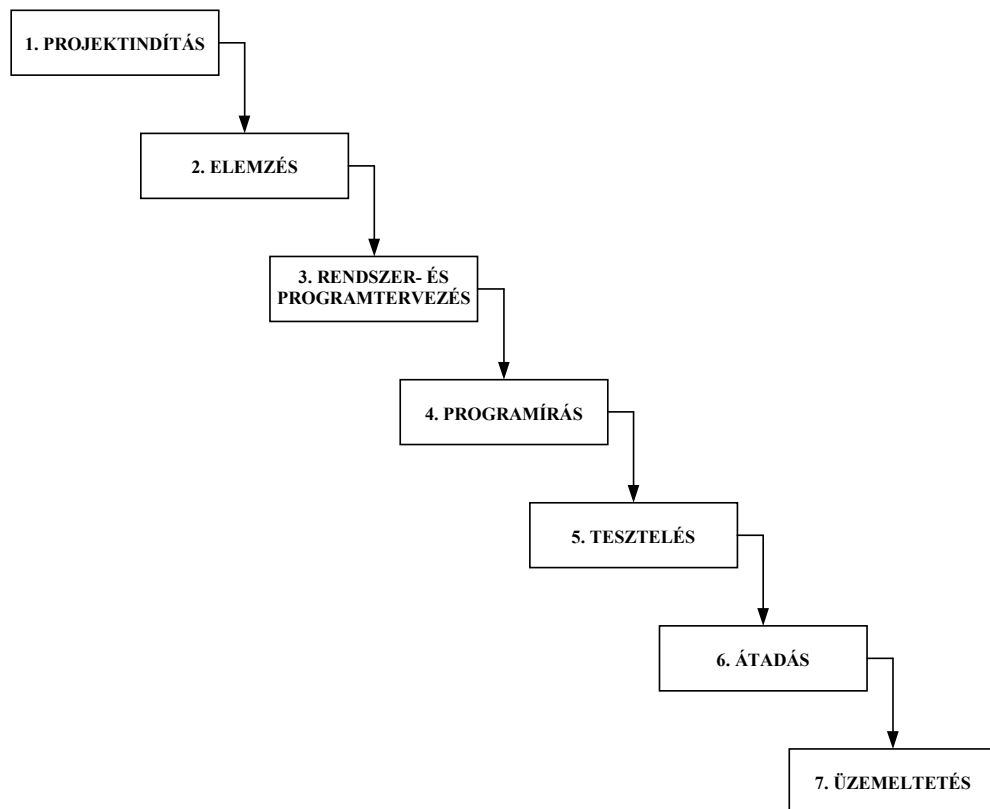
A hagyományos életciklus modell a legrégebbi rendszerfejlesztési technika. Az első modell még az ötvenes évek közepén született meg, bár elnevezését – **vízesés modell** – csak 1970-ben kapta. A név magyarázata, hogy az egyes fejlesztési szakaszok úgy követik egymást, mint egy többlépcsős vízesés.

A vízesés modell bizonyos változataiban az egyes fázisokat tovább bontják. A tesztelés például általában négy lépésben történik:

- modulteszt,
- integrációs teszt,
- rendszerteszt,
- elfogadási teszt;

az átadás pedig három, időben elkülönülő tevékenységből áll:

- installáció,
- dokumentáció,
- betanítás és tréning.



**2.1. ábra A rendszerfejlesztés hagyományos vízsesés modellje**

A vízsesés modell egy szigorúan dokumentumvezérelt folyamat. Ez azt jelenti, hogy egyik fázisból a soron következő fázisba csak a fázis teljes befejezése és a befejezést igazoló dokumentum elkészülte után lehet átmenni. E modellben a lépések sorrendje előre rögzített. Az egész folyamat a legapróbb részletekig bezárólag szabályozott. Ez a tény, bár lényegesen megnyújtja a fejlesztés időtartamát, egyúttal általában garancia is a minőségi munkára. A tradicionális vízsesés modellt (továbbfejlesztéseivel együtt) elsősorban nagy méretű és hosszú élettartamú rendszerek fejlesztésére használják. A szigorú előírások garantálják a jó minőségű, bár lassú fejlesztést.

A klasszikus vízsesés modellt gyakran hasonlítják egy olyan bevásárláshoz, ahol a vevőnek már az üzletbe lépve pontosan meg kell mondania, mit óhajt vásárolni, anélkül, hogy időt

hagynának neki az alapos körbenzésre, az árak összehasonlítására. Ebben a helyzetben nincs mód, pontosabban nagyon csekély lehetőség van csak arra, hogy menet közben módosítsunk eredeti elképzeléseinken.

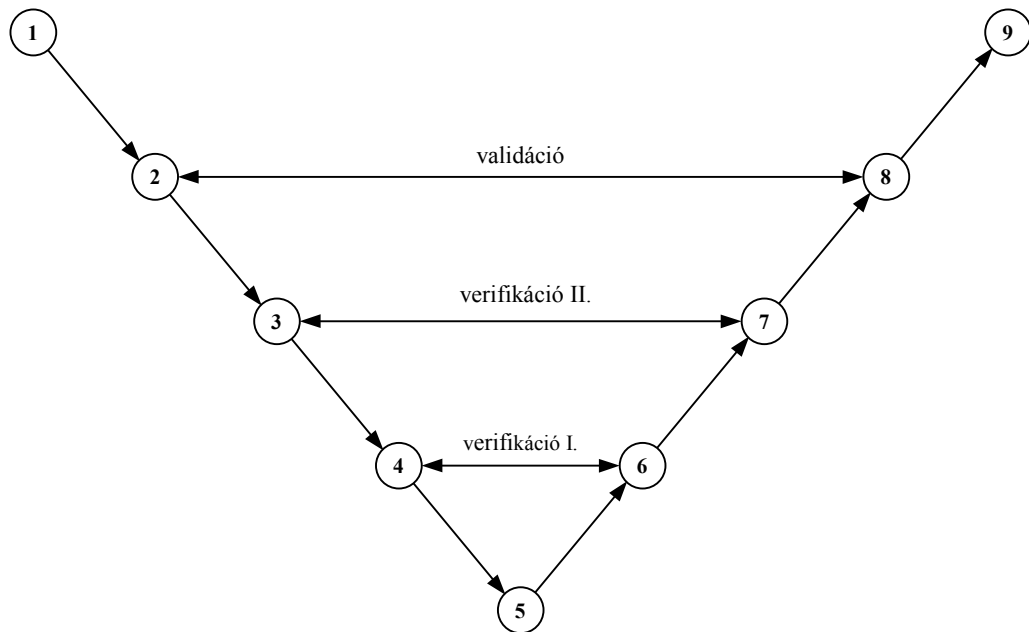
### **15.1.2 A V modell**

Az utóbbi években a német nemzetvédelmi minisztérium által a kilencvenes évek legelején kidolgozott folyamatmodell, az ún. V modell bizonyult népszerűnek a rendszerfejlesztők körében. A fejlesztés egyes fázisai egy V betűt formálnak, amelynek csúcsában a programírás tevékenysége áll, tőle balra az elemzés és a tervezés, tőle jobbra pedig a tesztelés és a karbantartás lépése található. A V alakzat magyarázata az, hogy ebben a folyamatmodellben a tesztelés kerül központi helyre. A modulteszt és az integrációs teszt verifikálja a részletes programtervet, azaz igazolja a terv helyességét, a rendszerteszt pedig verifikálja magát a rendszertervet. A megrendelő által elvégzett elfogadási teszt validálja, azaz érvényesíti, igazolja a rendszerelemzés helyességét. Ez utóbbi azt jelenti, hogy a megrendelő ellenőrzi, vajon minden, a rendszerrel szemben megfogalmazott követelmény megvalósult-e a rendszerfejlesztés során.

A V modellben 9 fejlesztési fázist különböztetünk meg, amelyek közül három a tesztelésre vonatkozik.

1. projektindítás,
2. elemzés, követelmények meghatározása,
3. rendszerterv elkészítése,
4. részletes programterv elkészítése,
5. programírás,
6. modul- és integrációs tesztelés,
7. rendszertesztelés,
8. elfogadási tesztelés,
9. átadás és üzemeltetés, karbantartás.





2.2. ábra A rendszerfejlesztés V modellje

*A fejlesztés fázisai egy V betűt formálnak, amelynek csúcsában a programírás tevékenysége áll.*

Mint azt korábban tisztáztuk, a validáció annyit jelent, hogy meggyőződünk arról, vajon a helyes, azaz a megfelelő rendszert építjük-e, a verifikáció pedig azt jelenti, hogy meggyőződünk arról, helyesen építjük-e a rendszert. Az elfogadási tesztelés során győződik meg a megrendelő, hogy a számára elkészített informatikai rendszer rendelkezik-e mindazon funkcionális és nem-funkcionális tulajdonságokkal és jellemzőkkel, amelyeket tőle elvárnak. Ha nem rendelkezik ezekkel, akkor vissza kell térni az elemzés fázisához, s azt újra el kell végezni.

A rendszer verifikációja kétféleképpen is megtehető. Az integrációs tesztelés során tapasztalt diszkrepancia (eltérések) esetén a részletes programtervhez, a rendszertesztesztelésnél tapasztalt problémák esetén pedig a rendszertervhez kell visszanyúlni.

### 15.1.3 Prototípusok alkalmazása

A vízesés modellen alapuló, tradicionális rendszerfejlesztés gyengeségeit viszonylag hamar felismerték. Ezek kiküszöbölésére számos elképzelés született, melyek közül a prototípusok alkalmazása az egyik legismertebb és leggyakrabban használt technika. A prototípus szó mintapéldányt jelent, és a rendszerfejlesztésben többféleképpen is használják a prototípusokat:

- az evolúciós rendszerfejlesztés eszközeként,
- a rendszerrel szemben támasztott követelmények megfogalmazásakor, specifikálásakor felmerülő problémák megoldására,

- valamely nagyobb befektetést igénylő döntés támogatására a lehetséges megoldásokkal való kísérletezéssel.

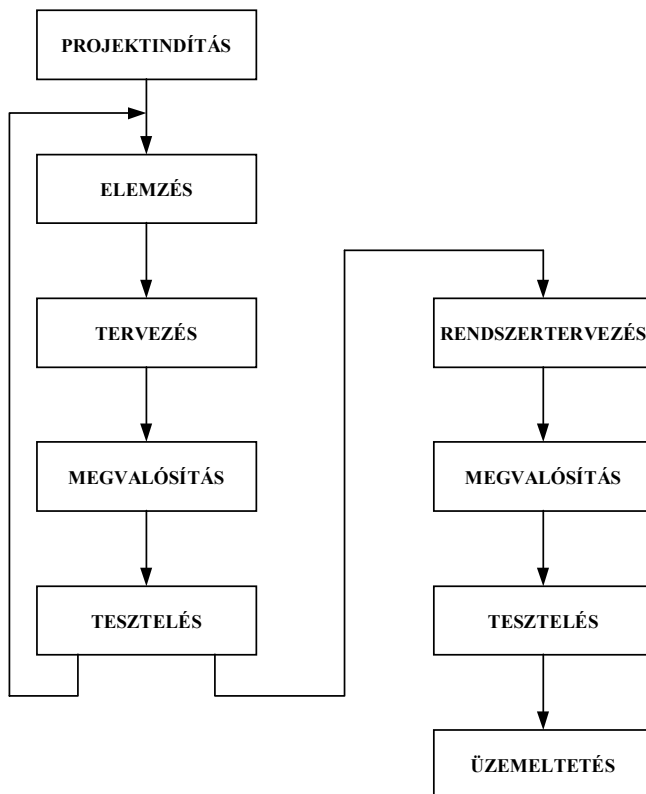
Az informatikai rendszerek megrendelői a projekt indításakor általában nehezen tudják valamennyi elvárásukat pontosan megfogalmazni a kifejlesztendő rendszerrel kapcsolatban, ami egyáltalán nem meglepő. Egy új rendszer iránti igény motivációja az esetek nagy részében az, hogy a megrendelő elégedetlen a jelenlegi helyzettel. Ennek egyik oka lehet az, hogy a jelenlegi rendszer egyáltalán nem képes kezelni azokat a problémákat, amelyek felmerülnek, vagy nem megfelelően, nem hatékonyan kezeli azokat. A meglevő rendszer ismerete azonban nem biztos, hogy elégséges az újjal szembeni elvárások teljes körének meghatározásához. További eshetőség az, ha a megrendelő nincs teljesen tisztában azzal, hogy az új rendszer révén egyáltalán milyen előnyökhöz juthat hozzá. Ilyen esetekben is hasznos egy működő, funkcionalitásában hasonló, ám sebességében, robosztusságában, összetettségében, komplexitásában a kívánt rendszertől esetleg lényegesen elmaradó mintapéldány megalkotása, amely bizonyos kérdések tisztázására alkalmas a rendszerfejlesztő csoporton belül, általában a megrendelő és a rendszerelemző között. Más területekkel (autógyártás, úrhajózás stb.) összehasonlítva a szoftverprototípusok elkészítése általában nem túl költséges és időigényes. Bizonyos esetekben akár több tucat prototípus is viszonylag könnyen kifejleszthető. Ezen prototípusok fejlesztésének nagyjából ugyanazok a folyamatai, mint a teljes rendszernek, azaz

- elemzés,
- tervezés,
- megvalósítás,
- tesztelés.

Ezek a lépések ismétlődnek, iterálódnak mindaddig, amíg a rendszerfejlesztő csapaton belül meg nem születik a konszenzus a követelményeket illetően. Ezután kerülhet sor a „nagy” rendszer fejlesztésének következő fázisára, a rendszertervezésre. Az elkészített prototípusokat a rendszerfejlesztés további fázisaiban nem használják. (Innen az elnevezés: eldobandó prototípus, bár ezek gyakran újra felhasználhatók más projektekben.)

Ez az eljárás tulajdonképpen annak ellenőrzése, hogy valóban a „jó” rendszert fejlesztik-e. Az „eldobandó” prototípusok készítésének a célja tehát nem több, mint a követelmények validációja, azaz érvényességük igazolása.

Bár a követelmények meghatározása nehezkesebb, mint a tradicionális rendszerfejlesztés során, a tapasztalatok azt mutatják, hogy a projekt összköltsége általában kisebb.



2.3. ábra A rendszerfejlesztés folyamatmodellje prototípusok alkalmazásával

Létezik a prototípusok használatának egy másik módja is, melyre két dolog jellemző:

- lényegében nincs elemző fázis,
- a kezdeti prototípus alakul át, „fejlődik” az iterációk, az evolúció során a végtermékké.

Kissé leegyszerűsítve az állítható, hogy az evolúciós prototípusok használatakor a megrendelő először egy olyan, távolról sem teljes mintarendszert kap, amely a követelmények halmazának egyfajta korlátozott megértésén alapul. Ez a kezdeti rendszer azután fokozatosan módosul, mind több követelménynek téve eleget. Ez a technika különösen akkor hasznos, ha a rendszer egy nem, vagy rosszul strukturált probléma megoldására szolgál. Ilyenkor általában a fejlesztők nem értik teljes mélységében az automatizálni kívánt folyamatot, mert például az rendkívül bonyolult. (A mesterséges intelligencia területéről sok ilyen rendszert lehetne említeni.)

A követelmények hiánya miatt a rendszeres elemzés fázisa kiesik. A követelmények meghatározása, amely az egész rendszerelemzés lényege, fokozatosan, lépésről lépésre történik.

Természetesen az evolúciós prototípusok technikáját olyan rendszerek esetén célszerű csak alkalmazni, ahol egy-egy iteráció gyorsan végrehajtható, azaz a javasolt módosításokat gyorsan lehet megvalósítani, implementálni, azaz ahol a ciklusidő rövid. Ennek a gyorsaságnak azonban vannak hátrányai is:

- A folytonos változtatások miatt a rendszer szerkezete általában kevésbé átlátható és következetes. (Az egyetlen „termék” a programkód.)
- A specifikáció hiánya a karbantartást igen megnehezítheti és megdrágítja. (Ezért az így fejlesztett rendszerek élettartama viszonylag rövid.)
- Fontos nem-funkcionális rendszerjellemzőket, mint például a teljesítményt, biztonságot, megbízhatóságot feláldozzák a gyorsaság miatt, ami nehézkessé teszi ezen nem-funkcionális követelmények utólagos teljesítését.
- Nem készül komoly rendszerdokumentáció, hiszen annak időigényessége lehetetlenné tenné a változtatások gyors realizálását.

Az evolúciós prototípusok használatának egyik további problémája, hogy a verifikáció lépése értelmezhetetlen. A verifikáció annyit jelent, hogy meggyőződünk arról, helyesen építjük-e a rendszert. Ez a gyakorlatban annyit tesz, hogy ellenőrizzük, a rendszer megfelel-e a megadott specifikációknak. Itt azonban nincsenek specifiációk, tehát formálisan nincs verifikáció sem. Ehelyett egy meglehetősen szubjektív értékelést szokás elvégezni, amely nem könnyen mérhető.

A fenti problémák és hátrányok azonban nem jelentik azt, hogy az evolúciós prototípusok használata bizonyos körülmények között ne lenne ajánlható. Például szakértői rendszereket vagy felsővezetői informatikai rendszereket általában ezzel a technikával fejlesztnek. Látható azonban, hogy igazán nagy méretű és hosszú élettartamú rendszerek fejlesztésére ez a technika nem alkalmas.

Összefoglalva azt mondhatjuk, hogy az eldobandó prototípusok használatának célja az ismeretlen, nem világos követelmények megértése. Azon követelményekre, amelyek egyértelműek, amelyeket nem kell tisztázni, nem készül prototípus. Az evolúciós prototípusok viszont először a jól megértett követelményeket építik be a modellbe, s azután lépésről lépésre, a megrendelő és a fejlesztő team szoros együttműködésében dolgozzák fel a hiányzó, vagy kevésbé világos, „fuzzy” igényeket.

#### **15.1.4 A spirál modell**

Az 1988-ban bevezetett Boehm-féle spirális folyamatmodell ötvözi a klasszikus vízésés modell és a prototípusok alkalmazásának technikáját, s ezenkívül beépíti a kockázatelemzést is a modellezésbe.

Négy fő tevékenységet különböztet meg:

1. tervezés (célok meghatározása, kényszerek és kockázatok azonosítása, alternatív stratégiák tervezése),
2. kockázatelemzés és csökkentés,
3. termékfejlesztés (prototípusok készítése) és V&V (validáció és verifikáció),

#### 4. értékelés.

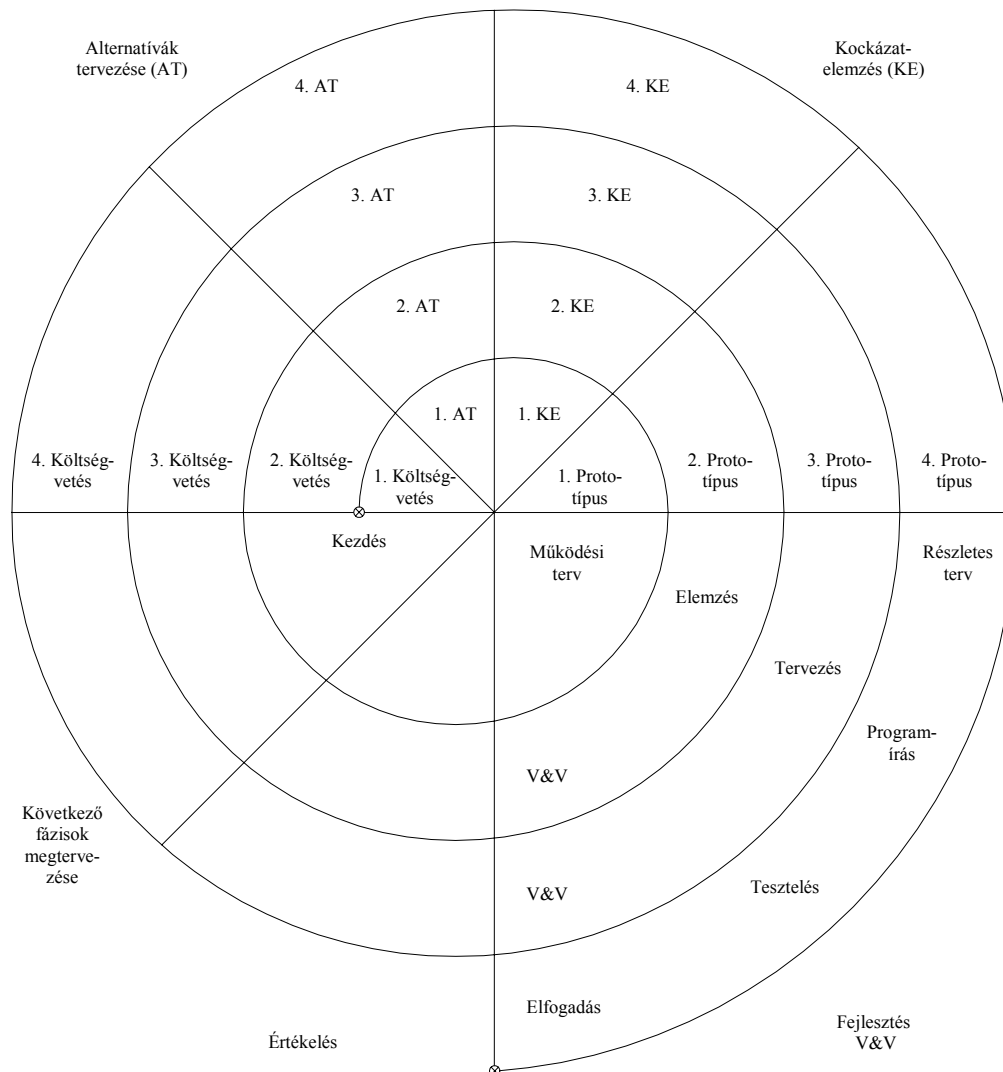
A spirálmodellben minden egyes iteráció során az alábbiakat kell megadni:

1. célok,
2. korlátok (pénz, idő stb.),
3. alternatívák,
4. kockázatok, veszélyek,
5. kockázatsökkentő stratégiák,
6. eredmények,
7. tervek,
8. döntés a folytatásról.

Cél lehet például a szoftverminőség radikális javítása, korlátozás pedig, hogy e célt nagyobb tőkebefektetés és komolyabb szervezeti változtatások nélkül szeretnék elérni. A folytatásról meghozott döntés lehet az, hogy további 10 hónapig folytatják a finanszírozást.

Az első körben meghatározzák a követelményeket (pontosabban azok egy minél bővebb részhalmazát), és előzetes tervet készítenek a fejlesztésre, ide értve a költségek, külső kényszerek (a rendelkezésre álló pénz és idő), valamint a tervezési, környezeti alternatívák meghatározását. Értékelik a kockázatot és a prototípus alternatívákat. A követelmények rendszere fokozatosan válik teljessé és konzisztenssé. Az első iteráció terméke a megvalósíthatósági tanulmány, a másodiké követelmények teljes körének meghatározása, a harmadiké a terv és a negyediké a tesztelés.

Minden iterációs lépés során kockázatelemzéssel mérlegelik a lehetséges alternatívákat a követelmények és a kényszerek függvényében. A prototípusok verifikálják a megvalósíthatóságot, mielőtt egy alternatíva kiválasztásra kerülne. Amikor a veszélyeket, kockázatokat azonosították, akkor a projektvezetőnek el kell döntenie, hogyan minimalizálja a kockázatot. Ez pedig úgy történik, hogy prototípusok készülnek, és ezeken tesztek végeznek.



2.4. ábra A rendszerfejlesztés spirálmodellje

A harmadik rendszerfejlesztési technika az újrafelhasználáson alapul. A tapasztalatok szerint a különböző informatikai rendszerek fejlesztésekor lényegében nagyon hasonló szakmai kérdések, problémák merülnek fel. Ennek az a nyilvánvaló oka, hogy az üzleti életben szinte minden hasonló profilú cég, vállalat, szervezet lényegében ugyanazokkal a kihívásokkal szembesül, ugyanazok vagy nagyon hasonlóak az üzleti célok és stratégiák, a környezeti feltételek, külső körülmények, jogi és számviteli szabályok stb. Évtizedeken keresztül több ezer párhuzamos fejlesztés történt, több ezer vagy több tízezer informatikus dolgozott lényegében azonos problémák megoldásán. Az általuk megtalált megoldások valószínűleg nagyon hasonlítottak egymásra, bár erről viszonylag keveset tudunk, mivel az egyes rendszerfejlesztő projektek meglehetősen titkolózás közepette zajlottak és zajlanak, a részletmegoldások ritkán váltak publikussá, s a résztvevők nemigen kommunikáltak egymással. Ezt a helyzetet tehát a szellemi erőforrások erőteljes pazarlása jellemezte.

Viszonylag későn (a nyolcvanas évek közepén, de igazából csak az objektum-alapú technológia térnyerésével) merült csak fel annak az igénye, hogy az egyszer már jól megírt,

alaposan tesztelt és sikeresen felhasznált szoftverrészeket, komponenseket későbbi hasonló projektekben újra hasznosítsák, s ebből a célból egy jól katalogizált speciális komponenskönyvtárat hozzanak létre. A tudományos számításokban már a hatvanas évek eleje óta használnak ún. szoftverkönyvtárakat (NAG, IMSL), amelyek a leggyakrabban alkalmazott numerikus matematikai eljárások valamilyen programnyelvben (FORTRAN, Pascal, C++) megírt és újrafelhasználásra alkalmassá tett forráskódjait tartalmazzák. A kilencvenes évekre azonban ez általános gyakorlattá kezdett válni az üzleti életben is, a szoftver újrahasználat (*software reusability*) pedig az egyik leginkább kutatott informatikai területté lett.

A szoftver újrafelhasználása nem feltétlenül jelenti azt, hogy az egyes részeket változtatás nélkül alkalmazzák az elkövetkező projektekben, hanem inkább arról van szó, hogy megszűnt a „semmiből újat teremtés” gyakorlata. Ehelyett korábbi rendszerek komponenseit vizsgálják meg, vajon azok adaptálhatók-e, vagy ami még szerencsésebb, változtatás nélkül újrafelhasználhatók-e a fejlesztés során. Fontos megjegyezni, hogy ezt akkor is érdemes és célszerű megnézni, ha az új informatikai projekt egésze lényegesen különbözőnek is tűnik a korábbiakhoz képest. (Az angol nyelvű informatikai irodalomban a módosítás nélküli újrafelhasználásra szokás a *black-box reuse*, az adaptációt magában foglaló változatra pedig a *white-box reuse* kifejezések használata.)

Az adaptációs újrafelhasználás esetében mindig érdemes mérlegelni, hogy a már létező komponens szerkezetének és működésének megértése, valamint a szükséges változtatások átvezetése nem igényel-e több erőfeszítést és hosszabb időt, mint egy vadonatúj programmodul megalkotása. Felismerve ezt a problémát, a könyvtárakban általában olyan komponenseket helyeznek el, amelyek a paraméterek megválasztásának függvényében sokoldalúan alkalmazhatók. A feltétlenül szükségesnél több paraméter használata nagy mértékben javíthatja az újrafelhasználás esélyeit.

A szoftver-újrafelhasználás – s ezt a gyakorlat igazolta – jelentősen csökkenti a fejlesztés költségét, és szignifikánsan növeli a termelékenységet. (Ez például KLOC/PM egységben mérhető). A termelékenység növekedése nemcsak annak a következménye, hogy kevesebb időt kell programírással tölteni, hanem annak is, hogy a tesztelésre és a dokumentációk készítésére fordítandó idő is jelentősen csökkenthető. Az újrafelhasználás, s ezt lényeges hangsúlyozni, nemcsak a forráskódú programsorokra vonatkozik, hanem a követelmények meghatározására, a folyamatokra, a tervezésre, a tesztesetekre és tesztadatokra, valamint a dokumentációra is.

A szoftver-újrafelhasználás a rendszerfejlesztés folyamatában a leggyakrabban úgy történik, hogy a már említett komponenskönyvtárakban fellelt kész programrészeket szemléletesen szólva építőkövekként kezelik, s azok köré, azokra épül rá a teljes szoftver. Ez a megközelítés tehát alulról felfelé haladva építkezik, alapul véve azt, ami már rendelkezésre áll. Az újrafelhasználás legnehezebb problémájának az tűnik, hogyan lehet a szoftverkönyvtárakban

az adott szituációra legalkalmasabb komponenst megtalálni a nagy számú lehetőség közül. Tekintettel arra, hogy a könyvtárba egyre több tétel kerül, a kikeresés gyorsasága elsőrendűen fontos. Az egyik lehetséges megoldás ahhoz hasonlít, amit a hagyományos könyvtárakban alkalmaznak az ETO-szám (Egyetemes Tizedes Osztályozás) bevezetésével.

Ha rendelkezésre áll egy megfelelő hierarchikusan szervezett katalógus, valamint egy alkalmas tárgyszóindex és egy keresztreferencia-táblázat, akkor nem szükséges minden könyvet végigolvasni ahhoz, hogy rábukkanjunk a keresett tételre.

Egy további lehetőség az, hogy a komponenskönyvtárban minden egyes tételről készül egy kötött leírás, amelyből kiderül, hogy az adott programrészlet

- milyen alkalmazási területen,
- melyik programozási nyelvben,
- melyik operációs rendszerhez,
- milyen objektumról,
- milyen funkcióhoz

készült. Ehhez társulhat egy rövid szöveges leírás a komponensről szerzett tapasztalatokról, a megbízhatóságról, a fejlesztőkről stb.

Egy hatékony visszakereső, lekérdező rendszerrel automatizálni lehet a keresést a felhasználó igényeinek megfelelően. A keresést például az nehezíti, hogy egy funkciót többféleképpen is le lehet írni különböző szinonimák (ugyanarra a fogalomra vonatkozó eltérő elnevezések) használatával. A rendszernek képesnek kell tehát lennie a szinonimák és a tartalmilag hasonló, de nem teljesen ugyanazt leíró megnevezések kezelésére is.

Egy további elvárás a visszakereső rendszerrel szemben az, hogy amennyiben túl sokan keresnek eredménytelenül egy szoftverkomponenst, akkor jelezze az igényt a komponens kifejlesztésére.

A gyakorlati tapasztalatok azt bizonyítják, hogy az új informatikai projekteknél az újrafelhasznált komponensek alkalmazásának mértéke meghaladhatja a 60-80%-ot, ami jelentős költségcsökkenést és minőségjavulást eredményez. Számos fejlesztőcég külön ösztönzőrendszert dolgozott ki a szoftver-újrafelhasználás támogatására.

Az eddig leírtak a szoftverkomponensek újrafelhasználására vonatkoztak. Az érem másik oldala az ilyen komponensek megírása, létrehozása. Egy, az elmúlt években végzett összehasonlító vizsgálat megállapítása szerint ennek relatív költsége átlagosan a kétszerese-ötszöröse egy ugyanolyan méretű, de nem újrafelhasználásra szánt programrészlet elkészítésének. Ugyanezen vizsgálat az újrafelhasználás relatív költségét átlagosan 10-20%-ra



teszi. Ezek az adatok, még ha nagy szórást is mutatnak, jól jelzik, hogy milyen előnyökkel és milyen extraköltségekkel jár a szoftverkomponensek újrafelhasználása.

Az alábbi felsorolás összefoglalóan tartalmazza az újrafelhasználás potenciális előnyeit és problémáit:

#### Előnyök

- Kisebb kockázat — kisebb a bizonytalansági tényező a fejlesztési költségekben
- Fokozott megbízhatóság — működő rendszerekben már kipróbált és alaposan tesztelt komponensek
- Gyorsabb fejlesztés és csökkenő költségek — szignifikáns javulás érhető el
- Szabványoknak való megfelelés — szabványok implementálása szabványos újrafelhasználható komponensekben
- Szakemberek hatékonyabb alkalmazása — komponensek és nem a szakemberek újrafelhasználása kívánatos

#### Problémák

- Növekvő karbantartási költségek — a rendszer változtatásával az újrafelhasznált elemek inkompatibilissé válhatnak
- Az eszköztámogatás hiánya — a CASE-eszközkészletek nem támogatják az újrafelhasználáson alapuló szoftverfejlesztést
- Komponenskönyvtárak karbantartása — magas költséggel jár a katalogizálásra és a visszakeresésre szolgáló technikák használata
- Komponensek megtalálása és adaptálása — komoly jártasság kell hozzá

A negyedik rendszerfejlesztési technika a kész alkalmazási programcsomagok használata. E módszer alap gondolata, hogy bár minden vállalat különbözik a többiektől, valójában az őket kiszolgáló informatikai rendszerek nagyon hasonlítanak egymásra. Mivel vállalatok ezreiről, esetleg (ha nem csak egyetlen országban tekintjük őket) százezreiről van szó, így meglehetősen természetesen adódik a felvetés, hogy miért ne lehetne olyan programcsomagokat készíteni, amelyek minimális módosításokkal alkalmasak nagy számú vállalat azonos típusú problémáinak megoldására. Általában ezek a vállalatok hasonló méretűek, és a piac azonos szegmensében találhatók, bár léteznek olyan programcsomagok is, amelyek sokkal széles körben használhatóak. A bevezetőben említett fázisok, a

- projektindítás, projektmeghatározás,
- elemzés,
- tervezés,
- implementálás, integráció,
- üzemeltetés, karbantartás, támogatás

ugyanúgy felismerhetők a kész alkalmazási programcsomagokkal történő fejlesztés esetén is. Téves az a vélekedés, hogy egy kész alkalmazási programcsomag megvásárlása feleslegessé teszi az életciklus modellt, bár kétségtelenül jelentősen lerövidítheti a fejlesztés időtartamát.

Az első két fázis (projektmeghatározás és elemzés pontosan úgy történik, mint a tradicionális rendszerfejlesztés esetén. Az elemzés fázisa és a funkcionális specifikációk elkészítése nélkülözhetetlen, hiszen világosan és egyértelműen meg kell fogalmazni, hogy mit várunk el a rendszertől.

A további fázisok jelentősen leegyszerűsödnek, bár például ekkor is szükség van annak eldöntésére, hogy a megvásárolt szoftver pontosan milyen funkcióira tartunk igényt, ekkor is át kell adni és tesztelni kell a szoftvert (legfeljebb a modulteszt lépése maradhat el), s a dokumentációt is „testre kell szabni”.

Az installálás lépése lényegében abból áll, hogy el kell dönteni, a programcsomag által kínált opciókból melyekre tartunk igényt. Ez történhet például úgy, hogy megadjuk, milyen paraméterekkel kívánjuk a rendszert használni.

Kész alkalmazási programcsomag megvásárlása esetén is gondoskodni kell a felhasználók betanításáról, bár ez általában egyszerűbb feladat, mert a szoftvergyártó cég, pontosan a szoftvereladások nagy száma miatt, jelentős tapasztalattal rendelkezik a tréning terén.

Az utolsó fázis, az üzemeltetés és karbantartás vonatkozásában szintén nincs különbség a hagyományos és az alkalmazási programcsomaggal történő rendszerfejlesztés között, bár a tanácsadást illetően ugyanaz a helyzet, mint a betanítással.

Mint a prototípusok esetében is, a kész alkalmazási programcsomaggal történő rendszerfejlesztést is a tradicionális módszerrel szokás előnyök és hátrányok vonatkozásában összehasonlítani.

A kész alkalmazási programcsomaggal történő fejlesztés egyebek mellett az alábbi előnyökkel jár:

- A fejlesztés időtartama lényegesen lerövidülhet, s ennek következtében a projekt haszna hamarabb jelentkezik.
- A fejlesztőprojekt kockázata jelentősen lecsökken, valószínűtlen a határidő túllépése, hiszen a szoftver azonnal hozzáférhető.
- Mind a dokumentáció, mind a tanácsadás színvonala általában magasabb, mint a saját fejlesztésé.
- A szoftvergyártó cég szakmai felkészültsége általában jobb, mint az *ad hoc* fejlesztői csapaté.

Ugyanakkor a kész alkalmazási programcsomaggal történő fejlesztésnek hátrányai is vannak:

- A szoftver nem teljesen illeszkedik a vállalat létező üzleti folyamataihoz, ezért gyakran kompromisszumokat kell kötni, s módosítani kell a folyamatokon.
- Körülményes a programsomagok adaptálása új helyzetekhez, mivel például a forráskód alig hozzáférhető.
- Gyakran kell új hardvert, szoftvert vásárolni, mivel a meglévők nem illeszkednek a vásárolt rendszerhez, vagy nem kompatibilisek azzal.
- Bár a kész alkalmazási szoftverek többfunkciósak, gyakran az egyes modulok kidolgozottsága igen eltérő színvonalú, s néhány esetben jelentősen gyengébb, mint a saját fejlesztésűek.
- Bár a szoftverek eladási ára viszonylag nem túl magas, legalább is a saját fejlesztés költségével összehasonlítva, számos rejtett költség jelentkezik az installálásnál. Egy viszonylag friss felmérés szerint az installáció teljes költsége akár a tízszerese is lehet a vételárnak, különösen akkor, ha a rendszernek nagyon sok külső illesztése van.

A rendszerfejlesztés ötödik technikája a végfelhasználói fejlesztés. Az informatikusok szerint ennek a módszernek az a lényege, hogy nem halat adunk a rászorulóknak, hanem megtanítjuk halászni, természetesen úgy, hogy ellátjuk korszerű felszereléssel.

A nyolcvanas évek elejétől a legkülönbözőbb eszközök:

- táblázatkezelők,
- adatbáziskezelő programok,
- negyedik generációs programozási nyelvek,
- adatelemző/statisztikai szoftverek

segítségével sok vállalatnál indultak végfelhasználói fejlesztői (EUD, *end-user development*) projektek. Ha az ilyen típusú fejlesztések életciklusát megvizsgáljuk, akkor megállapítható, hogy az elemzés fázisa jelentősen leegyszerűsödik, mivel nincs szükség formális specifikációra, hiszen nem lehet kommunikációs probléma a fejlesztő és a felhasználó között. Hasonlóan az implementálás is egyszerűbb, hiszen nem kell a felhasználót megtanítani a rendszer használatára. A karbantartás és tanácsadás fázisa viszont nehezebb lehet, hiszen ha professzionális programozóra van szükség, akkor annak előbb meg kell értenie a nem feltétlenül hatékonyan írt programot.

Természetesen a végfelhasználói rendszerfejlesztés szinte kivétel nélkül kisebb méretű feladatokra szorítkozik, s így nem alternatívája a másik három fejlesztési technikának. Ez a megállapítás azonban semmiképp nem csökkent a végfelhasználói rendszerfejlesztés előnyeit. Egy kisvállalkozás teljes könyvelését automatizálni lehet például Excel makrók megírásával. Ehhez a felhasználónak mindössze a Visual Basic for Application (VBA) programnyelvet kell elsajátítania, s az Excel kínálta programozási környezetben a feladatok kényelmesen elvégezhetők.

## 16 Függelék

### 16.1 A C# kulcsszavai

Kulcsszó	Jelentés
<b>abstract</b>	Módosító, amely arra utal, hogy egy osztályt csak egy másik osztály őseként használhatjuk.
<b>as</b>	Típuskényszerítés.
<b>base</b>	Az ősosztály értékeit és típusait teszi elérhetővé.
<b>bool</b>	A logikai adattípus kulcsszava.
<b>break</b>	Ciklus, feltételes utasítás vagy esetszétválasztás elhagyása.
<b>byte</b>	A byte adattípus kulcsszava.
<b>case</b>	Esetszétválasztáson belül egy eset (feltétel).
<b>catch</b>	A try-catch kivételkezelés része. A catch blokk segítségével elkaphatjuk a kivételeket, és meghatározott hibakezelő műveleteket futtathatunk le.
<b>char</b>	A karakter adattípus kulcsszava.
<b>checked</b>	Ellenőrzött kódrészt azonosító kulcsszó.
<b>class</b>	Osztálydefiníció kulcsszava.
<b>const</b>	Adattaghoz vagy változóhoz kapcsolódó módosító, amely azt jelzi, hogy a kérdéses adat tartalma nem változtatható.
<b>continue</b>	Ugrás a következő ciklusfordulóra.
<b>decimal</b>	A decimal adattípus kulcsszava.
<b>default</b>	Címke a switch esetszétválasztó utasításban. Itt folytatódik a végrehajtás, ha egyik case feltétel sem teljesül.
<b>delegate</b>	Referencia típus, amely meghatározott formájú tagfüggvényeket tartalmazhat.
<b>do</b>	Ciklusszervező utasítás.
<b>double</b>	A double adattípus kulcsszava.
<b>else</b>	Az if feltételes utasítás máskülönben ágát jelzi.
<b>enum</b>	Adattípus, amely előre meghatározott állandó értékeket tartalmaz.
<b>event</b>	Események, eseménykezelők definiálására szolgáló kulcsszó.
<b>explicit</b>	A felhasználó által létrehozott adattípusok átalakítását jelző kulcsszó.
<b>extern</b>	Programunkon kívül levő függvény jelzése.
<b>false</b>	Logikai hamis érték.
<b>finally</b>	A try-catch utasítás része, a try blokk után hajtódik végre – függetlenül attól, hogy a végrehajtás során fellépett-e hiba vagy sem.
<b>fixed</b>	Kezeletlen kódban ezzel a kulcsszóval zárolhatjuk a memóiahivatkozásokat, hogy az automatikus szeméttgyűjtő ne szabadítsa fel azokat.

<b>float</b>	A float lebegőpontos adattípus kulcsszava.
<b>for</b>	Ciklusszervező utasítás.
<b>foreach</b>	Ciklusszervező utasítás.
<b>goto</b>	A program egy adott címkével ellátott utasítására lehet ugrani vele.
<b>if</b>	A feltételes utasítás kulcsszava.
<b>implicit</b>	Kulcsszó, amely automatikusan végrehajtható típusátalakítást ír le.
<b>in</b>	A foreach kulcsszóval együtt használandó; Az in kulcsszó azonosítja a gyűjteményt vagy tömböt, amit a ciklus végigjár.
<b>int</b>	Az int adattípus kulcsszava.
<b>interface</b>	Olyan hivatkozást definiálunk vele, amely jelzi tagmetódusait, de nem írja le azokat.
<b>internal</b>	Hozzáférés módosító. Lehetővé teszi, hogy az adott adattípust csak az azonos gyűjteményben lévő fájlokból érhesük el.
<b>is</b>	Objektum típusvizsgálatához használt kulcsszó.
<b>lock</b>	Kritikus kódrészek jelölése. Az ilyen kódrész egyszerre csak egy szál számára lesz elérhető.
<b>long</b>	A long adattípus kulcsszava.
<b>namespace</b>	Kulcsszó, amellyel típusainkat csoportosíthatjuk. Segít a névütközés elkerülésében.
<b>new</b>	Objektum létrehozása.
<b>null</b>	Érték, amely azt jelzi, hogy a hivatkozás nem mutat semmire.
<b>object</b>	A .NET keretrendszer System.Object osztályán alapuló típus. A C#-ban minden más adattípus ebből származik.
<b>operator</b>	Az osztályok és struktúrák műveleteinek létrehozására vagy túlterhelésére szolgáló kulcsszó.
<b>out</b>	Csak kimenő metódus-paraméter jelzése.
<b>override</b>	Virtuális metódus felváltást előidéző felüldefiniálása a leszármazott osztályban.
<b>params</b>	Változó taglétszámú paraméterlista(rész) jelzése.
<b>private</b>	Láthatóság módosító.
<b>protected</b>	Láthatóság módosító.
<b>public</b>	Láthatóság módosító.
<b>readonly</b>	Adattag módosító. Azt jelzi, hogy a kezdeti értékadás után a tag tartalma már nem változtatható.
<b>ref</b>	Cím szerinti paraméterátadás jelzője a formális paraméternél.
<b>return</b>	Kilépés a függvényből és a visszatérési érték megadása.
<b>sbyte</b>	Az sbyte adattípus kulcsszava.
<b>sealed</b>	Osztálymódosító, amely meggátolja, hogy az osztályból további osztályokat származtassanak.
<b>short</b>	A short adattípus kulcsszava.
<b>sizeof</b>	Adattípus méretének meghatározása.
<b>stackalloc</b>	Kulcsszó, amellyel memóriát foglalhatunk a veremben. A visszakapott mutató nem tartozik az automatikus szeméthyűjtés hatókörébe.
<b>static</b>	Láthatóság módosító.

<b>string</b>	A string adattípus kulcsszava.
<b>struct</b>	Adattípus, amely adatokat és tagfüggvényeket egyaránt tartalmazhat.
<b>switch</b>	Az esetszétválasztó utasítás kulcsszava.
<b>this</b>	Az aktuális struktúrát vagy objektum-példányt jelöli.
<b>throw</b>	Kivételt generáló utasítás.
<b>true</b>	Logikai igaz.
<b>try</b>	A kivételkezelő blokk védett utasításait tartalmazó blokkját jelző kulcsszó.
<b>typeof</b>	Objektum típusát adja vissza, .NET formátumban.
<b>uint</b>	Az uint adattípus kulcsszava.
<b>ulong</b>	Az ulong adattípus kulcsszava.
<b>unchecked</b>	Azt jelzi, hogy a hatáskörébe utalt egész típusú műveleteknél mellőzni kell a túlsordulás-vizsgálatot.
<b>unsafe</b>	Olyan kódrészletet jelöl, amelyez a kezelt (managed) környezetben nem biztonságos végrehajtani (pl. mutatók használatánál).
<b>ushort</b>	Az ushort adattípus kulcsszava.
<b>using</b>	Álnevek létrehozása névterekhez.
<b>virtual</b>	Virtuális metódus jelölése.
<b>void</b>	Metódusoknál jelzi, hogy nincs visszatérési értékük.
<b>while</b>	Ciklusszervező utasítás.