

Interfețe Windows în C#

Scurt istoric

C# (pronunțați *C sharp* - în traducere literală *C diez*) este un limbaj de programare orientat pe obiecte asemănător limbajului C++. El a fost dezvoltat de firma Microsoft, ca soluție standard pentru dezvoltarea aplicațiilor Windows. Creatorii lui sunt Anders Hejlsberg (autor al limbajelor *Turbo Pascal* și *Delphi*), Scot Wiltamuth și Peter Golde.

C# respectă standardul ECMA-334 (*European Computer Manufactures Association*).

Limbajul C# folosește *Microsoft .NET Framework*, o colecție de clase care poate fi descărcată prin Internet și care este întreținută și ameliorată permanent de Microsoft. Prima versiune datează din 2001, deci toată această soluție tehnologică este de dată recentă.

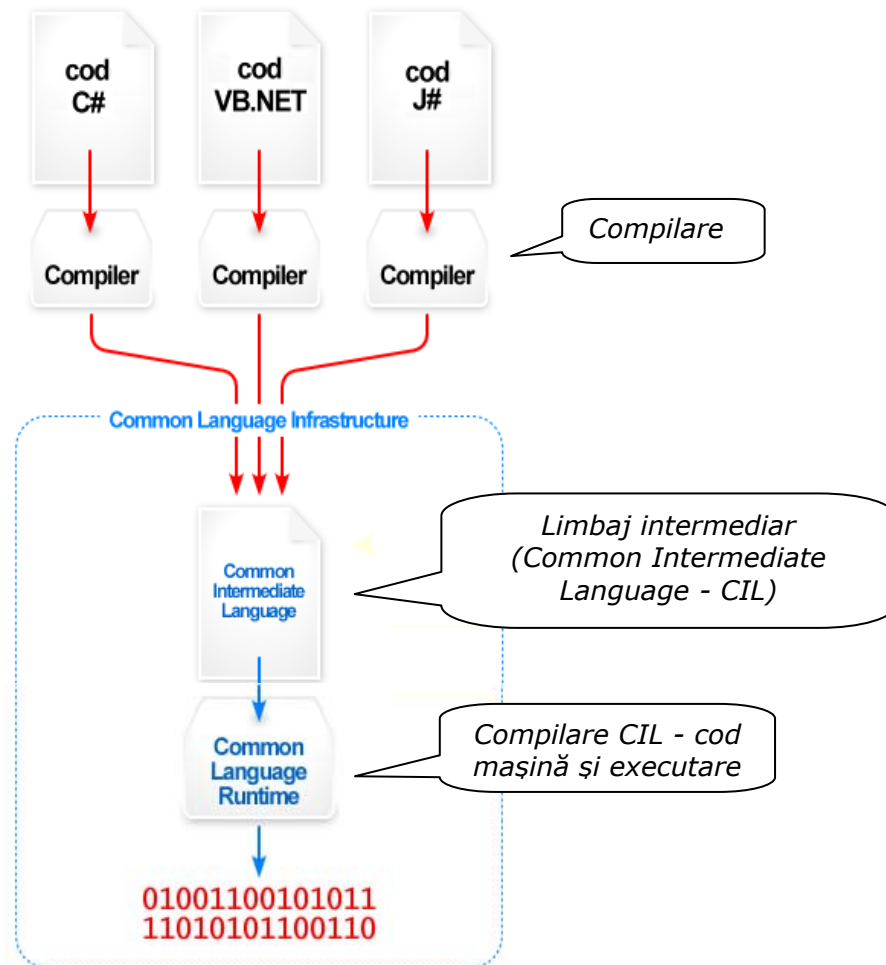
Clasele conținute în *.NET* sunt neutre față de limbaj (aplicațiile pot fi scrise în diverse limbaje) și ușurează munca de programare în realizarea interfețelor aplicațiilor, accesul la date, conectarea la servere de baze de date, accesarea resurselor din Internet, programarea aplicațiilor bazate pe comunicații prin rețele, etc..

Aplicațiile care folosesc *.NET* sunt executate într-un mediu software denumit *CLR (Common Language Runtime)*. Acesta poate fi asimilat unui procesor virtual, asemănător mașinii virtuale Java, care furnizează servicii de management a resurselor calculatorului, asigurare a stabilității și securității execuției. Deși Microsoft a conceput CLR ca soluție destinată propriilor sisteme de operare, la ora actuală există astfel de procesoare virtuale și pentru alte sisteme de operare. Astfel, în cadrul proiectului *Mono* - www.gomono.com, susținut de compania Novell, s-au dezvoltat soluții alternative pentru Linux, Solaris, Mac OS, BSD, HP-UX, și chiar Windows, asigurându-se astfel portabilitatea aplicațiilor scrise în limbaje bazate pe *.NET*.

Tratarea programelor scrise în C#

Un program scris în C# este supus unei prime compilări în urma căreia se obține un cod scris într-un limbaj intermediar (*CIL - Common Intermediate Language*). În

această formă aplicația este apoi trimisă procesorului virtual (*CLR*) care realizează traducerea în cod mașină și execută aplicația.



Realizarea unei aplicații elementare în C#

Dezvoltarea de aplicații în C# se realizează folosind un mediu de programare, firma Microsoft oferind mediul profesional *Visual C# .NET*. O soluție alternativă gratuită poate fi versiunea *Express Edition* a mediului profesional (<http://www.microsoft.com/express/vcsharp/>), utilizabilă pentru învățarea limbajului și pentru realizarea interfețelor aplicațiilor Windows. Microsoft oferă însă și varianta realizării aplicațiilor folosind pachetul gratuit *.NET Framework Software Development Kit (SDK)*. De altfel simpla descărcare a pachetului de bază *.NET Framework* pus la dispoziție de Microsoft adaugă în directorul *C:\Windows\Microsoft.NET\Framework\vxx*

(xx este versiunea descărcată) compilatorul `csc.exe` utilizabil în mod linie de comandă și care produce cod executabil.

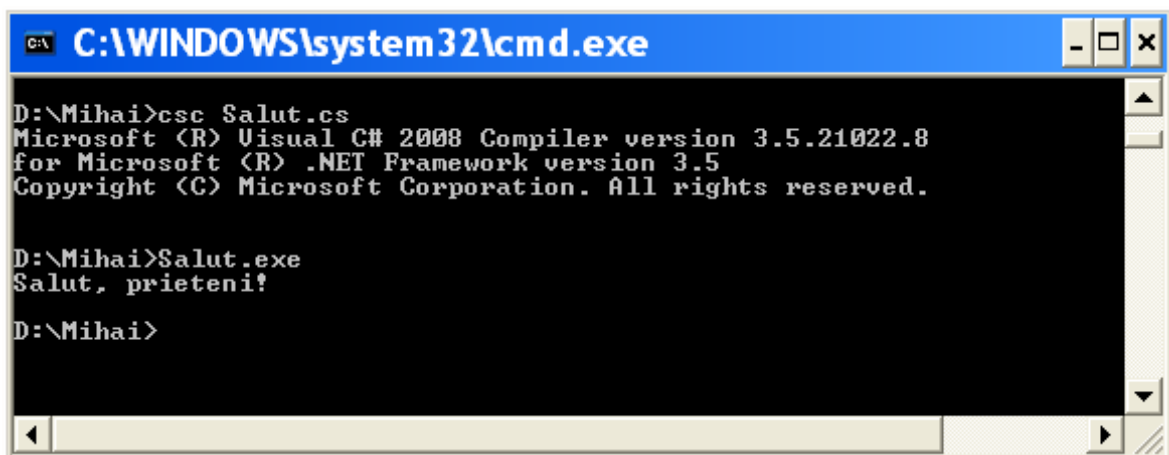
Exemplu de aplicație tip consolă:

```
class Salut
{
    public static void Main()
    {
        System.Console.WriteLine("Salut, prieteni!");
    }
}
```

Aplicația tip consolă va fi salvată într-un fișier denumit *Salut.cs* și apoi va fi compilată folosind `csc.exe`:

```
>csc Salut.cs
```

În urma compilării va rezulta fișierul *Salut.exe* care poate fi executat din linia de comandă:



```
C:\WINDOWS\system32\cmd.exe

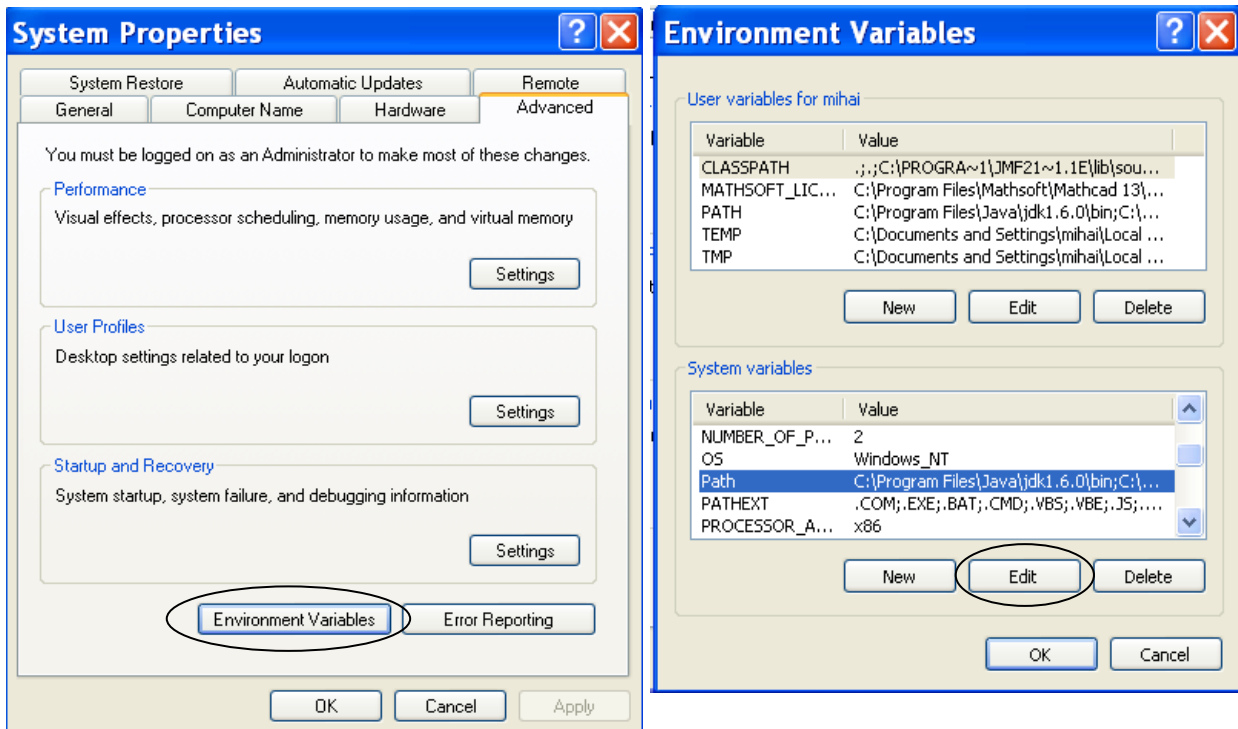
D:\Mihai>csc Salut.cs
Microsoft (R) Visual C# 2008 Compiler version 3.5.21022.8
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

D:\Mihai>Salut.exe
Salut, prieteni!

D:\Mihai>
```

Observație:

Pentru a folosi compilatorul `csc.exe` fără a preciza calea este necesară modificarea variabilei de mediu *Path*. Pentru aceasta se va accesa *Control Panel / System / Advanced*. În fereastra care se afișează se va selecta *Environment Variables* iar în fereastra *Environment Variables* va fi editată variabila *Path* adăugând căilor deja existente calea spre directorul care conține compilatorul `csc.exe`. Pentru *.NET* versiunea 3.5, cea mai recentă în momentul scrierii acestui curs, calea este: `C:\WINDOWS\Microsoft.NET\Framework\v3.5`.



Noțiuni de C#

Spații de nume

Spațiul de nume reprezintă un concept apărut în C++ odată cu extinderea programării obiectuale. Clasele care alcătuiesc o aplicație pot proveni din surse diferite și pot avea din întâmplare aceeași denumire. Dacă programatorii au creat clasele în spații de nume distincte, eventualele coincidențe de nume pot fi ușor rezolvate, denumirea fiecărei clase fiind precedată de denumirea spațiului de nume căruia îi aparține.

Pentru a evita tastarea spațiului de nume înaintea fiecărei denumiri de clasă în C# se folosește ca și în C++ clauza *using*.

.NET Framework definește peste 90 de spații de nume, majoritatea având denumiri începând cu *System*. În exemplele conținute în cursurile următoare vor fi frecvent folosite spațiile de nume *System*, *System.Windows.Form* și *System.Drawing*.

Exemplul de aplicație prezentat poate fi rescris folosind clauza *using*, clasa *Console* aparținând spațiului de nume *System*.

```
using System;
class Salut
{
    public static void Main()
    {
        Console.WriteLine("Salut, prieteni!");
    }
}
```

Comentarii

Programele scrise în C# pot conține explicații sub forma unor *comentarii*. Acestea vor asigura în timp înțelegerea diferitelor metode și rolul funcțional al variabilelor folosite.

Ca și în C++, în C# pot fi folosite două tipuri de comentarii:

- comentarii "în linie", introduse prin `//`. Un astfel de comentariu se încheie la sfârșitul liniei curente;
- comentarii pe unul sau mai multe rânduri consecutive. Zona care conține comentariul este delimitată prin `/*` (comentariu) `*/`.

De regulă se folosesc comentariile în linie, cel de-al doilea tip fiind folosit frecvent în timpul depanării programelor mari și complexe, pentru a suprima din program anumite zone.

Tipuri de date

În mare C# folosește *tipurile de date simple* existente și în C++. Fiecare dintre tipurile simple este însă dublat de o clasă din spațiul de nume *System*. Rolul acelei clase este puțin evident în acest moment. În cele ce urmează va fi inclus un exemplu în care vor fi referite proprietăți ale tipurilor simple în același mod în care sunt referite proprietăți ale claselor. De exemplu `int.MaxValue`, furnizează valoarea întreagă cea mai mare care poate fi păstrată într-o variabilă de tip `int`. Tot dublarea tipurilor simple prin clase va face posibilă folosirea unor variabile aparținând unor tipuri simple în situații în care sunt acceptate doar obiecte.

a. Tipuri de date întregi:

Număr de biți	Tip simplu	Clasă	Semn
8	sbyte	System.SByte	cu semn
8	byte	System.Byte	fără semn
16	short	System.Int16	cu semn
16	ushort	System.UInt16	fără semn
32	int	System.Int32	cu semn
32	uint	System.UInt32	fără semn
64	long	System.Int64	cu semn
64	ulong	System.UInt64	fără semn

Exemple de declarații:

int a = 12, b, contor; // Se putea scrie și System.Int32 a = 12, b, contor;

b. Tipuri de date reprezentate în virgulă mobilă

Număr de biți	Tip simplu	Clasă
32	float	System.Single
64	double	System.Double

c. Tipul *decimal*

Limbajul C# admite pe lângă tipurile de date numerice menționate un tip nou, *decimal*. Pentru *decimal* lungimea de reprezentare este de 128 biți iar clasa corespunzătoare din spațiul de nume *System* este *System.Decimal*.

d. Tipul *bool*

C# admite declararea de variabile logice, ale căror valori posibile sunt *true* și *false*. Pentru declararea unei astfel de variabile se folosește *bool*, clasa corespunzătoare din spațiul de nume *System* fiind *System.Boolean*.

Operatorii relaționali (==, !=, <, <=, >, >=) generează o valoare de tip *bool*.

e. Tipul *char*

Tipul *char* permite declararea unei variabile care va conține un caracter în format *Unicode* (pe 16 poziții binare). Clasa corespunzătoare din spațiul de nume *System* este *System.Char*.

f. tipul *string*

O simplificare semnificativă pe care o aduce C# în raport cu C++ este introducerea tipului *string* pentru declararea variabilelor care conțin șiruri de caractere. Clasa corespunzătoare din spațiul de nume *System* este *System.String*.

Exemple:

```
string a = "X = "; // Se putea și System.String a = "X = ";
double p = 129.2;
int nr = a.Length; // .Length furnizează lungimea șirului a.
string b = a + p + "mm";
```

În C# operatorul '+' indică fie operația de adunare fie cea de concatenare (unire) a două șiruri de caractere. Pentru toate tipurile de date menționate (mai exact pentru toate tipurile derivate din clasa *Object*, rădăcina ierarhiei de clase destinate păstrării datelor) este definită metoda *ToString()* care returnează o reprezentare sub formă de șir de caractere a valorii variabilei pentru care este apelată. Expresia care definește valoarea variabilei *b* începând cu *a*, un *string*, variabila reală *p* este automat convertită în șir de caractere chiar dacă nu este apelată în mod explicit metoda *ToString()*. O soluție mai clară ar fi fost însă:

```
string b = a + p.ToString() + "mm";
```

g. Variabile de tip tablou

O variabilă de tip tablou se declară adăugând o pereche de paranteze drepte ([]) după cuvântul rezervat care definește tipul valorilor conținute. Exemplu:

```
int [ ] tab;
```

După declarare valoarea variabilei *tab* este *null*. Variabila va primi o valoare diferită de *null* în momentul rezervării unei zone de memorie pentru păstrarea elementelor tabloului, folosind operatorul *new*:

```
tab = new int[12];
```

De obicei declararea se combină cu alocarea de memorie, scriindu-se:

```
int [ ] tab = new int[12];
```

sau, dacă se realizează concomitent și atribuirea de valori:

```
int tab [ ] = new int[4] {1, 4, -2, 5}; // sau chiar mai simplu: int tab [ ] = {1, 4, -2, 5};
```

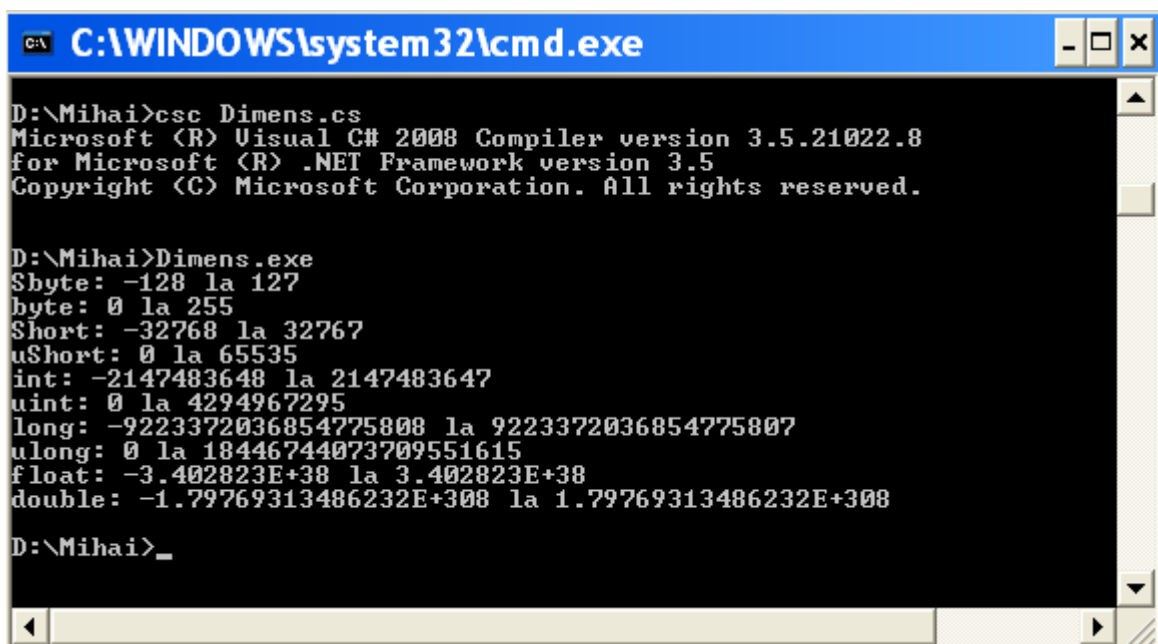
Ulterior se poate folosi variabila *tab* pentru un alt șir de valori *int*, astfel:

```
tab = new int[5];
```

În acest caz, deși nu s-a realizat o eliberare explicită a memoriei (în C# nu există operatorul *delete*), blocul de memorie alocat inițial va fi în mod automat eliberat și repus la dispoziția aplicației. Procesul de identificare a blocurilor de memorie devenite disponibile face obiectul activității unei componente a *CLR* denumită *garbage collector* (colector de reziduuri).

Exemplu: Să se scrie o aplicație tip consolă care afișează valorile numerice maxime și minime care pot fi memorate folosind diferite tipuri de variabile.

```
using System;
class Dimens
{
    public static void Main()
    {
        Console.WriteLine("Sbyte: {0} la {1}", sbyte.MinValue, sbyte.MaxValue);
        Console.WriteLine("byte: {0} la {1}", byte.MinValue, byte.MaxValue);
        Console.WriteLine("Short: {0} la {1}", short.MinValue, short.MaxValue);
        Console.WriteLine("ushort: {0} la {1}", ushort.MinValue, ushort.MaxValue);
        Console.WriteLine("int: {0} la {1}", int.MinValue, int.MaxValue);
        Console.WriteLine("uint: {0} la {1}", uint.MinValue, uint.MaxValue);
        Console.WriteLine("long: {0} la {1}", long.MinValue, long.MaxValue);
        Console.WriteLine("ulong: {0} la {1}", ulong.MinValue, ulong.MaxValue);
        Console.WriteLine("float: {0} la {1}", float.MinValue, float.MaxValue);
        Console.WriteLine("double: {0} la {1}", double.MinValue, double.MaxValue);
    }
}
```



```
C:\WINDOWS\system32\cmd.exe

D:\Mihai>csc Dimens.cs
Microsoft (R) Visual C# 2008 Compiler version 3.5.21022.8
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

D:\Mihai>Dimens.exe
Sbyte: -128 la 127
byte: 0 la 255
Short: -32768 la 32767
ushort: 0 la 65535
int: -2147483648 la 2147483647
uint: 0 la 4294967295
long: -9223372036854775808 la 9223372036854775807
ulong: 0 la 18446744073709551615
float: -3.402823E+38 la 3.402823E+38
double: -1.79769313486232E+308 la 1.79769313486232E+308

D:\Mihai>_
```


Operatorii limbajului C#

În definirea unui limbaj, stabilirea operatorilor care pot fi folosiți la scrierea expresiilor și a regulilor lor de utilizare reprezintă o etapă importantă. Limbajele de nivel înalt folosesc un mare număr de operatori și chiar permit extinderea setului de operatori pentru a putea scrie expresii în care intervin variabile aparținând tipurilor structurate (obiecte).

Operatorii matematici din C#

+ - * / % (modulo)

$x \% y$ furnizează restul împărțirii întregi a lui x la y .

Exemplu de utilizare a operatorului '%':

```
if((an % 4 == 0 && an % 100 != 0) || an % 400 == 0)
    System.Console.WriteLine(an + " este bisect.");
```

Operatorul % se aplică doar tipurilor întregi.

Operatorul "/" poate provoca trunchiere în cazul operanzilor întregi.

Evaluarea expresiilor aritmetice se efectuează de la stânga la dreapta respectând ordinea de evaluare normală.

Operatorii de comparare și logici

Operatorii de comparare (relaționali) din C# sunt : > >= < <= == !=

Operatorii de comparare au o prioritate inferioară celor aritmetici :

$i < lim + 1$ se evaluează ca și $i < (lim + 1)$

O expresiile în care intervine o succesiune de *operatori logici* && (și) sau o succesiune de operatori || (sau) sunt evaluate de la stânga la dreapta și evaluarea expresiei încetează când se știe deja valoarea de adevăr a rezultatului.

&& e mai prioritar ca și || iar ambele sunt mai puțin prioritare ca și operatorii de comparație.

Operatorul ! (negație)

if (!corect) . . . identic cu *if* (corect == false) . . .

În C# termenii pentru care sunt folosiți operatorii `&&` și `||` sunt obligatoriu de tip *bool*. În C# nu operează regula din C conform căreia o valoare numerică diferită de 0 are valoarea de adevăr *true* și o valoare 0 are valoarea de adevăr *false*.

Conversiile de tip

Limbajul C# este mult mai riguros în ceea ce privește corespondența tipurilor termenilor dintr-o expresie. Frecvent utilizarea într-o expresie a unor operanzi de tipuri diferite este semnalată ca eroare. Eliminarea erorilor se va face fie folosind operatorii de transtipaj (ca în C) fie apelând metode din clasa *Convert*.

Operatorii de conversie de tip (transtipaj sau *cast*)

(nume-tip) expresie

expresie este convertită în tipul specificat :

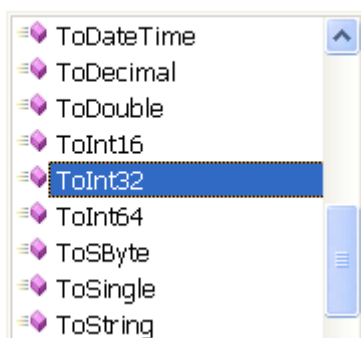
```
x = Math.Sqrt((double)n)
alfa = (float)a / j ;
```

Conversia explicită folosind metodele clasei *Convert* permite trecerea unei variabile sau a unei expresii dintr-un tip în altul. Exemplu:

```
int z = Convert.ToInt32(312.14);
```

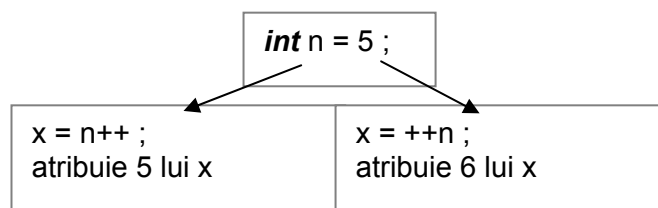
Practic dacă este necesară o conversie se va folosi facilitatea mediului de programare de a afișa permanent posibilitățile de continuare a scrierii și se va selecta varianta dorită.

```
int z = Convert.
```



Operatorii de incrementare și de decrementare ++ și --

În funcție de poziția operatorului față de variabila incrementată, se realizează **pre** și **post** incrementare (decrementare)



Operatorii și expresiile de atribuire

Expresia :

$i = i + 3;$

în care *variabila din stânga egalului se repetă în dreapta acestuia* se mai poate scrie astfel :

$i += 3;$

Instrucțiunile limbajului C#

Instrucțiunea de decidere - *if*

Ca și în C, ansamblul de instrucțiuni cuprinse între acolade formează o instrucțiune compusă sau un bloc de instrucțiuni. Instrucțiunile din interiorul unui bloc de instrucțiuni sunt executate în ordinea în care sunt scrise.

Variante de scriere a instrucțiunii if	
if (<i>expLogica</i>) <i>instrucțiune</i> ;	if (<i>expLogica</i>) { <i>mai multe instrucțiuni</i> }
if (<i>expLogica</i>) <i>instrucțiune</i> ; else <i>instrucțiune</i> ;	if (<i>expLogica</i>) { <i>mai multe instrucțiuni</i> } else { <i>mai multe instrucțiuni</i> }
if (<i>expLogica</i>) <i>instrucțiune</i> ; else { <i>mai multe instrucțiuni</i> }	if (<i>expLogica</i>) { <i>mai multe instrucțiuni</i> } else <i>instrucțiune</i> ;

Indentarea instrucțiunilor sau a blocurilor de instrucțiuni (scrierea decalată) din **if** nu este obligatorie dar are mare importanță în înțelegerea și depanarea programului.

Instrucțiunea **while**

Instrucțiunea **while** permite realizarea unei structuri repetitive (ciclu) *condiționate anterior*. Corpul ciclului poate fi executat o dată, de mai multe ori sau de loc.

Exemplu fundamental:

```
int contor = 1;
while ( contor <= 3 )
{
    Console.WriteLine("Contor = "+contor);
    contor = contor + 1; // sau mai bine contor++
}

Console.WriteLine("La revedere!");
```

Sintaxa instrucțiunii **while** este:

```
while ( condiție )
    instrucțiune
```

Condiția este o expresie logică (evaluată la o valoare de tip *bool*).

Sintaxa instrucțiunii while	
while (<i>condiție</i>) <i>instrucțiune</i> ;	while (<i>condiție</i>) { <i>una sau m. m.instrucțiuni</i> }

Instrucțiunea **do**

Pentru programarea unui ciclu poate fi folosită întotdeauna instrucțiunea **while**. Deoarece **while** începe prin executarea unui test, în cazul în care variabila testată nu

poate primi valori decât în interiorul ciclului, programatorul trebuie să-i dea la început o valoare convenabilă pentru a determina intrarea în ciclu. Exacutarea instrucțiunilor din corpul ciclului va corecta automat valoarea inițială.

Pentru astfel de cazuri există însă și o cale mai simplă, respectiv folosirea instrucțiunii *do*.

Exemplu:

```
g = 9.81;
t = 0;
do
{
    d = (g * t * t) / 2;
    Console.WriteLine( t + " " + d);
    t = t + 1;
}
while (d <= 500);
```

Instrucțiunea for

Instrucțiunea *for* este preferată ori de câte ori trebuie realizat un ciclu care folosește un contor.

Dacă înaintea unui ciclu trebuie realizate mai multe inițializări, acestea pot scrise în *for* una după alta, despărțite prin virgule:

```
class Program
{
    public static void Main(String[] args)
    {
        {
            int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            int i, sum;
            for (i = 0, sum = 0; i < a.Length; i++)
                sum = sum + a[i];
            Console.WriteLine("Suma elementelor sirului a este: " + sum);
        }
    }
}
```

Oricare dintre cele trei părți ale instrucțiunii *for* pot lipsi. Lipsa condiției de reluare este echivalentă cu a scrie o condiție care este întotdeauna adevărată. Ieșirea dintr-un astfel de ciclu se poate face folosind instrucțiunea *break*.

Exemplu:

```
for ( ; ; )
```

```
{
  ..
  break;
  ..
}
```

Executarea instrucțiunii *break* provoacă o ieșire imediată din ciclul *for*. Ea poate fi folosită și pentru a ieși din ciclurile *while* sau *do*.

Instrucțiunea *foreach*

Instrucțiunea *foreach* a fost preluată în C# din Visual Basic. Ea permite parcurgerea sistematică a elementelor unui șir sau a unei liste, realizabilă în C# folosind una dintre clasele din spațiul de nume *System.Collection*.

Exemple:

```
int sir = new int[12];
foreach (int i in sir)
    i = 10;

...
foreach (int i in sir)
    Console.Write("{0} ", i);
```

În exemplu, primul *foreach* pune 10 în toate elementele din șirul de valori șir iar al doilea afișează valorile sdin șir pe o linie.

Instrucțiunea *switch*

Instrucțiunile de ciclare prezentate, sunt folosite pentru realizarea blocurilor de instrucțiuni a căror execuție începe cu prima linie și este controlată de către programator. Instrucțiunea ***switch*** este folosită tot pentru a construi un bloc de instrucțiuni dar punctul de unde începe execuția depinde de valoarea unei expresii având valori întregi.

Sintaxa instrucțiunii este:

switch (expresie)

```
{
case constanta_1 : instrucțiuni
case constanta_2 : instrucțiuni
```

```

. . .
default : instrucțiuni;
}

```

Fiecare dintre punctele de unde poate începe execuția blocului este etichetat printr-o constantă având valoare întreagă. Dacă expresia de testat corespunde uneia dintre constante, execuția blocului începe din punctul indicat de aceasta.

Cazul **default** este facultativ.

Exemplu :

```

switch (t)
{
    case 's':
        rez = Math.Sin(Math.PI * x / 180.0);
        Console.WriteLine("rezultatul este : " + rez);
        break;
    case 'c':
        rez = Math.Cos(Math.PI * x / 180.0);
        Console.WriteLine("rezultatul este : " + rez);
        break;
    case 't':
        rez = Math.Sin(Math.PI * x / 180.0) / Math.Cos(3.14159 * x / 180.0);
        Console.WriteLine("rezultatul este : " + rez);
        break;
    default:
        Console.WriteLine("Caracter incorect!");
        break;
}

```

Ca și în cazul ciclurilor, instrucțiunea **break** provoacă o ieșire imediată din blocul realizat folosind **switch**.

Observație: În C# *break* apare și în secvența introdusă prin clauza *default*.

Programare obiectuală în C#

Limbajul C# permite scrierea de aplicații folosind exclusiv programarea obiectuală. O aplicație scrisă în C# este alcătuită dintr-un ansamblu de clase, una dintre ele fiind clasa principală deoarece conține metoda *Main()*, punctul de intrare în aplicație.

Exemplu fundamental:

Se consideră o aplicație care afișează a câta zi din an este o zi dată prin definirea anului, lunii și a zilei din lună.

Pentru rezolvarea problemei s-a creat clasa *DCalend*:

```
class DCalend
{
    private int anul;
    private int luna;
    private int ziua;

    public DCalend(int aa, int ll, int zz)
    {
        anul = aa;
        luna = ll;
        ziua = zz;
    }

    public bool AnBisect()
    {
        return (anul % 4 == 0) && ((anul % 100 != 0) || (anul % 400 == 0));
    }

    public int ZiuaDinAn()
    {
        int[] luniz = new int[] { 0,31, 59, 90, 120, 151,
                                   181, 212, 243, 273, 304, 334};
        return luniz[luna - 1] + ziua + (luna > 2 && AnBisect() ? 1 : 0);
    }
}
```

Ca și în C++, în C# o clasă conține în principal un număr de *membri date* sau *câmpuri* (*anul*, *luna*, *ziua*) și un număr de metode (funcții) care operează asupra membrilor date. Metodele clasei *DCalend* sunt:

- DCalend(), o metodă care *poartă numele clasei* și inițializează membrii date ai acesteia. În programarea obiectuală o astfel de metodă se numește *constructor*;
- AnBisect(), care returnează o valoare *bool* care indică dacă *anul* este an bisect și
- ZiuaDinAn() care calculează a câta zi din an este ziua definită prin valorile curente din câmpurile clasei.

Spre deosebire de C++ în clasele C# metodele sunt scrise integral.

Folosind clasa *DCalend* s-a scris următoarea aplicație tip consolă:

```
using System;

namespace POB01
```



```

{
    class Program
    {
        public static void Main(String[] args)
        {
            DCalend data = new DCalend(2007, 8, 29);
            Console.WriteLine("Ziua din an = {0}", data.ZiuaDinAn());
        }
    }

    class DCalend
    {
        private int anul;
        private int luna;
        private int ziua;

        public DCalend(int aa, int ll, int zz)
        {
            anul = aa;
            luna = ll;
            ziua = zz;
        }

        public bool AnBisect()
        {
            return (anul % 4 == 0) && ((anul % 100 != 0) || (anul % 400 ==
0));
        }

        public int ZiuaDinAn()
        {
            int[] luniz = new int[] { 0, 31, 59, 90, 120, 151,
                181, 212, 243, 273, 304, 334 };
            return luniz[luna - 1] + ziua + (luna > 2 && AnBisect() ? 1 : 0);
        }
    }
}

```

Pe lângă clasa *DCalend* aplicația mai are clasa *Program* care conține doar metoda principală *Main()*. În metoda principală se declară și se inițializează (prin apelul constructorului) obiectul *data* și apoi se afișează pe ecran a câta zi din an este *data*. Pentru apelul metodelor clasei *DCalend* se scrie:

- *data.ZiuaDinAn()* dacă metoda este apelată dintr-o altă clasă, respectiv
- *AnBisect()* dacă metoda este apelată dintr-o altă metodă a aceleiași clase.

În programarea obiectuală se definește și o a treia variantă de apel a unei metode, *nume_clasă.metodă*. Un exemplu este apelul deja utilizat, *Math.Sin(...)*. *Math* este o clasă din spațiul de nume *System* care conține definițiile funcțiilor matematice implementate în C#. O funcție astfel apelată operează asupra argumentelor sale, pentru apelul ei nefiind necesară definirea în prealabil a unui obiect aparținând aceleiași

clase. Pentru a declara o astfel de funcție, în programarea obiectuală se folosește cuvântul rezervat *static*. În exemplul dat metoda *AnBisect()* putea fi declarată statică. Avantajul ar fi fost acela de a o putea folosi în orice aplicație, fără a fi necesară cunoașterea clasei *DCalend* și fără a defini un obiect din clasa *DCalend*.

Câmpurile pot fi și ele definite ca fiind statice. O astfel de soluție este indicată în cazul câmpurilor care au aceleași valori pentru toate instanțele clasei. În exemplul dat, șirul de valori întregi *luniz* este constant, deci poate fi declarat static.

Clasa *DCalend* ar fi atunci definită astfel:

```
class DCalend
{
    private int anul;
    private int luna;
    private int ziua;
    static int[] luniz = new int[] { 0,31, 59, 90, 120, 151,
                                     181, 212, 243, 273, 304, 334};

    public DCalend(int aa, int ll, int zz)
    {
        anul = aa;
        luna = ll;
        ziua = zz;
    }

    public static bool AnBisect(int anul)
    {
        return (anul % 4 == 0) && ((anul % 100 != 0) || (anul % 400 == 0));
    }

    public int ZiuaDinAn()
    {
        return luniz[luna - 1] + ziua + (luna > 2 && AnBisect(anul) ? 1 : 0);
    }
}
```

Proprietăți

Ca și în orice alt limbaj care permite programarea obiectuală, în C# o clasă are membri date denumiți și câmpuri și membrii conținând cod denumiți metode. Clasele C# mai pot conține și alți membrii conținând cod denumiți *proprietăți*, extrem de importanți în *.NET Framework*.

Originea problemei rezolvate prin introducerea proprietăților este următoarea:

În POO membrii date sunt de regulă declarați privați (*private*). Accesul la un astfel de câmp dinafara clasei care îl conține este atunci rezolvat prin adăugarea unei perechi de metode:

- o metodă pentru modificarea valorii, denumită de obicei ***setnume_camp(tip val_noua)*** și
- o metodă pentru preluarea valorii, denumită de obicei ***getnume_camp()***.

O astfel de soluție este destul de greoaie și mărește considerabil numărul de metode ale claselor care trebuie scrise. De exemplu mediul de programare Netbeans destinat scrierii de aplicații în Java oferă ca facilitare dezvoltatorilor scrierea automată a perechii de funcții de acces *set - get*.

În C#, prin introducerea *proprietăților* se încearcă o simplificare accesului la câmpurile private.

Exemplu:

Se consideră câmpul *anul* din clasa *DCalend* prezentată deja. Se poate adăuga clasei *DCalend* secvența următoare de cod care definește proprietatea *Anul*:

```
public int Anul
{
    set
    {
        anul = value;
    }
    get
    {
        return anul;
    }
}
```

Se observă că proprietatea definită are același nume cu câmpul privat pe care îl accesează, cu diferența că începe cu o majusculă. De altfel programatorii în C# denumesc în mod sistematic membrii publici ai claselor cu denumiri care încep cu o majusculă. De asemenea se vede că definirea unei proprietăți înseamnă tratarea în două secvențe de cod a impunerii valorii câmpului (*set : câmp = value*) și a preluării acesteia (*get : return câmp*). Evident secvența de cod introdusă prin *set* poate testa valabilitatea valorii *value*.

Exemplu de utilizare a proprietății în funcția *Main()*:

```
public static void Main(String[] args)
{
```

```

    DCalend data = new DCalend(2007, 8, 29);
    Console.WriteLine("Ziua din an = {0}", data.ZiuaDinAn());
    data.Anul = 2006;
    Console.WriteLine("Ziua din an = {0}", data.ZiuaDinAn());
}

```

Parametrii metodelor

În C# parametrii metodelor sunt întodeauna transferați prin valoare. Ca și în C, în acest caz modificările operate în cadrul metodei asupra valorilor parametrilor formali ai acestora nu schimbă valorile variabilelor corespunzătoare din metoda apelantă.

Exemplu:

```

using System;

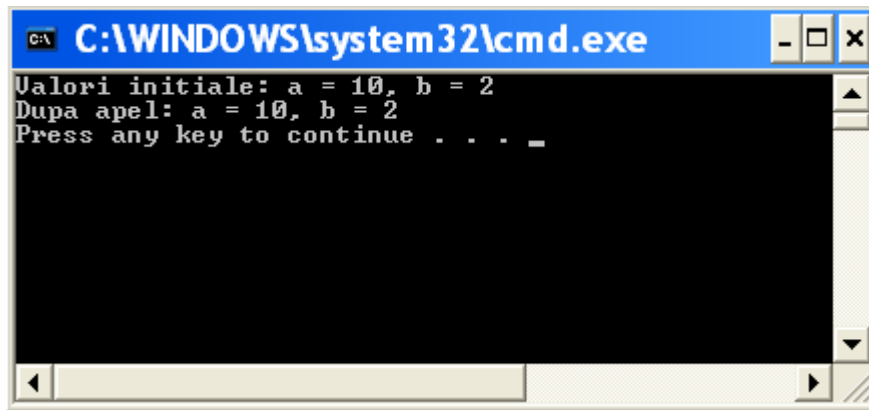
namespace Test
{
    class Invers
    {
        public static void Main(string[] args)
        {
            Invers z = new Invers();
            z.Calcule();
        }

        public void Calcule()
        {
            int a=10, b=2;
            Console.WriteLine("Valori initiale: a = {0}, b = {1}", a, b);
            Inv(a, b);
            Console.WriteLine("Dupa apel: a = {0}, b = {1}", a, b);
        }

        public void Inv(int x, int y)
        {
            int aux;
            aux = x;
            x = y;
            y = aux;
        }
    }
}

```

Metoda *Inv()* din exemplul dat este eronată deoarece transmiterea prin valoare a parametrilor face ca inversarea realizată să nu afecteze variabilele din metoda apelantă (*a* și *b*).



Pentru a impune ca transmiterea parametrilor să nu se realizeze prin valoare ci prin referință, în C# se folosește clauza *ref*. Aplicația din exemplul precedent ar fi trebuit scrisă atunci astfel:

```
using System;

namespace Test
{
    class Invers
    {
        public static void Main(string[] args)
        {
            Invers z = new Invers();
            z.Calcule();
        }

        public void Calcule()
        {
            int a=10, b=2;
            Console.WriteLine("Valori initiale: a = {0}, b = {1}", a, b);
            Inv(ref a, ref b);
            Console.WriteLine("Dupa apel: a = {0}, b = {1}", a, b);
        }

        public void Inv(ref int x, ref int y)
        {
            int aux;
            aux = x;
            x = y;
            y = aux;
        }
    }
}
```

Moștenirea

Toate limbajele destinate programării obiectuale permit realizarea unor clase *derivate* din clase existente. Clasa derivată *moștenește* membrii date, metodele și proprietățile clasei de bază. În cadrul ei se pot însă adăuga *noi membri*, date sau metode, prin care se realizează practic un proces de specializare a noii clase.

La definirea unei clase derivate programatorul poate de asemenea înlocui unele dintre metodele clasei de bază cu metode noi, mai bine adaptate. Redefinirea metodelor clasei de bază se mai numește și *suprascrisere*, noile metode având evident aceeași declarație cu cele din clasa de bază.

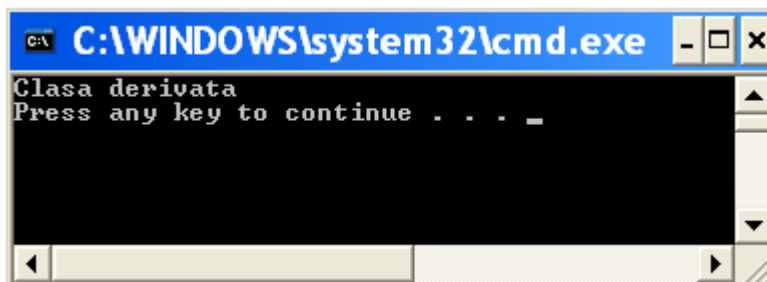
Exemplu:

```
using System;

namespace TestOvr
{
    class Program
    {
        static void Main(string[] args)
        {
            Derivata obj = new Derivata();
            obj.Autentifica_te();
        }
    }

    public class ClasaDeBaza
    {
        public void Autentifica_te()
        {
            Console.WriteLine("Clasa de baza");
        }
    }

    public class Derivata : ClasaDeBaza
    {
        public void Autentifica_te()
        {
            Console.WriteLine("Clasa derivata");
        }
    }
}
```



Noua clasă, *Derivata* a fost declarată folosind sintaxa `public class Derivata : ClasaDeBaza` astfel indicându-se relația de derivare. În C# o clasă poate avea o singură clasă părinte (nu se admite *moștenirea multiplă*).

O situație aparte apare în cazul metodelor declarate *virtuale*. Acestea fac posibil un comportament normal al obiectelor declarate ca aparținând tipului de bază și instanțiate cu obiecte aparținând unui tip derivat. În C# la suprascrierea unei astfel de metode se precizează printr-un cuvânt rezervat modul în care noua metodă va opera. Modurile posibile sunt *override* și *new*. Diferența dintre cele două declarații va fi prezentată prin două exemple.

a. Modul *override*

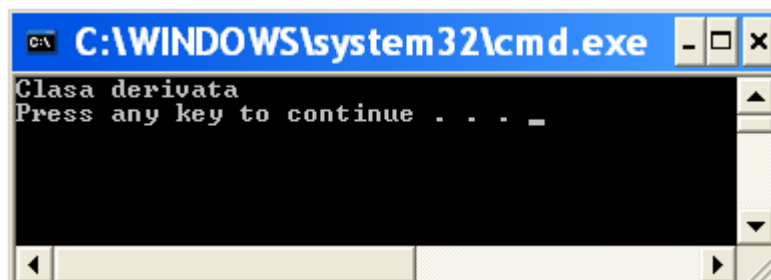
```
using System;

namespace TestOvr
{
    class Program
    {
        static void Main(string[] args)
        {
            ClasaDeBaza obj = new Derivata();
            obj.Autentifica_te();
        }
    }

    public class ClasaDeBaza
    {
        public virtual void Autentifica_te()
        {
            Console.WriteLine("Clasa de baza");
        }
    }

    public class Derivata : ClasaDeBaza
    {
        public override void Autentifica_te()
        {
            Console.WriteLine("Clasa derivata");
        }
    }
}
```

Rezultat:



b. Modul *new*

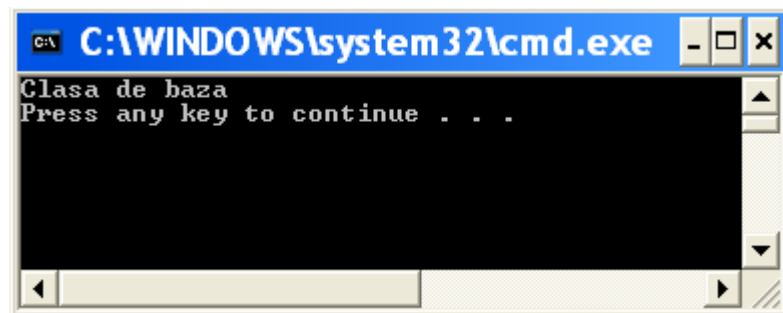
```
using System;

namespace TestOvr
{
    class Program
    {
        static void Main(string[] args)
        {
            ClasaDeBaza obj = new Derivata();
            obj.Autentifica_te();
        }
    }

    public class ClasaDeBaza
    {
        public virtual void Autentifica_te()
        {
            Console.WriteLine("Clasa de baza");
        }
    }

    public class Derivata : ClasaDeBaza
    {
        public new void Autentifica_te()
        {
            Console.WriteLine("Clasa derivata");
        }
    }
}
```

Rezultat:



Tratarea excepțiilor

În timpul exectării unui program pot apărea situații în care o eroare (denumită și *excepție program*) produsă într-un modul al programului face imposibilă continuarea acestuia. Dacă uneori se pot include secvențe de validare a datelor pentru ca situațiile de eroare să fie evitate, în alte cazuri acest lucru nu este posibil sau ar complica mult scrierea programului. O modalitate elegantă de rezolvare a situațiilor care pot apărea o

constituie interceptarea prin program și tratarea excepțiilor. În principiu trebuie realizată o încadrare în structuri *try - catch* a secvențelor de cod care ar putea genera erori în timpul execuției, după modelul:

```
try
{
    . . (secventa de cod generatoare de erori)
}
catch (Exception ex)
{
    . . . (tratare excepție)
}
```

O secvență de cod poate declanșa uneori mai multe tipuri excepții. C# permite în aceste cazuri interceptarea separată a fiecăruia dintre tipurile care ar putea apărea, după modelul:

```
try
{
    . . . (secventa de cod generatoare de erori)
}
catch (NullReferenceException ex)
{
    . . . (tratare excepție 1)
}

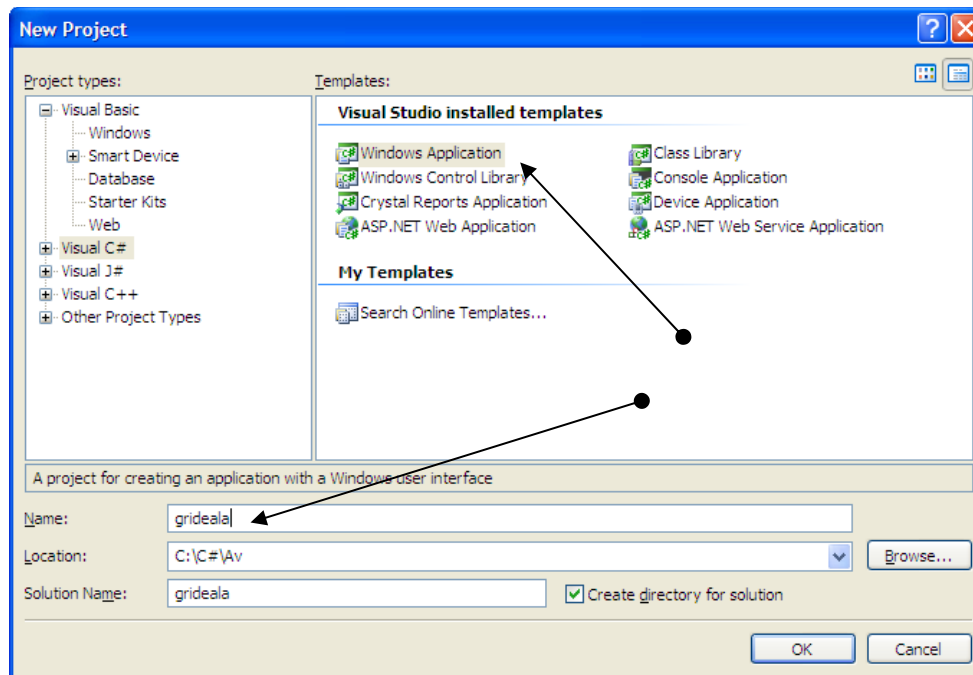
catch (ArgumentOutOfRangeException ex)
{
    . . . (tratare excepție 2)
}

catch (Exception ex)
{
    . . . (tratare excepție 3)
}
```

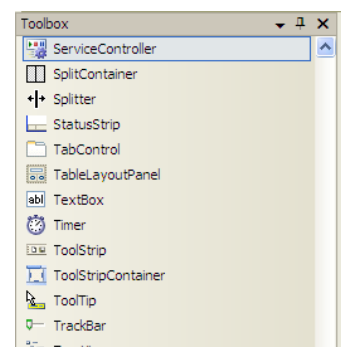
Aplicația astfel structurată conține trei blocuri de tratare a excepțiilor, blocul executat stabilindu-se în mod automat în timpul execuției. Ultimul bloc primește ca argument un obiect din clasa *Exception*. Această clasă este rădăcina ierarhiei de clase folosite la specificarea naturii excepției, deci indiferent ce excepție este generată, dacă niciunul dintre blocurile *catch* n-a fost executat, se execută ultimul bloc.

Realizarea unui program simplu în format windows

Pentru a exemplifica modul de funcționare al Visual Studio 2005 în format windows, să încercăm să implementăm aplicația de calcul a greutatei ideale, pe care o cunpașteți. Să pornim Visual Studio 2005 și să creăm un nou proiect. De data aceasta, vom alege ca șablon de proiect *Windows Application*. Fie *grideala* numele proiectului.



Aplicația ne va afișa un panou, în care putem realiza interfața aplicației. După cum se observă, în centrul panoului se găsește o machetă, numită *Form1*, pe care vom construi interfața. În partea stângă, este disponibilă o bară de instrumente (Toolbox) care va conține controalele pe care le vom utiliza la construirea interfeței. Aceste controale pot fi aduse pe macheta formei cu ajutorul mouse-ului. Toolbox-ul este echivalentul componentei *Palette* din NetBeans.



În partea dreaptă, avem disponibile panourile *Solution Explorer* (echivalent oarecum cu *Inspector* din NetBeans) și *Properties* (același rol ca și în NetBeans). Primul ne afișează componentele proiectului, iar cel de-al doilea, proprietățile obiectelor de pe interfață. Să încercăm acum să construim interfața din figura de mai jos.

Pentru construirea interfeței, am utilizat 3 controale *Label*, pentru care am modificat proprietatea **Text** în *Inaltimea(cm):*, *Varsta (ani):* și respectiv *Greutatea (Kg):*, 3 controale *TextBox* aliniate cu controalele *Label*, un control *CheckBox* căruia i-am

schimbat proprietatea **Text** în *Barbat* și 2 controale Button, pentru care am schimbat proprietatea **Text** în *Calcul* și respectiv *Gata*.

Fiecărui control de pe machetă, i se atribuie automat un identificator (nume), în concordanță cu tipul controlului. Astfel, dacă veți selecta pe rând controalele și veți vedea ce nume li s-au atribuit (proprietatea **Name** din Properties), veți vedea de exemplu că controalele TextBox au fost numite implicit *textBox1*, *textBox2*, *textBox3*, controlul CheckBox a fost numit *checkBox1*, iar butoanele *button1*, respectiv *button2*. Putem lucra fără probleme cu aceste denumiri, însă e util de obicei să le schimbăm în concordanță cu contextul programului, deoarece aceste denumiri vor fi utilizate (la fel ca și în NetBeans) pentru specificarea controalelor în codul aplicației.

Să redenumim controalele astfel: controalele TextBox, de sus în jos: ***h***, ***v*** și respectiv ***g***, iar controlul CheckBox ***s***.

```

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

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

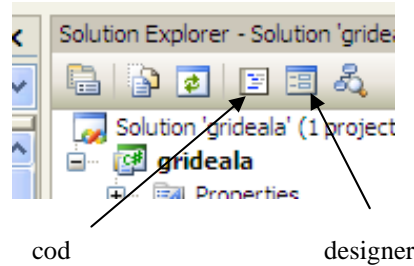
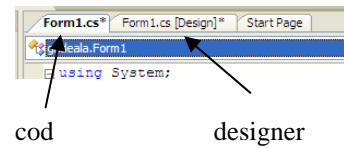
        private void button1_Click(object sender, EventArgs e)
        {
        }
    }
}

```

La fel ca și în NetBeans, apăsarea unui buton va lansa în execuție o funcție. Putem genera această funcție foarte simplu, pur și simplu apăsând dublu click pe buton. A fost generată automat funcția *button1_Click()*, care va fi executată la apăsarea

butonului. Trebuie observat, ca numele funcției este dat de numele controlului căruia îi este asociată (button1) și de numele evenimentului produs asupra acestuia (Click).

Trecerea între Designer (machetă) și cod, se poate face în 2 moduri: fie se selectează din tab-urile din partea de sus a codului, respectiv designerului (figura din dreapta), fie se apasă butoanele din partea de sus a Solution Explorer (fig de mai jos.)



Să implementăm acum funcția:

```
private void button1_Click(object sender, EventArgs e)
{
    Double gre;
    gre=50+0.75*(Convert.ToInt16(h.Text)-150)+0.25*(Convert.ToInt16(v.Text)-20);
    if (!s.Checked)
        gre=gre*0.9;
    g.Text=Convert.ToString(gre);
}
```

Ce observăm.

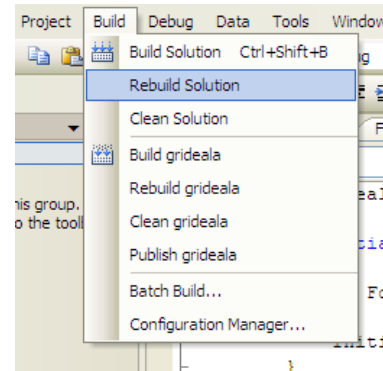
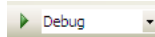
1. Și în C#, la fel ca și în Java, controalele de tip TextBox conțin informații de tip String. Spre deosebire de Java însă, conținutul controlului va putea fi citit sau scris mai simplu. Nu mai este nevoie să folosim metodele `getText()` și `setText()`, ci este suficient să utilizăm conținutul proprietății **Text**. Astfel, pentru controlul `h` de exemplu, linia `h.getText()` din Java va deveni pur și simplu `h.Text` în C#
2. Conversia datelor de la un tip la altul se face cu ajutorul unui obiect **Convert**. Astfel, `Convert.ToInt16()` va converti argumentul funcției într-un întreg, `Convert.ToDouble()` va converti într-o mărime Double, `Convert.ToString()` într-un șir de caractere, etc.
3. Nu mai este nevoie să precizăm tipul explicit al literalilor, expresia fiind evaluată automat la tipul variabilei din stânga (nu mai scriem 50., este suficient să scriem 50).
4. Starea de bifare a controlului CheckBox nu mai este returnată de metoda `isSelected()` ci din nou este inspectată pur și simplu ca o proprietate a acestuia (**Checked**).

În rest, deosebirile sunt minore. Să implementăm acum funcția care se va executa la apăsarea butonului cu proprietatea **Text** Gata. Va trebui să întăim în Designer și să apăsăm dublu click pe buton. Apoi, implementăm codul:

```
private void button2_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

Se poate observa imediat că echivalentul liniei *System.exit(0)* din Java este *Application.Exit()*.

Înainte de execuție, aplicația va trebui compilată. Pentru aceasta, ca în figura din stânga, în meniul Build se alege Rebuild Solution. Dacă apar erori, acestea vor fi afișate în partea de jos a ecranului. Se poate ajunge ușor la linia pe care este eroarea apăsând dublu click pe fiecare linie de eroare în parte. După corectarea erorilor, aplicația este lansată în execuție pri apăsarea butonului triunghiular verde.



Utilizarea controalelor Windows

Să începem să învățăm cum pot fi utilizate diferitele controale puse la dispoziție de Windows Form Designer. Primul control despre care vom discuta îl cunoașteți deja. Este ...

Butonul

Probabil butonul este unul din cele mai utilizate controale în interfețele Windows. Deși are o formă grafică, funcționând ca un buton real, el este în realitate un obiect al clasei **Button**. Spre deosebire de alte clase asociate controalelor, această clasă nu este derivată direct din clasa **Control**, ci din clasa **BaseButton**, deoarece există mai multe tipuri de butoane, despre care vom vorbi în cele ce urmează. Ca obiect, butonul va oferi utilizatorului un set de proprietăți, un set de metode și va fi capabil să producă un set de evenimente. Câteva din proprietăți sunt:

- **FlatStyle** – setează aspectul butonului. Dacă stilul este PopUp, butonul va apare ca fiind plat, chiar dacă utilizatorul pune prompterul mouse-ului deasupra lui. În caz contrar butonul va avea aspect tridimensional;
- **Enabled** – setează starea de activ/inactiv al butonului. Dacă această proprietate este false, butonul apare ca inactiv, toate informațiile de pe buton sunt gri și el nu poate fi apăsător;
- **Image** – permite utilizatorului să specifice o imagine (pictogramă, bitmap, etc.) ce urmează să fie afișată pe buton;
- **ImageAlign** – permite specificarea poziției imaginii în cadrul butonului;
- **TextAlign** – permite specificarea poziției textului în cadrul butonului;

Dacă vorbim despre evenimente, în mod evident evenimentul principal produs de buton este Click. Acesta se produce atunci când utilizatorul apasă butonul cu ajutorul mouse-ului, sau când se apasă tasta Enter și butonul are focusul. Pentru a intercepta acest eveniment, să ne reamintim, va trebui să-i asociem un handler în cadrul formei.

Butoane `RadioButton` și `CheckBox`

Deși aspectul acestor controale este diferit de cel al butonului, din punct de vedere al programării obiectuale ele au aceeași clasă de bază. Vor fi însă obiecte de clase **`RadioButton`** și respectiv **`CheckBox`**.

Butoanele radio sunt utilizate pentru a specifica acțiuni mutual-exclusive, în sensul că la un momentdat, un singur buton radio poate fi selectat, adică poate avea un punct negru în interiorul lui. În mod normal, un singur buton radio din toate cele incluse în formă va putea fi selectat. Pentru a realiza grupări distincte de butoane radio, acestea vor trebui incluse în interiorul unui alt control, numit `GroupBox`, despre care vom vorbi mai târziu. În acest caz, doar un singur buton radio din grup va putea fi selectat. Prin utilizarea lor, butoanele radio permit utilizatorului să aleagă o singură opțiune din mai multe posibile.

Butoanele de opțiune (`CheckBox`) sunt afișate ca și niște casete mici, care pot fi sau nu bifate. Spre deosebire de butoanele radio, starea de bifare al unui astfel de buton este independentă de starea de bifare al celorlalte butoane. Astfel, utilizarea butoanelor de opțiune, permite alegerea unui număr de opțiuni din mai multe posibile.

Să vedem acum care sunt proprietățile și evenimentele specifice fiecărui tip de control în parte.

Câteva din proprietățile `RadioButton`-ului sunt:

- **Appearance** – specifica forma standard al butonului radio, sau forma asemănătoare cu al unui buton obișnuit. În acest al doilea caz, la selectarea butonului radio, acesta va rămâne apăsat;
- **AutoCheck** – dacă această proprietate este true, la selectarea controlului este afișat un punct negru în interiorul lui, controlul devenind marcat. Dacă este false, punctul negru nu este afișat;
- **CheckAlign** – prin intermediul acestei proprietăți, se poate alinia textul asociat controlului în raport cu controlul însuși. Poate fi left, middle, right;
- **Checked** – indică starea controlului. Dacă controlul afișează punctul negru este true;

Evenimentele specifice butonului radio sunt:

- **CheckChanged** – este generat la schimbarea stării butonului radio. Dacă există un grup de butoane radio, acest eveniment va fi generat de două ori: o dată pentru controlul care era marcat și acum devine nemarcat și apoi pentru controlul care se marchează;
- **Click** – este generat atunci când se apasă click pe buton.
În mod similar și controlul `CheckBox` va prezenta un set de proprietăți specifice, care suplimentar față de cele ale butonului radio sunt:
- **CheckState** – Spre deosebire de controlul `RadioButton`, care are doar 2 stări (marcat sau nemarcat), controlul `CheckBox` poate avea 3 stări: *marcat*, *nedeterminat* și *nemarcat* (*Checked*, *Indeterminate* și *Unchecked*). În starea nedeterminat, eticheta asociată controlului devine gri, pentru a specifica faptul că valoarea marcatului este nevalidă, sau, starea controlului nu are nici o semnificație în circumstanța actuală.

- **ThreeState** – dacă această proprietate este false, controlul nu poate intra în starea nedeterminat prin setarea proprietăților, ci doar prin intermediul metodelor asociate.

Și acum evenimentele:

- **CheckedChanged** – este generat la modificarea stării de marcare a controlului (proprietatea **Checked**). Interesant este că la un control cu proprietatea **ThreeState** poziționată pe **true**, se poate întâmpla ca să se schimbe starea de marcare fără ca acest eveniment să se producă. Aceasta se întâmplă când starea controlului devine nedeterminată.
- **CheckedStateChanged** – este generat la modificarea valorii proprietății **CheckedState**. Evenimentul este generat și la trecerea în starea nedeterminat. Cam atât despre aceste controale.

Controlul GroupBox

Este un control utilizat pentru a realiza grupări de controale de același tip. Uzual este utilizat împreună cu controale **RadioButton** sau **CheckBox**. Este afișat ca un cadru dreptunghiular în interiorul căruia se poate realiza gruparea celorlalte controale. Dacă un set de controale de același tip sunt plasate în interiorul casetei de grupare, aceasta devine controlul părinte al grupului, în locul formei. Astfel, de exemplu, ca efect, un singur buton radio din interiorul casetei de grupare va putea fi marcat la un momentdat. De asemenea, prin plasarea controalelor în interiorul casetei de grupare, un set de proprietăți ale acestora va putea fi modificat prin simpla modificare a proprietății corespunzătoare pentru caseta de grupare.

Controalele Label și LinkLabel

Sunt controale care în principiu afișează etichete pentru clarificarea funcției altor controale sau, respectiv, legături spre diferite adrese de internet. Apar ca texte afișate pe ecran (în cazul **LinkLabel** textul este subliniat). Câteva din proprietățile comune celor 2 controale, care pot fi modificate de utilizator sunt:

- **BorderStyle** – specifică tipul chenarului care înconjoară controlul. Implicit, nu există chenar.
- **FlatStyle** – specifică modul în care este afișat controlul.
- **Image** – permite specificarea unei imagini (bitmap, icon, jpg, etc) care va fi afișată în interiorul controlului.
- **ImageAlign** – specifică poziția imaginii afișate referitor la marginile controlului.
- **Text** – specifică textul afișat de către control.
- **TextAlign** - specifică poziția textului referitor la marginile controlului.

Controlul TextBox

Controalele **TextBox** sunt probabil cele mai utilizate controale pentru interfețele intrare-ieșire. Prin modul lor de funcționare permit introducerea sau afișarea unor tipuri de date diferite, permițând de asemenea ascunderea acestora față de utilizator (parole). Deși există 2 clase de astfel de controale (**TextBox** și respectiv **RichTextBox**) derivate din clasa **TextBaseBox**, în acest moment ne vom ocupa doar de controalele standard, adică de prima clasă.

Controalele de tip TextBox permit manipularea textelor în formele cu care sunteți obișnuiți (copy, paste, delete, etc.). Și acum sa vedem câte proprietăți și evenimente pentru aceste controale.

Proprietăți:

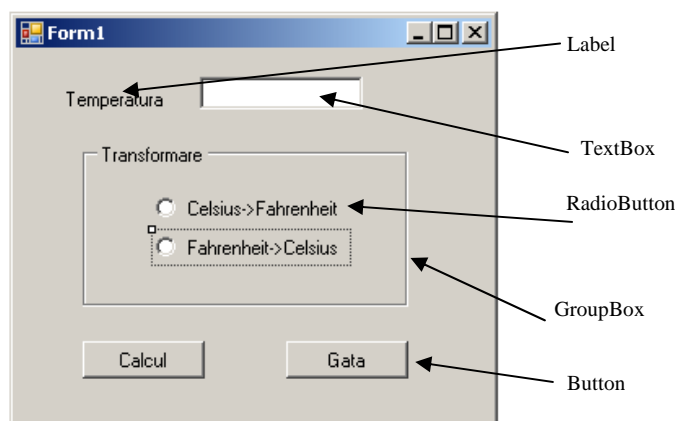
- **CausesValidation** – dacă această proprietate este true, la primirea focusului controlul va genera 2 evenimente: Validating și Validated. Aceste evenimente sunt uzual utilizate pentru validarea datelor conținute de control înainte de pierderea focusului.
- **CharacterCasing** – specifică tipul literelor cu care sunt afișate textele în control: Lower – toate literele sunt minuscule, Normal – și minuscule și majuscule, respectiv Upper – toate literele sunt majuscule.
- **MaxLength** – specifică numărul maxim de caractere a textului din control. Dacă valoarea acestei proprietăți este 0, lungimea textului este limitată doar de capacitatea de memorie a calculatorului.
- **Multiline** – uzual, controlul afișează o singură linie de text. Prin setarea acestei proprietăți, controlul va fi capabil să afișeze mai multe linii.
- **PasswordChar** – textul este afișat sub formă de parolă (steluțe). Dacă proprietatea Multiline este true, această proprietate nu funcționează.
- **ReadOnly** – dacă această proprietate este true, controlul va permite doar afișarea textelor, nu și introducerea lor.
- **ScrollBars** – forțează controlul să afișeze bare de derulare.
- **SelectedText** – proprietatea specifică textul selectat în interiorul controlului.
- **SelectionLength** – Lungimea textului selectat în control.
- **SelectionStart** – indicele primului caracter din textul selectat în control.

Câte ceva despre evenimente:

- **Enter, GotFocus, Leave, Validating, Validated, LostFocus** – Aceste evenimente sunt generate în ordinea în care au fost prezentate. Sunt așa numitele evenimente de focus. Sunt generate ori de câte ori controlul de tip TextBox primește focusul, mai puțin Validating și Validated care se produc dacă și numai dacă controlul are proprietatea CausesValidation setată la true.
- **KeyDown, KeyPress, KeyUp** – Sunt așa numitele evenimente de taste. Permit monitorizarea modificărilor produse în textul din control prin intermediul tastaturii. KeyDown și KeyUp recepționează codul de scanare al tastei acționate (vezi **unu**). KeyPress recepționează în schimb codul ASCII al tastei.
- **Change** – este generat ori de câte ori textul conținut de control este modificat.

Și acum, pentru a înțelege mai bine modul de funcționare al acestor controale, să încercăm să refacem exemplele din capitolul de Java: calculu temperaturii și respectiv formatarea textului.

1. **Butoane radio.** Să creăm un proiect intitulat *Temperatura*, de tip Windows Application. Vom realiza interfața din figura în dreapta. Pentru controalele de pe interfață,



vom modifica proprietățile în felul următor: pentru controlul Label, proprietatea **Text** în *Temperatura*; pentru controlul TextBox, proprietatea **Name** în *temp*. Pentru controalele RadioButton, de sus în jos, proprietatea **Text** în *Celsius->Fahrenheit* și **Name** în *cf*, respective **Text** în *Fahrenheit->Celsius* și **Name** în *fc*; pentru controlul GroupBox, proprietatea **Text** în *Transformare*; Pentru controalele Button, de la stânga la dreapta, proprietatea **Text** în *Calcul* și respectiv *Gata*.

Putem crea imediat funcția asociată apăsării butonului Gata: dubli click pe buton și implementăm codul:

```
private void button2_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

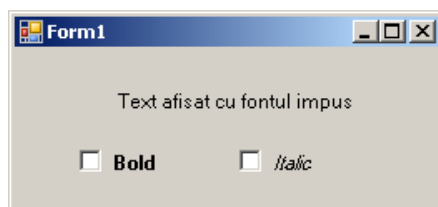
Am dori ca implicit, transformarea să se faca Celsius->Fahrenheit, adică la pornirea programului să fie bifat implicit controlul CF. Starea de bifare a unui astfel de control, după cum am văzut mai înainte, este memorată în proprietatea **Checked**. Deci vom adăuga programului linia

```
public Form1()
{
    InitializeComponent();
    cf.Checked = true;
}
```

Și acum să implementăm funcția asociată apăsării butonului Calcul:

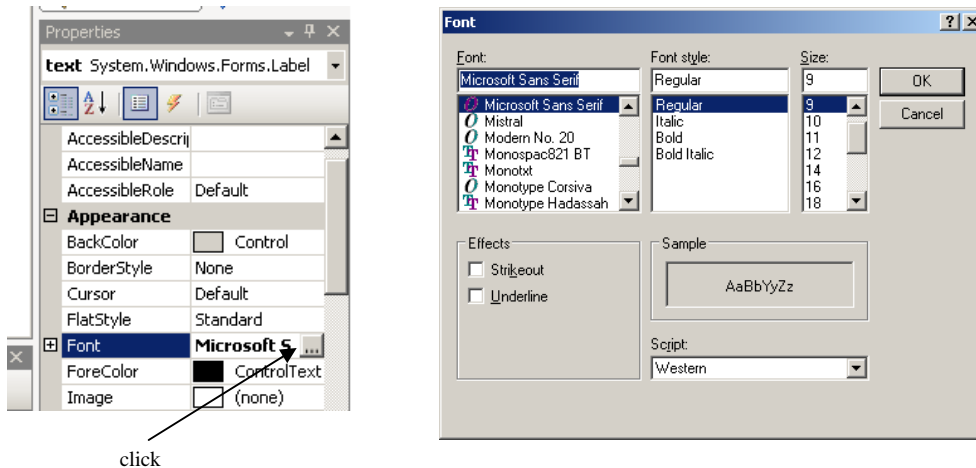
```
private void button1_Click(object sender, EventArgs e)
{
    Double v = Convert.ToDouble(temp.Text);
    if (cf.Checked)
        v = 9.0 / 5.0 * v + 32;
    else
        v = 5.0 / 9.0 * (v - 32);
    temp.Text = Convert.ToString(v);
}
```

2. **CheckBox** – aplicația va fi la fel de simplă. Să creăm un nou proiect, numit *FormatText*, pentru care să construim interfața de mai jos:



După cum se poate vedea, avem un control Label și 2 controale CheckBox. Să modificăm pentru controlul Label proprietatea **Text** în *Text afișat cu fontul impus*, proprietatea **Name** în *text* și respectiv la proprietatea **Font** să modificăm dimensiunea

fontului la 9. Pentru aceasta, apăsați click în partea dreaptă a proprietății și apăsați butonul apărut. Apoi, în caseta de dialog Font, modificați dimensiunea fontului.



click

În mod similar, vom modifica pentru cele 2 controale CheckBox, de la stânga la dreapta, proprietățile în felul următor: **Text** în *Bold*, respectiv *Italic*; **Name** în *B* respectiv *I*; **Font** se selectează *Bold*, respectiv *Italic*.

Apăsarea succesivă a controlului CheckBox va produce evenimentul **CheckChange**. Pentru a asocia acestui eveniment o funcție, este suficient să apăsăm dublu click asupra lui.

Funcția asociată controlului B va fi:

```
private void B_CheckedChanged(object sender, EventArgs e)
{
    FontStyle f = text.Font.Style;
    text.Font = new Font("Microsoft Sans Serif", 9, f ^ FontStyle.Bold);
}
```

Să explicăm puțin. În C# literele sunt obiecte de clasă Font, deci ele vor putea fi construite, astfel încât să arate cum dorim noi. Unele dintre caracteristicile fontului sunt la rândul lor obiecte de clase specifice. Astfel, stilul literei este un obiect de clasă FontStyle. În prima linie de program, am declarat un obiect de clasă FontStyle, căruia i-am atribuit stilul curent al literelor cu care este scris textul. Sintaxa este ușor de înțeles: stilul unui obiect de clasă Font este memorat în proprietatea **Style**, iar fontul textului este memorat în proprietatea **Font**, rezultând sintaxa *text.Font.Style*.

Un obiect de clasă FontStyle este în esență o mască de biți, având un singur bit 1 pe o poziție specifică stilului respectiv, restul biților fiind 0. Pentru exemplificare, să presupunem ca stilul este reprezentat pe 8 biți, bitul de pondere 1 fiind Bold. Să presupunem de asemenea ca valoare curentă a stilului este 00110000. Biții 1 identifică componentele de stil prezente în stilul fontului. Pentru a adăuga stilul Bold, va trebui să setăm bitul de pondere 1 la valoarea 1. Acest lucru este foarte ușor, prin intermediul unei operații SAU pe bit.

	0	1
0	0	1
1	1	1

&	0	1
0	0	0
1	0	1

^	0	1
0	0	1
1	1	0

Cum masca pentru Bold va fi 00000010, prin intermediul operației SAU pe bit vom obține

```
00110000 |
00000010
00110010
```

Adică stilului curent îi va fi adăugat și Bold. Operația de retragere a stilului Bold aparent se poate face similar, printr-o operație ȘI pe bit cu inversul măștii stilui, adică

```
00110000 &
11111101
00110000
```

Din păcate, nu se poate scrie `!FontStyle.Bold`, pentru că obiectele `FontStyle` nu suportă operația de negare. Ce-i de făcut? Mecanismul este simplu și este moștenit din tehnicile caracteristice limbajelor de asamblare. Se realizează o operație SAU EXCLUSIV cu masca de biți. Astfel, dacă bitul luat în considerare este setat, el va fi resetat, în caz contrar va fi setat.

```
00110000 ^      00110010 ^
00000010      00000010
00110010      00110000
```

Este ceea ce am făcut în a doua linie de program. Am impus fontului cu care este scris textul, un nou font, cu numele „Microsoft Sans Serif”, de dimensiune 9, cu stilul `f ^ FontStyle.Bold`. Astfel, dacă se bifează butonul, textul va deveni Bold, iar dacă se debifează, textul va deveni normal. Mecanismul este în acest caz complet diferit față de cel din Java, în care stilul asociat unui font era o valoare întreagă.

Similar procedăm pentru Italic:

```
private void I_CheckedChanged(object sender, EventArgs e)
{
    FontStyle f = text.Font.Style;
    text.Font = new Font("Microsoft Sans Serif", 9, f ^ FontStyle.Italic);
}
```

Controale cu listă

Controlul *ComboBox*

Acest control combină în fapt mai multe controale. În acest control vom regăsi un control `TextBox`, prezentat anterior și respectiv un control `ListBox` despre care vom vorbi puțin mai târziu. Controlul `ComboBox` apare ca un `TextBox` având în partea stângă un buton cu săgeată. Dacă se apasă această, controlul se deschide, prezentând o listă de elemente. Orice astfel de control este un obiect de clasă **ComboBox**.

Proprietăți ale acestui control sunt:

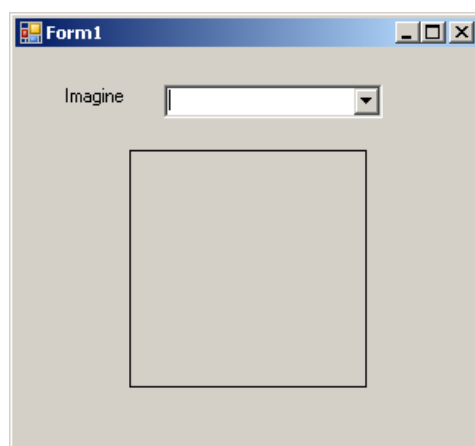
- **DropDownStyle** - stilul în care este afișat controlul la deschiderea listei.
 - **DropDown** – utilizatorul poate edita partea de `TextBox` a controlului și trebuie să apese butonul săgeată pentru a deschide partea de `ListBox`.
 - **Simple** – la fel ca și `DropDown`, cu excepția faptului că partea de `ListBox` a controlului este tot timpul vizibilă.

- **DropDownList** – utilizatorul nu poate edita partea de TextBox a controlului și trebuie să apese butonul săgeată pentru a deschide partea de ListBox.
- **DroppedDown** – Indică dacă partea de listă a controlului este deschisă sau nu.
- **Items** – este reprezentată sub forma unei colecții care stochează obiectele conținute de ComboBox.
- **MaxLength** – prin setarea acestei proprietăți la o valoare diferită de 0, se specifică numărul maxim de caractere ce pot fi introduse în partea de TextBox a controlului.
- **SelectedIndex** – indică indexul obiectului curent selectat în lista controlului. Obiectele sunt ordonate în listă pe baza unui index bazat pe 0.
- **SelectedItem** – indică obiectul curent selectat în listă.
- **SelectionStart** – indică indexul primului obiect selectat în lista asociată controlului.
- **SelectionLength** – lungimea textului selectată în partea de TextBox asociată controlului.
- **Sorted** – dacă această proprietate este true, elementele din lista asociată controlului vor fi sortate în ordine alfabetică.

Câteva evenimente sunt:

- **DropDown** – se produce când partea de ListBox a controlului se deschide.
- **SelectedIndexChanged** – se produce la schimbarea selecției în zona de listă a controlului.
- **KeyDown** – se produce când o tastă este apăsată în zona de TextBox a controlului.
- **KeyUp** - se produce când o tastă este eliberată în zona de TextBox a controlului.
- **TextChanged** – se produce la schimbarea textului din partea de TextBox a controlului.

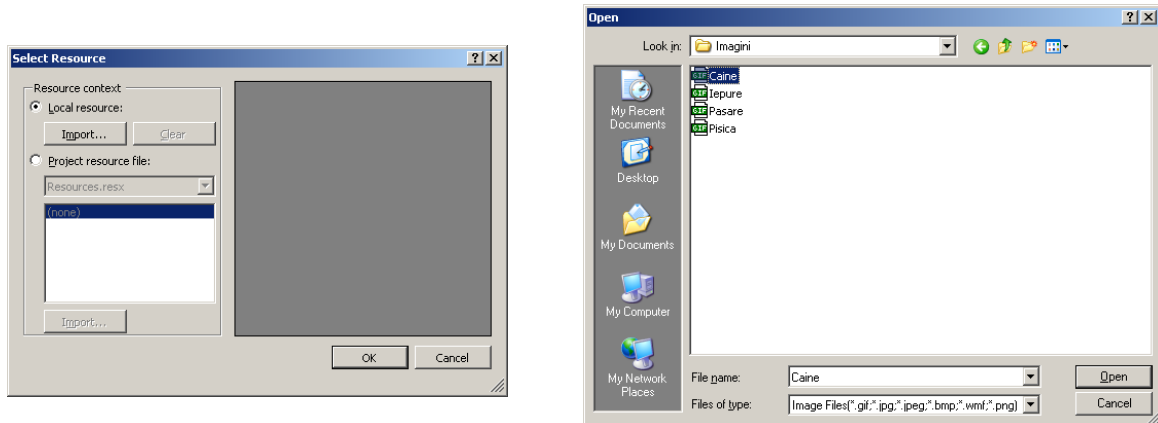
Pentru a înțelege modul de manifestare al acestui contro, să reluăm aplicația din capitolul de Java. Vom crea un noi proiect de tip Windows Application, numit Combomagini. Pentru acest control vom realiza interfața din figura de mai jos.



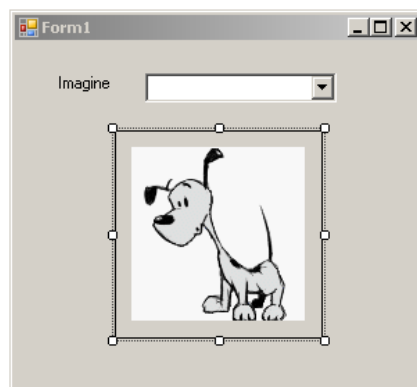
Interfața conține 2 controale Label și un control ComboBox. Să modificăm proprietățile controalelor astfel: pentru controlul Label de sus, proprietatea **Text** în *Imagine*; pentru controlul ComboBox, proprietatea **Name** în *cbimagine*. Pentru controlul Label de jos: ștergem conținutul proprietății **Text**; modificăm proprietatea **Name** în *image*; modificăm proprietatea **AutoSize** în *False* (astfel vom putea stabili noi dimensiunile controlului); deschidem subarborele proprietății **Size** și modificăm valorile de la **Width** și **Height** la *150*; modificăm proprietatea **BorderStyle** la *FixedSingle*.

Acum trebuie să adăugăm proiectului directorul Imagini care conține fișierele .gif care urmează să fie afișate. Probabil că ați observat că pentru fiecare soluție deschisă, aplicația wizzard crează un director cu același nume, care conține la rândul lui un număr de subdirectoare. Noi va trebui să copiem directorul Imagini în subdirectorul care conține programul executabil. Acesta este *Combolmagini\Combolmagini\bin\Debug*.

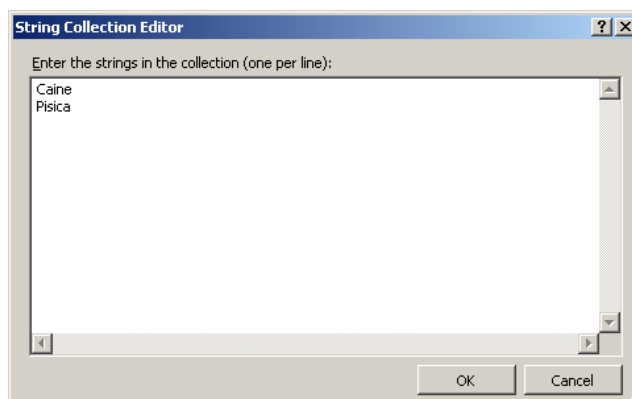
Vom insera acum în controlul **imagini** imaginea din fișierul *Câine.gif*. Pentru aceasta, vom apăsa click pe proprietatea **Image** și apoi butonașul apărut.



În caseta **Select Resource**, vom bifa opțiunea **Local resource**: și vom apăsa butonul **Import...**. Apoi, în caseta **Open** vom selecta fișierul dorit, vom apăsa butonul **Open** și apoi **OK**. Astfel imaginea va fi afișată în control.



Acum să populăm controlul **ComboBox**. Putem să-l populăm adăugând elemente la proprietatea **Items**, apăsând butonul din dreapta și adăugând informația în caseta apărută:



După introducerea tuturor informațiilor, se apasă butonul OK și controlul este populat. Dar, așa cum spuneam și în capitolul de Java, o soluție mult mai elegantă este popularea controlului prin cod.

Adăugarea elementelor în lista controlului `ComboBox` se face prin completarea colecției `Items`. Două dintre metodele care permit completarea acestei colecții sunt:

- **Add(Object obj)** – adaugă la începutul listei obiectul specificat;
- **Insert(int Index, Object obj)** – adaugă obiectul specificat în listă pe poziția `Index`.

Vom popula obiectul `ComboBox` prin codul de mai jos:

```
public Form1()
{
    InitializeComponent();
    cbimage.Items.Add("Caine");
    cbimage.Items.Add("Pisica");
    cbimage.Items.Add("Pasare");
    cbimage.Items.Insert(1, "Iepure");
    cbimage.SelectedIndex = 0;
}
```

Observați faptul că prin stabilirea indicelui selectat ca fiind 0 în controlul `ComboBox` este afișat implicit textul "Caine", iar ordinea în care apar textele în listă este "Caine", "Iepure", "Pisica" respectiv "Pasare".

La schimbarea selecției în controlul `ComboBox`, va trebui ca imaginea să se schimbe. Adică, va trebui să implementăm o funcție, care să se execute la producerea evenimentului **SelectedIndexChanged**, care să schimbe imaginea. Este de menționat că acest eveniment se tratează în peste 90% din aplicațiile care implică controale `ComboBox`. Crearea acestei funcții este din nou foarte simplă, pur și simplu apăsăm dubli click pe controlul `ComboBox`:

```
private void cbimage_SelectedIndexChanged(object sender, EventArgs e)
{
    String nume;
    nume = "Imagini/" + cbimage.SelectedItem + ".gif";
    image.Image = Image.FromFile(nume);
}
```

Codul funcției este foarte asemănător cu cel din Java. Se compune întâi numele complet al fișierului care conține imaginea, apoi proprietatea **Image** a controlului *image* este actualizată cu o imagine preluată din acel fișier (**Image.FromFile()**).

Controalele `ListBox` și `CheckedListBox`

Controalele de acest tip sunt utilizate pentru a afișa un set de stringuri, din care unul sau mai multe pot fi selectate la un momentdat. Clasa **ListBox** oferă funcționalitate atât controlului `ListBox` cât controlului `ComboBox`. Clasa **CheckedListBox** este derivată din aceasta și adaugă fiecărui string din listă un control de tip `CheckBox`, utilizat pentru selectare.

Câteve din proprietățile furnizate de clasa **ListBox** sunt:

- **SelectedIndex** – indică indicele bazat pe 0 a elementului selectat în listă, sau a primului element selectat în listă, în cazul selecției multiple.
- **ColumnWidth** – specifică lățimea coloanelor, în listele cu coloane multiple.
- **Items** – conține sub forma unei colecții toate elementele stocate în listă.
- **Multicolumn** – specifică numărul de coloane din listă.
- **SelectedIndices** – o colecție care conține toți indicii elementelor selectate din listă.
- **SelectedItem** – această proprietate conține elementul selectat în listă dacă selecția este simplă, respectiv primul element selectat din listă în cazul selecției multiple.
- **SelectedItems** – o colecție care conține elementele selectate din listă.
- **Sorted** – dacă această proprietate este true, elementele vor fi afișate în listă în ordine alfabetică.
- **CheckedIndices** - o colecție care conține indicii elementelor din listă care au caseta checkbox bifată su în stare nedeterminată (doar pentru CheckedListBox).
- **CheckedItems** - o colecție care conține elementele din listă care au caseta checkbox bifată su în stare nedeterminată (doar pentru CheckedListBox).
- **CheckOnClick** – dacă această proprietate este true, starea unui element se schimbă când asupra lui se efectuează click.

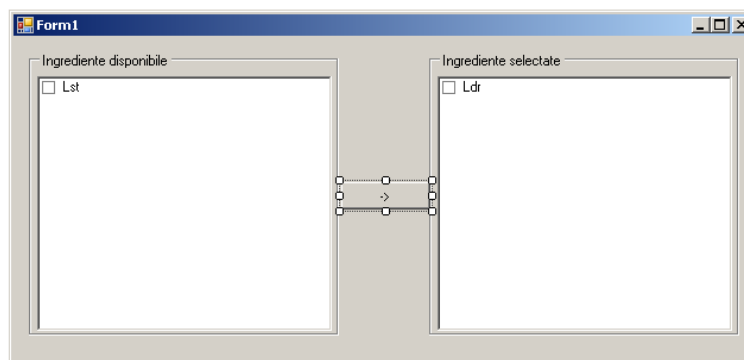
Câteva din metodele clasei:

- **ClearSelected()** – șterge toate selecțiile (nu elementele selectate!) din listă.
- **FindString()** – caută primul string care începe cu șirul specificat ca parametru în listă.
- **GetSelected()** – returnează o valoare care specifică dacă un element este selectat.
- **SetSelected()** – setează sau șterge selectarea unui element.
- **GetItemChecked()** – returnează o valoare care indică faptul că checkbox-ul asociat unui element este bifat (doar pentru CheckedListBox).
- **GetItemCheckState()** – returnează o valoare care indică starea casetei checkbox asociată elementului (doar pentru CheckedListBox).
- **SetItemChecked()** – setează starea casetei checkbox a elementului specificat într-una din stările posibile (doar pentru CheckedListBox).
- **SetItemCheckState()** – Setează starea unui element (doar pentru CheckedListBox).

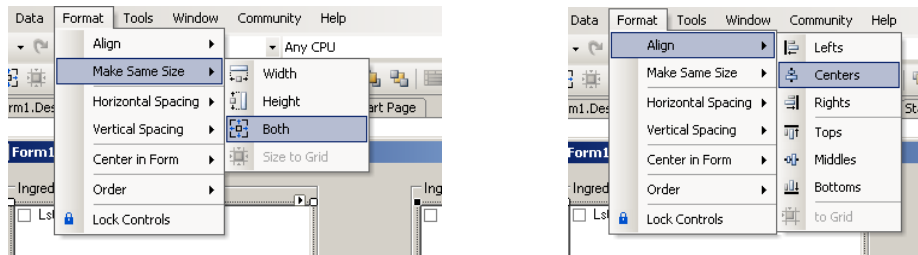
Câteva evenimente:

- **ItemCheck** – se produce când starea de check a unui element se schimbă.
- **SelectedItemChanged** – se produce la schimbarea indexului elementelor selectate.

Să refacem exemplul din Java. Vom deschide un nou proiect numit *Liste*, cu interfața de mai jos.



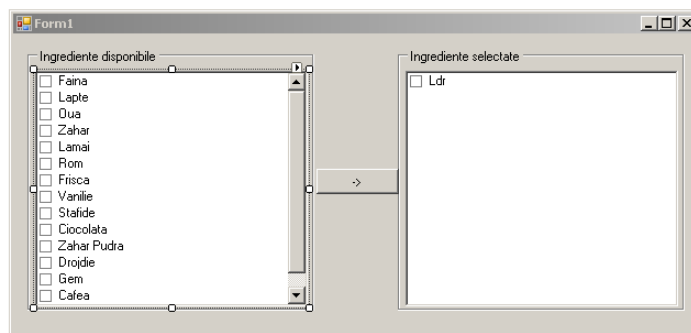
Am utilizat 2 controale **GroupBox** pentru care am modificat proprietatea **Text** în *Ingrediente disponibile*, respectiv *Ingrediente selectate*, 2 controale **CheckedListBox** cu proprietatea **Name** *Lst* și respectiv *Ldr* și un control **Button** cu proprietatea **Text** ->. De asemenea, pentru lista din stânga vom seta proprietatea **CheckOnClick** la *true*, pentru a putea bifa elementele selectate. Înainte de a face altceva, pentru a obține o interfață estetică, este de dorit să aliniem și să facem de aceeași dimensiune controalele de același tip. Pentru aceasta, se selectează pe rând, 2 câte 2, controalele **GroupBox** și **CheckedListBox**. Dimensiunea se face egală ca în figura de mai jos din stânga, iar alinierea ca în figura de mai jos din dreapta.



Vom adăuga acum ingredientele în lista din stânga. Să adăugăm următoarele ingrediente: *Faina, Lapte, Oua, Zahar, Lamai, Rom, Frisca, Vanilie, Stafide, Ciocolata, Zahar Pudra, Drojdie, Gem, Cafea, Scortisoara*. Avem din nou 2 posibilități de adăugare: fie la proprietatea **Items** a obiectului *Lst*, completând lista, fie prin cod, cu ajutorul unor instrucțiuni de forma

```
...
Lst.Items.Add("Faina");
...
```

Vom adăuga de această dată ca și informație la proprietatea **Items**. Observați adăugarea automată a barei de scroll atunci când informația depășește dimensiunea listei.



Și acum, la apăsarea butonului, să trecem elementele selectate în lista din dreapta. Pentru aceasta, dublu click pe buton și:

```
private void button1_Click(object sender, EventArgs e)
{
    if (Lst.CheckedItems.Count > 0)
    {
        Ldr.Items.Clear();
        foreach (string item in Lst.CheckedItems)
        {
```



```

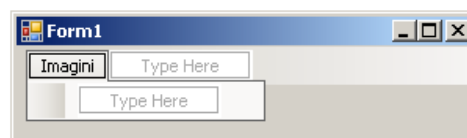
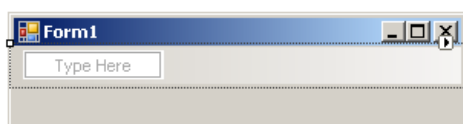
{
    Ldr.Items.Add(item.ToString());
}
for (int i = 0; i < Lst.Items.Count; i++)
    Lst.SetItemChecked(i, false);
}
}

```

Ce se petrece de fapt. Se numără întâi elementele selectate în lista din stânga (*Lst.CheckedItems.Count*). Dacă există elemente selectate, se șterge lista din dreapta. Apoi, **fiecare element selectat în lista din stânga** (*foreach (string item in Lst.CheckedItems)*) este adăugat (*după convertirea la șir de caractere! Să nu uităm că elementele colecției Items sunt obiecte **Objects**!*) în lista din dreapta. Apoi, proprietatea *Checked* este resetată pentru toate obiectele din colecția listei din stânga.

Meniu

Un meniu este foarte ușor de adăugat și configurat. În **Toolbar**, la **Menus&Toolbars** căutați obiectul *MenuStrip*. Acesta poate fi adus pe machetă și configurat.



Veți observa apariția unei casete în care este afișat textul *Type Here*. Introduceți acolo textul primei intrări de tip popup. În momentul tastării textului, veți observa că se deschid alte 2 casete, permițând adăugarea intrărilor de meniu, respective a unei alte intrări popup.

Realizați un meniu cu 2 intrări popup, *Imagini* și *Culori*. Adăugați în lista intrării *Imagini* elementele de meniu *Caine*, *Pisica*, *Pasare* și *Iepure*, iar în lista intrării *Culori* *Rosu*, *Albastru*, *Verde* și *Isire*. Pentru fiecare element de meniu, schimbați proprietatea **Name** la fel ca și proprietatea **Text** (adică *Caine* – *Caine*, etc).

Să adăugăm acum cele 2 controale *Label* ca și în exemplul din Java. Primului control să-l schimbăm proprietatea **Text** în *Imagine*, iar celui de al doilea în *Nume*. De asemenea, pentru primul control să modificăm proprietatea **AutoSize** în *False* și valorile de la **Width** și **Height** la 150; modificăm proprietatea **BorderStyle** la *FixedSingle*. De asemenea, pentru cel de-al doilea control să modificăm proprietatea **Text** în *Caine*.

Să copiem acum directorul *Imagini* în directorul *Debug* și să afișăm imaginea *Caine.gif* în controlul *Imagini*.

Și acum, să implementăm metodele care se lansează în execuție la click pe un element de meniu. Cum? Foarte simplu. Apăsăm dublu click pe fiecare element de meniu în parte:

```

private void Caine_Click(object sender, EventArgs e)
{
    String nume;
    nume = "Imagini/" + Caine.Text + ".gif";
    Imagini.Image = Image.FromFile(nume);
}

```

```

    Nume.Text = Caine.Text;
}

```

Ar trebui să obținem 4 astfel de funcții, în principiu identice, singura diferență fiind numele elementului de meniu care le lansează în execuție. E corect, dar neelegant din punctul de vedere al programării, pentru că rezultă o mulțime de funcții identice. Mai elegant ar fi să scriem o singură funcție, care să fie lansată în execuție de click pe oricare din intrările de meniu, iar în codul funcției să fie identificată intrarea și tratată în consecință.

Pentru a vedea cum se poate face acest lucru, să ne uităm mai atent la antetul funcției *Caine_Click()*. Acest antet este

```
private void Caine_Click(object sender, EventArgs e)
```

Se observă că numele dat de către infrastructură funcției este dat de elementul căreia îi este asociată și evenimentul care o lansează în execuție. De asemenea, observăm că funcția primește 2 parametri. Primul este un obiect *object*, iar al doilea un obiect *EventArgs*. La apelul funcției, infrastructura transmite întotdeauna pe primul parametru numele obiectului căreia îi este asociată funcția. Este de clasă *object*, pentru că, să ne reamintim, toate obiectele grafice utilizate pe interfață sunt de clase derivate din aceasta. Deci, vom putea obține clasa corectă a elementului ce generează funcția, printr-o simplă operație de transtipaj. În cazul nostru, elementele de meniu sunt obiecte de clasă *ToolStripMenuItem*. Deci, am putea scrie funcția astfel:

```

private void Caine_Click(object sender, EventArgs e)
{
    ToolStripMenuItem intrare = (ToolStripMenuItem) sender;
    String nume;
    nume = "Imagini/" + intrare.Text + ".gif";
    Imagini.Image = Image.FromFile(nume);
    Nume.Text = intrare.Text;
}

```

Obiectul *intrare* va conține întotdeauna intrarea de meniu pe care s-a apăsă click. Dar cum facem acum să poată fi și lansată în execuție la selectarea unei intrări de meniu? Mecanismul prin care o funcție este lansată în execuție la producerea unui eveniment de către un obiect de interfață este simplu. Se crează un **handler**, care face legătura dinte obiect și funcție, precizând evenimentul care o lansează. Forma generală a unui astfel de handler, este

```
obiect.eventiment += new tip_handler(funcție);
```

unde *obiect* este obiectul căruia îi este asociat handlerul, *eveniment* este evenimentul care lansează funcția, *tip_handler* este în general (depinde de eveniment) *EventHandler* iar *funcție* este numele funcției care va fi lansată în execuție. Deci, pentru a asocia evenimentului *Click* aceeași funcție și pentru celelalte 3 intrări de meniu, va fi suficient să definim cei trei handleri asociați:

```

public Form1()
{

```

```

InitializeComponent();
Pisica.Click += new EventHandler(Caine_Click);
Pasare.Click += new EventHandler(Caine_Click);
Iepure.Click += new EventHandler(Caine_Click);
}

```

Compilați și lansați în execuție programul. Observați funcționarea corectă.

Să schimbăm acum culoarea textului. Dublu click pe *Rosu* și implemenăm codul:

```

private void Rosu_Click(object sender, EventArgs e)
{
    Nume.ForeColor = Color.Red;
}

```

Codul nu necesită explicații suplimentare. Din nou, ar fi bine să scriem o singură funcție care să trateze intrările din meniul *Culori*. Întâi vom scrie handlerii:

```

public Form1()
{
    InitializeComponent();
    ...
    Iepure.Click += new EventHandler(Caine_Click);
    Albastru.Click += new EventHandler(Rosu_Click);
    Verde.Click += new EventHandler(Rosu_Click);
    Iesire.Click += new EventHandler(Rosu_Click);
}

```

Și apoi vom modifica funcția:

```

private void Rosu_Click(object sender, EventArgs e)
{
    ToolStripMenuItem intrare = (ToolStripMenuItem)sender;
    if (intrare == Rosu)
        Nume.ForeColor = Color.Red;
    if (intrare == Albastru)
        Nume.ForeColor = Color.Blue;
    if (intrare == Verde)
        Nume.ForeColor = Color.Green;
    if (intrare == Iesire)
        Application.Exit();
}

```

Compilați și executați programul.

Desenarea în C#

Desenele în C# sunt obiecte ale clasei *Graphics*. La fel ca și în Java, desenarea se poate face într-o funcție specializată, numită *OnPaint()*. În acest caz, codul care trebuie implementat are forma

```

Protected override void OnPaint(PaintEventArgs e)
{
    Graphics g=e.Graphics;
    // cod de implementat
}

```

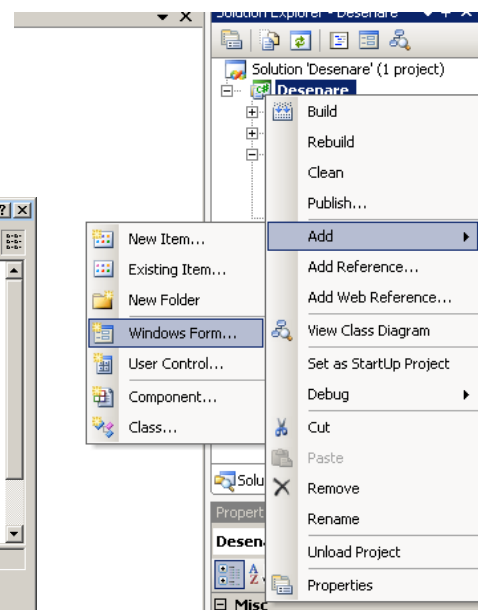
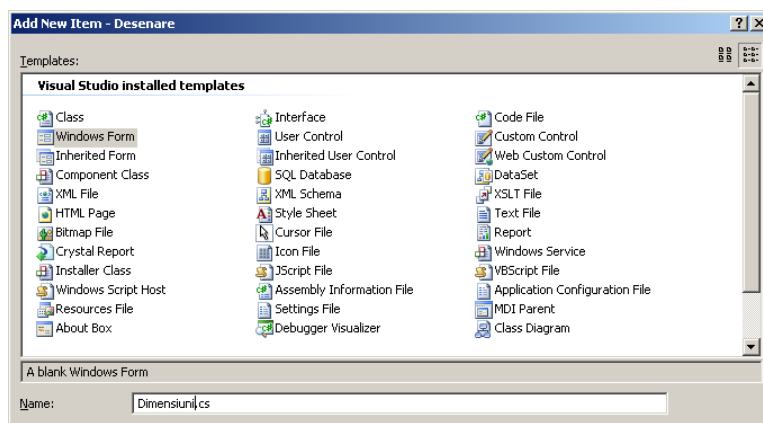
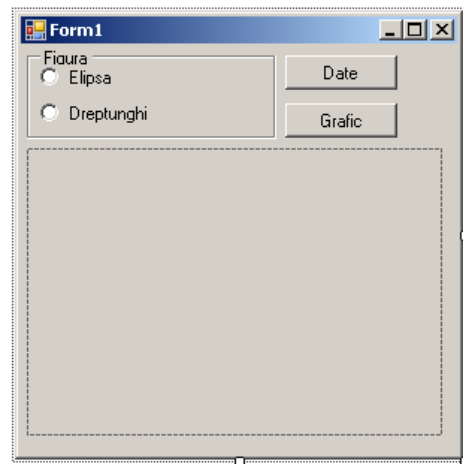
Această funcție va fi apelată de fiecare dată la generarea formei. Dar, spre deosebire de Java, desenarea se poate face și în orice altă funcție, de exemplu în funcția generată la apăsarea unui buton. În acest caz, sintaxa va fi de forma

```

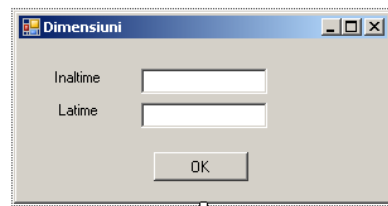
Protected void button1_Click(object sender, EventArgs e)
{
    Graphics g=this.CreateGraphics();
    // cod de implementat
    g.Dispose();
}

```

Pentru realizarea aplicației din Java, vom utiliza această a doua metodă. Să deschidem un nou proiect, numit *Desenare* și să implementăm interfața din figura din dreapta. Am adăugat un control Panel, care va reprezenta zona de desenare, căruia i-am schimbat proprietatea **Name** în *panel* și **BorderStyle** în *FixedSingle*. Am adăugat de asemenea un control GroupBox cu proprietatea **Text** *Figura*, 2 controale RadioButton cu proprietatea **Text** *Elipsa* și *Dreptunghi* și proprietatea **Name** *el*, respectiv *dr* și 2 controale Button, cu proprietățile **Text** și **Name** *Date* și respectiv *Grafic*. Acum va trebui să implementăm macheta de dimensiuni. Pentru aceasta, click dreapta în Solution Explorer pe rădăcină (*Desenare*), **Add** și apoi **Windows Form**. Va apare macheta *Add New Item*, în care vom alege *Windows Form*, iar la **Name** vom tasta *Dimensiuni.cs* și apoi **Add**.



Observați că a apărut o nouă machetă, pe care putem acum construi noua interfață. Aceasta va fi ca în figura din dreapta. Am adăugat 2 controale Label cu proprietatea **Text** *Inaltime* și *Latime*, 2 controale TextBox cu proprietatea **Name** *h* respectiv *l* și un control Button, cu proprietățile **Text** și **Name** **OK**. În plus, pentru ca butonul să se comporte implicit ca un buton de confirmare a acțiunilor din machetă, se setează proprietatea **DialogResult** la **OK**.



Vom implementa acum funcția asociată evenimentului **Click** pe butonul *Date*. Dar înainte, vom adăuga clasei Form1, 2 variabile de tip int, în care vom salva dimensiunile setate în macheta copil.

```
public partial class Form1 : Form
{
    int H, L;
    public Form1()
    {
        InitializeComponent();
    }
}
```

Deosebirea față de Java este ca noua formă a fost implementată într-o clasă separată. Problema care apare este ca, la fel ca și în Java, obiectele *h* și *l* sunt obiecte private ale clasei Dimensiuni. Pentru a le putea accesa, este nevoie să adpugăm clasei 2 metode publice, care să returneze valorile stocate în aceste controale. Vom intra deci în zona de cod a clasei Dimensiuni și vom adăuga metodele

```
public int GetH()
{
    return Convert.ToInt16(h.Text);
}

public int GetL()
{
    return Convert.ToInt16(l.Text);
}
```

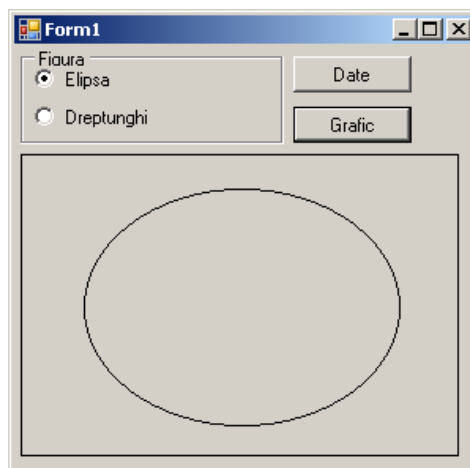
Și acum, dublu click pe butonul *Date* și:

```
private void Date_Click(object sender, EventArgs e)
{
    Dimensiuni dm = new Dimensiuni();
    if (dm.ShowDialog() == DialogResult.OK)
    {
        H = dm.GetH();
        L = dm.GetL();
    }
}
```

Ce face funcția? Crează un nou obiect Dimensiuni, pe care-l afișează cu ajutorul metodei *ShowDialog()*. La apăsarea butonului **OK**, revenirea din forma copil se face cu transmiterea spre forma părinte a codului *DialogResult.OK*. Dacă revenirea se face în acest fel, preluăm dimensiunile înscrise în forma copil. Și acum, desenarea. Dubli click pe butonul *Grafic*, și

```
private void Grafic_Click(object sender, EventArgs e)
{
    Graphics g = panel.CreateGraphics();
    Color fundal=panel.BackColor;
    g.Clear(fundal);
    Pen pen = new Pen(Color.Black);
    int x=panel.Width/2 - L/2;
    int y=panel.Height/2 - H/2;
    Rectangle r = new Rectangle(x, y, L, H);
    if (dr.Checked)
        g.DrawRectangle(pen, r);
    if (el.Checked)
        g.DrawEllipse(pen, r);
    g.Dispose();
}
```

Ce am facut? Întâi am construit un context grafic, g, peste controlul panel. Apoi am salvat în obiectul Color fundal culoarea fundalului controlului panel și am șters imaginea din acest control cu culoarea de fundal. Am construit apoi un obiect Pen de culoare neagră. În C#, linia de desenare este întotdeauna un obiect Pen. După care, am determinat centrul contextului de desenare asociat controlului panel și am desenat elipsa sau dreptunghiul în funcție de dimensiunile preluate din forma copil. Pentru desenarea elipsei și dreptunghiului, s-a folosit un obiect Rectangle, la care obiectele grafice sunt tangente.



4. Proiectarea aplicațiilor complexe

În general, în cazul proiectării unor aplicații complexe, se utilizează un model de proiectare bazat pe mai multe nivele, separate între ele. Un model de proiectare des utilizat este modelul bazat pe 3 straturi (fig. 4.1). Acestea sunt:

- **Stratul prezentare (*Presentation tier*):** este reprezentat de interfața utilizator expusă de program. Uzual, este sub forma unei interfețe grafice, prietenoase pentru utilizator, implementată fie sub forma unor forme Windows, fie sub forma unor interfețe accesibile prin intermediul browserelor web.
- **Stratul logic al aplicației (*Business Logic tier*):** este reprezentat de codul aplicației
- **Stratul de date (*Data tier*):** va stoca datele manipulate de program. Acestea pot fi stocate în baze de date, fișiere xml, fișiere binare, etc. Marea majoritate a aplicațiilor vor implementa clase care gestionează accesul la acest nivel.

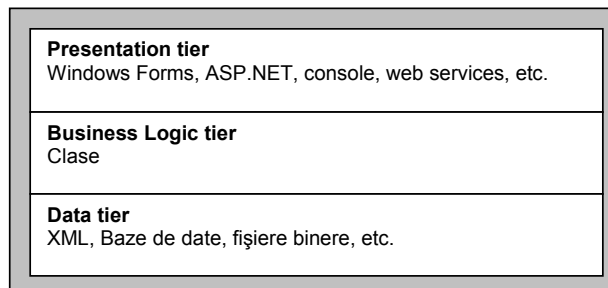


Figura 4.1

Înainte de a se trece la proiectarea propriuzisă a aplicației, este necesar ca proiectantul să aibă o idee clară asupra a ceea ce trebuie să facă aplicația. Bazat pe aceasta, se poate trece la proiectare și implementarea stratului logic. Aici, urmează să fie implementate clasele necesare creării de obiecte capabile să implementeze și să rezolve problema dorită, capabile să furnizeze informația cerută de stratul de prezentare și să manipuleze datele din stratul de date. Modul de concepere al unei astfel de aplicații multistrat, o vom detalia în continuare.

4.1 Punerea problemei

Dorim să efectuăm un sondaj cu privire la cursurile oferite de Școala Academică Postuniversitară. Astfel, cursanții vor fi invitați să voteze cursul preferat, având acces la o aplicație de tip web, cu interfața prezentată în fig. 4.2. Prin votarea unui curs, numărul de voturi asociat acestuia va fi incrementat cu 1.

Primul pas pe care trebuie să-l facem este să încercăm să gândim stratul logic. Pentru aceasta, vom porni de la a evalua entitățile implicate în aplicația noastră. Acestea ar putea fi: site web, vizitator al sitului, sondaj, întrebare, răspuns, vot. În mod evident, legat de stratul logic, doar 4 entități sunt semnificative: **sondaj**, **întrebare**, **răspuns**, **vot**. Situl web ține de stratul prezentare, iar vizitatorul este în fapt utilizatorul aplicației. Întrebarea este dacă stratul logic va trebui să definească câte o clasă pentru fiecare din aceste entități. Să le luăm pe rând:

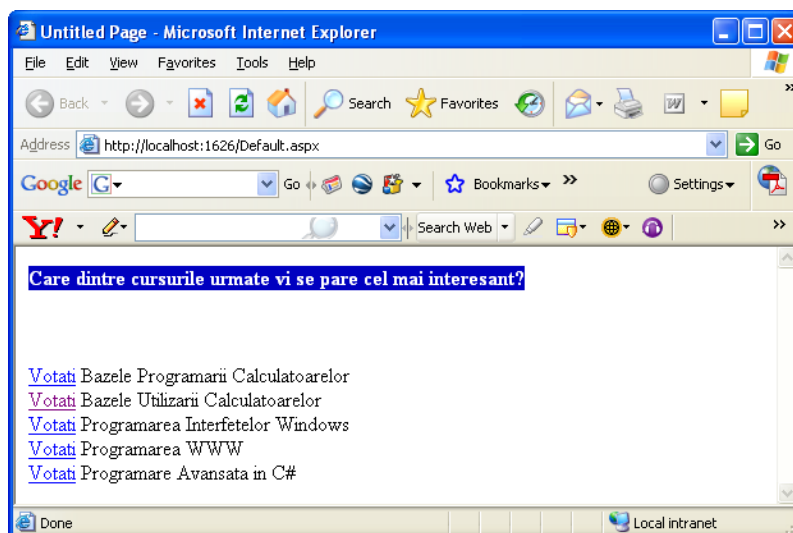


Figura 4.2

Clasa Sondaj

Când ne gândim la un sondaj, trebuie să ne gândim la un set de date asociate. Acestea sunt:

- O întrebare
- Mai multe răspunsuri
- Un contor de voturi pentru fiecare din aceste răspunsuri.

Acestea fiind datele care trebuiesc stocate într-un obiect de tip **Sondaj**, să vedem acum ce funcționalitate trebuie să-i oferim acestui obiect:

- **Votare** – este relativ evident că obiectul va trebui să conțină o metodă `Votare()` care să înregistreze voturile referitoare la răspunsurile din sondaj.
- **Preia întrebarea** – este evident că va trebui să putem avea acces la întrebarea sondajului, în vederea afișării.
- **Preia răspuns** – este de asemenea evident că va trebui să putem citi lista de răspunsuri din interiorul obiectului pentru a le putea afișa și a permite vizitatorilor sitului să voteze unul din răspunsuri
- **Preia voturile** – pentru fiecare răspuns, va trebui să putem vedea câte voturi a primit până în acel moment și pentru a le putea afișa asociat fiecărui răspuns.

În acest moment, avem o idee în mare despre cum va arăta clasa **Sondaj**. Evident, ideea aceasta trebuie transpusă în practică, așa că trebuie s-o mai finisăm puțin.

Clasa Intrebare

În cazul nostru, întrebarea este un simplu șir de caractere, de forma “care dintre cursurile urmate vi se pare cel mai interesant?”. Fiind atât de simplă, în mod evident nu va necesita crearea unei clase. Este suficient ca întrebarea să fie stocată pur și simplu prin intermediul unui câmp string într-o altă clasă. În care? Nu știm încă. Vom vedea în momentul în care vom analiza relațiile dintre clase.

Clasa Raspuns

La fel ca și întrebarea, și răspunsul este un string. Dar, un sondaj va avea o singură întrebare și mai multe răspunsuri! De asemenea, să nu uităm că fiecare

răspuns va avea asociat un număr de voturi. Dacă vom implementa o clasă specială pentru răspuns, vom putea include proprietatea `Text` pentru textul răspunsului, o proprietate care să conțină numărul de voturi asociat și respectiv metoda `Votare()` care să permită votarea răspunsului. Dar, totuși, răspunsul e un simplu string! Oare merită să construim o clasă specială pentru el? Vom decide mai târziu, problema rămâne în suspensie.

Clasa **Votare**

Un vot este în esență un da sau un nu asociat unui răspuns. Voturile în cazul nostru vor fi un șir de numere întregi, asociate fiecărui răspuns în parte. Nu are rost să ne complicăm prin a crea o clasă specială pentru voturi.

Relațiile între clase

Este evident că clasa principală este clasa **Sondaj**. Ce se întâmplă dacă implementăm și o clasă pentru **Răspuns**? Cum va fi de exemplu înregistrat un vot? Ar putea fi înregistrat astfel:

```
raspunsulMeu=sondajulMeu.Raspunsuri[3].Text;  
sondajulMeu.Raspunsuri[3].Votare();
```

Dacă în schimb vom implementa atât răspunsurile cât și voturile ca și colecții în cadrul clasei **Sondaj**, lucrurile se simplifică:

```
raspunsulMeu = sondajulMeu.Raspunsuri[3];  
sondajulMeu.Votare(3);
```

Deoarece dorim ca aplicația noastră să fie cât mai simplă și mai ușor de întreținut, vom alege această ultimă versiune. Deci, pe lângă trăsăturile enunțate mai sus, clasa **Sondaj** va putea conține și un șir de răspunsuri sub forma unui șir de stringuri, precum și un șir de voturi, de aceeași dimensiune cu șirul de răspunsuri. În plus, clasa va trebui să ofere un mecanism de sincronizare a celor 2 șiruri, pentru că în fond fiecare răspuns va avea asociat un număr de voturi pe poziția de același indice. În aceste condiții, putem finisa ideile despre structura stratului aplicație. Vom avea deci o singură clasă, **Sondaj**, care va avea următoarele componente:

- Intrebare – proprietate
- Raspunsuri[] – proprietate (colecție)
- Voturi[] – proprietate (colecție)
- Votare() – metodă

Totul pare ok. Dar mai există o foarte mică problemă. Putem avea mai multe sondaje! În această situație, e normal să știm exact care este sondajul cu care lucrăm. Deci ar fi bine să adăugăm clasei **Sondaj** o metodă statică (de ce statică? Pentru că este o metodă care interesează clasa, nu modelează comportamentul unui obiect), hai s-o numim `Curent()` care să ne returneze sondajul activ. Deci, clasa **Sondaj** va arăta cam așa:

- Intrebare – proprietate
- Raspunsuri[] – proprietate (colecție)
- Voturi[] – proprietate (colecție)
- Votare() – metodă
- Curent() – metodă statică

Ar mai fi o mică modificare de făcut. Nu absolut necesară, dar ne poate ajuta să simplificăm codul. În mod evident, parcurgerea tuturor răspunsurilor se va face într-o secvență de forma

```
Sondaj sondajulMeu=Sondaj.Curent();  
for (int i=1;i<sondajulMeu.Raspunsuri.Length;i++)
```

iar parcurgerea voturilor

```
for (int i=1;i<sondajulMeu.Voturi.Length;i++)
```

Dar cum cele 2 șiruri au același număr de elemente și mai mult, trebuie să fie sincronizate, nu ne împiedică nimic să parcurgem răspunsurile cu secvența a doua și respectiv voturile cu prima secvență. Pentru a evita această confuzie, este util să mai adăugăm clasei o proprietate:

- NumarRaspunsuri – conține numărul de răspunsuri din sondaj.

Și acum, putem gândi prototipurile pentru metode și proprietăți:

Intrebare – este suficient să fie definită ca un șir de caractere:

```
public string Intrebare;
```

Raspunsuri – reprezintă o colecție de răspunsuri, fiecare răspuns în parte fiind un string. Este normal în această situație să definim această proprietate ca un tablou de stringuri:

```
public string[] Raspunsuri;
```

Voturi – reprezintă câte o mărime întreagă, asociată fiecărui răspuns. Va fi implementată în consecință ca un tablou unidimensional de întregi:

```
public int[] Voturi;
```

NumarRaspunsuri – o mărime întreagă, care va menține numărul de răspunsuri din sondaj:

```
public int NumarRaspunsuri;
```

Ca o observație, doar proprietatea *Voturi* poate fi modificată de utilizator, dar și aceasta doar prin intermediul funcției *Votare()*. Deci, pentru aceste 3 proprietăți vom implementa doar clauza *get*, nu și *set*.

Votare() – va trebui să primească parametru un întreg, reprezentând indicele în tabloul de răspunsuri a răspunsului votat. Ea nu va returna nici o valoare ci va incrementa intern votul de același indice din tabloul de voturi.

```
public void Votare(int raspuns);
```

Curent() – metoda nu primește nici un parametru, dar returnează instanța curentă a clasei **Sondaj** (obiectul **Sondaj** activ):

```
public static Sondaj Curent();
```

În acest moment, avem o imagine exactă asupra conținutului stratului logic, urmând doar implementarea lui efectivă.

4.2 Stratul de date

Cum stocăm datele? Să ne reamintim că datele noastre constau într-o întrebare, mai multe răspunsuri și mai multe voturi. Cu alte cuvinte, datele au o structură extrem de simplă, fără a fi nevoie de implementarea unor relații complexe între date. Este deci suficient să stocăm datele în fișiere xml. Fișierul xml va avea următoarea structură:

```
<Sondaj Intrebare="Care dintre cursurile urmate vi se pare cel mai interesant?">
  <Raspuns Text="Bazele Programarii Calculatoarelor" Voturi="10" />
  <Raspuns Text="Bazele Utilizarii Calculatoarelor" Voturi="7" />
  <Raspuns Text="Programarea Interfetelor Windows" Voturi="8" />
  <Raspuns Text="Programarea WWW" Voturi="5" />
  <Raspuns Text="Programare Avansata in C#" Voturi="10" />
  <Raspuns Text="Baze de Date" Voturi="7" />
</Sondaj>
```

Se poate observa că avem un marcaj Sondaj, care încapsulează întregul sondaj. El conține un atribut Intrebare care reprezintă întrebarea sondajului. În interiorul blocului Sondaj, avem mai multe marcaje Raspuns, fiecare având câte 2 attribute: Text, care conține răspunsul și Voturi care conține numărul de voturi asociat. Avantajul stocării datelor în format xml este dat de faptul că există o mulțime de modalități de a le prelua în aplicație și manipula. O astfel de modalitate am văzut-o în capitolul 2. Aici, vom încerca să lucrăm altfel.

4.3 Implementarea stratului logic

Să începem implementarea stratului logic. Vom implementa clasa Sondaj și vom testa funcționalitatea ei. Pentru aceasta, să deschidem un nou proiect de tip Console Application, pe care să-l numim SondajConsola. De ce am ales acest tip de proiect? Pentru că după implementarea clase, va fi foarte ușor într-un program main() să îi testăm funcționalitatea.

Pentru că nu am mai lucrat cu astfel de proiecte, observați ce schelet ne-a oferit infrastructura:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SondajConsola
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Este practic scheletul pentru construirea funcției main(). Să adăugăm acum noua clasă. Cum, ar trebui să știm deja: clic dreapta pe rădăcină în Solution Explorer,

Add, **New Item** și respectiv **Class**. La **Name**: tastați **Sondaj.cs** și apoi apăsați butonul **Add**.

Dacă intrăm în codul clasei, observăm că mediul de programare a realizat următorul schelet:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SondajConsola
{
    class Sondaj
    {
    }
}
```

Este scheletul pe care vom începe să construim noua clasă. Să adăugăm întâi proprietățile. Să ne reamintim că vom adăuga numele interne ale câmpurilor care stochează proprietățile, nu numele sub care ele vor fi expuse în exterior prin intermediul clauzelor **get** și **set**.

```
namespace SondajConsola
{
    class Sondaj
    {
        private string inIntrebare;
        private string[] inRaspunsuri;
        private int[] inVoturi;
        private int inNumarCursuri;
        private string inNumeFisier;
    }
}
```

Observați că pe lângă proprietățile despre care am discutat anterior, am adăugat și o proprietate care va stoca numele fișierului în care se găsește sondajul. Acest lucru va fi necesar deoarece citirea și încărcarea datelor din fișierul xml se va face în interiorul metodelor claselor. De altfel, cu acest mod de abordare v-ați obișnuit în exemplul din capitolul 2.

Apropos de acest fișier, creați un fișier cu numele de exemplu **SondajIASP.xml** și conținutul de la capitolul 4.2 în directorul **Debug** al proiectului. L-am creat în acest director pentru a nu mai fi nevoiți să precizăm calea spre el.

Să începem deci. Este evident că la construcția unui obiect **Sondaj**, conținutul atributului **Intrebare** din fișierul xml va trebui stocat în câmpul **inIntrebare**, iar conținutul atributelor **Text** și respectiv **Voturi** de la marcajele **Raspuns** stocate în șirurile **inRaspunsuri** și **inVoturi**. Pentru aceasta, constructorul clasei va trebui să primească parametru numele fișierului în care este stocată structura xml. Apoi, va trebui să citească această structură și să o depoziteze în structura de date a obiectului.

Există mai multe modalități de a extrage datele dintr-un fișier xml și a le stoca în structura de date ale obiectului. O primă metodă ar fi de a citi nod cu nod conținutul fișierului și de a stoca datele citite în câmpurile corespunzătoare ale obiectului. O altă tehnică am studiat-o în capitolul 2, respectiv completarea câmpurilor obiectului prin deserializarea conținutului fișierului xml. O a treia metodă, cea pe care o vom folosi

aici, este de a citi conținutul fișierului xml într-un obiect DataSet, care apoi poate fi tratat ca o sursă de date pentru completarea structurii obiectului Sondaj.

Obiecte DataSet sunt realizate pentru a permite accesul la date independent de sursa acestora. Un obiect DataSet poate conține o colecție de unul sau mai multe tabele, datele fiind accesibile pe baza intersecției dintre linie și coloană în fiecare tabel.

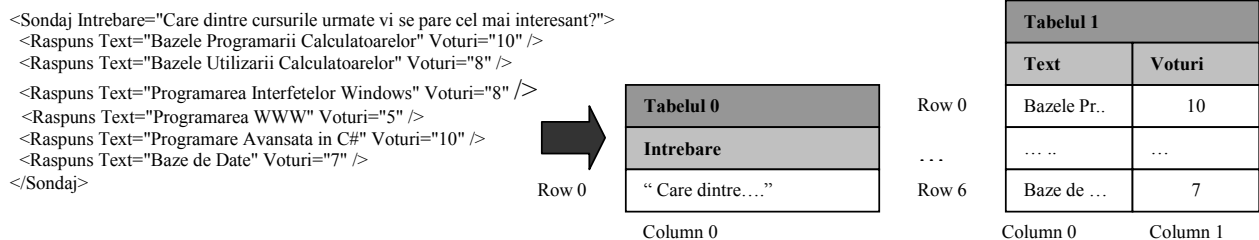


Figura 4.3

În cazul nostru, datele stocate în fișierul xml, vor fi stocate într-un obiect DataSet care conține 2 tabele (fig. 4.3).

Tabelul 0 va stoca datele marcăjului Sondaj și conține o singură linie și o singură coloană, care stochează întrebarea. Tabelul 1 va conține un număr de linii egal cu numărul de marcaje Raspuns, având o coloană pentru atributul Text și o coloană pentru atributul Voturi.

Acum, să începem implementarea constructorului. Acesta va trebui să primească parametru de intrare numele fișierului xml care stochează datele, va extrage aceste date în cele 2 tabele, după care, din aceste tabele va construi obiectul Sondaj. Dar, înainte de asta, trebuie să mai includem câteva namespace-uri, pentru a putea lucra cu fișiere (System.IO), pentru a putea accesa informația xml (System.Xml) și respectiv pentru a putea lucra cu obiecte DataSet (System.Data).

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.IO;
using System.Xml;
...

```

Și acum, constructorul:

```

private Sondaj(string fisier)
{
    inNumFisier = fisier;
    DataSet inDataSet = new DataSet();
    FileStream fsCitescXml = new FileStream(fisier, FileMode.Open);
    XmlTextReader inXmlReader = new XmlTextReader(fsCitescXml);
    inDataSet.ReadXml(inXmlReader);
    inXmlReader.Close();
    inNumarRaspunsuri = inDataSet.Tables[1].Rows.Count;
    inIntrebare = inDataSet.Tables[0].Rows[0].ItemArray[1].ToString();
    inRaspunsuri = new string[inNumarRaspunsuri];
}

```

```

for (int i=0;i<inNumarRaspunsuri;i++)
    inRaspunsuri[i] = inDataSet.Tables[1].Rows[i].ItemArray[0].ToString();
inVoturi=new int[inNumarRaspunsuri];
for (int i=0;i<inNumarRaspunsuri;i++)
    inVoturi[i] = int.Parse(inDataSet.Tables[1].Rows[i].ItemArray[1].ToString());
}

```

Să-l explicăm linie cu linie.

```

private Sondaj(string fisier)
{
    inNumeFisier = fisier;
}

```

Constructorul primește ca parametru numele fișierului xml. Acest nume este salvat în proprietatea internă inNumeFisier. Este puțin ciudat că am definit constructorul private, deoarece știm că acesta trebuie să fie public pentru a se putea construi obiectul. Aici nu există nici o problemă în acest sens, deoarece obiectul va fi construit prin apelarea metodei Curent(), internă clasei, deci constructorul nu trebuie să fie neapărat public. L-am pus private pentru a evita încercarea de construire a unui obiect Sondaj din exterior altfel decât prin apelarea metodei Curent().

```

DataSet inDataSet = new DataSet();
FileStream fsCitescXml = new FileStream(fisier, FileMode.Open);

```

Apoi creăm un obiect DataSet și un flux (stream) de citire asociat fișierului care conține codul xml.

```

XmlTextReader inXmlReader = new XmlTextReader(fsCitescXml);
inDataSet.ReadXml(inXmlReader);
inXmlReader.Close();

```

Declarăm și construim un obiect XmlTextReader asociat fluxului de citire din fișier. Acest obiect permite extragerea informațiilor din format xml și stocarea lor directă în obiectul DataSet prin intermediul metodei ReadXml(). Metoda va crea câte un tabel pentru fiecare marcaj întâlnit, deci în cazul nostru, va crea cele 2 tabele despre care am discutat mai sus.

```

inNumarRaspunsuri = inDataSet.Tables[1].Rows.Count;

```

În DataSet, primul tabel va fi referit ca și Tables[0] iar al doilea ca și Tables[1]. Tables[1].Rows.Count numără liniile din tabel. Cum pe fiecare linie este salvat un ansamblu Text-Vot, astfel putem determina numărul de răspunsuri din sondaj, număr care îl stocăm în proprietatea internă inNumarRaspunsuri.

```

inIntrebare = inDataSet.Tables[0].Rows[0].ItemArray[1].ToString();

```

Extragem acum întrebarea. Va trebui să ne uităm în tabelul 0, pe linia 0 (prima linie) și să referim singurul obiect de pe linia respectivă (ItemArray[1]). Cum ItemArray returnează un obiect generic object, va trebui să-l convertim în string. Rezultatul îl depunem în proprietatea internă inIntrebare.

```

inRaspunsuri = new string[inNumarRaspunsuri];
for (int i=0;i<inNumarRaspunsuri;i++)

```

```
inRaspunsuri[i]= inDataSet.Tables[1].Rows[i].ItemArray[0].ToString();
```

Acum completăm proprietatea internă `InRaspunsuri`. Aceasta este un șir de stringuri de dimensiune egală cu `inNumarRaspunsuri`. Vom crea un șir de stringuri de dimensiune cerută și îi completăm pe rând fiecare intrare. Cum răspunsurile se află în tabelul 1 pe coloana 0 fiecărei linii, ne vom referi la `Tables[1].Rows[i].ItemArray[0]`.

```
inVoturi=new int[inNumarRaspunsuri];
for (int i=0;i<inNumarRaspunsuri;i++)
    inVoturi[i] = int.Parse(inDataSet.Tables[1].Rows[i].ItemArray[1].ToString());
```

Procedăm în același fel cu proprietatea `inVoturi`, cu observația că aceasta este un șir de întregi și o extragem de pe coloana 1 a fiecărei linii. Vom converti întâi obiectul `ItemArray[1]` în string și apoi, prin intermediul `int.Parse()` îl vom converti în valoarea întreagă.

Cu aceasta constructorul este terminat. Urmează implementarea metodei `Curent()`.

```
public static Sondaj Curent()
{
    Sondaj sondajulMeu = new Sondaj("SondajIASP.xml");
    return sondajulMeu;
}
```

Metoda creează un nou obiect **Sondaj** extrăgând datele din fișierul `SondajIASP.xml` și-l returnează. Nu are nevoie de explicații suplimentare.

Clasa va trebui să permită incrementarea numărului de voturi pentru răspunsul selectat de utilizator, prin intermediul metodei `Votare()`.

```
public void Votare(int raspuns)
{
    if (raspuns < 0 || raspuns >= inNumarRaspunsuri)
    {
        throw (new Exception(" Raspuns invalid!"));
    }
    inVoturi[raspuns]++;
    DataSet inDataSet = new DataSet();
    FileStream fsCitescXml = new FileStream(inNumeFisier, FileMode.Open);
    XmlTextReader inXmlReader = new XmlTextReader(fsCitescXml);
    inDataSet.ReadXml(inXmlReader);
    inXmlReader.Close();
    DataRow inVoturiRow = inDataSet.Tables[1].Rows[raspuns];
    inVoturiRow["Voturi"] = (inVoturi[raspuns]).ToString();
    StreamWriter inStream = new StreamWriter(inNumeFisier);
    inDataSet.WriteXml(inStream, XmlWriteMode.IgnoreSchema);
    inStream.Close();
}
```

Să defalcăm și această metodă. În primul rând, ea primește ca parametru indicele din șirul `inRaspunsuri` al răspunsului votat.

```
if (raspuns < 0 || raspuns >= inNumarRaspunsuri)
{
    throw (new Exception(" Raspuns invalid!"));
}
```

```
}
```

Se verifică întâi validitatea acestuia, adică să fie în mulțimea de indici din șir. În caz contrar, se generează o excepție care aruncă șirul “Raspuns invalid”.

```
inVoturi[raspuns]++;
```

Se incrementează numărul de voturi pentru răspunsul ales. Urmează salvarea lui în fișierul xml.

```
DataSet inDataSet = new DataSet();  
FileStream fsCitescXml = new FileStream(inNumeFisier, FileMode.Open);  
XmlTextReader inXmlReader = new XmlTextReader(fsCitescXml);  
inDataSet.ReadXml(inXmlReader);  
inXmlReader.Close();
```

Avem aceeași secvență ca și în constructor, pentru a completa tabele din obiectul DataSet.

```
DataRow inVoturiRow = inDataSet.Tables[1].Rows[raspuns];
```

Va trebui acum să reconstruim linia corespunzătoare răspunsului votat în tabelul 1. Pentru început, vom extrage acea linie într-un obiect DataRow.

```
inVoturiRow["Voturi"] = (inVoturi[raspuns]).ToString();
```

Acum, salvăm în acea linie pe coloana “Voturi” noua valoare convertită la string. Tot ce mai avem de făcut este să salvăm din nou obiectul DataSet în fișierul xml.

```
StreamWriter inStream = new StreamWriter(inNumeFisier);  
inDataSet.WriteXml(inStream, XmlWriteMode.IgnoreSchema);  
inStream.Close();
```

Vom crea un StreamWriter asociat fișierului și vom scrie în acest flux obiectul DataSet. Cum fișierul xml nu conține nici o schemă, vom ignora scrierea acesteia. Cu aceasta, funcția de votare este gata.

Mai avem de implementat clauzele get pentru proprietăți:

```
public int NumarRaspunsuri  
{  
    get  
    {  
        return inNumarRaspunsuri;  
    }  
}  
public string Intrebare  
{  
    get  
    {  
        return inIntrebare;  
    }  
}  
public string[] Raspunsuri  
{
```



```

        get
        {
            return inRaspunsuri;
        }
    }
    public int[] Voturi
    {
        get
        {
            return inVoturi;
        }
    }
}

```

Gata! Clasa este implementată. Acum urmează testarea. Să implementăm un program de test. Pentru aceasta, imediat după `main()` vom insera o funcție `AfiseazaSondaj()`, care nu face altceva decât să afișeze proprietățile `Raspunsuri` și `Voturi` din sondajul pe care-l primește ca parametru:

```

private static void AfiseazaSondaj(Sondaj sSondaj)
{
    Console.WriteLine("=====SONDAJ=====");
    Console.WriteLine(sSondaj.Intrebare);
    for (int i = 0; i < sSondaj.NumarRaspunsuri; i++)
    {
        Console.WriteLine(sSondaj.Raspunsuri[i] + " : ");
        Console.WriteLine(sSondaj.Voturi[i] + " voturi");
    }
    Console.WriteLine("=====");
    Console.ReadLine();
}

```

Și acum să completăm funcția `main()`. Vom testa întâi situația de excepție. Să implementăm codul:

```

static void Main(string[] args)
{
    Sondaj unSondaj = Sondaj.Curent();
    AfiseazaSondaj(unSondaj);
    try
    {
        unSondaj.Votare(9);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.WriteLine("Voturi intregistrare " + unSondaj.Raspunsuri[2]);
    AfiseazaSondaj(unSondaj);
}

```

Programul principal generează un sondaj și-i afișează proprietățile. Încearcă apoi să voteze răspunsul de indice 9, care nu există în șirul de răspunsuri. Rezultatul este prezentat în fig. 4.4.

```

file:///E:/Aplicatii/Documente/Cursuri/C#Av/programe/SondajConsola/SondajConsola/bin/De...
=====SONDAJ=====
Care dintre cursurile urmate vi se pare cel mai interesant?
Bazele Programarii Calculatoarelor : 10 voturi
Bazele Utilizarii Calculatoarelor : 8 voturi
Programarea Interfetelor Windows : 9 voturi
Programarea WWW : 5 voturi
Programare Avansata in C# : 10 voturi
Baze de Date : 7 voturi
=====
Raspuns invalid?
Voturi intregistrare Programarea Interfetelor Windows
=====SONDAJ=====
Care dintre cursurile urmate vi se pare cel mai interesant?
Bazele Programarii Calculatoarelor : 10 voturi
Bazele Utilizarii Calculatoarelor : 8 voturi
Programarea Interfetelor Windows : 9 voturi
Programarea WWW : 5 voturi
Programare Avansata in C# : 10 voturi
Baze de Date : 7 voturi
=====

```

Figura 4.4

Se observă că a fost generată excepția și numărul de voturi nu s-a schimbat. Dacă de exemplu vom vota răspunsul de indice 3 (al patrulea răspuns) vom obține rezultatul din fig. 4.5. Pentru aceasta vom modifica funcția main() ca mai jos:

```

...
try
{
    unSondaj.Votare(3);
}
...

```

```

file:///E:/Aplicatii/Documente/Cursuri/C#Av/programe/SondajConsola/SondajConsola/bin/De...
=====SONDAJ=====
Care dintre cursurile urmate vi se pare cel mai interesant?
Bazele Programarii Calculatoarelor : 10 voturi
Bazele Utilizarii Calculatoarelor : 8 voturi
Programarea Interfetelor Windows : 9 voturi
Programarea WWW : 5 voturi
Programare Avansata in C# : 10 voturi
Baze de Date : 7 voturi
=====
Voturi intregistrare Programarea Interfetelor Windows
=====SONDAJ=====
Care dintre cursurile urmate vi se pare cel mai interesant?
Bazele Programarii Calculatoarelor : 10 voturi
Bazele Utilizarii Calculatoarelor : 8 voturi
Programarea Interfetelor Windows : 9 voturi
Programarea WWW : 6 voturi
Programare Avansata in C# : 10 voturi
Baze de Date : 7 voturi
=====
-

```

Figura 4.5

Se poate observa că în urma votării, Programarea WWW nu mai are 5 voturi, ci 6. Cu aceasta, implementarea stratului logic s-a terminat. Fiind făcută și verificarea, se poate trece la implementarea stratului de prezentare.

4.4 Implementarea stratului prezentare

Să trecem acum la implementarea nivelului prezentare. Dorim ca interfața utilizator să fie accesibilă în 2 pagini web, prima permițând votarea iar a doua să arate

rezultatul votării. Interfața va trebui construită astfel încât să utilizeze stratul de date și stratul logic construit anterior.

Să începem. Să deschidem un nou proiect ASP.NET Web application, pe care să-l numim SondajWeb.

Primul pas pe care îl avem de făcut, este să adăugăm clasa Global.asax, pentru a putea afla și stoca calea spre server (vezi cap. 2). Apoi, modificăm scheletul generat de vrăjitor (wizard) ca mai jos:

```
public class Global : System.Web.HttpApplication
{
    public static String CaleXml;
    protected void Application_Start(object sender, EventArgs e)
    {
        CaleXml = Server.MapPath("");
    }
    ...
}
```

În directorul de pe calea astfel obținută, copiați fișierul SondajIASP.xml creat anterior. Astfel, stratul de date este rezolvat. Urmează stratul logic.

Pentru aceasta, să adăugăm proiectului o clasă, pe care s-o numim Sondaj. Să copiem apoi codul clasei Sondaj din aplicația anterioară, începând cu declararea proprietăților și terminând cu ultima funcție get(). De asemenea, să adăugăm cele 3 clauze uses adăugate și acolo. Singura modificare pe care trebuie s-o facem este să adaptăm calea spre fișierul xml din metoda Curent() în așa fel încât să țină cont de calea de pe server. În rest, să avem încredere, doar clasa e testată!

```
public static Sondaj Curent()
{
    Sondaj sondajulMeu = new Sondaj(Global.CaleXml+"\\SondajIASP.xml");
    return sondajulMeu;
}
```

Cu aceasta și stratul logic este implementat. Urmează construirea stratului prezentare, în așa fel încât să utilizeze straturile anterioare.

Prima dată setăm pagina default.aspx să fie pagină de start. În primul rând, aplicația noastră nu trebuie să ruleze pe server, ci local. Așa că vom șterge marcasele `<form id="form1" runat="server">` și `</form>`. Apoi, construim interfața din fig. 4.6. Pentru prima etichetă setăm proprietatea **ID** la `txtIntrebare`, **BackColor** și **BorderColor** albastru, **ForeColor** alb și pentru **Font** punem **Bold** și **Italic** pe **True**. Pentru a doua etichetă modificăm proprietatea **ID** la `txtRaspuns`. În mod evident, prima etichetă va afișa întrebarea, iar prin cod, vom itera a doua etichetă astfel încât să afișeze răspunsurile. Ar trebui ca pagina noastră să arate ca mai jos:

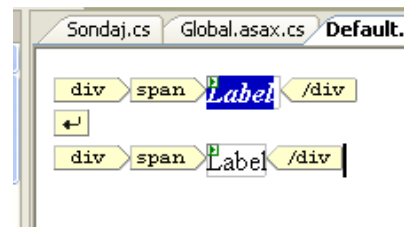


Figura 4.6

```
...
<body>
    <div>
```

```

        <asp:Label ID="txtIntrebare" runat="server" Text="Label" BackColor="#0000C0"
        BorderColor="Navy" Font-Bold="True" Font-Italic="True"
        ForeColor="White"></asp:Label></div>
        <br />
    <div>
        <asp:Label ID="txtRaspunsuri" runat="server" Text="Label"></asp:Label></div>
    </body>
    ...

```

Și acum să completăm funcția Page_Load().

```

protected void Page_Load(object sender, EventArgs e)
{
    Sondaj unSondaj = Sondaj.Curent();
    txtIntrebare.Text = unSondaj.Intrebare;
    string HtmlLink = "<br />";
    for (int i = 0; i < unSondaj.NumarRaspunsuri; i++)
    {
        HtmlLink += "<a href=\"votare.aspx?id=" + i + "\">Votati </a>";
        HtmlLink += "&nbsp;" + unSondaj.Raspunsuri[i] + "<br />";
    }
    txtRaspunsuri.Text = HtmlLink;
}

```

Să defalcăm codul.

```

Sondaj unSondaj = Sondaj.Curent();
txtIntrebare.Text = unSondaj.Intrebare;

```

Am creat un obiect **Sondaj**, pe care-l declarăm ca fiind sondajul curent. Completăm prima etichetă cu proprietatea Intrebare a sondajului.

```

string HtmlLink = "<br />";

```

Acum urmează scriptarea. Declarăm un string pe care-l numim HtmlLink și îl completăm cu marcajul
.

```

for (int i = 0; i < unSondaj.NumarRaspunsuri; i++)
{
    HtmlLink += "<a href=\"votare.aspx?id=" + i + "\">Votati </a>";
    HtmlLink += "&nbsp;" + unSondaj.Raspunsuri[i] + "<br />";
}

```

Apoi, pentru fiecare răspuns din setul de răspunsuri al sondajului, adăugăm stringului HtmlLink o referință spre o pagină numită votare.aspx pe care urmează s-o implementăm, afișând în același timp textul Votați. Observați că am scriptat toate marcasele asociate. Completăm șirul HtmlLink cu un spațiu , textul răspunsului curent și apoi
.

```

txtRaspunsuri.Text = HtmlLink;

```

În final, șirul astfel construit îl atribuim proprietății **Text** a celei de-a doua etichete. Astfel, eticheta va conține în fapt un cod HTML, care va fi interpretat și afișat de către browser. Compilați și executați programul. Ar trebui să obțineți imaginea din fig. 4.7.

Acum, evident, urmează să adăugăm pagina votare.aspx. Vom adăuga deci proiectului un nou item de tip WebForm, pe care îl vom numi votare. Și pentru această pagină, vom șterge marcasele `<form id="form1" runat="server">` și `</form>` și apoi vom construi interfața din fig. 4.8. Pentru controlul Label modificăm proprietățile identic cu primul control Label din pagina default.aspx, singura deosebire fiind proprietatea **ID** pe care o modificăm la txtTitlu. Al doilea control, este un Panel. Pentru acesta modificăm proprietățile astfel: **ID** în panelRezultate; **BackColor** în albastru deschis; **BorderColor** în albastru închis; **BorderStyle** în Solid.

Controlul Panel are facilitatea că permite afișarea în interiorul lui a unor tabele. Să ne reamintim că informația este structurată în fișierul xml asemănător unui

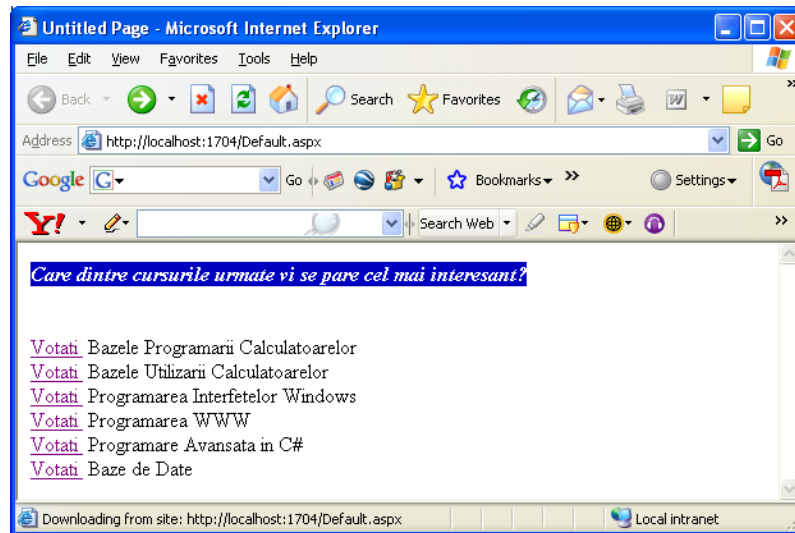


Figura 4.7

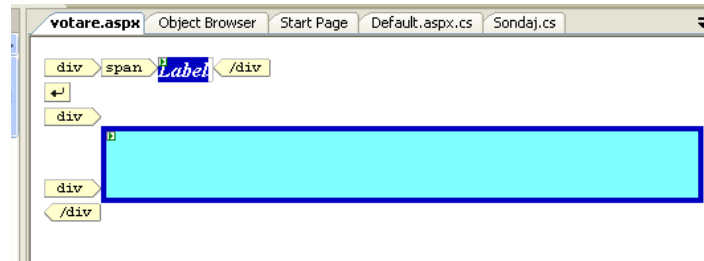


Figura 4.8

tabel.

Vom completa acum funcția Page_Load().

```
protected void Page_Load(object sender, EventArgs e)
{
    Sondaj unSondaj = Sondaj.Curent();
    string parametru = Request.QueryString["id"];
    if (parametru != null)
        unSondaj.Votare(int.Parse(parametru));
    txtTitlu.Text = "Numarul de voturi pentru \"" + unSondaj.Intrebare + "\"";
    Table unTablou = new Table();
    for (int i = 0; i < unSondaj.NumarRaspunsuri; i++)
    {
```

```

        TableRow oLinie = new TableRow();
        unTablou.Controls.Add(oLinie);
        TableCell coloanaRaspunsuri = new TableCell();
        coloanaRaspunsuri.Text = unSondaj.Raspunsuri[i]+" ";
        oLinie.Controls.Add(coloanaRaspunsuri);
        TableCell coloanaVoturi = new TableCell();
        coloanaVoturi.Text = unSondaj.Voturi[i].ToString() + " voturi";
        oLinie.Controls.Add(coloanaVoturi);
    }
    panelRezultate.Controls.Add(unTablou);
}

```

Să explicăm din nou funcția pas cu pas.

```
Sondaj unSondaj = Sondaj.Curent();
```

Evident, primul pas este să creăm un nou obiect **Sondaj**.

```

string parametru = Request.QueryString["id"];
if (parametru != null)
    unSondaj.Votare(int.Parse(parametru));

```

Obiectul Request va permite accesul în interiorul paginii de web, pentru că acesta reține ceea ce browserul trimite. Ca să înțelegem mecanismul, să ne reamintim cum am lansat pagina curentă:

```
"<a href=\"votare.aspx?id=\" + i + \">Votati </a>"
```

Adică, de exemplu dacă am ales al 3-lea răspuns, i=2 și deci browserul va interpreta linia

```
<a href="votare.aspx?id=2">Votati </a>
```

Request.QueryString["id"] va returna valoarea parametruului id, adică 0 pentru primul răspuns, 1 pentru al doilea ș.a.m.d. Cu alte cuvinte, astfel vom identifica ce răspuns a fost votat. Evident, șirul din parametru nu poate fi null, pentru că putem vota doar un răspuns existent, dar nu este rău să ne luăm precauții. De asemenea, el va fi întotdeauna un șir reprezentând o valoare numerică, deci putem ușor să-l transmitem funcției int.Parse() pentru a-l transforma într-un întreg. Valoarea întreagă respectivă va fi apoi furnizată funcției Votare() pentru a incrementa numărul de voturi.

```
txtTitlu.Text = "Numarul de voturi pentru \"" + unSondaj.Intrebare + "\"";
```

Afișăm textul întrebării. Acum urmează să construim tabelul cu răspunsuri-voturi.

```

Table unTablou = new Table();
for (int i = 0; i < unSondaj.NumarRaspunsuri; i++)
{

```

Declarăm un obiect de tip **Table**. Evident, acesta va fi compus din linii și coloane. Vom avea atâtea linii câte răspunsuri există în sondaj, fiecare linie având 2

coloane: Text și Voturi. Deci, în mod evident, va trebui să facem un ciclu după NumarRaspunsuri.

```
TableRow oLinie = new TableRow();  
unTablou.Controls.Add(oLinie);
```

TableRow reprezintă o linie. Vom declara un obiect linie, pe care îl vom adăuga tabelului. Astfel, la fiecare iterație, tabelului îi va fi adăugată o nouă linie.

```
TableCell coloanaRaspunsuri = new TableCell();  
coloanaRaspunsuri.Text = unSondaj.Raspunsuri[i] + " ";  
oLinie.Controls.Add(coloanaRaspunsuri);
```

TableCell reprezintă o celulă într-o linie, fiind echivalentă pentru întregul tablou cu o coloană. Deci, declarăm o celulă de tabel, în care afișăm răspunsul corespunzător liniei pe care ne aflăm. Apoi adăugăm coloană la linie. Evident, va trebui să mai adăugăm o coloană, corespunzătoare numărului de voturi. Acest lucru este realizat de codul de mai jos:

```
TableCell coloanaVoturi = new TableCell();  
coloanaVoturi.Text = unSondaj.Voturi[i].ToString() + " voturi";  
oLinie.Controls.Add(coloanaVoturi);  
}
```

Mai adăugăm o coloană. Fiind a doua adăugată, va reprezenta coloana 2 de pe linie.

```
panelRezultate.Controls.Add(unTablou);
```

În final, adăugăm tabloul astfel construit în panel. Compilați și executați programul. Dacă veți vota un curs, veți obține rezultatul din fig. 4.9.

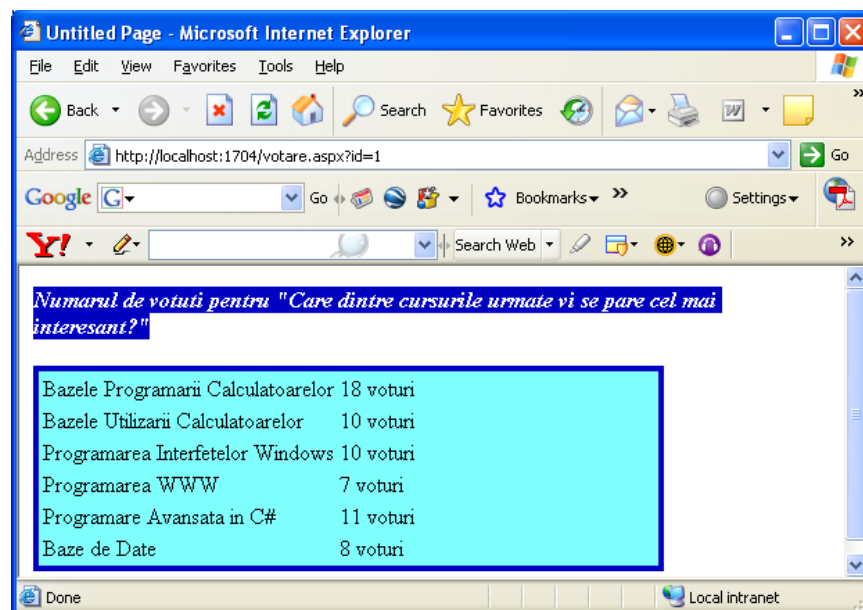


Figura 4.9

4.4.1 Limitarea numărului de voturi

În mod normal, o persoană poate vota o singură dată. Cel mai simplu mod de a limita numărul de voturi este de a asocia paginii de web o mică porțiune de cod, care se numește *cookie*. Să ne reamintim că cookie-ul este o mică bucată de cod, salvată de browser pe hard-discul utilizatorului în momentul în care pagina de web este apelată și de fiecare dată când pagina este reapelată, cookie-ul este retrimis. Dacă de exemplu vom descoperi existența unui cookie pe hard-discul celui care votează, vom ști că el a mai votat o dată și vom putea ignora un nou vot. Dezavantajul utilizării cookie-urilor, este că dacă browserul celui care votează are dezactivate cookie-urile, el va putea vota din nou.

Pentru început, pentru a vedea dacă există cookie-uri pe hard-discul utilizatorului, să facem o mică modificare în program. În codul html al paginii default.aspx, inserăm următorul cod:

```
<%@ Page Language="C#" Trace="true" AutoEventWireup="true" ...
```

Acest cod va permite browserului să afișeze toate informațiile de depanare, inclusiv cookie-urile. Dacă vom compila și executa acum programul, vom obține următoarele informații din fig. 4.10.

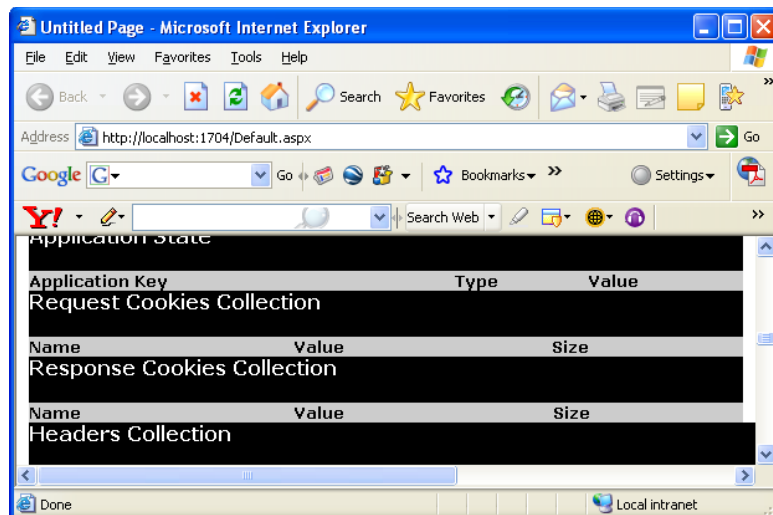


Figura 4.10

După cum se poate vedea, la secțiunile cookie nu avem nimic. Acum urmează să implementăm codul care ne va permite trimiterea unui cookie. Ce avem de făcut? De fiecare dată când o persoană votează, vom salva un cookie îpe hard-discul acesteia. Deci, crearea și trimiterea cookie-ului va trebui făcută în metoda *Votare()*. Primul lucru care trebuie să-l facem este să verificăm dacă metoda a fost apelată din interiorul unei pagini web, sau, în termenii ASP.NET a fost apelată sub un *HttpContext*. Acest fapt va evita testarea cookie-ului dacă metoda este apelată în alt fel, de exemplu dintr-un program windows sau dintr-o aplicație consolă.

În metoda *Votare*, după închiderea streamului de scriere, vom insera codul:

```
public void Votare(int raspuns)
{
```



```

...
inDataSet.WriteXml(inStream, XmlWriteMode.IgnoreSchema);
inStream.Close();
if (HttpContext.Current != null)
{
    HttpCookie unCookie = new HttpCookie("Sondaj");
    unCookie.Value = (inIntrebare);
    HttpContext.Current.Response.Cookies.Add(unCookie);
}
}

```

Să detaliem instrucțiunile:

```
if (HttpContext.Current != null)
```

Verificăm dacă contextul curent Http este null. El va fi diferit de null ori de câte ori metoda este apelată dintr-un browser web, deci dintr-un cod html. Doar în acest caz vom crea cookie-ul.

```
HttpCookie unCookie = new HttpCookie("Sondaj");
unCookie.Value = (inIntrebare);
```

Orice cookie are un nume și un conținut. Prima linie crează un cookie cu numele Sondaj, iar a doua linie înscrie în acest cookie textul întrebării sondajului. Astfel, cookie-ul poate fi ușor identificat în raport cu alte cookie-uri.

```
HttpContext.Current.Response.Cookies.Add(unCookie);
```

În acest fel, vom forța ca atunci când serverul trimite răspunsul la cererea browserului, să adauge acestei vereri și cookie-ul astfel creat.

Compilați și executați programul. Observați apariția cookie-ului (fig. 4.11).

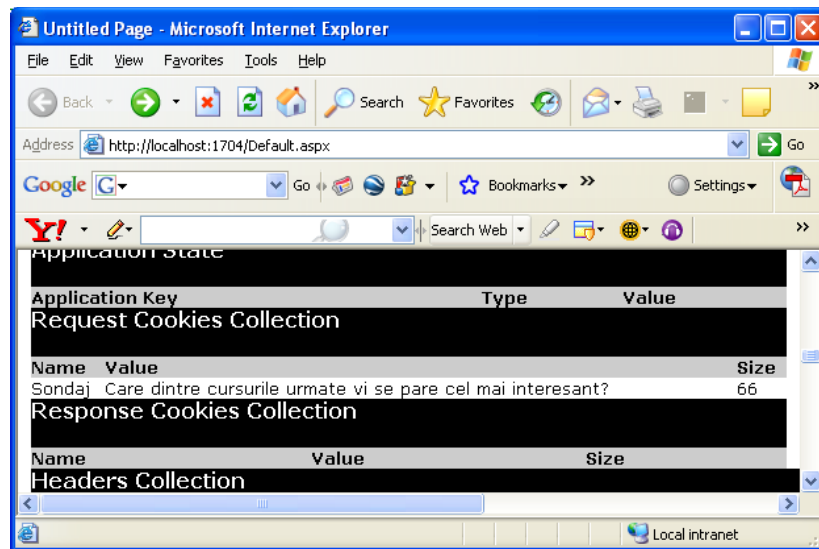


Figura 4.11

Pentru a bloca votarea repetată, trebuie să adăugăm la începutul metodei codul de mai jos:

```
public void Votare(int raspuns)
{
    if (HttpContext.Current.Request.Cookies["Sondaj"] != null)
    {
        if (HttpContext.Current.Request.Cookies["Sondaj"].Value == inIntrebare)
        {
            throw (new Exception("Ati votat deja o data in acest sondaj!"));
        }
    }
    if (raspuns < 0 || raspuns >= inNumarRaspunsuri)
    ....
}
```

Să detaliem.

```
if (HttpContext.Current.Request.Cookies["Sondaj"] != null)
```

Pentru început se caută existența unui cookie cu numele Sondaj. Dacă există ...

```
if (HttpContext.Current.Request.Cookies["Sondaj"].Value == inIntrebare)
```

... se testează dacă are stocat în el întrebarea sondajului. Testarea conținutului cookie-ului se face deoarece pot exista mai multe cookie-uri cu același nume, fie create de aceeași aplicație (nu în cazul nostru) fie create de aplicații diferite. Dacă cookie-ul este găsit, se aruncă o excepție, ceea ce va avea ca efect întreruperea execuției funcției.

Acum va trebui să mai modificăm apelul funcției Page_Load() din pagina votare.aspx astfel încât să intercepteze excepția aruncată:

```
protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        Sondaj unSondaj = Sondaj.Curent();
        ...
        panelRezultate.Controls.Add(unTablou);
    }
    catch (Exception ex)
    {
        txtTitlu.Text = ex.Message;
    }
}
```

4.5 Alt stil: servicii web + aplicație windows

Să deschidem un proiect de tip ASP.NET Web Service Application, pe care să-l numim SondajServiciu. Haideți să-l particularizăm puțin conform aplicației noastre. În primul rând, apăsați click dreapta pe Service1.asmx, Rename și schimbați numele în ServSondaj.asmx. Observați că declarația clasei s-a schimbat în

```
public class ServSondaj : System.Web.Services.WebService
```

Nu e suficient și din păcate următoare modificare nu o putem face din interiorul mediului de programare. Mergeți în directorul SondajServiciu/SondajServiciu și deschideți într-un editor fișierul ServSondaj.aspx. Modificați linia de cod astfel încât numele serviciului să nu fie Service1 ci ServSondaj:

```
<%@ WebService Language="C#" CodeBehind="ServSondaj.aspx.cs"
Class="SondajServiciu.ServSondaj" %>
```

Cu aceasta, serviciul oferit se va numi ServSondaj. Haideți să facem acum aceleași modificări ca și în cazul anterior: Creăm clasa **Sondaj** și copiem codul din aplicația precedentă. Creăm apoi clasa Global.aspx cu exact același cod ca și în aplicația precedentă. Acum, tot ce mai avem de făcut este să implementăm metodele web expuse de clasa ServSondaj. Ștergem metoda HelloWorld() și inserăm codul:

```
[WebMethod(Description = "Returneaza intrebarea in sondaj")]
public string Intrebare()
{
    Sondaj unSondaj = Sondaj.Curent();
    return unSondaj.Intrebare;
}
[WebMethod(Description = "Returneaza rezultatul sondajului")]
public int[] Voturi()
{
    Sondaj unSondaj = Sondaj.Curent();
    return unSondaj.Voturi;
}
[WebMethod(Description = "Returneaza raspunsurile sondajului")]
public string[] Raspunsuri()
{
    Sondaj unSondaj = Sondaj.Curent();
    return unSondaj.Raspunsuri;
}
[WebMethod(Description = "Permite votarea")]
public void Votare(int raspuns)
{
    Sondaj unSondaj = Sondaj.Curent();
    unSondaj.Votare(raspuns);
}
```

Tot ce mai avem de făcut este să copiem fișierul SondaIASP.xml în directorul serverului (ServSondaj/ServSondaj). Compilați și executați programul(fig. 4.12). Testați metodele expuse de serviciu.

Acum să trecem la implementarea programului client. Să deschidem un nou proiect de tip Windows Form Application, pe care să-l numim SondajWindows. Să-I proiectăm o interfață care conține 3 controale **Label** ca în fig. 4.12. Să modificăm proprietățile acestora ca mai jos: eticheta de sus: **Name** în txtIntrebare, **BackColor** în albastru, **ForeColor** în alb, **Font** cu **Bold** și **Italic** în true. Eticheta din stânga jos: **Text** în Numar voturi:, iar eticheta din dreapta jos **Name** în txtNrVoturi si **Text** în "

Să intrăm acum în zona de cod. Primele linii sunt deja

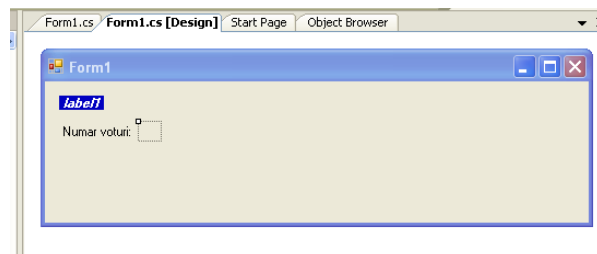


Figura 4.12

cunoscute pentru noi. Va trebui să apelăm serviciul web, să completăm textul etichetei txtIntrebare și să ne creăm și completăm șirul de răspunsuri. Ce este interesant, este că datorită faptului că nu știm în principiu câte răspunsuri avem în test, vom crea și afișa în mod dinamic, prin program, controalele care vor afișa răspunsurile și voturile asociate. Deci să începem:

```
public Form1()
{
    InitializeComponent();
    localhost.ServSondaj rezSondaj = new localhost.ServSondaj();
    txtIntrebare.Text = " Incarc datele.....";
    string[] raspSondaj = rezSondaj.Raspunsuri();
    int nrRaspunsuri = rezSondaj.Raspunsuri().Length;
}
```

Deci știm și numărul de răspunsuri. Este timpul să adăugăm formei controale pentru afișarea răspunsurilor, voturilor și care să ne permită votarea. Răspunsurile și voturile le vom afișa în controale **Label**, vom adăuga de asemenea controale **ProgressBar** pentru a reprezenta numărul de răspunsuri în procente și controale **Button** pentru votare. Primul lucru care trebuie făcut este să declarăm șiruri de astfel de controale, vizibile în toate metodele clasei:

```
public partial class Form1 : Form
{
    private Label[] controaleRaspuns;
    private Label[] controaleVot;
    private ProgressBar[] controaleEvolutie;
    private Button[] controaleVotare;

    public Form1()
    ...
}
```

După cum se observă, am declarat 4 șiruri de controale. După cum ne sugerează și numele, în primul șir vom afișa răspunsurile, în al doilea voturile, în al treilea evoluția voturilor iar în al patrulea butoanele care ne vor permite votarea. Să începem acum să le construim. Construim și afișăm întâi șirul de etichete care va afișa răspunsurile:

```
public Form1()
{
    InitializeComponent();
    ...
    int nrRaspunsuri = rezSondaj.Raspunsuri().Length;
    controaleRaspuns = new Label[nrRaspunsuri];
    for (int i=0;i<nrRaspunsuri;i++)
    {
        Label txtRaspuns=new Label();
        txtRaspuns.Location=new System.Drawing.Point(10, 60+(20*i));
        txtRaspuns.Size = new System.Drawing.Size(200, 16);
        txtRaspuns.BackColor = Color.Azure;
        controaleRaspuns[i] = txtRaspuns;
        Controls.Add(txtRaspuns);
    }
}
```

Cum am făcut? Întâi am construit efectiv șirul de controale, de dimensiunea nrRaspunsuri. Apoi, pentru fiecare element (control) din șir am făcut următoarele: am construit un nou control **Label**, numit txtRaspuns. Controlul este construit (proprietatea **Location**) cu colțul stânga sus în punctul de coordonate $x=10$, $y=60+(20*i)$. Deci, poziția pe x va fi fixă, iar poziția pe y va evolua în jos pentru fiecare control cu câte 20 de pixeli. Coordonatele sunt relative la colțul stânga sus a formei. Controlul are lungimea de 200 de pixeli și înălțimea de 16 pixeli (proprietatea **Size**) și culoarea Azure (proprietatea **Color**). După construire, controlul este atribuit poziției curente din șirul de etichete. Controls.Add() adaugă efectiv pe formă controlul.

De o manieră similară, adăugăm și șirul de etichete care va stoca răspunsul:

```
public Form1()
{
    ...
    Controls.Add(txtRaspuns);
}
controaleVot = new Label[nrRaspunsuri];
for (int i = 0; i < nrRaspunsuri; i++)
{
    Label txtVot = new Label();
    txtVot.Location = new System.Drawing.Point(220, 60 + (20 * i));
    txtVot.Size = new System.Drawing.Size(80, 16);
    txtVot.BackColor = Color.Azure;
    controaleVot[i] = txtVot;
    Controls.Add(txtVot);
}
}
```

Secvența de instrucțiuni nu necesită explicații suplimentare, fiind similară secvenței anterioare. Acum să adăugăm șirul de controale **ProgressBar**:

```
public Form1()
{
    ...
    Controls.Add(txtVot);
}
controaleEvolutie = new ProgressBar[nrRaspunsuri];
for (int i = 0; i < nrRaspunsuri; i++)
{
    ProgressBar prVot = new ProgressBar();
    prVot.Location = new System.Drawing.Point(310, 60 + (20 * i));
    prVot.Minimum = 0;
    prVot.Size = new System.Drawing.Size(103, 16);
    controaleEvolutie[i] = prVot;
    Controls.Add(prVot);
}
}
```

Singura instrucțiune care necesită explicații este prVot.Minimum=0. Pentru un **ProgressBar**, proprietatea **Minimum** reprezintă valoarea inițială de la care acesta începe să evolueze, iar proprietatea Maximum valoarea finală până la care controlul poate evolua. Deci, controlul nostru va începe să evolueze de la valoarea 0.

```
public Form1()
```

```

{
    ...
    Controls.Add(prVot);
}
controaleVotare = new Button[nrRaspunsuri];
for (int i = 0; i < nrRaspunsuri; i++)
{
    Button btnVot = new Button();
    btnVot.Location = new System.Drawing.Point(450, 60 + (20 * i));
    btnVot.Size = new System.Drawing.Size(60, 20);
    btnVot.Text = "Votati";
    btnVot.Name = i.ToString();
    controaleVotare[i] = btnVot;
    Controls.Add(btnVot);
}
}

```

Observăm că am atribuit proprietății **Text** pentru fiecare buton valoarea “Votați” și proprietății **Name** chiar valoarea contorului. De ce am atribuit o valoare pentru **Name**? Pentru că aceste butoane vor trebui apăsate, iar în funcția care răspunde la apăsarea butoanelor va trebui să identificăm care dintre butoane a lansat-o în execuție (evident, vom avea o singură funcție!). Prin intermediul numelui, vom ști ce buton a fost apăsător și respectiv pentru ce răspuns să incrementăm numărul de voturi.

Compilați și executați programul. Observați apariția controalelor pe interfață (fig. 4.13)

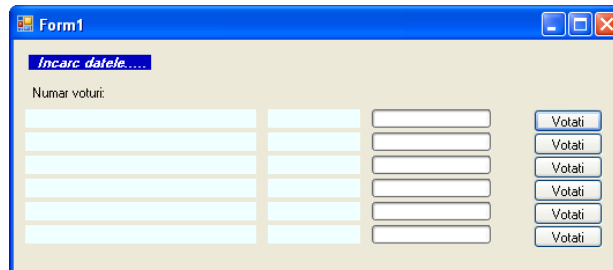


Figura 4.13

Și acum să trecem la afișarea informațiilor. Ce dorim să facem. Datorită faptului că fișierul cu sondajul se găsește pe server, este posibil ca mai mulți utilizatori să voteze într-o perioadă scurtă de timp. De aceea, ar fi corect ca afișarea datelor să se facă periodic, astfel încât să se ia în considerare modificările făcute de alți utilizatori. Avem posibilitatea de a lansa în execuție o funcție periodică, prin declararea unui **Timer**. Timerul va fi setat să contorizeze un interval de timp și, prin intermediul unui handler va lansa în execuție o funcție specificată la scurgerea intervalului de timp respectiv. Pentru a putea declara și utiliza un timer, va trebui să adăugăm proiectului un namespace corespunzător:

```

using System;
...
using System.Windows.Forms;
using System.Timers;

```

Și acum să implementăm și să configurăm timerul.

```

public Form1()
{
    ...
    Controls.Add(btnVot);
}
System.Timers.Timer unTimer = new System.Timers.Timer();
unTimer.Elapsed += new ElapsedEventHandler(CitesteSondaj);
unTimer.Interval = 5000;
unTimer.Enabled = true;
}

```

Să explicăm mai în detaliu codul.

```
System.Timers.Timer unTimer = new System.Timers.Timer();
```

Am creat un timer, numit unTimer.

```
unTimer.Elapsed += new ElapsedEventHandler(CitesteSondaj);
```

Am asociat timerului nou creat un handler pentru evenimentul **Elapsed**. Acest eveniment se produce în momentul în care timpul asociat timerului expiră. La producerea acestui eveniment, va fi lansată în execuție o funcție, numită CitesteSondaj().

```
unTimer.Interval = 5000;
unTimer.Enabled = true;
```

Mărcăm timpul măsura de timer ca fiind 5000 de milisecunde (5 secunde) și pornim timerul (oprirea forțată a timerului se face cu unTimer.Enabled=false).

Și acum, să implementăm funcția CitesteSondaj() care va popula controalele:

```

public void CitesteSondaj(object sursa, ElapsedEventArgs e)
{
    localhost.ServSondaj rezSondaj = new localhost.ServSondaj();
    string[] rezRaspunsuri = rezSondaj.Raspunsuri();
    int nrRaspunsuri = rezRaspunsuri.Length;
    txtIntrebare.Text = rezSondaj.Intrebare();
    int totalVoturi = 0;
    int maximumVot = 0;
    int[] rezVoturi = rezSondaj.Voturi();
    for (int i = 0; i < nrRaspunsuri; i++)
    {
        totalVoturi = totalVoturi + rezVoturi[i];
        if (rezVoturi[i] > maximumVot)
            maximumVot = rezVoturi[i];
    }
    int procentVot;
    for (int i=0;i<nrRaspunsuri;i++)
    {
        controaleRaspuns[i].Text=rezRaspunsuri[i];
        float parteVot=((float)rezVoturi[i]/(float) totalVoturi);
        procentVot=(int)(100*parteVot);
        controaleVot[i].Text=rezVoturi[i].ToString()+" ";
        controaleVot[i].Text+="("+procentVot.ToString()+"%)";
    }
}

```

```

        controaleEvolutie[i].Maximum=maximumVot;
        controaleEvolutie[i].Value=rezVoturi[i];
    }
    txtNrVoturi.Text=totalVoturi.ToString();
}

```

Să detaliem funcția. În primul rând observați antetul. Este o funcție care răspunde la producerea unui eveniment, deci are antetul cunoscut deja.

```

localhost.ServSondaj rezSondaj = new localhost.ServSondaj();
string[] rezRaspunsuri = rezSondaj.Raspunsuri();
int nrRaspunsuri = rezRaspunsuri.Length;
txtIntrebare.Text = rezSondaj.Intrebare();

```

Creăm sondajul, completăm șirul de răspunsuri, aflăm numărul de răspunsuri și afișăm întrebarea. Până aici, nimic nou.

```

int totalVoturi = 0;
int maximumVot = 0;

```

Declarăm 2 variabile întregi, inițializate la 0. Prima variabilă va stoca numărul total de voturi exprimate, iar a doua numărul cel mai mare de voturi exprimate pentru un răspuns.

```

int[] rezVoturi = rezSondaj.Voturi();

```

Extragem în șirul rezVoturi voturile din sondaj, pentru a le putea utiliza în program.

```

for (int i = 0; i < nrRaspunsuri; i++)
{
    totalVoturi = totalVoturi + rezVoturi[i];
    if (rezVoturi[i] > maximumVot)
        maximumVot = rezVoturi[i];
}

```

Parcurgem tot șirul de voturi, pentru a calcula numărul total de voturi. La fiecare iterație, comparăm valoarea maximă a votului cu valoarea curentă. Dacă întrebarea curentă are un număr de voturi mai mare decât vechea valoare maximă, actualizăm valoarea maximă de voturi pentru o întrebare.

```

int procentVot;
for (int i=0;i<nrRaspunsuri;i++)
{
    controaleRaspuns[i].Text=rezRaspunsuri[i];
}

```

Declarăm o variabilă înțregă, în care vom calcula procentual numărul voturilor pentru fiecare răspuns Din numărul total de voturi. Apoi, atribuim proprietății **Text** pentru șirul controaleRaspuns răspunsurile. Astfel, în primul șir de etichete vor fi afișate răspunsurile.

```

float parteVot=((float)rezVoturi[i]/(float) totalVoturi);
procentVot=(int)(100*parteVot);

```


Pentru fiecare răspuns, calculăm proporția reprezentată de numărul propriu de voturi din numărul total de voturi în valoare absolută și respectiv procentual.

```
controaleVot[i].Text=rezVoturi[i].ToString()+" ";
controaleVot[i].Text+="("+procentVot.ToString()+"%)";
```

În etichetele controaleVot afișăm numărul absolut de voturi pentru fiecare răspuns, respectiv procentul reprezentat de acesta din numărul total de voturi.

```
controaleEvolutie[i].Maximum=maximumVot;
controaleEvolutie[i].Value=rezVoturi[i];
```

Marcăm valoarea maximă afișată de ProgressBar-uri la numărul maxim de voturi asociate unui răspuns și setăm cursorul controlului la valoarea reprezentată de numărul de voturi pentru răspunsul curent.

```
}
txtNrVoturi.Text=totalVoturi.ToString();
```

În final, afișăm numărul total de voturi. Compilați și executați programul. Deși nu avem erori de compilare, observăm că debuggerul nu ne permite execuția programului, generând o eroare de execuție. Acest lucru se întâmplă doar în Visual Studio 2005, în versiunile anterioare programul rulează corect. Această versiune nu permite accesul la controalele definite într-o clasă dintr-o funcție care implementează un alt fir de execuție, considerându-l acces nesigur. Nu explicăm aici ce înseamnă fir de execuție și nici metoda corectă de implementare a accesului. Putem rezolva simplu problema în cazul nostru, dezactivând această protecție impusă de Visual Studio 2005, prin adăugarea liniei de cod:

```
public Form1()
{
    InitializeComponent();
    Form1.CheckForIllegalCrossThreadCalls = false;
    ...
}
```

Compilați acum și executați programul. Veți obține interfața din fig. 4.15.

Curs	Numar voturi	Procent	Progress Bar	Buton
Bazele Programarii Calculatoarelor	16	22%	[Progress Bar]	Votati
Bazele Utilizarii Calculatoarelor	11	15%	[Progress Bar]	Votati
Programarea Interfetelor Windows	14	19%	[Progress Bar]	Votati
Programarea WWW	11	15%	[Progress Bar]	Votati
Programare Avansata in C#	11	15%	[Progress Bar]	Votati
Baze de Date	9	12%	[Progress Bar]	Votati

Figura 4.15

Acum urmează să implementăm funcția de votare. Nu putem crea funcția care răspunde la apăsarea butoanelor prin dublu click asupra lor, pentru simplu motiv că ele nu există, fiind create dinamic de program. Va trebui deci să adăugăm handlerul prin cod:

```

public partial class Form1 : Form
{
    ...
    Controls.Add(btnVot);
    controaleVotare[i].Click += new EventHandler(Btn_Click);
}
...

```

Cu alte cuvinte, la apăsarea oricărui buton, va fi lansată în execuție funcția Btn_Click(). S-o implementăm.

```

private void Btn_Click(object sender, EventArgs e)
{
    Button b = (Button)sender;
    localhost.ServSondaj rezSondaj = new localhost.ServSondaj();
    rezSondaj.Votare(int.Parse(b.Name));
    // De aici, CitesteSondaje
    string[] rezRaspunsuri = rezSondaj.Raspunsuri();
    int nrRaspunsuri = rezRaspunsuri.Length;
    txtIntrebare.Text = rezSondaj.Intrebare();
    int totalVoturi = 0;
    int maximumVot = 0;
    int[] rezVoturi = rezSondaj.Voturi();
    for (int i = 0; i < nrRaspunsuri; i++)
    {
        totalVoturi = totalVoturi + rezVoturi[i];
        if (rezVoturi[i] > maximumVot)
            maximumVot = rezVoturi[i];
    }
    int procentVot;
    for (int i = 0; i < nrRaspunsuri; i++)
    {
        controaleRaspuns[i].Text = rezRaspunsuri[i];
        float parteVot = ((float)rezVoturi[i] / (float)totalVoturi);
        procentVot = (int)(100 * parteVot);
        controaleVot[i].Text = rezVoturi[i].ToString() + " ";
        controaleVot[i].Text += "(" + procentVot.ToString() + "%)";
        controaleEvolutie[i].Maximum = maximumVot;
        controaleEvolutie[i].Value = rezVoturi[i];
    }
    txtNrVoturi.Text = totalVoturi.ToString();
}

```

Funcția trebuie explicată doar până la comentariul De aici CitesteSondaje. Din acel punct este identică cu funcția CitesteSondaje(), pentru că după o votare trebuie reafășată situația curentă în sondaj. Deci, să începem:

```

Button b = (Button)sender;
localhost.ServSondaj rezSondaj = new localhost.ServSondaj();

```

Declarăm un obiect **Button**, pe care îl vom încărca cu butonul al cărui apăsare a lansat în execuție funcția. Creăm un nou sondaj.

```

rezSondaj.Votare(int.Parse(b.Name));

```

Apelăm funcția `Votare()` pentru răspunsul de același indice cu numele butonului care a fost apăsat. Astfel, efectuăm efectiv votarea.

Compilați și executați programul. Observați evoluția voturilor. Cu aceasta, aplicația este terminată.

1. Suportul pentru directoare și fișiere

La fel ca și limbajul C (revezi cursul de C), în C# salvarea și restaurarea datelor din fișiere implică noțiunea de *stream*. Un stream este o reprezentare abstractă a unui dispozitiv serial. Prin utilizarea acestui mecanism, programatorul nu este nevoit să cunoască structura hardware a echipamentului care stochează datele, accesul fiind făcut la nivel logic, prin intermediul fișierelor. Scrierea și citirea datelor din fișiere se face în mod secvențial, putând fi citiți și scriși fie octeți, fie caractere.

Citirea datelor se face prin intermediul unui stream de intrare. Cel mai des utilizat stream de intrare este tastatura. Dar mecanismul poate fi aplicat și altor echipamente periferice, sau fișierelor obișnuite.

Scrierea datelor se face prin intermediul unui stream de ieșire. Să ne reamintim ca streamul standard de ieșire este videoterminalul, dar pot fi definite și alte echipamente ca streamuri de ieșire, sau pur și simpli fișiere.

1.1 Clase pentru operații intrare-ieșire

Toate clasele implicate în transferul și stocarea datelor în fișiere sunt definite în spațiul de nume **System.IO**. Pentru a putea utiliza aceste clase, va trebui să specificăm la începutul fișierului sursă în care se declară clasele de lucru utilizarea acestui spațiu (dacă nu există) prin directiva

using System.IO;

Există mai multe clase conținute în acest spațiu de nume, clase care ne permit să manipulăm atât fișiere cât și directoare. Câteva din aceste clase sunt:

- **File** – este o clasă care expune un set de metode statice (să ne reamintim, *o metodă statică este apelată prin intermediul clasei și nu a unui obiect din clasa respectivă, fiind un atribut al clasei*) pentru crearea, deschiderea, copierea, mutarea sau ștergerea fișierelor.
- **Directory** - este o clasă care expune un set de metode statice pentru crearea, deschiderea, copierea, mutarea sau ștergerea directoarelor.
- **Path** – este o clasă care expune metode de manipulare a căilor de directoare.
- **FileInfo** – este o clasă a cărei obiecte reprezintă fișiere fizice pe disc și expune metode de manipulare a acestor fișiere. Pentru operații de scriere-citire asupra acestor fișiere va trebui creat un obiect **Stream**.
- **DirectoryInfo** - este o clasă a cărei obiecte reprezintă directoare fizice pe disc și expune metode de manipulare a acestor directoare.
- **FileStream** – un obiect al acestei clase reprezintă un fișier care poate fi scris, citit sau scris și citit.
- **StreamReader** – obiectele acestei clase sunt utilizate pentru citirea unui stream dintr-un fișier de tip **FileStream**.
- **StreamWriter** - obiectele acestei clase sunt utilizate pentru scrierea unui stream într-un fișier de tip **FileStream**

Aparent, clasele **File** și **FileInfo**, respectiv **Directory** și **DirectoryInfo** expun metode similare. Poate apare întrebarea care din clase este mai bine să fie folosite.

În realitate metodele nu sunt chiar identice, iar pentru cele care sunt, alegerea rămâne la latitudinea programatorului.

1.2 Clasele File și Directory

Așa cum am arătat mai sus, sunt clase care expun un set de metode statice pentru manipularea fișierelor și directorilor. Prin intermediul lor, fișierele și directorii pot fi manipulate fără a crea instanțe ale acestor clase. Câteva metode mai des utilizate ar fi:

File

- **public static FileStream Create (String path)** – crează un fișier cu numele specificat în variabila path. Returnează un obiect **FileStream**, care poate fi utilizat pentru scrierea-citirea datelor la nivelul fișierului.
- **public static FileStream Open (String path, FileMode mode)** – deschide un fișier specificat de calea path în modul descris de mode. Returnează obiectul **FileStream** pentru manipularea datelor. Enumerarea FileMode specifică modul de deschidere al fișierului. Membrii acestei enumerări sunt (fără a mai preciza semnificația lor, fiind foarte semănători cu ce ați învățat în cursul de C): Append, Create, CreateNew, Open, OpenOrCreate, Truncate.
- **public static void Delete (string path)** – șterge fișierul cu numele specificat de variabila path.
- **public static void Copy (String sourceFileName, String destFileName)** – copiază fișierul cu numele sourceFileName în fișierul cu numele destFileName. Ca o observație, acesta din urmă nu trebuie să existe, pentru că la copiere este creat automat.
- **public static void Move (String sourceFileName, String destFileName)** – mută fișierul sursă în destinație.
- **public static bool Exists (string path)** - returnează true dacă directorul specificat există.

Directory

- **public static DirectoryInfo CreateDirectory (String path)** – crează directorul cu numele specificat de variabila path. Returnează un obiect **DirectoryInfo** care reprezintă chiar acel director. Despre clasa **DirectoryInfo** vom vorbi mai jos.
- **public static void Delete (String path)** – șterge directorul cu numele specificat de variabila path. Acest director trebuie să fie gol în momentul ștergerii.
- **public static string GetCurrentDirectory ()** – returnează calea spre directorul curent.
- **public static void SetCurrentDirectory (String path)** – setează directorul curent la calea path.
- **public static String[] GetDirectories (String path)** – returnează un șir de nume de directoare, reprezentând subdirectoarele directorului cu numele conținut în variabila path.
- **public static string[] GetFiles (String path)** – returnează numele fișierelor.

- **public static bool Exists (string path)** – returnează true dacă directorul specificat există.
- **public static void Move (String sourceDirName,String destDirName)** – mută directorul sourceDirName la calea destDirName.

Toate metodele în caz de eroare aruncă un set de excepții. Le vom detalia în momentul implementării unui exemplu.

1.3 Clasele FileInfo și DirectoryInfo

Spre deosebire de clasele anterioare, aceste clase nu expun metode statice. Metodele expuse de acestea pot fi utilizate doar prin intermediul unor obiecte ale claselor.

Un obiect **FileInfo** reprezintă un fișier fizic pe disc. El nu este un stream! Pentru manipularea datelor acestuia, va trebui creat și asociat un obiect **FileStream**. Implementarea acestui mecanism o vom descrie mai târziu.

Construcția unui obiect **FileInfo** se face prin specificarea pentru constructorul clasei a căii și numelui fișierului asociat. În exemplul de mai jos, se creează fișierul "C:/postuniv.exemplu.txt".

```
FileInfo fisier=new FileInfo("C:/postuniv.exemplu.txt");
```

Pe lângă metodele expuse, în mare parte asemănătoare cu metodele clasei **File**, clasa **FileInfo** expune și un set de proprietăți utile pentru manipularea fișierelor. Câteva din acestea sunt:

- **Attributes** – conține atributele fișierului asociat.
- **CreationTime** – conține data și ora creării fișierului asociat.
- **DirectoryName** – numele directorului în care se găsește fișierul.
- **Exists** – true dacă fișierul cu numele specificat există.
- **FullName** – numele complet (cale+nume) pentru fișier.
- **Length** – lungimea fișierului.
- **Name** – numele (fără cale) fișierului.

Clasa **DirectoryInfo** expune în mare parte aceleași metode și proprietăți ca și clasa **FileInfo**.

1.4 Obiecte FileStream

Clasa **FileStream** implementează obiecte care reprezintă fișiere pe disc. Este utilizată în general pentru a manipula adte de tip octet. Pentru manipularea datelor de tip caracter, este mai simplu să se utilizeze alte 2 clase, respectiv **StreamReader** și **StreamWriter**.

Există mai multe căi prin care se poate crea un obiect FileStream. Cea mai simplă metdă este de a preciza constructorului clasei numele fișierului asociat și modul de deschider a fișierului:

```
FileStream fs=new FileStream("fis.txt", FileMode.OpenOrCreate);
```

În acest caz, obiectului `fs` îi este asociat fișierul "fis.txt", care este deschis dacă există, respectiv creat dacă nu există. Enumerarea `FileMode` oferă mai multe moduri de deschidere a unui fișier:

- **Append** – deschide sau creează un fișier, fără să șteargă datele existente. Datele nou introduse se adaugă la sfârșitul fișierului. Fișierul trebuie să fie deschis în scriere.
- **Create** – creează un nou fișier gol. Dacă el deja există, datele conținute se pierd.
- **CreateNew** – creează un nou fișier. Dacă un fișier cu numele specificat există, se generează o excepție.
- **Open** – deschide un fișier existent. Dacă acesta nu există, se generează o excepție.
- **OpenOrCreate** – deschide un fișier existent, sau, dacă nu există, îl creează.

Elementele `FileMode` se pot utiliza în conjuncție cu elemente din enumerarea `FileAccess`, care precizează modul de acces la datele fișierului. Acestea sunt:

- **Read** – fișierul este deschis în citire.
- **Write** – fișierul este deschis în scriere.
- **ReadWrite** – fișierul este deschis în scriere și citire.

Deci, o altă formă de creare a unui obiect **FileStream** este

```
FileStream fs=new FileStream("fis.txt",FileMode.Open,
FileAccess.Write);
```

De altfel, și clasele **File** și **FileInfo** expun metode de deschidere a unui fișier: `OpenRead()`, respectiv `OpenWrite()`. Deci, construcția unui obiect **FileStream** poate fi făcută și în unul din modurile de mai jos:

```
FileStream fs=File.OpenRead("fis.txt");
```

sau

```
FileInfo fi=new FileInfo("fis.txt");
FileStream fs=fi.OpenRead();
```

1.4.1 Poziționarea într-un fișier

Metoda clasei `FileStream` care permite poziționarea în interiorul unui fișier este

```
public long Seek(long offset, SeekOrigin origin)
```

Parametrul `offset` specifică numărul de octeți peste care se face saltul, iar parametrul `origin` specifică originea saltului. Enumerarea `SeekOrigin` are următoarele elemente:

- **Begin** – saltul se face cu `offset` octeți, numărați de la începutul fișierului.

- **Current** - saltul se face cu offset octeți, numărați de la poziția curentă în fișier..
- **End** - saltul se înapoi face cu offset octeți, numărați de la sfârșitul fișierului.

Din momentul apelului metodei, noua poziție devine poziție curentă în fișier. Saltul în afara fișierului generează excepție.

Cîteva exemple de utilizare ar fi:

```
FileStream fi=File.OpenRead("fis.dat");
fi.Seek(12,SeekOrigin.Begin);
```

Poziția curentă în fișier este octetul al 12-lea, numărat de la începutul fișierului.

```
fi.Seek(10,SeekOrigin.Current);
```

Se face un salt de 10 octeți față de poziția curentă.

```
fi.Seek(-8,SeekOrigin.End);
```

Poziția curentă în fișier este cu 5 octeți înainte de sfârșitul fișierului.

Evident, orice operație de citire-scriere ulterioară se va face de la noua poziție curentă în fișier.

1.4.2 Citirea și scrierea datelor din fișier

Clasa **FileStream** furnizează și metode de scriere-citire. Le vom prezenta, dar nu vom intra în amănunte, deoarece marea majoritate a operațiilor intrare-ieșire se realizează cu ajutorul claselor **StreamReader** și **StreamWriter**.

Citirea octeților dintr-un fișier se face cu metoda

```
public int Read(byte[] array, int offset, int count)
```

Primul paramtru este numele unui șir de octeți în care se vor stoca octeții citați din fișier. Al doilea paramtru specifică poziția în șir începînd cu care se vor depune octeții, iar al treilea câți octeți vor fi citați. În caz de eroare, metoda generează excepție. De exemplu secvența:

```
byte[] data=new byte[100];
try
{
    FileStream fi=new FileStream("fis.dat",FileMode.Open);
    fi.Seek(20,SeekOrigin.Begin);
    fi.Read(data,40,10);
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
```


va citi 10 octeți din fișier, începând cu poziția 20 și îi va stoca în șirul `data` începând cu indexul 40.

Similar, scrierea de octeți în fișier se face cu ajutorul metodei
`public int Write(byte[] array, int offset, int count)`

Exemplul

```
byte[] data=new byte[100];
for (int i=0;i<30;i++)
    data[i]=i;
try
{
    FileStream fi=new FileStream("fis.dat",FileMode.OpenOrCreate);
    fi.Seek(0,SeekOrigin.Begin);
    fi.Write(data,0,30);
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

va scrie primii 30 de octeți din șirul `data` în fișier, începând cu poziția 0.

1.5 Clasele **StreamReader** și **StreamWriter**

Atunci când avem de transferat caracter, clasele **StreamReader** și **StreamWriter** oferă o modalitate mult mai elegantă de lucru. Clasa **StreamReader** permite citirea șirurilor de caracter dintr-un fișier, iar clasa **StreamWriter** scrierea șirurilor de caractere. Metodele acestor clase permit manipularea de obiecte **String**, cu toate eventajele care decurg din aceasta.

Modalitatea de construcție a obiectelor **StreamReader** și **StreamWriter** este foarte simplă: fie se construiește un obiect **FileStream** și noul obiect **StreamReader** sau **StreamWriter** se asociază acestuia, sau pur și simplu obiectele sunt construite prin furnizarea către constructor a numelui fișierului. Secvența

```
FileStream fi=new FileStream("fis.txt", FileMode.Create);
StreamWriter sw=new StreamWriter(fi);
sw.WriteLine(" Salut!");
sw.Write("Ptem inchide");
sw.Close();
```

va crea un obiect **StreamWriter** asociat fișierului "fis.txt" nou creat și va scrie cele 2 texte, în primul caz trecând pe linie nouă după scrierea textului. O secvență similară, poate fi

```
StreamWriter sw=new StreamWriter("fis.txt",false);
sw.WriteLine(" Salut!");
sw.Write("Ptem inchide");
sw.Close();
```

Efectul este același, dar constructorului clasei **StreamWriter** i se transmite direct numele fișierului și true dacă ca datele să fie adăugate în fișier, respectiv false dacă fișierul se creează gol. Deci

```
StreamWriter sw=new StreamWriter("fis.txt",false);
```

este echivalent cu

```
FileStream fi=new FileStream("fis.txt", FileMode.Create);  
StreamWriter sw=new StreamWriter(fi);
```

respectiv

```
StreamWriter sw=new StreamWriter("fis.txt",true);
```

cu

```
FileStream fi=new FileStream("fis.txt", FileMode.Append);  
StreamWriter sw=new StreamWriter(fi);
```

Obiectele **StreamReader** funcționează de o manieră similară:

```
String sir;  
FileStream fi=new FileStream("fis.txt", FileMode.Open);  
StreamReader sr=new StreamReader(fi);  
sir=sr.ReadLine();  
sr.Close();
```

va citi o linie din fișierul "fis.txt" și o va salva în stringul sir.

```
String sir, temp;  
StreamReader sr=new StreamReader("fis.txt");  
temp=sr.ReadLine();  
while ( temp!=null)  
{  
    sir+=temp;  
    temp=sr.ReadLine();  
}  
sr.Close();
```

În secvența de mai sus, se poate observa modalitatea de construcție a obiectului **StreamReader** prin transmiterea direct către constructor a numelui fișierului. Secvența citește toate liniile din fișier (când se ajunge la sfârșitul fișierului ReadLine() returnează null) și le salvează în șirul sir.

Să construim un exemplu. Pentru aceasta să creăm un nou proiect de tip **WindowsApplication**, pe care haideți să-l numim "unu". Să construim apoi machete din fig. 1.1. Prin intermediul acestei aplicații, vom realiza citirea, editarea și salvarea conținutului fișierelor cu extensia ".txt". Macheta conține un control TextBox și 3 controale Button. Pentru controlul TextBox, să modificăm proprietatea **Name** la eFis proprietatea **Multiline** la True (permițând astfel inserarea mai multor linii în control) și proprietatea **ScrollBars** la Both (am adăugat astfel în caz de necesitate bare de

scroll orizontală și verticală). De asemenea, pentru controalele Button, să modificăm proprietățile **Name** și **Text** în felul următor (de la stânga la dreapta): open respectiv Open, save respectiv Save și exit respectiv Exit.

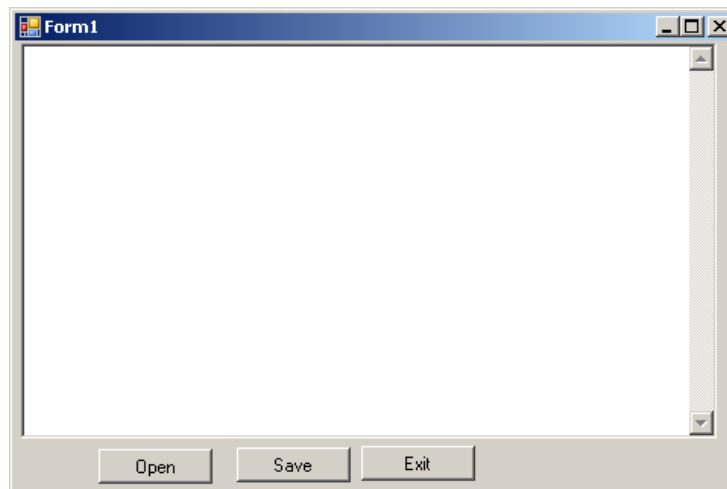


Figura 1.1

Evident, apăsarea butonului Open va permite alegerea și deschiderea unui fișier, iar butonul Save salvarea acestuia. Dorim să implementăm mecanismul de deschidere-salvare standard în sistemul de operare Windows. Pentru aceasta, infrastructura ne pune la dispoziție 2 clase, **OpenFileDialog** și respective **SaveFileDialog**, care permit manipularea fișierelor prin intermediul unor controale de dialog.

1.5.1 OpenFileDialog

Infrastructura C# oferă casete de dialog pentru manipularea numelor de fișiere. O astfel de casetă este OpenFileDialog (fig. 1.2).

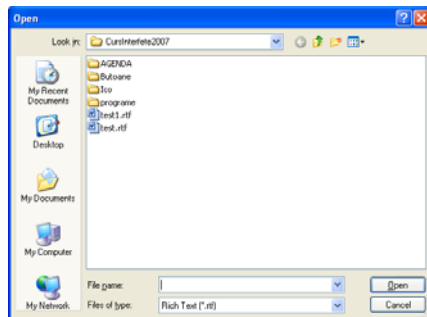


Figura 1.2

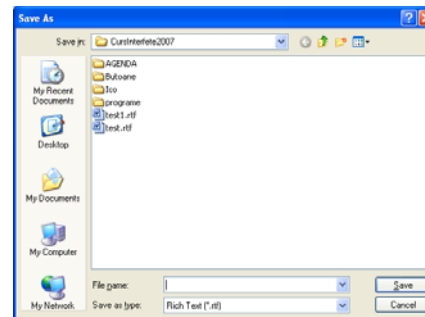


Figura 1.3

În această casetă se poate căuta numele unui fișier existent sau adăuga un nou nume. De asemenea, se permite specificarea tipului de fișier care urmează să fie încărcat. Pentru specificarea tipurilor de fișiere acceptate, trebuie construit un filtru, după următorul mecanism:

"mesaj₁ | tip₁ fișier | mesaj₂ | tip₂ fisier | ... | mesaj_n | tip_n fișier ;

Acest filtru trebuie aplicat casetei de dialog după creare cu instrucțiunea `new`, dar înainte de afișare cu funcția `ShowDialog()`. Permite utilizarea așa numitelor wildcards (semnele `?`, respectiv `*`).

Caseta de dialog oferă o serie de proprietăți utile pentru utilizator. Câteva din acestea sunt:

- **Title** – permite modificarea textului afișat în bara de titlu a casetei de dialog;
- **FileName** – numele fișierului (fișierelor) selectate;
- **InitialDirectory** – numele directorului în care începe căutarea

1.5.2 SaveFileDialog

În figura 1.3 este prezentată caseta de dialog `SaveFileDialog`. Aceasta permite salvarea cu un nume ales sau dat a unui fișier. Modul de lucru cu această casetă de dialog este absolut similar casetei `OpenFileDialog`.

Deoarece fișierele sunt entități externe programului, încercarea de accesare a unui fișier cu nume eronat poate duce la erori în cadrul programului, respectiv la oprirea lui forțată de către sistemul de operare. De aceea, înainte de a implementa funcția care se execută la apăsarea butonului `Open`, mai trebuie să vorbim despre

1.5.3 Excepții

Complexitatea aplicațiilor actuale, în special a celor orientate pe obiecte, presupune o modularizare pe nivele ierarhice. Această situație impune tratarea deosebit de riguroasă a situațiilor deosebite, în special a erorilor care pot să apară la nivelul diferitelor componente ale unui proiect.

În cele ce urmează, vom înțelege prin excepție o situație deosebită care poate să apară pe parcursul execuției unei componente soft parte a unei aplicații. Erorile pot fi privite ca și un caz particular de excepții, dar în general nu toate excepțiile sunt erori.

Excepțiile se împart în două categorii: excepții *sincrone* și excepții *asincrone*. Excepțiile sincrone sunt excepții a căror apariție poate fi prevăzută de programator (de exemplu, un fișier care se dorește a fi prelucrat nu se află în directorul așteptat). Excepțiile asincrone sunt excepții a căror apariție nu poate fi prevăzută în momentul implementării programului (de exemplu, defectarea accidentală a hard-disk-ului). Mecanismul de tratare a excepțiilor ia în considerare numai situația excepțiilor sincrone.

Problema se pune astfel: în faza de programare, se poate presupune că un modul de program va detecta în anumite situații o excepție, dar nu va putea oferi o soluție generală de tratare a acesteia, în timp ce un alt modul al aplicației poate oferi o soluție de tratare a excepției, dar nu poate detecta singur apariția excepțiilor.

Exemple tipice de excepții: împărțire cu 0, accesul la un fișier inexistent, operații cu o unitate logică nefuncțională, etc.

Uzual, la apariția unei excepții, sistemul de operare va opri forțat execuția modului de program în care excepția s-a produs, pentru ca acesta să nu afecteze buna funcționare a celorlalte programe. De multe ori însă este necesar ca excepția apărută să nu fie transmisă sistemului de operare, ci să fie *“prinsă”* și tratată în interiorul modului care a generat-o. Acest lucru este posibil prin intermediul unui mecanism bazat pe 2 instrucțiuni: `try` și `catch`.

Mecanismul este următorul: zona critică, adică zona de cod în care programatorul prevede posibilitatea apariției unei excepții este scrisă într-un bloc try. Blocul try este urmat de unul sau mai multe blocuri catch, câte unul pentru fiecare tip de excepție ce poate să apară. Execuția programelor în acest caz este prezentată în fig. 1.4 a și b.

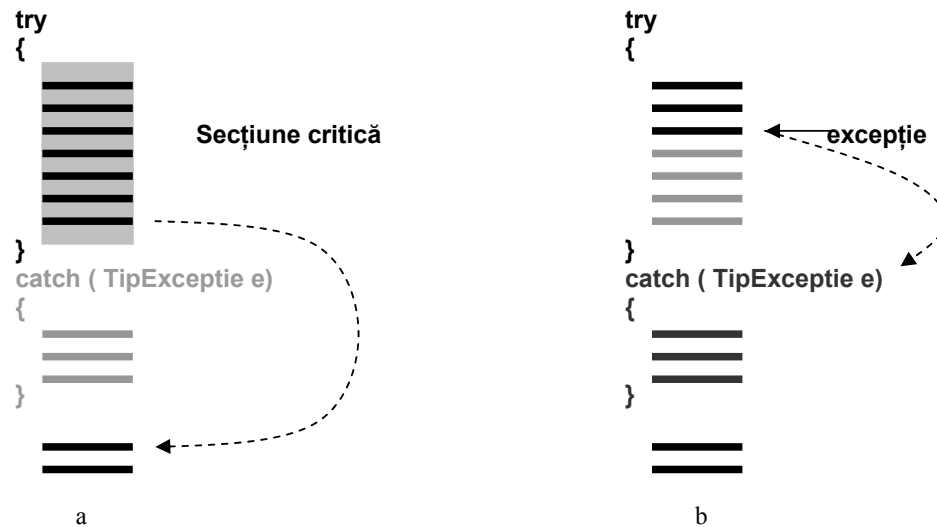


Figura 1.4

Fig. 1.4a prezintă situația normală de execuție, când în secțiunea critică nu apare excepție. În acest caz, se execută în succesiune normală toate instrucțiunile din secțiunea critică, după care, următoarea instrucțiune executată este prima de după blocul catch, acesta fiind neglijat.

Fig. 1.4b prezintă situația apariției unei excepții. În acest caz, următoarea uninstrucțiune executată după cea care a generat excepția este prima din blocul catch, în acesta urmând a fi tratată excepția. După execuția instrucțiunilor din blocul catch, se continuă în secvență normală, cu prima instrucțiune de după acesta.

Marea majoritate a claselor care lucrează cu entități externe memoriei generează excepții, pentru ca posibilele erori să poată fi tratate încă din faza de programare. Este și cazul claselor care lucrează cu fișiere și este indicat ca toate operațiile făcute asupra fișierelor să fie tratate prin mecanismul try-catch.

Și acum să trecem la implementare. Vom apăsa dublu-click pe butonul Open și vom implementa codul:

```
private void open_Click(object sender, EventArgs e)
{
    eFis.Clear();
    OpenFileDialog op = new OpenFileDialog();
    op.InitialDirectory = Directory.GetCurrentDirectory();
    op.Filter = "txt files (*.txt) | *.txt | All files (*.*) | *.*";
    if (op.ShowDialog() == DialogResult.OK)
    {
        String strLin;
        try
        {
            StreamReader sr = new StreamReader(op.FileName);
            strLin = sr.ReadLine();
            while (strLin != null)
            {
```

```

        eFis.Text += strLin + "\r\n";
        strLin = sr.ReadLine();
    }
    sr.Close();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
}
}

```

Conținutul funcției este foarte ușor de înțeles dacă ținem cont de cele prezentate anterior. Se șterge conținutul controlului TextBox și se crează un obiect OpenFileDialog, pentru care se definește filtrul și directorul inițial de căutare ca fiind directorul curent. Apoi se afișează controlul. La revenire din caseta OpenFileDialog, dacă s-a revenit apăsând butonul OK, se creează un StreamReader asociat fișierului ales în caseta OpenFileDialog (prin transmiterea către StreamReader a numelui acestuia) și conținutul fișierului este citit linie cu linie. Fiecare linie citită este adăugată proprietății **Text** a controlului TextBox, după care se adaugă șuril CR-LF.

De o manieră similară se implementează și funcția executată la apăsarea butonului Save.

```

private void save_Click(object sender, EventArgs e)
{
    SaveFileDialog sv = new SaveFileDialog();
    sv.InitialDirectory = Directory.GetCurrentDirectory();
    sv.Filter = "txt files (*.txt) | *.txt | All files (*.*) | *.*";
    if (sv.ShowDialog() == DialogResult.OK)
    {
        try
        {
            StreamWriter sr = new StreamWriter(sv.FileName, false);
            sr.WriteLine(eFis.Text);
            sr.Close();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
}

```

Această funcție nu mai necesită explicații. Și în fine, ultima implementare,

```

private void exit_Click(object sender, EventArgs e)
{
    Application.Exit();
}

```

Compilați și executați programul.

1.6 Serializarea datelor

Până acum am văzut cum se pot salva și restaura în și din fișiere datele de tip octet sau caracter. Dar ce e de făcut, dacă avem de salvat și restaurat date mai complexe, de exemplu obiecte ale unei clase? Evident, obiectele se pot salva sub formă de șiruri de octeți, dar va fi foarte greu la restaurare să împărțim octeții astfel încât să recompunem obiectele salvate. Din fericire, majoritatea limbajelor de nivel înalt, inclusiv C#, implementează un mecanism simplu și fiabil de salvare și restaurare a obiectelor prin intermediul fișierelor. Acest mecanism poartă denumirea de *serializare*.

Orice clasă ale cărei obiecte pot fi salvate sau restaurate prin intermediul fișierelor, va trebui fie să fie declarată serializabilă, fie să implementeze o interfață specifică, numită **ISerializable**. În momentul în care infrastructura primește o cerere de serializare a unor obiecte, va verifica întâi dacă clasa din care fac parte obiectele implementează interfața **ISerializable** și dacă nu, va verifica dacă a fost declarată ca fiind serializabilă. Dacă nici una din condiții nu este îndeplinită, obiectele nu vor putea fi serializate.

Declararea unei clase ca fiind serializabilă este foarte simplă. Pur și simplu, clasei îi este specificat atributul **[Serializable]**. Să lămurim printr-un exemplu. Să creăm un nou proiect, fie unu-unu numele acestuia și proiectului să-i adăugăm o nouă clasă, numită **Rezervare**: click dreapta pe rădăcina unu-unu în Solution explorer, Add, Class..., și la Name scriem Rezervare. Să adăugăm attributele clasei, un constructor și s-o declarăm serializabilă.

```
namespace unu_unu
{
    [Serializable]
    class Rezervare
    {
        public Rezervare()
        {
        }
        public String Nume;
        public DateTime DataS;
        public DateTime DataP;
        public int nrCam;
    }
}
```

Am adăugat clasei un constructor, 4 attribute de tip String, DateTime și int. Obiectele clasei pot reprezenta de exemplu rezervări la un hotel, pentru care se specifică numele clientului, data sosirii, data plecării și numărul de camere rezervate. Prin adăugarea atributului **[Serializable]**, clasa a fost declarată automat serializabilă, deci instanțele ei vor putea fi salvate și restaurate din fișiere.

Să implementăm acum interfața de lucru. Vom implementa interfața din

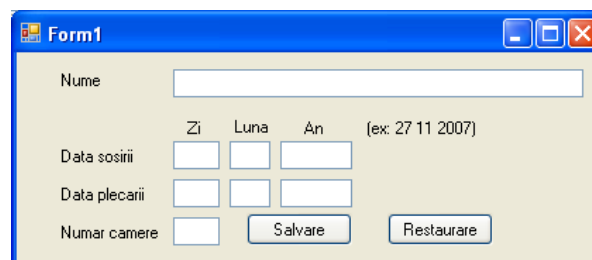


Figura 1.5

figura 1.5, pentru care controalele TextBox au următoarele attribute Name (de sus în jos și de la stânga la dreapta): nume, zis, ls, ans, zip, lp, anp, respectiv nrc, iar butoanele (de la stânga la dreapta) salvare și restaurare.

Transformarea obiectelor în șiruri de octeți pentru a fi executat transferul la nivelul fișierului, se face utilizând *formatori* (Formatters). Există 2 tipuri de formatori: BinaryFormatter și SoapFormatter. Pe primul îl vom folosi aici, pentru a formata obiectele în și din șiruri de octeți, iar pe al doilea îl vom întâlni în capitolele viitoare. Pentru a putea utiliza aceste formatoare, va trebui să adăugăm fișierului care implementează forma spațiul de nume System.Runtime.Serialization.Formatters.Binary, iar pentru a putea implementa mecanismul efectiv de serializare, spațiul de nume System.Runtime.Serialization. Să ne reamintim de asemenea, că pentru a putea lucra cu fișiere, va trebui să adăugăm și spațiul de nume System.IO:

```
using System;
using System.Collections.Generic;
...
using System.Windows.Forms;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
```

```
namespace unu_unu
{
    public partial class Form1 : Form
    {
        ...
    }
}
```

Vom adăuga clasei **Form1** un obiect **Rezervare**, care să fie vizibil în toate metodele clasei:

```
public partial class Form1 : Form
{
    Rezervare rez = new Rezervare();
    public Form1()
    {
        ...
    }
}
```

Și acum să implementăm metodele de serializare și deserializare a obiectelor **Rezervare**. Vom adăuga clasei o metodă care realizează serializarea, numită Salvare().

```
public void Salvare()
{
    Stream s = File.Open("rezervare.rez", FileMode.Create);
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(s, rez);
    s.Close();
    MessageBox.Show("Salvat");
    this.Close();
}
```


Cum lucrează metoda? Întâi crează un Stream asociat fișierului “rezervare.rez” pe care-l deschide în mod creare. Crează apoi un nou obiect de tip BinaryFormatter, prin intermediul căruia apelează metoda

```
public void Serialize (Stream serializationStream, Object graph)
```

unde serializationStream este stream-ul în care se serializează obiectul graph. Cu alte cuvinte, în cazul nostru, obiectul rez de clasă **Rezervare** va fi salva în streamul s, adică în fișierul “rezervare.rez”. Gata. E simplu, formatorul s-a ocupat singur de serializarea obiectului.

Să vedem cum arată restaurarea. Vom implementa metoda Refacere():

```
public void Refacere()
{
    Stream s = File.Open("rezervare.rez", FileMode.Open);
    BinaryFormatter bf = new BinaryFormatter();
    Object obj = bf.Deserialize(s);
    rez = obj as Rezervare;
}
```

Simplu din nou. Pentru deserializare am folosit metoda (care cred că nu mai necesită explicații suplimentare)

```
public Object Deserialize (Stream serializationStream)
```

Deoarece metoda salvează obiectul serializat într-un obiect generic de clasă **Object**, pe acesta l-am particularizat ulterior la clasa **Rezervare**. Evident, metoda se poate rescrie cu ajutorul conversiei explicite de tip (cast):

```
public void Refacere()
{
    Stream s = File.Open("rezervare.rez", FileMode.Open);
    BinaryFormatter bf = new BinaryFormatter();
    rez = (Rezervare)bf.Deserialize(s);
}
```

Ambele modalități de scriere sunt echivalente. Acum tot ce ne mai rămâne de făcut e să implementăm funcțiile care răspund la apăsarea butoanelor:

```
private void salvare_Click(object sender, EventArgs e)
{
    rez.Nume = nume.Text;
    rez.DataS = Convert.ToDateTime(zis.Text + "." + ls.Text + "." +
                                   ans.Text);
    rez.DataP = Convert.ToDateTime(zip.Text + "." + lp.Text + "." +
                                   anp.Text);
    rez.nrCam = Convert.ToInt16(nrc.Text);
    Salvare();
}
```

Nu am făcut altceva decât să completăm attributele obiectului rez cu datele din controalele TextBox și să convertim mărimile asociate la tipul DateTime. După care am apelat funcția de serializare.

```

private void restaurare_Click(object sender, EventArgs e)
{
    Refacere();
    if (rez != null)
    {
        nume.Text = rez.Nume;
        String x;
        int i1, i2;
        x = Convert.ToString(rez.DataS);
        i1 = x.IndexOf('.');
        i2 = x.LastIndexOf('.');
        zis.Text = x.Substring(0, i1);
        ls.Text = x.Substring(i1 + 1, i2-i1-1);
        ans.Text = x.Substring(i2+1, 4);
        x = Convert.ToString(rez.DataP);
        i1 = x.IndexOf('.');
        i2 = x.LastIndexOf('.');
        zip.Text = x.Substring(0, i1);
        lp.Text = x.Substring(i1 + 1, i2 - i1 - 1);
        anp.Text = x.Substring(i2 + 1, 4);
        nrc.Text = Convert.ToString(rez.nrCam);
    }
    else
        MessageBox.Show("Citire nereusita");
}

```

Ce am făcut? Am deserializat din fișier obiectul rez, după care componentele lui sunt afișate în controalele TextBox. Deoarece stringul rezultat în urma conversiei din DateTime este de forma “zz.ll.aaa”, am extras subșirurile corespunzătoare zilei, lunii și anului, pentru a putea fi afișate. Dacă deserializarea a reușit, obiectul rez se încarcă cu datele preluate din fișier. În caz contrar, funcția Deserialize() returnează null.

Compilați și executați programul.

Baze de date în C#

În acest capitol vom studia modul în care se pot realiza programe client în C# pentru accesul la baze de date rezidente pe un servere SQL Server 2005 Express. Tehnologia utilizată este ADO.NET.

ADO.NET este cea mai recentă tehnologie introdusă de Microsoft pentru accesul la date, înglobând și înlocuind tehnologiile mai vechi: ADO (ActiveX Data Objects), COM (Component Object Model) sau tehnologii web, cum ar fi ASP (Active Server Pages). Modalitatea de utilizare a ADO.NET este mult mai ușoară decât cea a tehnologiilor anterioare, oferind de asemenea o flexibilitate sporită în accesul datelor.

ADO.NET pune la dispoziția programatorilor 7 clase de bază:

- `DbConnection`
- `DbCommand`
- `DbDataReader`
- `DbDataAdapter`
- `DataTable`
- `DataRelation`
- `DataSet`

Clasa *DbConnection*

Este clasa care furnizează conexiuni la baza de date. Pentru a realiza o astfel de conexiune, programul va instanția un obiect `DbConnection`, precizând o serie de informații, cum ar fi locația bazei de date, numele utilizatorului, parola și respectiv numele bazei de date. Toate celelalte clase ADO.NET care realizează accesul la date vor utiliza această clasă pentru a-și putea realiza sarcinile. Funcționalitatea clasei `DbConnection` este prezentată în figura 5.1.

Obiectul `DbConnection` va utiliza pentru a realiza efectiv conexiunea la baza de date un șir de conexiune (connection string). Acest șir este stocat în proprietatea `DbConnection.ConnectionString` a obiectului. În general, șirurile de conexiune sunt furnizate de producătorii serverelor de baze de date.

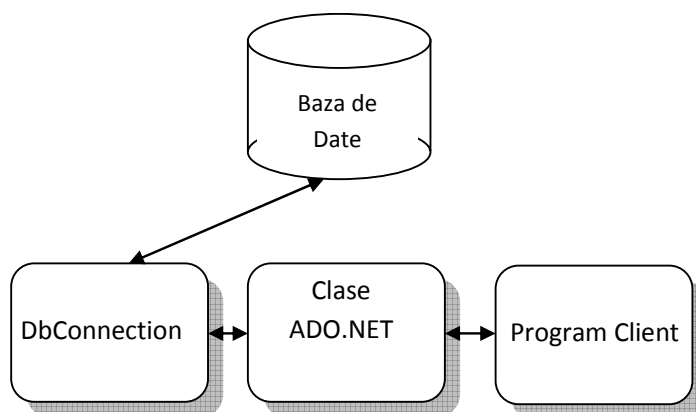


Figura 5.1

Instanțierea unui obiect `DbConnection` nu înseamnă conectarea automată la baza de date, deoarece această conexiune uneori nu este necesar să existe pe tot parcursul execuției programului client. Pentru a conecta și deconecta programul de la baza de date, clasa `DbConnection` expune metodele `Open()` și `Close()`.

Dacă conexiunea este realizată cu un server SQL Server, în locul clasei `DbConnection` poate fi utilizată clasa `SqlConnection`, care este derivată din aceasta. `DbConnection` și clasele derivate vor putea fi specificate ca și fiind *clase de conexiune*.

Clasa `DbCommand`

Clasa `DbCommand` implementează metode de interacțiune primară cu baza de date. Obiectele `DbCommand` vor putea executa interogări SQL, proceduri stocate, etc. Clasele `DbCommand` și clasele derivate vor fi acceptate ca și fiind *clase de comandă*.

Mecanismul de acces implementat de obiectele `DbCommand` este prezentat în figura 5.2.

O proprietate deosebit de importantă a clasei `DbCommand` este `DbCommand.CommandText`. Această proprietate va conține textul frazelor SQL care urmează să fie executate. Tipul comenzii care urmează să fie executată poate fi stocat în proprietatea `DbCommand.CommandType`,

urmând ca accesarea conexiunii sau tranzacției corespunzătoare să se facă utilizând `DbCommand.Connection`, respectiv `DbCommand.Transaction`.

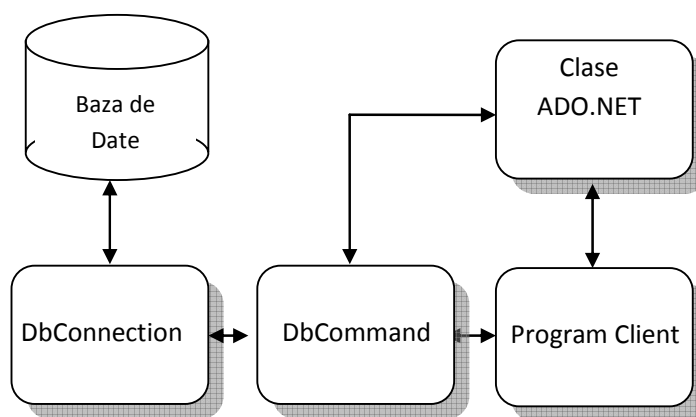


Figura 5.2

Când programul client execută o comandă prin intermediul unui obiect `DbCommand`, programatorul are la dispoziție trei opțiuni de programare a acesteia, în funcție de ceea ce execută comanda. Pentru comenzi care nu returnează nici un rezultat, clasa expune metoda `DbCommand.ExecuteNonQuery()`. Pentru comenzi care returnează un singur rezultat, este expusă metoda `DbCommand.ExecuteScalar()`. Pentru comenzi care returnează date reprezentate pe mai multe rânduri, va fi utilizată metoda `DbCommand.ExecuteReader()`, care returnează un obiect de tip `DbDataReader`. La fel ca și în cazul `DbConnection`, la utilizarea SQL Server va putea fi folosită clasa `SqlCommand`.

Clasa `DbDataReader`

Clasa `DbDataReader` permite citirea datelor dintr-un set de rezultate, citirea fiind optimizată din punctul de vedere al timpului de acces la baza de date. Un obiect

DbDataReader va permite citirea liniilor de o manieră secvențială, ca și consecință a optimizării, nefiind posibilă reîntoarcerea la linii citite anterior. Rolul obiectelor DbDataReader este prezentat în figura 5.3.

Pentru citirea primei linii dintr-un set de date, clasa expune metoda [DbDataReader.Read\(\)](#). Această metodă va fi apoi apelată în mod repetat pentru a se citi restul liniilor din setul de date. Metoda returnează [true](#) dacă s-a reușit citirea liniei, respectiv [false](#) în caz contrar. Înainte de utilizarea acestei metode, este indicat să se citească proprietatea [DbDataReader.HasRows](#) pentru a se vedea dacă setul de date rezultat are mai multe linii.

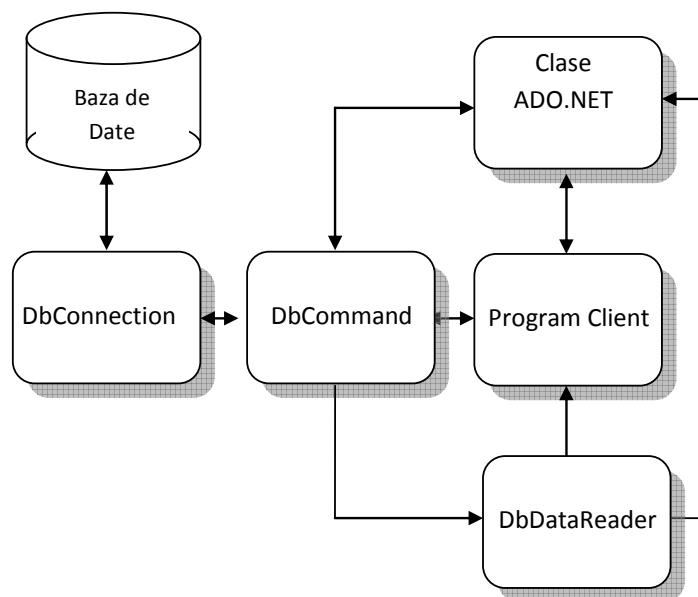


Figura 5.3

Clasa expune un număr relativ mare de proprietăți și metode care permit examinarea liniei curent citite. Citeva exemple ar fi: [DbDataReader.FieldCount](#) va conține numărul de coloane din linia citită. Fiecare coloană va putea fi apoi accesată fie pe baza poziției (bazată pe 0) în linie, fie pe baza numelui. Valoarea conținută de fiecare coloană este un obiect [object](#), care apoi poate fi convertit explicit la orice alt obiect.

Datele pot fi obținute direct în formatul dorit dintr-o coloană selectată pe baza indexului pe baza unui set de metode expuse de clasă, cum ar fi [DbDataReader.GetString\(\)](#), [DbDataReader.GetInt16\(\)](#), etc. Singura problemă care apare la astfel de citiri este că anumite coloane pot să nu conțină valori, situație în care vor fi generate excepții. Înaintea utilizării acestor metode este indicat să de utilizeze [DbDataReader.IsDBNull\(\)](#), care returnează [true](#) în cazul în care o coloană nu conține date.

În cazul utilizării SQL Server, poate fi folosită și clasa [SqlDataReader](#).

Clasa DbDataAdapter

Această clasă are rolul de permite interschimbarea datelor între cursoare (data set) și baza de date, cu o intervenție minimă din partea utilizatorului. Figura 5.4 prezintă modul în care operează obiectele [DbDataAdapter](#).

Obiectul DbDataAdapter pune la dispoziția programatorului cele 4 comenzi de bază pentru lucrul cu baze de date: Select, Insert, Update și Delete, sub forma a 4 proprietăți:

[SelectCommand](#) este utilizată pentru regăsirea datelor; [InsertCommand](#) pentru adăugarea datelor; [UpdateCommand](#) pentru editarea datelor și respectiv [DeleteCommand](#) pentru ștergerea datelor.

Clasa [DbDataAdapter](#) expune de asemenea și un set de metode. Cel mai des utilizate dintre acestea sunt [DbDataAdapter.Fill\(\)](#), care realizează încărcarea obiectului [DbDataAdapter](#) cu datele citite din baza de date, respectiv [DbDataAdapter.Update\(\)](#), care realizează transmiterea înspre baza de date a datelor conținute de obiect. Ambele metode permit schimbarea de date atât la nivelul cursoroarelor, cât și la nivel de tabele individuale.

În plus, se poate determina o schemă a datelor cu [DbDataAdapter.FillSchema\(\)](#).

Dacă se utilizează un server SQL Server, poate fi utilizată clasa [SqlDataAdapter](#).

Clasa [DataTable](#)

Clasa [DataTable](#) este utilizată pentru stocarea tabelelor de date. Un tabel stocat într-un obiect [DataTable](#) nu trebuie în mod necesar să conțină exact un tabel din baza de date. Poate conține și un subset al unui tabel, un număr oarecare de linii, un număr oarecare de coloane, date combinate din mai multe tabele ale bazei de date, etc.

Figura 5.5 prezintă funcționalitatea unui obiect [DataTable](#).

Pentru obținerea unui obiect [DataTable](#) populat cu date, se utilizează un obiect [DataAdapter](#). Odată populat, programatorul poate accesa linii, coloane, constrângeri sau alte informații pe care obiectul le conține. Fiecare din cele 3

proprietăți amintite anterior returnează un obiect de o clasă colecție ([DataRowCollection](#), [DataColumnCollection](#), respectiv [DataConstraintCollection](#)).

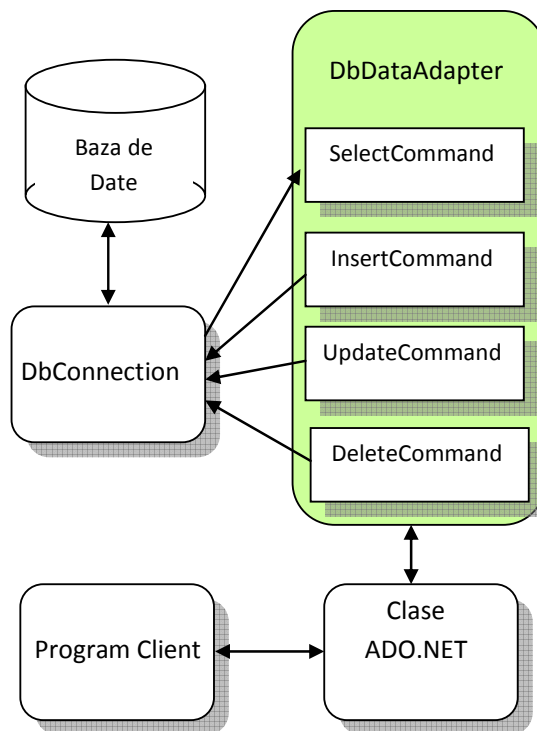


Figura 5.4

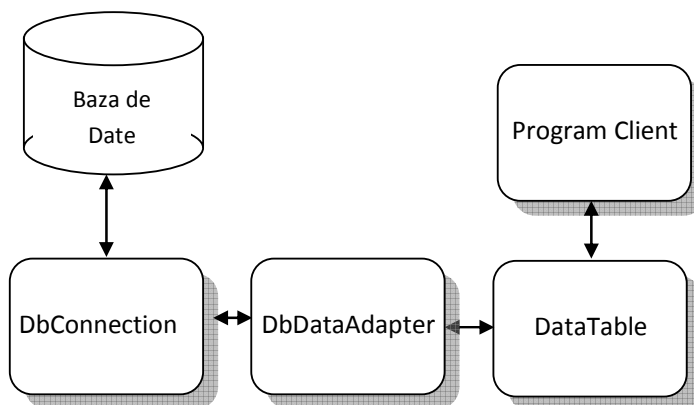


Figura 5.5

Modificarea unei linii dintr-un obiect `DataTable` nu înseamnă automat modificarea acelei linii din baza de date. De exemplu ștergerea unei linii nu va șterge efectiv linia din obiect ci o va marca ca fiind ștearsă. Modificările efectuate asupra datelor din obiect pot fi puse în evidență prin utilizarea metodei `DataTable.GetChanges()`. Acceptarea modificărilor este făcută prin intermediul metodei `DataTable.AcceptChanges()`. Anularea modificărilor se face prin apelul metodei `DataTable.RejectChanges()`.

Obiectele `DataTable` expun de asemenea o serie de evenimente care pot fi interceptate pentru a pune în evidență funcționalitatea programului client: `RowDeleted`, `ColumnChanged`, etc.

- `DataRow` – Clasa `DataRow` este utilizată pentru a stoca date dintr-o linie a obiectului `DataTable`. Proprietatea `DataTable.Rows` permite accesul la un obiect `DataRowCollection` care este o colecție de obiecte `DataRow`. Colanele din fiecare astfel de obiect pot fi accesate pe baza indexului lor. Starea curentă a unei linii poate fi aflată prin intermediul proprietății `DataRow.RowState`. Această proprietate este o marime de tipul enumerare `DataRowState`, care încapsulează toate stările posibile ale unei linii. Acceptarea sau anularea unei modificări se face similar obiectelor `DataTable`, cu metodele `DataRow.AcceptChanges()`, respectiv `DataRow.RejectChanges()`.
- `DataColumn` – Clasa `DataColumn` stochează toate informațiile necesare pentru definirea completă a unei coloane dintr-un obiect `DataTable`. Proprietatea `DataTable.Columns` conține un obiect de clasă `DataColumnCollection`, care este o colecție de obiecte `DataColumn`. Un obiect `DataColumn` va expune un set de proprietăți, cum ar fi `DataColumn.ColumnName`, `DataColumn.DataType`, `DataColumn.AllowDBNull`, `DataColumn.DefaultValue`, etc.
- `Constraint` – Clasa `Constraint` (proprietatea `DataTable.Constraints` care conține obiecte de clasă `ConstraintCollection`) stochează toate informațiile care nu sunt conținute în coloane. Un exemplu de proprietate a acestei clase este `Constraint.ForeignKeyConstraint`, care forțează relații între tabele.

Clasa `DataRelation`

Obiectele de clasă `DataRelation` implementează relații între tabele. Mai multe obiecte `DataRelation` pot fi grupate împreună într-un obiect `DataRelationCollection`.

Relațiile între tabele sunt puse în evidență de diferite proprietăți ale acestei clase, cum ar fi: `DataRelation.ChildTable`, `DataRelation.ChildColumns`, `DataRelation.ChildKeyConstraint`, `DataRelation.ParentTable`, `DataRelation.ParentColumns`, `DataRelation.ParentKeyConstraint`, etc. Numele relației va fi stocată în proprietatea `DataRelation.RelationName`.

Clasa DataSet

Este cea mai importantă clasă pentru manipularea datelor. Poate fi văzută în cel mai simplu mod ca o colecție de obiecte [DataTable](#) și [DataRelation](#). Structura unui obiect [DataSet](#) este prezentată în figura 5.6.

La fel ca și clasa [DataTable](#), modificările făcute în obiectul [DataSet](#) sunt marcate, putând fi aflate, [DataSet.GetChanges\(\)](#), acceptate [DataSet.AcceptChanges\(\)](#) sau rejectate [DataSet.RejectChanges\(\)](#). Proprietăți importante ale obiectelor [DataSet](#) sunt [DataSet.Tables](#), care conține obiectele [DataTable](#) din [DataSet](#) sub forma de colecții [DataTableCollection](#), respectiv [DataSet.Relations](#), care conține relațiile sub formă de colecții [DataRelationCollection](#).

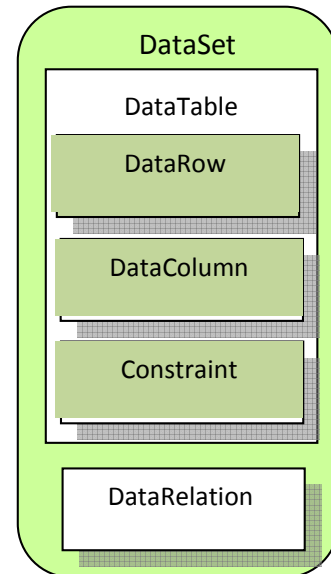


Figura 5.6

Data Binding

[DataBinding](#) reprezintă o tehnică prin care datele stocate în baza de date populează diferitele obiecte de date. Mecanismul necesită existența a 2 elemente: o sursă de date și un obiect (control) ce urmează a fi populat.

Si acum... să creăm prima bază de date

Vom crea o bază de date, numită [Curs](#), care este găzduită de un server SQL Server 2005 Express. Baza de date va conține 2 tabele, [persoane](#) și [functii](#), cu o relație de legătură de tip 1 la 1 între ele.

Primul lucru care îl avem de făcut este să verificăm ca SQL Server este activ. Pentru aceasta, vom lansa [SQL Server Configuration Manager](#) și vom selecta [SQL Server 2005 Services](#). Dacă serverul este activ, imaginea afișată este cea din figura 5.7. Dacă serverul nu este pornit se apasă click dreapta și în meniul contextual se alege **Start**.

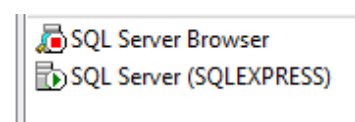


Figura 5.6

Pentru crearea bazei de date vom lansa [SQL Server Management Studio Express](#). Vom accepta conectarea la server și vom începe crearea bazei de date.

- Click dreapta pe folderul [Databases](#) și apoi se tastează [Curs](#) în caseta [Database name](#): (figurile 5.7 și 5.8). După apăsarea butonului [OK](#) baza de date este creată;
- Se expandează subarboarele folderului [Databases](#) și se apasă click dreapta pe butonul [Tables](#). În meniul contextual se alege [New Table...](#) (figura 5.9). Se completează structura tabelului ca și în figura 5.10.

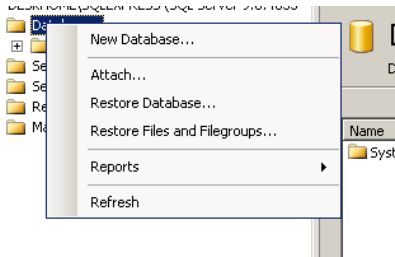


Figura 5.7

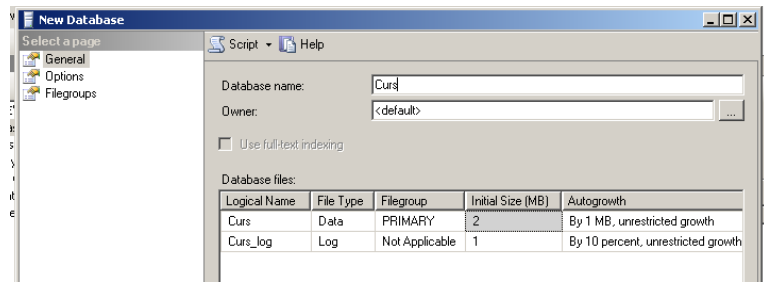


Figura 5.8

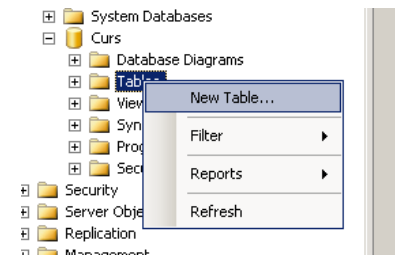


Figura 5.9

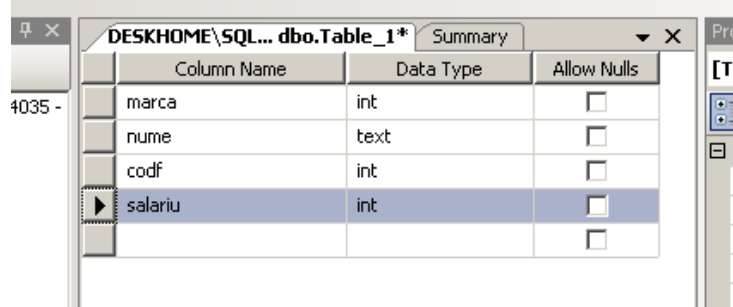


Figura 5.10

- Se apasă click dreapta pe prima linie și în meniul contextual se alege **Set Primary Key** (figura 5.11). În urma acestei operații în dreptul primei linii va apare o cheie, ceea ce va specifica faptul că acest câmp este cheie primară

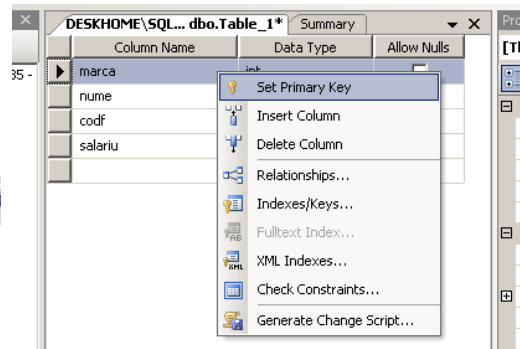



Figura 5.11

- Se salvează tabelul apăsând butonul  și în caseta **Enter table name** se tastează *persoane*;

- Se crează un nou tabel, cu structura din figura 5.12, salvându-se sub numele de *functii*;

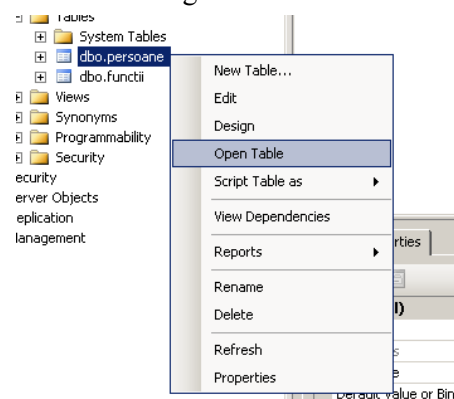



Figura 5.12

- Acum urmează să populăm tabelele. Se expandează subarborul folderului **Tables** apăsă click dreapta pe tabelul *persoane* și în meniul contextual alegem **Open Table** (figura 5.12). Se completează tabelul ca în figura 5.13.

- Se salvează datele apăsând butonul 

- Se repetă operația cu tabelul *functii*, care se completează ca în figura 5.14.

marca	nume	codf	salariu
132	Pop Vasile	22	1000
763	Dan Maria	22	950
312	Miclea Mircea	33	600
286	Pop Livia	10	1800
193	Cosma Gheorghe	10	1900
111	Miclea Liviu	1	3000
421	Nascu Ioan	2	2500
333	Festila Clement	10	15000
*	NULL	NULL	NULL

Figura 5.13

codf	denf
1	Director General
2	Director Tehnic
10	Inginer
22	Muncitor Calificat
*	NULL

Figura 5.14

- Urmează implementarea relației între tabele. Pentru aceasta se apasă click dreapta pe tabelul *persoane* și se alege din meniul contextual **Design**. Se face click dreapta pe linia *codf* și în meniul contextual se alege **Relationships...** (figura 5.15);
- Se apasă butonul **Add** și se apasă butonul din dreapta liniei **Tables and Columns Specifications** (figura 5.16);

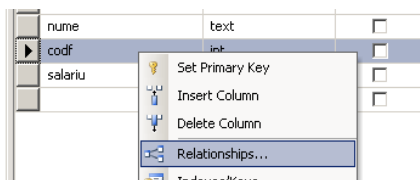


Figura 5.15

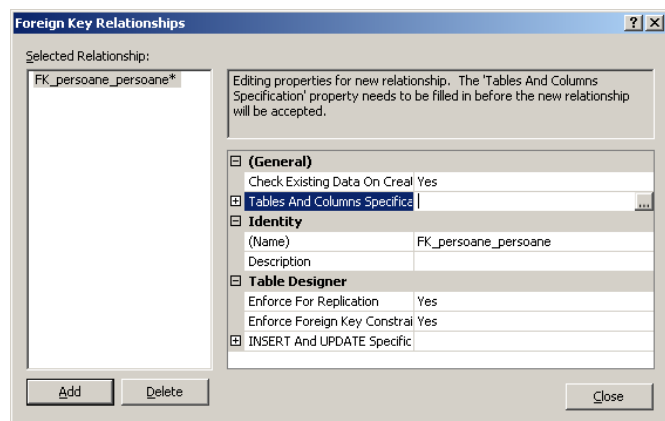


Figura 5.16

- Se selectează relația dintre coloanele tabelor ca în figura 5.17 și se apasă **OK** și **Close**;
- Cu aceasta implementarea relației între tabele este terminată. În continuare se poate verifica funcționalitatea ei, executând un *query* asupra unei structuri de tip *join* între cele 2 tabele. Pentru aceasta:
- Se apasă click dreapta pe folderul **Curs** și în meniul contextual se alege **New Query**;
- În fereastra goală din mijlocul panoului se apasă click dreapta și se alege **Design Query in Editor...**;

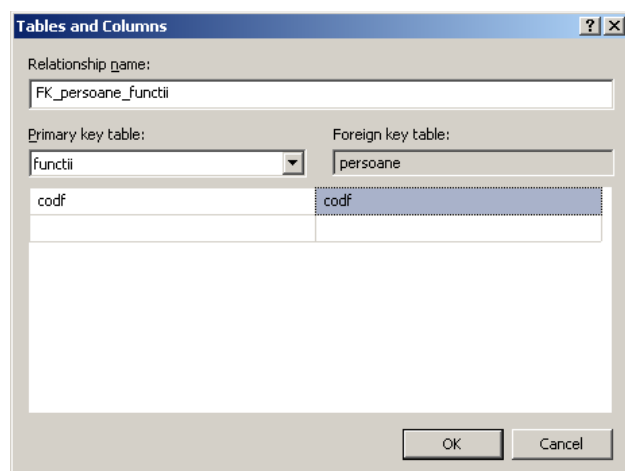


Figura 5.17

- Se selectează ambele tabele (figura 5.18) și se apasă **Add** și **Close**;

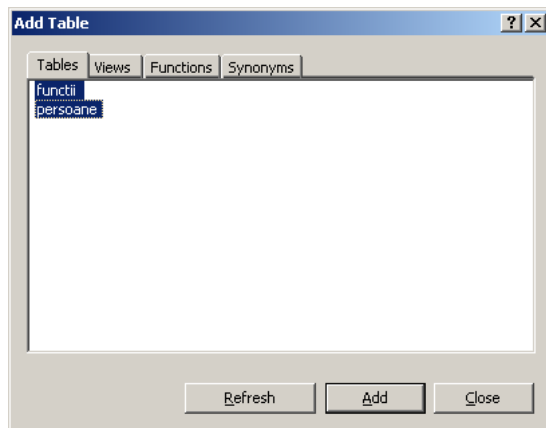


Figura 5.18

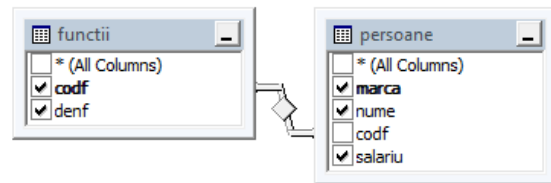


Figura 5.19

- În tabelele apărute, se selectează toate coloanele, mai puțin coloana **codf** din tabelul **persoane** (figura 5.19);
- Se apasă **OK**. Apoi se face iar click dreapta în fereastra din mijloc și în meniul contextual se alege **Execute**. Va fi afișat un grid cu informațiile cumulate din cele 2 tabele (figura 5.20);

	codf	denf	marca	nume	salariu
1	1	Director General	111	Miclea Liviu	3
2	22	Muncitor Calificat	132	Pop Vasile	1
3	10	Inginer	193	Cosma Gheorghe	1
4	10	Inginer	286	Pop Livia	1
5	33	Muncitor Necalificat	312	Miclea Mircea	6
6	10	Inginer	333	Festila Clement	1
7	2	Director Tehnic	421	Nascu Ioan	2
8	22	Muncitor Calificat	763	Dan Maria	9

Figura 5.20

- Cu aceasta implementarea bazei de date este gata. Urmează implementarea programului client.

Implementarea unui program client simplu

Pentru exemplificarea modului de realizare al unui program client simplu, să creăm un proiect de tip **Windows Application** pe care să-l numim *Client1*. Apoi va trebui să conectăm programul la sursa de date. Pentru aceasta:

- În meniul **Data** se alege **Add New Data Source...**. În caseta data **Source Configuration Wizzard** se alege **Database** și se apasă butonul **Next**.
- În ecranul 2 al casetei de configurare se apasă **New Connection**;

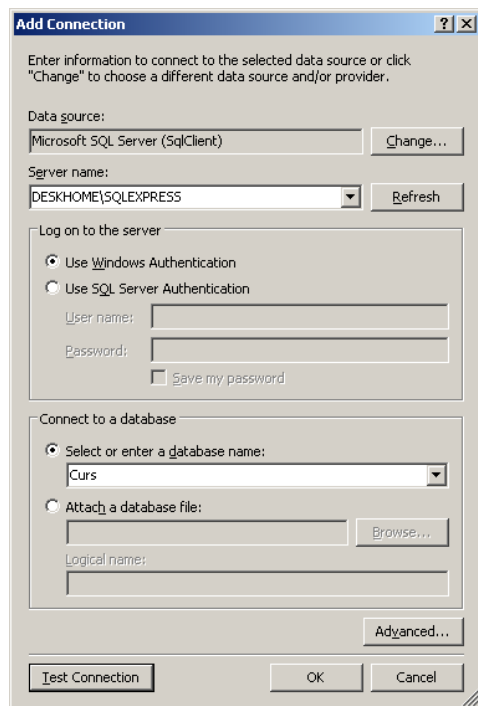


Figura 5.21

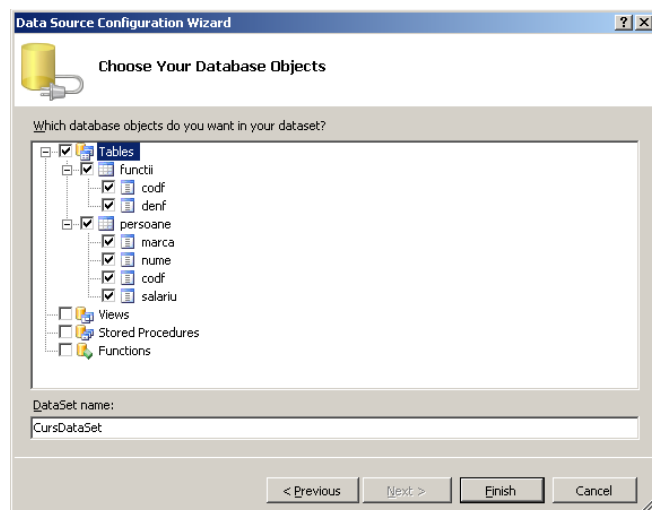


Figura 5.22

- În caseta **Add Connection** (figura 5.21) se alege numele serverului care găzduiește baza de date și numele bazei de date. Se poate testa funcționalitatea conexiunii prin apăsarea butonului **Test Connection**. La apăsarea butonului **OK**, va fi returnat șirul de conexiune necesar conectării aplicației la baza de date. Se apasă apoi **Next**;
- Se acceptă salvarea conexiunii și se apasă din nou **Next**;
- Se selectează checkbox-urile corespunzătoare celor 2 tabele (figura 5.22) astfel încât acestea să fie adăugate la aplicație și se apasă **Finish**. Cu aceasta aplicația este conectată complet la baza de date.

Se poate trece la construirea interfeței. Vom construi o interfață ca în figura 5.23, prin adăugarea unui control **ListBox** și a 6 controale **Label**. Modificăm proprietățile **Text** a celor 3 controale **Label** din stânga ca în figură și proprietățile **Name** a controalelor din dreapta, de sus în jos în **txtMarca**, **txtFct** și respectiv **txtSal**. Modificăm proprietatea **Name** pentru controlul **ListBox** în **txtPers**.

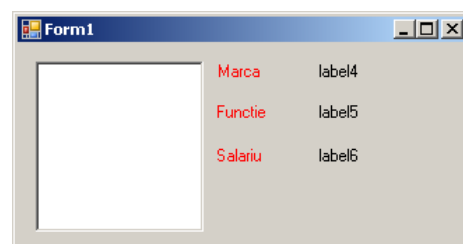


Figura 5.23

Urmează să conectăm controlul **ListBox** la sursa de date. Pentru acesta

- Se selectează săgeata din partea de sus a controlului (figura 5.24) și se marchează controlul **Use data bound items**.
- În caseta **Data Source** se alege **Other Data Sources**, **Project Data Sources**, **Curs DataSet** și respectiv tabelul **persoane** (figura 5.25). În acest mod controlul a fost conectat la un **DataSet** care conține datele din tabelul **persoane**.

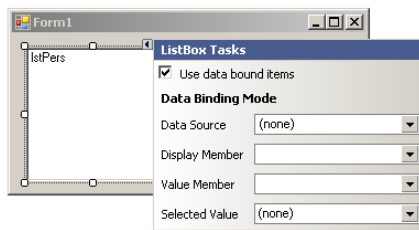


Figura 5.24

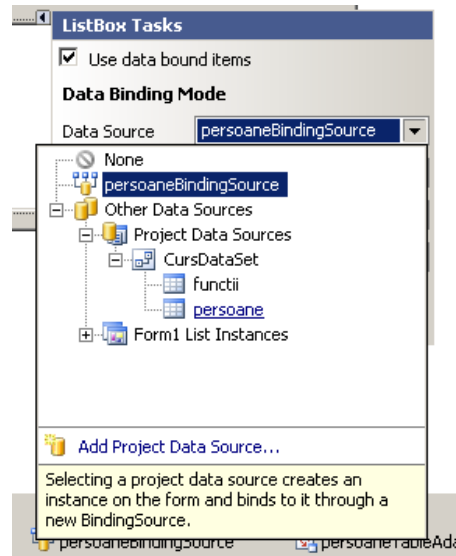


Figura 5.25

- În caseta **Display Member** se alege câmpul **nume**. În acest fel, în casetă vor fi afișate câmpurile **nume** din tabelul *persoane*.

Să intrăm acum în zona de cod. Se observă că în `Form1Load()` a fost inserată automat următoarea linie:

```
private void Form1_Load(object sender, EventArgs e)
{
    // TODO: This line of code loads data into the 'cursDataSet.persoane' table. ...
    this.persoaneTableAdapter.Fill(this.cursDataSet.persoane);
}
```

În program, a fost declarat un **TableAdapter** asociat controlului **ListBox**. Acesta este „umplut” cu conținutul **DataSet**-ului asociat tabelului *persoane*. Atât adapterul cât și dataset-ul au fost generate automat. Cum în baza de date există 2 tabele, va trebui să creăm un table adapter și pentru tabelul **functii**, astfel încât să putem manipula în program conținutul acestuia:

```
...
public partial class Form1 : Form
{
    CursDataSetTableAdapters.functiiTableAdapter functiiTableAdapter =
        new CursDataSetTableAdapters.functiiTableAdapter();
    public Form1()
    ...
```

Evident, pasul următor va fi să „umplem” acest table adapter cu informația din **DataSet**-ul asociat tabelului *functii*.

```
private void Form1_Load(object sender, EventArgs e)
{
```

```
// TODO: This line of code loads data into the 'cursDataSet.persoane' table...
this.persoaneTableAdapter.Fill(this.cursDataSet.persoane);
this.functiiTableAdapter.Fill(this.cursDataSet.functii);
}
```

Acum va trebui să actualizăm conținutul etichetelor la fiecare selecție a unei noi persoane în controlul **ListBox**. Pentru a genera o metodă asociată evenimentului **SelectIndexChanged** în **ListBox**, apăsăm dublu click pe control și implementăm codul:

```
private void lstPers_SelectedIndexChanged(object sender, EventArgs e)
{
    if (lstPers.SelectedItem != null)
    {
        DataRowView SelectedRowView = (lstPers.SelectedItem as DataRowView);
        CursDataSet.persoaneRow selPers = SelectedRowView.Row as CursDataSet.persoaneRow;
        txtMarca.Text = Convert.ToString(selPers.marca);
        txtFct.Text = selPers.functiiRow.denf;
        txtSal.Text = Convert.ToString(selPers.salariu);
    }
}
```

Să explicăm codul. Instrucțiunea **if** asigură faptul că în controlul **ListBox** nu a fost inserată (din greșeală) o linie vidă, situație în care nu am putea regăsi nici o înregistrare din tabel. Apoi:

```
DataRowView SelectedRowView = (lstPers.SelectedItem as DataRowView);
```

Prin intermediul acestei instrucțiuni, se selectează linia din tabelul persoane corespunzător liniei selectate din controlul **ListBox**. Se declară un obiect de clasă **DataRowView**, care se încarcă cu linia corespunzătoare elementului selectat în controlul **ListBox**.

```
CursDataSet.persoaneRow selPers = SelectedRowView.Row as CursDataSet.persoaneRow;
```

Bazat pe linia extrasă anterior, se selectează linia corespunzătoare din dataset-ul asociat bazei de date curs. Tot ce mai avem de făcut este să completăm conținutul etichetelor. Singurul lucru demn de luat în seamă este instrucțiunea

```
txtFct.Text = selPers.functiiRow.denf;
```

care extrage informația din tabelul **functii**, bazat pe legătura permanentă definită în baza de date.

Tot ce mai avem de făcut este să afișăm conținutul etichetelor și pentru prima înregistrare:

```
...
this.persoaneTableAdapter.Fill(this.cursDataSet.persoane);
this.functiiTableAdapter.Fill(this.cursDataSet.functii);
DataRowView SelectedRowView = (lstPers.SelectedItem as DataRowView);
CursDataSet.persoaneRow selPers = SelectedRowView.Row as CursDataSet.persoaneRow;
```

```

txtMarca.Text = Convert.ToString(selPers.marca);
txtFct.Text = selPers.functiiRow.denf;
txtSal.Text = Convert.ToString(selPers.salariu);
}

```

Se compilează și se execută programul.

Operații cu baze de date

Citirea bazei de data cu SqlDataReader

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace tcdb1
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection cursCon = new SqlConnection(@"Data Source=DESKHOME\SQLEXPRESS;
                                                    Initial Catalog=Curs;Integrated Security=True");

            cursCon.Open();
            SqlCommand cursCom1 = cursCon.CreateCommand();
            cursCom1.CommandText = "SELECT * FROM persoane ORDER BY marca";
            SqlDataReader cursRead1 = cursCom1.ExecuteReader();
            while (cursRead1.Read())
            {
                Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cursRead1["marca"], cursRead1["nume"],
                                cursRead1["codf"], cursRead1["salariu"]);
            }
            cursRead1.Close();
            cursCon.Close();
            Console.ReadLine();
        }
    }
}

```

Citirea bazei de date cu DataSet

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

```

```

namespace tcdb21
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection cursCon = new SqlConnection(@"Data Source=DESKHOME\SQLEXPRESS;
                                                    Initial Catalog=Curs;Integrated Security=True");

            cursCon.Open();
            SqlDataAdapter cursAdapter = new SqlDataAdapter("SELECT * FROM persoane
                                                         ORDER BY marca", cursCon);

            DataSet persoaneDataSet = new DataSet();
            cursAdapter.Fill(persoaneDataSet, "persoane");
            foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
            {
                Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                                cRow["codf"], cRow["salariu"]);
            }
            cursCon.Close();
            Console.ReadLine();
        }
    }
}

```

Modificarea unei înregistrări

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace tcdb3
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection cursCon = new SqlConnection(@"Data Source=DESKHOME\SQLEXPRESS;
                                                    Initial Catalog=Curs;Integrated Security=True");

            //cursCon.Open();
            SqlDataAdapter cursAdapter = new SqlDataAdapter("SELECT * FROM persoane
                                                         ORDER BY marca", cursCon);

            SqlCommandBuilder cursBuilder = new SqlCommandBuilder(cursAdapter);
            DataSet persoaneDataSet = new DataSet();
            cursAdapter.Fill(persoaneDataSet, "persoane");
            Console.WriteLine(" Inainte de schimbare");
            foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
            {
                Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                                cRow["codf"], cRow["salariu"]);
            }
        }
    }
}

```



```

    }
    persoaneDataSet.Tables["persoane"].Rows[3]["salariu"] = 30000;
    cursAdapter.Update(persoaneDataSet, "persoane");
    Console.WriteLine(" Dupa schimbare");
    foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
    {
        Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
            cRow["codf"], cRow["salariu"]);
    }
    cursCon.Close();
    Console.ReadLine();
}
}
}

```

Adăugarea unei înregistrări

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace tcdb4
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection cursCon = new SqlConnection(@"Data Source=DESKHOME\SQLEXPRESS;
                Initial Catalog=Curs;Integrated Security=True");

            //cursCon.Open();
            SqlDataAdapter cursAdapter = new SqlDataAdapter("SELECT * FROM persoane
                ORDER BY marca", cursCon);

            SqlCommandBuilder cursBuilder = new SqlCommandBuilder(cursAdapter);
            DataSet persoaneDataSet = new DataSet();
            cursAdapter.Fill(persoaneDataSet, "persoane");
            Console.WriteLine(" Inainte de adaugare");
            foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
            {
                Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                    cRow["codf"], cRow["salariu"]);
            }
            DataRow persoaneRow = persoaneDataSet.Tables["persoane"].NewRow();
            persoaneRow["marca"] = 232;
            persoaneRow["nume"] = "bla bla";
            persoaneRow["codf"] = 10;
            persoaneRow["salariu"] = 2000;
            persoaneDataSet.Tables["persoane"].Rows.Add(persoaneRow);
            try
            {
                cursAdapter.Update(persoaneDataSet, "persoane");
            }
        }
    }
}

```

```

        Console.WriteLine(" Dupa adaugare");
        foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
        {
            Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                cRow["codf"], cRow["salariu"]);
        }
    }
    catch (SqlException ex)
    {
        Console.WriteLine(" Nu am putut actualiza baza de date: " + ex.Message);
    }
    cursCon.Close();
    Console.ReadLine();
}
}
}

```

Căutarea unei înregistrări

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace tcdb5
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection cursCon = new SqlConnection(@"Data Source=DESKHOME\SQLEXPRESS;
                Initial Catalog=Curs;Integrated Security=True");

            cursCon.Open();
            SqlDataAdapter cursAdapter = new SqlDataAdapter("SELECT * FROM persoane
                ORDER BY marca", cursCon);

            SqlCommandBuilder cursBuilder = new SqlCommandBuilder(cursAdapter);
            DataSet persoaneDataSet = new DataSet();
            cursAdapter.Fill(persoaneDataSet, "persoane");
            Console.WriteLine(" Tabelul este");
            foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
            {
                Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                    cRow["codf"], cRow["salariu"]);
            }
            DataColumn[] pk = new DataColumn[1];
            pk[0] = persoaneDataSet.Tables["persoane"].Columns["marca"];
            persoaneDataSet.Tables["persoane"].PrimaryKey = pk;
            DataRow caut = null;
            while (caut == null)
            {
                Console.Write(" Dati marca: ");
            }
        }
    }
}

```

```

        int mrc = Convert.ToInt16(Console.ReadLine());
        caut = persoaneDataSet.Tables["persoane"].Rows.Find(mrc);
    }
    Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", caut["marca"], caut["nume"],
        caut["codf"], caut["salariu"]);

    cursCon.Close();
    Console.ReadLine();
}
}
}

```

Ștergerea unei înregistrări

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace tcdb6
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection cursCon = new SqlConnection(@"Data Source=DESKHOME\SQLEXPRESS;
                Initial Catalog=Curs;Integrated Security=True");

            cursCon.Open();
            SqlDataAdapter cursAdapter = new SqlDataAdapter("SELECT * FROM persoane
                ORDER BY marca", cursCon);

            SqlCommandBuilder cursBuilder = new SqlCommandBuilder(cursAdapter);
            DataSet persoaneDataSet = new DataSet();
            cursAdapter.Fill(persoaneDataSet, "persoane");
            Console.WriteLine(" Inainte de stergere");
            foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
            {
                Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                    cRow["codf"], cRow["salariu"]);
            }
            DataColumn[] pk = new DataColumn[1];
            pk[0] = persoaneDataSet.Tables["persoane"].Columns["marca"];
            persoaneDataSet.Tables["persoane"].PrimaryKey = pk;
            DataRow caut = null;
            while (caut == null)
            {
                Console.Write(" Dati marca: ");
                int mrc = Convert.ToInt16(Console.ReadLine());
                caut = persoaneDataSet.Tables["persoane"].Rows.Find(mrc);
            }
            try
            {

```

```

        caut.Delete();
        cursAdapter.Update(persoaneDataSet, "persoane");
        Console.WriteLine(" Dupa de stergere");
        foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
        {
            Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                cRow["codf"], cRow["salariu"]);
        }
    }
    catch (SqlException ex)
    {
        Console.WriteLine(" Nu s-a sters. Eroare " + ex.Message);
    }
    cursCon.Close();
    Console.ReadLine();
}
}
}

```

Implementarea relațiilor între tabele

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace tcdb7
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection cursCon = new SqlConnection(@"Data Source=DESKHOME\SQLEXPRESS;
                Initial Catalog=Curs;Integrated Security=True");

            cursCon.Open();
            SqlDataAdapter persoaneAdapter = new SqlDataAdapter("SELECT * FROM persoane
                ORDER BY marca", cursCon);

            SqlDataAdapter functiiAdapter = new SqlDataAdapter("SELECT * FROM functii
                ORDER BY codf", cursCon);

            SqlCommandBuilder cursBuilder = new SqlCommandBuilder(persoaneAdapter);
            DataSet persoaneDataSet = new DataSet();
            persoaneAdapter.Fill(persoaneDataSet, "persoane");
            functiiAdapter.Fill(persoaneDataSet, "functii");
            DataRelation relPersFct = persoaneDataSet.Relations.Add("codf",
                persoaneDataSet.Tables["functii"].Columns["codf"],
                persoaneDataSet.Tables["persoane"].Columns["codf"]);

            Console.WriteLine(" Acum complet");
            foreach (DataRow cRow in persoaneDataSet.Tables["persoane"].Rows)
            {
                Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", cRow["marca"], cRow["nume"],
                    cRow["codf"], cRow["salariu"]);
            }
        }
    }
}

```

```
        foreach(DataRow fRow in cRow.GetParentRows(relPersFct))
        {
            Console.WriteLine("\t{0}", fRow["denf"]);
        }
    }
    cursCon.Close();
    Console.ReadLine();
}
}
```

2. Aplicații ASP.NET

ASP.NET permite realizarea de aplicații web complexe. Spre deosebire de celelalte tehnologii pe care le-ați studiat, realizarea aplicațiilor web în ASP.NET aduce o serie de avantaje importante:

- aplicațiile pot fi scrise utilizând toate facilitățile unor limbaje și medii de programare dezvoltate. În ceea ce urmează, vom realiza implementarea aplicațiilor ASP.NET în C#, dar există posibilitatea implementării acestora în multe alte limbaje;
- sunteți obișnuiți cu tehnicile de scriptare. Ați întâlnit 2 astfel de tehnici, cea bazată pe limbajul JavaScript și respectiv cea bazată pe PHP. În primul caz, scriptarea se face la client, tehnica fiind numită *client-side-scripting*, iar în cel de-al doilea caz, scriptarea se face la nivelul serverului, fiind numită *server-side-scripting*. Față de tehnicile uzuale de *server-side-scripting*, ASP.NET introduce și noțiunea de *server-side-controls*. În tehnologiile uzuale, controalele sunt implementate manual și în plus programatorul trebuie să aibă grijă să implementeze modul de manifestare al acestora la acțiunile utilizatorului. Problema este destul de complexă în realitate, indiferent de locația la care se face scriptarea. Aparent problema este relativ simplă în cazul scriptării la client, de exemplu cu JavaScript. Se pot implementa în acest caz variabile care să rețină conținutul de moment al controalelor și respectiv comportamentul acestora. Problema majoră apare din faptul că browserele implementează în mod diferit JavaScript și este destul de dificil de scris rutine care să funcționeze absolut similar pe toate browserele. O alternativă este de a scrie cod HTML simplu pentru controale și de a retrimite pagina spre server pentru refacerea ei în noul context. Și în acest caz mecanismul este complicat, deoarece toate informațiile stocate în variabilele din pagină se vor pierde. ASP.NET rezolvă această problemă prin intermediul controalelor pe server. Aceste controale scriptează codul HTML necesar afișării controlului în browser, dar, de asemenea, generează și funcțiile JavaScript și respectiv codul HTML ascuns care păstrează starea controlului. Când utilizatorul acționează asupra controlului, informația este furnizată înapoi serverului, iar controlul procesează automat informația și alterează codul HTML astfel încât afișarea acestuia să se facă în noul context.
- probabil cel mai mare avantaj al ASP.NET este posibilitatea creării de *servicii web*. La această problemă vom reveni.

2.1 Crearea unei aplicații simple

Pentru crearea unei aplicații simple ASP.NET vom crea un nou proiect (intitulat web1) de tip **ASP.NET Web Application**. Vom constata că se crează un proiect cu o interfață diferită de cea cu care suntem obișnuiți și afiează un text HTML (fig. 2.1). Este un început pentru pagina pe care urmează să o dezvoltăm. Observăm că această pagină poartă denumirea **Default.aspx**. Va trebui să precizăm mediului de programare că această pagină este cea pe care dorim s-o depanăm în caz de eroare, ea fiind pagina de start a aplicației. Pentru aceasta, apăsăm click dreapta pe Default.aspx în Solution Explorer și selectăm Set as Start Page.

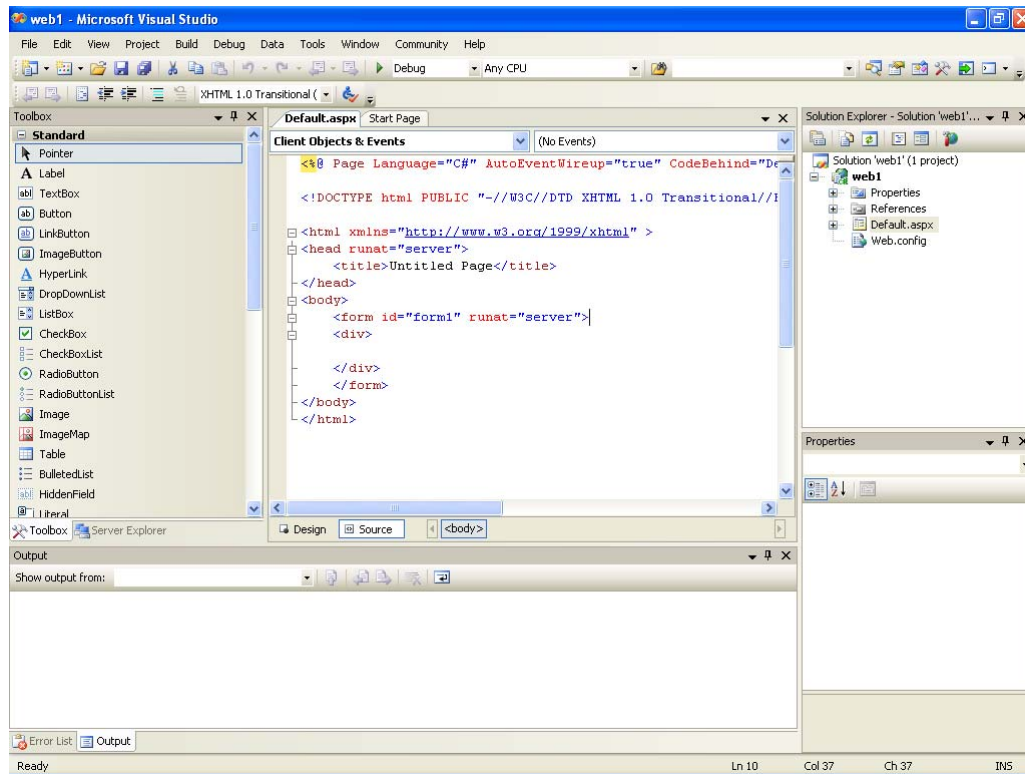


Figura 2.1

Pagina afișată de Visual Studio ne prezintă pagina care va fi trimisă spre browser. Aceasta poate fi abordată și în Designer, în mod grafic, selectând tab-ul **Design**. Selectați-l și observați ce afișază mediul.

Haideți să revenim la tab-ul Source și să-i adăugăm câteva linii de cod:

```

<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Default.aspx.cs"
Inherits="web1._Default" %>

```

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

```

```

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
<title>Untitled Page</title>
</head>
<body>

```

```

<form id="form1" runat="server">
<div>

```

Cursuri urmate:

```

</div>

```

```

<br>

```

```

<div>

```

Programare avansată în C#

```

</div>

```

```

</form>

```

```

</body>

```

```

</html>

```

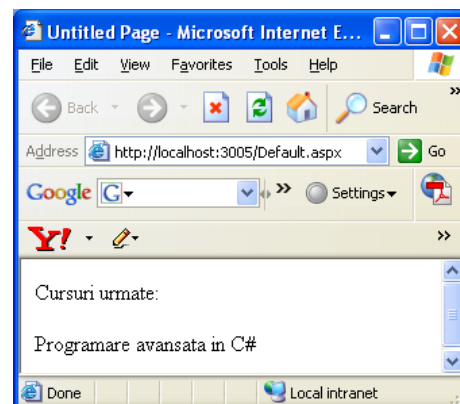


Figura 2.2

Compilați și executați programul. Vom obține rezultatul din fig. 2.2.

Cum funcționează? Simplu. Am creat un proiect Visual Studio și acesta s-a ocupat de tot ceea ce se petrece în spatele scenei, în așa fel încât pagina să poată fi scriptată și trimisă spre browser. Este adevărat, în această pagină încă nu avem nimic dinamic, este o aplicație foarte simplă, dar mecanismul este cel care este utilizat și pentru realizarea paginilor complexe.

Să încercăm acum să adăugăm componente dinamice paginii noastre. Dacă revenim în **Designer**, vom avea imaginea din fig. 2.3:

Să încercăm să reedităm pagina în designer. Selectăm și ștergem textul “Programare avansată în C#” și adăugăm din **Toolbox** un control **Label** (fig. 2.4). Să schimbăm acum pentru controlul **Label** proprietatea **Text** în “Curs” și proprietatea **ID** în Curs.

Acum, fie apăsăm dublu click în interiorul Designer-ului, fie selectăm iconița de cod din Solution Explorer. Putem observa că trecerea între Designer și cod se face cu aceleași iconițe care făceau trecerea între formă și codul clasei asociate în proiectele de tip Windows Form. Am ajuns în zona de cod, în care se implementează codul care dă caracterul dinamic al paginii. Să modificăm metoda `Page_Load()` ca mai jos:

```
protected void Page_Load(object sender, EventArgs e)
{
    int an = DateTime.Now.Year;
    if (an == 2007)
        Curs.Text="Programare avanasata in C# - examen in anul "+an;
    else
        Curs.Text="Programare avanasata in C# - examen in anii 2007 – "
+an;
}
```

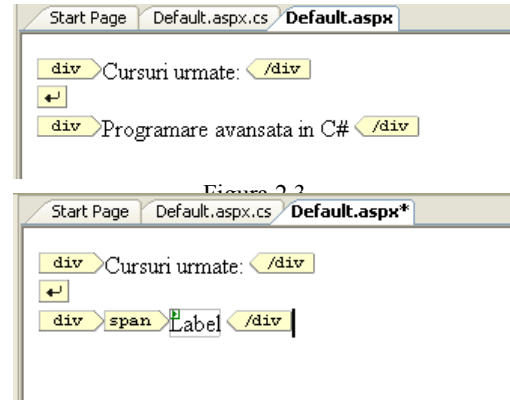


Figura 2.4

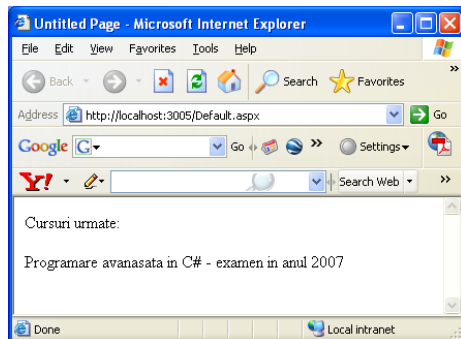


Figura 2.4a

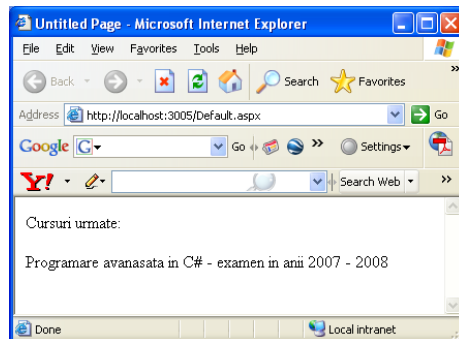


Figura 2.4b

Compilați și executați programul. Veți obține pagina din fig. 2.4a. Dacă veți intra în Settings și veți schimba anul, de exemplu la 2008 și veți da Refresh în browser, veți obține pagina din fig. 2.4b.

Cum funcționează? Putem observa că mesajul referitor la curs este dinamic. Ca o observație importantă, controlul Label fiind un server-side-control, informația referitoare la an este luată de pe server, calculatorul dumneavoastră funcționând acum ca server de web.

Când serverul construiește pagina, utilizează metoda Page_Load(). Aceasta extrage data curentă prin intermediul metodei DateTime.Now() într-un obiect **DateTime** și respectiv obține anul curent din proprietatea Year a acestuia. Apoi, face comparația și asociază proprietății **Text** a controlului Label textul corespunzător. Apoi, face scriptarea. Dar cum ajunge alamentul activ să fie afișat în pagină? Pentru aceasta să revenim în tab-ul **Source** al Designer-ului. Observăm ca Visual Studio a scriptat următorul cod:

```
<body>
  <form id="form1" runat="server">
    <div>
      Cursuri urmate:
    </div>
    <br>
    <div>
      <asp:Label ID="Curs" runat="server"
text="Curs"></asp:Label>&nbsp;</div>
    </form>
  </body>
</html>
```

Tag-ul <asp> implementează zona activă a codului. Înainte ca fișierul să fie trimis spre browser, ASP.NET inspectează complet codul acestuia, pentru a determina părțile statice și părțile dinamice. Apoi, înlocuiește părțile dinamice cu scriptarea HTML corespunzătoare. În cazul nostru, elementul dinamic este proprietatea **Text** a controlului Label. În fapt, textul trimis spre browser (View Source) este următorul:

```
<div>
  Cursuri urmate:
</div>
<br>
<div>
  <span id="Curs">Programare avansata in C# - examen in anul
    2007</span>&nbsp;</div>
</form>
</body>
</html>
```

Prin scrierea paginii web utilizând Visual Studio, practic aceasta va fi codificată în 2 jumătăți: prima, salvată într-un fișier .aspx, va conține șablonul de desenare al paginii și va defini ce elemente active conține și unde apar acestea și respectiv o a doua jumătate, salvată într-un fișier .aspx.cs care conține codul de manipulare al evenimentelor.

Sintetizând, putem spune că modul de lucru este următorul: ASP.NET încarcă codul .aspx și caută elementele dinamice. Pentru aceasta, păstrează în memorie imaginea HTML a paginii dar marchează fiecare linie din cod în care găsește elemente dinamice. Apoi, permite controalelor să-și pună propriul cod HTML în locul marcajelor. În cazul nostru, prin modificarea proprietății **Text** a controlului Label, i-

am spus acestuia că atunci când ASP.NET îi cere codul HTML corespunzător, să-l insereze în locul precizat:

```
<span id="Curs">Programare avansata in C# - examen in anul 2007</span>
```

sau cu alte cuvinte, să insereze conținutul proprietății **Text** și să adauge tag-urile `` și `` în jurul acestuia.

2.2 Gestiunea informațiilor externe

Unul din marile avantaje al utilizării paginilor active este că dacă dorim să adăugăm sau să schimbăm conținutul paginii, nu e nevoie să rescriem codul HTML al acesteia. Astfel, noul conținut va trebui să fie stocat sub o formă accesibilă pentru scriptare și apoi adăugat conținutului HTML. Metoda descrisă în continuare va stoca informația în fișiere .xml, fără a fi nevoie să înțelegem în detaliu structura acestora.

Să presupunem, de exemplu, că am urmat mai multe cursuri, pentru care am dat examen la diferite momente. Informațiile legate de aceste examene le vom stoca în fișiere XML. Pentru ca infrastructura .NET să fie capabilă să încarce aceste fișiere, va trebui să precizăm calea spre acestea. După cum știți, fișierele afișate ca pagini de web, vor trebui să fie stocate în directoare bine definite ale serverului de web. La fel și în cazul nostru. Pentru început, va trebui să aflăm calea spre pagina Default.aspx. Să adăugăm pentru aceasta un nou control **Label**. Vom adăuga un nou control **Label** și la tab-ul **Source** vom adăuga tag-urile `<div>` și `</div>`:

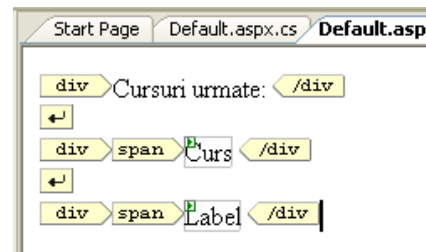


Figura 2.5

```
<div>
  <asp:Label ID="Curs" runat="server"
text="Curs"></asp:Label>&nbsp;</div>
  <br>
  <div>
  </div>
</form>
```

Să modificăm proprietatea **Name** pentru noul control în **Cale** și de asemenea să modificăm funcția `Page_Load()` ca mai jos:

```
protected void Page_Load(object sender, EventArgs e)
{
  ...
  Curs.Text = "Programare avansata in C# - examen in anii 2007 - " + an;
  Cale.Text = " Calea spre server: " + Server.MapPath("");
}
```

Ce am făcut? Am apelat metoda `Server.MapPath()` care transformă o cale virtuală în `www` într-o cale fizică pe serverul de web. În acest fel am aflat unde este găzduită pagina noastră.

Haideți să creăm în directorul returnat un subdirector numit **Cursuri**. În acest director, să creăm (eventual cu ajutorul Notepad) fișierul **unu.xml**, cu conținutul de mai jos:

```
<?xml version="1.0" ?>
<InfCurs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Titlu> CURS 1 </Titlu>
    <Detalii> Primul curs: Programare avansata in C# </Detalii>
</InfCurs>
```

Un astfel de fișier va stoca informația pe care o vom afișa în pagina de web. Evident, în aceeași manieră în care vom afișa informația dintr-un singur fișier, vom reuși să afișăm și informația din mai multe. Probabil tag-urile vi se par ciudate. Vom lămuri și acest lucru imediat.

Primul pas este să încărcăm acest fișier. Pentru aceasta, va trebui să generăm o metodă de încărcare într-o clasă specială, numită **Global**, care stochează date și metode globale, accesibile tuturor paginilor din aplicația web. Această clasă trebuie definită într-o pagină numită **Global.asax**. Pentru a adăuga pagina, apăsăm click dreapta în Solution Explorer pe rădăcină, apoi **Add, Component** și în final alegem **Global Application Class**. Cu aceasta, pagina **Global.asax** a fost adăugată proiectului (fig. 2.6).

Ca să putem încărca fișierele **.xml**, va trebui ca în clasa **Global** să declarăm o variabilă care va stoca calea spre acestea. În codul **Global.asax** vom defini această cale:

```
public class Global : System.Web.HttpApplication
{
    public static String CaleDirCurs;
    ...
}
```

Această cale, va trebui să fie completată la pornirea aplicației. La acel moment, se lansează automat în execuție o funcție numită **Application_Start()**, al cărei schelet îl puteți vedea în cod. Să modificăm această funcție:

```
protected void Application_Start(object sender,
EventArgs e)
{
    CaleDirCurs =
    Server.MapPath("Cursuri");
}
```

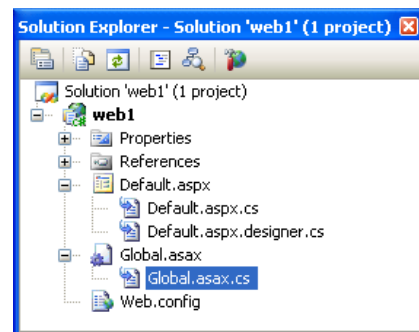


Figura 2.6

Prin aceasta, am încărcat variabila **CaleDirCurs** cu calea spre directorul unde am stocat fișierul **.xml**.

Informațiile pe care dorim să le manipulăm sunt: data asociată elementelor unui curs, titlul și detalii despre acestea. Pentru a manipula aceste informații, vom adăuga proiectului o nouă clasă, care să conțină ca attribute mărimile pe care dorim să le manipulăm. Fie **InfCurs** această clasă (click dreapta pe rădăcină în Solution Explorer, **Add, Class** și respectiv scriem numele clasei). De ce **InfCurs**? Mai țineți

mintre structura fișierului .xml? Acolo marcajul exterior era <InfCurs></InfCurs>. Acest marcaj definește o structură de date care este descrisă în interior, de marcajele <Titlu></Titlu> și respectiv <Detalii></Detalii>. Aceste denumiri va trebui să le respectăm și pentru noua clasă.

Obiectele acestei clase vor trebui serializate în și din fișiere .xml. Vă mai amintiți ce trebuie să facem ca să permită lucrul cu fișiere și serializarea? Evident, să includem namespace-urile corespunzătoare. Deci, în noua clasă vom include

```
...
using System.Web.UI.HtmlControls;
using System.IO;
using System.Xml.Serialization;

namespace web1
{
    public class InfCurs
    {
    }
}
```

Acum să adăugăm structura de date a clasei și metodele care manipulează datele private:

```
public class InfCurs
{
    private DateTime _data;
    private String _titlu;
    private String _detalii;

    [XmlIgnore()]
    public DateTime Data
    {
        get
        {
            return _data;
        }
        set
        {
            _data = value;
        }
    }
    public String Titlu
    {
        get
        {
            return _titlu;
        }
        set
        {
            _titlu = value;
        }
    }
    public String Detalii
    {
        get
```

```

    {
        return _detalii;
    }
    set
    {
        _detalii = value;
    }
}
}

```

Ce am făcut? Întâi am declarat trei variabile private în structura de date, conform informațiilor pe care dorim să le afișăm. Apoi, am generat pentru fiecare din cele 3 atribute funcțiile `get()` și `set()` care permit returnarea lor spre clasa apelantă, respectiv completarea valorii lor. `value` este o mărime predefinită în mediu, pentru transferul de date. Observați că metodele `get()` și `set()` sunt prefixate de linii de tipul

```
public Tip Nume
```

unde `Nume` este numele tag-ului corespunzător din fișierul `.xml` iar `Tip` tipul câmpului asociat în structura de date a clasei.

Linia `[XmlIgnore()]` specifică faptul că următoarele linii (deci implementarea metodelor) sunt excluse de la serializare când sunt serializate obiectele clasei.

Să revenim acum la `Global.asax` și să-i spunem și acestei clase că trebuie să lucreze cu fișiere și să permită serializarea:

```

using System.Web.SessionState;
using System.IO;
using System.Xml.Serialization;

```

Să adăugăm acum acestei clase metoda care realizează efectiv deserializarea fișierului `.xml` și încărcarea datelor acestuia în obiectele `InfCurs`:

```

public static InfCurs IncarcaCurs(String numfis)
{
    String calefis = CaleDirCurs + "\\numfis;
    FileStream fi = new FileStream(calefis, FileMode.Open);
    XmlSerializer ser = new XmlSerializer(typeof(InfCurs));
    InfCurs cursNou = (InfCurs)ser.Deserialize(fi);
    fi.Close();
    return cursNou;
}

```

Metoda returnează un obiect **`InfCurs`** cu câmpurile din structura de date completate prin deserializarea informațiilor din fișierul `.xml`. Modul de implementare al metodei vă este deja cunoscut, din capitolul 1.

Metoda construiește șirul `calefis` în care salvează calea completă și numele fișierului din care se face deserializarea. Apoi construiește un `FileStream` asociat acestui fișier pe care-l deschide în citire și un nou deserializator. Spre deosebire de decoderele binare utilizate în capitolul 1, acest deserializator este construit pentru formatul XML ca intrare, respectiv `InfCurs` ca ieșire. Realizează deserializarea efectivă și returnează obiectul rezultat.

Acum urmează să construim interfața pe care vom afișa datele. Să revenim în sursa Designer-ului și să adăugăm 2 noi tag-uri <div>:

```
<div>
    Cursuri urmate:
</div>
<br>
<div></div>
<div></div>
<br>
```

Acum în Designer să adăugăm 2 noi controale **Label**, pentru care să schimbăm proprietățile **ID** în **etTitlu** și **etDetalii** (fig. 2.7).

Să apăsăm dublu click în pagina Designer-ului astfel încât să revenim la codul care implementează evenimente și să modificăm metoda **Page_Load()** ca mai jos:

```
protected void Page_Load(object sender, EventArgs e)
{
    ...
    else
        Curs.Text = "Programare avansata in C# - examen in anii 2007 - " + an;
        Cale.Text = " Calea spre fisier: " + Global.CaleDirCurs;
        InfCurs crs = Global.IncarcaCurs("unu.xml");
        etTitlu.Text = crs.Titlu;
        etDetalii.Text = crs.Detalii;
}
```

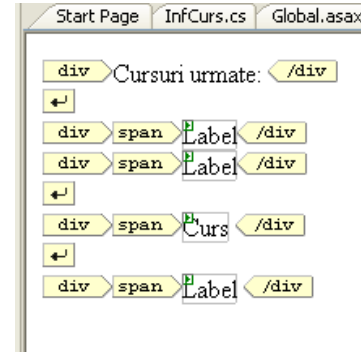


Figura 2.7

Ce face metoda? Afișează calea spre fișier în eticheta în care anterior afișa calea spre pagina **Default.aspx**. Apoi declară un obiect **InfCurs** al cărui structură de date o încarcă cu informația din fișierul **unu.xml** prin apelul metodei **IncarcaCurs()**. Apoi câmpurile **Titlu** și **Detalii** sunt afișate în controalele **Label** asociate.

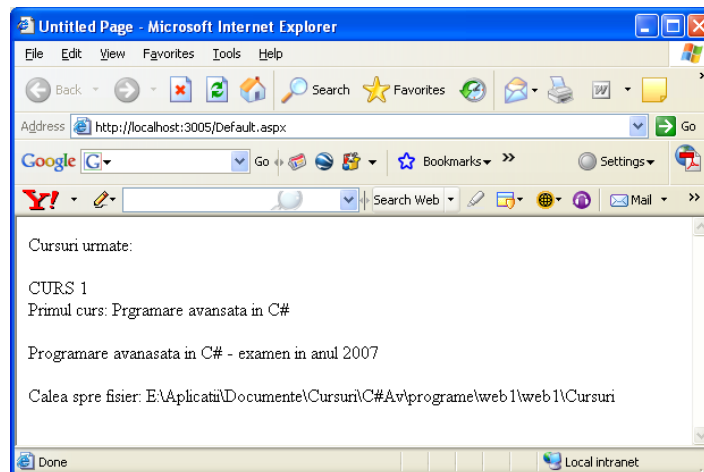


Figura 2.8

Observație: *Observați că deși câmpurile structurii de date din clasa **InfCurs** sunt **_titlu** și **_detalii**, am utilizat pentru obiectul **crs** denumirile **Titlu** și **Detalii**. Primele denumiri sunt denumiri interne clasei, acestea fiind făcute vizibile la nivelul interfeței automat, prin intermediul metodelor **get()** și **set()** sub denumirile publice.*

Compilați și executați programul. Ar trebui să obțineți ceva similar fig. 2.8.

Dacă doriți să verificați că într-adevăr informația este preluată din fișierul **unu.xml**, nu aveți decât să schimbați conținutul acestuia și să refreșați pagina.

Să completăm acum și câmpul **Data**. Pentru aceasta, în **Global.asax** să modificăm din metoda **IncarcăCurs()**:

```
public static InfCurs IncarcaCurs(String numfis)
{
    ...
    fi.Close();
    cursNou.Data = new FileInfo(calefis).LastWriteTime;
    return cursNou;
}
```

Ce am adăugat? Am creat un obiect **FileInfo** pentru fișierul în care sunt stocate datele și am atribuit proprietatea **LastWriteTime** (timpul ultimei scrieri în fișier) câmpului **Data**. Și acum să afișăm:

```
protected void Page_Load(object sender, EventArgs e)
{
    ...
    etTitlu.Text = crs.Titlu;
    etDetalii.Text = crs.Data.ToString("dddd") + ", " +
                   crs.Data.ToLongDateString() + " - " + crs.Detalii;
}
```

Compilați și executați programul.

2.3 Să îmbunătățim aspectul paginii

Metoda cea mai simplă de creare a unui aspect plăcut pentru pagină este utilizarea cascade style sheet (css). Acest mecanism îl cunoașteți de la cursul de pagini web, așa ca nu vom intra în amănunte. Primul pas pe care trebuie să îl facem este să adăugăm o nouă pagină de stiluri. Pentru aceasta, apăsăm click dreapta pe rădăcină în Solution Explorer, **Add**, **NewItem**, și selectăm **Style Sheet**. Să dăm numele pentru această pagină de stiluri **Stil.css**.

Observație: *Puteți folosi o pagină de stiluri existente. Pentru aceasta., apăsați click dreapta în Solution Explorer, **Add**, **ExistingItem**, selectați fișierul **.css** dorit și apoi apăsați butonul **Add**.*

Să completăm pagina de stiluri ca mai jos:

body

```

{
    padding-right: 0px;
    padding-left: 0px;
    font-size: 8pt;
    padding-bottom: 0px;
    margin: 0px;
    padding-top: 0px;
    font-family: Verdana, Arial;
}
.header
{
    padding-right: 5px;
    padding-left: 5px;
    padding-bottom: 10px;
    padding-top: 10px;
    background-color: #000099;
    font-weight: bold;
    font-size: 14pt;
    color: White;
}
.normal
{
    padding-right: 5px;
    padding-left: 5px;
    font-size: 8pt;
}
.normalHeading
{
    padding-right: 5px;
    padding-left: 5px;
    font-size: 12pt;
    font-weight: bold;
}
.crsTitlu
{
    padding-right: 5px;
    padding-left: 5px;
    padding-bottom: 1px;
    padding-top: 1px;
    font-weight: bold;
    font-size: 10pt;
    color: White;
    background-color: #66cc99;
}
.crsData
{
    font-weight: bold;
    color: #333399;
    font-size: 8pt;
}
.crs
{
    padding-right: 5px;
    padding-left: 5px;
    padding-top: 2px;
    font-size: 8pt;
}

```


Tot ce mai avem de făcut este să aplicăm stilul fișierului HTML. Să intrăm în sursa Designer-ului și să modificăm fișierul ca mai jos:

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Default.aspx.cs" Inherits="web1._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <link rel="stylesheet" href="Stil.css" />
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <div class="header">
            Cursuri urmate:
        </div>
        <br>
        <div class="crsTitlu">
            <asp:Label ID="etTitlu" runat="server" Text="Label"></asp:Label></div>
        <div class="crs">
            <asp:Label ID="etDetalii" runat="server"
Text="Label"></asp:Label></div>
        <br />
        <hr color=#000000 />
        <div class="normal">
            <asp:Label ID="Curs" runat="server"
Text="Curs"></asp:Label>&nbsp;</div>
            <br>
            <div class="normal">
                <asp:Label ID="Cale" runat="server" Text="Label"></asp:Label>
            </div>
        </form>
    </body>
</html>
```

Compilați și executați programul. Veți obține rezultatul din fig. 2.9.

2.4 Afișarea unei colecții de informații externe

Să vedem acum ce se poate face dacă avem mai multe fișiere .xml care conțin date. Pentru început, vom șterge controalele **Label** etTitlu și etDetalii precum și tag-urile <div> aferente și vom adăuga în locul lor un control DataList (îl găsiți la secțiunea Data al Toolbox), ca în fig. 2.10. Acesta este un control care permite adăugarea în interiorul lui al altor controale. De asemenea, permite

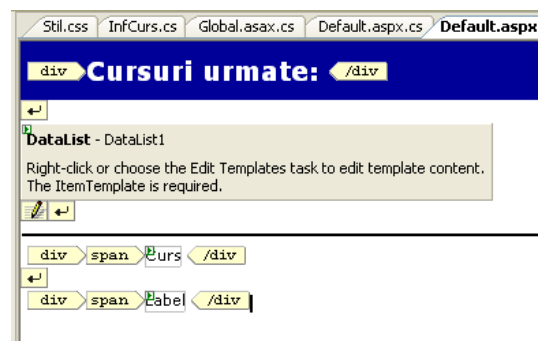


Figura 2.10

iterarea informațiilor din controalele din interior pentru toate datele de tipul specificat

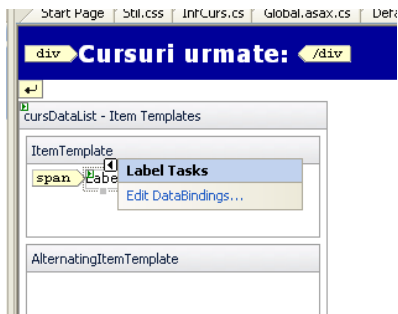


Figura 2.11a

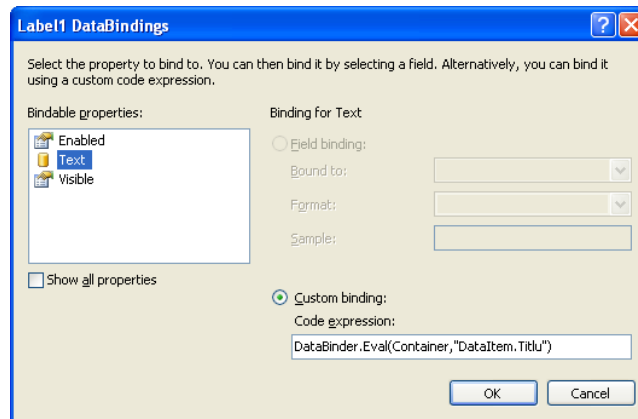


Figura 2.11b

existente. Să modificăm proprietatea **ID** pentru acest control în `datalistCurs`. Acum, să adăugăm controale în interiorul lui. Pentru aceasta, apăsăm click dreapta pe control și în meniul contextual alegem `Edit Template` și `Item Templates`. Controlul va fi afișat sub formă tabelară, permițând adăugarea altor controale. Să tragem acum un control `Label` în secțiunea `Item Templates`. Pentru a respecta stilul de afișare, să modificăm proprietatea **CssClass** al controlului `Label` în `crsTitlu`. Acum va trebui să precizăm ce se afișează în acest control. Pentru aceasta, apăsăm pe săgeata din partea dreaptă-sus a controlului și alegem `Edit Data Bindings` (fig. 2.11a). Va fi afișată caseta de dialog din fig. 2.11b. În aceasta, avem grijă ca la **Bindable Properties**: să fie selectată proprietatea `Text` și apoi introducem în caseta `Code Expression` textul:

`DataBinder.Eval(Container, \"DataItem.Titlu\")`

Acest text precizează că în faza de completare al controlului, proprietății `Text` al acestuia i se va atribui `DataItem.Titlu`, adică câmpul `Titlu` din obiectul conectat la control. Urmează să adăugăm încă 2 controale `Label`, ca în figura 2.12. Pentru controlul `Label2` vom modifica proprietatea `CssClass` în `crsData` și vom introduce în caseta `Code Expression` textul

`DataBinder.Eval(Container, \"DataItem.DataInSir\")`

iar pentru controlul `Label3`, vom modifica proprietatea `CssClass` în `crs` și vom introduce textul

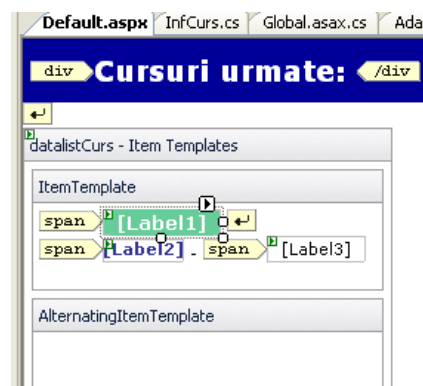


Figura 2.12

`DataBinder.Eval(Container, \"DataItem.Detalii\")`

Cu alte cuvinte, am precizat pentru fiecare control stilul de afișare luat din pagina de stiluri și respectiv cu cine este conectat. O mică problemă apare la controlul

Label2, pentru care conectarea nu se face la un câmp cunoscut al clasei **InfCurs**. Despre aceasta vom vorbi în cele ce urmează.

Va trebui acum să adăugăm în fișierul Global.aspx o metodă care să permită încărcarea tuturor fișierelor din directorul cu date:

```
public static InfCurs[] IncToateC()
{
    DirectoryInfo subdirector = new DirectoryInfo(CaleDirCurs);
    FileInfo[] fisiere = subdirector.GetFiles();
    InfCurs[] cursuri = new InfCurs[fisiere.Length];
    int index = 0;
    foreach (FileInfo fis in fisiere)
    {
        cursuri[index] = IncarcaCurs(fis.Name);
        index++;
    }
    return cursuri;
}
```

Cum lucrează metoda? Pentru început crează un obiect **DirectoryInfo**, numit subdirector, care reprezintă tocmai directorul care conține fișierele. Apoi apelează pentru acesta metoda **GetFiles()**. Să ne reamintim că această metodă încarcă un șir de obiecte **FileInfo** cu fișierele din director (vezi cap. 1). Declară apoi un șir de obiecte **InfCurs**, de dimensiune egală cu dimensiunea șirului de obiecte **FileInfo**, după care încarcă șirul de obiecte **InfCurs** cu informațiile din fiecare fișier în parte, apelând metoda **IncarcaCurs()** pentru deserializarea fiecărui fișier. În final, funcția returnează șirul de obiecte **InfCurs**.

Practic, metoda va prelua fiecare fișier .xml din director, va deserializa informația și o va salva în câte un obiect **InfCurs** din șir. Astfel, vom avea disponibilă pentru afișare informația din toate fișierele.

Acum, în funcția **Page_Load()**, va trebui să eliminăm secțiunea care asociază câmpurile obiectului fostelor controale **Label** și să asociem câmpurile tuturor obiectelor obținute prin deserializare controlului **DataList**.

Astfel, vom șterge liniile

```
InfCurs crs = Global.IncarcaCurs("unu.xml");
etTitlu.Text = crs.Titlu;
etDetalii.Text = "<font class=crsData>" + crs.Data.ToString("dddd")+ ",
                "+crs.Data.ToLongDateString()+ "</font> - " + crs.Detalii;
```

și în locul lor vom adăuga liniile:

```
protected void Page_Load(object sender, EventArgs e)
{
    int an = DateTime.Now.Year;
    ...
    Cale.Text = " Calea spre fisier: " + Global.CaleDirCurs;

    InfCurs[] cursuri = Global.IncToateC();
    dataListCurs.DataSource = cursuri;
```

```

    datalistCurs.DataBind();
}

```

Ce am făcut? Am încărcat șirul de obiecte **InfCurs** cursuri cu informația deserializată, am precizat obiectului **DataList** sa-și ia informațiile din acest șir de obiecte **InfCurs** (ca sursă) și să itereze pentru toate aceste obiecte și am conectat cele 2 obiecte între ele (**datalistCurs** și **cursuri**).

Ce mai avem de făcut? Să implementăm proprietatea **DataInSir** pentru obiectele **InfCurs**. De ce e nevoie de aceasta. Până acum data creării fișierelor o afișam separat, în metoda **Page_Load()**. Deoarece acum vom manipula mai multe fișiere, data creării va trebui să fie o proprietate intrinsecă a obiectelor. Cum facem asta? Simplu. Adăugăm clasei **InfCurs** codul:

```

public String DataInSir
{
    get
    {
        return Data.ToString("dddd") + ", " + Data.ToLongDateString();
    }
}

```

Proprietatea va putea fi doar citită (expune doar metoda **get()**). Proprietatea va conține data curentă sub forma unui șir de caractere.

Compilați și executați programul.

2.5 Adăugarea fișierelor .xml

Până acum am completat fișierele .xml manual. Putem ușor crea o pagină care să le creeze automat, prin mecanismul de serializare a obiectelor **InfCurs**. Pentru aceasta, la început, va trebui să adăugăm proiectului o nouă pagină web: click dreapta pe rădăcină în **Solution Explorer**, **Add, New Item** și respectiv **Web Form**. Să denumim această nouă formă **Adaugare.aspx**. Să completăm codul HTML pentru această pagină ca mai jos:

```

<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Adaugare.aspx.cs" Inherits="web1.Adaugare" %>

...
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
<link rel="stylesheet" href="Stil.css" />
    <title>Adaugare</title>
</head>
<body>
    <form id="form1" runat="server">
        <div class="normalHeading">
            Adaugare curs
        </div>
        <br />
        <div class="normal">

```

```

<table cellpadding="0" cellspacing="3">
<tr>
<td class="normal">
    Titlu:
</td>
<td>
</td>
</tr>
<tr>
<td class="normal">
    Detalii:
</td>
<td>
</td>
</tr>
<tr>
<td colspan="2" align="right">
</td>
</tr>
</table>
</div>
</form>
</body>
</html>

```

Ar trebui să obțineți în Designer imaginea din fig. 2.13. Acum, să tragem două controale **TextBox** în coloanele 2 din tabel (casetele mici din dreapta textelor Titlu și Detalii). Să modificăm proprietatea **ID** al controlului TextBox de sus în textTitlu și a controlului de jos în textDetalii. De asemenea, pentru controlul de jos, să modificăm proprietatea **TextMode** în MultiLine.

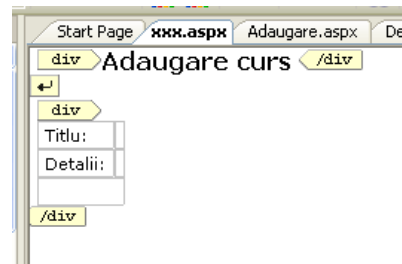


Figura 2.13

Să adăugăm acum în ultima line (cea fără coloane) un control **Button**, pentru care să modificăm proprietatea **ID** în OK și proprietatea **Text** în Salvare.

Ar mai fi de făcut. Deoarece în general introducerea de texte implică și valiări, să validăm cele 2 controale de tip **TextBox** ca să nu permită salvarea schimbărilor decât atunci când în ambele controale este introdus text. Pentru aceasta, în secțiunea Validation al Toolbox, găsiți un control **RequiredFieldValidator**

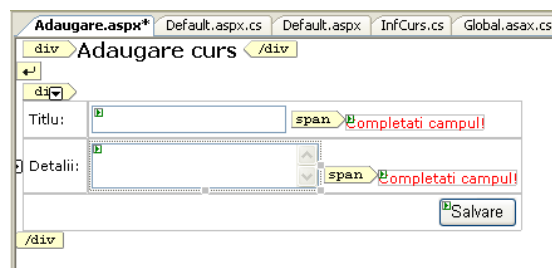


Figura 2.14

și trageți 2 astfel de controale în dreapta fiecărui control TextBox. Pentru ambele controale, completați proprietatea **ErrorMessage** cu "Completati campul!", iar proprietatea **ControlToValidate** cu textTitlu respectiv textDetalii (fig. 2.14).

Va trebui acum să furnizăm numele fișierului în care vor fi salvate informațiile din clasa **InfCurs**. Pentru aceasta, să intrăm în codul acestei clase și să completăm următoarele informații:

```
public class InfCurs
```

```

{
    private String _fisier;
    private DateTime _data;
    ...
    public String DataInSir
    {
        get
        {
            return Data.ToString("dddd") + ", " + Data.ToLongDateString();
        }
    }
}
public String Fisier
{
    get
    {
        return _fisier;
    }
    set
    {
        _fisier = value;
    }
}
...

```

Am adăugat o proprietate internă care va stoca numele fișierului și modul de acces la această proprietate. Acum urmează să adăugăm clasei metoda care realizează serializarea informației în fișierul cu numele specificat de proprietatea **Fisier**:

```

public void Salvare()
{
    if (Fisier == null)
        Data = DateTime.Now;
    DirectoryInfo subdirector = new DirectoryInfo(Global.CaleDirCurs);
    FileInfo[] fisiere = subdirector.GetFiles();
    Fisier = String.Format("{0:d2}{1:d2}{2:d4}_{3:d3}.xml",
        (int)Data.Day, (int)Data.Month, (int)Data.Year,
        fisiere.Length);
    String cale = Global.CaleDirCurs + "\\\" + Fisier;
    FileInfo fi = new FileInfo(cale);
    if (fi.Exists == true)
        fi.Delete();
    FileStream fs = new FileStream(fi.FullName, FileMode.Create);
    XmlSerializer ser = new XmlSerializer(this.GetType());
    ser.Serialize(fs, this);
    fs.Close();
}

```

Primul lucru pe care trebuie să-l facem este să atribuim un nume fișierului. Acest nume trebuie să fie unic, dacă numele există deja, vom șterge fișierul cu numele respectiv și vom crea unul nou. Numele fișierului este de forma **zzllaaaa_nr.xml**, unde primele 8 caractere reprezintă data curentă (preluată cu `DateTimeNow`) iar ultimele 3 al cătelea fișier din ziua respectivă este cel tocmai creat. Creăm apoi obiectul **DirectoryInfo** pentru calea pe care se află fișierele și aflăm numele tuturor fișierelor existente deja în acel subdirector. Compunem numele complet al fișierului prin adăugarea căii. Dacă deja există un fișier cu acest nume îl ștergem și deschidem

un nou fișier în mod Create. Declarăm un serializator XML care serializează obiecte de tipul **InfCurs** (`XmlSerializer ser = new XmlSerializer(this.GetType());`) și serializăm obiectul curent în fișier. Cu aceasta, o nouă intrare de tip XML este creată.

Acum, ca de obicei, va trebui să modificăm funcția `Page_Load()` în așa fel încât să fie luate în considerare modificările din clasa **InfCurs**. Nu vom modifica metoda `Page_Load()` din pagina `Default.aspx`, ci metoda din pagina `Adaugare.aspx`, pentru că aceasta se ocupa de crearea fișierelor și serializarea informației:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack == true)
    {
        InfCurs curs = new InfCurs();
        curs.Titlu = textTitlu.Text;
        curs.Detalii = textDetalii.Text;
        curs.Salvare();
        Response.Redirect("Default.aspx");
    }
}
```

Ce face metoda? Pentru început, testează metoda **IsPostBack** pentru pagina `Adaugare.aspx`. Această proprietate este `true` dacă pagina a fost încărcată ca urmare a cererii utilizatorului (apăsarea butonului `Salvare`, în cazul nostru) și `false` dacă a fost încărcată pentru prima dată. Dacă da, se crează un nou obiect **InfCurs**, ale cărui câmpuri sunt completate din casetele de tip `TextBox`. Apoi este apelată metoda `Salvare()` pentru crearea fișierului și serializarea obiectului și se navigează înapoi în pagina `Default.aspx`, prin apelarea `Response.Redirect()`.

Tot ce mai avem de făcut este să creăm un link în pagina `Default.aspx` către pagina `Adaugare.aspx`.

```
...
</asp:DataList>
<br />
<div class="Normal">
    <a href="Adaugare.aspx"> Adaugare curs</a>
</div>
<br />
<br />
<hr color=#000000 />
<div class="normal">
    <asp:Label ID="Curs" runat="server"
Text="Curs"></asp:Label>&nbsp;</div>
<br>
<div class="normal">
...
```

Compilați și executați programul. Observați că puteți crea fișiere în format XML fără să mai fie nevoie să le completăm manual. De fapt, care este mecanismul? Când apăsăm butonul `Salvare`, se petrec 2 lucruri. Întâi, codul de validare validează informația din controalele din pagină, adică testează dacă există informație în fiecare din controalele **TextBox**. Dacă validările sunt realizate, pagina este încărcată. O dată

cu încărcarea paginii, este lansată în execuție metoda `Load_Page()`, care crează un nou obiect **InfCurs**, deschide un fișier cu numele prezentat anterior și serializează obiectul în acel fișier. După care, se revine la pagina principală.

2.6 Editarea fișierelor .xml

Ar fi normal, ca un fișier XML odată creat, să poată fi editat. În cele ce urmează vom implementa mecanismul de editare al acestor fișiere. Pentru aceasta:

- intrăm în Designer în pagina `Default.aspx`.
- selectăm controlul **DataList**, să facem click dreapta pe el și în meniul contextual să alegem `Edit Templates` și `Item Template`.
- adăugăm un control `Hyperlink` în dreapta etichetei `Label1`.
- pentru acest control, modificăm proprietatea **cssClass** în `curs` și proprietatea **Text** în `Editare`.
- apăsăm click pe săgeata din partea dreaptă sus al controlului și selectăm din tabel `Edit DataBindings`.
- selectăm proprietatea **NavigateURL** și în caseta `Code expression`: inserăm textul:

`"adaugare.aspx?Fisier="+DataBinder.Eval(Container,"DataItem.Fisier")`

- în `Global.asax`, în metoda `IncarcaCurs()`, inserăm linia:

```
public static InfCurs IncarcaCurs(String numfis)
{
    ...
    cursNou.Fisier = numfis;
    return cursNou;
}
```

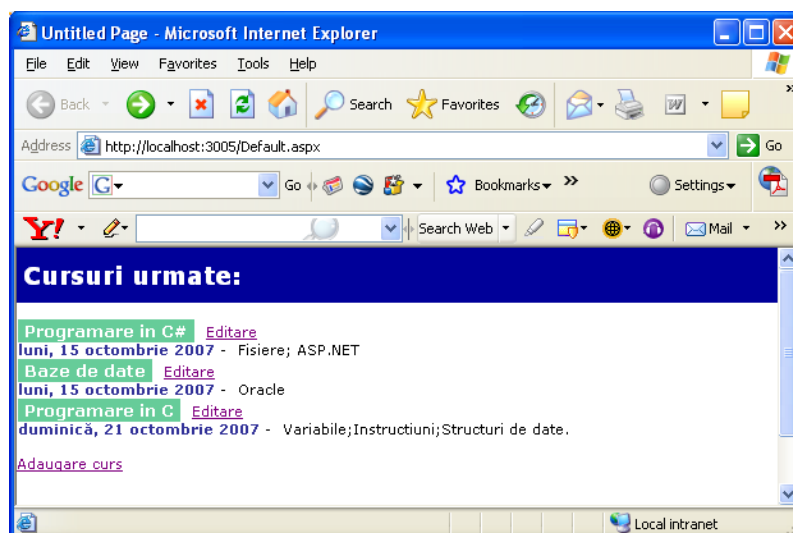


Figura 2.15

Acum să compilăm și să executăm programul. Observăm că obținem rezultatul din fig. 2.15.

Ce se observă? În dreapta fiecărei intrări, a apărut un link, cu textul Editare. Acest text este cel introdus în proprietatea **Text** al controlului **Hyperlink** și apare cu stilul precizat în proprietatea **cssClass**. Să apăsăm acum pe link-ul asociat unui fișier. De exemplu, dacă apăsăm pe editarea primului fișier, vom obține rezultatul din fig. 2.16.

Observăm că am intrat din nou în pagina Adaugare.aspx. Nu acesta este lucrul interesant. Să privim mai atent la ce scrie în zona de adresă a browserului (fig. 2.17):

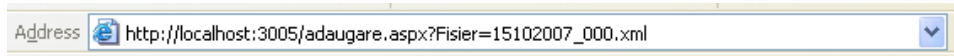


Figura 2.16

Dacă ne uităm cu atenție, observăm că în textul scris în secțiunea **NavigateURL** a controlului **Hyperlink**, componenta **DataSourceID** este setată la **DefaultDataSourceID**. În acest caz, **DataSourceID** este o proprietate a controlului **Hyperlink** care este setată la valoarea **DefaultDataSourceID** a controlului **Page**. În acest caz, **DataSourceID** este o proprietate a controlului **Hyperlink** care este setată la valoarea **DefaultDataSourceID** a controlului **Page**. Astfel, fișierul a fost deschis și s-a afișat pagina de adăugare. Este evident că pentru editarea conținutului fișierului, casetele din pagina de adăugare nu trebuie să fie goale, ca și în cazul adăugării unui nou fișier, ci trebuie să conțină informația deserializată din fișier. Pentru aceasta va trebui să modificăm metoda **Page_Load()** a paginii **Adaugare.aspx**:

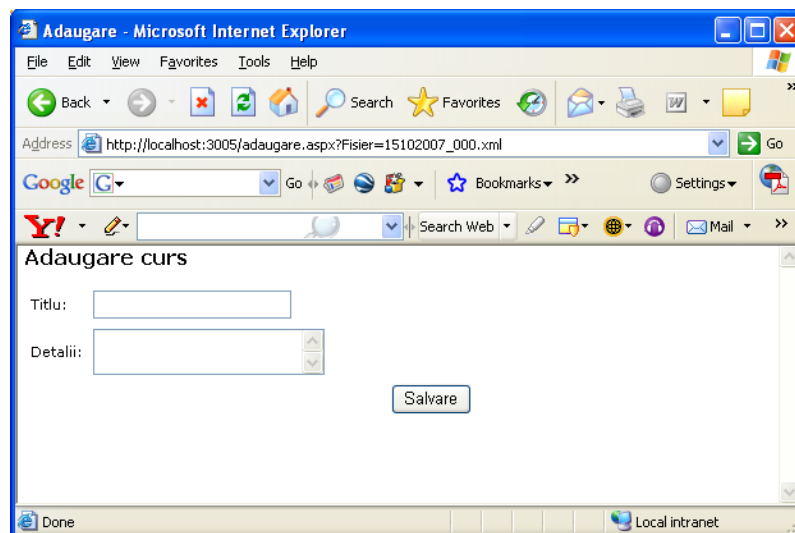


Figura 2.16

```
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack == true)
    {
        InfCurs curs = new InfCurs();
        if (Request.Params["Fisier"] != null)
            curs.Fisier = Request.Params["Fisier"];
        curs.Titlu = textTitlu.Text;
        curs.Detalii = textDetalii.Text;
        curs.Salvare();
        Response.Redirect("Default.aspx");
    }
    else
    {

```

```

    {
        String numfis=Request.Params["Fisier"];
        if (numfis!=null)
        {
            InfCurs curs=Global.IncarcaCurs(numfis);
            textTitlu.Text=curs.Titlu;
            textDetalii.Text=curs.Detalii;
        }
    }
}

```

Să vedem întâi ce se întâmplă pe ramura else. În acest caz, știm că **IsPostBack** este false, deci pagina a fost încărcată pentru prima dată.

Proprietatea **Request.Params** poate fi utilizată pentru a accesa parametrii scriptați în pagină, adică acei parametri care au fost trecuți paginii pentru a fi afișați în procesul de scriptare. Ceea ce ne interesează pe noi în acest moment, este numele fișierului selectat, adică conținutul proprietății **Fisier** al obiectului **InfCurs** curent. Este proprietatea pe care o vom transmite spre **Request.Params**. În mod evident, odată obținut numele fișierului, vom apela metoda **IncarcăCurs()** pentru deserializarea conținutului acestuia și vom afișa informațiile deserializate în casele de tip **TextBox**.

Dar de ce am modificat și ramura corespunzătoare **IsPostBack==true**? Pentru că va trebui să aflăm dacă avem vreun fișier deja selectat (**Request.Params !=null**). În acest caz, vom folosi din nou numele acestuia pentru a specifica fișierul în care se face salvarea în caz contrar (nu am ajuns în pagină prin Editare), metoda **Salvare()** va reconstitui numele fișierului din data curentă. Dar pentru aceasta, trebuie să rescriem această metoda ca mai jos:

```

public void Salvare()
{
    if (Fisier == null)
    {
        Data = DateTime.Now;
        DirectoryInfo subdirector = new DirectoryInfo(Global.CaleDirCurs);
        FileInfo[] fisiere = subdirector.GetFiles();
        Fisier = String.Format("{0:d2}{1:d2}{2:d4}_{3:d3}.xml",
            (int)Data.Day,(int)Data.Month, (int)Data.Year,
fisiere.Length);
    }
    String cale = Global.CaleDirCurs + "\\\" + Fisier;
    FileInfo fi = new FileInfo(cale);
    if (fi.Exists == true)
        fi.Delete();
    FileStream fs = new FileStream(fi.FullName, FileMode.Create);
    XmlSerializer ser = new XmlSerializer(this.GetType());
    ser.Serialize(fs, this);
    fs.Close();
}

```

Ce am făcut? Am șters liniile cu bold din locul lor și le-am inserat în if. Aceasta pentru că este nevoie de completarea numelui fișierului doar dacă nu este preluat deja prin mecanismul descris anterior.

Compilați și executați programul.

2.7 Restricționarea accesului la pagina de web

Până în acest moment, orice utilizator poate accesa pagina de web, putând atât să vizualizeze informația. Cât și să adauge noi fișiere și respectiv să le editeze. În mod normal, secțiunea de adăugare și respectiv editare trebuie să fie protejate.

Să implementăm acum o metodă prin care utilizatorul se poate autentifica înainte de a-i fi permise anumite operații, respectiv adăugarea și editarea. Acest mecanism se bazează pe noțiunea de *Sesiune*.

O sesiune permite serverului de web să asocieze un set de date cu un anumit tip de browser. Prin aceasta, browserul rămâne, hai să spunem “agățat” în prima pagină pe care o afișează. Informația utilizată pentru aceasta este unică și este utilizată ca și cheie de validare pentru accesul la următoarele pagini.

Serverul de web “agață” browserul prin utilizarea unui **cookie**. Un cookie este o secvență de cod de dimensiune mică, trimisă de server clientului la accesarea unei pagini de web și care este citită de fiecare dată când pagina de web este reînărcată. LA primul apel al paginii de web, cookie-ul nu există, astfel încât serverul de web va crea o nouă sesiune și va plasa ID-ul acelei sesiuni într-un cookie, pe care-l va trimite apoi browserului. În principiu, se poate stoca diferite informații într-un cookie, dar deoarece cu cât mai multa informație este stocată, cu atât mai multă memorie este necesară la server, uzual cookie-ul conține un număr minim de informații.

Bazat pe acest mecanism, să încercăm să restricționăm accesul în adăugare și editare. Informația stocată în cookie este suficient să fie o simplă valoare boolean. Care să precizeze co utilizatorul are sau nu acces la aceste pagini.

Orice sesiune este lansată la apelul primei pagini dintr-o cerere și începe prin lansarea în execuție a unei funcții numite `Session_Start()`, localizată în `Global.asax`. Să adăugăm această funcție:

```
protected void Session_Start(Object sender, EventArgs e)
{
    Session["permis"] = false;
}
```

Proprietatea **Session** permite adăugarea de variabile, numite *variabile de sesiune*. În funcție, noi am adăugat o variabilă numită `permis`, căreia i-am atribuit valoarea `false`. Deci, la încărcarea paginii `Default.aspx`, această variabilă va fi făcută implicit `false`.

Să adăugăm acum proiectului o nouă pagină de web, pe care o vom numi `Interzis.aspx`. Să completăm codul acesteia cu următoarele linii:

```
@ Page Language="C#" AutoEventWireup="true" CodeBehind="Interzis.aspx.cs"
Inherits="web1.Interzis" %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      Ne pare rau. Nu aveti dreptul sa accesati aceasta pagina
      <br />
      <a href="Default.aspx">Continuare</a>
    </div>
  </form>
</body>
</html>
```

Acesată pagină va trebui să fie afișată dacă se accesează pagina Adaugare.aspx (deci în adăugare sau editare) fără ca utilizatorul să fie autentificat, în cazul nostru dacă permis==false. Va trebui deci să testăm valoarea permis în metoda Page_Load() din Adaugare.aspx:

```
protected void Page_Load(object sender, EventArgs e)
{
  if ((bool)Session["permis"] == false)
    Response.Redirect("Interzis.aspx");
  if (IsPostBack == true)
    ...
}
```

Este simplu de înțeles. Dacă variabila permis are valoarea false, se afișează pagina Interzis.aspx.

Acum, tot ce mai avem de făcut este să realizăm autentificarea utilizatorului. Pentru aceasta, să adăugăm o nouă pagină web, numită Login.aspx. În Designer, să adăugăm pe această pagină un control **TextBox** și un control **Button** (fig. 2.17). Pentru controlul **TextBox** să schimbăm proprietatea **ID** în textParola și **TextMode** în Password, iar pentru controlul **Button** proprietatea **ID** în btnLogin și **Text** în Login. Să apăsăm acum dublu-click în Designer pentru a intra în zona de cod și să completăm metoda Page_Load() ca mai jos:

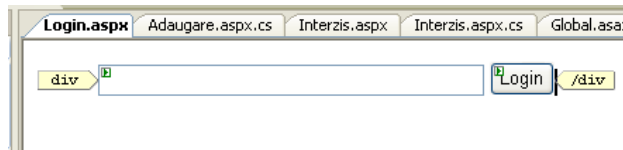


Figura 2.17

```
protected void Page_Load(object sender, EventArgs e)
{
  if (IsPostBack == true)
  {
    if (textParola.Text.CompareTo("nabucodonosor") == 0)
      Session["permis"] = true;
    Response.Redirect("Default.aspx");
  }
}
```

Ce face metoda? Compară textul din controlul textParola cu "nabucodonosor". Dacă coincide, setează valoarea variabilei permis la true. Aceasta va avea ca efect permiterea adăugării și editării de fișiere. Apoi, se revine în pagina Default.aspx.

Tot ce mai avem de făcut, este să creăm în pagina principală un link spre această pagină. Deci, în Default.aspx adăugăm codul:

```
...  
<br />  
<div class="Normal">  
  <a href="Adaugare.aspx"> Adaugare curs</a>  
</div>  
<div class="Normal">  
  <a href="Login.aspx"> Login</a>  
</div>  
<br />  
...
```

Cu aceasta aplicația este finalizată. Compilați și executați programul.