Ingineria Programării și Limbaje de Asamblare

Note de curs

Marian Dârdală

Cristian Ioniță

ianuarie 2016

1. Aspecte generale referitoare la framework-uri și mediul .NET

1.1 Framework-uri pentru dezvoltarea de aplicații software

Un framework software este o platformă software universală și reutilizabilă folosită la dezvoltarea aplicațiilor, produselor și soluțiilor software. Scopul unui framework este de a îmbunătății eficiența procesului de dezvoltare de noi aplicații software. Framework-ul poate imbunătății productivitatea procesului de dezvoltare de software și a: calității, fiabilității și robusteții noului software. Productivitatea dezvoltatorului este îmbunătățită prin faptul că acesta se concentrează pe cerințele specifice ale aplicației pe care o construiește în loc să consume timpul pentru a programa infrastructura aplicației.

Conceptul de framework este mai extins decât cel de bibliotecă. In cazul utilizării unei biblioteci, obiectele și metodele implementate sunt instanțiate și apelate în aplicația dezvoltatorului. Pentru acest lucru trebuie să se cunoască care obiecte și metode trebuie utilizate și apelate astfel încât aplicația să-și atingă scopul. Pe de altă parte, la utilizarea unui framework dezvoltatorul definește obiecte și implementează metode care particularizează aplicația sa și acestea sunt instanțiate și apelate prin intermediul framework-ului. In acest caz, framework-ul definește un flux de control pentru aplicație. Un mod obișnuit de a particulariza un framework constă în a suprascrie entitățile implementate de către acesta. Pentru alinierea la framework, programatorul poate să definească clase derivate din clasele lui, să implementeze interfețe definite de framework și să definească metode virtuale și clase abstracte din framework.

In funcție de destinație, framework-urile se pot clasifica în: framework-uri destinate dezvoltării aplicațiilor software cu caracter general, în diferite tehnologii, respectiv framework-uri pentru dezvoltarea de aplicații

software specializate. In general, indiferent de tipul de framework folosit, dezvoltarea aplicațiilor software necesită utilizarea unui mediu de dezvoltare care să fie capabil să coopereze cu framework-ul dorit.

Reprezentativ pentru categoria framework-urilor cu caracter general este framework-ul construit de *Microsoft*, care poartă numele de *.NET Framework*. Astfel, dacă se dorește construirea unei aplicații de tip *Windows Forms*, programatorul își poate defini propria interfață prin intermediul unei clase (*Class_Forma*) care va fi derivată din clasa *Form* a framework-ului .NET, după cum urmează:

```
class Class_Forma : Form
    {

    }
```

Prin moștenire se preiau proprietăți de bază pentru fereastră, astfel încât aceasta va avea iconiță, butoane pentru: minimizare, maximizare și închidere, după cum se poate observa în figura 1.4, precum și funcționalități implicite aferente butoanelor generate.

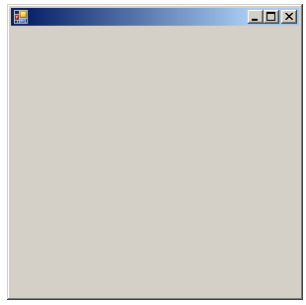

Figura 1.4 Forma generată implicit

Pentru a putea executa aplicația a fost definită și clasa *Principala* în care s-a definit metoda statică *Main*, necesară pentru startarea execuției aplicației. Practic codul din *Main* rulează o instanță a clasei *Class_Forma* definită anterior.

```
class Principala
    {
        static void Main()
        {
            Application.Run(new Class_Forma());
```

```
        }
    }
```

Pe lângă framework-urile cu caracter general, există și framework-uri care dau posibilitatea programatorului de a dezvolta aplicații software specializate. Astfel, produsele software specializate pe anumite domenii de prelucrare sunt construite pe baza acestor framework-uri. Se vor prezenta două astfel de framework-uri pentru a putea dezvolta aplicații bazate pe *Office* și *GIS*. Dezvoltatorii au la dispoziție mai multe variante de elaborare a modulelor si anume: să construiască aplicații personalizate într-un mediu de dezvoltare care să folosească framework-ul, să programeze diferite module personalizate pe care să le integreze în aplicațiile de bază (de exemplu *MS Word* sau *Excel*) folosind limbajul VBA (*Visual Basic for Application*) sau să folosească medii de dezvoltare pentru a construi extensii care să fie integrate ulterior în aplicațiile de bază și să conlucreze și cu celelalte elemente software ale produsului.

Există totodată și framework-uri care nu stau la baza unor produse software specializate ci au fost făcute pentru a dezvolta diferite tipuri de aplicații. De exemplu, *DirectX* are pe lângă versiunea *run-time* și versiunea pentru dezvoltare – *DirectX SDK*. In general, framework-urile pentru dezvoltare de aplicații au în denumire *SDK*, acronimul de la *Software Development Kit*.

Framework-urile permit proiectanților și dezvoltatorilor de soft să reutilizeze codul, să abstractizeze și să proiecteze noi funcționalități, în general sunt construite pentru utilizarea lor pe scară largă în dezvoltarea de aplicații specializate.

1.2 Mediul .NET – caracterizare generală

Microsoft a lansat prima versiune a lui .NET Framework în anul 2000. Principalele proprietăți ale mediului de programare .NET sunt:

- Mediu independent pentru execuția programelor, în sensul că toate limbajele se bazează pe același nucleu (de exemplu, nu există nici o

diferență în rularea unei aplicații indiferent dacă aceasta a fost scrisă în VB.NET sau C#);

- Mediu orientat obiect în formă pură, în sensul că implementează complet proprietățile programării orientate obiect și arhitectura framework-ului este orientată obiect;
- Integrare completă între aplicații, asigurată prin faptul că .NET Framework poate fi utilizat independent în raport de limbajele de programare;
- Interoperabilitate cu alte componente software existente, ceea ce permite aplicațiilor .NET să reutilizeze DLL-uri existente, componente COM, ActiveX etc;
- Model simplificat de livrare a aplicațiilor, astfel încât nu mai este necesară înregistrarea componentelor sau a aplicațiilor în regiștrii din Windows ci după compilarea componentelor acestea se copiază în directorul dorit și apoi se referă pentru a fi utilizate în alte aplicații;
- Model general de securitate, conținut în .NET Framework pentru a seta caracteristici de securitate aplicațiilor fără a rescrie aplicațiile. In timpul execuției unei aplicații framework-ul verifică setările astfel încât sa nu permită accesul neautorizat, procesarea unor operații interzise etc.

Din punct de vedere istoric, .NET Framework a cunoscut o evoluție continuă, de la lansarea lui, fiecare versiune a adăugat noi caracteristici sau capabilități, principalele fiind ilustrate în figura 1.2. Din punct de vedere arhitectural, .NET Framework conține următoarele componente:

- *Common Language Run-time* (CLR), care furnizează un nivel de abstractizare peste sistemul de operare. Partea cea mai importantă a CLR se află în fișierul *mscore.dll* și furnizează următoarele caracteristici:
    - Identificarea *assembly*-urilor necesare;
    - Încărcarea lor;

- Identificarea tipurilor corespunzătoare folosind metadatele *assembly*-ului;
- Translatarea codului *MSIL* (Microsoft Intermediate Language).

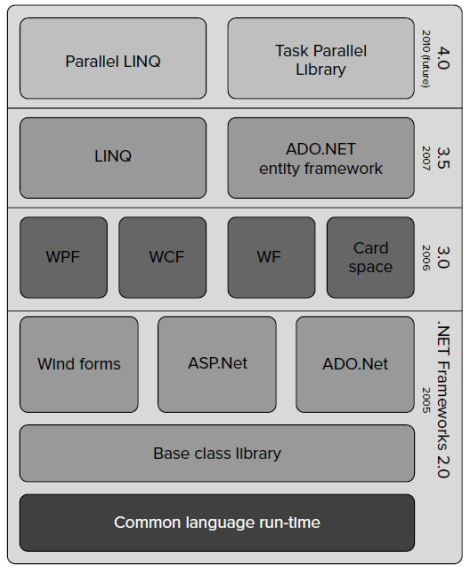

Figura 1.2 Evoluția .NET Framework[1]

- Biblioteca de clase de bază care se constituie din cod preconstruit pentru sarcini des utilizate în programarea de nivel scăzut. Principalul rol al acestei componente este de a furniza servicii de infrastructură dezvoltatorilor. Aceste servicii se constituie ca un strat peste interfața de programare WIN API. Directorul bibliotecii de clase conține o mulțime de *assembly*-uri;
- Framework-uri și tehnologii de dezvoltare, ce reprezintă soluții reutilizabile și care se pot personaliza pentru o largă varietate de sarcini de programare.

2. Utilizarea interfețelor și a evenimentelor în dezvoltarea de aplicații

2.1 Interfețe

O interfață se definește similar unei clase având următoarele particularități:

- în loc de folosirea cuvântului cheie *class* se folosește *interface*;
- nu poate conține atribute;
- proprietățile și metodele au doar semnăturile nu și implementarea.

Interfețele se folosesc pentru a implementa aceleași operații dar într-un alt context de lucru. Altfel spus, o clasă care implementează o interfață trebuie să implementeze metodele și proprietățile interfeței pentru a se alinia la un mod de utilizare. In acest sens, un exemplu sugestiv se referă la faptul că, în vederea implementării unei colecții se dorește ca elementele ei să fie iterate folosind instrucțiunea specifică *foreach*.

Se propune o implementare minimală a structurii de dată listă simplu înlănțuită ca fiind o colecție de noduri, nodurile fiind implementate tot printr-o clasă de sine stătătoare inclusă în clasa lista.

```
public class Lista
{ // clasa nod
       public class nod
       {
        object inf;
        public nod next;
        public nod(object k, nod urm) { inf = k; next = urm; }
        public object informatia
            {
            get { return inf; }
            set { inf = value; }
            }
       }
       nod cap;
       protected uint nrn;

       public Lista() { cap=null; nrn=0; }
```

```
        public void Adaugare(object k)
        {
                nod tmp = new nod(k, cap);
                nrn++;
                cap=tmp;
        }
}
```

Pentru colecţia definită prin intermediul clasei `Lista`, instrucţiunea `foreach` nu funcţionează. Acest lucru devine posibil făcând următoarele completări:

- clasa **`Lista`** va moşteni interfaţa **`IEnumerable`**;
- ca o consecinţă a punctului anterior este necesar să se definească metoda **`GetEnumerator()`**, în clasa `Lista`, după cum urmează:

```
public IEnumerator GetEnumerator()
               { return nrn==0 ? null : new ListEnum(cap); }
```

In cazul în care lista e vidă, metoda returnează `null`, altfel metoda întoarce un obiect care implementează interfaţa **`IEnumerator`** pentru o listă. Obiectul este de clasă **`ListEnum`** şi referă elementele listei prin intermediul capului listei (`cap`);

- astfel se defineşte clasa **`ListEnum`** care implementează interfaţa `IEnumerator`:

```
class ListEnum : IEnumerator
{
        Lista.nod aux, init;
        bool vb;
        public ListEnum(Lista.nod ccp) { aux = init = ccp; vb = true; }
        public object Current {  get { return aux.informatia; }  }
        public bool MoveNext()
        {
                if(aux==init && vb )  { vb=false; return true;}
                if (aux.next!=null)
                {
                        aux=aux.next;
                        return true;
                }
                else return false;
        }
        public void Reset() { aux=init; vb=true; }
}
```

Din codul sursă se observă următoarele:

- constructorul iniţializează membrii de tip **nod** cu capul listei şi variabila booleană **vb** cu **true**;
- metoda **Reset()** repune referinţa auxiliară **aux** pe începutul listei şi **vb** pe **true**;
- proprietatea **Current** este de tip read-only și furnizează informaţia utilă a nodului curent (cel referit prin **aux**);
- metoda **MoveNext()** are ca scop referirea următorului nod valid al listei, caz în care returnează **true**; dacă nu mai există noduri în listă returnează **false**.

Instrucţiunea **foreach** începe prin a apela metoda **Reset(),** după care apelează metoda **MoveNext()** cu scopul de a referi primul element al colecţiei, adică primul nod valid al listei; scopul variabilei **vb** este de a face distincţie între primul apel al metodei **MoveNext()** şi celelalte, care vor avansa referinţa **aux** atâta timp cât mai există un nod valid în listă. Utilizarea instrucţiunii **foreach** împreună cu colecţia **Lista** se poate face acum sub forma:

```
Lista l = new Lista();
l.Adaugare(100); l.Adaugare(200); l.Adaugare(150);
try
{
	foreach (int k in l) { Console.WriteLine(k); }
}
catch
{
	Console.WriteLine("Lista vida!!!");
}
```

S-a inclus instrucţiunea `foreach` într-un bloc `try` cu scopul de a lansa o excepţie în cazul în care lista este vidă.

O altă modalitate de a utiliza interfețele o constituie partajarea accesului la elementele unei clase complexe în concordanță cu o logică a utilizării elementelor clasei. Astfel, spre exemplificare, se va implementa clasa cont bancar care conține date personale cum ar fi: nume, prenume, cod numeric personal, domiciliu etc cât și date financiare, ca de exemplu: iban, sold etc. Evident, pentru cele două categorii de date stocate în clasa cont se

vor defini și implementa operații specifice: determinarea vârstei persoanei, operația de depunere respectiv retragere în / din cont etc. Interfețele vor fi astfel definite și apoi implementate în clasa cont astfel încât numai prin intermediul acestora se vor utiliza obiectele. Pentru rezolvarea problemei se va defini clasa operație bancară ce va conține data la care se efectuează operația, tipul operației (depunere / extragere) și suma:

```
class operatie_bancara
    {
        DateTime dop;
        public int suma;
        public char tip;
        public operatie_bancara(DateTime fdop, int fs, char ft)
        {
              dop = new DateTime(fdop.Year, fdop.Month, fdop.Day);
              suma = fs; tip = ft;
        }
    }
```

Interfețele construite sunt:

- pentru date personale:

```
interface IDate_Pers
    {
        string Nume // proprietatea pentru numele persoanei
        {
            get;
            set;
        }
// seteaza data nasterii
        void set_dn(int fy, int fm, int fz);
// determina varsta
        int get_varsta();
    }
```

- pentru date financiare:

```
    interface IDate_Fin
    {
        string Iban // proprietatea pentru cont
        {
            get;
            set;
        }
        void Depune(DateTime dt, int s); //depunere
        void Extrage(DateTime dt, int s);//extragere
        int get_sold(); // determina sold
    }
```

Clasa Cont moștenește și implementează cele două interfețe și are forma:

```
class Cont : IDate_Pers, IDate_Fin
{
    int si;
    string np, iban;
    DateTime dn = new DateTime();
    List<operatie_bancara> lopb = new List<operatie_bancara>();

    public Cont(int fsi) { si = fsi; }

//implementarea interfetei Date_Pers
    string IDate_Pers.Nume
    {
        get { return np; }
        set { np = value; }
    }

    void IDate_Pers.set_dn(int fy, int fm, int fz)
    {
       dn = DateTime.Parse(fy.ToString()+"-"+
              fm.ToString()+"-"+fz.ToString());
    }

    int IDate_Pers.get_varsta()
    {
       return DateTime.Now.Year - dn.Year;
    }

//implementarea interfetei Date_Fin
    string IDate_Fin.Iban
    {
       get { return iban; }
       set { iban = value; }
    }

    void IDate_Fin.Depune(DateTime dt, int s)
    {
       lopb.Add(new operatie_bancara(dt, s, 'd'));
    }

    void IDate_Fin.Extrage(DateTime dt, int s)
    {
       lopb.Add(new operatie_bancara(dt, s, 'e'));
    }

    int IDate_Fin.get_sold()
    {
       int sd = 0, se = 0;
       foreach (operatie_bancara opi in lopb)
          if(opi.tip == 'd') sd += opi.suma; else se += opi.suma;
       return si+sd-se;
```

```
    }
}
```

Se observă că la implementarea elementelor de interfață, în clasa *Cont*, s-a folosit o definire explicită a interfeței prin precedarea numelui metodei sau proprietății de numele interfeței acest lucru permite ca elementele din interfață să poată fi folosite doar prin intermediul unei interfețe și nu direct prin intermediul obiectului. Tot legat de acest lucru se observă ca modificatorul *public* nu mai este indicat la implementarea metodelor / proprietăților deoarece, prin definirea explicită a interfeței, caracterul public este asociat automat datorită faptului că elementele unei interfețe au întotdeauna caracter public.

2.2 Lucrul cu evenimente

Pentru a înțelege mecanismul evenimentelor se urmărește ca el să se implementeze folosind interfețe ce definesc un *pattern* ce trebuie implementat de clasele care urmează să comunice în acest fel. Din punct de vedere al logicii de implementare se pot stabili două categorii de clase: cele care generează obiecte observabile cât și cele care generează obiecte ce observă (observatori). Pentru o legătură cu lumea reală vă puteți imagina relația între părinți și copilul lor în sesul că părinții își observă copilul, deci joacă rolul de observatori, în timp ce copilul este entitatea observabilă. Obiectul observabil notifică observatorii când acesta își schimbă starea. De exemplu, când copilul nu mai are bani, acesta își înștiințează părinții care, apoi realizează o acțiune (de exemplu, mama îi dă bani copilului iar tata îl atenționează că a cheltuit prea mult). Un alt element util de observat în implementare constă în faptul că oobiectele observabile se cuplează la observatori și atunci conlucrează iar când se realizează operația de decuplare atunci ele nu mai conlucrează.

O clasă care generează obiecte ce observă (observatori) trebuie să implementeze o interfață care conține semnătura metodei (*Notificare*) care implementează funcționalitatea ce se execută atunci când observatorul este

notificat. Evident, fiecare clasă care generează observatori va implementa în mod propriu notificarea.

```
interface IObservator
    {
        void Notificare();
    }
```

Clasa care generează obiecte observabile trebuie să implementeze o interfață specifică *IObservabil* care definște semnăturile a două metode: una care cuplează observatori și alta care-i decuplează:

```
interface IObservabil
    {
        void Cupleaza(IObservator o);
        void Decupleaza(IObservator o);
    }
```

Pentru a defini mecanismul de organizare a colecției de observatori (*lobs*), cuplarea și decuplarea s-a construit clasa *Observabil_Impl* care implementează interfața *IObservabil*.

```
class Observabil_Impl : IObservabil
    {
        protected List<IObservator> lobs = new List<IObservator>();

        public void Cupleaza(IObservator o)
        {
            lobs.Add(o);
        }

        public void Decupleaza(IObservator o)
        {
            lobs.Remove(o);
        }
    }
```

Codul descris până în acest moment este de maximă generalitate și poate fi folosit de orice clase care generează obiecte care conlucrează în sensul că unele obiecte notifică alte obiecte care reacționează la notificările primite.

Pentru exemplificarea cât mai simplă a mecanismului s-a construit clasa observabilă numită *Element* care conține un atribut de tip întreg (*el*) iar la schimbarea valorii lui obiectul va notifica alte obiecte (observatorii).

```
class Element : Observabil_Impl
    {
        int el;
```

```
        public int Valoare_el
        {
            get { return el; }
            set
            {
                if (el != value)
                {
                    el = value;
                    foreach (IObservator o in lobs) o.Notificare();
                }
            }
        }
    }
```

Se poate observa că la schimbarea stării elementului, el notifică toți observatorii iar prin apelul metodei *Notificare* a fiecărui observator, observatorul va declanșa acțiunea ca răspuns al notificării.

Se propune clasa *Cons_obs* care va genera un observator ce va afișa la consolă mesajul *`Element modificat!!`* atunci când obiectul observabil își schimbă starea:

```
class Cons_obs : IObservator
    {
        public void Notificare()
        {
            Console.WriteLine("Element modificat!!");
        }
    }
```

Pentru a exemplifica funcționalitatea se construiește obiectul *oel* de tip *Element*, se construiește obiectul cu rol de observator (*cobs1*), se cuplează observatorul la obiectul observabil și se schimbă starea elementului fapt care generează apelul metodei *Notificare* din observator care determină afișarea mesajului corespunzător, după care se vizualizează noua valoare a elementului.

```
Element oel = new Element();
Cons_obs cobs1 = new Cons_obs();
oel.Cupleaza(cobs1);
oel.Valoare_el = 10;
Console.WriteLine("elment = {0}", oel.Valoare_el);
```

Vă propunem să testați întreaga funcționalitate și anume să realizați decuplarea observatorului după care să schimbați starea elementului și veți

observa că mesajul de notificare nu se va mai afișa pentru că obiectele nu mai conlucrează după decuplare.

Pentru a complica exemplul se mai adaugă un alt observator de data aceasta o clasă derivată din clasa formă (*Form*) care va implementa interfața *IObservator*. Metoda de notificare va afișa mesajul *Modificat* pe câte o linie nouă într-ul control (*mes*) de tip Label de pe formă. Pe de altă parte, din cadrul formei se dă posibilitatea modificării valorii elementului prin intermediul controlului (*tval*) de tip TextBox la apăsarea pe butonul *Set_val* fapt care va determina notificarea ambilor observatori ceea ce va duce la apariția unui mesaj la consolă cât și în formă. Pentru a putea modifica elementul din formă, la constructorul formei se trimite ca parametru o referință la obiectul de tip *Element* care se reține într-un atribut al formei și prin care se va permite modificarea stării elementului.

```
public partial class Form1 : Form, IObservator
{
        Element rel = null;
        public Form1(object fel)
        {
            InitializeComponent();
            rel = (Element)fel;
        }

        public void Notificare()
        {
            mes.Text += "Modificat"+ Environment.NewLine;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            rel.Valoare_el = Convert.ToInt32(tval.Text);
        }
}
```

Testarea funționalității s-a realizat prin definirea metodei *Main* în forma:

```
static void Main(string[] args)
        {
            Element oel = new Element();

            Cons_obs cobs1 = new Cons_obs();
            Form1 forma = new Form1(oel);

            oel.Cupleaza(cobs1);
            oel.Cupleaza(forma);
```

```
            oel.Valoare_el = 10;

            forma.ShowDialog();

            Console.WriteLine("elment = {0}", oel.Valoare_el);
        }
```

ceea ce determina apariția interfeței aplicației ca în figura 2.1. Primul mesaj a fost afișat ca urmare a schimbării stării obiectului prin atribuirea realizată în *Main* iar cel de-al doilea ca urmare a modificării stării elementului din formă.

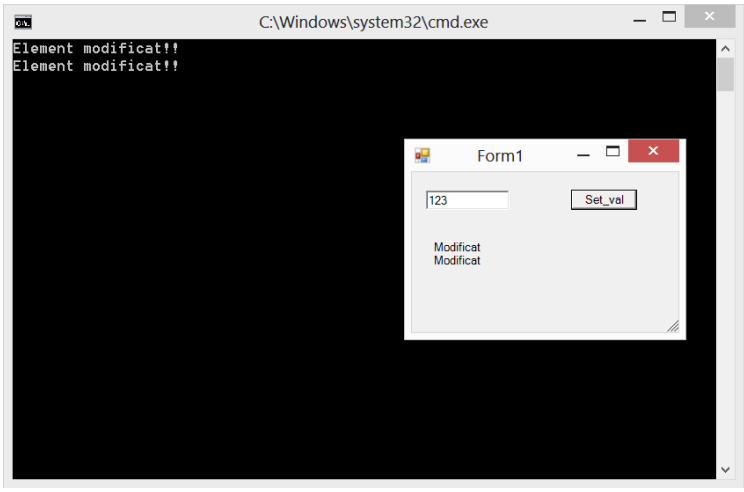

Figura 2.1 Interfața aplicației

O clasă poate avea pe lângă atribute, proprietăți, metode și evenimente. Prin eveniment se înțelege un tip de membru al unei clase care este activat când evenimentul se declanșează, în timpul execuției unei aplicații. La momentul declanșării evenimentului metoda asociată va fi apelată.

Spre exemplu, dacă considerăm un obiect de tip *Timer*, atunci el are definit evenimentul *Tick* care se declanșează periodic, după ce se scurge o perioadă de timp. Declanșarea evenimentului impune apelul unei metode care implică efectuarea unei procesări ce ține de specificul aplicației, deci nu se poate defini metoda în clasa *Timer* însă ea poate fi cuplată la eveniment, din exteriorul clasei *Timer*. Concret, se va construi o aplicație care să asocieze

efectul de *blinking* unui text (textul devine vizibil respectiv invizibil la un interval de 300 milisecunde). Pentru acest lucru se va declara un obiect de tip *Timer* (*t*) în clasa *Form1*: `Timer t;` Pe evenimentul *Load*, de inițializare a formei, se va construi obiectul *Timer*, se va stabili durata intervalului de timp, în milisecunde și se va asocia evenimentului *Tick* metoda (*actiune*) care se va apela când se va declanșa evenimentul:

```
private void Form1_Load(object sender, EventArgs e)
        {
            t = new Timer();
            t.Interval = 300;
            t.Tick += new EventHandler(actiune);
        }
```

Se observă că asocierea metodei de utilizator *actiune*, evenimentului *Tick*, s-a realizat cu operatorul += adică prin adăugarea unui nou obiect de tip *EventHandler*.

Metoda *actiune* are un prototip prestabilit în sensul că aceste metode care se apelează la declanșarea evenimentelor primesc: obiectul care a interceptat evenimentul (*sender*) și un obiect (*e*) care conține argumentele ce însoțesc evenimentul. Această metodă se va defini în aplicație și va schimba starea de vizibilitate a textului care este un control de tip *Label* având numele *et*:

```
void actiune(object sender, EventArgs e)
        {
            et.Visible = ! et.Visible;
        }
```

Pornirea timer-ului se va face prin aplelul metodei *Start* în timp ce, oprirea lui, se face prin apelul metodei *Stop*. Ambele metode se apelează prin obiectul *t*, de tip *Timer* și operațiile se vor declanșa prin apăsarea a două butoane special construite în acest sens.

Detașare metodei evenimentului se face prin construcția:

```
t.Tick -= new EventHandler(actiune);
```

după care, la apăsarea pe butonul care startează *timer*-ul nu se va mai apela funcția *actiune*; deci, efectul de *blinking* pentru acel text nu se va mai produce.

La începutul acestui subcapitol s-a precizat faptul că modalitatea de a construi obiecte observabile și observatori reprezintă, de fapt, modul în care se implementează evenimentele. Folosind evenimente în aplicații .NET noi utilizăm un mecanism mai evoluat și mai simplu de folosit dar care ascunde în esență mecanismul descris inițial. Pentru o mai bună înțelegere a problemei se propune ca exemplul din debutul acestui subcapitol sa-și păstreze funcționalitatea doar că se va utiliza conceptul de eveniment pentru ca obiectele să conlucreze.

Clasa *Element* va fi modificată astfel încât ea va conține un delegat care este similar unui pointer la funcție adică se precizează o semnătură de funcție fără a-i preciza implementarea și un eveniment care este similar unei proprietăți asociate unui delegat.

```
class Element
    {
        public delegate void DElement();
        public event DElement Schimba_el;

        int el;

        public int Valoare_el
        {
            get { return el; }
            set
            {
                if (el != value)
                {
                    el = value;
                    if(Schimba_el!=null) Schimba_el();
                }
            }
        }
    }
```

Se observă că la schimbarea stării elementului se apelează funcția *Schimba_el* care este de tipul precizat prin delegat și care la acest moment nu are o implementare concretă. Testul `Schimba_el!=null` ne asigură că delegatul referă o implementare concretă altfel s-ar genera eroare la apel.

Observatorii elementului trebuie conform acestui mecanism să implementeze metoda *Schimba_el*. Pentru similitudine se păstrează aceeași observatori iar

metoda *Schimba_el* va păstra conținutul metodei *Notificare* din exemplul inițial.

```
class Cons_obs
{
        public void Schimba_el()
        {
            Console.WriteLine("Element modificat!!");
        }
}

public partial class Form1 : Form
{
        Element obel = null;
        public Form1(object fel)
        {
            InitializeComponent();
            obel = (Element)fel;
        }

        public void Schimba_el()
        {
            mes.Text += "Modificat!!"+Environment.NewLine;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            obel.Valoare_el = int.Parse(tb.Text);
        }
}
```

Se observă că practic observatorii au rămas la fel doar ca s-a modificat numele metodei *Notificare* care acum a devenit *Schimba_el*. Dacă am fi denumit evenimentul *Notificare* observatorii ar fi rămas ca în exemplu inițial.

O altă modificare importantă ce tine de lucrul cu evenimente o constituie cuplarea și decuplarea care acum se realizează folosind operatorii += respectiv -= pe eveniment. Acest lucru este vizibil în metoda Main, care acum are forma:

```
static void Main(string[] args)
{
   Element el = new Element();

   Cons_obs cobs1 = new Cons_obs();
   Form1 frm = new Form1(el);

   el.Schimba_el += new Element.DElement(cobs1.Schimba_el);
   el.Schimba_el += new Element.DElement(frm.Schimba_el);
```

```
    el.Valoare_el = 10;

    frm.ShowDialog();

    Console.WriteLine("elment = {0}", el.Valoare_el);
}
```

Construirea unei clase care să conțină un membru de tip eveniment trebuie proiectată astfel încât acel eveniment să aibă condiții bine precizate pentru a se declanșa. Cu alte cuvinte, cel ce proiectează clasa trebuie să definească evenimentele la care obiectele respective vor raspunde (de exemplu, clasa *Button* definește evenimentul *Click* care se va declanșa la momentul când se apasă pe butonul mouse-ului și cursorul acestuia se află pe suprafața butonului sau clasa *Timer* definește evenimentul *Tick* care se declanșează periodic la un anumit interval de timp). Pentru exemplificare, se dorește a se construi o clasă care să implementeze animația prin vizualizarea succesivă a unor imagini (tehnica filmului). Se urmărește definirea unui eveniment care să notifice momentul terminării afișării secvenței de imagini.

Un alt aspect deosebit de important în implementarea lucrului cu evenimente se referă la mecanismul prin care este posibilă asocierea unei metode care se va apela la momentul declanșării evenimentului. Ideea de baza constă în faptul că cel care proiectează clasa definește evenimentul însă acțiunea concretă care se va efectua la momentul declansării evenimentului se definește de către cel ce utilizează clasa. Pentru a implementa mecanismul în clasa care definește evenimentul trebuie declarat un delegat, de exemplu:

```
public delegate void MM_EventHandler(object sender, EventArgs e);
```

S-a declarat de această dată un delegat care se aliniază la prototipul delegaților ce se asociază evenimentelor în ceea ce privește programarea aplicațiilor Windows. Prototipul este standard în sensul că parametrii au semnificațiile: obiectul care trimite evenimentul (*sender*) și obiectul (*e*) care încapsulează alte elemente ce sunt trimise odată cu evenimentul. După declararea delegatului se declară evenimentul (*Notificare_Animatie*) pe baza lui, astfel:

```
        public event MM_EventHandler Notificare_Animatie;
```

Clasa care implementează animația (*MM_anim*), prin tehnica filmului este:

```
namespace animatie
{
    public delegate void MM_EventHandler(object sender, EventArgs e);

    public class MM_anim
    {
        public event MM_EventHandler Notificare_Animatie;

//folosit pentru a stoca imaginile
        ArrayList vimag = new ArrayList();
//reprezinta colectia de imagini
        int k; //variabila pentru referirea unei imagini din colectie
        PictureBox cadru; // controlul pentru afisarea imaginilor
// timer pentru controlul periodicitatii afisarii imaginilor
        Timer t;
// constructor care primeste controlul de afisare si
// intervalul de timp la care vor fi afisate imaginile
        public MM_anim(PictureBox fcadru, int intv)
        {
            cadru = fcadru;
            t = new Timer();
            t.Interval = intv;
            t.Tick += new EventHandler(actiune);
        }
//adaugarea unei imagini la colectie;
//s – numele fisierului ce contine imaginea
        public void Add_imag(string s)
        {
            vimag.Add(new Bitmap(s));
        }
// metoda care declanseaza animatia
       public void Play()
        {
            k = 0; // refera prima imagine din colectie
            t.Start(); // porneste timer-ul
        }
// metoda cuplata la evenimentul Tick al Timer-ului
// ea realizeaza afisarea tuturor imaginilor din colecție
        private void actiune(object sender, EventArgs e)
        {
            cadru.Image = (Bitmap)vimag[k++];
            if (k == vimag.Count)
            {
// in cazul in care s-a afisat si ultima imagine din colectie
                t.Stop(); // se opreste timer-ul
```

```
            // se invoca evenimentul Notificare_Animatie
              if (Notificare_Animatie != null)
                   Notificare_Animatie(this, new EventArgs());

          }
       }
    }
}
```

Se observă că, după ce se afișează și ultimul cadru, se oprește *timer*-ul și se apelează metoda `Notificare_Animatie` care primește obiectul (*this*) și un obiect de tip *EventArgs* pentru a realiza concordanța cu prototipul delegatului; în exemplul prezentat nu se realizează și trimiterea de informații suplimentare odată cu evenimentul.

Testarea funcționalității clasei s-a realizat prin construirea unei aplicații *Windows Forms* care utilizează un control de tip *PictureBox* numit *canvas* și un buton pentru a derula animație (*Play*). In cadrul clasei formă s-a declarat un obiect de tip *MM_anim*: `MM_anim anim;` Metoda `Form1_Load` ce relizează inițializarea formei are următoarea definiție:

```
private void Form1_Load(object sender, EventArgs e)
{
// construirea obiectului anim
    anim = new MM_anim(canvas, 300);
    int i;
// calea directorului in care se regasesc imaginile
    string cale = "d:\\media\\pamintul\\";
// adaugarea imaginilor in colectie – fisierele au numele
// c1.jpg, c2.jpg, ... c15.jpg
    for (i = 0; i < 15; i++)
       anim.Add_imag(cale + "c" + (i + 1).ToString() + ".jpg");
// asocierea metodei actiune evenimentului Notificare_Animatie
    anim.Notificare_Animatie +=new MM_EventHandler(actiune);
}
```

Apăsarea butonului *Play* determină declanșarea animației:

```
private void button1_Click(object sender, EventArgs e)
{
       anim.Play();
}
```

Metoda *actiune* care se va apela la declanșarea evenimentului de terminare a derulării animației are ca obiectiv afișarea mesajului:

```
Animatia s-a terminat!!:
```

```
public void actiune(object sender, EventArgs e)
{
       MessageBox.Show("Animatia s-a terminat!!", "Notificare",
              MessageBoxButtons.OK, MessageBoxIcon.Asterisk);
}
```

Interfața aplicației precum și notificarea terminării animației se ilustrează în figura 2.2.

Figura 2.2 Afișarea mesajului de notificare a terminării animației

Dacă se dorește a se trimite odată cu mesajul și alte informații atunci este neceasar a se deriva o clasă din clasa *EventArgs* care să conțină propriile elemente. In exemplul prezentat ar putea fi util să se transmită odată cu evenimentul de notificare și informația suplimentară cu privire la numărul cadrului. Astfel, în domeniul de nume *animatie* se va defini clasa *MM_EvArgs* după cum urmează:

```
public class MM_EvArgs : EventArgs
{
        int cs; // numarul cadrului
       // constructorul care primeste numarul cadrului
        public MM_EvArgs(int z)
        {
            cs = z;
        }
       // proprietatea pentru obtinerea numarului de cadre
        public int cadru_stop
        {
            get { return cs; }
        }
}
```

Modificările care decurg din folosirea unei clase proprii pentru argumentele evenimentului sunt:

- în clasa *MM_anim* se modifică metoda *actiune* astfel încât apelul metodei *Notificare_Animatie* va avea forma: `Notificare_Animatie(this, new MM_EvArgs(k));`
- în clasa *Form1* unde se testează funcționalitatea clasei *MM_anim*, se modifică metoda *actiune* care se apelează pe evenimentul *Notificare_Animatie* astfel încât mesajul va conține și numărul de cadre afișate:

```
public void actiune(object sender, MM_EvArgs e)
    {
     MessageBox.Show("Animatia s-a terminat ultimul cadru: "
                     + e.cadru_stop.ToString()+"!!",
                       "Notificare", MessageBoxButtons.OK,
                                      MessageBoxIcon.Asterisk);
    }
```

3. *Reflection*

Reflection este o tehnică de citire a *dll*-urilor managed, mai precis a *assembly*-urilor. Această tehnică furnizează un mecanism de a descoperi tipurile și de a invoca proprietățile la *runtime*. Se utilizează în mod deosebit la dezvoltarea aplicațiilor proiectate în arhitectură decuplată.

Un modul managed conține metadate și cod în limbaj de asamblare intermediar (IL – Intermediar Language). Un astfel de modul se obține prin compilarea codului sursă, de exemplu C#, ca în figura 3.1

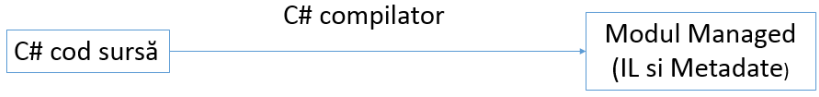

Figura 3.1 Compilarea

Metadatele descriu tipurile și membrii din codul sursă cât și tipurile și membrii referiți în codul sursă. Codul sursă compilat este descris în modul sub forma

limbajului intermediar (IL). Acest limbaj intermediar este unic indiferent de limbajul sursă (Visual Basic.Net, C#, F# etc), din mediul Visual Studio.Net, care a fost folosit la scrierea codului sursă.

Pentru a obține codul executabil (figura 3.2), codul compilat (IL) este translatat prin CLR în instrucțiuni procesor (CPU). CLR-ul este dependent de platforma pe care rulează aplicația.

Figura 3.2 Obținerea codului executabil

CLR-ul lucrează cu assembly-uri. Un assembly este o grupare logică a unuia sau mai multor module managed sau fișiere de resurse (figura 3.3). De asemenea, el reprezintă unitatea atomică ce poate fi reutilizată, securizată și versionată.

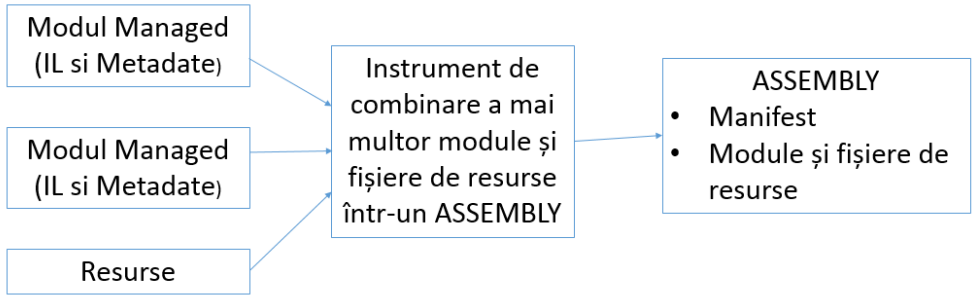

Figura 3.3 Construire assembly

Se observă din figura 3.3 că un assembly mai conține și o secțiune numită *Manifest* care conține metadate ce descriu setul de fișiere dintr-un assembly. Din punct de vedere fizic, un assembly se poate identifica printr-un fișier *dll* sau *exe*. Un alt concept cu care operează o aplicație din punct de vedere structural este domeniul aplicației (*AppDomain*). Domeniul aplicației conține assembly-urile cu care operează o aplicație (figura 3.4).

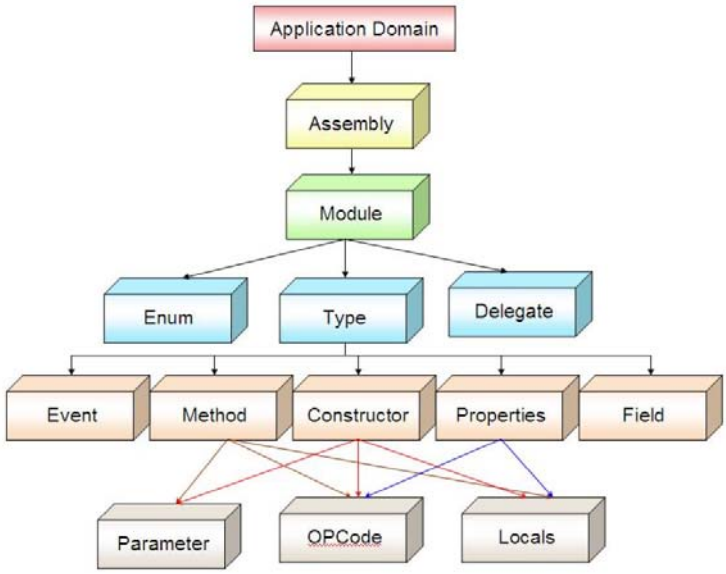

Figura 3.4 AppDomain

Din figura 3.4 se observă că domeniul aplicației conține assembly-uri care la rândul lor conțin module sau resurse. Modulele la rândul lor pot conține elemente de tip enumerativ (Enum), tipuri definite prin intermediul claselor sau delegați. Tipurile au elemente specifice claselor: constructori, atribute, proprietăți, metode sau evenimente. Atât metodele cât și constructorii au parametri, cod și variabile locale, în timp ce proprietățile au doar cod respectiv variabile locale.

3.1 Lucru cu *assembly*-uri care conțin resurse

Pentru a exemplifica modul de exploatare a assembly-urilor se construiește un assembly care să conțină doar resurse și apoi acestea vor fi utilizate într-o aplicație de tip Windows Forms.

Pentru a crea un *assembly* numai cu resurse trebuie să se parcurgă pasii:

- Se construiește un proiect de tip *Empty Project* având numele, de exemplu, *res_demo*;

- In fereastra *Solution Explorer* se apasă click dreapta pe numele proiectului și se selectează *Add / New Item* si se adaugă la proiect fișierul *Resource1.resx* în vederea stocării de resurse;
- Se adaugă la fișierul *Resource1.resx* resursa de tip șir de caractere cu numele *s1* care conține valoarea *sir resursa*;
- In fereastra *Solution Explorer* se dă click dreapta pe numele proiectului și se alege opțiunea *Properties*; iar proprietatea *Output Type* va fi setată cu valoarea *Class Library*;
- Se efectuează *Build / Build Solution*.

Astfel, resursele sunt compilate într-un *assembly* și vor fi împachetate în fișierul *res_demo.dll*.

Resursele existente într-un *assembly* pot fi accesate prin intermediul clasei *ResourceManager*. Având fișierul *res_demo.dll* care conține resursele se va dezvolta o nouă aplicație pentru a accesa resursele.

In primul pas se va adăuga *assembly*-ul la noul proiect. Acest lucru se realizează apăsând click dreapata pe *References* în fereastra *Solution Explorer* și se selectează *Add Reference*. Se identifică fișierul *res_demo.dll* și se adaugă la referințele proiectului.

Pentru a accesa resursa împachetată în fișierul *res_demo.dll* se va utiliza secvența de cod:

```
System.Reflection.Assembly ass;
ass = System.Reflection.Assembly.Load("res_demo");
System.Resources.ResourceManager rm =
    new System.Resources.ResourceManager("res_demo.Resource1", ass);
string sir;
sir = rm.GetString("s1");
MessageBox.Show(sir);
```

A fost construit un obiect de tip *Assembly* care a încărcat *assembly*-ul *res_demo*. Din acest *assembly* se accesează resursele din *Resource1* (numele fișierului *resx* din proiectul *res_demo*) și se crează un obiect (*rm*) al clasei *ResourceManager*. Prin intermediul acestui obiect, utilizând metode specializate în concordanță cu tipul resursei se poate accesa resursa prin numele ei (în exemplu *s1*).

3.2 Accesarea dinamică a elementelor unui *assembly*

Biblioteca *System.Reflection* conține clase destinate accesării dinamice (în timpul execuției programului) a informațiilor stocate în secțiunea de metadate a *assembly*-urilor. Cele mai importante clase utilizate în acest scop sunt:

- *Assembly*: permite încărcarea dinamică a *assembly*-urilor de pe disc și enumerarea tipurilor (clase, structuri, interfețe, …) conținute în acesta (sub formă de obiecte de tip *Type*);
- *Type*: permite obținerea de informații detaliate despre un tip definit în interiorul unui *assembly* (denumire, membri, …); clasa este utilizată și pentru instanțierea dinamică a obiectelor;
- **Info*: clase care permit obținerea de informații despre și manipularea dinamică pentru membrii unui tip (constructori, metode, proprietăți, câmpuri, …).

Obiectele de tip *Assembly* se pot obține prin încărcarea unui *assembly* existent:

```
// Încărcare assembly din GAC pe baza denumirii complete
Assembly asmSystemData = Assembly.Load(
    "System.Data, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089");

// Încărcare assembly din fișier
Assembly asmTest = Assembly.LoadFrom("...\test.dll");
```

Obiectele *Type* se pot obține prin interogarea unui obiect existent, enumerarea tipurilor dintr-un *assembly* sau prin folosirea operatorului *typeof*:

```
string strTest = "test";
Assembly asm = Assembly.Load(
    "mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089");

// Obținere tip prin interogarea obiectului –
// se folosește metoda GetType
// definită în clasa de bază System.Object
Type t1 = strTest.GetType();

// Obținere tip folosind operatorul typeof
Type t2 = typeof(string);
```

```
// Obținere tip pe baza obiectului Assembly
Type t3 = asm.GetType("System.String");

// Toate cele trei obiecte (t1, t2, t3) sunt obiecte
// type corespunzătoare
// clasei System.String definite în assembly-ul mscorlib.dll
```

Pe baza obiectului *Type* putem construi dinamic obiecte, afla informații despre ele și invoca dinamic membrii acestora:

```
Assembly asm = Assembly.Load(
    "mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089");
Type type = asm.GetType("System.String");

// Construirea dinamică a unui obiect string cu valoarea "ttt"
object str = Activator.CreateInstance(type, 't', 3);

// Enumerarea proprietăților disponibile
foreach(PropertyInfo prop in type.GetProperties())
{
    Console.WriteLine(prop.Name);
}

// Obținerea informațiilor și invocarea dinamică a unei metode
MethodInfo method = type.GetMethod("Insert");
str = method.Invoke(str, new object[] { 0, "a" });
Console.WriteLine(str);

// Citirea dinamică a valorii unei proprietăți prin
// intermediul metodei get
PropertyInfo propLen = type.GetProperty("Length");
int len = (int)propLen.GetGetMethod().Invoke(str, new object[]{});
Console.WriteLine(len);
```

Exemple de utilizare:
1) Afișarea tuturor claselor derivate din *System.Exception* dintr-un *assembly*

```
static void AfisareClaseExceptie(Assembly asm)
{
    // Obținem tipul clasei de bază
    Type exceptionType = typeof(System.Exception);

    // Enumerăm toate tipurile din assembly-ul primit
    foreach (Type type in asm.GetTypes())
    {
        // Dacă tipul curent este derivat direct
       // sau indirect din System.Exception
```

```
        if (type.IsSubclassOf(exceptionType))
        {
            // Îl afișăm
            Console.WriteLine(type.Name);
        }
    }
}
```

2) Afișarea valorilor câmpurilor pentru un obiect oarecare

```
static void AfisareValoriCampuri(object obj)
{
    // Obținem obiectul Type
    Type type = obj.GetType();

    // Enumerăm toate câmpurile (și cele private)
    foreach(FieldInfo fieldInfo in type.GetFields(
        BindingFlags.Instance | BindingFlags.Public |
BindingFlags.NonPublic))
    {
        Console.WriteLine("{0} ({1}): {2}",
            fieldInfo.Name,
            fieldInfo.FieldType,
            fieldInfo.GetValue(obj));
    }
}
```

3) Afișarea metodelor pentru un obiect oarecare

```
static void AfisareMetode(object obj)
{
    // Obținem obiectul Type
    Type type = obj.GetType();

    // Enumerăm toate metodele publice
    foreach (MethodInfo method in type.GetMethods())
    {
        // Afișăm tipul returnat și denumirea metodei
        Console.WriteLine("{0} {1}",
            method.ReturnType.Name,
            method.Name);

        // Afișăm tipul și denumirea parametrilor
        foreach (ParameterInfo param in method.GetParameters())
        {
            Console.WriteLine("   {0} {1}",
                param.ParameterType.Name,
                param.Name);
        }
        Console.WriteLine();
    }
```

```
}
```

4) Modificarea valorii pentru un câmp privat

```
static void ModificareValoareCampPrivat(object obj, string numeCamp,
object valoare)
{
    // Obținem obiectul Type
    Type type = obj.GetType();

    // Identificăm câmpul
    FieldInfo camp = type.GetField(
        numeCamp, BindingFlags.Instance | BindingFlags.NonPublic);

    // Modificăm valoarea
    camp.SetValue(obj, valoare);
}
```

4. Dezvoltarea aplicațiilor software bazate pe *MS Office*

4.1 Introducere

Aplicațiile bazate pe Office sunt deosebit de utile, în practică, deoarece folosesc facilitățile existente în aplicațiile pachetului software Microsoft Office. Pentru a dezvolta astfel de aplicații sau extensii, pe platforma .NET a fost construit un framework cunoscut sub denumirea de Visual Studio Tools for Office (*VSTO*). Acesta oferă suport de programare .NET pentru *Word*, *Excel*, *Outlook*, *PowerPoint*, *Project*, *Visio* și *InfoPath* în *Visual Studio*. De asemenea, VSTO permite ca documente Word și Excel să folosească caracteristici de programare din .NET cum ar fi suport pentru legarea datelor, controale care se pot folosi în forme windows etc.

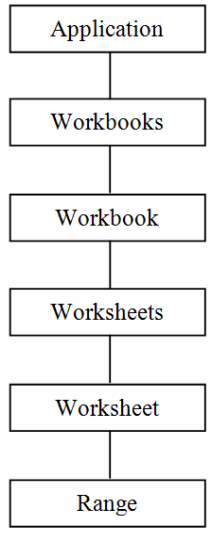

Figura 4.1 Ierarhia modelului EOM

Scrierea codului pentru aplicațiile bazate pe Office implică utilizarea modelului cunoscut sub denumirea de *OOM* – Office Object Model. Acest model conține un set de clase și obiecte necesare pentru pentru controlul aplicațiilor Office. Modelele sunt particularizate în funcție de aplicațiile pe

care le controlează, de exemplu: *EOM* – Excel Object Model, *WOM* – Word Object Model etc. In general, aceste modele conțin o ierarhie de clase și sunt organizate astfel încât în rădăcina ierarhiei se află clasa *Application* care modelează comportamentul unei aplicații particulare din pachetul software MS Office. Pe lângă clasa *Application* care este prezentă în toate ierarhiile, există și clase particulare în ierarhie ce depind de aplicația propriu-zisă și corespund entităților pe care le manipulăm efectiv în aplicațiile *Office*. De exemplu, pentru aplicația *Excel*, principalele clase cât și relațiile dintre ele sunt ilustrate în figura 4.1.

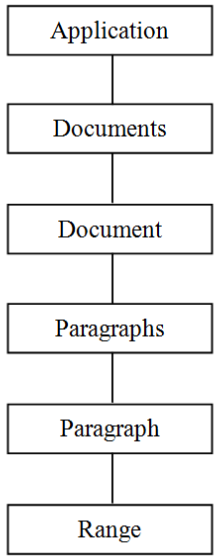

Figura 4.2 Ierarhia modelului WOM

Se observă că, la baza ierarhiilor se află clasa *Application* care modelează comportamentul aplicației Office și prin intermediul acestei clase se construiește o instanță a aplicației. Instanța aplicației poate conține o colecție de documente modelată prin intermediul colecției *Documents* în WOM sau *Workbooks* în EOM. Un element al colecției, adică un obiect de tip *Document* sau *Workbook*, poate conține o colecție de paragrafe (*Paragraphs*) sau o colecție de foi de calcul *Worksheets*.
Pentru aplicația Word, ierarhia este prezentată în figura 4.2.

Pentru dezvoltarea de aplicații bazate pe *PowerPoint*, modelul este *PPOM* și are ierarhia prezentată în figura 4.3.

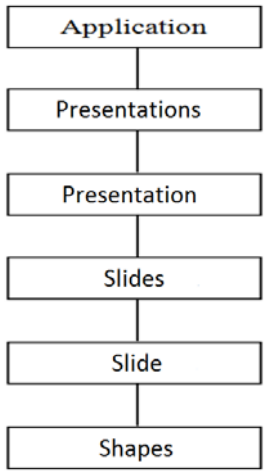

Figura 4.3 Ierarhia modelului PPOM

Din figură se observă că aplicația este compusă dintr-o colecție de prezentări (*Presentations*) care are elemente de tip prezentare (*Presentation*), o prezentare la rândul ei se compune dintr-o colecție de slide-uri (*Slides*) de elemente *Slide* în care se găsesc diferite tipuri de obiecte, cu corespondent vizual, conținute într-o colecție de elemente *Shapes*.

Ierarhiile de clase specifice tipurilor de aplicații Office conțin clase care descriu entitățile cu care operează aplicațiile pachetului Office.

Folosind elemente ale modelului de programare OOM se pot dezvolta aplicații în diferite modalități:

- *Aplicații de utilizator ce interacționează cu aplicații Office*; adică programatorul își dezvoltă propria sa aplicație care în *background* va controla și interacționa cu aplicații și documente Office. De exemplu, se construiește o aplicație independentă, în mod consolă sau *Windows Forms*, care are ca scop introducerea unor date într-un document *Word* pentru a fi ulterior stocat sau imprimat pe hârtie. Executarea unei astfel de aplicații va determina lansarea în execuție a aplicației Office pentru a efectua diferite procesări, specifice. In acest mod, aplicația de utilizator se va executa într-un proces distinct față de aplicația Office.

- *Construirea de module funcționale* existente sub formă de biblioteci cu legare dinamică (DLL – Dynamic Link Library) care se includ în aplicațiile Office pentru a personaliza prelucrările aplicate documentelor. De exemplu, construirea unui modul add-in care să aibă ca element de interfață un buton pentru efectuarea unei prelucrări nestandard asupra datelor existente într-o foaie de calcul;
- *Atașarea de cod documentelor Office*, implică construirea unei aplicații pe baza unui șablon de aplicație Office, definit în mediul Visual Studio ca tip distinct de proiect (de exemplu, pe baza unui șablon de aplicație *Excel*). Codul atașat permite efectuarea de prelucrări personalizate și impune construirea de noi interfețe pentru declanșarea procesărilor. Secvențele de cod se pot atașa fie documentelor Office fie șabloanelor de documente.

Ultimele două modalități prezentate de a dezvolta aplicații bazate pe Office au caracteristic faptul că secvențele de cod personalizate rulează cu aplicația Office în același proces; conceptul poartă numele de *cod gazduit*. Pentru a rula codul în procesul aplicației Office, aplicația respectivă trebuie să recunoască codul, sa-l încarce în procesul său și apoi sa-l execute. Componentele Office add-in sunt înregistrate în registri astfel încât ele pot fi găsite și startate de către aplicațiile Office. Codul asociat documentelor nu necesită a fi înregistrat, în schimb el este asociat cu documentul prin adăugarea de proprietăți specifice care se stochează în fișierul documentului. Când documentul Office este încărcat atunci se consultă proprietățile și se încarcă și codul asociat documentului.

Inainte de VSTO, dezvoltatorii de aplicații bazate pe Office puteau folosi limbajul VBA pentru a scrie secvențe de cod prin care să-și definească propriile prelucrări. Aceste secvențe erau scrise direct în interfața aplicației Office. Toate secvențele de cod VBA se stocau odată cu documentul pentru care acestea erau scrise. In VSTO, această modalitate de a asocia cod unui document se numește *soluție la nivel de document*. Astfel, în VSTO, se poate construi o componentă add-in care să fie livrată împreună cu documentul

pentru care procesează date. Această componentă devine activă la momentul încărcării documentului respectiv în aplicația Office. Acest tip de soluție la nivel de document este relevant în cazul aplicațiilor Word și Excel.

VSTO mai introduce un nou concept și anume *soluție la nivelul aplicației*. Astfel, soluția furnizată sub forma unui *add-in* se folosește pentru a realiza o procesare specifică și se poate aplica oricărui document deschis în aplicația pentru care a fost creată componenta.

5.2 Aplicații ce interacționează cu aplicații Office

Acest mod de interacțiune cu aplicațiile din Office presupune existența unei aplicații principale care va avea ca scop construirea unei noi instanțe a unei aplicații Office, apoi prin folosirea modelului orientat obiect, controlul interacțiunii cu aceasta. La final, aplicația Office trebuie închisă în mod explicit deoarece închiderea aplicației principale nu determină în mod implicit și închiderea aplicației Office. Pe de altă parte, aplicația Office rulează în background adică ea nu este vizibilă pentru utilizator. Prin schimbarea valorii proprietății *Visible*, fereastra aplicației Office devine vizibilă pentru utilizator astfel încât acesta poate realiza vizual alte operații folosind interfața acesteia.

Pentru exemplificare, se va construi o aplicație *Windows Forms* care va conține două controale, unul de tip *TextBox* și unul de tip *DataGridView* pe care utilizatorul le poate încărca cu date (figura 4.4). La apăsarea unui buton, textul din *TextBox* va fi exportat într-un document *Word*. De asemenea și datele conținute în controlul de tip *DataGridView* se vor exporta în document sub forma unui tabel. După efectuarea transferului, fereastra aplicației *Word* va deveni vizibilă astfel încât utilizatorul va putea salva, tipări sau închide documentul.

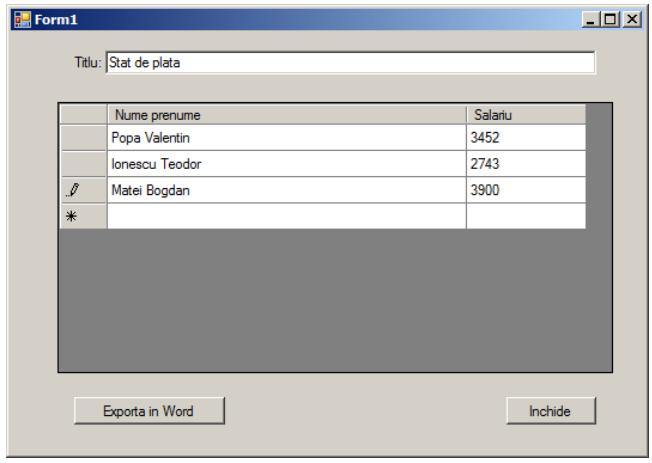

Figura 4.4 Forma aplicației principale completată cu date

La apăsarea pe butonul *Exporta in Word* datele introduse în forma aplicației vor fi exportate într-un document Word, după cum se ilustrează în figura 4.5. Pentru ca o aplicație să interacționeze cu un document al unei aplicații Word, prin modelul WOM, trebuie ca la proiect să se adauge referința către componenta .NET: *Microsoft.Office.Interop.Word*, pentru care se adaugă și domeniul de nume sub forma:

```
using Word = Microsoft.Office.Interop.Word;
```

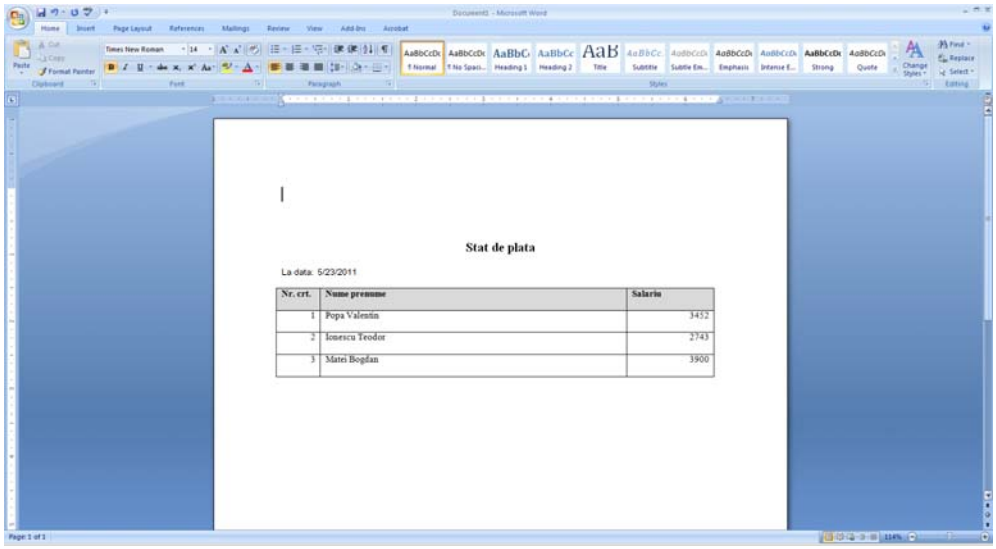

Figura 4.5 Document Word construit prin intermediul unei aplicații de utilizator

A. Construirea documentului, necesită mai întâi crearea cadrului de lucru:

- Crearea unei noi instanțe a unei aplicații Word:

```
Word.Application wapl = new Word.Application();
```

- Construirea în cadrul acestei instanțe a unui nou document:

```
Word.Document wdoc = wapl.Documents.Add();
```

B. Popularea documentului cu date și formatarea lor corespunzătoare:

Unul dintre cele mai utilizate și flexibile obiecte care sunt necesare pentru explorarea și popularea cu date a unui document Word, prin modelul WOM, este *Range*. Prin intermediul lui se definește un bloc continuu de text dintr-un document. După ce documentul a fost creat și este vid, se definește un obiect de tip *Range* mapat pe întregul document, prin apelul metodei *Range()* a obiectului document (*wdoc*):

```
Word.Range wr = wdoc.Range();
```

După definirea unui astfel de obiect se poate seta proprietatea *Text* pentru a scrie un anumit text în document, în zona marcată de obiectul *Range*:

```
wr.Text = "\n\n"+titlu.Text+"\n";
```

din secvența de cod, se observă că s-au lăsat două linii vide în document, iar textul celei de-a treia a fost preluat din obiectul *titlu* de tip *TextBox* care conține șirul: *Stat de plata* după cum se poate observa în figura 4.4. După scrierea acestui text se mai adaugă o linie vidă în cadrul documentului.

Textul din cadrul unui Range poate fi formatat corespunzător:

```
wr.Font.Name = "Times New Roman";
wr.Font.Size = 14;
wr.Font.Bold = -1;
```

din secvență se observă că s-a ales fontul *Times New Roman* de dimensiune *14* și îngroșat (*Bold*).

Pentru a continua cu adăugarea de alte texte, la document, trebuie să repoziționăm în mod corespunzător obiectul de tip *Range*. In cazul exemplului, se dorește ca noul *Range* să fie poziționat la sfârșitul celui curent adică să permită adăugarea unui text în continuarea celui care a fost scris în document. Prin apelul metodei *Collapse()*, pentru un obiect de tip *Range*, se realizează generare unei noi poziții care presupune ca poziția de început să fie identică cu cea de sfârșit. In funcție de direcția în care se repoziționează

domeniul, obiectul *Range* se poate poziționa la sfârșitul sau la începutul domeniului curent:

```
wr.Collapse(Word.WdCollapseDirection.wdCollapseEnd);
```

din apelul metodei se observă că direcția este indicată spre sfârșitul documentului adică poziția de început a noii zone este identică cu cea de sfârșit a obiectului *Range* curent. Apelând metoda *InsertAfter()* se poate insera un text după această nouă poziție:

```
wr.InsertAfter("La data: " + DateTime.Now.Date.ToShortDateString()+
                                                                    "\n");
```

linia de text adăugată include și data sistem și determină trecerea la o nouă linie în document. După adăugarea textului, el este formatat diferit de primul text introdus în document:

```
wr.Font.Name = "Arial";
wr.Font.Size = 10;
wr.Font.Bold = 0; // neingrosat
```

Alinierea textului în pagină se face, în mod implicit, la stânga. Pentru a modifica alinierea textului introdus în document se va modifica proprietatea *Alignment* care este specifică unui paragraf din document. Un paragraf se termină când se introduce caracterul de linie nouă în text; după care începe un nou paragraf. Astfel, se observă în figura 4.5 că textul *Stat de plată* este centrat în pagină. Deoarece au fost întroduse două linii vide, înseamnă că textul *Stat de plată* formează cel de-al treilea paragraf din document, care se centrează prin secvența:

```
wdoc.Paragraphs[3].Alignment =
Word.WdParagraphAlignment.wdAlignParagraphCenter;
```

Datele din controlul *gv* de tip *DataGridView* s-au exportat, în documentul Word, într-un tabel. Adăugarea unui tabel la un document se face prin intermediul unui obiect de tip *Range*. De aceea, s-a repoziționat obiectul de tip *Range* la sfârșitul celui anterior definit:

```
wr.Collapse(Word.WdCollapseDirection.wdCollapseEnd);
```

S-a construit un tabel (*ts*) având un număr de linii identic cu numărul de linii a controlului *DataGridView* (*gv*), în timp ce, la numărul de coloane a controlului a mai fost adăugată una, pentru a înscrie și numărul liniei din tabel.

```
Word.Table ts = wr.Tables.Add(wr, gv.RowCount, gv.ColumnCount+1);
```

Lățimea coloanelor a fost stabilită în raport de lățimea paginii (*latp*) care s-a calculat pornind de la lățimea paginii și a marginilor:

```
float latp = wdoc.PageSetup.PageWidth –
       (wdoc.PageSetup.RightMargin +  wdoc.PageSetup.LeftMargin);
```

Pentru stabilirea propriu-zisă a lațimii coloanelor s-a aplicat un procent spațiului disponibil (*latp*), astfel:

```
ts.Columns[1].Width = (float)0.10 * latp;
ts.Columns[2].Width = (float)0.70 * latp;
ts.Columns[3].Width = (float)0.20 * latp;
```

prima coloană are 10% din lațimea totală, cea de-a doua coloană are 70% iar ultima 20%.

Capul de tabel s-a definit prin scrierea textului corespunzător în fiecare celulă a primului rând. O celulă (*Cell*) conține proprietatea *Range* prin intermediul căreia se scrie textul în celulă:

```
ts.Cell(1, 1).Range.Text = "Nr. crt.";
ts.Cell(1, 2).Range.Text = gv.Columns[0].HeaderText;
ts.Cell(1, 3).Range.Text = gv.Columns[1].HeaderText;
```

Pentru a marca mai bine capul de tabel, primului rând din tabel i s-a asociat o culoare de fundal (culoarea gri):

```
ts.Rows[1].Shading.BackgroundPatternColor =
                                              Word.WdColor.wdColorGray15;
```

Popularea cu date a tabelului s-a realizat în următoarea secvență iterativă:

```
for (int i = 0; i < gv.RowCount - 1; i++)
  {
    ts.Cell(i + 2, 1).Range.Text = (i + 1).ToString();
    ts.Cell(i + 2, 1).Range.Paragraphs.Alignment =
                      Word.WdParagraphAlignment.wdAlignParagraphRight;
    ts.Cell(i + 2, 2).Range.Text =
                             gv.Rows[i].Cells[0].Value.ToString();
    ts.Cell(i + 2, 3).Range.Text =
                             gv.Rows[i].Cells[1].Value.ToString();
    ts.Cell(i + 2, 3).Range.Paragraphs.Alignment =
```

```
                    Word.WdParagraphAlignment.wdAlignParagraphRight;
  }
```

Tabelul se completează linie cu linie, fiecare linie se completează celulă cu celulă. Se observă că alinierea datelor din coloanele 1 și 3 se face la dreapta deoarece ele sunt numerice.

După popularea cu date, tabelul se formatează la nivel global în sensul că se stabilește dimensiunea fontului, și se îngroașă (*Bold*) doar capul de tabel:

```
ts.Range.Font.Size = 10;
ts.Range.Font.Bold = 0;
ts.Cell(1, 1).Range.Font.Bold = -1;
ts.Cell(1, 2).Range.Font.Bold = -1;
ts.Cell(1, 3).Range.Font.Bold = -1;
```

Se trasează liniile de demarcație (*Borders*) a celulelor tabelului:

```
ts.Borders.Enable = 1;
```

Fereastra aplicației *Word* care include și documentul nou construit devine vizibilă utilizatorului:

```
wapl.Visible = true;
```

iar acesta poate modifica, salva sau tipării documentul.

Realizarea interacțiunii cu documente Excel se va face folosind clase din ierarhia EOM (Excel Object Model). Se va avea în vedere că trebuie să se construiască o instanță a unei noi aplicații Excel, aceasta poate conține o colecție de documente care la rândul lor conțin colecții de foi de calcul. Accesul la datele efective ale unei foi de calcul se realizează prin obiecte de tip *Range* sau prin obiecte de tip celulă (*Cell*).

Ca exemplu, se consideră un document Excel care este structurat astfel încât să poată realiza calculul unei serii de date pornind de la o valoare inițială și de la un procent de creștere a valorii respective, care se va aplica anual, valorii aferente anului precedent. Perioada de timp, exprimată în ani, va constitui un alt parametru de intrare al modelului. Documentul Excel, se află în fișierul *sablon.xlsx* și este folosit ca motor de calcul pentru a obține rezultatele dorite. Astfel, foaia de calcul este organizată în modul următor:

- celule care conțin date primare; valorile introduse în aceste celule reprezintă parametrii de intrare ai modelului; și
- celule care conțin formule pentru a genera rezultatul, valorile acestor celule reprezintă parametrii de ieșire ai modelului.

Foaia de calcul este prezentată în figura 4.6 iar celulele care au culoarea de fundal gri sunt celule care conțin valori elementare ce reprezintă parametrii de intrare ai modelului:

- perioada pentru care se face estimarea (C1);
- procentul de creștere anuală (C2);
- anul inițial (E4);
- valoarea inițială (F4).

In funcție de perioadă, se completează celulele E5, E6, ... și celulele F6, F7, ... Pentru diversitate, prin programare, se va înscrie formula în celula E5 după care ea se va copia în toate celulele următoare, în concordanță cu perioada, pentru a genera anii seriei de timp, în timp ce pentru a estima valorile, în celula F5 s-a introdus, în document, formula de calcul urmând ca prin programare aceasta să fie copiată și în celelalte celule. Formula din celula F5 realizează o creștere a valorii inițiale cu procentul specificat prin parametrul de intrare existent în celula C2 și are forma: =F4+F4*C2/100.

Aplicația care controlează aplicația Excel este de tip *Windows Forms* și are controlale de tip *TextBox* pentru a se putea introduce parametrii modelului iar pentru vizualizarea rezultatelor s-a construit un control de tip *DataGridView* dupa cum se poate observa în figura 4.7.

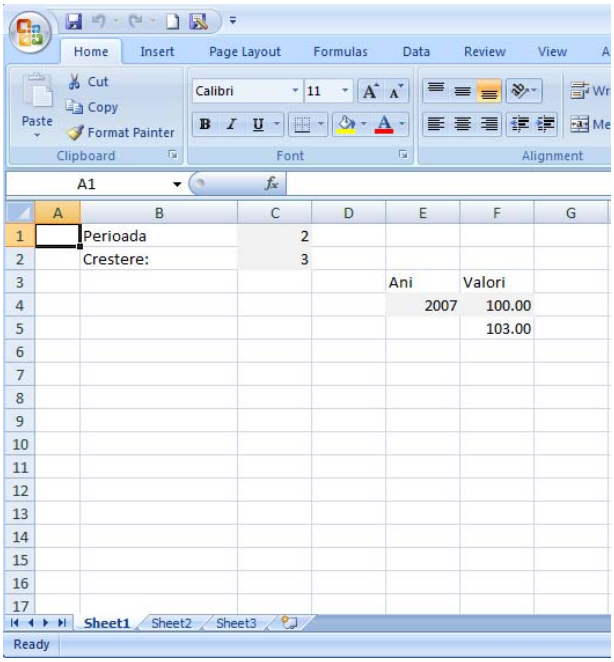

Figura 4.6 Foaie de calcul

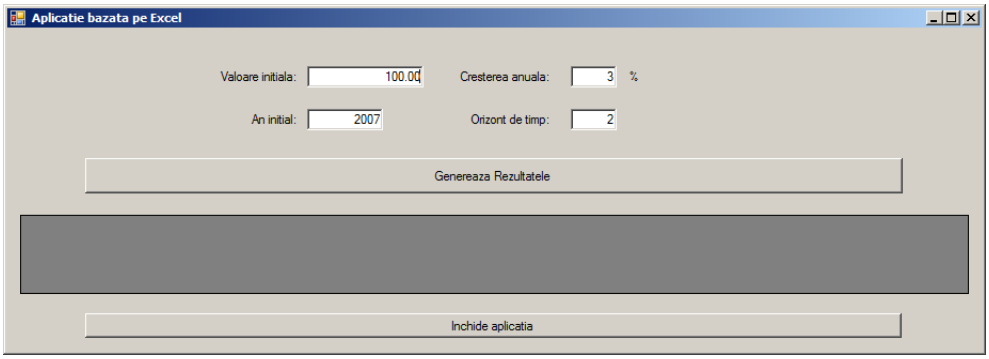

Figura 4.7 Forma aplicației principale

Folosirea modelului EOM impune adaugarea referinței:

```
Microsoft.Office.Interop.Excel
```

la proiect și totodată utilizarea domeniului corespunzător ceea ce presupune adăugarea în codul sursă a declarației:

```
using Excel = Microsoft.Office.Interop.Excel;
```

In cadrul clasei forma (*Form1*) se declară următoarele variabile:

```
Excel.Application eapl; // pentru a referi obiectul aplicație Excel
Excel.Workbook wb; // pentru a referi obiectul document Excel
Excel.Worksheet sh; // pentru a referi o foaie de calcul din document
```

pentru a fi vizibile și utilizabile din orice metodă a clasei formă.

La încărcarea formei se realizează deschiderea documentului *sablon.xlsx* și valorile din celulele în care se stochează parametrii de intrare sunt vizualizate în controalele de tip *TextBox* corespunzătoare:

```
//se construiește o nouă instanță a unei aplicații Excel
eapl = new Excel.Application();
//în cadrul aplicației se deschide documentul
wb = eapl.Workbooks.Open
            (@"d:\post_doc\documentatie\aplicatii\VSTO\Sablon.xlsx");
//se referă prima foie de calcul din document (Sheet 1)
sh = wb.Sheets[1] as Excel.Worksheet;
//se încarcă în controale valorile existente în celulele indicate
vi.Text = ((double)sh.get_Range("f4").Value).ToString("f");
ani.Text =((int)sh.get_Range("e4").Value).ToString();
ca.Text = ((int)sh.get_Range("c2").Value).ToString();
per.Text = ((int)sh.get_Range("c1").Value).ToString();
```

unde, *vi, ani, ca, per* sunt numele controalelor de tip *TextBox* din forma ilustrată în figura 4.7.

Se observă că referirea unei celule se face prin aplelul metodei *get_Range* care returnează un obiect de tip *Range* mapat pe un domeniu format dintr-o singură celulă. Stilul folosit pentru a adresa celula este de a indica coloana printr-o literă iar linia printr-un număr, de exemplu *f4*. Prin intermediul proprietății *Value* se accesează valoarea celulei care se convertește la un tip numeric, pentru că acele celule conțin valori numerice, după care ea se convertește la șir de caractere pentru a fi încărcată în controlul de tip *TextBox*.

Utilizatorul aplicației poate să modifice valorile inițiale prin editarea elementelor conținute în controalele de tip *TextBox* după care, pentru a genera rezultatele aferente noului set de valori, se apasă pe butonul *Genereaza Rezultate*. La apăsarea pe acest buton se realizează următoarele procesări:

- se rescriu în celulele corespunzătoare, din Excel, noile valori introduse:

```
int t,i
sh.get_Range("f4").Value = Convert.ToDouble(vi.Text);
sh.get_Range("e4").Value = Convert.ToInt32(ani.Text);
sh.get_Range("c2").Value = Convert.ToInt32(ca.Text);
sh.get_Range("c1").Value = t = Convert.ToInt32(per.Text);
```

- se generează anii seriei de timp (se pornește de la anul inițial și de la perioada analizată)

După cum se poate observa din figura 4.6, în celula e4, se stochează valoarea anului inițial. De aceea, pentru a genera anii perioadei analizate, pe coloană, în celula e5 se va înscrie o formulă care permite generarea valorii celulei prin adunarea cifrei 1 la valoarea celulei anterioare:

```
sh.get_Range("e5").Formula = "=e4+1";
```

astfel, prin intermediul proprietății *Formula*, celulelor unui domeniu (*Range*) li se asociază formule.

Pentru a genera anii întregii perioade, formula existentă în celula e5 se copiază în toate celulele domeniului. Un domeniu se adresează prin două celule: cea care marchează colțul stânga sus și cea care marchează colțul dreapta jos. In cazul în care domeniul este constituit dintr-o singură celulă atunci metodei *get_Range* i se furnizează doar un singur parametru care indică celula respectivă. Domeniul în care se va copia formula, existentă în celula e5, se stabilește dinamic în funcție de numărul de ani ai perioadei de referință. Astfel, domeniul este marcat de celula e5 și de celula e(4+t-1), unde, *t* reprezintă perioada:

```
sh.get_Range("e5").Copy(sh.get_Range
                                ("e5", "e" + (4 + t - 1).ToString()));
```

- generarea valorilor seriei de timp

In cadrul documentului care joacă rol de șablon, în celula f5 este deja scrisă formula de creștere a valorii inițiale cu procentul specificat așa că, pentru a genera valorile seriei se va face doar copierea formulei în domeniul indicat:

```
sh.get_Range("f5").Copy(sh.get_Range
                            ("f5", "f" + (4 + t - 1).ToString()));
```

- încărcarea seriei de timp din foaia de calcul în controlul de tip *DataGridView* (*gv*) din forma aplicației

In funcție de numărul de ani luați în considerare, se stabilește numărul de coloane a gridului:

```
for (i = 0; i < t; i++)
```

```
{
    gv.Columns.Add("c" + (i + 1).ToString(),
     ((int)sh.get_Range("e" + (4 + i).ToString()).Value).ToString());
    gv.Columns[i].HeaderCell.Style.Alignment =
                        DataGridViewContentAlignment.MiddleCenter;
}
```

Fiecare coloană va avea ca text, în *header*, anul care va fi centrat pe lățimea coloanei.

Valorile propriu-zise, se vor adăuga în grid, din celulele domeniului corespunzător din foia de calcul Excel și se vor alinia la dreapta:

```
gv.Rows.Add();
for (i = 0; i < t; i++)
{
    gv.Rows[0].Cells[i].Value =
((double)sh.get_Range("f" + (4 + i).ToString()).Value).ToString("f");
    gv.Rows[0].Cells[i].Style.Alignment =
                            DataGridViewContentAlignment.MiddleRight;
}
```

In figura 4.8 se prezintă forma aplicației pentru care s-a generat o serie de valori pentru un orizont de 5 ani.

In cazul în care utilizatorul dorește să vizualizeze și conținutul foii de calcul, atunci se poate executa și instrucțiunea: `eapl.Visible = true;` ce determină vizualizarea ferestrei aplicației Excel cu seria de timp generată, ca în figura 4.9.

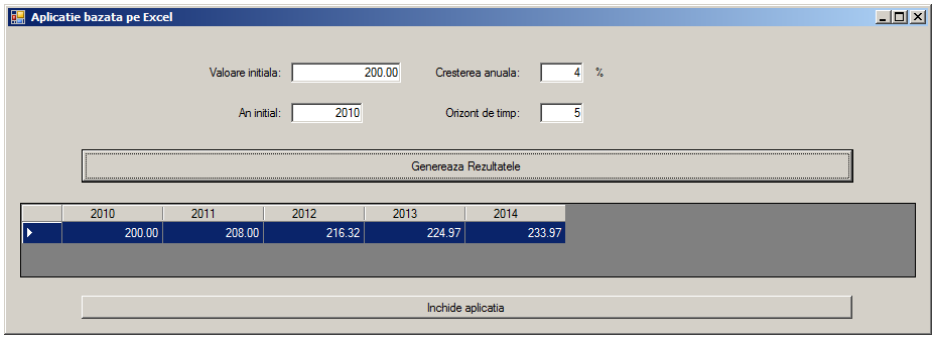

Figura 4.8 Generarea seriei de timp

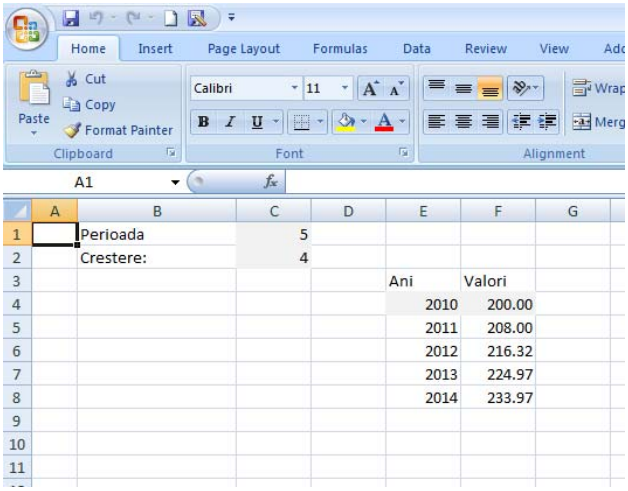

Figura 4.9 Foaia de calcul în care s-a generat seria de timp

- închiderea aplicației trebuie să realizeze închiderea explicită a instanței aplicației Excel altfel ea va ramâne deschisă în sistem lucru care se poate observa în fereastra aplicației *Task Manager*. Inainte de închiderea propriu-zisă a aplicației, utilizatorul trebuie să închidă documentul (fișierul *sablon.xlsx*) lucru care se poate face, fie cu salvarea în document a datelor, fie fără salvarea lor:

```
bool save = false;
if (MessageBox.Show("Doriti salvarea documentului?", "Salvati?",
      MessageBoxButtons.YesNo, MessageBoxIcon.Question) ==
                                                      DialogResult.Yes)
{
   save = true;
}
//inchiderea documentului
wb.Close(save);
//inchiderea aplicației Excel
eapl.Quit();
//inchiderea aplicației părinte
Application.Exit();
```

Tot în aceeași idee de dezvoltare se propune construirea unei aplicații care să construiască o prezentare *PowerPoint* pe care apoi să o treacă în regim *SlideShow* cu vizualizarea slide-urilor dinamic la un interval de 3 secunde. Construirea prezentării se realizează dintr-o aplicație de tip Windows Forms care permite construirea a două tipuri de slide-uri:

A. unul format din două obiecte care conțin text, unul fiind titlul slide-ului și celălalt cu conținutul propriu-zis;
B. unul care nu are nici un element predefinit (*Blank Slide*), elementele vor fi construite pe slide la momentul execuției după cum urmează: un titlu (element de tip TexBox) și o imagine (element de tip Picture) care va afișa o imagine preluată dintr-un fișier imagine.

Pe forma aplicației se află butonul *New PowerPoint* care instanțiază aplicația *Power Point* și creează o nouă prezentare. Butoanele *Adauga Slide* adaugă slide-uri de tipul asociat slide-ului iar butonul *Vizualizare* trece prezentarea în regim *SlideShow* și pornește derularea prezentării. Forma aplicației este prezentată în figura 4.10, completată cu informații care constituie conținutul prezentării formată din două slide-uri.

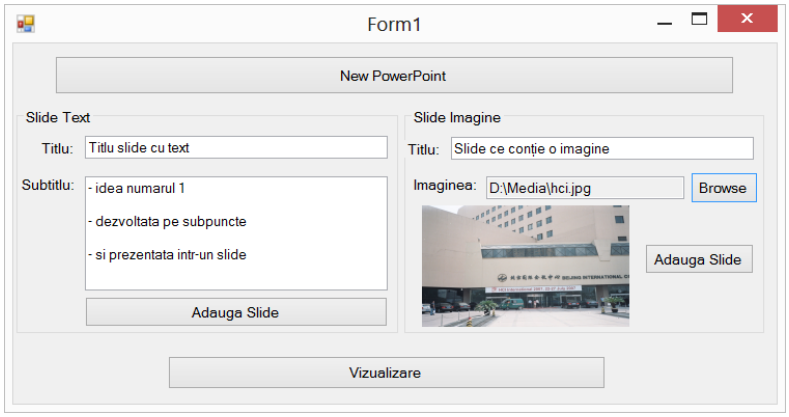

Figura 4.10 Forma aplicației ce generează o prezentare PowerPoint

Mai întâi se apasa butonul *New PowerPoint* care instanțiază o aplicație PowerPoint și construiește o nouă prezentare, după care se vor adăuga slide-uri. Apasând pe butonul *Adaugă Slide* din grupul *Slide Text* se va adăuga la prezentare slide-ul ce conține elemente de prezentare bazate pe text iar daca se apasă pe butonul de la grupul Slide Imagine se va adăuga la prezentare slide-ul ce conține titlul și imaginea selectată de pe disc și care este previzualizată în formă. Prezentarea construită este vizualizată în figura 4.11.

Din punct de vedere al implementării la aplicația de generare a prezentării s-au adăugat referințe către *Microsoft.Office.Interop.PowerPoint* și către *Office* iar în codul sursă:

```
using PP = Microsoft.Office.Interop.PowerPoint;
using Office = Microsoft.Office.Core;
```

In clasa *Form1* s-au adăugat explicit variabilele:

```
PP.Application papl;
PP.Presentation pprs;
```

```
PP.Slide pslide;
string calea; //stochează fisierul ce contine imaginea
```

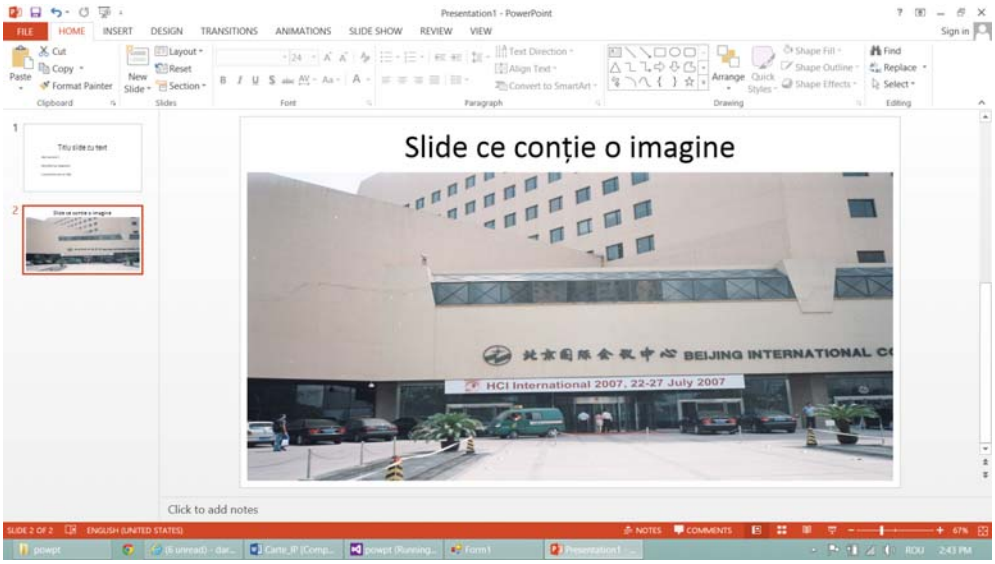

***Figura 4.11 Prezentarea PowerPoint* generată**

In constructorul clasei forma se instanțiază aplicația *PowerPoint* și se atribuie *null* la variabila *calea*.

```
public Form1()
{
    InitializeComponent();
    papl = new PP.Application();
    calea = null;
}
```

La click pe butonul *New PowerPoint* se construiește o nouă prezentare și aplicația *PowerPoint* devine vizibilă:

```
private void button1_Click(object sender, EventArgs e)
{
  pprs = papl.Presentations.Add(Office.MsoTriState.msoTrue);
}
```

La click pe butonul *Browse* se deschide o fereastră de dialog de tip *OpenFileDialog* care permite utilizatorului să aleagă fișierul ce conține imaginea care urmează a fi inclusă în prezentare. Imaginea aleasă va fi afișată și în formă în controlul *pbx* de tip *PictureBox* iar numele fișierului cu tot cu cale se va afișa în controlul *img_fis* de tip *TextBox*.

```
private void button4_Click(object sender, EventArgs e)
{
   OpenFileDialog fd = new OpenFileDialog();
```

```
   fd.InitialDirectory = "d:\\";
   fd.Filter = "Image Files(*.BMP;*.JPG;*.GIF)|*.BMP;*.JPG;*.GIF|
                    All files (*.*)|*.*";
   fd.FilterIndex = 1;
   fd.RestoreDirectory = true;
   if (fd.ShowDialog() == DialogResult.OK)
            {
                img_fis.Text = calea = fd.FileName;
                Bitmap img = new Bitmap(fd.FileName);
                pbx.Image = img;
            }
}
```

Construirea propriu-zisă a slide-urilor se face la apăsarea pe butoanele *Adaugă Slide*. Codul asociat celui care generează slide-uri numai cu controale de tip text are forma:

```
private void button2_Click(object sender, EventArgs e)
{
  int nrs = pprs.Slides.Count;
// se stabileste un tip de slide adica cel care are doua textbox-uri
// unul pentru titlu si altul pentru continut (ppLayoutTitle)
  PP.CustomLayout cl =
    pprs.SlideMaster.CustomLayouts[PP.PpSlideLayout.ppLayoutTitle];
// se adauga un slide de acest tip la prezentare
  pslide = pprs.Slides.AddSlide(nrs+1, cl);
// se seteaza textul asociat titlului din controlul (titlu) de tip
// textbox de pe forma aplicatiei
  pslide.Shapes.Title.TextFrame.TextRange.Text = titlu.Text;
// textul pentru continutul propriu-zis se preia din controlul stitlu
// de tip textbox de pe forma aplicatiei
  pslide.Shapes[2].TextFrame.TextRange.Text = stitlu.Text;
// se aliniaza textul din cea de-a doua formă la stanga
pslide.Shapes[2].TextFrame.TextRange.ParagraphFormat.Alignment =
            PP.PpParagraphAlignment.ppAlignLeft;
}
```

Butonul *Adaugă Slide* care construiește un slide ce conține titlul și imaginea pornește cu generarea unui slide fără nici un element după care se construiesc elementele grafice în care se va afișa titlul și apoi imaginea:

```
private void button5_Click(object sender, EventArgs e)
{
  int nrs = pprs.Slides.Count;
//tip de slide: blank slide (valoarea 7)
  PP.CustomLayout cl = pprs.SlideMaster.CustomLayouts[7];
//adaugare slide de acest tip
  pslide = pprs.Slides.AddSlide(nrs + 1, cl);
//preluarea latime si inaltime slide
  float lat = pslide.CustomLayout.Width;
  float inalt = pslide.CustomLayout.Height;
```

```
//adaugare textbox la slide cu localizare si directie de scriere
  pslide.Shapes.AddTextbox(
    Office.MsoTextOrientation.msoTextOrientationHorizontal,
    10, 10, lat-20, 60);
//setare text pentru titlu din controlul s2titlu de pe forma
  pslide.Shapes[1].TextFrame.TextRange.Text = s2titlu.Text;
  pslide.Shapes[1].TextFrame.TextRange.Font.Size = 48;
  pslide.Shapes[1].TextFrame.TextRange.ParagraphFormat.Alignment =
      PP.PpParagraphAlignment.ppAlignCenter;
//adaugare imagine pe slide cu localizare si dimensionare
  pslide.Shapes.AddPicture(calea, Office.MsoTriState.msoCTrue,
      Office.MsoTriState.msoFalse, 10, 80, lat - 20, inalt-90);
}
```

Pentru vizualizarea prezentării se apasă pe butonul *Vizualizare* de pe formă ce are asociat codul:

```
private void button3_Click(object sender, EventArgs e)
{
//stabilirea domeniului de slide-uri (toate)
   PP.SlideRange ssr = pprs.Slides.Range();
//stabilirea caracteristicilor de tranzitie slide-uri
   PP.SlideShowTransition sst = ssr.SlideShowTransition;
//derularea sa se realizeze in timp
   sst.AdvanceOnTime = Office.MsoTriState.msoTrue;
//intervalul dintre doua slide-uri de 3 secunde
   sst.AdvanceTime = 3;
//efect de tranzitie slide-uri
   sst.EntryEffect = PP.PpEntryEffect.ppEffectBlindsVertical;
//trecerea în modul SlideShow
   PP.SlideShowSettings ssp = pprs.SlideShowSettings;
//stabilirea slide de inceput a vizualizarii
   ssp.StartingSlide = 1;
//stabilirea slide de sfarsit
   ssp.EndingSlide = pprs.Slides.Count;
//vizualizare slide-uri in mod slideshow
   ssp.Run();
 }
```

4.3 Dezvoltarea de extensii pentru aplicațiile Office

Personalizarea prelucrărilor în cadrul aplicațiilor Office se realizează prin construirea de componente software care se atașează la cele oferite de aplicația de bază. Aceste componente poartă denumirea de *add-in* și necesită asocierea de elemente de interfață cu utilizatorul pentru ca acesta să poată declanșa operațiile implementate. In acest subcapitol, se prezintă modul în

care se construiesc astfel de module și de asemenea modul de organizare și de interacțiune cu ele din perspectiva versiunii 2007 a pachetului Office. Incepând cu versiunea Office 2007, aplicațiile Word, Excel etc folosesc un nou element de interfață cu utilizatorul, mai precis *ribbon*-ul (panglica) care substituie meniul aplicației. Acesta cuprinde mai multe grupuri de elemente de interacțiune, adică ceea ce la versiunile anterioare erau barele de instrumente, care conțin diverse controale, specializate, în funcție de natura interacțiunii cu utilizatorul.

Modalitatea oferită de aceste componente, în vederea personalizării procesărilor, presupune ca soluția să fie disponibilă la nivelul aplicației. Cu alte cuvinte, aplicația Office conține componenta care poate fi executată pe datele oricărui document se încarcă în aplicație.

Pentru exemplificare, se va construi un grup de operații care va include un buton folosit pentru a declanșa efectuarea sumei elementelor de pe diagonala principală a unei matrice de valori numerice. Matricea de valori se va marca prin selecția unui domeniu de celule dintr-o foaie de calcul a unui document *xlsx*. In aceste condiții, se poate deduce faptul că, se va construi o componentă add-in pentru aplicația Excel.

A. Se va construi un proiect în *Visual C#* din categoria *Office 2007* de tipul *Excel 2007 Add-in* având numele *Op_matrice* după cum se ilustrează în figura 4.12.

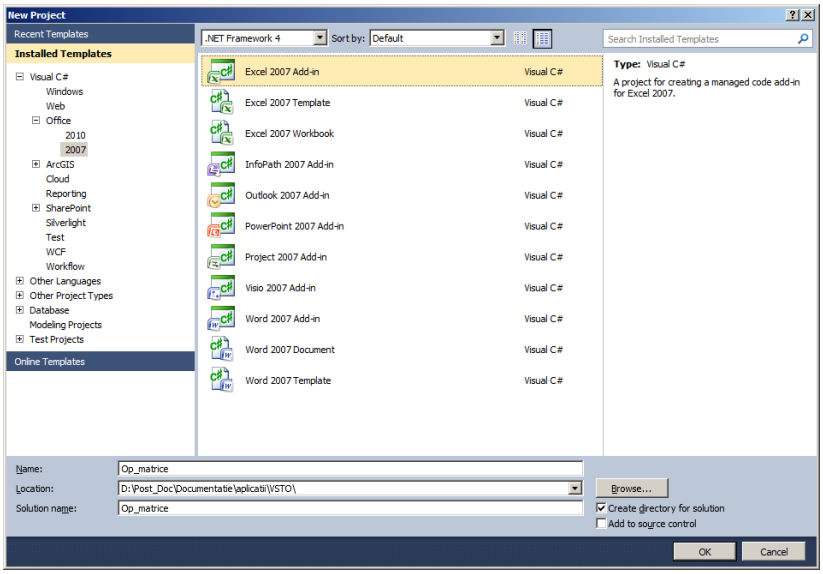

Figura 4.12 Tip de proiect Add-in pentru Excel

A. După construirea proiectului, în fereastra *Solution Explorer* se dă *click* dreapta pe numele proiectului și se adaugă un nou element la proiect (*Add/New Item*) și va apare fereastra ilustrată în figura 4.13.

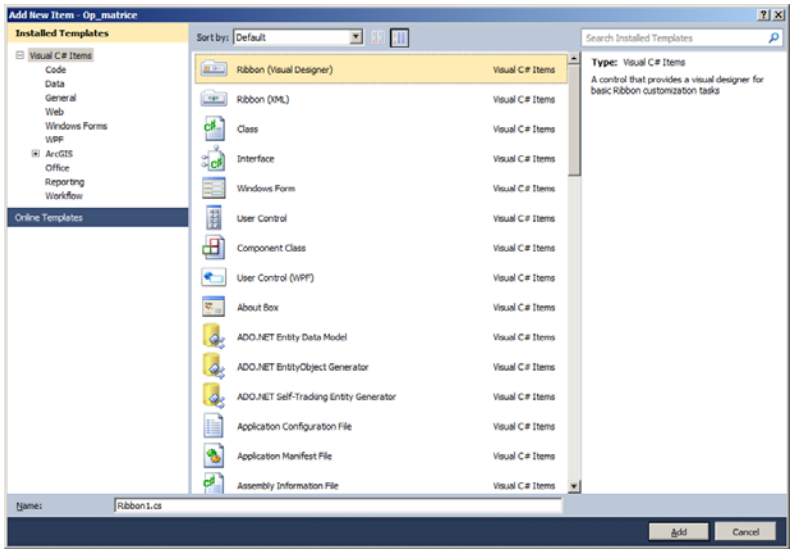

Figura 4.13 Adăugarea unui nou element la proiect

La proiect se va adăuga un nou element de tip panglică - *Ribbon (Visual Designer)*, pe care se vor insera elementele de interfață, mai precis, grupul de operații va conține butonul ce va declanșa prelucrarea. După cum se observă

în figura 4.14, dezvoltatorul are posibilitatea să-și personalizeze elementele de interacțiune din cadrul panglicii. Astfel, grupul de controale denumit *group1* poate fi personalizat prin intermediul proprietății *Label* dându-i un alt nume (*Operații matriceale*). Implicit, grupul nu are incluse controale însă ele pot fi ulterior adăugate din grupul de controale care se pot atașa panglicilor – a se observa fereastra *Toolbox* din figura 4.14. Din grupul de controale se adaugă în grupul de operații un control de tip *Button* (prin *drag and drop*), iar proprietatea *Label* a acestuia conține textul ce apare pe buton. Această proprietate se va modifica astfel încât, pe buton să apară textul: *Suma pe diagonala*.

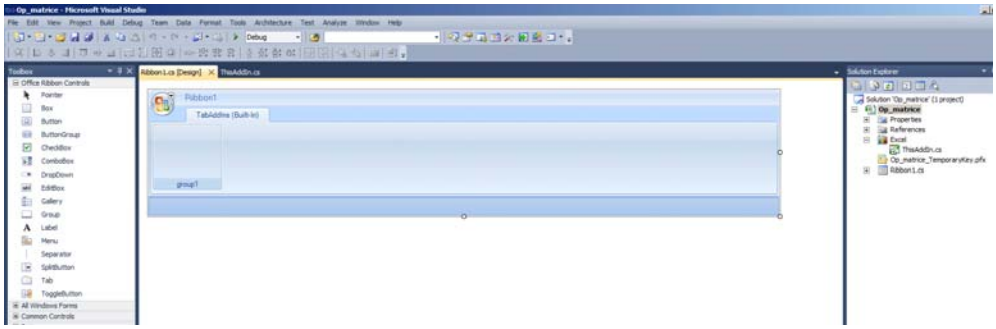

Figura 4.14 Proiectarea panglicii

După efectuarea acestor operații, se va executa proiectul, care va determina lansarea în execuția a aplicației Excel. In cadul aplicației Excel, se alege panglica *Add-Ins* în cadrul căreia se observă și grupul de operații construit anterior și care poartă numele *Operații matriceale*, după cum se observă în figura 4.15.

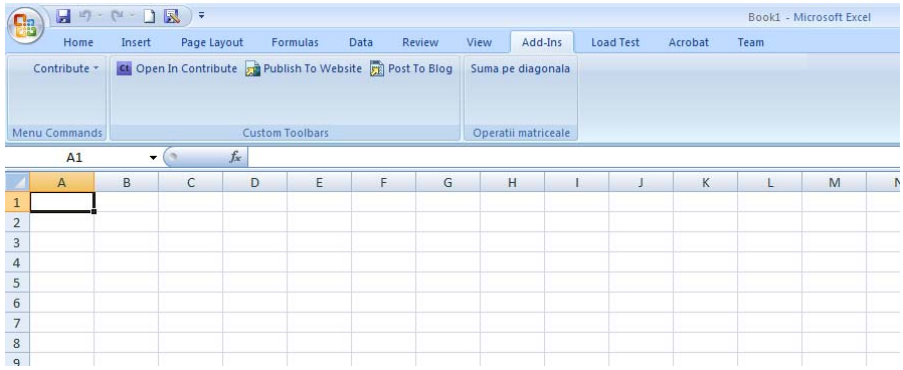

Figura 4.15 Personalizarea panglicii Add-Ins

A. Definirea funcționalității butonului creat se realizează prin tratarea evenimentului *Click*.

Acest lucru impune adăugarea metodei de tratare a evenimentului *Click* în clasa *Ribbon1*. In fișierul *Ribbon1.cs* se vor adăuga declarațiile:

```
using System.Windows.Forms;
using Excel = Microsoft.Office.Interop.Excel;
```

pentru a putea lucra cu clase din *Windows Forms* cum ar fi, de exemplu, *MessageBox* pentru afișarea de mesaje și pentru a putea folosi clase din *EOM*.

In clasa *Ribbon1* se va mai declara o variabilă statică de tip *object* care se va inițializa cu *Type.Missing*: `static object tm = Type.Missing;` Aceasta variabilă se folosește pentru a furniza parametrii care nu sunt obligatoriu de transmis în cadrul metodelor apelate.

Spre deosebire de modalitatea de dezvoltare a aplicațiilor bazate pe *Office*, prezentată la punctul 4.2, în acest caz, există o instanță a aplicației Excel, un document și o foaie de calcul. Cu alte cuvinte, aceste obiecte trebuie preluate și utilizate în aplicație:

```
Excel.Application eapl = Globals.ThisAddIn.Application;
Excel.Workbook wb = eapl.Workbooks[1];
Excel.Worksheet sh = wb.Sheets[1] as Excel.Worksheet;
```

In cadrul proiectului s-a generat clasa *Globals* care conține proprietatea *ThisAddIn* care, la rândul ei, conține instanța curentă a aplicației Excel. Pornind de la obiectul aplicație, se referă documentul (*Workbook*) și din cadrul documentului foaia de calcul (*Worksheet*).

Efectuarea operației de însumare a valorilor de pe diagonala principală a unei matrice impune selectarea domeniului de celule, din foaia de calcul, care definește matricea. Obiectul *Range* rezultat prin selecție se obține astfel:

```
Excel.Range rs = eapl.Selection as Excel.Range;
```

Pentru acest obiect de tip *Range* se preiau informații cu privire la numărul de linii și numărul de coloane:

```
int m = rs.Rows.Count, n = rs.Columns.Count, i;
```

variabila *i* se folosește drept contor.

In continuare, se verifică dacă matricea este pătratică:

```
if (m != n)
```

```
{
      MessageBox.Show("Matricea nu este patratica");
      return;
}
```

In cazul în care este pătratică se declară o variabilă în care să se memoreze suma elementelor de pe diagonala principală a matricei: `double s = 0;`

Se iterează elementele de pe diagonala principală pentru a le aduna. In cazul în care există celule care nu conțin valori numerice se aruncă o excepție și se termină procesarea:

```
for (i = 1; i <= n; i++)
    {
      try
       {
         s += (double)((Excel.Range)rs.Cells[i, i]).Value;
       }
      catch (Exception ex)
       {
         MessageBox.Show(ex.Message, "Eroare",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
         return;
       }
   }
```

Se observă că o celuă individuală din domeniu se referă prin colecția *Cells*, adresarea făcându-se prin furnizarea liniei și a coloanei. O celulă astfel adresată întoarce tot un obiect de tip *Range* care prin proprietatea *Value* întoarce valoarea stocată în celula respectivă.

Suma elementelor de pe diagonala principală va fi stocată, în foaia de calcul, în celula care urmează ultimei celule ce formează diagonala domeniului. In figura 4.16 se poate observa domeniul de celule selectat iar după apăsarea pe butonul *Suma pe diagonala* în celula E6, care continuă diagonala, s-a înscris rezultatul.

Un domeniu se referă furnizând celula din colțul stânga sus și celula din colțul dreapta jos. O celulă a unei foi de calcul se indică, în modul cel mai obișnuit, prin furnizarea coloanei prin literă (litere) și prin furnizarea liniei prin numărul ei, de exemplu E6. Un alt stil de a referi o celulă constă în a furniza linia și coloana prin numere, de exemplu R6C5, care referă tot celula E6. In contextul

aplicației, interesează care este ultima celulă (dreapta jos) a domeniului pentru a putea adresa celula următoare în vederea înscrierii rezultatului operației.

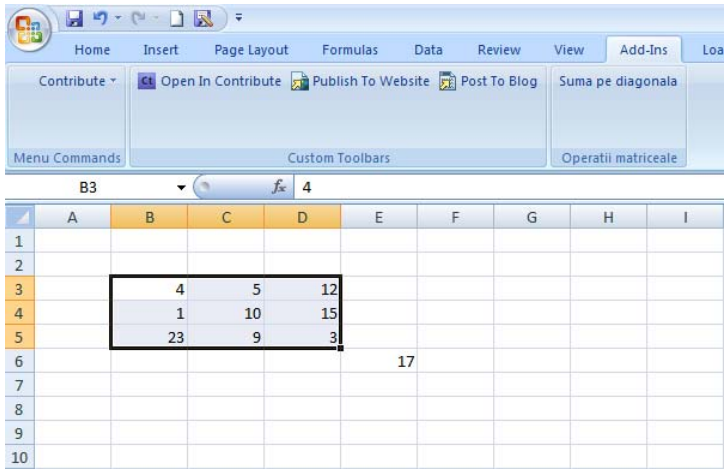

Figura 4.16 Efectuarea sumei pe diagonală cu înscrierea rezultatului

Prin secvența:

```
string adresa = rs.get_Address(tm, tm, Excel.XlReferenceStyle.xlR1C1);
```

se obține în șirul *adresa*, adresa domeniului selectat (*rs*) în stilul rând - coloană (*R1C1*). In exemplul prezentat în figura 4.16 șirul *adresa* are valoarea: *R3C2:R5C4*. Rezultatul operației se va înscrie în celula următoare pe diagonală, adică în celula *R6C5*. Pentru a calcula celula următoare pe diagonală s-a definit metoda `get_celula_dest` care primește șirul *adresa* și doi întregi, prin referință, cu rol de ieșire, care vor conține, după apel, numărul rândului și a coloanei unde se va înscrie rezultatul:

```
int a = 0, b = 0;
get_celula_dest(adresa, ref a, ref b);
```

Obținerea domeniului, prin metoda deja folosită, *get_Range*, folosește pentru a marca domeniul numai stilul de adresare cu literă și cifră (*E6*). Pentru a referi domeniul (în exemplul prezentat este vorba de o celulă), în stilul număr rând, număr coloană se folosește construcția:

```
sh.Range[sh.Cells[a, b], sh.Cells[a, b]]
```

care returnează un obiect de tip *Range* furnizând celula din colțul stânga sus – dreapta jos prin colecția *Cells* care permite referirea celulei propriu-zise prin

indicarea numărului liniei și a coloanei. Inscrierea propriu-zisă a valorii sumei în celulă se face:

```
sh.Range[sh.Cells[a, b], sh.Cells[a, b]].Value = s;
```

Metoda `get_celula_dest` are ca scop identificarea celulei în care se va stoca rezultatul operației. Ca parmetru de intrare primește șirul de caractere care conține adresa domeniului selectat iar parametrii de ieșire sunt furnizați sub formă de referință și întorc, sub forma a două numere întregi, linia și coloana în care se va înscrie rezultatul. Adresa domeniului se furnizează în stilul număr rând, număr coloană, șirul ce conține adresa este *parsat* astfel încât să se identifice numărul liniei și a coloanei pentru celula dreapta jos a domeniului selectat. In prima etapă se sparge șirul în două subșiruri, delimitatorul fiind caracterul *:* apoi se procesează al doilea subșir care referă celula din partea dreapta jos a domeniului. In cea de-a doua etapa se identifică subșirurile delimitate de caracterele *R* și *C* care furnizează numărul liniei și a coloanei care se convertesc din șir de caractere în numere întregi și se incrementează pentru a referi celula ce urmează, pe diagonală, domeniului selectat, celulă în care se va înscrie rezultatul operației.

```
private void get_celula_dest(string sir, ref int x, ref int y)
 {
  string[] vs = sir.Split(new char[] { ':' },
                    StringSplitOptions.RemoveEmptyEntries);
  string[] snc = vs[1].Split(new char[] { 'R', 'C' },
                    StringSplitOptions.RemoveEmptyEntries);
  x = Convert.ToInt32(snc[0]) + 1;
  y = Convert.ToInt32(snc[1]) + 1;
 }
```

4.4 Atașarea de cod documentelor Office

La punctul 4.3 a fost prezentată o modalitate de a adăuga o procesare personalizată unei aplicații Office care permite efectuarea operației asupra datelor unui document. In cadrul acestui subcapitol se vor definii operații personalizate care se vor putea efectua numai asupra datelor unui document Office bine precizat. Cu alte cuvinte, în primul caz este vorba de personalizare

la nivel de aplicație Office în timp ce, în al doilea caz este vorba de personalizare la nivel de document.

Datorită faptului că personalizarea procesării se face în acest caz la nivel de document atunci tot la nivelul documentului se va defini și încărca interfața cu utilizatorul pentru a putea declanșa aceste operații. In cazul personalizării procesării la nivelul aplicației s-a putut observa că interfața a fost construită la nivelul aplicației Office (sub forma unui grup de operații dintr-o panglică).

Exemplificarea unei astfel de aplicații bazate pe Office se va face atasând cod unui document Word 2007 în forma *docx*. Astfel, se construiește un șablon de document în care se vor insera controale specifice pentru un document Word pentru ca utilizatorul să-și poată personaliza conținutul documentului. Se va atașa documentului o fereastră de tip *Document Actions* care va conține două butoane cu funcționalitate personalizată: butonul *Adauga date in tabela* are ca scop preluarea datelor introduse în controalele inserate în document și stocarea lor într-o tabelă a unei bază de date Access, în timp ce, butonul *Incarca date in document* se va folosi pentru ca utilizatorul să vizualizeze datele stocate în tabela *Persoane*, din baza de date, sub formă tabelară, cu posibilitatea de a selecta un tuplu pentru a retrimite datele respective în documentul Word. Această modalitate de lucru combină facilitatea de machetare, ce permite utilizatorului sa-și personalizeze conținutul documentului, cu formatarea textului în cadrul unui document, atribut specific aplicațiilor care realizează editarea de texte. In practică, astfel de aplicații sunt utile mai ales în domeniile unde se construiesc fișe pentru diverse entități cum ar fi, de exemplu, fișa de înscriere a unui cursant la o școală de șoferi care are o parte fixă, nemodificabilă de la un cursant la altul cât și o parte ce depinde de datele personale ale individului, are un format impus, trebuie să fie ușor de listat pe suport hârtie și necesită includerea unor elemente neconvenționale, cum ar fi poza cursantului. Asocierea de cod unui astfel de document permite transformarea acestuia dintr-un document static într-un document dinamic și astfel, se oferă utilizatorului posibilitatea să

implementeze funcționalități specifice. In exemplul descris anterior se pot arhiva documentele prin stocarea datelor specifice fiecărui cursant într-o bază de date, partea fixă a documentului pastrându-se în fișierul *docx*.

Dezvoltarea unei astfel de aplicații în *Visual Studio 2010* presupune construirea unui proiect (*WordDocument1*) bazat pe *Office 2007* de tipul *Word 2007 Document*, după cum se ilustrează în figura 4.17.

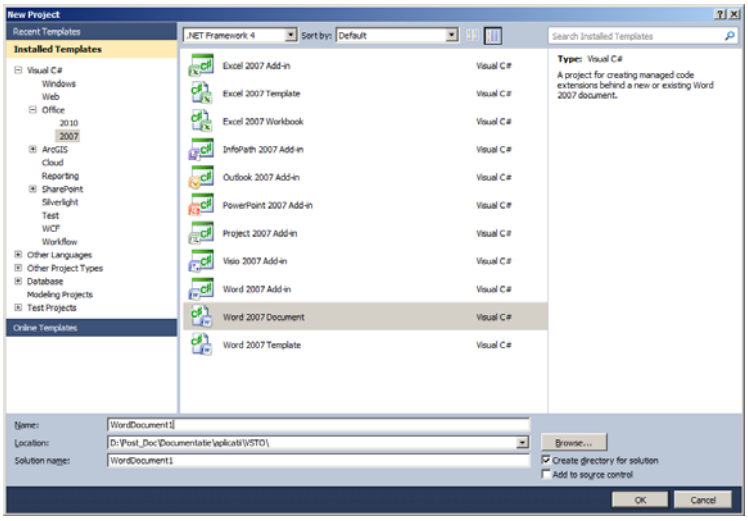

Figura 4.17 Construirea proiectului tip Word 2007 Document

Deoarece aplicația se atașează unui document atunci în pasul următor fie se indică documentul căruia i se atașează codul, fie se construiește un document nou prin interfața Visual Studio (figura 4.18).

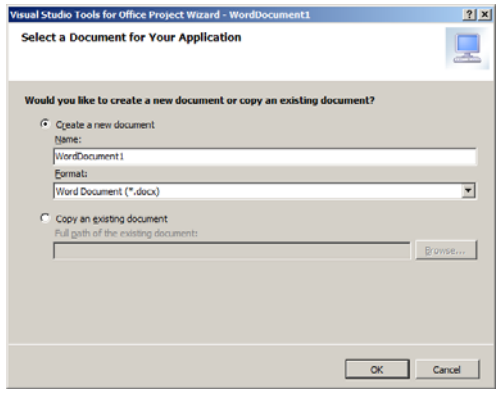

Figura 4.18 Alegerea documentului

In exemplu, s-a ales varianta construirii unui nou document (*WordDocument1.docx*) în interfața Visual Studio. Astfel, în modul proiectare, va apare interfața aplicației Word care permite editarea noului document după cum se poate observa în figura 4.19. Documentul este construit sub forma unui șablon în sensul că are o parte fixă reprezentată prin texte obișnuite și o parte ce se va personaliza cu informații despre persoane și care este reprezentată, în document, prin controale adecvate pentru interacțiune. Pentru simplificare, în exemplu, se propune un șablon de document pentru descrierea unei persoane prin: nume, sex și salariu. Acestor caracteristici li s-au asociat controale din grupul *Word Controls*, existent în fereastra *ToolBox*, în vederea editării datele după cum urmează: pentru câmpurile nume și salariu s-au folosit controale de tip *PlainTextContentControl* pentru a se putea edita valorile iar pentru sex s-a utilizat un control de tip *DropDownListContentControl* pentru a alege valoarea din mai multe variante. Pentru lizibilitate, în document, culoarea de fundal a controalelor este gri (figura 4.19). In mod vizual, controalelor li s-au modificat proprietățile:

- controlul pentru introducerea numelui a fost denumit *cnume*, prin schimbarea valorii proprietății *Name*, prin proprietatea *Text* a fost schimbat șirul de caractere afișat în control (*Vasile*);
- controlul pentru alegerea valorii câmpului sex poartă denumirea *csx*, prin intermediul proprietății *DropDownListEntries* a fost definită colecția de valori (*Masculin*, *Feminin*), prin intermediul proprietății *PlaceholderText* s-a definit șirul care apare vizualizat în control până la prima utilizare a lui (*Alege o optiune*);
- controlul pentru introducerea salariului a fost denumit *csal*;

Toate controalele au proprietatea *LockContentControl* cu valoarea *true* pentru ca utilizatorul să nu poată șterge controalele din document, ci să opereze doar asupra conținutului lor.

Documentul *WordDocument1.docx* are asociată clasa *ThisDocument* care permite programatorului să extindă funcționalitatea documentului și să

răspundă la evenimente declanșate în lucrul cu documentul. Această clasă conține în mod implicit definițiile metodelor: *ThisDocument_Startup* care se apelează la evenimentul de încărcare a documentului în aplicația Word și *ThisDocument_Shutdown* pe evenimentul de închidere document.

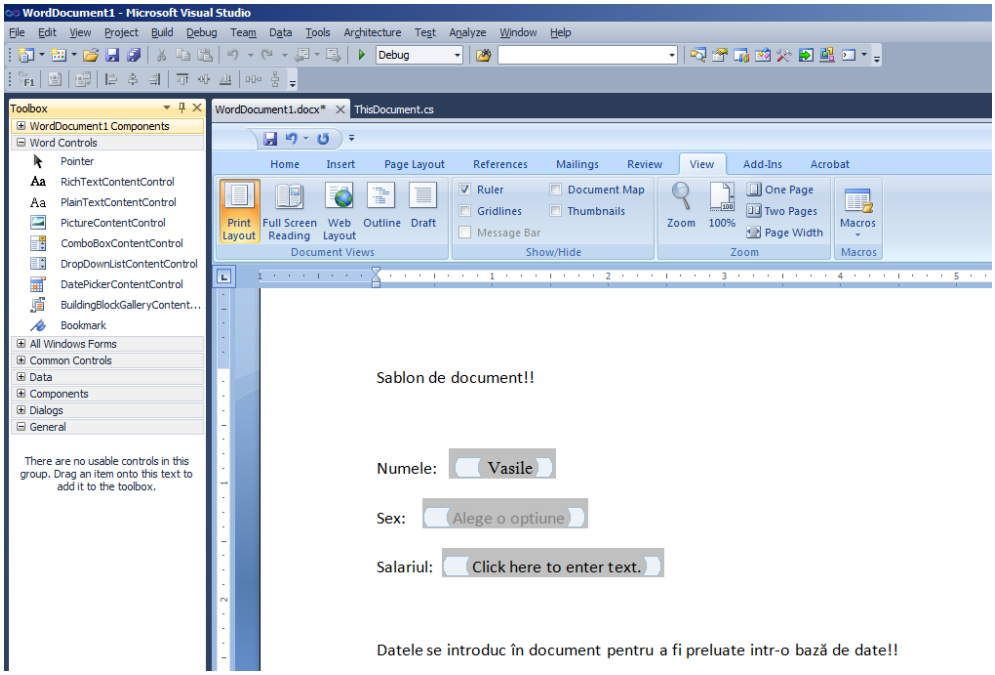

Figura 4.19 Editarea noului document

Datele care se vor introduce în controalele documentului se vor prelua și se vor stoca în baza de date Access *date_doc.accbd* mai precis, în tabela *Persoane* care are următoarea structură:

- *ID*, de tip *AutoNumber* care joacă rol de cheie primară;
- *nume* de tip șir de caractere;
- *sex* de tip șir de caractere având marimea de un caracter;
- *sal* de tip numeric (valori întregi).

Declanșarea operațiilor de stocare a datelor în baza de date și de încărcare a șablonului cu date din baza de date se face prin apăsarea a două butoane construite special în acest scop. In general, controalele de utilizator se pot construi și vizualiza pe o formă care apare odată cu documentul în aplicația Word și se numește *Document Actions*. Construirea acestei forme cât

și inițializarea controalelor incluse în documentul șablon se realizează pe evenimentul de încărcare a documentului (*Startup*). In cadrul clasei *ThisDocument* se fac declarațiile:

```
// doua obiecte de tip buton
Button badd, bincarcare;
// sir de conexiune la sursa de date
string sircc = @"Provider=Microsoft.ACE.OLEDB.12.0;
       Data Source=D:\post_doc\documentatie\aplicatii\date_doc.accdb";
```

Metoda care se aplelează ca răspuns la evenimentul de încărcare a documentului este definită după cum urmează:

```
private void ThisDocument_Startup(object sender, System.EventArgs e)
  {
// initializare proprietati aferente controalelor Word
       cnume.Text = "Aici introduce numele!";
       csal.Text = "0";
// construirea butonului pentru a adauga datele din document
// in baza de date
       badd = new Button();
// stabilirea textului ce va apare pe buton
       badd.Text = "Adauga date in tabela";
// asocierea metodei care se va apela ca urmare a evenimentului Click
       badd.Click += new EventHandler(badd_Click);
// adaugarea controlului pe forma asociata documentului
       Globals.ThisDocument.ActionsPane.Controls.Add(badd);
// construirea butonului pentru a incarca date din
// baza de date in document
       bincarcare = new Button();
// stabilirea textului ce va apare pe buton
       bincarcare.Text = "Incarca date in document";
// asocierea metodei care se va apela ca urmare a evenimentului Click
       bincarcare.Click += new EventHandler(bincarcare_Click);
// adaugarea controlului pe forma asociata documentului
       Globals.ThisDocument.ActionsPane.Controls.Add(bincarcare);
  }
```

Rularea aplicației determină încărcarea documentului șablon, în Word, după cum se observă în figura 4.20. Se observă că, în controalele aferente numelui și salariului s-au înscris valorile de inițializare în concordanță cu codul metodei *ThisDocument_Startup*. Pe de altă parte, se observă că, în partea din dreapta a documentului a apărut forma (*Document Actions*) cu cele două butoane definite în metoda descrisă anterior. Fereastra *Document Actions* se

poate deschide / închide apasând butonul *Document Actions* ce ține de grupul *Show/Hide* din panglica *View*.

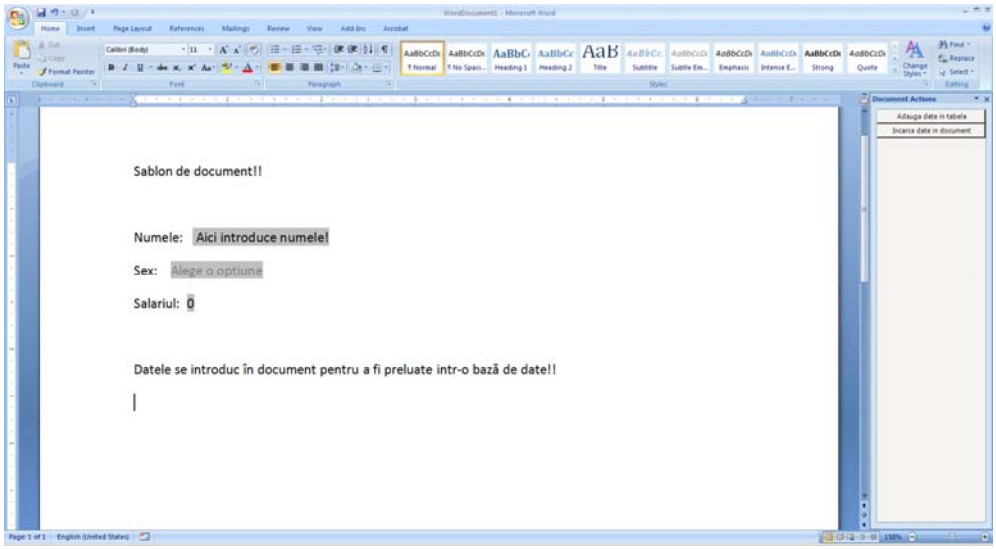

Figura 4.20 Sablon de document cu interfața asociată

După completarea cu date a controalelor din document (cele cu fundal gri), dacă se apasă pe butonul *Adauga date in tabela*, datele vor fi adăugate la cele existente în tabela *Persoane* din baza de date *date_doc.accdb*. Pentru întroducerea câmpului sex utilizatorul trebuie să aleagă o opțiune (*Masculin* sau *Feminin*). In control este afișat textul care joacă rol de etichetă și poate fi modificat prin proprietatea *Placeholder Text*. Acest text se păstrează vizibil doar până la prima utilizare a controlului după care va apare în control doar ultima opțiune selectată. In cazul în care utilizatorul nu selectează nici o opțiune, el este notificat să aleagă ceva din listă pentru a putea adăuga datele în baza de date. In figura 4.21 se prezintă și opțiunile existente în listă, ce apar doar când se interacționează cu controlul.

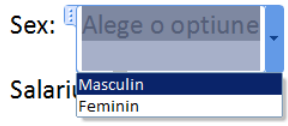

Figura 4.21 Alegerea opțiunilor

Codul care se execută la declanşarea evenimentului *Click* pe butonul *Adauga date in tabela* se află în metoda *badd_Click*:

```
void badd_Click(object sender, EventArgs e)
 {
// daca nu s-a ales nici o optiune,
// utilizatorul este atentionat sa o faca
   if (csx.Text == string.Empty)
              MessageBox.Show("Alegeti o valoare pentru sex!!!");
   else
     {
// se construieste un obiect de tip OleDbConnection
// care face conexiunea la sursa de date
       OleDbConnection con = new OleDbConnection(sircc);
// formarea şirului dintr-un singur caracter în urma optiunii
// introduse pentru sexul persoanei
       string ssx = (csx.Text=="Masculin") ? "M" : "F";
// construirea sirului ce contine comanda de inserare
// a tuplului în tabela Persoane
       string sinsert = @"INSERT INTO Persoane(nume, sex, sal) VALUES
              ('" + cnume.Text + "','" + ssx + "'," + csal.Text + ")";
//deschiderea conexiunii
       con.Open();
//construirea obiectului comanda
       OleDbCommand cinsert = new OleDbCommand(sinsert, con);
//executia comenzii de inserare
       cinsert.ExecuteNonQuery();
//inchiderea conexiunii
       con.Close();
//mesaj de confirmare pentru adaugarea datelor in tabela
       MessageBox.Show("Date adaugate in tabela!!!");
     }
 }
```

Preluarea datelor din baza de date şi încărcarea lor în documentul şablon se realizează prin apăsarea butonului *Incarca date in document* care va determina apariţia unei ferestre de dialog în care se va afişa un *DataGridView* din care utilizatorul îşi va putea selecta tuplul dorit. Primul pas în elaborarea acestei funcţionalităţi constă în a construi o nouă forma la proiect (*Project*/*Add New Item*) şi se selectează tipul de element - *Windows Form*, eventual se asociază un nou nume şi se apasă butonul *Add*. In cazul exemplului prezentat a fost păstrat numele implicit pentru formă (*Form1*). Vizual, formei i se adaugă elemente de interacţiune cu utilizatorul. Astfel, se adaugă un control de tip *DataGridView* (*gv*) care se va lega de tabela *Persoane* pentru

afișarea tuplurilor, câmpurile afișate fiind: nume, sex și salariu. Selecția unui tuplu din controlul de tip *DataGridView* marchează datele care se vor încărca în documentul Word, lucru care se realizează efectiv prin apăsarea butonului *Incarca datele in document* sau se anulează operația prin apăsarea butonului *Anulare*. Butoanele de pe formă au asociate valori pentru proprietatea *DialogResult*: butonul *Anulare* are asociată valoarea *Cancel* în timp ce butonul *Incarca datele in document* are asociată valoarea *OK*. Ambele prin apăsare determină închiderea ferestrei de dialog. Forma astfel construită este ilustrată în figura 4.22.

Metoda care se aplelează ca urmare a evenimentului *Click* pe butonul *Incarca date in document* este *bincarcare_Click*:

```
void bincarcare_Click(object sender, EventArgs e)
 {
//se construieste o noua forma
     Form1 fviz = new Form1();
//la apasarea pe butonul Incarca datele in document
     if (DialogResult.OK == fviz.ShowDialog())
     {
//se identifica rindul selectat
        DataGridViewRow rind = fviz.gv.SelectedRows[0];
// se preiau valorile câmpurilor din rindul selectat și
// se înscriu în controalele din documentul Word
        cnume.Text = (string)rind.Cells[0].Value;
        csal.Text = ((int)rind.Cells[2].Value).ToString();
// pentru controlul din document (csx), de tip DropDownList,
// se reface optiunea asa dupa cum
// apare ea în lista de optiuni a controlului
        string sitem = (string)rind.Cells[1].Value == "M" ?
                                          "Masculin" : "Feminin";
// se itereaza prin lista de optiuni a controlului csx
// pina este identificata cea preluata din grid și aceasta se
// selectează pentru a fi afișată în control
        int i;
        for(i=1;i<=csx.DropDownListEntries.Count;i++)
               if(sitem == csx.DropDownListEntries[i].Text)
                                csx.DropDownListEntries[i].Select();
     }
 }
```

Operația de preluare a datelor din baza de date și încărcarea lor în documentul Word se ilustrează în figurile 4.23 și 4.24. Astfel, apăsarea pe butonul *Incarca date in document* din forma *Document Actions* a

documentului Word determină apariția ferestrei de dialog construită în proiect având controlul de tip grid populat cu datele din tabela *Persoane* a bazei de date *date_doc.accdb*.

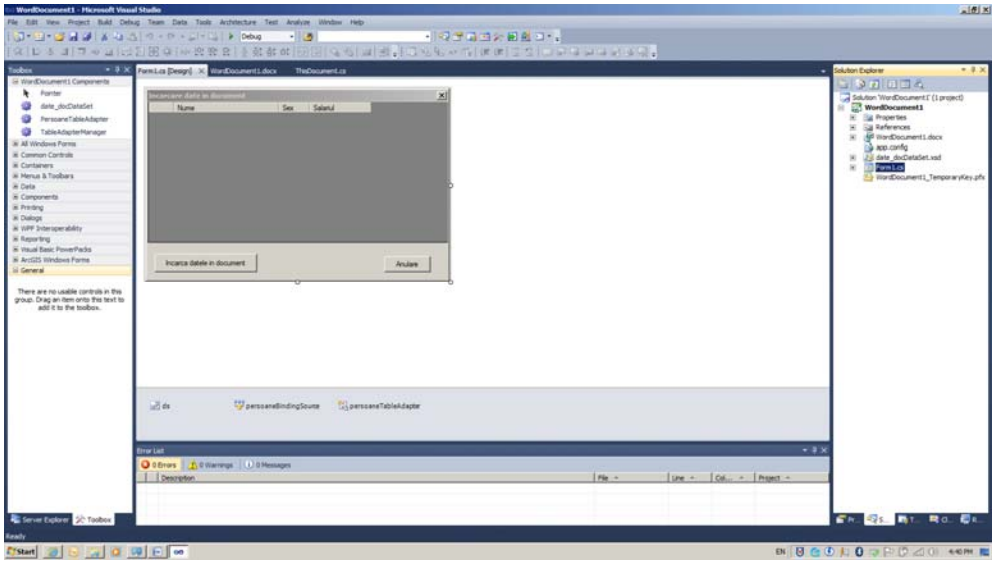

Figura 4.22 Fereastra de dialog pentru selectia datelor

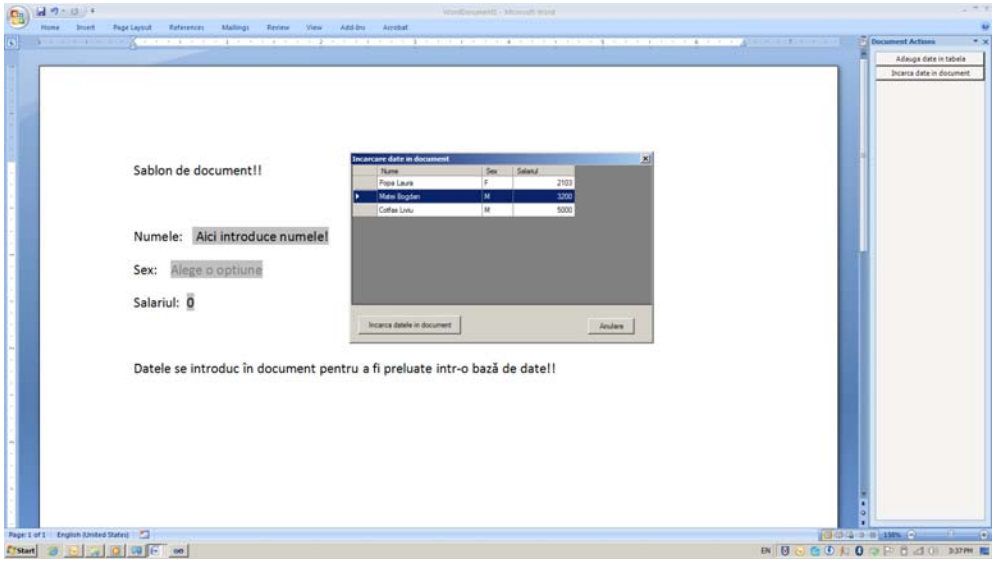

Figura 4.23 Vizualizarea datelor și selecția tuplului

Utilizatorul poate selecta un tuplu din tabelă, prin intermediul grid-ului (figura 4.23) și apoi are două opțiuni: una să anuleze operația, în acest caz apasă pe

butonul *Anulare* sau să încarce datele selectate, în document, prin apăsarea pe butonul *Incarca datele in document*. După apăsarea butonului de încărcare a datelor în document se închide fereastra de dialog iar datele selectate se încarcă, în document, în câmpurile corespunzătoare, după cum se poate observa în figura 4.24.

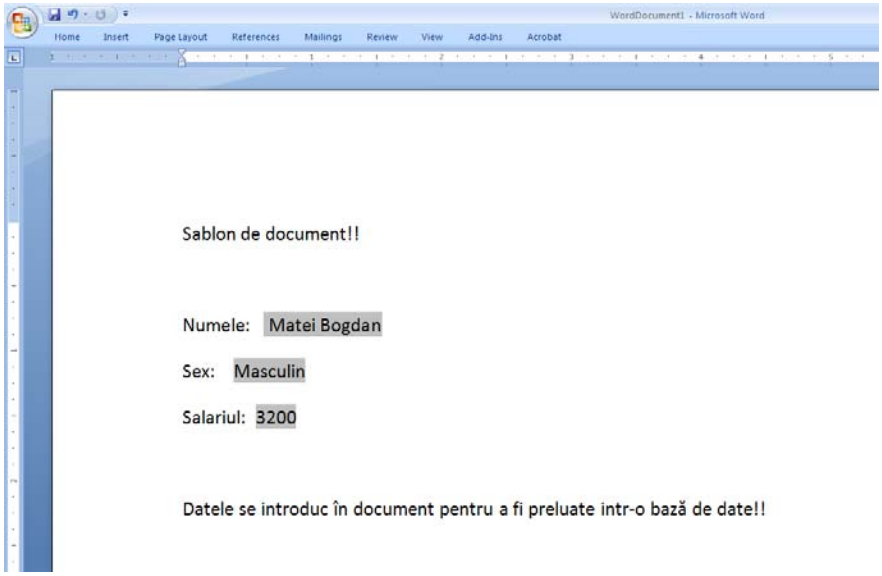

Figura 4.24 Incărcarea datelor selectate în documentul șablon

Bibliografie

1. ***, *Exploring ArcObjects*, ESRI Press, 2002;
2. ***, *Understanding Map Projections*, ESRI Press, 2000;
3. Baker, M., *What is a Software Framework? And why should you like 'em?*, http://info.cimetrix.com/blog/?Tag=software+framework, posted 2009;
4. Bruney, A., *Professional VSTO 2005*, Wiley Publishing, Inc., 2006;
5. Burke, R., *Getting to Know ArcObjects: Programming ArcGIS with VBA*, ESRI Press, 2003;
6. Carter, E., Lippert, E., *Visual Studio Tools for Office 2007*, Addison-Wesley, 2009;
7. Chang, K.-T., *Programming ArcObjects with VBA*, CRC Press, 2008;
8. Dârdală, M., *Accessing Excel files using XML format from Director multimedia applications,* The 3rd International Workshop IE & SI, Facultatea de Stiinţe Economice, Universitatea de Vest, Timişoara, 2006, Editura Mirton, Timişoara, 2006;
9. Dârdală, M., *Communication between C#.NET applications and Excel*, The Proceedings of the Seventh International Conference on Informatics in Economy, ASE Bucureşti, 2005, în volumul *Information & Knowledge Age*, ASE, Bucureşti, 2005;
10. Dârdală, M., Tuşa, E., *Designing Informatics Systems Using Economic Models Defined by Excel Spreadsheets,* Economy Informatics, vol. VI, nr. 1-4, Editura INFOREC, Bucureşti, 2006;
11. Dârdală, M., *Implementarea structurilor de date dinamice ca obiecte de tip colecţie în C#,* Sesiunea anuală de comunicări ştiinţifice, Universitatea Europeană „Drăgan", Lugoj, Editura Dacia Europa Nova, Lugoj, 2004;
12. Dârdală, M., Reveiu, A., *Using Resources in Visual C#.NET Applications*, Economy Informatics, vol V, nr. 1-4, Editura INFOREC, Bucureşti, 2005;
13. Dârdală, M., Reveiu, A., Smeureanu, I., *Using DLL as Interface between API and VC#.NET Applications,* Informatica Economică, vol. X, nr. 1, Editura INFOREC, Bucureşti, 2006;
14. Dârdală, M., Smeureanu, I., Reveiu, A., *Tehnologii multimedia*, Editura ASE, Bucureşti, 2008;

15. Dârdală, M., Smeureanu, I., *Tehnologii de acces la date. ASP.NET*, Editura ASE, Bucureşti, 2008;
16. Hillier, S., *Microsoft SharePoint – Building Office 2007 Solutions in C# 2005*, Apress, 2007;
17. Novak, I., Velvart, A., Granicz, A., Balassy, G., Hajdrik, A., Sellers, M., Hillar, G., Molnar, A., Kanjilal, J., *Visual Studio 2010 and .NET4 SIX-IN-ONE*, Wiley Publishing, Inc., 2010;
18. Reveiu, A., *Techniques for Representation of Regional Clusters in Geographical Information Systems*, Revista Informatică Economică, nr. 1/2011, ISSN 1453-1305, pag 129-139;
19. Rimmer, S., *Multimedia Programing for Windows*, McGraw-Hill, 1994;
20. Smeureanu, I., Dârdală, M., *Multimedia Programming Objects,* Al cincilea Simpozion de Informatică Economică, A.S.E., Bucureşti, 2001;
21. Smeureanu, I., Dârdală, M., Reveiu, A., *Visual C#.NET,* Editura CISON, Bucureşti, 2004;
22. Thangaswamy, V., *VSTO 3.0 for Office 2007 Programming*, PACKT Publishing, 2009;
23. http://labs.cs.upt.ro/labs/lft/html/LFT00.htm
24. http://cursuri.cs.pub.ro/~cpl/Curs/CPL-Curs01.pdf
25. http://mszalai.xhost.ro/html/capitolul_ii1.html
26. http://mcpc.cigas.net/progenv.html