Interfața grafică cu utilizatorul

1. Interfața cu utilizatorul

Interfața cu utilizatorul permite acestuia să interacționeze cu aplicația. Există două tipuri principale de interfețe:

- interfață cu utilizatorul de tip *caracter* (engl. "character user interface", *CUI*);
- interfață cu utilizatorul de tip *grafic* (engl. "graphical user interface", *GUI*).

Un exemplu de interfață de tip caracter este interfața la linia de comandă a sistemului de operare MS-DOS. Când se folosește o astfel de interfață, în general utilizatorul trebuie să memoreze și să tasteze comenzi text. De aceea, interfețele de acest tip pot fi mai dificil de utilizat și necesită o oarecare pregătire a utilizatorului. O interfață grafică încearcă să simplifice comunicarea. Graficele reprezintă obiecte pe care utilizatorul le poate manipula și asupra cărora poate efectua acțiuni. Deoarece utilizatorul nu trebuie să știe un limbaj de comandă, o interfață grafică bine proiectată este mai ușor de folosit decât o interfață de tip caracter.

1.1. Proiectarea comenzilor şi interacţiunilor

Dacă ierarhia de comenzi trebuie să se integreze într-un sistem de interacțiuni deja existent, trebuie mai întâi studiat acesta.

Se stabilește o ierarhie inițială de comenzi care poate fi prezentată utilizatorilor în mai multe moduri: o serie de opțiuni dintr-un meniu, o bară de instrumente (*toolbar*) sau o serie de imagini (*icons*).

Apoi, această ierarhie se rafinează prin ordonarea serviciilor din fiecare ramură a ierarhiei în ordinea logică în care trebuie să se execute, cele mai frecvente servicii apărând primele în listă.

Lățimea și adâncimea ierarhiei trebuie proiectate în așa fel încât să se evite supraîncarea memoriei de scurtă durată a utilizatorului. De asemenea, trebuie minimizat numărul de pași sau de acțiuni (apăsări ale *mouse*-ului, combinații de taste) pe care trebuie să le efectueze acesta pentru a-și îndeplini scopul.

Interacțiunile cu factorul uman pot fi proiectate pe baza următoarelor criterii:

- *coerența*: se recomandă utilizarea unor termeni și acțiuni cu semnificații unice, bine precizate și care se regăsesc în mod unitar în tot sistemul;
- *numărul mic de pași*: trebuie să se minimizeze numărul de acțiuni pe care trebuie să le îndeplinească utilizatorul;

- *evitarea „aerului mort”* (engl. “dead air”): utilizatorul nu trebuie lăsat singur, fără niciun semnal, atunci când așteaptă ca sistemul să execute o acțiune. El trebuie să știe că sistemul execută o acțiune și cât din acțiunea respectivă s-a realizat;
- *operațiile de anulare* (engl. “undo”): se recomandă furnizarea acestui serviciu datorită erorilor inerente ale utilizatorilor;
- *timpul scurt și efortul redus de învățare*: de multe ori, utilizatorii nu citesc documentația, de aceea se recomandă furnizarea în timpul execuției a unor soluții pentru problemele apărute;
- *aspectul estetic al interfeței:* oamenii utilizează mai ușor un produs software cu un aspect plăcut.

Se recomandă folosirea de icoane și controale similare celor din produsele software cu care utilizatorul este familiarizat. Dacă are de-a face cu același aspect exterior, acesta își va folosi cunoștințele anterioare pentru navigarea prin opțiunile programului, ceea ce va reduce și mai mult timpul de instruire.

1.2. Considerente practice

Elementele interfeței grafice trebuie să fie coerente, adică să aibă stiluri, culori și semnificații similare în toată aplicația. Un tabel centralizat de cuvinte cheie poate ajuta la alegerea textelor sau etichetelor de către proiectanții interfeței care lucrează la același sistem. Aceast tabel conține lista de cuvinte folosite în toată interfața și care înseamnă aceeași lucru peste tot.

O interfață clară este ușor de înțeles. Metaforele utilizate trebuie să fie în acord cu experiența utilizatorilor în legătură cu obiectele din lumea reală pe care le reprezintă. De exemplu, icoana unui coș de gunoi poate reprezenta o funcție de gestionare a fișierelor nedorite (gen *Recycle Bin*): fișierele pot fi introduse în coș, unde rămân și pot fi regăsite până când coșul este golit.

Erorile trebuie identificate imediat folosind un mesaj inofensiv. Utilizatorul trebuie să-și poată repara o greșeală ori de câte ori este posibil acest lucru iar documentația programului sau manualul de utilizare trebuie să îl ajute în acest sens. Mesajele critice trebuie folosite numai atunci când utilizatorul trebuie avertizat că unele date pot fi pierdute sau deteriorate dacă nu acționează imediat.

Chiar dacă nu sunt probleme, furnizarea informațiilor despre starea sistemului face interfața mai prietenoasă. Totuși, aceste mesaje pot deveni supărătoare dacă sunt folosite prea des. Pot fi afișate și mesaje despre cursul unei acțiuni sau despre timpul după care se va termina o activitate. Dacă o acțiune durează aproximativ 6-8 secunde, se poate folosi un indicator de tip clepsidră. Pentru activități mai lungi de 8 secunde, se pot folosi indicatoare de tip procentaj sau timp rămas. Utilizatorii trebuie să aibă de asemenea posibilitatea întreruperii sau anulării acestor acțiuni de durată.

Mesajele sonore trebuie în general evitate deoarece pot fi supărătoare, distrag atenția iar semnificația sunetelor poate depinde de contextul cultural al utilizatorilor.

Culoarea joacă un rol important în proiectarea unei interfețe grafice. Folosirea corectă a culorilor face interfața clară și ușor de navigat. Totuși, dacă nu sunt folosite cu atenție, culorile pot distrage atenția utilizatorului.

Culorile pot fi utilizate pentru a identifica părțile importante ale interfeței. Prea multe culori strălucitoare fac textul dificil de citit. Trebuie de asemenea evitat un fundal complet alb și nu trebuie folosite mai mult de 4 culori într-o fereastră. Interpretarea culorilor este foarte subiectivă. Poate depinde de asociații culturale, psihologice și individuale. Deci, în general, cel mai bine este să folosim culori subtile și neexagerate. Utilizatorii presupun de multe ori că există o legătură între obiectele de aceeași culoare, așa că trebuie să fim atenți să nu folosim aceeași culoare pentru obiecte fără legătură.

Culorile nu trebuie să fie singura sursă de informații deoarece unii utilizatori nu pot distinge anumite culori, iar alții pot avea monitoare care nu suportă o gamă largă de culori.

Ca și culorile, **icoanele** pot pune în valoare o interfață grafică, dacă sunt folosite corect. Icoanele bine proiectate oferă utilizatorului un mod accesibil de comunicare cu aplicația. Există câteva elemente de care trebuie să ținem seama la proiectarea acestora:

- un stil și o dimensiune comună pentru toate icoanele dau interfeței un aspect coerent;
- desenele ajută utilizatorul să recunoască metaforele și să-și amintească funcțiile;
- conturarea icoanelor cu negru le face să iasă în evidență din fundal;
- icoanele pot fi afișate în trei mărimi: 48 x 48, 32 x 32 și 16 x 16 pixeli; acestea trebuie să fie ușor de recunoscut chiar și atunci când sunt afișate la dimensiunea de 16 x 16 pixeli;
- deși o icoană poate fi accentuată prin culoare, ea trebuie să poată fi recunoscută și în varianta alb-negru.

Icoanele bine proiectate își comunică funcțiile cu claritate și cresc utilizabilitatea interfeței, însă o icoană neclară poate deruta utilizatorii și poate crește numărul de erori. Putem folosi etichete text pentru a ne asigura că semnificația unei icoane este clară. Câteodată este cel mai bine să folosim imagini tradiționale deoarece utilizatorul este familiarizat cu ele. O icoană bine proiectată trebuie să poată fi distinsă cu ușurință de celelalte icoane din jurul său, iar imaginea trebuie să fie simplă și potrivită contextului interfeței.

Corpurile de literă (*font*-urile) utilizate într-o interfață grafică nu trebuie amestecate. Ca și în cazul culorilor, prea multe *font*-uri pot distrage atenția utilizatorului.

1.3. Profilurile utilizatorilor

Pentru a crea o interfață grafică utilizabilă, trebuie să cunoaștem profilul utilizatorului, care descrie așteptările și nevoile acestuia. Un mod potrivit de a determina profilul utilizatorului este prin observare la locul său de muncă. Poate fi folositoare sugestia ca utilizatorul „să gândească cu voce tare” atunci când lucrează cu prototipul unei interfețe.

Aproape întotdeauna va exista un procent de utilizatori începători iar interfața trebuie să aibă grijă de aceștia. De exemplu, putem asigura alternative la acceleratori (combinații de taste pentru anumite funcții) și putem preciza *shortcut*-urile în opțiunile meniurilor.

Profilurile utilizatorilor se încadrează în general în trei categorii:

- *Utilizatorul comod* dorește să folosească interfața imediat, cu foarte puțin antrenament. Acest tip de utilizator preferă utilizarea *mouse*-ului, atingerea sensibilă a ecranului sau stiloul electronic. Navigarea simplă este importantă deoarece utilizatorul comod nu ține minte căi complicate. Afișarea unei singure ferestre la un moment dat simplifică navigarea. Pentru a face o interfață grafică accesibilă unui utilizator de acest tip, ea trebuie să se bazeze pe recunoașterea unei icoane, mai degrabă decât pe amintirea a ceea ce reprezintă icoana. Acest lucru se poate realiza prin folosirea unei multitudini de grafice și de opțiuni în meniuri;
- *Utilizatorul rapid* dorește un timp de răspuns cât mai mic, așa încât trebuie evitate prea multe redesenări ale ferestrelor. Acest tip de utilizator preferă în general să folosească tastatura și mai puțin *mouse*-ul. Utilizatorii de acest tip au în general timp pentru instruire și sunt dispuși să renunțe la facilități în favoarea vitezei. Acceleratorii le permit să lucreze mai repede;
- *Utilizatorul energic* este de nivel avansat și are experiență cu interfețele grafice. Acesta nu își dorește o instruire de durată și se așteaptă să folosească interfața imediat. Deoarece este sigur pe sine și îi place să exploreze, trebuie întotdeauna asigurată o opțiune de anulare (engl. "undo"). Alte trăsături pe care le așteaptă sunt schimbări limitate ale modului de afișare, multitasking și posibilitatea particularizării și individualizării aspectului interfeței grafice.

2. Realizarea programelor cu interfața grafică cu utilizatorul în Microsoft Visual Studio .NET

Când este creat un nou proiect C# de tip *Windows Application*, în mijlocul ecranului (figura 1), apare un „formular" (Form) – fereastra principală a programului, în care se vor adăuga diverse componente de control: butoane, *text-box*-uri etc.

În partea din stânga a ecranului există o bară de instrumente (*View → Toolbox* sau *Ctrl+Alt+X*) din care se aleg cu *mouse*-ul componentele ce trebuie adăugate în fereastră.

Pentru adăugarea unei componente, programatorul va face click cu *mouse*-ul pe imaginea corespunzătoare din *toolbox*, apoi va face click în formular, în locul unde dorește să apară componenta respectivă. Odată introduse în fereastră, componentele pot fi mutate, redimensionate, copiate sau șterse. În dreapta este o fereastră de proprietăți (*View → Properties Window* sau *F4*). De aici, fiecărei componente folosite i se pot modifica proprietățile, adică aspectul exterior, așa cum va apărea în program, sau caracteristicile funcționale interne. De asemenea, se pot selecta evenimentele corespunzătoare componentei care vor fi tratate în program.

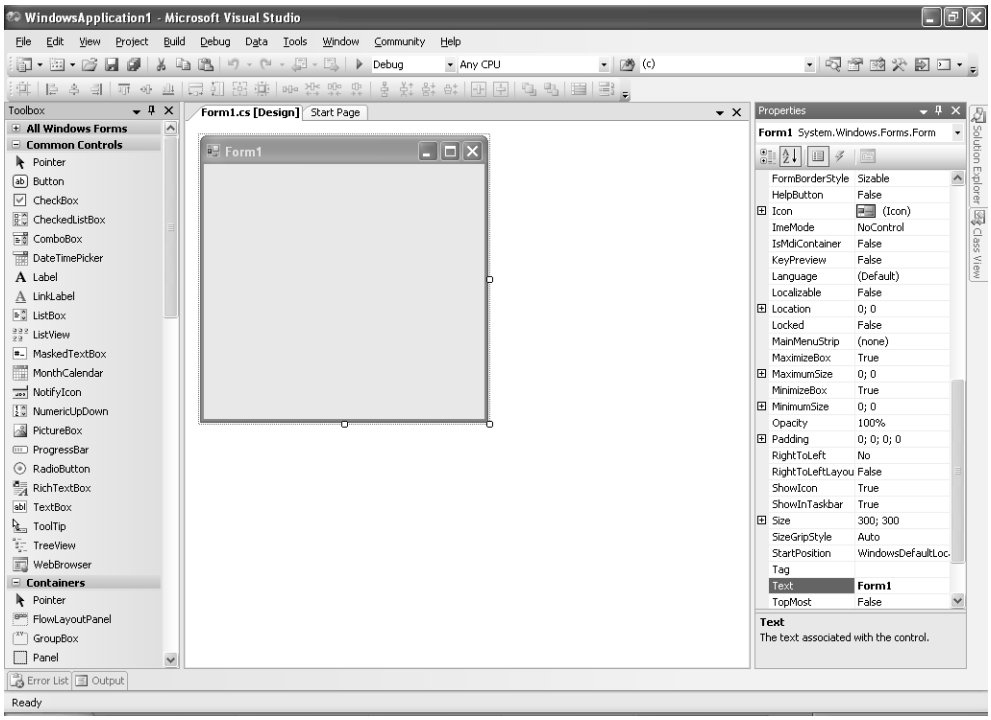


Figura 1. *Mediul de dezvoltare Microsoft Visual Studio*

În continuare, vor fi prezentate câteva componente de control folosite practic în orice program Windows. Pentru fiecare componentă, vor fi amintite unele proprietăți și metode uzuale. Pentru o descriere mai amănunțită, se recomandă consultarea documentației *MSDN*.

Application

Clasa Application încapsulează o aplicație Windows. Clasa conține metode și proprietăți statice pentru managementul unei aplicații, cum ar fi metode pentru pornirea și oprirea programului, prelucrarea mesajelor Windows și proprietăți corespunzătoare informațiilor despre aplicație.

Se poate observa că în scheletul de program creat implicit de mediul de dezvoltare, în metoda Main() este pornit programul pe baza clasei corespunzătoare ferestrei principale:

```
Application.Run(new MainForm());
```

Form

Clasa System.Windows.Forms.Form corespunde unei ferestre standard. O aplicație poate avea mai multe ferestre – una principală, câteva secundare și câteva ferestre de dialog.

Unele proprietăți:

- Icon – icoana care apare în bara de titlu a ferestrei;
- FormBorderStyle – înfățișarea și comportamentul chenarului, de exemplu, dacă fereastra poate fi redimensionată;

- Text – titlul ferestrei, care apare în bara de titlu și în *taskbar*;
- StartPosition – locul unde apare fereastra pe ecran;
- Size – dimensiunea (înălțimea și lățimea ferestrei); de obicei se stabilește prin redimensionarea ferestrei cu *mouse*-ul, în procesul de proiectare.

Câteva evenimente:

- Load, Closed – pentru diverse inițializări în momentul creării ferestrei sau prelucrări în momentul închiderii acesteia;

În general, pentru tratarea unui eveniment în C#, este selectat mai întâi obiectul de tipul dorit (la noi fereastra), apoi în fereastra de proprietăți se alege *tab*-ul de evenimente și se identifică evenimentul căutat.

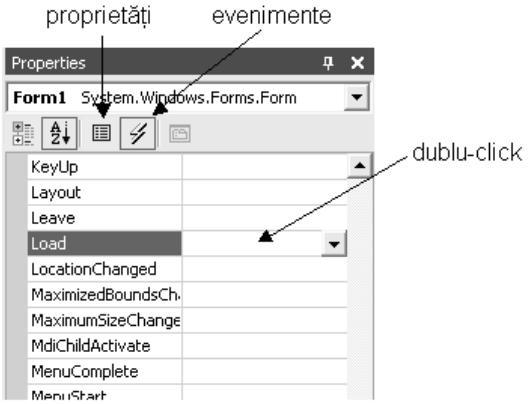


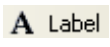
Figura 2. *Editarea proprietăților componentelor*

După un dublu-click, ca în figura 2, se va crea automat o nouă metodă vidă corespunzătoare evenimentului, iar utilizatorul va trebui numai să scrie în corpul funcției comenzile dorite.

Button 

Clasa System.Windows.Forms.Button corespunde unui buton. Câteva proprietăți și evenimente:

- Text – textul înscris pe buton;
- Click – funcția executată când butonul este apăsat.

Label 

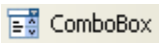
Clasa System.Windows.Forms.Label înscrie un text undeva în fereastră. Una din proprietăți:

- Text – textul înscris.

TextBox 

Clasa System.Windows.Forms.TextBox corespunde unei căsuțe de editare de text. Câteva proprietăți și evenimente:

- Text – textul din căsuță (de tip string);
- Multiline – textul poate fi introdus pe o singură linie (*false*) sau pe mai multe (*true*);
- ScrollBars – indică prezența unor bare de derulare (orizontale, verticale) dacă proprietatea Multiline este *true*;
- Enabled – componenta este activată sau nu (*true / false*);
- ReadOnly – textul poate fi modificat sau nu de utilizator (*true / false*);
- CharacterCasing – textul poate apărea normal (*Normal*), numai cu litere mici (*Lower*) sau numai cu litere mari (*Upper*);
- TextChanged – evenimentul de tratare a textului în timp real, pe măsură ce acesta este introdus.

ComboBox 

Clasa System.Windows.Forms.ComboBox corespunde unui *combo-box*, care combină un *text-box* cu o listă. Câteva proprietăți și evenimente:

- Text – textul din partea de editare;
- Items – lista de obiecte din partea de selecție, care se poate introduce și prin intermediul ferestrei de proprietăți;
- SelectedIndex – numărul articolului din listă care este selectat (0 – primul, 1 – al doilea, etc., –1 dacă textul din partea de editare nu este ales din listă);
- TextChanged, SelectedIndexChanged – evenimente de tratare a schimbării textului prin introducerea directă a unui nou cuvânt sau prin alegerea unui obiect din listă.

MenuStrip 

Clasa System.Windows.Forms.MenuStrip corespunde meniului principal al unei ferestre. Mod de folosire:

- se introduce o componentă de acest tip în fereastră;
- se editează meniul, direct în fereastră sau folosind proprietățile;
- pentru separatori se introduce în câmpul Caption un minus („ – ”);
- literele care se vor a fi subliniate trebuie precedate de „&”;
- pentru implementarea metodei de tratare a unei opțiuni din meniu se va face dublu-click pe aceasta (sau pe evenimentul Click în fereastra de proprietăți).

Timer 

Clasa System.Windows.Forms.Timer încapsulează funcțiile de temporizare din Windows. Câteva proprietăți și evenimente:

- Tick – evenimentul care va fi tratat o dată la un interval de timp;
- Interval – intervalul de timp (în milisecunde) la care va fi executat codul corespunzător evenimentului Tick;
- Enabled – indică dacă *timer*-ul e activat sau nu (*true / false*).

string

Este o clasă care permite lucrul cu șiruri de caractere.
Există operatorul +, care permite concatenarea șirurilor:

```
string str1 = "Microsoft ";
string str2 = "Word";
str1 = str1 + str2; // sau str1 += str2; => str1 == "Microsoft Word"
```

Clasa conține multe proprietăți și metode utile, dintre care amintim:

- **int** Length – lungimea șirului;
- **int** IndexOf(...) – poziția în șir la care apare prima dată un caracter sau un subșir;
- **string** Substring(...) – returnează un subșir;
- **string** Remove(**int** startIndex, **int** count) – returnează șirul rezultat prin ștergerea a *count* caractere din șir, începând cu poziția *startIndex*;
- **string**[] Split(...) – împarte șirul în mai multe subșiruri delimitate de anumite secvențe de caractere.

O metodă statică a clasei este Format(...), care returnează un șir de caractere corespunzător unui anumit format. Sintaxa este asemănătoare cu cea a funcției *printf* din C. De exemplu:

```
double d = 0.5;
string str = string.Format("Patratul numarului {0} este {1}", d, d * d);
```

Același rezultat s-ar fi putut obține astfel:

```
str = "Patratul numarului " + d.ToString() + " este " + (d * d).ToString();
```

Orice obiect are metoda ToString(), care convertește valoarea sa într-un șir. Pentru obiecte definite de programator, această metodă poate fi suprascrisă.

Dacă în exemplul de mai sus $d = 0.72654$ și dorim să afișăm numerele numai cu 2 zecimale, metoda Format își dovedește utilitatea (figura 3).

```
string str=string.Format("Patratul numarului {0:F2} este {1:F2}", d, d*d);
MessageBox.Show(str);
```

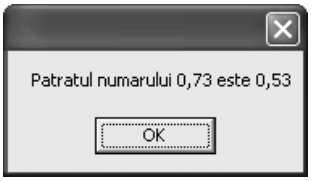


Figura 3. *Mesaj cu numere formatate*

StringBuilder

Majoritatea programatorilor utilizează de obicei clasa string când operează cu șiruri de caractere. În cazul concatenării unor șiruri, folosirea clasei StringBuilder aduce o importantă creștere de performanță. Să considerăm următorul bloc:

```
string listaNumere = "";
for (int i=0; i<1000; i++)
   listaNumere = listaNumere + " " + (i + 1).ToString();
```

În acest caz, se creează un nou obiect string la fiecare atribuire, adică de 1000 de ori!

Codul echivalent folosind StringBuilder este următorul:

```
StringBuilder sbListaNumere = new StringBuilder(10000);
for (int i = 0; i < 1000; i++)
{
   sbListaNumere.Append(" ");
   sbListaNumere.Append((i+1).ToString());
}
string listaNumere2 = sbListaNumere.ToString();
```

Pentru a avea acces la clasa StringBuilder, trebuie inclus *namespace*-ul System.Text la începutul programului:

```
using System.Text;
```

Directivele using sunt asemănătoare directivelor #include din C/C++. *Namespace*-urile sunt colecții de clase predefinite sau definite de programator. Dacă nu se dorește utilizarea unui *namespace*, clasa trebuie referită cu tot cu *namespace*. În cazul de față:

```
System.Text.StringBuilder sbListaNumere = new System.Text.StringBuilder(10000);
```

Ca dezavantaje, trebuie să menționăm că la inițializarea unui obiect StringBuilder performanțele scad. De asemenea, sunt foarte multe operații care nu se pot efectua cu StringBuilder, ci cu string. Pentru concatenarea a 2 șiruri nu este nevoie de StringBuilder. Se poate recomanda utilizarea acestei clase dacă e nevoie de concatenarea a cel puțin 4 șiruri de caractere. De asemenea, pentru creșterea vitezei, trebuie să estimăm lungimea finală a șirului (parametrul constructorului). Dacă folosim constructorul vid sau lungimea estimată este mai mică decât în realitate, alocarea spațiului se face automat iar performanța scade.

3. Elemente de C#

3.1. Clase parțiale

În clasele corespunzătoare ferestrelor, există metode predefinite generate automat de mediul Visual Studio, care conțin inițializarea controalelor grafice. Aceste metode sunt de obicei de dimensiuni mai mari și sunt separate de logica aplicației fiind plasate într-un alt fișier, numit *.Designer.cs.

Definiția unei clase, structuri sau interfețe se poate împărți în două sau mai multe fișiere sursă. Fiecare fișier sursă conține o parte din definiția clasei și toate părțile sunt combinate când aplicația este compilată. Există câteva situații când este utilă împărțirea definiției clasei:

- Când se lucrează la proiecte mari, împărțirea clasei în fișiere separate permite mai multor programatori să lucreze la ele simultan;
- Când se lucrează cu surse generate automat, codul poate fi adăugat clasei fără să mai fie nevoie de recrearea fișierului sursă. Visual Studio folosește această cale când creează ferestrele Windows, codul *wrapper* pentru servicii web ș.a.m.d. Programatorul poate scrie codul care folosește aceste porțiuni fără să mai fie nevoie de modificarea fișierului creat de Visual Studio.

Pentru a împărți definiția unei clase, se folosește cuvântul cheie partial, după cum este prezentat mai jos:

```
public partial class Student
{
   public void Learn()
   {
   }
}

public partial class Student
{
   public void TakeExam()
   {
   }
}
```

Modificatorul partial nu este valabil în declarațiile pentru funcții delegat sau enumerări.

3.2. Proprietăți. Accesori

Proprietățile sunt membri care furnizează un mecanism flexibil de citire, scriere, sau calculare de valori ale câmpurilor private. Ele pot fi folosite ca și cum ar fi membri publici, dar ele sunt de fapt metode speciale, care permit ca datele să fie accesate mai ușor însă în același timp păstrează siguranța și flexibilitatea metodelor.

Proprietățile combină aspecte specifice atât câmpurilor cât și metodelor. Pentru utilizatorul unui obiect, o proprietate apare ca un câmp, deoarece accesul la proprietate necesită exact aceeași sintaxă. Pentru realizatorul clasei, o proprietate este formată din unul sau două blocuri de cod, reprezentând un accesor get și/sau un accesor set. Blocul de cod pentru accesorul get este executat când proprietatea este citită, iar blocul de cod pentru accesorul set este executat când proprietății îi este atribuită o valoare.

O proprietate fără accesorul set este considerată *doar pentru citire* (*read-only)*. O proprietate fără accesorul get este considerată *doar pentru scriere* (*write-only)*. O proprietate cu ambii accesori este pentru *citire și scriere (read-write)*.

3.2.1. Accesorul *get*

Corpul accesorului get este similar cu cel al unei metode și trebuie să returneze o valoare de tipul proprietății. Execuția accesorului get este echivalentă cu citirea valorii câmpului corespunzător. În următorul exemplu, este prezentat un accesor get ce întoarce valoarea unui câmp privat _name:

```
class Person
{
   private string _name;    // câmp

   public string Name       // proprietate
   {
      get
      {
         return _name;
      }
   }
}
```

Din exteriorul clasei, citirea valorii proprietății se realizează astfel:

```
Person p1 = new Person();
Console.Write(p1.Name);  // se apelează accesorul get
```

Accesorul get trebuie să se termine cu instrucțiunea return sau throw, iar fluxul de control nu poate depăși corpul accesorului.

Când se întoarce doar valoarea unui câmp privat și sunt permise optimizările, apelul către accesorul get este tratat *inline* de către compilator în așa fel încât să nu se piardă timp prin

efectuarea unui apel de metodă. Totuși, un accesor get virtual nu poate fi tratat *inline* întrucât compilatorul nu știe în timpul compilării ce metodă va fi de fapt apelată în momentul rulării.

Schimbarea stării unui obiect utilizând accesorul get nu este recomandată. De exemplu, următorul accesor produce ca efect secundar schimbarea stării obiectului de fiecare dată când este accesat câmpul _number.

```
private int _number;

public int Number
{
   get
   {
      return _number++;      // !!! nerecomandat
   }
}
```

Accesorul get poate fi folosit fie ca să returneze valoarea unui câmp, fie să o calculeze și apoi să o returneze. De exemplu:

```
class Student
{
   private string _name;

   public string Name
   {
      get
      {
         return _name != null ? _name : "NA";
      }
   }
}
```

În secvența de cod anterioară, dacă _name este null, proprietatea va întoarce totuși o valoare, și anume *"NA"* (engl. "No Account").

3.2.2. Accesorul *set*

Accesorul set (numit și *mutator*) este similar cu o metodă ce întoarce void. Folosește un parametru implicit numit value (valoare), al cărui tip este tipul proprietății. În exemplul următor, este adăugat un accesor set proprietății Name:

```
class Person
{
   private string _name;

   public string Name
   {
      get
      {
         return _name;
      }
      set
      {
         _name = value;
      }
   }
}
```

Când se atribuie o valoare proprietății, este apelat accesorul set cu un argument ce furnizează noua valoare. De exemplu:

```
Person p1 = new Person();
p1.Name = "Joe";  // se apelează accesorul set cu value = "Joe"
```

3.2.3. Aspecte mai complexe ale lucrului cu proprietăți

În exemplul următor, clasa TimePeriod reține o perioadă de timp în secunde, dar are o proprietate numită Hours care permite unui client să lucreze cu timpul în ore.

```
class TimePeriod
{
   private double _seconds;

   public double Hours
   {
      get { return _seconds / 3600; }
      set { _seconds = value * 3600; }
   }
}

class Program
{
   static void Main()
   {
      TimePeriod t = new TimePeriod();

      // atribuirea proprietății Hours determină apelul accesorului set
      t.Hours = 24;
```

```
      // evaluarea proprietății Hours determină apelul accesorului get
      Console.WriteLine("Timpul in ore: " + t.Hours);
   }
}
```

În mod implicit, accesorii au aceași vizibilitate, sau nivel de acces cu al proprietății căreia îi aparțin. Totuși, uneori li se poate restricționa accesul. De obicei, aceasta implică restricționarea accesibilității accesorului set, în timp ce accesorul get rămâne public. De exemplu:

```
public string Name
{
   get
   {
      return _name;
   }
   protected set
   {
      _name = value;
   }
}
```

Aici, accesorul get primește nivelul de accesibilitate al proprietății însăși, public în acest caz, în timp ce accesorul set este restricționat în mod explicit aplicând modificatorul de acces protected.

Proprietățile au mai multe utilizări: pot valida datele înainte de a permite o modificare, pot returna date când acestea sunt de fapt provenite din alte surse, precum o bază de date, pot produce o acțiune când datele sunt modificate, cum ar fi invocarea unui eveniment sau schimbarea valorii altor câmpuri.

În exemplul următor, Month este o proprietate:

```
public class Date
{
   private int _month = 7;

   public int Month
   {
      get
      {
         return _month;
      }
      set
      {
         if ((value > 0) && (value < 13))
         {
            _month = value;
         }
      }
   }
}
```

Aici, accesorul set garantează că valoarea câmpului _month este între 1 și 12. Locația reală a datelor proprietății este de obicei denumită „memorie auxiliară” (engl. “backing store”). Marea majoritate a proprietăților utilizează în mod normal câmpuri private ca memorie auxiliară. Câmpul este marcat privat pentru a se asigura faptul că nu poate fi accesat decât prin apelarea proprietății.

O proprietate poate fi declarată statică folosind cuvântul cheie static. În acest fel proprietatea devine disponibilă apelanților oricând, chiar dacă nu există nicio instanță a clasei.

Spre deosebire de câmpuri, proprietățile nu sunt clasificate ca variabile și de aceea nu se poate transmite o proprietate ca parametru ref sau out.

Important! Deși se pot genera proprietăți *read-write* pentru toate câmpurile, dacă într-o clasă există multe câmpuri, trebuie să ne întrebăm dacă toate acestea trebuie să fie accesibile din exterior. Nu are sens ca toate câmpurile private ale unei clase să fie expuse în afară, chiar și prin intermediul proprietăților.