Curs 5

Clase - polimorfism, clase abstracte. Structuri, interfeţe, delegaţi

5.1 Polimorfismul

Polimorfismul este capacitatea unei entităţi de a lua mai multe forme. În limbajul C# polimorfismul este de 3 feluri: parametric, ad–hoc şi de moştenire.

5.1.1 Polimorfismul parametric

Este cea mai slabă formă de polimorfism, fiind regăsită în majoritatea limbajelor. Prin polimorfismul parametric se permite ca o implementare de funcţie să poată prelucra orice număr de parametri. Acest lucru se poate obţine prin folosirea în C# a unui parametru trimis prin *params* (a se vedea secţiunea 3.2).

5.1.2 Polimorfismul ad–hoc

Se mai numeşte şi supraîncărcarea metodelor, mecanism prin care în cadrul unei clase se pot scrie mai multe metode, având acelaşi nume, dar tipuri sau numere diferite de parametri formali. Alegerea funcţiei care va fi apelată se va face la compilare, pe baza corespondenţei între tipurile parametrilor de apel şi tipurile parametrilor formali.

5.1.3 Polimorfismul de moştenire

Este forma cea mai evoluată de polimorfism. Dacă precedentele forme de polimorfism sunt aplicabile fără a se pune problema de moştenire, în acest caz este necesar să existe o ierarhie de clase. Mecanismul se bazează pe faptul că o clasă de bază defineşte un tip care este compatibil din punct de vedere al atribuirii cu orice tip derivat, ca mai jos:

```
class B{...}
class D: B{...}
class Test
{
 static void Main()
 {
  B b = new D();//upcasting=conversie implicita catre baza B
 }
}
```

Într–un astfel de caz se pune problema: ce se întâmplă cu apeluri către metode având acelaşi nume şi aceiaşi parametri formali şi care se regăsesc în cele două clase?

Să considerăm exemplul următor: avem o clasă *Shape* care conţine o metodă *public void Draw()*; din *Shape* se derivează clasa *Polygon* care implementează aceeaşi metodă în mod specific. Problema care se pune este cum se rezolvă un apel al metodei *Draw* în context de upcasting:

```
class Shape
{
 public void Draw()
 {
  System.Console.WriteLine(``Shape.Draw()'');
 }
}
class Polygon: Shape
{
 public void Draw()
 {
  System.Console.WriteLine(``Polygon.Draw()'');
  //desenarea s-ar face prin GDI+
 }
}
class Test
```

```
{
 static void Main()
 {
  Polygon p = new Polygon();
  Shape s = p;//upcasting
  p.Draw();
  s.Draw();
 }
}
```

La compilarea acestui cod se va obţine un avertisment:

```
warning CS0108: The keyword new is required on Polygon.Draw()
because it hides inherited member Shape.Draw()
```

dar despre specificatorul *new* vom vorbi mai jos (oricum, adăugarea lui nu va schimba cu nimic comportamentul următor, doar va duce la dispariţia avertismentului). Codul va afişa:

```
Polygon.Draw()
Shape.Draw()
```

Dacă prima linie afişată este conformă cu intuiţia, cea de-a doua este discutabilă, dar de fapt este perfect justificată: apelul de metodă *Draw()* este rezolvat pentru ambele apeluri de mai sus la compilare pe baza tipului declarat al obiectelor; ca atare apelul precedent este legat de corpul metodei *Draw* din clasa *Shape*, chiar dacă *s* a fost instanţiat de fapt pe baza unui obiect de tip *Polygon*.

Este posibil ca să se dorească schimbarea acestui comportament: apelul de metodă *Draw* să fie rezolvat în funcţie de tipul efectiv al obiectului pentru care se face acest apel şi nu de tipul formal declarat. În cazul precedent, apelul *s.Draw()* trebuie să se rezolve de fapt ca fiind către metoda *Draw()* din *Polygon*, pentru că acesta este tipul obiectului *s* la momentul rulării. Cu alte cuvinte, apelul ar trebui să fie rezolvat la rulare şi nu la compilare, în funcţie de natura efectivă a obiectelor. Acest comportament polimorfic este referit sub denumirea *polimorfism de moştenire*.

5.1.4 *Virtual* şi *override*

Pentru a asigura faptul că legarea apelului la o metodă de metodă care va fi executată se face la rulare şi nu la compilare, e necesar ca în clasa de bază să se specifice că metoda *Draw()* este virtuală, iar în clasa derivată pentru aceeaşi metodă trebuie să se spună că este o suprascriere a celei din bază:

```
class Shape{
 public virtual void Draw(){...}
}
class Polygon : Shape{
 public override void Draw(){...}
}
```

În urma executării metodei *Main* din clasa de mai sus, se va afişa:

```
Polygon.Draw()
Polygon.Draw()
```

asta însemnând că s–a apelat metoda corespunzătoare "tipului efectiv" la rulare, în fiecare caz.

În cazul în care clasa *Polygon* este la rândul ei moştenită şi se doreşte ca polimorfismul să funcţioneze în continuare va trebui ca în această a treia clasă să suprascric (*override*) metoda *Draw()*.

Un astfel de comportament polimorfic este benefic atunci când se foloseşte o colecţie de obiecte de tipul unei clase de bază:

```
Shape[] painting = new Shape[10];
painting[0] = new Shape();
painting[1] = new Polygon();
...
foreach( Shape s in painting)
 s.Draw();
```

5.1.5 Modificatorul *new* pentru metode

Modificatorul *new* se foloseşte pentru a indica faptul că o metodă dintr-o clasă derivată care are aceeaşi semnătură cu una dintr–o clasă de bază nu este o suprascriere polimorfică a ei, ci apare ca o nouă metodă. Este ca şi cum metoda declarată *new* ar avea nume diferit şi nu se mai sesizează incercare de suprascriere.

Dacă nu se doreşte suprascrierea polimorfică şi nu se foloseşte *new*, se obţine avertismentul de compilare pomenit mai sus iar suprascrierea rămâne încă nepolimorfică.

Să presupunem următorul scenariu: compania A crează o clasă *A* care are forma:

```
public class A{
 public void M(){
  Console.WriteLine(``A.M()'');
```

```
 }
}
```

O altă companie B va crea o clasă *B* care moşteneşte clasa A. Compania B nu are nici o influenţă asupra companiei A sau asupra modului în care aceasta va face modificări asupra clasei *A*. Ea va defini în interiorul clasei B o metodă *M()* şi una *N()*:

```
class B: A{
 public void M(){
  Console.WriteLine(``B.M()'');
  N();
  base.M();
 }
 protected virtual void N(){
  Console.WriteLine(``B.N()'');
 }
}
```

Atunci când compania B compilează codul, compilatorul C# va produce următorul avertisment:

```
warning CS0108: The keyword new is required on 'B.M()' because
it hides inherited member 'A.M()'
```

Acest avertisment va notifica programatorul că clasa *B* defineşte o metodă *M()*, care va ascunde metoda *M()* din clasa de bază *A*. Această nouă metodă ar putea schimba înţelesul (semantica) lui *M()*, aşa cum a fost creată iniţial de compania A. Este de dorit în astfel de cazuri compilatorul să avertizeze despre posibile nepotriviri semantice şi suprascrieri – posibil accidentale. Oricum, programatorii din B vor trebui să pună specificatorul *new* înaintea metodei *B.M()* pentru a elimina avertismentul.

Să presupunem că o aplicaţie foloseşte clasa *B()* în felul următor:

```
class App{
 static void Main(){
  B b = new B();
  b.M();
 }
}
```

La rulare se va afişa:

```
B.M()
B.N()
A.M()
```

Să presupunem că A decide adăugarea unei metode virtuale *N()* în clasa sa, metodă ce va fi apelată din *M()*:

```
public class A
{
 public void M()
 {
  Console.WriteLine(``A.M()'');
  N();
 }
 protected virtual void N()
 {
  Console.WriteLine(``A.N()'');
 }
}
```

La o recompilare făcută de B, este dat următorul avertisment:

```
warning CS0114: 'B.N()' hides inherited member 'A.N()'. To make
the current member override that implementation, add the
override keyword. Otherwise, add the new keyword.
```

În acest mod compilatorul avertizează că ambele clase oferă o metodă *N()* a căror semantică poate să difere. Dacă B decide că metodele *N()* nu sunt semantic legate în cele două clase, atunci va specifica *new*, informând compilatorul de faptul că versiunea sa este una nouă, care nu suprascrie polimorfic metoda din clasa de bază. Să presupunem în continuare că se specifică *new*.

Atunci când codul din clasa *App* este rulat, se va afişa la ieşire:

```
B.M()
B.N()
A.M()
A.N()
```

Ultima linie afişată se explică tocmai prin faptul că metoda *N()* din *B* este declarată *new* şi nu *override* (dacă ar fi fost *override*, ultima linie ar fi fost *B.N()*, din cauza suprascrierii polimorfice).

Se poate ca B să decidă că metodele *M()* şi *N()* din cele două clase sunt legate semantic. În acest caz, ea poate şterge definiţia metodei *B.M*,

iar pentru a semnala faptul că metoda *B.N()* suprascrie polimorfic metoda omonimă din clasa părinte, va înlocui cuvântul *new* cu *override*. În acest caz, metoda *App.Main* va produce:

```
A.M()
B.N()
```

ultima linie fiind explicată de faptul că *B.N()* suprascrie polimorfic o metodă virtuală.

5.1.6 Metode *sealed*

O metodă declarată *override* poate fi declarată ca fiind de tip *sealed*, astfel împiedicându–se suprascrierea ei într–o clasă derivată din cea curentă:

```
using System;
class A
{
 public virtual void F()
 {
  Console.WriteLine(``A.F()'');
 }
 public virtual void G()
 {
  Console.WriteLine(``A.G()'');
 }
}
class B: A
{
 sealed override public void F()
 {
  Console.WriteLine(``B.F()'');
 }
 override public void G()
 {
  Console.WriteLine(``B.G()'');
 }
}
class C: B
{
 override public void G()
 {
```

```
  Console.WriteLine(‘‘C.G()’’);
 }
}
```

Modificatorul *sealed* pentru $B.F$ va împiedica tipul C să suprascrie metoda F.

5.1.7 Exemplu folosind *virtual, new, override, sealed*

Să presupunem următoare ierarhie de clase, reprezentată în Fig. 5.1; o clasă X moşteneste o clasă Y dacă sensul săgeţii este de la X la Y. Fiecare clasă are o metodă *void foo()* care determină afişarea clasei în care este definită şi pentru care se vor specifica *new, virtual, override, sealed.* Să

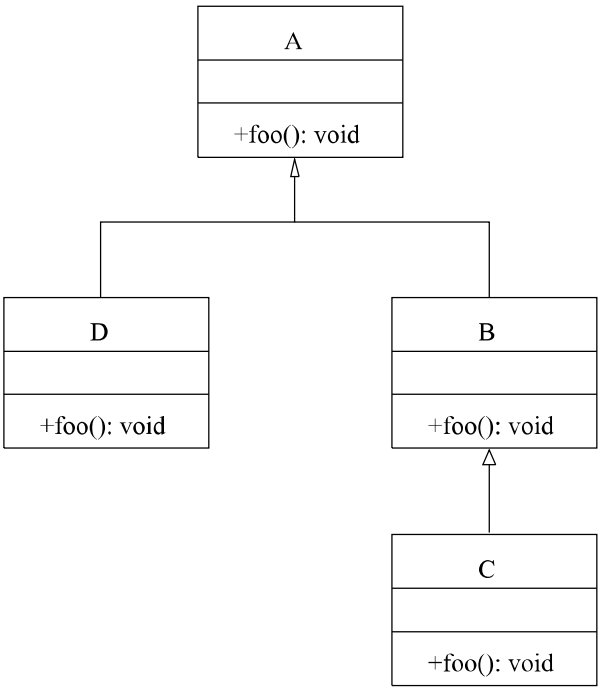

Figura 5.1: Ierarhie de clase

presupunem că clasa de test arată astfel:

```
public class Test
{
 static void Main()
 {
  A[] x = new A[4];
```

```
  x[0] = new A();
  x[1] = new B();
  x[2] = new C();
  x[3] = new D();

  A a = new A();
  B b = new B();
  C c = new C();
  D d = new D();

  /* secventa 1 */
  for(int i=0; i<4; i++)
  {
   x[i].foo();
  }
  /* secventa 2 */
  a.foo();
  b.foo();
  c.foo();
  d.foo();
 }
}
```

În funcţie de specificatorii metodelor *f()* din fiecare clasă, se obţin ieşirile din tabelul 5.1:

Tabelul 5.1: Efecte ale diferiţilor specificatori.

Metoda	A.f()	B.f()	C.f()	D.f()
Specificator	virtual	override	override	override
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	B.f B.f	C.f C.f	D.f D.f
Specificator	virtual	override	new	override
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	B.f B.f	B.f C.f	D.f D.f
Specificator	virtual	new	new	new
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	A.f B.f	A.f C.f	A.f D.f
Specificator	virtual	new	override	override
Eroare de compilare deoarece				

Tabelul 5.1 (continuare)

Metoda	A.f()	B.f()	C.f()	D.f()
C.f nu poate suprascrie metoda nevirtuală B.f()				
Specificator	virtual	virtual	override	override
Ieşire secv. 1 Ieşire secv. 2 Avertisment la compilare deoarece B.f înlocuieşte A.f	A.f A.f	A.f B.f	A.f C.f	D.f D.f
Specificator	virtual	sealed override	override	override
Eroare de compilare deoarece deoarece B.f nu poate fi suprascrisă de C.f				

5.2 Clase şi metode abstracte

Deseori pentru o anumită clasă nu are sens crearea de instanţe, din cauza unei generalităţi prea mari a tipului respectiv. Spunem că această clasă este *abstractă*, iar pentru a împiedica efectiv crearea de instanţe de acest tip, se va specifica cuvântul *abstract* înaintea clasei. În exemplele de anterioare, clasele *Employee* şi *Shape* ar putea fi gândite ca fiind abstracte: ele conţin prea puţină informaţie pentru a putea crea instanţe utile.

Analog, pentru o anumită metodă din interiorul unei clase uneori nu se poate specifica o implementare. De exemplu, pentru clasa *Shape* de mai sus, este imposibil să se dea o implementare la metoda *Draw()*, tocmai din cauza generalităţii clasei. Ar fi util dacă pentru această metodă programatorul ar fi obligat să dea implementări specifice ale acestei metode pentru diversele clase derivate. Pentru a se asigura tratarea polimorfică a acestui tip abstract, orice metodă abstractă este automat şi virtuală. O metodă declarată abstractă implică declararea clasei ca fiind abstractă.

Exemplu:

```
abstract class Shape
{
 public abstract void Draw();
 //remarcam lipsa implementarii si semnul punct si virgula
}
```

Orice clasă care este derivată dintr–o clasă abstractă va trebui fie să nu aibă nici o metodă abstractă moştenită fără implementare, fie să se declare ca fiind abstractă. O clasă abstractă nu poate fi instanţiată direct de către programator.

5.3 Tipuri parţiale

Începând cu versiunea 2.0 a platfomei .NET este posibil ca definiţia unei clase, interfeţe sau structuri să fie făcută în mai multe fişiere sursă. Definiţia clasei se obţine din reuniunea părţilor componente, lucru făcut automat de către compilator. Această spargere în fragmente este benefică în următoarele cazuri:

- atunci când se lucrează cu proiecte mari, este posibil ca la o clasă să trebuiască să lucreze mai mulţi programatori simultan - fiecare concentrându–se pe aspecte diferite.

- când se lucrează cu cod generat automat, acesta poate fi scris separat astfel încât programatorul să nu interfereze accidental cu el. Situaţia este frecvent întâlnită în cazul aplicaţiilor de tip Windows Forms.

De exemplu, pentru o formă nou creată (numită Form1) mediul Visual Studio va scrie un fişier numit Form1.Designer.cs care conţine partea de iniţializare a controalelor şi componentelor introduse de utilizator. Partea de tratare a evenimentelor, constructori etc este definită într-un alt fişier (Form1.cs).

Declararea unei părţi a unei clase se face folosind cuvântul cheie *partial* înaintea lui *class*.

Exemplu:

```
//fisierul Browser1.cs
public partial class Browser
{
  public void OpenPage(String uri)
  {...}
}
//fisierul Browser2.cs
public partial class Browser
{
  public void DiscardPage(String uri)
  {...}
}
```

Următoarele sunt valabile pentru tipuri parţiale:

- cuvântul *partial* trebuie să apară exact înainte cuvintelor: *class*, *interface*, *struct*
- dacă pentru o parte se specifică un anumit grad de acces, aceasta nu trebuie să ducă la conflicte cu declaraţiile din alte părţi
- dacă o parte de clasă este declarată ca abstractă, atunci întreaga clasă este considerată abstractă
- dacă o parte declară clasa ca fiind *sealed*, atunci întreaga clasă este considerată sealed
- dacă o parte declară că moşteneşte o clasă, atunci într-o altă parte nu se mai poate specifica o altă derivare
- părţi diferite pot să declare că se implementează interfeţe multiple
- aceleaşi câmpuri şi metode nu pot fi definite în mai multe părţi.
- clasele imbricate pot să fie declarate în părţi diferite, chiar dacă clasa conţinătoare e definită într-un singur fişier:

```
class Container
{
  partial class Nested
  {
    void Test() { }
  }
  partial class Nested
  {
    void Test2() { }
  }
}
```

- Următoarele elemente vor fi reunite pentru definiţia clasei: comentarii XML, interfeţe, atribute, membri.

  Exemplu:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

  este echivalent cu:

```
class Earth : Planet, IRotate, IRevolve { }
```

5.4 Structuri

Structurile reprezintă tipuri de date asemănătoare claselor, cu principala diferenţă că sunt tipuri valoare (o astfel de variabilă va conţine direct valoarea, şi nu o adresă de memorie). Sunt considerate versiuni "uşoare" ale claselor, sunt folosite predilect pentru tipuri pentru care aspectul comportamental este mai puţin pronunţat.

Declaraţia unei structuri se face astfel:
atribute$_{opt}$ modificatori-de-struct$_{opt}$ `struct` identificator :interfeţe$_{opt}$ corp ;$_{opt}$
Modificatorii de structură sunt: *new, public, protected, internal, private.* O structură este automat derivată din *System.ValueType*, care la rândul ei este derivată din *System.Object*; de asemenea, este automat considerată *sealed* (nederivabilă). Nu poate fi supertip pentru alte tipuri de date, poate să implementeze una sau mai multe interfeţe.

O structură poate să conţină declaraţii de constante, câmpuri, metode, proprietăţi, evenimente, indexatori, operatori, constructori, constructori statici, tipuri imbricate. Nu poate conţine destructor.

La atribuire, se face o copiere a valorilor conţinute de către sursă în destinaţie (indiferent de natura câmpurilor: valoare sau referinţă).

Exemplu:

```
using System;
public struct Point
{
  public Point(int xCoordinate, int yCoordinate)
  {
    xVal = xCoordinate;
    yVal = yCoordinate;
  }
  public int X
  {
    get
    {
      return xVal;
    }
    set
    {
      xVal = value;
    }
  }
  public int Y
```

```
  {
    get
    {
      return yVal;
    }
    set
    {
      yVal = value;
    }
}

  public override string ToString( )
  {
    return (String.Format(``{0}, {1}'', xVal.ToString(),yVal.ToString()));
  }
  public int xVal;
  public int yVal;
}

public class Tester
{
  public static void MyFunc(Point loc)
  {
    loc.X = 50;
    loc.Y = 100;
    Console.WriteLine(``In MyFunc loc: {0}'', loc);
  }
  static void Main( )
  {
    Point loc1 = new Point(200,300);
    Console.WriteLine(``Loc1 location: {0}'', loc1);
    MyFunc(loc1);
    Console.WriteLine(``Loc1 location: {0}'', loc1);
  }
}
```

După cum este dat în exemplul de mai sus, crearea unei instanţe se face folosind operatorul *new*; dar în acest caz, nu se va crea o instanţă în memoria heap, ci pe stivă. Transmiterea lui loc1 se face prin valoare, adică metoda *myFunc* nu face decât să modifice o copie de pe stivă a lui *loc1*. La revenire, se va afişa tot valoarea setată iniţial:

```
Loc1 location: 200, 300
In MyFunc loc: 50, 100
Loc1 location: 200, 300
```

Deseori pentru o structură se declară câmpurile ca fiind publice, pentru a nu mai fi necesare definirea accesorilor (simplificare implementării). Alţi programatori consideră însă că accesarea membrilor trebuie să se facă precum la clase, folosind proprietăţi. Oricare ar fi alegerea, limbajul o sprijină.

Alte aspecte demne de reţinut:

- Câmpurile nu pot fi iniţializate la declarare; altfel spus, dacă în exemplul de mai sus se scria:

  ```
  public int xVal = 10;
  public int yVal = 20;
  ```

  s-ar fi semnalat o eroare la compilare.

- Nu se poate defini un constructor implicit. Cu toate acestea, compilatorul va crea întotdeauna un astfel de constructor, care va iniţializa câmpurile la valorile lor implicite (0 pentru tipuri numerice sau pentru enumerări, *false* pentru *bool*, *null* pentru tipuri referinţă).

  Pentru tipul *Point* de mai sus, următoarea secvenţă de cod este corectă:

  ```
  Point a = new Point(0, 0);
  Point b = new Point();
  ```

  şi duce la crearea a două puncte cu abcisele şi ordonatele 0. Un constructor implicit este apelat atunci când se creează un tablou de structuri:

  ```
  Point[] points = new Points[10];
  for( int i=0; i<points.Length; i++ )
  {
    Console.WriteLine(points[i]);
  }
  ```

  va afişa de 10 ori puncte de coordonate (0, 0). De asemenea este apelat la iniţializarea câmpurilor de tip structură ai unei clase.

  De menţionat pentru exemplul anterior că se creează un obiect de tip tablou în heap, după care în interiorul lui (şi nu pe stivă!) se creează cele 10 puncte (alocare inline).

- Nu se poate declara destructor. Aceştia se declară numai pentru clase.
- Dacă programatorul defineşte un constructor, atunci acesta trebuie să dea valori iniţiale pentru câmpurile conţinute, altfel apare eroare la compilare.
- Dacă pentru instanţierea unei structuri declarate în interiorul unei metode sau bloc de instrucţiuni în general nu se apelează *new*, atunci respectiva instanţă nu va avea asociată nici o valoare (constructorul implicit nu este apelat automat!). Nu se poate folosi respectiva variabilă de tip structură decât după ce i se iniţializează toate câmpurile:

```
//bloc de instructiuni
{
 Point p;//variabila locala
 Console.WriteLine(p);//nota: aici se face boxing
}
```

va duce la apariţia erorii de compilare:

```
Use of unassigned local variable 'p'
```

Dar după nişte asignări de tipul:

```
p.xVal=p.yVal=0;
```

afişarea este posibilă (practic, orice apel de metodă pe instanţă este acum acceptat).

- Dacă se încearcă definirea unei structuri care conţine un câmp de tipul structurii, atunci va apărea o eroare de compilare:

```
struct MyStruct
{
  MyStruct s;
}
```

va genera un mesaj din partea compilatorului:

```
Struct member 'MyStruct.s' of type 'MyStruct' causes a
cycle in the structure layout
```

- Dacă o instanţă este folosită acolo unde un *object* este necesar, atunci se va face automat o conversie implicită către *System.Object* (boxing). Ca atare, utilizarea unei structuri poate duce (în funcţie de context) la un overhead datorat conversiei.

5.4.1 Structuri sau clase?

Structurile pot fi mult mai eficiente în alocarea memorici atunci când sunt reţinute într–un tablou. De exemplu, crearea unui tablou de 100 de elemente de tip *Point* va duce la crearea unui singur obiect (tabloul), iar cele 100 de instanţe de tip structură ar fi alocate inline în vectorul creat (şi nu referinţe ale acestora). Dacă *Point* ar fi declarat ca şi clasă, ar fi fost necesară crearea a 101 instanţe de obiecte în heap (un obiect pentru tablou, alte 100 pentru puncte), ceea ce ar duce la mai mult lucru pentru garbage collector şi ar putea duce la fragmentarea heap-ului.

Dar în cazul în care structurile sunt folosite în colecţii de tip Object (de exemplu un *ArrayList*), se va face automat un boxing, ceea ce duce la overhead (consum suplimentar de memorie şi cicli procesor). De asemenea, la transmiterea prin valoare a unei structuri, se va face copierea tuturor câmpurilor conţinute pe stivă, ceea ce poate duce la un overhead semnificativ.

5.5 Interfeţe

O interfaţă defineşte un contract comportamental. O clasă sau o structură care implementează o interfaţă aderă la acest contract. Relaţia dintre o interfaţă şi un tip care o implementează este deosebită de cea existentă între clase (este un/o): este o relaţie de implementare.

O interfaţă conţine metode, proprietăţi, evenimente, indexatori. Ea însă nu va conţine implementări pentru acestea, doar declaraţii. Declararea unei interfeţe se face astfel:
atribute$_{opt}$ modificatori-de-interfaţă$_{opt}$ `interface` identificator baza-interfeţei$_{opt}$ corp-interfaţă ;$_{opt}$
Modificatorii de interfaţă sunt: *new, public, protected, internal, private.* O interfaţă poate să moştenească de la zero sau mai multe interfeţe. Corpul interfeţei conţine declaraţii de metode, fără implementări. Orice membru are gradul de acces public. Nu se poate specifica pentru o metodă din interiorul unei interfeţe: *abstract, public, protected, internal, private, virtual, override*, ori *static.*

Exemplu:

```
interface IStorable
{
 void Read( );
 void Write();
}
```

O clasă care implementează o astfel de interfaţă se declară ca mai jos:

```
class Document: IStorable
{
 public void Read(){/*cod*/}
 public void Write(){/*cod*/}
 //alte declaratii
}
```

O clasă care implementează o interfaţă trebuie să definească toate metodele care se regăsesc în interfaţa respectivă, sau să declare metodele din interfaţă ca fiind abstracte. Următoarele reguli trebuie respectate la implementarea de interfeţe:

1. Tipul de retur al metodei din clasă trebuie să coincidă cu tipul de retur al metodei din interfaţă
2. Tipul parametrilor formali din metodă trebuie să fie acelaşi cu tipul parametrilor formali din interfaţă
3. Metoda din clasă trebuie să fie declarată publică şi nestatică.

Aceste implementări pot fi declarate folosind specificatorul *virtual* (deci subclasele clasei curente pot folosi *new* şi *override*).

Exemplu:

```
using System;
interface ISavable
{
 void Read();
 void Write();
}
public class TextFile : ISavable
{
 public virtual void Read()
 {
  Console.WriteLine("TextFile.Read()");
 }
 public void Write()
 {
  Console.WriteLine("TextFile.Write()");
 }
}
public class CSVFile : TextFile
{
```

```
 public override void Read()
 {
  Console.WriteLine("CSVFile.Read()");
 }
 public new void Write()
 {
  Console.WriteLine("CSVFile.Write()");
 }
}
public class Test
{
 static void Main()
 {
  Console.WriteLine("\nTextFile reference to CSVFile");
  TextFile textRef = new CSVFile();
  textRef.Read();
  textRef.Write();
  Console.WriteLine("\nISavable reference to CSVFile");
  ISavable savableRef = textRef as ISavable;
  if(savableRef != null)
  {
   savableRef.Read();
   savableRef.Write();
  }
  Console.WriteLine("\nCSVFile reference to CSVFile");
  CSVFile csvRef = textRef as CSVFile;
  if(csvRef!= null)
  {
   csvRef.Read();
   csvRef.Write();
  }
 }
}
```

La ieşire se va afişa:

```
TextFile reference to CSVFile
CSVFile.Read()
TextFile.Write()

ISavable reference to CSVFile
CSVFile.Read()
```

```
TextFile.Write()

CSVFile reference to CSVFile
CSVFile.Read()
CSVFile.Write()
```

În exemplul de mai sus se foloseşte operatorul *as* pentru a obţine o referinţă la interfeţe, pe baza obiectelor create. În general, se preferă ca apelul metodelor care sunt implementate din interfaţă să se facă via o referinţă la interfaţa respectivă, obţinută prin intermediul operatorului *as* (ca mai sus) sau după o testare prealabilă prin *is* urmată de conversie explicită, ca mai jos:

```
if (textRef is ISavable)
{
 ISavable is = textRef as ISavable;
 is.Read();//etc
}
```

În general, dacă se doreşte doar răspunsul la întrebarea "este obiectul curent un implementator al interfeţei I?", atunci se recomandă folosirea operatorului *is*. Dacă se ştie că va trebui făcută şi o conversie la tipul interfaţă, atunci este mai eficientă folosirea lui *as*. Afirmaţia se bazează pe studiul codului IL rezultat în fiecare caz.

Să presupunem că exista o interfaţă `I` având metoda `M()` care este implementată de o clasă `C`, deci defineşte metoda `M()`. Este posibil ca această metodă să nu aibă o semnificaţie în afara clasei `C`, ca atare a e de dorit ca metoda `M()` să nu fie declarată publică. Mecanismul care permite acest lucru se numeşte *implementare explicită*. Această tehnică permite ascunderea metodelor moştenite dintr-o interfaţă, acestea devenind private (calificarea lor ca fiind publice este semnalată ca o eroare). Implementarea explicită se obţine prin calificarea numelui de metodă cu numele interefţei:

```
interface IMyInterface
{
void F();
}
class MyClass : IMyInterface
{
void IMyInterface.F()//metoda privata!
{
//...
}
}
```

Metodele din interfeţe care s–au implementat explicit nu pot fi declarate *abstract, virtual, override, new.* Mai mult, asemenea metode nu pot fi accesate direct prin intermediul unui obiect (*obiect.NumeMetoda*), ci doar prin intermediul unei conversii către interfaţă respectivă, deoarece prin implementare explicită a metodelor aceste devin private în clasă/structură şi singura modalitate de acces a lor este upcasting-ul către interfaţă.

Exemplu:

```
using System;
public interface IDataBound
{
 void Bind();
}

public class EditBox : IDataBound
{
 // implementare explicita de IDataBound
 void IDataBound.Bind()
 {
  Console.WriteLine("Binding to data store...");
 }
}

class NameHidingApp
{
 public static void Main()
 {
  EditBox edit = new EditBox();
  Console.WriteLine("Apel EditBox.Bind()...");
  //EROARE: Aceasta linie nu se compileaza deoarece metoda
  //Bind nu mai exista ca metoda publica in clasa EditBox
  edit.Bind();
  Console.WriteLine();

  IDataBound bound = edit;
  Console.WriteLine("Apel (IDataBound) EditBox.Bind()...");
  // Functioneaza deoarece s-a facut conversie la IDataBound
  bound.Bind();
 }
}
```

Este posibil ca un tip să implementeze mai multe interfeţe. Atunci când

două interfeţe au o metodă cu aceeaşi semnătură, programatorul are mai multe variante de lucru. Cel mai simplu, el poate să furnizeze o singură implementare pentru ambele metode, ca mai jos:

```
interface IFriendly
{
 void GreetOwner() ;
}
interface IAffectionate
{
 void GreetOwner() ;
}
abstract class Pet
{
 public virtual void Eat()
 {
  Console.WriteLine( "Pet.Eat" ) ;
 }
}
class Dog : Pet, IAffectionate, IFriendly
{
 public override void Eat()
 {
  Console.WriteLine( "Dog.Eat" ) ;
 }
 public void GreetOwner()
 {
  Console.WriteLine( "Woof!" ) ;
 }
}
```

O altă modalitate este să se specifice explicit care metodă (mai exact: din ce interfaţă) este implementată.

```
class Dog : Pet, IAffectionate, IFriendly
{
 public override void Eat()
 {
  Console.WriteLine( "Dog.Eat" ) ;
 }
 void IAffectionate.GreetOwner()
 {
```

```
  Console.WriteLine( "Woof!" ) ;
 }
 void IFriendly.GreetOwner()
 {
  Console.WriteLine( "Jump up!" ) ;
 }
}
public class Pets
{
  static void Main()
  {
    IFriendly mansBestFriend = new Dog() ;
    mansBestFriend.GreetOwner() ;
    (mansBestFriend as IAffectionate).GreetOwner() ;
  }
}
```

La ieşire se va afişa:

```
Jump up!
Woof!
```

Dacă însă în clasa Dog se adaugă metoda

```
public void GreetOwner()
{
Console.WriteLine( "Woof 2!" ) ;
}
```

atunci se poate face apel `dog.GreetOwner()` (variabila dog este instanţă de `Dog`); apelurile de metode din interfaţă rămân de asemenea valide. Rezultatul este afişarea mesajului `Woof 2`.

5.5.1 Clase abstracte sau interfeţe?

Atât interfeţele cât şi clasele abstracte au comportamente similare şi pot fi folosite în situaţii similare. Dar totuşi ele nu se pot substitui reciproc. Câteva principii generale de utilizare a lor sunt date mai jos.

Dacă o relaţie se poate exprima mai degrabă ca "este un/o" decât altfel, atunci entitatea de bază ar trebui gândită ca o clasă abstractă.

Un alt aspect este bazat pe obiectele care ar folosi capabilităţile din tipul de bază. Dacă aceste capabilităţi ar fi folosite de către obiecte care nu sunt legate între ele, atunci ar fi indicată o interfaţă.

Un dezavantaj al claselor abstracte este că nu poate fi decât bază unică pentru orice altă clasă. Partea bună este că pot conţine cod definit care se moşteneşte.

5.6 Tipul delegat

În programare deseori apare următoarea situaţie: trebuie să se execute o anumită acţiune, dar nu se ştie de dinainte care anume, sau chiar ce obiect va trebui efectiv utilizat. De exemplu, un buton poate şti că trebuie să anunţe pe oricine este intresat despre faptul că fost apăsat, dar nu va şti aprioric cum va fi tratat acest eveniment. Mai degrabă decât să se lege butonul de un obiect particular, butonul va declara un delegat, pentru care clasa interesată de evenimentul de apăsare va da o implementare.

Fiecare acţiune pe care utilizatorul o execută pe o interfaţă grafică declanşează un eveniment. Alte evenimente se pot declanşa independent de acţiunile utilizatorului: sosirea unui email, terminarea copierii unor fişiere, sfârşitul unei interogări pe o bază de date etc. Un eveniment este o încapsulare a ideii că "se întâmplă ceva" la care programul trebuie să răspundă. Evenimentele şi delegaţii sunt strâns legate deoarece răspunsul la acest eveniment se va face de către un event handler, care este legat de eveniment printr-un delegat,

Un delegat este un tip referinţă folosit pentru a încapsula o metodă cu un anumit antet (tipul parametrilor formali şi tipul de retur). Orice metodă care are acest antet poate fi legată la un anumit delegat. Într–un limbaj precum C++, acest lucru se rezolvă prin intermediul pointerilor la funcţii. Delegaţii rezolvă aceeaşi problemă, dar într–o manieră orientată obiect şi cu garanţii asupra siguranţei codului rezultat, precum şi cu o uşoară generalizare (vezi delegaţii multicast).

Un delegat este creat după următoarea sintaxă:
atribute$_{opt}$ modificatori-de-delegat$_{opt}$ `delegate` tip-retur identificator(lista-param-formali$_{opt}$);
Modificatorul de delegat poate fi: *new, public, protected, internal, protected internal, private.* Un delegat se poate specifica atât în interiorul unei clase, cât şi în exteriorul ei, fiind de fapt o declaraţie de clasă derivată din *System.Delegate.* Dacă este declarat în interiorul unei clase, atunci este şi static (asemănător cu statutul claselor imbricate).

Exemplu:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

5.6.1 Utilizarea delegaţilor pentru a specifica metode la runtime

Să presupunem că se doreşte crearea unei clase container simplu numit *Pair* care va conţine două obiecte pentru care va putea face şi sortare. Nu se va şti aprioric care va fi tipul obiectelor conţinute, deci se va folosi pentru ele tipul *object*. Sortarea celor două obiecte se va face diferit, în funcţie de tipul lor efectiv: de exemplu pentru nişte persoane (clasa *Student* în cele ce urmează) se va face după nume, pe când pentru animale (clasa *Dog*) se va face după alt criteriu: greutatea. Containerul *Pair* va trebui să facă faţă acestor clase diferite. Rezolvarea se va da prin delegaţi.

Clasa *Pair* va defini un delegat, *WhichIsFirst*. Metoda *Sort* de ordonare va prelua ca (unic) parametru o instanţă a metodei *WhichIsFirst*, care va implementa relaţia de ordine, în funcţie de tipul obiectelor conţinute. Rezultatul unei comparaţii între două obiecte va fi de tipul enumerare *Comparison*, definit de utilizator:

```
public enum Comparison
{
  TheFirstComesFirst = 0,
  //primul obiect din colectie este primul in ordinea sortarii
  TheSecondComesFirst = 1
  //al doilea obiect din colectie este primul in ordinea sortarii
}
```

Delegatul (tipul de metodă care realizează compararea) se declară astfel:

```
//declarare de delegat
public delegate Comparison WhichIsFirst(
                          object obj1, object obj2);
```

Clasa Pair se declară după cum urmează:

```
public class Pair
{
  //tabloul care contine cele doua obiecte
  private object[] thePair = new object[2];

  //constructorul primeste cele doua obiecte continute
  public Pair( object firstObject, object secondObject)
  {
    thePair[0] = firstObject;
    thePair[1] = secondObject;
```

```
  }

  //metoda publica pentru ordonarea celor doua obiecte
  //dupa orice criteriu
  public void Sort( WhichIsFirst theDelegatedFunc )
  {
    if (theDelegatedFunc(thePair[0],thePair[1]) ==
        Comparison.TheSecondComesFirst)
    {
      object temp = thePair[0];
      thePair[0] = thePair[1];
      thePair[1] = temp;
    }
  }
  //metoda ce permite tiparirea perechii curente
  //se foloseste de polimorfism - vezi mai jos
  public override string ToString( )
  {
    return thePair[0].ToString()+", "+thePair[1].ToString();
  }
}
```

Clasele Student şi Dog sunt:

```
public class Dog
{
  public Dog(int weight)
  {
    this.weight=weight;
  }
  //Ordinea este data de greutate
  public static Comparison WhichDogComesFirst(
    Object o1, Object o2)
  {
    Dog d1 = o1 as Dog;
    Dog d2 = o2 as Dog;
    return d1.weight > d2.weight ?
     Comparison.TheSecondComesFirst :
     Comparison.TheFirstComesFirst;
  }
  //pentru afisarea greutatii unui caine
  public override string ToString( )
```

```
  {
    return weight.ToString( );
  }
  private int weight;
}

public class Student
{
  public Student(string name)
  {
    this.name = name;
  }
  //studentii sunt ordonati alfabetic
  public static Comparison WhichStudentComesFirst(
    Object o1, Object o2)
  {
    Student s1 = o1 as Student;
    Student s2 = o2 as Student;
    return (String.Compare(s1.name, s2.name) < 0 ?
      Comparison.TheFirstComesFirst :
      Comparison.TheSecondComesFirst);
  }
  //pentru afisarea numelui unui student
  public override string ToString( )
  {
    return name;
  }
  private string name;
}
```

Clasa de test este:

```
public class Test
{
  public static void Main( )
  {
    //creaza cate doua obiecte
    //de tip Student si Dog
    //si containerii  corespunzatori
    Student Stacey = new Student(``Stacey'');
    Student Jesse = new Student (``Jess'');
    Dog Milo = new Dog(10);
```

```
    Dog Fred = new Dog(5);
    Pair studentPair = new Pair(Stacey, Jesse);
    Pair dogPair = new Pair(Milo, Fred);
    Console.WriteLine(‘‘studentPair\t: {0}’’, studentPair);
    Console.WriteLine(‘‘dogPair\t: {0}’’, dogPair);
    //Instantiaza delegatii
    WhichIsFirst theStudentDelegate =
     new WhichIsFirst(Student.WhichStudentComesFirst);
    WhichIsFirst theDogDelegate =
     new WhichIsFirst(Dog.WhichDogComesFirst);
    //sortare folosind delegatii
    studentPair.Sort(theStudentDelegate);
    Console.WriteLine(‘‘Dupa sortarea pe studentPair\t: {0}’’,
     studentPair.ToString( ));
    dogPair.Sort(theDogDelegate);
    Console.WriteLine(‘‘Dupa sortarea pe dogPair\t\t: {0}’’,
     dogPair.ToString( ));
  }
}
```

5.6.2 Delegaţi statici

Unul din aspectele neelegante ale exemplului anterior este că e necesar ca în clasa *Test* să se instanţieze delegaţii care sunt necesari pentru a ordona obiectele din *Pair*. O modalitate mai bună este să se obţină delegaţii direct din clasele *Student* şi *Dog*. Acest lucru se obţine prin crearea unui delegat static în interiorul fiecărei clase:

```
public static readonly WhichIsFirst OrderStudents =
new WhichIsFirst(Student.WhichStudentComesFirst);
```

(analog pentru clasa *Dog*; de remarcat că “static readonly” nu se poate înlocui cu “const”, deoarece iniţializatorul nu este considerat expresie constantă). Declaraţia de mai sus se foloseşte astfel:

```
...
studentpair.Sort(Student.OrderStudent);
...
```

rezultatul fiind identic.

În [2] este dată şi o implementare de delegat ca proprietate statică, prin care se creează un delegat doar în cazul în care este nevoie de el[1].

[1] Tehnică numită iniţializare târzie (lazy initialization)