Zaawansowane zagadnienia programowania obiektowego

Jacek Matulewski
Materiały dla Podyplomowego Studium Programowania i Zastosowania Komputerów,
sekcja Projektowanie i tworzenie aplikacji dla platformy .NET (pod patronatem Microsoft)

Dziedziczenie

W tym miejscu zaczynają się zagadnienia, które można nazwać zaawansowanym programowaniem obiektowym. Z pewnością należą one do kanonu wiedzy o programowaniu w C#, ale nie będą używane w dalszej części książki, dlatego omawiam je bardziej hasłowo.

Definiując klasę możemy wskazać klasę, z której definiowany typ ma przejąć gotowe pola, metody, własności i zdarzenia. Nazywa się to **dziedziczeniem** (ang. *inheritance*). Klasę, która przejmuje te składowe nazywa się **potomną**, a klasę, z której są one przejmowane – **bazową**. A zatem klasa potomna dziedziczy po klasie bazowej. Jak to wygląda w praktyce? Mogliśmy się już o tym przekonać definiując klasę `Para<>`. Nie wskazaliśmy wprawdzie jawnie jej klasy bazowej, ale w takiej sytuacji jest nią klasa `System.Object`. Z niej klasa `Para<>` odziedziczyła m.in. metodę `ToString`, którą my nadpisaliśmy (listing 4.19).

W odróżnieniu od klas, definiując struktury nie możemy wskazać klasy bazowej. Swobodne dziedziczenie dotyczy tylko typów referencyjnych. Typy wartościowe zawsze dziedziczą z klasy `ValueType`, która dziedziczy bezpośrednio z klasy `Object`.

Klasy bazowe i klasy potomne

Aby lepiej zrozumieć mechanizm dziedziczenia przygotujmy prosty przykład klasy bazowej `Osoba` i dziedziczącej po niej klasy potomnej `OsobaZameldowana`. Obie klasy widoczne są na listingu 4.32. Towarzyszy im klasa `Adres`, której używam w klasie potomnej. Wszystkie te klasy można wstawić zarówno do klasy `Program` z projektu aplikacji konsolowej, jak i bezpośrednio do przestrzeni nazw. Na tym etapie nie ma to większego znaczenia.

Listing 4.32. Ilustracja relacji “jest” i “ma”

```
class Osoba
{
    public string Imię;
    public string Nazwisko;
    public int Wiek;

    public override string ToString()
    {
```

```
            return Imię + " " + Nazwisko + " (" + Wiek + ")";
        }
    }

    class Adres
    {
        public string Miasto;
        public string Ulica;
        public int NumerDomu;
        public int? NumerMieszkania;

        public override string ToString()
        {
            return Miasto + ", ul. " + Ulica + " " + NumerDomu +
                (NumerMieszkania.HasValue ? ("/" + NumerMieszkania) : "");
        }
    }

    class OsobaZameldowana : Osoba
    {
        public Adres AdresZameldowania;

        public override string ToString()
        {
            return base.ToString() + "; " + AdresZameldowania.ToString();
        }
    }
```

We wszystkich trzech klasach zdefiniowaliśmy metody `ToString`. **Nadpisują** one metodę `ToString` z klasy `Object`, co oznaczyliśmy używając modyfikatora `override`. Ponadto metoda `ToString` z klasy `OsobaZameldowana` nadpisuje metodę z klasy bazowej `Osoba`. Nie definiuje jednak całej swojej funkcjonalności od nowa, a używa implementacji z klasy bazowej do utworzenia tej części łańcucha, która związana jest z polami zdefiniowanymi w klasie `Osoba`. W tym celu użyte zostało słowo kluczowe `base`. W konstrukcji `base.ToString` wskazuje ono, że chodzi o wywołanie metody `Osoba.ToString`, a nie metody bieżącego obiektu (bez niego uzyskalibyśmy nieskończoną rekurencję wywołań).

Możemy sprawdzić działanie powyższych klas tworząc przykładową instancję klasy `OsobaZameldowana`. Pokazuje to listing 4.33.

Listing 4.33. Tworzenie instancji klasy potomnej

```
    static void Main(string[] args)
    {
        OsobaZameldowana jk = new OsobaZameldowana()
        {
            Imię = "Jan",
            Nazwisko = "Kowalski",
            Wiek = 42,
            AdresZameldowania = new Adres
            {
                Miasto = "Toruń",
```

```
            Ulica = "Grudziadzka",
            NumerDomu = 5,
            NumerMieszkania = null
        }
    };

    Console.WriteLine(jk.ToString());
}
```

W listingu 4.32 zdefiniowane są trzy klasy. Zachodzą między nimi różne relacje. W założeniu mają one odpowiadać relacjom między pojęciami z „prawdziwego" świata, który chcemy w klasach odwzorować. Po pierwsze relacja dziedziczenia między klasą `OsobaZameldowana` i jej klasą bazową `Osoba` odpowiada relacji „jest". Osoba zameldowana **jest** osobą. Jeżeli taką relację możemy wskazać między dwoma pojęciami użytymi do opisu problemu, to znaczy, że definiując klasy powinniśmy użyć dziedziczenia. Pies jest zwierzęciem, a więc klasa `Pies` powinna dziedziczyć z klasy `Zwierzę`. Analogicznie samochód osobowy jest pojazdem, ułamek jest liczbą, trójkąt, prostokąt i okrąg są figurami itd. Dziedziczenie może być wielokrotne: np. pies jest ssakiem, który jest zwierzęciem, który jest organizmem żywym itd.

Inna relacja zachodzi między klasą `OsobaZameldowana` a klasą `Adres`. Osoba zameldowana **ma** adres. W takiej sytuacji adres powinien być jej elementem składowym klasy opisującej osobę zameldowaną. Adres jest tym, co odróżnia osobę zameldowaną od osoby jako takiej. Rozszerzamy zatem klasę `Osoba` dodając adres. Tym samym zawężamy zakres obiektów, które klasa potomna `OsobaZameldowana` opisuje. Mówimy, że klasa `OsobaZameldowana` jest bardziej **wyspecjalizowana** od klasy `Osoba`.

Projektując architekturę systemu i zastanawiając się jakie klasy musimy zdefiniować i jakie powinny być między nimi relacje, zawsze powinniśmy zadawać sobie dwa pytania. Pierwszym jest pytanie o to jakich pojęć (rzeczowników) używam do opisu problemu. Powinny im odpowiadać typy, jakie należy zdefiniować. Natomiast drugie pytanie to: jakie relacje wiążą te pojęcia. Czy do opisu relacji między tymi pojęciami używam słów „jest" czy „ma". Determinuje to wybór między dziedziczeniem, a definiowaniem jako elementu składowego. Oczywiście nie zawsze relacja „ma" musi oznaczać posiadanie lub zawierane przestrzenne. Rodzic może mieć dzieci – wówczas polem składowym klasy `Rodzic` jest kolekcja `Dzieci`, samochód może mieć silnik i skrzynię biegów, a figura pewną ilość wierzchołków. Relacja „ma" może również opisywać posiadanie jakiejś własności np. samochód może mieć jakąś prędkość, ma kolor, ma bagażnik, który ma jakąś wielkość itd.

Rzadko używanym w praktyce modyfikatorem jest `sealed`. Tak oznaczone powinny być klasy, które nie zostały zaprojektowane do dziedziczenia i w efekcie chcemy zablokować taką możliwość. Tak można również oznaczyć metody. Nie mogą one być nadpisywane, a tym samym ich funkcjonalność w klasach potomnych nie może być zmieniona.

Wszystkie pola składowe w klasie bazowej `Osoba` zadeklarowaliśmy jako publiczne (modyfikator `public`). Jeżeli byłyby prywatne (modyfikator `private`), to nie byłyby widoczne w klasie potomnej. Z kolei pola chronione (modyfikator `protected`) byłyby w klasie potomnej widoczne, ale już nie poza nią. Pola zdefiniowane w klasie bazowej jako chronione, w klasie potomnej stają się niejako prywatne. To samo dotyczy własności, metod i zdarzeń. Aby się o tym przekonać zdefiniujmy w klasie bazowej prywatną własność tylko do odczytu `Personalia` łączącą imię i nazwisko w jeden łańcuch (listing 4.34).

Listing 4.34. Elementy prywatne są dziedziczone przez klasy potomne, ale nie są z nich dostępne

```
class Osoba
{
    public string Imię;
    public string Nazwisko;
    public int Wiek;

    public override string ToString()
    {
        return Imię + " " + Nazwisko + " (" + Wiek + ")";
```

```
        }

        private string Personalia
        {
            get
            {
                return Imię + " " + Nazwisko;
            }
        }
    }
```

Korzystając z *IntelliSense* możemy się łatwo przekonać, że nowa własność nie będzie dostępna ani z metody `Main`, w której możemy próbować odwoływać się do niej na rzecz obiektu `jk`, ani z klasy potomnej (możemy próbować użyć jej chociażby w nadpisanej metodzie `ToString`). To się zmieni jeżeli modyfikator `private` zmienimy na `protected`. Własność `Personalia` nadal nie będzie dostępna z metody `Main`, ale będzie widoczna w metodach i własnościach klasy potomnej.

Klasy abstrakcyjne

Podany wyżej przepis na projektowanie systemu klas jako odwzorowania pojęć używanych do opisania problemu sprawdza się całkiem nieźle w praktyce. Zauważmy jednak, że nie wszystkim pojęciom odpowiadają rzeczywiste obiekty. O ile z łatwością możemy sobie wyobrazić przykłady prostokątów, okręgów czy trójkątów, to wyobrażenie sobie figury samej w sobie, która nie byłaby jakąś konkretną figurą z określoną liczbą wierzchołków jest niemożliwe. Co wcale nie oznacza, że nie możemy sprawnie operować abstrakcyjnym pojęciem figury. Podobnie łatwo wyobrazić sobie samochód osobowy, ale trudno pojazd, który nie byłby ani samochodem osobowym, ani ciężarowym, ani rowerem czy furmanką. Projektując zbiór klasy możemy zablokować możliwość tworzenia instancji takich klas opisujących najbardziej ogólne pojęcia. Używa się do tego słowa kluczowego `abstract`, a klasy takie nazywa się abstrakcyjnymi. Listing 4.35 pokazuje przykład abstrakcyjnej klasy `Figura` i trzech dziedziczących z niej klas `Okrąg`, `Trójkąt` i `Koło`.

Listing 4.35. Przykład z figurami to klasyczny przykład jeszcze z pierwszych książek o C++

```
abstract class Figura {};
class Okrąg : Figura {}
class Trójkąt : Figura {}
class Prostokąt : Figura {}
class Kwadrat : Prostokąt {}
```

Próba utworzenia instancji abstrakcyjnego obiektu zakończy się błędem kompilatora:

```
Figura f = new Figura(); //błąd kompilatora
```

Można natomiast używać klasy abstrakcyjnej do utworzenia zmiennej typu referencyjnego:

```
Figura f;
```

a w takiej zmiennej można zapisać referencję do instancji jej klasy potomnej:

```
Figura f = new Kwadrat();
```

Po co? To wyjaśnię za chwilę.

Klasa abstrakcyjna może, choć nie musi, zawierać metody abstrakcyjne, a więc takie, które są zadeklarowane, ale nie posiadają ciała (listing 4.36). Metody takie muszą być natomiast zdefiniowane w klasach potomnych, które nie są klasami abstrakcyjnymi. W przeciwnym razie pojawi się błąd kompilatora.

Listing 4.36. Metoda abstrakcyjna i jej implementacja w klasach potomnych

```
abstract class Figura
{
    public abstract int IleWierzchołków();
```

```
}

class Okrąg : Figura
{
    public override int IleWierzchołków()
    {
        return 0;
    }
}

class Trójkąt : Figura
{
    public override int IleWierzchołków()
    {
        return 3;
    }
}

class Prostokąt : Figura
{
    public override int IleWierzchołków()
    {
        return 4;
    }
}

class Kwadrat : Prostokąt {}
```

Abstrakcyjne mogą także być własności (ale nie pola). Oto przykład takiej własności z klasy bazowej:

```
abstract class Figura
{
    public abstract int IleWierzchołków();
    public abstract int LiczbaWierzchołków { get; }
}
```

Jej obecność w klasie bazowej `Figura` wymusza konieczność zdefiniowania tej własności we wszystkich klasach bazowych. Oto przykład jej nadpisania z klasy `Okrąg`:

```
class Okrąg : Figura
{
    public override int LiczbaWierzchołków
    {
        get
        {
            return IleWierzchołków();
        }
    }
    ...
```

Jak już wiemy, do oznaczenia, że własność lub metoda nadpisuje element składowy z klasy bazowej używamy modyfikatora `override`. Nadpisywane własności `LiczbaWierzchołków` nie mogą mieć sekcji `set`, ponieważ sekcja ta nie została uwzględniona we własności abstrakcyjnej z klasy `Figura`.

Metody wirtualne

Takie definiowanie własności `LiczbaWierzchołków` nie ma jednak większego sensu. Naszą uwagę powinno zwrócić to, że jej definicja wyglądałaby tak samo we wszystkich klasach potomnych. W takiej sytuacji bardziej naturalne byłoby zdefiniowanie jej w klasie bazowej. Nie szkodzi, że to jest klasa abstrakcyjna – nie wszystkie metody takiej klasy muszą być abstrakcyjne. Aby nie blokować możliwości nadpisywania oznaczymy ją jednak jako wirtualną:

```
abstract class Figura
{
    public abstract int IleWierzchołków();

    public virtual int LiczbaWierzchołków
    {
        get
        {
            return IleWierzchołków();
        }
    }
}
```

Aby można było zdefiniować metody abstrakcyjne konieczne jest oznaczenie całej klasy jako abstrakcyjnej. Metody takie nie mają bowiem ciała. Muszą być wobec tego zdefiniowane w klasach potomnych, które już abstrakcyjne nie są. Są więc de facto komunikatem, że klasy potomne mają mieć pewną funkcjonalność, która nie może być zdefiniowana w klasie bazowej albo z powodu braku odpowiednich informacji (np. liczby wierzchołków), albo dlatego, że jest ona zbyt abstrakcyjna, aby można było ją zaimplementować w kodzie.

Czasem możliwa jest jednak sytuacja, w której można pokusić się o zdefiniowanie metody w klasie bazowej dopuszczając jej przedefiniowanie w klasie potomnej. Z taką sytuacją mamy do czynienia w przypadku własności `LiczbaWierzchołków`. Takie metody nazywamy wirtualnymi i oznaczamy modyfikatorem `virtual`. Dobrym przykładem takiej metody jest `ToString` z klasy `Object`. Wersja tej metody zdefiniowana w klasie `Object` zwraca po prostu nazwę typu. Wiele klas nadpisuje ją jednak określając bardziej pożyteczny sposób konwersji na łańcuch. My zrobiliśmy to w strukturze `Ulamek` oraz klasach `Para<>` czy `Osoba`.

Różnica między metodą wirtualną i abstrakcyjną jest więc taka, że metodę abstrakcyjną trzeba nadpisać, a metodę wirtualną jedynie można. Co więcej nadpisując metodę wirtualną można odwołać się do jej implementacji w klasie bazowej (*vide casus* metody `OsobaZameldowana.ToString` z listingu 4.32). W przypadku metody abstrakcyjnej nie ma się do czego odwoływać, bo w klasie bazowej nie ma ona przecież ciała.

Polimorfizm

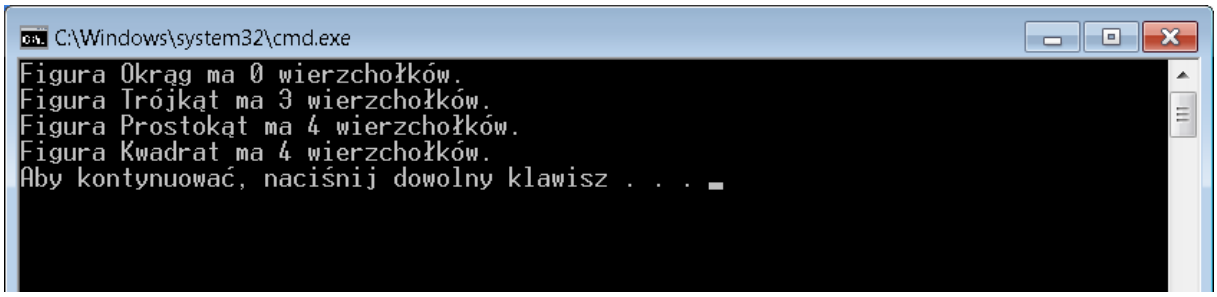
Załóżmy, że mamy program, który przechowuje zbiór figur. Chcemy mieć możliwość narysowania ich wszystkich w podobny sposób[1]. Problem w tym, że inaczej rysowany jest okrąg, trójkąt i prostokąt. Jak ten problem rozwiązać? Kanonicznym rozwiązaniem jest zdefiniowanie klasy bazowej `Figura` z abstrakcyjną metodą `Rysuj`. Następnie definiujemy zbiór klas potomnych `Okrąg`, `Trójką, Prostokąt` i `Kwadrat` odpowiadających poszczególnym figurom, w których metodę `Rysuj` nadpisujemy. Wówczas możemy utworzyć listę typu `List<Figura>`, która może przechowywać instancje wszystkich klas potomnych klasy `Figura`, a wiec obiekty typu `Okrąg`, `Trójkąt`, `Prostokąt` i `Kwadrat`. I wreszcie tworzymy pętlę, która wywołuje metodę `Rysuj` po kolei na rzecz wszystkich elementów tej kolekcji. Czy wywołanie metody `Rysuj` na rzecz referencji typu `Figura` nie skończy się jednak błędem? Przecież w klasie `Figura` ta metoda nie ma nawet ciała. Otóż

[1] Por. *http://msdn.microsoft.com/en-us/library/ms173152.aspx*.

właśnie nie i to jest kluczowa własność mechanizmu metod wirtualnych i abstrakcyjnych. Bez względu na rzecz jakiego typu referencji wywoływane będą metody obiektu, użyte będą definicje właściwe dla jego prawdziwego typu. Mechanizm ten nazywa się **polimorfizmem** i w skrócie oznacza, że obiekty różnych, choć spokrewnionych typów, możemy traktować w jednorodny sposób. Pokazuje to listing 4.37, który wprawdzie nie używa metody `Rysuj`, bo jej implementacja byłaby trudna w oknie konsoli, ale ilustruje działanie mechanizmu metod wirtualnych na przykładzie metody `IleWierzchołków`. Efekt jaki zobaczymy widoczny jest na rysunku 4.3.

Listing 4.37. Polimorfizm w najprostszym przykładzie

```
static void Main(string[] args)
{
    List<Figura> figury = new List<Figura>();
    figury.Add(new Okrąg());
    figury.Add(new Trójkąt());
    figury.Add(new Prostokąt());
    figury.Add(new Kwadrat());
    foreach (Figura figura in figury)
    Console.WriteLine("Figura " + figura.GetType().Name + " ma " +
        figura.IleWierzchołków() + " wierzchołków.");
}
```



Rysunek 4.3. Przykład użycia metod wirtualnych

Przykład ten obrazuje podstawową ideę polimorfizmu uzyskiwanego dzięki dziedziczeniu. A mianowicie obiekty wielu różnych typów (`Okrąg`, `Trójkąt`, `Prostokąt` i `Kwadrat`) traktujemy w podobny sposób. Umożliwia to ich wspólny mianownik, czyli klasa bazowa określająca zakres funkcjonalności, która jest na pewno dostępna w każdej z nich. Funkcjonalności te mogą się jednak różnić – choć sygnatura metod `IleWierzchołków` musi być taka sama we wszystkich klasach potomnych, to jej definicja może być inna w każdej z tych klas. Dzięki temu możemy obiekty poszczególnych figur gromadzić we wspólnej kolekcji zdefiniowanej dla parametru `Figura`. Scenariuszy użycia polimorfizmu jest zresztą wiele. Możemy na przykład przygotować metodę, która przyjmuje jako argument klasę bazową `Figura`. Wówczas funkcjonalność obiektów przesyłanych do tej metody ograniczona jest do tej określonej w klasie `Figura`[2], ale wywoływane metody są odpowiednie dla typu przesyłanego argumentu (listing 4.38).

Listing 4.38. Użycie polimorfizmu w metodzie obsługującej wiele typów

```
static void WyświetlIlośćWierzchołków(Figura figura)
{
    Console.WriteLine("Liczba wierzchołków figury: " + figura.IleWierzchołków());
}
```

[2] Cała prawda jest taka, że obiekt zawsze zna swój typ dzięki mechanizmowi *Reflection*. Zatem zawsze możemy upewnić się, że przesłana do metody figura jest np. prostokątem (pozwala na to operator `is`) i jeżeli tak, zrzutować na typ `Prostokąt`. W ten sposób można odzyskać pełną funkcjonalność typu `Prostokąt`. Co więcej jeżeli nie znamy typu obiektu, jaki przesyłany, mechanizm *Reflection* pozwala na jego analizę i wywołanie dowolnej metody.

Konstruktory a dziedziczenie

Warto w kontekście dziedziczenia zwrócić także uwagę na konstruktory. Na początek zdefiniujmy domyślny konstruktor w klasie bazowej `Figura` oraz konstruktor w jednej z jego klas potomnych np. `Trójkąt` (listing 4.39).

Listing 4.39. Konstruktory domyślne dodane do klasy bazowej i potomnej

```
abstract class Figura
{
    ...

    public Figura()
    {
        Console.WriteLine("Konstruktor klasy Figura");
    }
}


...


class Trójkąt : Figura
{
    ...

    public Trójkąt()
    {
        Console.WriteLine("Konstruktor klasy Trójkąt");
    }
}
```

Co się stanie, jeżeli utworzymy obiekt reprezentujący trójkąt poleceniem `new Trójkąt();`? Wydruk z konsoli przekona nas, że najpierw wywołany zostanie konstruktor klasy bazowej `Figura`, a dopiero po nim konstruktor klasy potomnej `Trójkąt`. Nie ma tu wobec tego mowy o żadnej wirtualności czy nadpisywaniu konstruktorów – gdy powstaje obiekt, który jest figurą, wywoływany jest konstruktor klasy `Figura`. Dzięki temu, w konstruktorze klasy potomnej nie trzeba, a wręcz nie należy powtarzać żadnych instrukcji inicjujących elementy klasy bazowej (chyba, że chcemy zmienić ich wartość). Za ich inicjację powinien być odpowiedzialny wyłącznie konstruktor klasy bazowej.

Teraz zdefiniujmy w klasie bazowej i potomnej konstruktory z jakimś argumentem, np. typu `string` (listing 4.40).

Listing 4.40. Konstruktory z argumentem typu string

```
abstract class Figura
{
    ...

    public Figura()
    {
        Console.WriteLine("Konstruktor klasy Figura");
     }

     public Figura(string argument)
     {
```

```
            Console.WriteLine("Konstruktor klasy Figura, komunikat: " + argument);
        }
    }

    ...

    class Trójkąt : Figura
    {
        ...

        public Trójkąt()
        {
            Console.WriteLine("Konstruktor klasy Trójkąt");
        }

        public Trójkąt(string argument)
        {
            Console.WriteLine("Konstruktor klasy Trójkąt, komunikat: " + argument);
        }
    }
```

Jakie konstruktory zostaną wywołane, gdy utworzymy obiekt trójkąta poleceniem `new Trójkąt("jestem trójkątem");`? Łatwo domyśleć się, że zostanie wywołany nowy konstruktor klasy potomnej `Trójkąt`, ale zaskoczeniem może być, że wcześniej zostanie wywołany domyślny, czyli bezargumentowy, konstruktor klasy bazowej `Figura`. Jeżeli chcemy, aby było inaczej tj. aby wywołany był konstruktor z argumentem `string`, należy to wyraźnie zaznaczyć w konstruktorze klasy potomnej w następujący sposób:

Listing 4.41. Wskazywanie konstruktora klasy bazowej

```
    public Trójkąt(string argument)
        : base(argument)
    {
        Console.WriteLine("Konstruktor klasy Trójkąt, komunikat: " + argument);
    }
```

Pamiętamy, że jeżeli klasa nie posiada żadnego konstruktora, kompilator tworzy automatycznie konstruktor domyślny inicjujący wszystkie pola wartościami domyślnymi dla ich typów. Jeżeli w takiej klasie sami zdefiniujemy konstruktor, ale nie będzie to konstruktor domyślny, to kompilator nie utworzy już konstruktora domyślnego. Zatem jeżeli w klasie bazowej zdefiniujemy konstruktor z jakimś argumentem nie definiując jednocześnie konstruktora domyślnego, to wszystkie klasy potomne będą musiały posiadać konstruktory, w których wskażą jawnie ów konstruktor. Inaczej pojawi się błąd kompilacji. Możemy się o tym przekonać usuwając konstruktor domyślny z klasy `Figura`.

Interfejsy

W C# nie ma wielodziedziczenia - klasy mogą dziedziczyć tylko po jednej klasie bazowej. Definiując struktury w ogóle nie możemy wskazać klasy bazowej. W zamian C# oferuje możliwość implementacji interfejsów. Poznaliśmy je już definiując strukturę `Ulamek` i klasę `Para<>` – w obu przypadkach definiowany przez nas typ implementował interfejs `IComparable`, który wymuszał na nas zdefiniowanie metody `CompareTo`. Teraz przyszedł czas, aby wiedzę o interfejsach uporządkować i nieco poszerzyć.

Wróćmy do klasy `OsobaZameldowana`. Załóżmy, że bierzemy pod uwagę, że osoba zameldowana posiada telefon stacjonarny. Moglibyśmy wobec tego zdefiniować klasę potomną `OsobaZameldowanaPosiadającaTelefon`, która dziedziczyłaby z klasy `OsobaZameldowana`. Ale przecież

nie tylko osoby mają telefony. Możemy zadzwonić także do urzędu lub restauracji. Czy należy stworzyć klasę bazową dla osób, urzędów, restauracji i innych miejsc, do których można zadzwonić? Nie, bo to niemożliwe. Klasa `OsobaZameldowana` już dziedziczy z klasy `Osoba`, w której nie można zakładać telefonu domowego, skoro nawet nie zakładamy zameldowania. Klasa `OsobaZameldowana` musiałaby wobec tego dziedziczyć z dwóch klas, co w C# nie jest możliwe. A może ten problem rozwiązać inaczej. W końcu telefon raczej się ma niż jest się osobą „utelefonicznioną". Może po prostu telefon powinien być elementem składowym. Być może tak, ale wówczas nie mielibyśmy żadnego wspólnego mianownika, który pozwoliłby nam np. tworzyć kolekcje obiektów z dostępnym numerem telefonu. Rozwiązaniem tego problemu jest możliwość zdefiniowania interfejsu. Jeżeli zdefiniujemy interfejs `IPosiadajacyTelefonStacjonarny` (listing 4.42), to implementować go będzie mogła nie tylko klasa `OsobaZameldowana`, ale również klasy `Urząd` czy `Restauracja`. Co więcej, można w analogiczny sposób zdefiniować interfejs `IPosiadajacyAdresEmail` i również go implementować. Nie ma ograniczenia na ilość implementowanych interfejsów.

Listing 4.42. Definicja interfejsu i jego implementacja

```
interface IPosiadajacyTelefonStacjonarny
{
    int? NumerTelefonu { get; set; }
}

class OsobaZameldowana : Osoba, IPosiadajacyTelefonStacjonarny
{
    public Adres AdresZameldowania;

    public override string ToString()
    {
         return base.ToString() + "; " + AdresZameldowania.ToString();
     }

     private int? numerTelefonu;

     public int? NumerTelefonu
     {
         get
         {
             return numerTelefonu;
         }
         set
         {
             numerTelefonu = value;
        }
    }

     public bool CzyPosiadaTelefonStacjonarny
    {
        get
        {
            return numerTelefonu.HasValue;
        }
    }
```

```
}
```

Czy zatem jest interfejs i czym różni się od klasy abstrakcyjnej? O abstrakcyjnej klasie bazowej należy myśleć jak o fundamencie, na którym budowana jest hierarchia jej klas potomnych. Natomiast interfejs jest raczej czymś w rodzaju deklaracji, że implementujące ją klasy będą posiadały pewną funkcjonalność (metody lub własności). Interfejs nie realizuje zatem relacji „jest", a raczej „obiecuję, że potrafi". Natomiast sposób użycia może być bardzo podobny, jak w przypadku klas abstrakcyjnych i często wybór między jednym i drugim mechanizmem może być dość trudny. W praktyce warto zastanowić się, czy w opisie problemu, jakaś grupa pojęć może być określona bardziej abstrakcyjnym pojęciem. Jeżeli tak, to temu pojęciu powinna odpowiadać klasa bazowa, może abstrakcyjna. Jeżeli natomiast jakąś grupę pojęć wiąże nie pojęcie (rzeczownik), a raczej wspólna czynność lub umiejętność (czasowniki), to powinna to być dla nas wskazówka, że warto rozważyć zdefiniowanie interfejsu, który tę czynność opisuje.

Jak wspomniałem sposób użycia interfejsów jest podobny do korzystania z klas bazowych. Możemy stworzyć kolekcję obiektów, które mogłyby posiadać telefon stacjonarny i np. odczytać ich numery. Interfejsy mogą też być używane jako argumenty metod. Pokazuje to listing 4.43.

Listing 4.43. Użycie interfejsu jako argumentu metody i parametru kolekcji

```
static void WyświetlNumerTelefonu(IPosiadajacyTelefonStacjonarny telefon)
{
    Console.WriteLine("Numer telefonu: " + telefon.NumerTelefonu);
}

static void Main(string[] args)
{
    List<IPosiadajacyTelefonStacjonarny> listaTelefonów =
        new List<IPosiadajacyTelefonStacjonarny>();
    listaTelefonów.Add(new OsobaZameldowana() { NumerTelefonu = 123456789 });
    listaTelefonów.Add(new OsobaZameldowana() { NumerTelefonu = 987654321 });
    foreach (IPosiadajacyTelefonStacjonarny telefon in listaTelefonów)
        WyświetlNumerTelefonu(telefon);
}

static void WyświetlIlośćWierzchołków(Figura figura)
{
    Console.WriteLine("Liczba wierzchołków figury: " + figura.IleWierzchołków());
}
```