Programowanie obiektowe w Visual Basic .NET dla każdego

Autorzy: Richard Simon, Leslie Koorhan, Ken Cox
Tłumaczenie: Jadwiga Gnybek
ISBN: 83-7197-783-2
Tytuł oryginału: TY Object Oriented Programming with Visual Basic.NET in 21 Days
Format: B5, stron: 374

Visual Basic to język programowania pozwalający nawet początkującym programistom pisać zaawansowane aplikacje wykorzystujące wszystkie możliwości systemów operacyjnych z rodziny Windows. Jego kolejne wersje w coraz większym stopniu korzystały z technik programowania zorientowanego obiektowo (OOP). Najnowsza z nich, Visual Basic .NET jest uwieńczeniem tej ewolucji.

Jeśli chcesz tworzyć aplikacje w VB .NET, musisz nauczyć się programować obiektowo. Z pewnością pomoże Ci w tym książka „Programowanie obiektowe w Visual Basic .NET". Informacje w niej zawarte przydadzą się także, gdy zaczniesz używać innych języków programowania opartych na obiektach, takich jak C++, C# czy Java.

Ten podręcznik, łączący praktyczne ćwiczenia z niezbędną dawką przydatnej teorii, nauczy Cię samodzielnie tworzyć aplikacje używające obiektów, dziedziczenia, poliformizmu i sterowania za pomocą zdarzeń. Dowiesz się również jak obsługiwać błędy. Odrębne rozdziały poświęcono obiektom zdalnym (aplikacjom klient-serwer), udostępnianiu aplikacji poprzez Internet tworzeniu atrakcyjnego interfejsu użytkownika oraz językowi XML. Poznasz także środowisko projektowe Visual Studio .NET. Swoją wiedzę będziesz mógł sprawdzić za pomocą licznych quizów.

Książka zawiera:

- Wprowadzenie do programowania obiektowego w Visual Basic .NET
- Wyjaśnienie terminologii związanej z programowaniem obiektowym
- Omówienie środowiska Visual Studio .NET oraz tworzenia interfejsów użytkownika za pomocą Windows Forms i WebForms
- Praktyczne ćwiczenia i quizy sprawdzające wiedzę

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 14.

Praca z interfejsami obiektów

W poprzednich wersjach Visual Basica jedynie interfejsy dawały możliwość zastosowania polimorfizmu. Visual Basic .NET może realizować polimorfizm poprzez interfejsy lub mechanizm dziedziczenia. Użycie interfejsów czy dziedziczenia jest kwestią wyboru pomiędzy realizacją podstawowych i specyficznych funkcji. Zwykle dziedziczenie używane jest w przypadku, gdy podstawowa funkcjonalność realizowana jest w klasach bazowych, natomiast klasy potomne jedynie ją rozszerzają. Interfejsy zaś są optymalnym rozwiązaniem w przypadku, gdy potrzebna jest podobna funkcjonalność w różnych implementacjach kilku klas mających ze sobą niewiele wspólnego.

W rozdziale tym przyjrzymy się bliżej roli interfejsów w Visual Basic .NET i sposobom ich najwłaściwszego wykorzystania w aplikacjach .NET. Ponieważ w poprzednich rozdziałach omówione zostały podstawowe zagadnienia związane z budowaniem interfejsów, teraz naszą uwagę skupimy na sposobach ich najlepszego wykorzystania. Omówimy więc sposoby:

- definiowania interfejsów,
- dziedziczenia pomiędzy interfejsami,
- implementacji interfejsów wewnątrz definicji klas,
- rozpoznawania sytuacji, w której wykorzystanie interfejsu jest najlepszym rozwiązaniem,
- użycia interfejsów do połączeń zwrotnych.

Interfejsy i OOP

Interfejsy, podobnie jak klasy, definiują zbiór właściwości, metod i zdarzeń. Różnica pomiędzy nimi polega na tym, że interfejsy same w sobie nie definiują implementacji, natomiast to klasy, które je wykorzystują, muszą zapewnić ich implementację.

Podstawową zaletą interfejsów jest to, że umożliwiają budowę systemu z opisanych przez interfejsy programowych komponentów, których właściwa implementacja może być zmieniana bez zmiany istniejącego kodu programu korzystającego z tych interfejsów. W rzeczywistości zmianie ulec mogą całe klasy i tak długo, jak implementują one interfejsy, mogą być nadal bez przeszkód używane przez resztę aplikacji.

Wielokrotne dziedziczenie

Podobnie jak ma to miejsce w przypadku klas, interfejsy mogą dziedziczyć po sobie i tworzyć kompletny zorientowany obiektowo projekt. W przypadku interfejsów możliwe jest dziedziczenie po więcej niż jednym obiekcie nadrzędnym, co nie jest możliwe w odniesieniu do klas, a jest to czasem przydatne w projektowaniu zorientowanym obiektowo.

Na przykład *combo box* składa się z pola tekstowego (*test box*) i listy wartości (*list box*). Podczas implementacji combo box za pomocą klas niezbędne jest użycie omówionej w rozdziale 6. kompozycji klas, ponieważ niemożliwe jest dziedziczenie klasy typu combo box jednocześnie po klasie pól tekstowych i klasie list wartości. Sposób użycia interfejsu combo box pokazano na listingu 14.1.

Listing 14.1. *Przykładowy interfejs kontrolki Combo Box*

```
Interface IUIControl
  Sub Paint()
End Interface

Interface ITextBox
  Inherits IUIControl
  Sub SetText(ByVal value As String)
End Interface

Interface IListBox
  Inherits IUIControl

  Sub SetItems(ByVal items() As String)
End Interface

Interface IComboBox
  Inherits ITextBox, IListBox
End Interface
```

Interfejs potomny dziedziczy po interfejsie bazowym wszystkie jego pola. Dlatego też zarówno interfejs `ItextBox`, jak i `IListBox` posiadają podprogram `Paint()`. Interfejs `IComboBox` posiada natomiast podprogramy `SetText()`, `SetItems()`i `Paint()`. Pomimo że podprogram `Paint()` przeniesiony jest z obu interfejsów bazowych, interfejs `IComboBox` ma tylko jedną jego implementację.

Polimorfizm interfejsów

Interfejsy Visual Basic .NET zapewniają kolejny sposób realizacji polimorfizmu. Polimorfizm jest w rzeczywistości nieodłączną cechą interfejsów, ponieważ z założenia nie posiadają one implementacji. Dlatego też każda klasa implementująca interfejs posiada odmienną jego implementację, co stanowi istotę polimorfizmu.

Przykładem polimorfizmu jest definiowanie kilku klas, które implementują ten sam interfejs. Używając interfejsów zamiast klas, zauważamy, w jaki sposób interfejsy obsługują polimorfizm. Listing 14.2 pokazuje użycie interfejsów w serii obiektów będących wielokątami.

Listing 14.2. *Polimorfizm interfejsów*

```
Public Interface Shape
  Function IsPtInShape(ByVal Pt As System.Drawing.Point) As Boolean
End Interface

Public Class Rectangle
  Implements Shape

  Private rect As Drawing.Rectangle

  Public Function PtInRect(ByVal Pt As System.Drawing.Point) As Boolean _
      Implements Shape.IsPtInShape
    PtInRect = Pt.Equals(rect)
  End Function
End Class

Public Class Circle
  Implements Shape

  Private rgn As Drawing.Region

  Public Function PtInCircle(ByVal Pt As System.Drawing.Point) As Boolean _
     Implements Shape.IsPtInShape
    PtInCircle = Pt.Equals(rgn)
  End Function
End Class

Sub CheckPoint(ByVal ShapeObject As Shape, ByVal Pt As System.Drawing.Point)
  If (ShapeObject.IsPtInShape(Pt)) Then
    Beep()
  End If
End Sub

Sub Test()
  Dim Pt As System.Drawing.Point
  Dim RectObject As Rectangle
  Dim CircleObject As Circle

  CheckPoint(RectObject, Pt)
  CheckPoint(CircleObject, Pt)
End Sub
```

analiza Zauważmy, w jaki sposób podprogram `CheckPoint()` pobiera `Shape` jako parametr, a następnie używa metody interfejsu. W dalszej kolejności wywoływana jest metoda implementująca funkcjonalność interfejsu. Metoda ta jest różna dla każdego typu obiektu. Możemy nazwać tę metodę dowolnie i implementować w sposób, jaki najbardziej nam w określonej sytuacji odpowiada, pod warunkiem zachowania zgodności listy parametrów. W ten sposób klasa może implementować interfejs, zachowując jednocześnie niezmienioną nazwę metody wewnątrz klasy. Możliwe jest również zastąpienie dowolnego obiektu implementującego interfejs `Shape` jako parametru podprogramu `CheckPoint()`.

Tworzenie interfejsu Contract

Interfejs reprezentuje „umowę" z klasą. Klasa, używając słowa kluczowego `Implements`, „przystaje na umowę" mówiącą o tym, że będzie implementować interfejs. Po wyrażeniu przez klasę chęci implementacji interfejsu jest ona zobowiązana do implementacji całego interfejsu. Klasa nie może zaimplementować części interfejsu, wypełniając jednocześnie zaakceptowany wcześniej kontrakt. Dlatego też, jeśli klasa implementuje interfejs, użytkownicy klasy pewni są tego, że mogą używać wszystkich metod zdefiniowanych w tym interfejsie, ponieważ z całą pewnością zostaną one zaimplementowane.

Należy pamiętać, że klasa może wypełnić umowę, dostarczając w szczególnym przypadku takiej implementacji, która nie wykonuje żadnej czynności. Jedynym warunkiem jest zgodność w stosunku do zapisów zdefiniowanych w interfejsie dotyczących parametrów i (w przypadku funkcji) zwracanych wartości. Wszystko inne zależne jest od klasy implementującej interfejs. W rzeczywistości zaletą użycia interfejsów jest to, że klasa decyduje o zastosowaniu określonej implementacji w konkretnym przypadku. Przykładowa klasa przedstawionej na listingu 14.3 implementuje trzy różne interfejsy.

Listing 14.3. *Implementacja wielu interfejsów przez jedną klasę oraz użycie obiektów klasy jako implementacji każdego z interfejsów*

```
Public Interface IEmployee
  Function GetSalary() As Decimal
End Interface

Public Interface IParent
  Function HasChildren() As Boolean
End Interface

Public Interface IFriend
  Function NumberOfFriends() As Integer
End Interface

Public Class Person
  Implements IEmployee
  Implements IParent
  Implements IFriend
```

```
  Private m_dSalary As Decimal
  Private m_bChildren As Boolean
  Private m_nFriends As Integer

  Public Function Salary() As Decimal Implements IEmployee.GetSalary
    Salary = m_dSalary
  End Function

  Public Function HasChildren() As Boolean Implements IParent.HasChildren
    HasChildren = m_bChildren
  End Function

  Public Function HowManyFriends() _
  As Integer Implements IFriend.NumberOfFriends
    HowManyFriends = m_nFriends
  End Function
End Class

Module PersonUser
  Sub CheckSalary(ByVal Employee As IEmployee)
    ' W tym miejscu należy umieścić kod programu..
    ' Tylko GetSalary() może być używana dla tego obiektu
  End Sub

  Sub CheckFamily(ByVal Parent As IParent)
    ' W tym miejscu należy umieścić kod programu..
    ' Tylko HasChildren() może być używana dla tego obiektu
  End Sub

  Sub CheckFriends(ByVal MyFriend As IFriend)
    ' W tym miejscu należy umieścić kod programu..
    ' Tylko NumberOfFriends() może być używana dla tego obiektu
  End Sub

  Sub TestPerson()
    Dim Person As Person = New Person()

    CheckSalary(Person)
    CheckFamily(Person)
    CheckFriends(Person)
  End Sub
End Module
```

Dla kompletności definicji klasy `Person` niezbędne jest zaimplementowanie wszystkich metod zdefiniowanych w interfejsach `IEmployee`, `IParent` i `IFriend`. Klasa `Person` definiować może również własne cechy charakterystyczne, jednakże te metody, właściwości i zdarzenia nie będą dostępne, gdy klasa będzie używana poprzez te interfejsy.

W każdym z trzech interfejsów pokazanych w podprogramie `TestPerson()` możliwe jest użycie klasy `Person`. Jest to szczególnie użyteczne podczas projektowania aplikacji posiadającej kilka niezależnych lub luźno powiązanych klas, które używane będą zamiennie. Stosując typ interfejsu jako parametr, użytkownik widzi jedynie metody, właściwości i zdarzenia zdefiniowane w interfejsie. Reszta implementacji wewnątrz klasy nie jest dla użytkownika interfejsu widoczna.

Programowanie interfejsów

Poznaliśmy już kilka przykładów zastosowania interfejsów w programowaniu zorientowanym obiektowo. Programowanie z wykorzystaniem interfejsów może jednak nieco różnić się od programowania z użyciem klas.

Pierwszym spostrzeżeniem poczynionym podczas projektowania interfejsów, które mają być implementowane później, jest to, że ich metody nie mają określonych zakresów dostępności. Wszystkie metody są jawne i dostępne do użytku jako publiczne. Na przykład wewnątrz klasy zdefiniować możemy metodę publiczną, chronioną lub prywatną. Interfejsy nie mają tej cechy. Każda metoda zdefiniowana w interfejsie dostępna jest jako publiczna, czyli tak, jakby została zadeklarowana jako `Public`. Nawet jeśli klasa implementująca implementuje metody jako `Private` dla użytkownika korzystającego z interfejsu, są one dostępne tak, jakby były zadeklarowane jako `Public`.

Posiadanie wszystkich metod interfejsu zadeklarowanych jako publiczne na sens, jeśli zdefiniowanie interfejsu było gruntownie przemyślane, gdyż interfejsy definiowane są w celu opisania wspólnych możliwych do użycia cech czy funkcjonalności. Czy więc może znaleźć się w interfejsie coś, co zostało zadeklarowane jako prywatne? Działanie takie nie ma żadnego logicznego wytłumaczenia z wyjątkiem tego, że mogłoby to wymuszać na klasie implementację metody prywatnej, która może być użyta tylko w obrębie danej klasy.

Kolejnym zagadnieniem, które na pierwszy rzut oka wydaje się być problemem (choć w rzeczywistości okazuje się to nieprawdą), jest fakt, że interfejs nie definiuje pól danych. Zadaniem interfejsu jest definiowanie jedynie publicznych interfejsów metod, zdarzeń i właściwości. Ponieważ, jak pamiętamy, interfejsy nie mają implementacji, nie ma sensu definiowanie pól danych, gdyż dostęp do danych powinien być realizowany poprzez ich metody i właściwości.

Ostatnim problemem, który omówimy, jest nieco frustrująca sytuacja, gdy dwa interfejsy mają zaprojektowaną tę samą metodę. Jeśli dwa interfejsy są interfejsami bazowymi dla nowego, trzeciego interfejsu, użytkownik zmuszony jest do określenia, którą z metod chce wywołać. Listing 14.4 pokazuje taki właśnie przypadek.

Listing 14.4. *Interfejs z kolizją nazw używających CType() do wskazania właściwego sposobu użycia metod*

```
Public Interface IArray
  Function Count() As Integer
End Interface

Public Interface ICounter
  Function Count() As Integer
End Interface

Public Interface ICountArray
  Inherits IArray
  Inherits ICounter
End Interface
```

```
Module CountArrayTest
  Sub Test(ByVal x As ICountArray)
    Dim nCount = CType(x, ICounter).Count()
    Dim nArrayCount = CType(x, IArray).Count()
    'Pozostały kod programu..
  End Sub
End Module
```

analiza W kodzie z listingu 14.4 funkcja `CType()` konwertuje wyrażenie precyzujące pierwszy argument (w tym przypadku `x`) do typu używanego w drugim parametrze. Dzięki temu podczas wykonywania kodu wybierany jest właściwy interfejs i używana jest właściwa metoda `Count`.

Nieco wcześniej wspominaliśmy, że jeśli ta sama metoda zawarta jest wewnątrz interfejsu i w jego interfejsie bazowym, to jest ona implementowana tylko raz. Dlatego też nie występuje wtedy problem pokazany na listingu 14.4.

Implementacja interfejsów

Ponieważ interfejs nie posiada implementacji, musi być przed użyciem zdefiniowany tak, aby dostarczał odpowiednich funkcji, metod, zdarzeń i właściwości. Interfejs może być implementowany zarówno przez klasy, jak i definicje struktur zawierających słowo kluczowe `Implements`.

Jak już wspominaliśmy, klasa może implementować jeden lub wiele interfejsów. Przedstawiony poniżej kod pokazuje sposób, w jaki struktura implementuje interfejs podobnie, jak robi to klasa.

```
Structure MyStructure
  Implements IMyInterface

  Public Function InterfaceFunction() _
           As Integer Implements IMyInterface.MyFunction
  End Function
End Structure
```

Jak już wspominaliśmy wcześniej, interfejs może deklarować metody (podprogramy i funkcje), zdarzenia i właściwości. Pamiętajmy jednak, że nie może deklarować pól danych.

Metody

W celu implementacji metod interfejsów poprzez klasy lub struktury, metody te deklarowane są wewnątrz klas lub struktur z wykorzystaniem słowa kluczowego `Implements`. Jedynym wymaganiem implementacji metody interfejsu jest to, aby metoda klasy lub struktury posiadała te same parametry i zwracała wartości zadeklarowane w metodzie interfejsu.

Tak długo jak zgodne są listy parametrów i wartości zwracanych danych, możliwe jest również implementowanie wielu interfejsów przez pojedynczą metodę klasy lub struktury.

Na przykład, zamieszczony poniżej kod pokazuje, jak w jednej funkcji zaimplementować dwie metody `Count()` zdefiniowane w listingu 14.4:

```
Public Function Count() As Integer Implements IArray.Count(), ICounter.Count()
  Count() = m_nCount
End Function
```

Poprzez implementowanie wielu metod interfejsów za pomocą tej samej metody klasy lub struktury możliwe jest powiązanie metod wielu interfejsów z tą samą funkcjonalnością bez konieczności pisania dodatkowego kodu.

Właściwości

Implementacja właściwości interfejsu podobna jest do implementacji metod. Realizowane jest to za pomocą słowa kluczowego `Implements` przypisującego właściwości klas lub struktur do właściwości interfejsów. Właściwości klas implementujące właściwości interfejsu muszą charakteryzować się tym samym typem (`zapis/odczyt`). Listing 14.5 ilustruje implementację właściwości interfejsu.

Listing 14.5. *Implementacja właściwości interfejsu*

```
Public Interface ICompany
  Property Name() As String
End Interface

Public Class Business
  Implements ICompany

  Dim m_sCompanyName As String

  Public Property CompanyName() As String Implements ICompany.Name
    Get
      CompanyName = m_sCompanyName
    End Get
    Set(ByVal Value As String)
      m_sCompanyName = Value
    End Set
  End Property

End Class
```

Jeśli właściwość zadeklarowana jest w interfejsie jako `ReadOnly` lub `WriteOnly`, to implementująca go klasa lub struktura musi implementować te właściwości z takimi samymi atrybutami.

Zdarzenia

Podobnie jak w przypadku metod i właściwości, zdarzenia implementowane są poprzez klasy lub struktury za pomocą słowa kluczowego `Implements`. Jak wspominaliśmy już w rozdziale 2., zdarzenia w definicjach klas nie są powiązane z kodem. Skutkiem użycia słowa kluczowego `Implements` jest literalne połączenie zdarzenia klasy lub struktury ze zdarzeniem interfejsu. Listing 14.6 pokazuje, jak implementować i używać zdarzenia za pośrednictwem interfejsów.

Listing 14.6. *Implementacja i użycie zdarzeń interfejsu*

```
Public Interface ICompany
  Property Name() As String
  Event NameChanged()
End Interface

Public Class Business
  Implements ICompany

  Dim m_sCompanyName As String

  Public Property CompanyName() As String Implements ICompany.Name
    Get
      CompanyName = m_sCompanyName
    End Get
    Set(ByVal Value As String)
      m_sCompanyName = Value

      'Sygnał o zdarzeniu - nazwa została zmieniona
      RaiseEvent NameChanged()
    End Set
  End Property

  Public Event NameChanged() Implements ICompany.NameChanged

End Class

Module CompanyTest
  Sub OnNameChanged()
    Beep()
  End Sub

  Sub Test(ByVal Company As ICompany)
    'Dodanie uchwytu interfejsu zdarzenia NameChanged
    AddHandler Company.NameChanged, AddressOf OnNameChanged

    'Ten fragment kodu powoduje wygenerowanie zdarzenia
    Company.Name = "My Company"
  End Sub
End Module
```

Użycie interfejsu jako wywołania zwrotnego

Używane poprawnie interfejsy mają wiele zalet. Mamy możliwość tworzenia interfejsów wolnych od problemów występujących podczas zmiany kodu bazowego. Prowadzi to również do większej elastyczności implementacji.

Innym przykładem użycia interfejsów jest zdefiniowane wywołania zwrotnego (*callback*). Na przykład, jeśli kilka typów implementuje ten sam interfejs, może być on wykorzystany jako metoda wywołania zwrotnego w sytuacji, w której pojedyncze wywołanie zwrotne jest niewystarczające. Użycie interfejsu pozwala na wywołanie zwrotne zawierające kilka metod, właściwości lub zdarzeń. Listing 14.7 przedstawia sposób użycia interfejsu zdefiniowanego jako wywołanie zwrotne. Ilekroć interfejs ten jest implementowany, obiekt implementujący go wie, jaki interfejs powinien wywołać zwrotnie. W odróżnieniu od pojedynczej funkcji lub podprogramu wywołanie zwrotne jest kompletnym interfejsem.

Listing 14.7. *Deklaracja wywołania zwrotnego sterowania interfejsem użytkownika*

```
Public Interface IUIControl
  Sub Paint()
  Sub ControlPressed()
  Function IsPtInControl(ByVal Pt As Drawing.Point) As Boolean
  Property ControlRect() As Drawing.Rectangle
  Event ControlHit()
End Interface
```

Implementacja interfejsu `IUIControl` poprzez różne klasy dostarcza wspólnego interfejsu używanego podczas pracy z kontrolkami. Listing 14.8 pokazuje prostą implementację kontrolki przycisku (`PushButton`) i przycisku opcji (`RadioButton`).

Listing 14.8. *Implementacja interfejsu IUIControl poprzez klasy PushButton i RadioButton*

```
Public Class PushButton
  Implements IUIControl

  Dim m_rcRect As Drawing.Rectangle

  'Implementacja interfejsu
  '.........................
  Public Sub Paint() Implements IUIControl.Paint
    ' Rysowanie kontrolki
  End Sub

  Public Function IsPtInControl(ByVal Pt As Drawing.Point) _
  As Boolean Implements IUIControl.IsPtInControl
    IsPtInControl = Pt.Equals(ControlRect())
  End Function

  Public Property ControlRect() As Drawing.Rectangle _
  Implements IUIControl.ControlRect
```

```
        Get
          ControlRect = m_rcRect
        End Get
        Set(ByVal Value As Drawing.Rectangle)
          m_rcRect = Value
        End Set
      End Property

      Public Sub ControlPressed() Implements IUIControl.ControlPressed
        RaiseEvent ControlHit()
      End Sub

      Public Event ControlHit() Implements IUIControl.ControlHit

      'Pozostała część kodu klasy
      '.........................

   End Class

   Public Class RadioButton
     Implements IUIControl

     Dim m_rcRect As Drawing.Rectangle

     'Implementacja interfejsu
     '........................
     Public Sub Paint() Implements IUIControl.Paint
       ' Rysowanie kontrolki
     End Sub

     Public Function IsPtInControl(ByVal Pt As Drawing.Point) _
     As Boolean Implements IUIControl.IsPtInControl
       IsPtInControl = Pt.Equals(ControlRect())
     End Function

     Public Property ControlRect() As Drawing.Rectangle _
     Implements IUIControl.ControlRect
       Get
         ControlRect = m_rcRect
       End Get
       Set(ByVal Value As Drawing.Rectangle)
         m_rcRect = Value
       End Set
End Property

     Public Sub ControlPressed() Implements IUIControl.ControlPressed
       RaiseEvent ControlHit()
     End Sub

     Public Event ControlHit() Implements IUIControl.ControlHit

     'Pozostała część kodu klasy
     '.........................

   End Class
```

Implementacja interfejsu `IUIControl` zależy od kontrolek i każda z kontrolek może mieć swoją własną implementację. Używanie kontrolek jest więc teraz ułatwione, ponieważ użytkownicy, chcąc wywołać zwrotnie kontrolkę, kontaktują się jedynie z interfejsem. Ilustruje to listing 14.9.

Listing 14.9. *Użycie interfejsu IUIControl do wywołania zwrotnego do klasy w przypadku wystąpienia zdarzenia*

```
Public Class MyForm
  Inherits System.Windows.Forms.Form

  Dim UIControls(2) As Object

  Public Sub OnMouseClick(ByVal sender As Object, _
  ByVal e As EventArgs) Handles MyBase.Click
    Dim i As Integer

    For i = 0 To 2
      If CType(UIControls(i), IUIControl).IsPtInControl(MousePosition()) Then
        CType(UIControls(i), IUIControl).ControlPressed()
        CType(UIControls(i), IUIControl).Paint()
      End If
    Next
  End Sub
End Class
```

Oczywiście możliwa jest również współpraca interfejsów z asynchronicznymi wywołaniami zwrotnymi służącymi do komunikacji sieciowej, dostępowi do plików itp.

Podsumowanie

Poznaliśmy właśnie sposoby użycia interfejsów w Visual Basic .NET, jako części programowania zorientowanego obiektowo. Zaletą interfejsów, w porównaniu z klasami, jest możliwość dziedziczenia z kilku interfejsów bazowych jednocześnie. Interfejsy są z natury polimorficzne, ponieważ każda implementująca je klasa lub struktura tworzy własną, oryginalną ich implementację. Klasy i struktury mogą implementować więcej niż jeden interfejs, co umożliwia ich wielokrotne użycie w aplikacji.

Pytania i odpowiedzi

Pytanie: Po co tworzyć interfejsy? Czy nie można tego wszystkiego zrobić za pomocą klas, skoro one również wykorzystują metodę dziedziczenia i polimorfizmu?

Odpowiedź: To prawda, interfejsy nie są dziś wykorzystywane tak często jak w poprzednich wersjach Visual Basica, gdy dziedziczenie i polimorfizm nie były możliwe na poziomie klas. Jednakże mają one nadal przewagę nad klasami, ponieważ

umożliwiają definiowanie ogólnych interfejsów implementowanych przez niezależne obiekty, umożliwiają wielokrotne dziedziczenie i sprawiają, że kod jest niewrażliwy na ewentualne przyszłe modyfikacje.

Pytanie: Czy interfejsy w .NET uległy dużym zmianom? Czy możemy nadal używać ich poza aplikacją lub eksportować ?

Odpowiedź: Interfejsy w Visual Basic .NET są generalnie podobne do tych z poprzednich wersji Visual Basica. .NET Framework ukrywa wiele szczegółów implementacyjnych, ale w rzeczywistości bazuje on nadal na identyfikatorach GUID, a dane, przechodząc przez interfejs, nadal są przekształcane.

Warsztaty

W ramach zajęć warsztatowych proponować będziemy zagadki i ćwiczenia, które pomogą uporządkować i przećwiczyć materiał poznany w kończącym się właśnie rozdziale książki. Odpowiedzi przedstawiono w dodatku A na końcu książki.

Quiz

1. Jakie słowo kluczowe deklaruje interfejs?
2. Jakie słowo kluczowe łączy metody, właściwości i zdarzenia z interfejsem?
3. Czy klasa może implementować część interfejsu?
4. Ile interfejsów może implementować klasa?
5. Jaki inny typ poza klasą może implementować interfejs?

Ćwiczenie

Stwórz interfejs `Error` mający metodę o nazwie `DisplayError()`, wyświetlającą komunikat o błędzie. Następnie zadeklaruj klasę `MyClass` implementującą interfejs `Error` i wyświetlającą komunikat o błędach podany w parametrze metody `DisplayError()`.