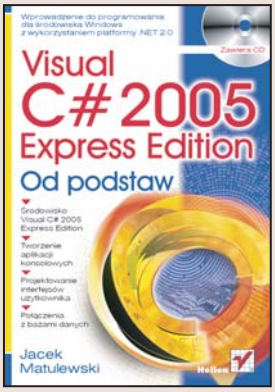
Visual C# 2005 Express Edition. Od podstaw



Autor: Jacek Matulewski
ISBN: 83-246-0334-4
Format: B5, stron: 216
Przykłady na ftp: 2564 kB

Umiejętność tworzenia aplikacji dla platformy .NET coraz częściej pojawia się na liście wymagań stawianych kandydatom do pracy na stanowisku programisty. Popularność języków programowania związanych z tą platformą stale rośnie. C#, jeden z najczęściej wykorzystywanych języków platformy .NET, doczekał się już wersji 2.0. Pojawiły się w niej elementy wyraźnie wskazujące kierunek rozwoju platformy. Dla programistów, którzy chcą poznać możliwości nowej wersji C#, Microsoft przygotował doskonałe narzędzie – środowisko programistyczne Visual C# 2005 Express Edition.

Książka „Visual C# 2005 Express Edition. Od podstaw" jest doskonałym podręcznikiem dla osób zamierzających tworzyć aplikacje z wykorzystaniem języka C# oraz platformy .NET 2.0. Przedstawia zasady korzystania ze środowiska Visual C# 2005 Express Edition, opisuje możliwości języka C# oraz komponenty platformy .NET i związane z nią technologie. Zawiera informacje na temat tworzenia aplikacji konsolowych i „okienkowych", łączenia ich z bazami danych oraz korzystania z bibliotek Win API.

- Komponenty platformy .NET 2.0
- Podstawowe elementy języka C# 2.0
- Programowanie obiektowe w C# 2.0
- Tworzenie okien dialogowych i menu
- Usuwanie błędów z aplikacji
- Wywoływanie funkcji Win API
- Obsługa zdarzeń w aplikacjach
- Implementacja mechanizmu „przeciągnij i upuść"
- Komunikacja z bazami danych za pomocą ADO.NET

Poznaj możliwości platformy .NET 2.0

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

Spis treści

Wstęp 5

Rozdział 1. Poznajemy Visual C# 2005 Express Edition 7

Pierwsza aplikacja 7
Analiza kodu pierwszej aplikacji 11
Metody zdarzeniowe 17

Rozdział 2. Microsoft .NET Framework 2.0 23

Dlaczego platforma .NET? 23
Co nowego w .NET 2.0? 24

Rozdział 3. Język C# 2.0 i kolekcje 27

Podstawowe typy danych 27
- Typy liczbowe oraz znakowy 27
- Operatory 29
- Konwersje typów podstawowych 29
- Łańcuchy 32
- Typ wyliczeniowy 32
- Delegacje i zdarzenia 34

Sterowanie przepływem 35
- Deklaracja i zmiana wartości zmiennej 36
- Instrukcja warunkowa if..else 36
- Instrukcja wyboru switch 36
- Pętle 37

Zwracanie wartości przez argument metody 39
Wyjątki 40
Dyrektywy preprocesora 42
- Kompilacja warunkowa. Ostrzeżenia 42
- Definiowanie stałych preprocesora 43
- Bloki 44

Atrybuty 44
Kolekcje 45
- „Zwykłe” tablice 45
- Pętla foreach 47
- Sortowanie 48
- Kolekcje List i ArrayList 49
- Kolekcja SortedList i inne 51

Rozdział 4. Projektowanie zorientowane obiektowo w C# 2.0 53
Typy wartościowe i referencyjne 53
Przykład struktury (Ulamek) 55
Implementacja interfejsu IComparable 63
Definiowanie typów parametrycznych 65

Rozdział 5. Przykłady aplikacji dla platformy .NET 73
Dywan graficzny 73
Edytor tekstu nieformatowanego 78
Projektowanie interfejsu aplikacji. Menu główne 78
Okna dialogowe i pliki tekstowe 83
Edycja i korzystanie ze schowka 90
Drukowanie 91
Elektroniczna kukułka 99
Ekran powitalny (splash screen) 99
Przygotowanie ikony w obszarze powiadamiania 101
Odtwarzanie pliku dźwiękowego 104
Lista uruchomionych procesów 106

Rozdział 6. Debugowanie kodu w Visual C# 109
Teoria Murphy'ego wyjaśniająca źródło błędów w kodach programów 109
Kontrolowane uruchamianie aplikacji w Visual C# 110
Śledzenie wykonywania programu krok po kroku (F10 i F11) 112
Run to Cursor (Ctrl+F10) 113
Punkt wstrzymania (F9) 114
Okna Locals i Watch 114
Stan wyjątkowy 116

Rozdział 7. Aplikacje konsolowe 119
Klasa Console 119
Informacje o środowisku aplikacji 124

Rozdział 8. Mechanizm PInvoke, funkcje WinAPI i komunikaty Windows 129
Mechanizm PInvoke i funkcje WinAPI 129
Na początek coś prostego 129
Problemy z argumentami 131
Zwracanie wartości przez argumenty 133
Komunikaty Windows 135
Wysyłanie komunikatów Windows 136
Odbieranie komunikatów Windows 138

Rozdział 9. Projektowanie kontrolek .NET 143
Komponent FileListBox 144
Rozbudowa komponentu FileListBox o możliwość zmiany katalogu 154
Właściwości 156
Zdarzenia — interakcja z komponentem 160
Nadpisywanie metody Refresh i automatyczne śledzenie zmian w prezentowanym katalogu 167
Kompilacja komponentu do postaci biblioteki DLL 171
Przykład wykorzystania komponentu — przeglądanie plików tekstowych 175

Rozdział 10. Mechanizm drag & drop 179
Przeciąganie z opóźnieniem 180
Przykład „zaawansowanego" przenoszenia 182

Rozdział 11. Krótki wstęp do aplikacji bazodanowych ADO.NET 189

Skorowidz 201

Rozdział 9.

Projektowanie kontrolek .NET

Narzędzia projektowania RAD oraz języki obiektowe są tak zgodnym małżeństwem dzięki ich potomstwu — komponentom. Komponenty są połączeniem obu idei. Modularność i kopertowanie kodu pozwala na wielokrotne korzystanie z raz przygotowanych rozwiązań bez potrzeby ich modyfikacji. Komponenty tworzą klocki, z których za pomocą myszy można w szybki i całkiem przyjemny sposób stworzyć interfejs programu. Poza typowymi kontrolkami, tj. komponentami wyposażonymi w interfejs, a więc przede wszystkim przyciskami, rozwijanymi listami, panelami i wieloma innymi, w bibliotece komponentów .NET są dostępne także tzw. komponenty niewidoczne, tj. niemające swojej reprezentacji na podglądzie okna w widoku projektowania. Są to dla przykładu komponenty implementujące okna dialogowe, komponent `Timer` służący do odmierzania czasu i cyklicznego wykonywania czynności czy komponent `SerialPort` reprezentujący port szeregowy. Ponadto mamy możliwość korzystania z kontrolek ActiveX zainstalowanych w systemie, które — choć zwykle mniej elastyczne i trudniejsze w obsłudze — stanowią uzupełnienie komponentów .NET[1]. Mimo to zdarzają się sytuacje, w których dostępne komponenty nie wystarczają do zrealizowania naszego projektu. Wówczas warto rozejrzeć się w sieci w poszukiwaniu odpowiedniego komponentu[2]. Modularność i „domknięcie" komponentów w bibliotekach *.dll* sprawia, że są doskonałą formą dzielenia się kodem. Może się jednak zdarzyć, że nasze potrzeby są na tyle wyjątkowe, iż komponent trzeba napisać samodzielnie. I temu właśnie jest poświęcony niniejszy rozdział.

Zanim przystąpimy do projektowania komponentu, musimy sobie odpowiedzieć na pytanie, czy warto podejmować taki wysiłek. Jeżeli kontrolka będzie potrzebna tylko raz, to lepiej zbudować potrzebną konstrukcję z kilku komponentów bezpośrednio w interfejsie.

---

[1] Kontrolki ActiveX można umieścić w podoknie *Toolbox*. W tym celu należy z menu *Tools* wybrać polecenie *Choose Toolbox Items…*. W oknie, które wówczas się pojawi, można wskazać w zakładce *COM Components* pliki *.ocx*, czyli kontrolki ActiveX.

[2] Kilka adresów, które warto sprawdzić to: *http://www.windowsforms.net* (Microsoft, dział Control Gallery), *http://www.codeproject.com* czy *http://codecentral.borland.com* (Borland).

Naprawdę szkoda wówczas czasu i trudu na zabawę w budowanie własnego komponentu. Jednak jeżeli chcemy wykorzystać komponent wiele razy, to warto się postarać i napisać go w miarę ogólnie — koniec końców to się opłaci.

Przykładami komponentów, których rzeczywiście brakuje w bibliotece .NET, także w jej wersji 2.0, są przede wszystkim kontrolki obsługujące pliki, a więc lista zawierająca zawartość wskazanego katalogu, drzewo katalogów czy lista dostępnych dysków. Poniżej zajmiemy się uzupełnieniem tego braku, tworząc kontrolkę `FileListBox` zawierającą listę dostępnych plików i katalogów we wskazanym miejscu na dysku z możliwością przeglądania ich (włącznie ze zmianą katalogu). Kontrolka będzie również pozwalać na zmianę eksplorowanego dysku. Postaramy się, aby powstał dość uniwersalny komponent. Co więcej zaprojektujemy go tak, żeby mógł stanowić wzorzec do pisania kolejnych.

Komponent FileListBox

Komponenty najłatwiej tworzyć z komponentów. W przypadku kontrolki `FileListBox` jest jasne, że najwygodniej zaprojektować ją na podstawie dostępnej w platformie .NET kontrolki `ListBox`. Nie miałoby sensu „rysowanie" nowego interfejsu kontrolki od podstaw. Dzięki temu, moim zadaniem, pozostaje jedynie pobranie listy plików i katalogów oraz odpowiednie zareagowanie na zdarzenia wywoływane przez użytkownika komponentu.

Efektem naszej pracy będzie plik zarządzanej biblioteki DLL zawierający kontrolkę. Będzie on mógł być wykorzystany przez aplikacje platformy .NET pisane w dowolnym języku, także w projektach uruchamianych w systemie Linux z użyciem Mono. Tworzenie od razu komponentu w takiej postaci, tj. w projekcie Class Library, nie jest jednak wygodne. Poręczniej jest utworzyć go najpierw jako składnik projektu Windows Application, a dopiero po zakończeniu projektowania i przetestowaniu przenieść go do osobnego projektu. Tak też zrobimy w opisanym poniżej przypadku.

Ćwiczenie 9.1. Aby stworzyć projekt i interfejs nowego komponentu

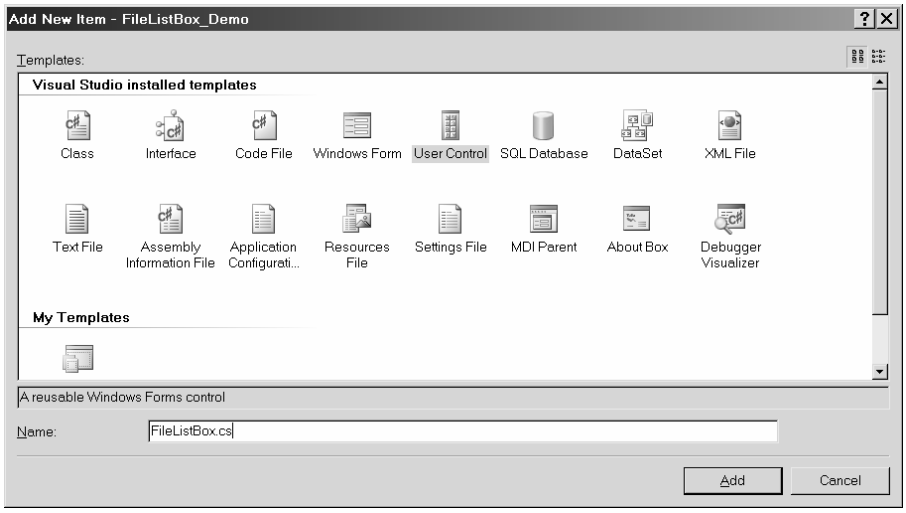
0. Tworzymy projekt typu Windows Application o nazwie `FileListBox_Demo`, w którym będziemy testować nowy komponent.

0. Następnie z menu *Project*, wybieramy pozycję *Add User Control…*.

0. W oknie *Add New Item — FileListBox_Demo* (rysunek 9.1) zaznaczamy pozycję *User Control* i w polu *Name* podajemy nazwę pliku źródłowego nowej kontrolki *FileListBox.cs*. Następnie klikamy *Add*.

0. Od razu zapiszmy nowy plik, klikając przycisk *Save All* na pasku narzędzi (lub naciskając klawisze *Ctrl*+*Shift*+*S*).

0. Przechodzimy do widoku projektowania pliku *FileListBox.cs* (zakładka na górze edytora powinna wskazywać ten plik z dodatkiem *[Design]*). Możemy nieco powiększyć domyślny rozmiar kontrolki.



Rysunek 9.1. *Dodawanie do projektu kontrolki użytkownika*

0. Na interfejsie komponentu umieszczamy komponent `ListBox` z zakładki *Common Controls* podokna *Toolbox*.

0. Zaznaczamy stworzony w poprzednim punkcie komponent `listBox1` i za pomocą okna właściwości ustawiamy jego właściwość `Dock` na `Fill`.

0. Kompilujemy projekt, naciskając *Ctrl+Shift+B*.

0. Przechodzimy do zakładki pliku *Form1.cs* tj. formy, którą wykorzystamy jako środowisko testowania projektowanej kontrolki.

0. W podoknie *Toolbox* zobaczymy grupę *FileListBox_Demo* (pierwsza pozycja), a w niej nasz komponent `FileListBox` i umieszczamy go na podglądzie formy.

Punkt 10. można wykonać jedynie po skompilowaniu pliku komponentu (punkt 8.). Ogólną zasadą takiego tworzenia komponentów powinno być kompilowanie projektu w momencie zakończenia zmian, nawet częściowych, wprowadzanych w kodzie i po zmianie zakładki na plik, w którym jest on testowany, bo podgląd komponentu oraz widoczne w podoknie *Properties* jego właściwości i zdarzenia są widoczne wyłącznie dzięki skompilowanym plikom binarnym, a nie analizie kodu.

Wykonując ćwiczenie 9.1, stworzyliśmy wygodne środowisko testowania komponentów. Każde naciśnięcie klawisza *F5* spowoduje rekompilację zarówno kodu komponentu, jak i aplikacji, w oknie której został umieszczony. Jest to rozwiązanie optymalne, bo pozwala na rozwijanie komponentu i jego równoczesne testowanie bez konieczności zmiany projektu.

Przyjrzyjmy się plikom źródłowym nowej kontrolki. Podobnie jak dla plików *.cs* związanych z formą, także w przypadku kontrolki projektowanej przez użytkownika mamy do czynienia nie z jednym plikiem, ale z trzema: *FileListBox.cs*, *FileListBox.Designer.cs* i *FileListBox.resx*. Dwa pierwsze przechowują klasę `FileListBox`, przy czym, analogicznie do formy, pierwszy jest przeznaczony na kod wpisywany przez programistę

„ręcznie", natomiast drugi — na tę część klasy, którą tworzymy za pomocą narzędzi RAD. Trzeci plik przechowuje zasoby dołączone do komponentu.

Należy zwrócić uwagę, że nasz komponent nie rozszerza klasy `ListBox`. Wykorzystujemy komponent tego typu jako prywatne pole nowego komponentu. Klasą bazową jest natomiast `UserControl`. Dzięki takiemu podejściu możliwe jest ukrycie tych właściwości i metod komponentu `ListBox`, które nie będą miały zastosowania w nowym komponencie, ale zmusza też do samodzielnego udostępnienia tych, które chcemy upublicznić.

Zadaniem naszego komponentu ma być przedstawienie listy plików i katalogów znajdujących się we wskazanym miejscu na dysku. Będzie więc potrzebna zmienna typu `string`, która przechowa pełną ścieżkę do katalogu oraz dwie tablice tego samego typu zawierające listy plików i katalogów. Ponadto w komponencie pokażemy katalog nadrzędny (dwie kropki), oczywiście poza sytuacją, w której prezentowany katalog to katalog główny na dysku. Za listą plików i katalogów umieścimy także listę dysków, aby w każdej chwili użytkownik mógł przenieść się na inny dysk. Wszystkie te elementy powinny podlegać konfiguracji, a więc potrzebne będą również zmienne logiczne, które umożliwią użytkownikowi podjęcie decyzji, czy chce widzieć w komponencie pliki, katalogi oraz dyski. Warto uwzględnić również możliwość filtrowania plików widocznych w liście. Jak widać, zebrał się tego spory zbiór — nasz komponent będzie zatem rozbudowany. Zacznijmy od dodania odpowiednich pól do klasy komponentu.

Ćwiczenie 9.2. Aby zdefiniować prywatne pola wykorzystywane przez FileListBox

0. Przechodzimy do edycji pliku *FileListBox.cs* (należy wybrać odpowiednią zakładkę edytora).

0. W kodzie klasy `FileListBox` wstawiamy deklaracje nowych pól, definiując dla nich przy okazji oddzielny blok edytora o nazwie „Pola prywatne"[3] (listing 9.1).

Listing 9.1. *Pełen kod komponentu z dodanymi definicjami pól*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace FileListBox_Demo
{
      public partial class FileListBox : UserControl
      {
            #region Pola prywatne
            //wewnętrzne
            private string[] listaKatalogow = null;
```

[3] O dyrektywach preprocesora, m.in. dyrektywie `#region` definiującej bloki, Czytelnik może znaleźć więcej informacji w rozdziale 2.

```
            private string[] listaPlikow = null;
            private string[] listaDyskow = null;
            private bool pokazujDwieKropki = true;
            //konfigurowanie komponentu
            private string sciezkaKatalogu = null;
            private bool uwzglednijKatalogi = true;
            private bool uwzglednijPliki = true;
            private bool uwzglednijDyski = true;
            private bool uwzglednijKatalogNadrzedny = true;
            private string filtr = null;
            #endregion

            public FileListBox()
            {
                  InitializeComponent();
            }
      }
}
```

Pola w grupie oznaczonej komentarzem `//wewnętrzne` będą potrzebne do funkcjonowania „silnika" komponentu, natomiast pola z drugiej grupy określą sposób jego funkcjonowania. Po zaprojektowaniu komponentu dostęp do tych drugich powinien być możliwy za pomocą podokna *Properties*. Zrobimy to, definiując właściwości.

Potrzebujemy teraz metody, która będzie pobierała listę plików i podkatalogów znajdujących się w katalogu określonym przez pole `sciezkaKatalogu` i która umieści obie listy w komponencie `listBox1`. Metoda powinna oczywiście uwzględniać wartości pól zdefiniowanych w zadaniu 9.2, m.in. `uwzglednijKatalogi`, `uwzglednijPliki` i `pokazujDwieKropki`. W istocie będzie ona sercem naszego komponentu.

Ćwiczenie 9.3. Aby stworzyć metodę pobierającą pliki i podkatalogi znajdujące się we wskazanym katalogu

0. Na początku pliku *FileListBox.cs* dodajemy deklarację użycia przestrzeni nazw `System.IO`, wpisując linię z poleceniem: `using System.IO;`.

0. Do klasy dodajemy definicję metody `PobierzZawartoscKatalogu` (listing 9.2). Najlepiej wstawić ją zaraz za deklaracją zdefiniowanych wcześniej właściwości, ale w osobnym bloku o nazwie „Metody prywatne"[4].

Listing 9.2. *Blok zawierający najważniejszą metodę komponentu*

```
#region Metody prywatne
private void PobierzZawartoscKatalogu()
{
if (sciezkaKatalogu==null)
sciezkaKatalogu=Directory.GetCurrentDirectory();

pokazujDwieKropki=(sciezkaKatalogu!=Path.GetPathRoot(sciezkaKatalogu)
                  && uwzglednijKatalogNadrzedny);
```

[4] Oczywiście pozycja tej metody w klasie nie jest istotna dla kompilatora. Najważniejsze, żeby nie umieścić jej wewnątrz innej metody lub w ogóle poza klasą.

```
    if (!Directory.Exists(sciezkaKatalogu))
       throw new Exception("Katalog "+sciezkaKatalogu+
                           " nie istnieje!");

    listBox1.Items.Clear();

    if (uwzglednijKatalogi)
       {
       if (pokazujDwieKropki) listBox1.Items.Add("[..]");
       listaKatalogow=Directory.GetDirectories(sciezkaKatalogu);
       Array.Sort(listaKatalogow);
       listBox1.Items.AddRange(listaKatalogow);
       }
    if (uwzglednijPliki)
       {
       listaPlikow=Directory.GetFiles(sciezkaKatalogu);
       Array.Sort(listaPlikow);
       listBox1.Items.AddRange(listaPlikow);
       }
    if (uwzglednijDyski)
       {
       listaDyskow=Directory.GetLogicalDrives();
       listBox1.Items.AddRange(listaDyskow);
       }
}
#endregion
```

Jak działa ta metoda? Na początku sprawdzamy, czy właściwość określająca ścieżkę do katalogu nie jest przypadkiem pusta. Jeżeli jest, to umieszczamy w niej ścieżkę do bieżącego katalogu roboczego odczytanego za pomocą `Directory.GetCurrentDirectory`. Kolejny warunek sprawdza, czy katalog wskazywany przez właściwość `sciezkaKatalogu` istnieje na dysku. Jeśli nie — zgłaszamy wyjątek z odpowiednim komunikatem.

Następnie sprawdzamy, czy na początku listy powinny znajdować się dwie kropki reprezentujące katalog nadrzędny. Abyśmy mogli dodać owe dwie kropki, muszą być spełnione dwa warunki. Katalog, którego zawartość zamierzamy przedstawić, nie może być katalogiem głównym dysku i jednocześnie właściwość `uwzglednijKatalogNadrzedny` powinna być ustawiona na `true`.

Polecenie `listBox1.Items.Clear();` czyści zawartość komponentu `listBox1`. Następnie przystępujemy do odczytania listy plików, katalogów i dysków za pomocą odpowiednich metod statycznych klasy `Directory`. Sortujemy je i umieszczamy w `listBox1` za pomocą jego metody `listBox1.Items.AddRange`, uwzględniając wartość odpowiednich pól `uwzglednij...`.

Jak widać, zasadnicze znaczenie pełni w powyższym kodzie klasa `Directory`, która dostarcza statyczne metody pozwalające na pobieranie listy znajdujących się w niej plików i katalogów, co nas najbardziej interesuje, ale nie tylko: z pomocą tej klasy jest również możliwe manipulowanie katalogami (np. ich tworzenie lub usuwanie).

Najlepiej dodać wywołanie przygotowanej w ćwiczeniu 9.3 metody do konstruktora, wówczas po utworzeniu instancji komponentu `FileListBox` będzie on od razu pokazywał zawartość bieżącego katalogu. Ważne, żeby zrobić to za wywołaniem metody `InitializeComponent` inicjującej komponenty umieszczone w trakcie projektowania.

Ćwiczenie 9.4. Aby wykorzystać metodę PobierzZawartoscKatalogu do prezentacji katalogu w komponencie

0. Aby poprawić przejrzystość kodu, zwijamy w edytorze rejony „Pola prywatne" oraz „Metody prywatne".

0. Odnajdujemy konstruktor klasy, tj. metodę o sygnaturze `public FileListBox();` i dodajemy do niej polecenie `PobierzZawartoscKatalogu();` wywołujące metodę zdefiniowaną w zadaniu 9.3 (listing 9.3).

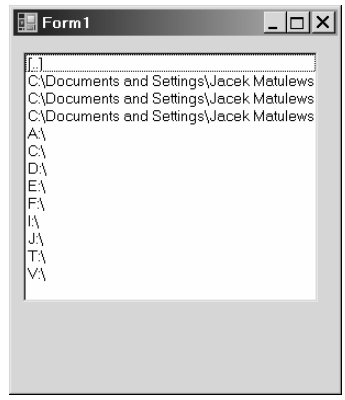
Listing 9.3. *Konstruktor uzupełniony o wywoływanie metody pobierającej zawartość bieżącego katalogu*

```
public FileListBox()
{
   InitializeComponent();
   PobierzZawartoscKatalogu();
}
```

0. Kompilujemy i uruchamiamy projekt, naciskając klawisz *F5*.

Nie musimy inicjować pola `sciezkaKatalogu`, bo metoda `PobierzZawartoscKatalogu` po wykryciu, że jest ono niezainicjowane, zapisze do niej ścieżkę bieżącego katalogu.

Jak zwykle przy pierwszej kompilacji kodu nie wszystko działa poprawnie (jeśli w ogóle się kompiluje i uruchamia). Po uruchomieniu aplikacji przekonamy się, że komponent przedstawia pliki z pełną ścieżką dostępu (rysunek 9.2). Musimy zatem zmodyfikować metodę `PobierzZawartoscKatalogu` w taki sposób, aby była widoczna tylko sama nazwa pliku lub katalogu.



Rysunek 9.2. *Pełna ścieżka do plików nie jest pożądana*

Ćwiczenie 9.5. Aby z pełnej ścieżki dostępu do plików wyłonić samą nazwę plików i katalogów

Modyfikujemy metodę `PobierzZawartoscKatalogu` tak, że na każdą pozycję z list `listaPlikow` i `listaKatalogow` przed dodaniem ich do `listBox1` działamy metodą `Path.GetFileName` (listing 9.4).

Listing 9.4. *Modyfikacja metody PobierzZawartoscKatalogu*

```
if (uwzglednijKatalogi)
{
   if (pokazujDwieKropki) listBox1.Items.Add("[..]");
   listaKatalogow=Directory.GetDirectories(sciezkaKatalogu);
   Array.Sort(listaKatalogow);
   //listBox1.Items.AddRange(listaKatalogow);
   for (int i = 0; i < listaKatalogow.Length; i++)
      listBox1.Items.Add(Path.GetFileName(listaKatalogow[i]));
}
if (uwzglednijPliki)
{
   listaPlikow=Directory.GetFiles(sciezkaKatalogu);
   Array.Sort(listaPlikow);
   //listBox1.Items.AddRange(listaPlikow);
   for (int i = 0; i < listaPlikow.Length; i++)
      listBox1.Items.Add(Path.GetFileName(listaPlikow[i]));
}
if (uwzglednijDyski)
{
   listaDyskow=Directory.GetLogicalDrives();
   listBox1.Items.AddRange(listaDyskow);
}
```

W tej sytuacji metodę `Items.AddRange` dodającą całą tablicę do `listBox1` zastąpiliśmy wywoływaną w pętli metodą `Items.Add` dodającą tylko jedną, już zmodyfikowaną, pozycję.

Zamiast typowej pętli `for` możemy wykorzystać także pętlę `foreach` omówioną w rozdziale 3. (listing 9.5).

Listing 9.5. *Wersja metody PobierzZawartoscKatalogu z pętlą foreach*

```
if (uwzglednijKatalogi)
{
   if (pokazujDwieKropki) listBox1.Items.Add("[..]");
   listaKatalogow=Directory.GetDirectories(sciezkaKatalogu);
   Array.Sort(listaKatalogow);
   foreach (string katalog in listaKatalogow)
      listBox1.Items.Add(Path.GetFileName(katalog));
}
if (uwzglednijPliki)
{
   listaPlikow=Directory.GetFiles(sciezkaKatalogu);
   Array.Sort(listaPlikow);
   foreach (string plik in listaPlikow)
```

```
        listBox1.Items.Add(Path.GetFileName(plik));
    }
    if (uwzglednijDyski)
    {
       listaDyskow=Directory.GetLogicalDrives();
       listBox1.Items.AddRange(listaDyskow);
    }
```

Przejdźmy do zakładki pliku *Form1.cs*. W widoku projektowania formy, na której został umieszczony komponent, powinna być widoczna zawartość katalogu Visual C# (to jest bieżący katalog dla środowiska w trakcie projektowania)[5]. Możemy w ten sposób sprawdzić, czy zmiany w kodzie odniosły zamierzony skutek (wcześniej należy koniecznie kod skompilować: *Ctrl+Shift+B*).

Po korekcie wprowadzonej w ostatnim zadaniu nazwy powinny być już wyświetlane bez ścieżki dostępu, ale warto byłoby jeszcze podkreślić jakoś różnicę między plikami a podkatalogami znajdującymi się w prezentowanym katalogu. Proponuję do nazw katalogów dodać nawiasy kwadratowe.

Ćwiczenie 9.6. Aby do nazw katalogów dodać wyróżniający je nawias kwadratowy „[ ]”

Modyfikujemy argument polecenia `listBox1.Items.Add`:

```
listBox1.Items.Add("["+Path.GetFileName(listaKatalogow[i])+"]");
```

lub

```
listBox1.Items.Add("["+Path.GetFileName(katalog)+"]");
```

w zależności, czy używamy standardowej pętli `for` czy pętli `foreach` przedstawionej w listingu 9.5.

Natomiast w przypadku listy dysków usuniemy znak ukośnika (*slash*), pozostawiając jedynie symbol typu „C:” i otoczymy go znakami < oraz >.

Ćwiczenie 9.7. Aby do symbolów dysków dodać wyróżniające je znaki „<” i „>”

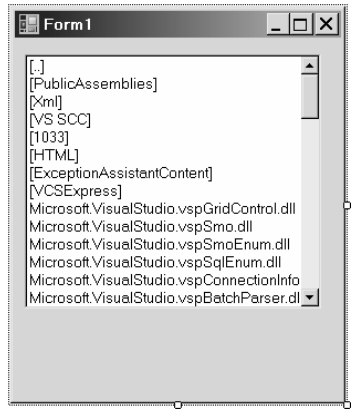
Modyfikujemy argument polecenia `listBox1.Items.Add`, jak w listingu 9.6.

Listing 9.6. *Modyfikacje metody PobierzZawartoscKatalogu*

```
if (uwzglednijDyski)
{
   listaDyskow=Directory.GetLogicalDrives();
   foreach (string dysk in listaDyskow)
   listBox1.Items.Add("<"+dysk.Substring(0,2)+">");
}
```

[5] Po uruchomieniu projektowanej aplikacji katalogiem bieżącym jest podkatalog *bin\Debug* katalogu projektu.

Z każdego łańcucha wybieramy jedynie pierwsze dwa znaki (metoda `Substring`) i dodatkowo otaczamy je znakami „<” i „>”. Po zmianach z ćwiczeń 9.6 i 9.7 komponent powinien wyglądać znacznie lepiej (rysunek 9.3).



Rysunek 9.3. *Wyróżnienie katalogów znacznie zwiększa czytelność listy*

Ostatnia operacja, jaką musimy dodać do metody `PobierzZawartoscKatalogu`, to uwzględnienie możliwości filtrowania, a więc uwzględnienie wartości pola `filtr`.

Ćwiczenie 9.8. Aby uwzględnić filtrowanie plików z maską określoną przez pole filtr

1. Wystarczy dodać do kodu jedną linię, która w przypadku, gdy referencja `filtr` nie jest równa `null` (nie jest pusta) wykorzystuje ją jako drugi argument przeciążonej metody `Directory.GetFiles`. W przeciwnym przypadku jest stosowana dotychczasowa jednoargumentowa wersja metody `GetFiles` (listing 9.7).

Listing 9.7. *Modyfikacje metody PobierzZawartoscKatalogu dodające filtrowanie prezentowanych plików*

```
if (uwzglednijPliki)
{
   if (filtr != null)
      listaPlikow = Directory.GetFiles(sciezkaKatalogu, filtr);
   else
      listaPlikow=Directory.GetFiles(sciezkaKatalogu);
   Array.Sort(listaPlikow);
   //listBox1.Items.AddRange(listaPlikow);
   foreach (string plik in listaPlikow)
      listBox1.Items.Add(Path.GetFileName(plik));
}
```

2. Aby przetestować działanie filtru, dodajmy do metody inicjującej nasz komponent polecenia, ustalając tymczasowo ścieżkę i filtr. W tym celu:

   a) odnajdujemy konstruktor klasy `FileListBox`;

b) przed umieszczonym tam wcześniej wywoływaniem metody `PobierzZawartoscKatalogu` wstawiamy dwa polecenia wyróżnione w listingu 9.8.

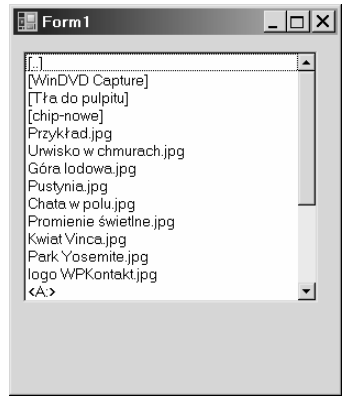
Listing 9.8. *Kolejne modyfikacje konstruktora klasy komponentu*

```
public FileListBox()
{
InitializeComponent();
   sciezkaKatalogu="C:\\Documents and Settings\\Jacek Matulewski\\Moje
      dokumenty\\Moje obrazy";
   filtr="*.jpg";
   PobierzZawartoscKatalogu();
}
```

W pierwszym punkcie wykorzystaliśmy wersję przeciążonej metody `Directory.GetFiles`, w której drugi argument jest maską[6] plików zwracanych w tablicy łańcuchów przez tę funkcję. Podobnie przeciążona jest metoda `Directory.GetDirectories`, ale filtrowanie katalogów nie ma w naszym przypadku sensu.

Możemy teraz skompilować i uruchomić projekt. Komponent przybrał ostateczny wygląd (rysunek 9.4).

Rysunek 9.4. *Filtrowanie plików z maską *.jpg*



Zmieńmy teraz ścieżkę podaną w listingu 9.8 tak, żeby wskazywała katalog główny jakiegoś dysku, aby upewnić się, że w takiej sytuacji nie są pokazywane dwie kropki symbolizujące katalog nadrzędny. Warto również przetestować pola `uwzgledniJKatalogi`, `uwzgledniJPliki` i `uwzgledniJDyski`.

Jeżeli wszystko jest w porządku, usuwamy z konstruktora dwie linie dodane w ćwiczeniu 9.8.

[6] Czyli np. **.jpg* lub **.**.