Delphi 2005. 303 gotowe rozwiązania

Autorzy: Jacek Matulewski,
Sławomir Orłowski,Michał Zieliński
ISBN: 83-7361-923-2
Format: B5, stron: 648

Najnowsza wersja Delphi – jednego z flagowych produktów firmy Borland, łączy w sobie trzy znane środowiska programistyczne: tradycyjne Delphi, Delphi dla .NET oraz C# Builder. Zawarcie w jednym systemie tak wielu możliwości pozwala programistom tworzyć różne rodzaje aplikacji – zarówno te, do których pisania przyzwyczaili się, korzystając z poprzednich wersji Delphi, jak i programy wykorzystujące zyskującą coraz większą popularność platformę .NET. Dzięki możliwości przenoszenia kodu do środowiska Kylix pracującego pod kontrolą systemu operacyjnego Linux Delphi jest jedną z najbardziej uniwersalnych platform dla programistów i projektantów rozbudowanych aplikacji przeznaczonych do działania w sieci.

„Delphi 2005. 303 gotowe rozwiązania" to książka dla programistów zainteresowanych tworzeniem rozbudowanych aplikacji i wykorzystywaniem w tym celu wszystkich możliwości oferowanych przez najnowszą wersję środowiska Delphi. Opisuje zasady tworzenia programów wykorzystujących biblioteki systemowe Windows i interfejs WinAPI oraz przedstawia rozwiązania problemów, na jakie można się natknąć pisząc aplikację wykraczającą poza możliwości oferowane przez standardowe komponenty dołączane do Delphi. Książka jest zbiorem funkcji, klas i sztuczek, za pomocą których można rozszerzyć możliwości biblioteki VCL, VCL.NET i Windows Forms, sięgając głębiej do zasobów systemu.

- Elementy środowiska Delphi 2005
- Programowanie obiektowe w Delphi 2005
- Obsługa wyjątków w języku Object Pascal
- Korzystanie z informacji zapisanych w rejestrze
- Obsługa systemu plików
- Tworzenie wygaszaczy ekranu
- Projektowanie własnych komponentów
- Stosowanie interfejsu WinAPI w aplikacjach
- Wykorzystywanie możliwości multimedialnych Windows
- Obsługa mechanizmów OLE oraz kontrolek ActiveX
- Elementy biblioteki Indy
- Tworzenie aplikacji sieciowych
- Grafika i biblioteki DirectX.NET

Jeśli chcesz tworzyć profesjonalne aplikacje, wykorzystując Delphi, przeczytaj tę książkę – znajdziesz tu wiadomości, dzięki którym unikniesz wielu problemów.

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

Spis treści

Rozdział 7.

Kontrolka Windows Forms

Przykładem kontrolek, których rzeczywiście brakuje w bibliotece .NET, są komponenty wizualne związane z plikami, a więc lista zawierająca zawartość wskazanego katalogu, drzewo katalogów czy lista dostępnych dysków. Poniżej zajmiemy się uzupełnieniem tego braku, tworząc komponent `TFileListBox` zawierający listę dostępnych plików i katalogów we wskazanym miejscu na dysku (z możliwością wędrowania po nich, łącznie ze zmianą dysku). To będzie dość uniwersalny komponent, zaprojektowany tak, żeby mógł stanowić dobry wzorzec dla pisania kolejnych.

Omówiony poniżej komponent został wcześniej opisany w wersji dla języka C# w książce napisanej przez Jacka Matulewskiego pt. *Język C#. Programowanie dla platformy .NET w środowisku Borland C# Builder*. Wydawnictwo HELP, 2004.

Komponenty najłatwiej tworzyć z innych komponentów. W przypadku komponentu `TFileListBox` jest oczywiste, że najwygodniej oprzeć się na komponencie `ListBox` dostępnym w bibliotece Windows Forms z platformy Microsoft .NET. Nie miałoby przecież sensu „rysowanie" nowego interfejsu komponentu od podstaw. Dzięki temu pozostaje nam jedynie problem pobrania listy plików i katalogów oraz odpowiedniego zareagowania na zdarzenia wywoływane przez użytkownika komponentu.

Część statyczna kontrolki

W praktyce najłatwiej projektować komponenty tworząc najpierw projekt aplikacji, a dopiero potem dodać do niego moduł komponentu. Dzięki temu uzyskujemy wygodną platformę testowania komponentu bez konieczności jego instalowania w środowisku. Tak postąpimy w poniższych przykładach — stworzymy projekt aplikacji, która będzie pełniła funkcję środowiska testowania, i dopiero do tego projektu dodamy komponent. Kompilacją komponentu do postaci DLL zajmiemy się po zakończeniu jego projektowania i testowania.

Projekt 122. Tworzenie projektu aplikacji testującej oraz interfejsu nowego komponentu

1. Tworzymy projekt *FileListBox_Demo*, w którym będziemy testować nowy komponent: z menu *File*, podmenu *New* wybieramy *Windows Forms Application — Delphi for .NET*.
2. Do projektu dodajemy moduł przyszłego komponentu:
   a) z menu *File*, podmenu *New*, wybieramy pozycję *Other...*;
   b) w oknie *New Items* na zakładce *Delphi for .NET Projects*, *New Files* zaznaczamy pozycję *User Control for Windows Forms* i klikamy *OK*.

Na tej samej zakładce dostępna jest również pozycja *Component for Windows Forms*, ale w jego rozwijaniu niedostępny jest widok projektowania, który mamy w przypadku kontrolki.

3. Od razu zapiszmy projekt wraz modułem kontrolki na dysku, klikając przycisk *Save All* (lub klawisze *Shift+Ctrl+S*) na pasku narzędzi. Zapiszmy kod pustego na razie komponentu do pliku *FileListBox.pas*. W menadżerze projektu nowy plik wyróżniony jest specyficzną ikoną wyróżniającą komponenty użytkownika.
4. W widoku projektowania pliku *FileListBox.pas* (zakładka na górze edytora powinna wskazywać ten plik) w inspektorze obiektów widoczne są własności klasy `TUserControl`. Przede wszystkim dzięki pozycji `(Name)` (w grupie *Design*) zmieniamy jej nazwę na `TFileListBox` (nie możemy w Delphi użyć nazwy `FileListBox`, ponieważ została już użyta do nazwania modułu).
5. W widoku projektowania zwiększamy rozmiar kontrolki.
6. Na palecie komponentów zaznaczamy `ListBox` i umieszczamy go na interfejsie komponentu.
7. Zaznaczamy go i za pomocą inspektora ustawiamy jego własność `Dock` na `Fill`.
8. Jego własność `HorizontalScrollbar` zmieniamy na `True`.
9. Kompilujemy projekt naciskając *Ctrl+F9*.
10. Przechodzimy na zakładkę modułu *WinForm* — naszego środowiska testowania komponentu.
11. W palecie komponentów znajdujemy grupę *My User Controls* (na końcu listy), a w niej nasz komponent `TFileListBox`. Umieśćmy go na powierzchni formy.

Punkt 11. będzie możliwy do wykonania jedynie dzięki skompilowaniu pliku komponentu (punkt 9.). Ogólną zasadą w tworzeniu komponentów w ten sposób powinno być kompilowanie projektu w momencie zakończenia, nawet częściowych, modyfikacji wprowadzanych w kodzie i zmianie zakładki na plik, w którym jest on testowany. Podgląd komponentu oraz widoczne w inspektorze jego własności i zdarzenia są bowiem widoczne dzięki skompilowanym plikom binarnym, a nie analizie kodu.

Wykonując czynności z projektu 122. stworzyliśmy wygodne środowisko testowania komponentu. Każde naciśnięcie klawisza *F9* spowoduje rekompilację zarówno kodu komponentu, jak i aplikacji, na której oknie został umieszczony.

Należy zwrócić uwagę, że nasz komponent nie rozszerza klasy `ListBox`. Wykorzystujemy komponent tego typu jako prywatne pole nowego komponentu (relacja *ma* zamiast *jest*). Klasą bazową jest natomiast `System.Windows.Forms.UserControl`. Jest to standardowa metoda, która pozwala na ukrycie niektórych własności i metod komponentu `ListBox` (w bibliotece Windows Forms nie ma bowiem znanych z VCL i VCL.NET komponentów typu `TCustomListBox`).

Projekt 123. Zdefiniuj prywatne pola wykorzystywane przez TFileListBox

Zanim przystąpimy do tworzenia komponentu, warto przemyśleć, jak ma być on zbudowany i jaką powinien pełnić funkcję. Warto spisać sobie nawet jego strukturę na kartce, choć szczerze mówiąc sam robię to dopiero wtedy, gdy pogubię się w kodzie. Zadaniem naszego komponentu ma być zaprezentowanie listy plików i katalogów znajdujących się we wskazanym miejscu na dysku. Potrzebna więc będzie 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, gdy prezentowany katalog to główny katalog na dysku. Za listą plików i katalogów umieścimy także listę dysków, aby w każdej chwili użytkownik mógł przenieść się na inny dysk. Wszystko te elementy powinny podlegać konfiguracji, a więc potrzebne nam będą również zmienne logiczne, które zapewnią możliwość decyzji użytkownikowi, czy chce widzieć w komponencie pliki, katalogi i (lub) dyski. Warto uwzględnić również możliwość filtrowania widocznych w liście plików.

1. Przechodzimy do pliku *FileListBox.pas* (zakładka na górze edytora).
2. W edytorze kodu (zakładka *Code* na dole edytora), do sekcji `interface` wstawiamy deklaracje nowych pól klasy `TFileListBox`, definiując przy okazji dla nich oddzielny blok edytora o nazwie „Prywatne pola i metody” (listing 7.1):

Listing 7.1. *Moduł komponentu*

```
unit FileListBox;

interface

uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data;

type
  TFileListBox = class(System.Windows.Forms.UserControl)
  {$REGION 'Designer Managed Code'}
  strict private
    /// <summary>
    /// Required designer variable.
    /// </summary>
```

```
    Components: System.ComponentModel.Container;
    ListBox1: System.Windows.Forms.ListBox;
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    procedure InitializeComponent;
  {$ENDREGION}
  {$REGION 'Prywatne pola i metody'}
  strict private
        //pola wewnetrzne
        listaKatalogow :Array of String;
        listaPlikow :Array of String;
        listaDyskow :Array of String;
        pokazujDwieKropki :Boolean;
        //pola przechowujace ustawienia komponentu
        sciezkaKatalogu :String;
        uwzglednijKatalogi :Boolean;
        uwzglednijPliki :Boolean;
        uwzglednijDyski :Boolean;
        uwzglednijKatalogNadrzedny :Boolean;
        filtr :String;
  {$ENDREGION}
  strict protected
    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    procedure Dispose(Disposing: Boolean); override;
  private
    { Private Declarations }
  public
    constructor Create;
  end;

  [assembly: RuntimeRequiredAttribute(TypeOf(TFileListBox))]


//dalsza modułu
```

3. W konstruktorze umieszczamy polecenia inicjujące nowe pola klasy (listing 7.2):

Listing 7.2. *Konstruktor klasy inicjuje własności komponentu wartościami domyślnymi*

```
constructor TFileListBox.Create;
begin
  inherited Create;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent;

  //inicjacja prywatnych pol klasy
  pokazujDwieKropki:=True;
  sciezkaKatalogu:=nil;
  uwzglednijKatalogi:=True;
  uwzglednijPliki:=True;
```

```
  uwzglednijDyski:=True;
  uwzglednijKatalogNadrzedny:=True;
  filtr:=nil;
end;
```

Pola w grupie oznaczonej komentarzem `//wewnetrzne` będą potrzebne do funkcjonowania „silnika" komponentu, natomiast własności z drugiej grupy określają jego funkcjonowanie i po zbudowaniu komponentu dostęp do nich powinien być możliwy w inspektorze obiektów. Zrealizujemy to definiując własności i dlatego wszystkie te pola klasy zadeklarowane zostały jako prywatne. A dokładniej — ściśle prywatne (ang. *stricte private*), a więc niedostępne nawet dla funkcji i procedur zdefiniowanych w module, ale nie będących metodami klasy.

Potrzebujemy teraz prywatnej 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 zdefiniowanych w projekcie 123 pól komponentu. Będzie to serce komponentu.

Projekt 124. Stworzyć metodę pobierającą nazwy plików i podkatalogów znajdujących się we wskazanym katalogu

1. Do sekcji `uses` modułu *FileListBox* dodajemy deklarację użycia przestrzeni nazw *System.IO*, dorzucając ją do listy.

```
uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data, System.IO;
```

2. Do zdefiniowanego przez nas bloku *Prywatne pola i metody*, za deklaracjami pól, umieszczamy deklarację metody (listing 7.3):

Listing 7.3. *Deklaracja metody PobierzZawartoscKatalogu dodana do bloku Prywatne pola i metody*

```
{$REGION 'Prywatne pola i metody'}
  strict private
        //pola wewnetrzne
        listaKatalogow :array of String;
        listaPlikow :array of String;
        listaDyskow :array of String;
        pokazujDwieKropki :Boolean;
        //pola przechowujace ustawienia komponentu
        sciezkaKatalogu :String;
        uwzglednijKatalogi :Boolean;
        uwzglednijPliki :Boolean;
        uwzglednijDyski :Boolean;
        uwzglednijKatalogNadrzedny :Boolean;
        filtr :String;
    //metody
    procedure PobierzZawartoscKatalogu;
  {$ENDREGION}
```

3. Do sekcji `implementation` dodajemy natomiast definicję tej metody (listing 7.4):

Listing 7.4. *Serce komponentu*

```
procedure TFileListBox.PobierzZawartoscKatalogu;
type TablicaObiektow = array of TObject;
begin
if sciezkaKatalogu=nil then sciezkaKatalogu:=Directory.GetCurrentDirectory;

pokazujDwieKropki:=(sciezkaKatalogu<>Path.GetPathRoot(sciezkaKatalogu))
                   and uwzglednijKatalogNadrzedny;

if (not Directory.Exists(sciezkaKatalogu)) then
         raise Exception.Create('Katalog '+sciezkaKatalogu+' nie istnieje!');

ListBox1.Items.Clear;

if uwzglednijKatalogi then
         begin
         if pokazujDwieKropki then ListBox1.Items.Add('[..]');
         listaKatalogow:=Directory.GetDirectories(sciezkaKatalogu);
         System.Array.Sort(listaKatalogow);
         listBox1.Items.AddRange(listaKatalogow as TablicaObiektow);
         end;
if uwzglednijPliki then
         begin
         listaPlikow:=Directory.GetFiles(sciezkaKatalogu);
         System.Array.Sort(listaPlikow);
         listBox1.Items.AddRange(listaPlikow as TablicaObiektow);
         end;
if uwzglednijDyski then
         begin
         listaDyskow:=Directory.GetLogicalDrives();
         listBox1.Items.AddRange(listaDyskow as TablicaObiektow);
         end;
end;
```

4. Do konstruktora klasy dodajemy wywołanie zdefiniowanej metody (listing 7.5):

Listing 7.5. *Wywołanie funkcji PobierzZawartoscKatalogu finalizuje inicjację komponentu*

```
constructor TFileListBox.Create;
begin
  inherited Create;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent;

  //inicjacja prywatnych pol klasy
  pokazujDwieKropki:=True;
  sciezkaKatalogu:=nil;
  uwzglednijKatalogi:=True;
  uwzglednijPliki:=True;
  uwzglednijDyski:=True;
  uwzglednijKatalogNadrzedny:=True;
```

```
  filtr:=nil;

  //zapelnianie listy
  PobierzZawartoscKatalogu;
end;
```

5. Kompilujemy i uruchamiamy projekt naciskając klawisz F9.

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

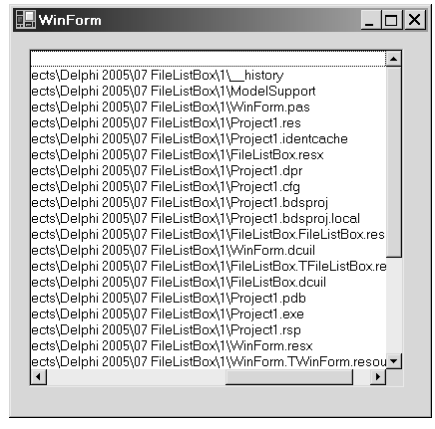
Następnie sprawdzamy, czy na początku listy powinny znajdować się dwie kropki reprezentujące katalog nadrzędny (to okaże się bardzo wygodne, gdy będzie już możliwe wędrowanie po katalogach za pomocą tego komponentu). Abyśmy mogli dodać owe dwie kropki, muszą być spełnione dwa warunki. Po pierwsze katalog, którego zawartość zamierzamy zaprezentować, nie może być katalogiem głównym dysku oraz równocześnie własność `uwzglednijKatalogNadrzedny` musi być ustawiona na `True`.

Za pomocą polecenia `ListBox1.Items.Clear` czyścimy zawartość `ListBox1` i przystępujemy do odczytania listy plików, katalogów i dysków przy użyciu odpowiednich metod statycznych klasy `Directory`. Wreszcie, po posortowaniu, umieszczamy je w `ListBox1` za pomocą jego metody `ListBox1.Items.AddRange`, uwzględniając wartość odpowiednich pól z grupy `uwzglednij...`. Aby skorzystać z metody `AddRange`, rzutujemy dynamiczną tablicę łańcuchów na tablicę obiektów (operator `as`). Korzystamy w tym celu ze zdefiniowanego w nagłówku metody typu `TablicaObiektow`. Zasadnicze znaczenie ma w powyższym kodzie klasa `Directory`. Poza wykorzystanym przez nas odczytywaniem zawartości wskazanego katalogu udostępnia ona także statyczne metody, pozwalające na manipulowanie katalogami (tworzenie, kasowanie).

Dodając wywołanie metody do konstruktora musimy zwrócić uwagę na to, aby znalazło się ono za wywołaniem metody `InitializeComponent` inicjującej umieszczone w trakcie projektowania komponenty. Nie musimy inicjować pola `sciezkaKatalogu` łańcuchem odpowiadającym konkretnemu katalogowi, bo gdy metoda `PobierzZawartoscKatalogu` wykryje, że jest ono niezainicjowane, sama zapisze do niej ścieżkę bieżącego katalogu.

Projekt 125. Z pełnej ścieżki dostępu do plików wyłaniamy samą nazwę plików i katalogów

Jak zwykle przy pierwszej kompilacji kodu nie wszystko działa jak należy (o ile w ogóle się kompiluje i uruchamia). Aplikacja Czytelnika powinna wyglądać jak na rysunku 7.1, co oznacza, że również u Czytelnika komponent prezentuje pliki z pełną ścieżką dostępu. Musimy zatem zmodyfikować tę metodę w taki sposób, aby widoczna była tylko sama nazwa pliku lub katalogu.



Rysunek 7.1. *Jak widać pierwsza wersja komponentu nie działa w pełni zgodnie z naszymi oczekiwaniami*

Inny sposób wyodrębnienia nazwy pliku za pomocą klasy `FileInfo` opisany został w projekcie 102.

1. Przystępujemy do edycji modułu komponentu (zakładka *FileListBox*).
2. Na każdą pozycję z list `listaPlikow` i `listaKatalogow` działamy metodą `Path.GetFileName` przed dodaniem jej do `ListBox1`. Nazwy katalogów uzupełniamy nawiasami kwadratowymi, a symbole dysków — nawiasami utworzonymi ze znaków < i > (listing 7.6):

Listing 7.6. *Poprawione serce komponentu*

```
procedure TFileListBox.PobierzZawartoscKatalogu;
type TablicaObiektow = array of TObject;
var
  i :Integer;
  sciezka :String;
begin
if sciezkaKatalogu=nil then sciezkaKatalogu:=Directory.GetCurrentDirectory;

pokazujDwieKropki:=(sciezkaKatalogu<>Path.GetPathRoot(sciezkaKatalogu))
                   and uwzglednijKatalogNadrzedny;

if (not Directory.Exists(sciezkaKatalogu)) then
         raise Exception.Create('Katalog '+sciezkaKatalogu+' nie istnieje!');

ListBox1.Items.Clear;

if uwzglednijKatalogi then
         begin
         if pokazujDwieKropki then ListBox1.Items.Add('[..]');
         listaKatalogow:=Directory.GetDirectories(sciezkaKatalogu);
         System.Array.Sort(listaKatalogow);
         for sciezka in listaKatalogow do
            listBox1.Items.Add('['+Path.GetFileName(sciezka)+']');
         end;

if uwzglednijPliki then
```

```
        begin
        if (filtr<>nil) then
          listaPlikow:=Directory.GetFiles(sciezkaKatalogu,filtr)
          else
          listaPlikow:=Directory.GetFiles(sciezkaKatalogu);
        System.Array.Sort(listaPlikow);
        for sciezka in listaPlikow do
          listBox1.Items.Add(Path.GetFileName(sciezka));
        end;

if uwzglednijDyski then
        begin
        listaDyskow:=Directory.GetLogicalDrives();
        for sciezka in listaDyskow do
           listBox1.Items.Add('<'+sciezka.Substring(0,2)+'>');
        end;
end;
```

Metodę `Items.AddRange`, dodającą całą tablicę do `ListBox1`, zastąpiliśmy wywoływaną w pętli metodą `Items.Add`, dodającą tylko jedną, już zmodyfikowaną pozycję. Poza tym podkreśliliśmy różnice między plikami a podkatalogami znajdującymi się w prezentowanym katalogu przez otoczenie nazw tych ostatnich nawiasami kwadratowymi. Natomiast w przypadku dysków usunęliśmy znak ukośnika (*slash*) i pozostawiliśmy jedynie symbol dysku typu „C:", który otoczyliśmy znakami < i >.

Sama pętla użyta w powyższym kodzie jest również interesująca. Zastosowaliśmy konstrukcję

```
for sciezka in listaPlikow do
  listBox1.Items.Add(Path.GetFileName(sciezka));
```

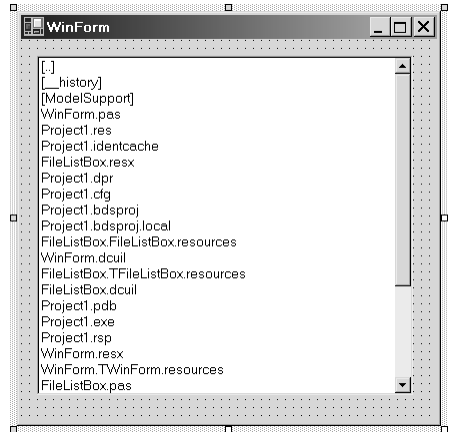
Konieczne jest oczywiście zadeklarowanie zmiennej `sciezka` typu `String`, która będzie reprezentowała bieżący element tablicy lub kolekcji wskazanej za słowem kluczowym `in`. Jest to konstrukcja analogiczna do znanej z C# `foreach` o identycznym działaniu jak zwykła pętla `for` następującej postaci:

```
for i:=0 to Length(listaPlikow)-1 do
  listBox1.Items.Add(Path.GetFileName(listaPlikow[i]));}
```

Przejdźmy na zakładkę pliku *WinForm.pas*. W widoku projektowania formy, na której umieszczony został komponent, powinna być widoczna zawartość katalogu projektu. Możemy skontrolować, czy zmiany w kodzie odniosły zamierzony skutek (wcześniej należy koniecznie kod skompilować). Po tych zmianach komponent powinien prezentować się znacznie lepiej (rysunek 7.2).

Projekt 126. Uwzględnić filtrowanie plików z maską określoną przez pole filtr

Wzorem analogicznego komponentu VCL dodamy możliwość filtrowania prezentowanej listy plików zgodnie z ustaloną przez użytkownika maską. Wystarczy do kodu metody `PobierzZawartoscKatalogu` dodać jedną linię, która w przypadku gdy referencja



Rysunek 7.2. *Podgląd formy w widoku projektowania*

`filtr` nie jest równa `nil` (nie jest pusta), wykorzystuje ją jako drugi argument przeciążonej metody `Directory.GetFiles` (listing 7.7)

Listing 7.7. *Uwzględniamy możliwość określenia maski plików w metodzie PobierzZawartoscKatalogu*

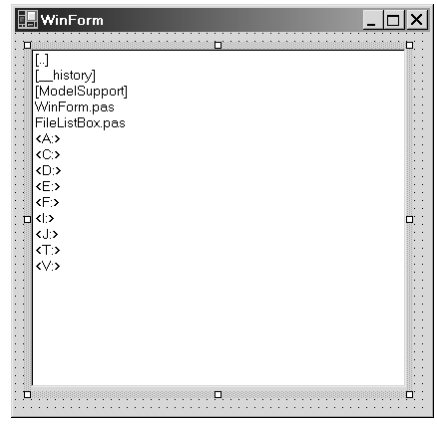
```
if uwzglednijPliki then
  begin
  if (filtr<>nil) then
    listaPlikow:=Directory.GetFiles(sciezkaKatalogu,filtr)
    else
  listaPlikow:=Directory.GetFiles(sciezkaKatalogu);
  System.Array.Sort(listaPlikow);
  for i:=0 to Length(listaPlikow)-1 do
  listBox1.Items.Add(Path.GetFileName(listaPlikow[i]));
  end;
```

Aby przetestować działanie filtru:

1. zmieniamy na chwilę polecenie inicjacji pola `filtr` w konstruktorze na: `filtr:='*.pas';`,
2. kompilujemy kod projektu i komponentu naciskając klawisze *Ctrl+F9*,
3. przechodzimy do widoku projektowania formy *WinForm*.

Wykorzystaliśmy wersję przeciążonej metody `Directory.GetFiles`, w której drugi argument jest maską plików zwracanych w tablicy łańcuchów przez tę funkcję. Efekt widoczny jest na rysunku 7.3. Podobnie przeciążona jest metoda `Directory.GetDirectories`. Po sprawdzeniu działania filtra przywracamy pierwotną postać inicjującego go polecenia, a mianowicie `filtr:=nil;`.

Wykonajmy jeszcze jeden test. Zmieńmy podaną w listingu 7.2 ścieżkę tak, żeby wskazywała na katalog główny jakiegoś dysku w celu upewnienia się, że w takiej sytuacji nie są pokazywane dwie kropki symbolizujące katalog nadrzędny. Warto również przetestować pola `uwzglednijKatalogi`, `uwzglednijPliki` i `uwzglednijDyski`.



Rysunek 7.3.
*Efekt działania filtra *.pas*