abc

14

PROJEKTOWANIE WŁASNEGO KOMPONENTU VCL/VCL.NET

Podejmowanie decyzji o tworzeniu komponentu

Biblioteka VCL posiada bogaty zestaw komponentów. Poza standardowymi komponentami, które starałem się przedstawić w drugiej części książki, jest jeszcze mnóstwo komponentów wyspecjalizowanych, w szczególności służących do projektowania aplikacji bazodanowych. To jest jednak nic w porównaniu z ilością komponentów, jaka dostępna jest w sieci. Warto zajrzeć na strony Torry (*http://torry.net/*), Borland Code Central (*http://cc.borland.com/*) i DSP (*http://delphi.icm.edu.pl/*), aby przekonać się, jak bogata może być inwencja programistów. Mimo to, gdy szukamy komponentu, który dokładnie pasowałby do naszych

potrzeb i spełniał wszystkie nasze wymagania, często okazuje się, że nie możemy takiego znaleźć. W takiej sytuacji możemy przemyśleć możliwość zaprojektowania własnego komponentu.

Kiedy warto projektować własny komponent?

Jednak zanim przystąpimy do napisania komponentu, trzeba sobie odpowiedzieć na pytanie, czy warto. Jeżeli komponent będzie potrzebny tylko raz, to lepiej zbudować potrzebną konstrukcję z kilku komponentów bezpośrednio w klasie formy. Naprawdę szkoda wówczas zachodu na zabawę w budowanie kontrolki. Jeżeli jednak chcemy komponent wykorzystać wiele razy lub udostępnić go innym programistom, warto się postarać i napisać go w miarę ogólnie — koniec końców to się opłaci.

Komponenty są doskonałą realizacją idei wielokrotnego korzystania z raz przygotowanego kodu; tworzą „klocki", z których za pomocą myszy możemy w szybki i całkiem przyjemny sposób stworzyć interfejs aplikacji. Ponadto ich modularność i ukrycie szczegółów implementacji sprawiają, że są optymalną formą dzielenia się kodem.

Jaki komponent my przygotujemy?

Jako wprowadzenie do projektowania komponentów proponuję przygotować prosty komponent, wyglądający jak etykieta `TLabel`, ale którego kliknięcie spowoduje otwarcie przeglądarki i załadowanie strony wskazanej przez własność `Adres` tego komponentu. Komponent nazwiemy `TLinkLabel` i będzie on bezpośrednim rozszerzeniem klasy `TCustomLabel`. Jest to klasa o niemal identycznych własnościach i zdarzeniach jak `TLabel`, ale większość jej własności jest chroniona, a nie opublikowana. Zalety wybrania tej klasy bazowej wyjaśnią się za chwilę.

Tworząc komponent ograniczymy się do rzeczy podstawowych. Jego prawdziwym celem nie jest tak bowiem przygotowanie komponentu użytkowego, a nauka pisania komponentów. Nie będziemy się wobec tego starać pisać go możliwie jak najbardziej ogólnie i elastycznie, a wręcz przeciwnie — postaram się, żeby jego kod był jak najbardziej zwarty i przejrzysty.

Tworzenie i testowanie komponentu

Ja zwykle projektuję komponent w obrębie projektu zwykłej aplikacji np. typu *VCL Forms*. Dzięki temu dysponuję wygodnym środowiskiem jego testowania. Dopiero gdy uznam, że komponent jest gotowy, tworzę dla niego osobny projekt. Proponuję też tak zrobić tym razem.

Tworzenie modułu komponentu

Stwórzmy projekt *VCL Forms Application — Delphi for Win32* lub *VCL Forms Application — Delphi for .NET* i zapiszmy go do osobnego katalogu na dysku.

Następnie z menu *Component* wybieramy pozycję *New VCL Component...*. Pojawi się okno o tej samej nazwie, widoczne na rysunku 14.1. Jest to kreator komponentu. W pierwszym kroku musimy wybrać klasę bazową nowego komponentu (rysunek 14.1, na górze). Proponuję wskazać klasę `TCustomLabel`. Klikamy przycisk *Next*. W kolejnym kroku wybieramy nazwę klasy komponentu (pole *Class Name*) i domyślną paletę (pole *Palete Page*), na której chcemy go umieścić (rysunek 14.1, na dole). Proponuję nadać mu nazwę `TLinkLabel` i przypisać go do palety *Helion*. Ze ścieżki przeszukiwania w polu *Search Path* można usunąć katalog związany z Indy, ale ja nie zaprzątałbym tym sobie głowy. Nie warto również zmieniać nazwy pliku, w którym umieszczony zostanie komponent (pole *Unit name*). O tym zdecydujemy ostatecznie i tak dopiero przy zapisywaniu go na dysku. Ponownie klikamy przycisk *Next* i na ostatniej stronie klikamy przycisk *Finish*. W efekcie do projektu dodany zostanie nowy moduł o nazwie *LinkLabel*. Należy go jeszcze zapisać na dysku, naciskając *Ctrl+S*. Pojawi się okno dialogowe, w którym należy odszukać katalog, w którym wcześniej zapisaliśmy projekt i zapisać nowy moduł, np. pod nazwą *LinkLabel.pas*.

Procedura Register

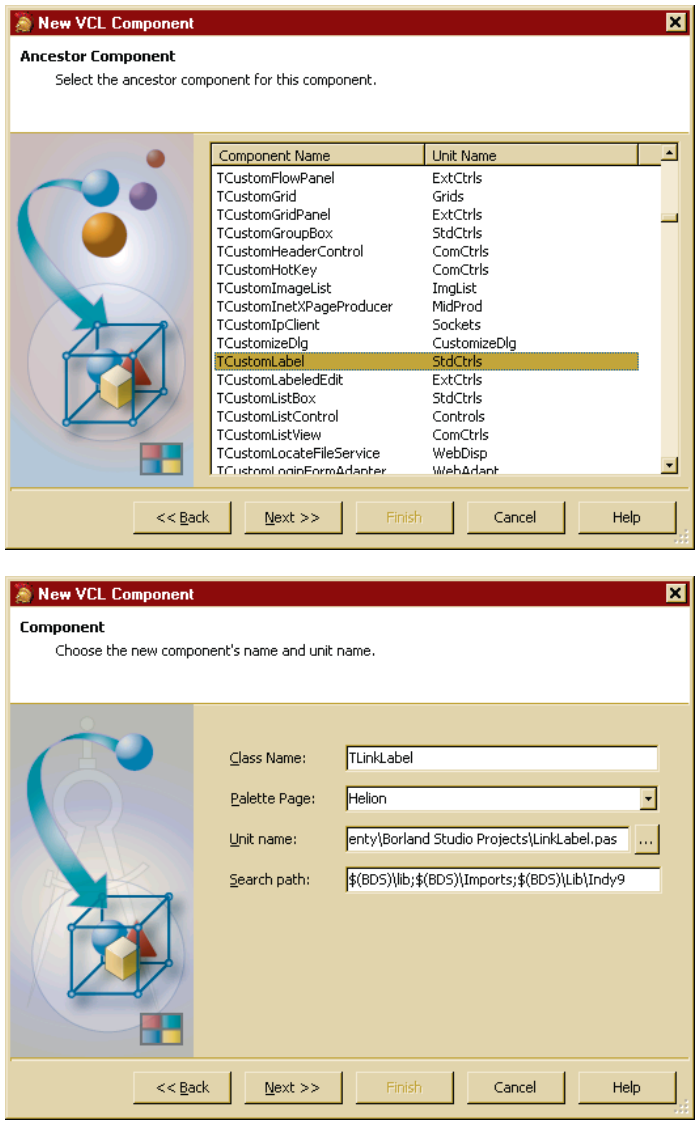
Moduł został automatycznie wyposażony w procedurę `Register`, która służy do rejestracji komponentu (listing 14.1). Funkcja ta wywołuje procedurę `RegisterComponents`, której pierwszym argumentem jest nazwa palety, na której mają być zainstalowane komponenty, a drugim — zbiór ich klas.

Listing 14.1. Utworzona przez Delphi procedura rejestrująca komponent w środowisku BDS

```
procedure Register;
begin
  RegisterComponents('Helion', [TLink]);
end;
```

Metoda testująca komponent

Przechodzimy do modułu formy naszego projektu, czyli do modułu *Unit1.pas* dostępnego na zakładce *Unit1*. Do sekcji uses interfejsu dodajemy moduł komponentu `LinkLabel`. Następnie na formie umieszczamy przycisk. Chciałbym, aby



Rysunek 14.1. Kreator komponentu VCL

jego kliknięcie powodowało utworzenie komponentu `TLinkLabel`. Musimy zatem wyposażyć ów przycisk w metodę zdarzeniową związaną ze zdarzeniem `OnClick`, która stworzy komponent i umieści go na formie[1] (listing 14.2). W ten sposób będziemy mogli testować komponent, zanim zainstalujemy go w środowisku BDS.

[1] Tworzenie obiektów w trakcie działania aplikacji omówione zostało w rozdziale 8.

Listing 14.2. Tworzenie komponentu z poziomu kodu

```
procedure TForm1.Button1Click(Sender: TObject);
var Link :TLinkLabel;
begin
Link:=TLinkLabel.Create(Form1);
Link.Caption:='Wydawnictwo Helion';
Link.Left:=100;
Link.Top:=100;
Link.Parent:=Form1;
end;
```

Plik *LinkLabel.pas* nie jest formalnie rzecz biorąc jednym z plików projektów — nie jest widoczny w oknie projektu. Ale ponieważ zapisaliśmy go w katalogu projektu, jego zawartość, w szczególności klasa `TLinkLabel`, będzie widoczna w naszym projekcie. Czytelnik może oczywiście bez problemu dodać moduł do projektu, korzystając z menu *Project/Add to Project...* lub klawiszy *Shift+F11*.

Dodawanie metod do komponentu

Wróćmy do modułu *LinkLabel* i dodajmy do klasy `TLinkLabel` metodę o nazwie `SprawdzAdres`, której zadaniem będzie sprawdzenie poprawności adresu URL. Po przejściu na zakładkę *LinkLabel* dodajemy do klasy w sekcji interfejsu deklarację tej metody (listing 14.3). Argumentem funkcji będzie łańcuch zawierający adres URL. Funkcja będzie zwracać wartość logiczną, która poinformuje, czy łańcuch zawiera poprawny adres sieciowy. Teraz musimy ją zdefiniować. Jej definicja widoczna jest również na listingu 14.3; musi oczywiście być umieszczona w sekcji implementacji. Jej działanie jest proste — sprawdza tylko, czy na początku metody znajduje się nazwa protokołu, a dokładnie łańcuch *http://*. Jeżeli tak, to metoda uznaje, że wszystko jest w porządku, i zwraca wartość `True`. W przeciwnym razie zwracane jest `False`.

Listing 14.3. Pełen listing modułu po dodaniu metody SprawdzAdres

```
unit LinkLabel;

interface

uses
  SysUtils, Classes, Controls, StdCtrls;
```

```
type
  TLinkLabel = class(TCustomLabel)
  private
    { Private declarations }
  protected
    { Protected declarations }
    function SprawdzAdres(AAdres :String) :Boolean;
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

function TLinkLabel.SprawdzAdres(AAdres :String) :Boolean;
begin
Result:=(Copy(AAdres,1,7)='http://');
end;

procedure Register;
begin
  RegisterComponents('Helion', [TLinkLabel]);
end;

end.
```

Jak działa dodana metoda? Wywołana w niej funkcja `Copy` zwraca fragment podanego jako pierwszy argument łańcucha, od pozycji wskazanej w drugim argumencie i o długości wskazanej w trzecim. My porównujemy pierwsze siedem znaków z ciągiem *http://*. Jeżeli oba łańcuchy są identyczne, to wartość zwracana przez operator porównania = równa jest `True`. W przeciwnym wypadku jest nią `False`. Każda z tych wartości przypisywana jest do `Result`, która reprezentuje wartość zwracaną przez funkcję. Identyczny efekt, ale w mniej elegancki sposób, można byłoby uzyskać, korzystając z instrukcji `if` (listing 14.4).

Listing 14.4. Dłuższa wersja funkcji SprawdzAdres

```
function TLinkLabel.SprawdzAdres(AAdres :String) :Boolean;
begin
if Copy(AAdres,1,7)='http://'
  then Result:=True
  else Result:=False;
end;
```

Krótka uwaga na temat metod statycznych

Zauważmy jeszcze jeden fakt. Metoda `SprawdzAdres` w żaden sposób nie odnosi się do pól lub metod klasy `TLinkLabel`. Jest zupełnie autonomiczna — potrzebne dane przekazywane są do niej przez argument, a wynik zwracany jest przez wartość. Jej działanie nie zależy od stanu obiektu (tzn. od aktualnej wartości jego pól). Jest to zatem dobra kandydatka na metodę statyczną[2], czyli taką, która może być uruchamiana bez tworzenia instancji klasy (bez tworzenia obiektu). Aby zmienić metodę `SprawdzAdres` w metodę statyczną, należy na początku jej sygnatury, zarówno w deklaracji w klasie, jak i w definicji w sekcji implementacji, dodać słowo kluczowe `class`:

```
class function SprawdzAdres(AAdres :String) :Boolean;
```

Metody takie wywołuje się w podobny sposób jak konstruktor podając przed jej nazwą nazwę klasy. Problem polega jednak na tym, że nie mamy zamiaru udostępniać tej metody (dopisaliśmy ją do sekcji `protected`), więc uczynienie jej metodą statyczną dałoby niewiele korzyści, bo i tak nie byłoby do niej dostępu.

Konstruktor komponentu

Zajmijmy się teraz możliwością inicjacji komponentu. Służy do tego konstruktor, który musi zostać dodany do klasy komponentu. Zgodnie z konwencją Delphi nazwiemy go `Create`. Konieczna jest oczywiście jego deklaracja w klasie i potem implementacja. Zmiany w module wyróżnione są na listingu 14.5. Niezbędne jest również dodanie do sekcji `uses` modułu *Graphics*, który jest wykorzystywany w konstruktorze.

Listing 14.5. Kod modułu LinkLabel po dodaniu konstruktora komponentu

```
unit LinkLabel;

interface

uses
  SysUtils, Classes, Controls, StdCtrls, Graphics;

type
  TLinkLabel = class(TCustomLabel)
  private
    { Private declarations }
```

[2] Nazywaną w Pascalu też metodą klasy. Jak zwróciłem już uwagę w rozdziale piątym, wydaje mi się to określeniem mylącym — wszystkie metody są w końcu metodami klas w szerszym rozumieniu tego terminu.

```
  protected
    { Protected declarations }
    function SprawdzAdres(AAdres :String) :Boolean;
  public
    { Public declarations }
    constructor Create(Owner :TComponent); override;
  published
    { Published declarations }
  end;

procedure Register;

implementation

constructor TLinkLabel.Create(Owner :TComponent);
var LinkStyle :TFontStyles;    //zbior
begin
   inherited Create(Owner);
   LinkStyle:=[fsUnderline];
   Font.Color:=clNavy;
   Font.Style:=LinkStyle;
   Cursor:=crHandPoint;
end;

function TLinkLabel.SprawdzAdres(AAdres :String) :Boolean;
begin
Result:=(Copy(AAdres,1,7)='http://');
end;

procedure Register;
begin
  RegisterComponents('Helion', [TLinkLabel]);
end;

end.
```

W deklaracji konstruktora widoczny jest modyfikator `override` (z ang. nadpisany). Dodaliśmy go, ponieważ konstruktor naszej klasy bazowej `TCustomLabel` zadeklarowany jest jako wirtualny. Nie chciałbym głębiej omawiać tego zagadnienia[3], ale warto zapamiętać, że jeżeli tego nie zrobimy — wirtualny konstruktor klasy bazowej zostanie ukryty (przy kompilacji pojawi się wówczas ostrzeżenie z tym związane). Jeżeli definiujemy konstruktor komponentu, to należy pamiętać o wywołaniu konstruktora klasy bazowej. To ważne. Jeżeli o tym zapomnimy, nasz komponent może nie działać właściwie, co w praktyce oznacza, że próba stworzenia jego instancji skończy się tylko zgłoszeniem wyjątku. Po wywołaniu konstruktora klasy bazowej ustalany jest wygląd naszego komponentu. Zmieniamy

[3] Nieco więcej wiadomości na ten temat znajduje się na końcu rozdziału w zadaniu domowym.

jego styl tak, żeby był podkreślony, zmieniamy jego kolor na granatowy i kształt kursora na „rękę". Teraz możemy skompilować projekt i przetestować nowy wygląd komponentu ,klikając przycisk na formie (rysunek 14.2).

Rysunek 14.2. Testowanie komponentu po przygotowaniu konstruktora

Własności komponentu

Aby użytkownik komponentu mógł wskazać adres strony WWW, z którą mamy się połączyć, dodamy do niego własność `Adres` typu `String`. Własności powinno towarzyszyć prywatne pole `FAdres`, które będzie przechowywać jej wartość. Przy odczycie adresu z własności `Adres` będzie ona wprost czytana z pola `FAdres`, ale przy próbie zapisu do własności sprawdzana będzie jego poprawność. Jeżeli użytkownik poda adres w stylu *www.helion.pl*, zostanie on automatycznie uzupełniony o nazwę protokołu.

A więc do dzieła! Zacznijmy od zdefiniowania samej własności. Służy do tego konstrukcja, której szablon może być następujący:

```
property Nazwa :Typ read FNazwa write SetNazwa;
```

lub

```
property Nazwa :Typ read GetNazwa write SetNazwa;
```

Te dwie definicje różnią się sposobem odczytywania wartości własności. W pierwszej odczytywana jest z pola o tej samej nazwie co własność, ale z przedrostkiem `F`. W drugim za pośrednictwem metody, której zwykle nadaje się nazwę rozpoczynająca się od `Get`. Podobnie można zrobić przy przypisywaniu wartości do własności — można ją przekazać bezpośrednio do pola lub za pośrednictwem metody sprawdzającej nową wartość.

W naszym przypadku deklaracja będzie miała postać:

```
property Adres :String read FAdres write SetAdres;
```

co oznacza, że powstanie własność Adres typu String, której wartość odczytywana jest z pola FAdres. Próba przypisania własności jakiejś wartości spowoduje wywołanie metody SetAdres, której argumentem jest łańcuch i do której przekazana będzie przypisywana własności wartość. Jak wspomniałem, zamiast odczytywać wartość wprost z pola, możemy też wywołać metodę. Musi to być jednak metoda, która zwraca typ String i nie pobiera żadnych argumentów. W tej metodzie możemy na przykład sprawdzić, czy zwracana wartość jest sensowna. My jednak będziemy to robić już w momencie przypisania wartości.

Oczywiście użycie jako nazwy pola FAdres, a jako nazwy metody SetAdres jest efektem konwencji nazw, a nie koniecznością. Jednak ta konwencja ułatwia zorientowanie się w kodzie komponentu i właściwe przypisanie własności do pól i metod.

Cały moduł komponentu wraz ze zdefiniowanym polem, metodą i konstrukcją definiującą własność jest widoczny na listingu 14.6.

Listing 14.6. Własność Adres w komponenie TLinkLabel

```
unit LinkLabel;

interface

uses
  SysUtils, Classes, Controls, StdCtrls, Graphics;

type
  TLinkLabel = class(TCustomLabel)
  private
    { Private declarations }
    FAdres :String;
  protected
    { Protected declarations }
    function SprawdzAdres(AAdres :String) :Boolean;
    procedure SetAdres(AAdres :String);
  public
    { Public declarations }
    constructor Create(Owner :TComponent); override;
  published
    { Published declarations }
    property Adres :String read FAdres write SetAdres;
  end;

procedure Register;

implementation
```

```
constructor TLinkLabel.Create(Owner :TComponent);
var LinkStyle :TFontStyles;    //zbior
begin
   inherited Create(Owner);
   LinkStyle:=[fsUnderline];
   Font.Color:=clNavy;
   Font.Style:=LinkStyle;
   Cursor:=crHandPoint;
end;

procedure TLinkLabel.SetAdres(AAdres :String);
begin
if not SprawdzAdres(AAdres) then AAdres:='http://'+AAdres;
FAdres:=AAdres;
end;

function TLinkLabel.SprawdzAdres(AAdres :String) :Boolean;
begin
Result:=(Copy(AAdres,1,7)='http://');
end;

procedure Register;
begin
  RegisterComponents('Helion', [TLinkLabel]);
end;

end.
```

Zalety własności

Korzystając do tej pory z własności komponentów, np. zmieniając etykietę przycisku (własność `Caption`) w inspektorze obiektów, traktowaliśmy te własności jak udostępnione w inspektorze pola. Własności są jednak czymś znacznie więcej. Można powiedzieć, że własności są konstrukcją — pomostem łączącym metody uruchamiane przy odczycie i przypisaniu wartości własności z prywatnym polem, które tę wartość przechowuje. Daje to programiście potężne możliwości. Przede wszystkim może on już w momencie przypisywania sprawdzić, czy podana wartość jest prawidłowa. Jeżeli nie, może ją poprawić samodzielnie lub zgłosić taką potrzebę użytkownikowi. W przypadku pól takich możliwości nie ma. Co więcej, zmiana wartości własności komponentu może wiązać się z automatyczną aktualizacją innych własności komponentu. Wyobraźmy sobie na przykład komponent implementujący koło. Jeżeli zmienimy jego własność odpowiadającą za promień, to metoda przypisująca wartość może automatycznie zaktualizować wartość obwodu i pola tego koła. Takich możliwości także nie ma w przypadku pól.

Przypomnijmy sobie ponadto rozterki, jakie mieliśmy w trakcie projektowania klasy `TRownanieKwadratowe` w rozdziale piątym. Z wieloma z nich można sobie poradzić, korzystając z własności (zob. zadanie na końcu rozdziału).

Testowanie własności

Aby przetestować dodaną do komponentu `TLinkLabel` własność `Adres` dodamy do metody testującej w module *Unit1* polecenie przypisania jakiejś wartości to tej własności np. *www.helion.pl*. Przejdźmy zatem do modułu *Unit1* i uzupełnijmy metodę `Button1Click` zgodnie ze wzorem z listingu 14.7.

Listing 14.7. Testowanie nowej własności komponentu

```
procedure TForm1.Button1Click(Sender: TObject);
var Link :TLinkLabel;
begin
Link:=TLinkLabel.Create(Form1);
Link.Caption:='Wydawnictwo Helion';
Link.Left:=100;
Link.Top:=100;
Link.Adres:='www.helion.pl';
Link.Parent:=Form1;
end;
```

Metoda prawie zdarzeniowa

Za chwilę do klasy dodamy jeszcze jedną metodę. Jej sygnatura będzie następująca:

```
procedure Polacz(Sender :TObject);
```

Jej sygnatura powinna nam coś przypominać. Tak jest! Wygląda jak metoda zdarzeniowa. I tak ją później wykorzystamy — połączymy ją bowiem ze zdarzeniem `OnClick` komponentu. Jednak na razie jest zwykłą metodą, przyjmującą za argument referencję do typu `TObject`.

Umieśćmy powyższą sygnaturę w sekcji `public` klasy komponentu, a do sekcji `implementation` dodajmy definicję widoczną na listingu 14.8.

Listing 14.8. Implementacja metody Polacz dla obu platform

```
procedure TLinkLabel.Polacz(Sender :TObject);
begin
Font.Color:=clBlack;
{$IF NOT DEFINED(CLR)}
ShellExecute(0,'open',PCHar(Adres),'','',SW_NORMAL);
{$ELSE}
ShellExecute(0,'open',Adres,'','',SW_NORMAL);
{$IFEND}
end;
```

Powyższa metoda korzysta z funkcji WinAPI `ShellExecute`, która zadeklarowana jest w module *ShellApi*. Ze względu na używane w argumentach łańcuchy jej definicja różni się w zależności od platformy, dla której przeznaczony jest projekt. Stąd konieczność kompilacji warunkowej. Użyta jako jej argument stała `SW_NORMAL` zdefiniowana jest w module *Windows*. Z tego wynika, że do sekcji `uses` musimy dodać dwa moduły:

```
uses
  SysUtils, Classes, Controls, StdCtrls, Graphics, Windows, ShellApi;
```

Jeszcze raz o łańcuchach, WinAPI i platformie .NET

Jak zwykle w przypadku funkcji WinAPI pojawia się problem łańcuchów. W oryginalnej sygnaturze tej funkcji, jaką można znaleźć w dokumentacji WinAPI, jej drugi, trzeci, czwarty i piąty argument są tablicami znaków, a więc typem łańcuchów, jaki używany jest w C i C++. W Delphi dla platformy Win32 nie ma z tym oczywiście żadnego problemu. Wystarczy łańcuch przechowywany w `String` zrzutować na typ `PChar`, który jest wskaźnikiem do znaku, a więc dokładnie takim samym typem, jak wskaźnik do tablicy znaków żądany przez `ShellExecute`. Jednak w platformie .NET, choć takie rzutowanie też jest możliwe, to nie jest mile widziane. Korzystając ze wskaźników, musielibyśmy zadeklarować cały moduł jako niebezpieczny (*unsafe*). Żeby tego uniknąć, w wersji dla .NET funkcja `ShellExecute` zadeklarowana została w taki sposób, że jej argumentami są po prostu łańcuchy typu `String`. Świetnie. Ale niestety wiąże się to z odmiennym przygotowywaniem kodu dla platformy Win32 i platformy .NET. Aby uniknąć tworzenia osobnych modułów, skorzystaliśmy z kompilacji warunkowej (więcej informacji o niej i powyższym problemie w rozdziale 3.).

Funkcja ShellExecute

Co robi `ShellExecute`? Jest to funkcja bibliotek WinAPI, a dokładnie jej części nazywanej warstwą powłoki. Związana jest tak naprawdę z Exploratorem Windows. Jest to funkcja wykorzystywana przez system w momencie, gdy w Eksploratorze klikniemy dwukrotnie jakiś plik. Funkcja ta sprawdzi, jakie jest rozszerzenie pliku. Jeżeli jest nim *.exe* lub *.bat*, to zostanie wykonany. Jeżeli jest to jedno z zarejestrowanych w systemie rozszerzeń dokumentów, to jego edytor zostanie uruchomiony i kliknięty plik będzie do niego wczytany. Funkcja `ShellExecute` jest jednak jeszcze „mądrzejsza". Sprawdza bowiem nie tylko rozszerzenie pliku, ale również podany na początku protokół. Domyślnym jest protokół *file://*. Wskazuje on, że adres podany w argumencie jest ścieżką dostępu do lokalnego pliku. Inny ciekawy protokół to *mailto:*. Jeżeli jako argument funkcji podamy *mailto:jacek@fizyka.umk.pl*, funkcja wywoła okno kompozycji listu e-mail domyślnego klienta

pocztowego. Następnym protokołem, który teraz interesuje nas w sposób szczególny, jest *http://*. Wskazuje on, że łańcuch zawiera adres strony WWW. W takiej właśnie postaci my przekazujemy adres do tej funkcji. W efekcie spowoduje to uruchomienie domyślnej przeglądarki i próbę wczytania podanego adresu WWW.

Uzupełnianie konstruktora

Zanim przetestujemy nową metodę, proponuję uzupełnić konstruktor o polecenia, które pozwolą na łatwiejsze wywołanie metody `Polacz`. Związemy ją ze zdarzeniem `OnClick` naszego komponentu. Dzięki temu kliknięcie komponentu spowoduje otworzenie przeglądarki i próbę wczytania strony WWW. Listing 14.9 zawiera odpowiednie modyfikacje. W konstruktorze ustalamy też domyślną wartość własności `Adres`.

Listing 14.9. Domyślna wartość własności Adres i przypisanie metody Polacz do zdarzenia OnClick

```
constructor TLinkLabel.Create(Owner :TComponent);
var LinkStyle :TFontStyles;    //zbior
begin
   inherited Create(Owner);
   LinkStyle:=[fsUnderline];
   Font.Color:=clNavy;
   Font.Style:=LinkStyle;
   Cursor:=crHandPoint;
   Adres:='http://helion.pl';
   OnClick:=Polacz;
end;
```

Teraz możemy skompilować nasz projekt i uruchomić go. Po kliknięciu przycisku powstanie komponent `TLabelLink`. Jeżeli go klikniemy, otworzona powinna zostać przeglądarka, w której zobaczymy stronę WWW wskazaną przez własność `Adres` komponentu.

Na niektórych komputerach otwarcie strony może być utrudnione przez zaporę sieciową, która blokuje dostęp do sieci przeglądarek uruchamianych przez aplikacje.

Zmienne proceduralne a zdarzenia

Przypisanie łańcucha do własności `Adres` to nic wielkiego, choć zdawać sobie powinniśmy sprawę, że wywoływana jest wówczas metoda `SetAdres`. Większe zdziwienie może wywoływać polecenie przypisujące metodę `Polacz` do zdarzenia

OnClick, a dokładnie prostota tego polecenia. Wygląda to bowiem tak, jakbyśmy do zmiennej OnClick przypisywali wartość Polacz. Czym jest OnClick? Jak wiemy, jest to **zdarzenie**. No tak, ale czym w takim razie są zdarzenia? Okazuje się, że są to zmienne proceduralne (wspomniałem o nich w rozdziale 4.), czyli zmienne, które mogą przechowywać odwołania do funkcji i procedur. Po przypisaniu do zdarzenia OnClick odwołania do metody Polacz użycie instrukcji OnClick(nil); da ten sam efekt, jak Polacz(nil);. Zupełnie to samo dzieje się, gdy ze zdarzeniem wiążemy metodę za pomocą inspektora obiektów. Zdarzenie, czyli zmienna proceduralna przechowuje wówczas odwołanie do metody wskazanej przez nas metody zdarzeniowej, która poprzez tą zmienną proceduralną może być łatwo przez komponent uruchomiona w momencie wystąpienia zdarzenia.

Zauważmy jednak, że w odróżnieniu od zmiennych proceduralnych, o których opowiadałem w czwartym rozdziale, zdarzenia przechowują odwołania do metod, a więc do funkcji i procedur składowych obiektu. Jest to zaznaczone w ich definicji przez dodanie do zwykłego typu zmiennej proceduralnej słów of object. Na przykład typem zmiennej OnClick jest TNotifyEvent, który jest zdefiniowany następująco:

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

Udostępnianie niektórych ukrytych własności

Teraz możemy wyjaśnić, dlaczego jako klasy bazowej użyliśmy TCustomLabel, a nie zwykłej klasy TLabel. Te dwie klasy różnią się jednym zasadniczym szczegółem — poziomem dostępu niektórych własności i zdarzeń. W klasie TCustomLabel większość z nich jest chroniona (ich deklaracje znajdują się w sekcji protected), podczas gdy w TLabel zostały one opublikowane, tzn. zadeklarowane ponownie w sekcji published. My zaraz też zrobimy to samo z niektórymi własnościami i zdarzeniami odziedziczonymi z TCustomLabel. Nie uczynimy tego jednak na pewno ze zdarzeniem OnClick. Nie chcielibyśmy przecież, aby użytkownik mógł zmienić metodę przypisaną do tego zdarzenia i w ten sposób zmienić również sposób działania komponentu. I właśnie po to są takie komponenty, jak TCustomLabel, TCustomMemo, TCustomListBox itp. — aby programista miał możliwość pozostawienia niektórych własności i zdarzeń ukrytych. O ile bowiem można publikować elementy składowe klas, to nie można ich już z powrotem ukryć.

Jakie własności i zdarzenia chcemy opublikować? Możemy się na przykład zastanowić, czy udostępnić własność Font. Ustaliśmy bowiem własność czcionki w taki sposób, by wygląd komponentu każdemu internaucie sugerował jego przeznaczenie. Pozostawiając tę własność jako chronioną, uniemożliwilibyśmy użytkownikowi komponentu na zmianę jego wyglądu. To byłoby jednak poważne ograniczenie użyteczności komponentu. Przecież nie zawsze taki wygląd, jaki my ustaliliśmy, musi pasować do interfejsu projektowanego przez innego programistę.

Proponuję zatem udostępnić tę własność. Podobnie proponuję postąpić z własnościami `AutoSize`, `Color` i `PopupMenu` oraz ze zdarzeniami związanymi z mechanizmem *drag & drop*. Powinniśmy również opublikować własność `Caption`. Jest ona co prawda publiczna, ale nie jest opublikowana. Jest zatem dostępna z poziomu kodu, ale nie byłoby jej w inspektorze obiektów komponentu. Użytkownik może, korzystając z dokumentacji klasy `TCustomLabel`, rozważyć opublikowanie innych zdarzeń i własności.

Do definicji klasy w module *LinkLabel* wprowadzamy zmiany wyróżnione w listingu 14.10. Przy publikowaniu własności i zdarzeń nie trzeba ich od nowa definiować. Wystarczy tylko „przypomnienie" ich nazwy umieszczone w sekcji `published`.

Listing 14.10. Definicja klasy komponentu z opublikowanymi własnościami

```
type
  TLinkLabel = class(TCustomLabel)
  private
    { Private declarations }
    FAdres :String;
  protected
    { Protected declarations }
    function SprawdzAdres(AAdres :String) :Boolean;
    procedure SetAdres(AAdres :String);
  public
    { Public declarations }
    constructor Create(Owner :TComponent); override;
    procedure Polacz(Sender :TObject);
  published
    { Published declarations }
    property Adres :String read FAdres write SetAdres;
    //publikowanie wlasnosci
    property Caption;
    property Font;
    property AutoSize;
    property Color;
    property PopupMenu;
    //publikowanie zdarzen
    property OnDragOver;
    property OnDragDrop;
    property OnEndDrag;
  end;
```

Jak widać, zdarzenia definiowane są słowem kluczowym `property`, tak samo jak własności. Nie powinno nas to dziwić, skoro wiemy już, że są one zmiennymi typu proceduralnego, i zapisywana może być do nich metoda zdarzeniowa.

Żeby dowiedzieć się, w jaki sposób zdefiniować własne zdarzenia, odsyłam do książki *Delphi 2005. 303 gotowe rozwiązania* (Wydawnictwo Helion, 2005), gdzie znajduje się szczegółowe wyjaśnienie.

Pakiet dla komponentu i jego instalacja w BDS

Jeżeli jesteśmy przekonani, że nasz komponent jest już gotowy... No dobrze, umówmy się, że nasze przekonanie tak naprawdę niewiele znaczy, bo mogę z góry zapewnić, że w trakcie jego używania szybko okaże się, że do kodu komponentu należy wprowadzić jeszcze wiele poprawek i udoskonaleń. Dlatego, zanim komponent udostępnimy innym w jakiejkolwiek formie, lepiej go pouzywać w swoich projektach przynajmniej przez kilka tygodni.

Ale jeżeli jednak jesteśmy przekonani, że komponent osiągnął już mniej więcej ostateczną postać, możemy przygotować dla niego pakiet. Jest to specjalny typ projektu, który pozwala na kompilację komponentu do postaci, w której może być zainstalowany w środowisku Borland Developer Studio i udostępniony na paletach komponentów.

Ikona komponentu

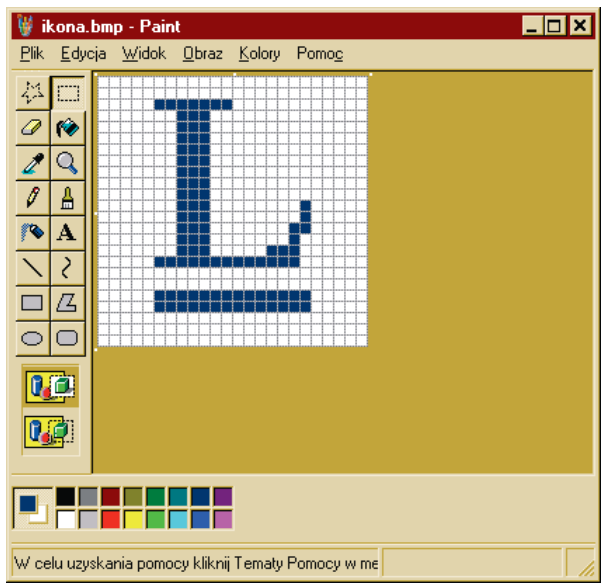
W Borland Developer Studio nie ma już narzędzia *Image Editor*, które pozwalało między innymi na przygotowanie ikon, kursorów i plików zasobów dla projektów Delphi. To nieco komplikuje sposób przygotowania ikony dla naszego komponentu, ale tego nie uniemożliwia. Przede wszystkim można ściągnąć z sieci alternatywny edytor, np. proponowany przez Borland XNResourceEditor[4]. Poniżej przedstawię jednak, jak poradzić sobie z samymi tylko narzędziami BDS 2006. Uprzedzam, że Delphi jest w tym względzie wymagające i literówka w nazwie pliku zasobu lub jego elementu może doprowadzić całą operację do niepowodzenia.

Jeżeli chcemy, żeby nasz komponent był reprezentowany na palecie komponentów przez wybraną przez nas ikonę, musimy przygotować plik zasobów *.res* zawierający obraz typu bitmap, którego identyfikator jest identyczny z nazwą klasy komponentu pisaną wielkimi literami. Ważna jest nazwa obrazu, jakim będzie

[4] Do ściągnięcia ze strony *http://www.wilsonc.demon.co.uk/d10resourceeditor.htm*.

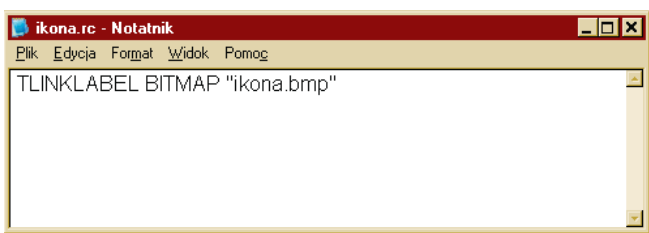
identyfikowany w pliku zasobów. Musi ona odpowiadać nazwie klasy komponentu, ale pisana wyłącznie wielkimi literami. W naszym przypadku będzie to zatem `TLINKLABEL`. Obraz powinien mieć rozmiar 24 na 24 piksele i 16 kolorów.

1. Za pomocą Windowsowej aplikacji *Paint* tworzymy obraz o rozmiarach 24 na 24 piksele i zapisujemy go w pliku *ikona.bmp* jako mapę bitową z szesnastoma kolorami (rysunek 14.3). Plik bitmapy powinien mieć 406 bajtów.



Rysunek 14.3. Paint i notatnik to w niektórych sytuacjach cenne narzędzia każdego programisty

2. Następnie za pomocą systemowego notatnika tworzymy plik *ikona.rc*, zawierający jedną linię, widoczną na rysunku 14.4. Ten plik dołączymy do projektu pakietu.



Rysunek 14.4. Plik .rc można przygotować choćby w notatniku

3. Uruchamiamy konsolę (polecenie `cmd` lub `command`), przechodzimy do katalogu, w którym znajdują się obraz i plik *.rc*, a następnie wydajemy polecenie kompilacji pliku *.rc* do pliku binarnego *.res*: `brc32 –r ikona.rc –foLinkLabel.dcr`. Kompilację umożliwia program *C:\Program Files\Borland\BDS\4.0\Bin\brc32.exe*, jedno z narzędzi znajdujące się w podkatalogu *bin* katalogu, do którego zainstalowany został BDS.

W wyniku kompilacji powstanie plik *LinkLabel.dcr*, który powinien znajdować się w katalogu projektu. Ważna jest jego nazwa — musi być taka sama, jak nazwa modułu, w którym jest komponent, ale z rozszerzeniem *.dcr*, a więc *LinkLabel.dcr*.

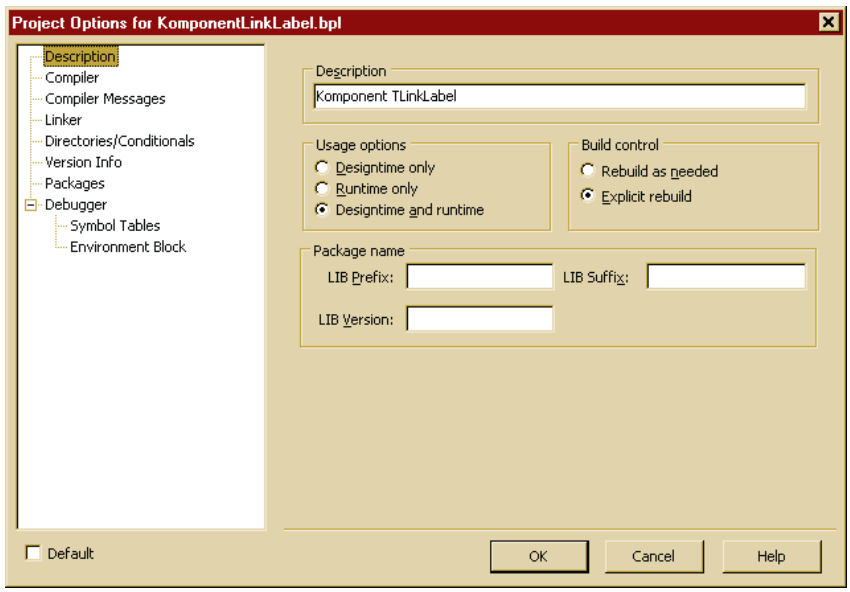
Plik *ikona.rc* dołączony do projektu aplikacji byłby kompilowany automatycznie. Niestety nie robi tego projekt pakietu.

Aby stworzyć projekt pakietu dla Win32

Przejdźmy teraz do utworzenia pakietu. Pakiet może zawierać więcej niż jeden komponent, ale tym razem tej możliwości nie wykorzystamy.

1. Nie zamykając projektu, w którym tworzymy komponent, możemy z menu *File* wybrać podmenu *New*, a w nim *Package — Delphi for Win32*.
2. Nowy projekt zapisujemy pod nazwą *KomponentLinkLabel.bdsproj* w tym samym katalogu, w którym jest plik *LinkLabel.pas*. Nie możemy użyć nazwy *LinkLabel.** dla projektu, aby nie nadpisać plików modułu, w którym zapisany jest komponent.
3. Do projektu pakietu musimy dodać moduł *LinkLabel.pas*. Czynność ta została już dokładnie omówiona w rozdziale 11., gdy do projektu przeglądarki dodawaliśmy moduł *Drukowanie.pas*, ale powtórzmy opis wszystkich czynności jeszcze raz:
   - W oknie projektu klikamy prawym klawiszem myszy na *KomponentLinkLabel.bpl* i z menu kontekstowego wybieramy *Add…*. Jest to alternatywny sposób dodawania plików do projektu, ale całkowicie równoważny poleceniu *Add to Project...* z menu *Project*.
   - W oknie z tytułem *Add* klikamy przycisk *Browse* i za pomocą okna dialogowego wskazujemy plik *LinkLabel.pas*.
   - Po powrocie do okna *Add* klikamy przycisk *OK*. W efekcie nowy plik zostanie dodany do gałęzi *Contains* tego projektu. Powinien być również widoczny plik *LinkLabel.dcr*, zawierający naszą ikonę komponentu.

4. Z menu kontekstowego pozycji *KolorowyPasekPostepu.bpl* w oknie projektu wybieramy *Options…*:
   - Na zakładce *Description* (opis; rysunek 14.5) w polu o tej samej nazwie wpisujemy `Komponent TLinkLabel`; jest to opis pakietu, który po zainstalowaniu komponentu będzie widoczny w środowisku Delphi.



Rysunek 14.5. Miejsce, w którym można wpisać opis pakietu

   - Na zakładce *Version Info* możemy ustalić numer wersji; warto zaznaczyć opcję *Auto-increment build number* (automatycznie zwiększaj numer pełnej kompilacji) — chodzi o zwiększanie ostatniego numeru w czteroczęściowym oznaczeniu wersji każdorazowo przed kompilacją.
   - Warto również wypełnić pola opisujące pakiet (opis pliku, nazwę pakietu itp.) — są to informacje, które w systemie Windows można zobaczyć w oknie własności plików.

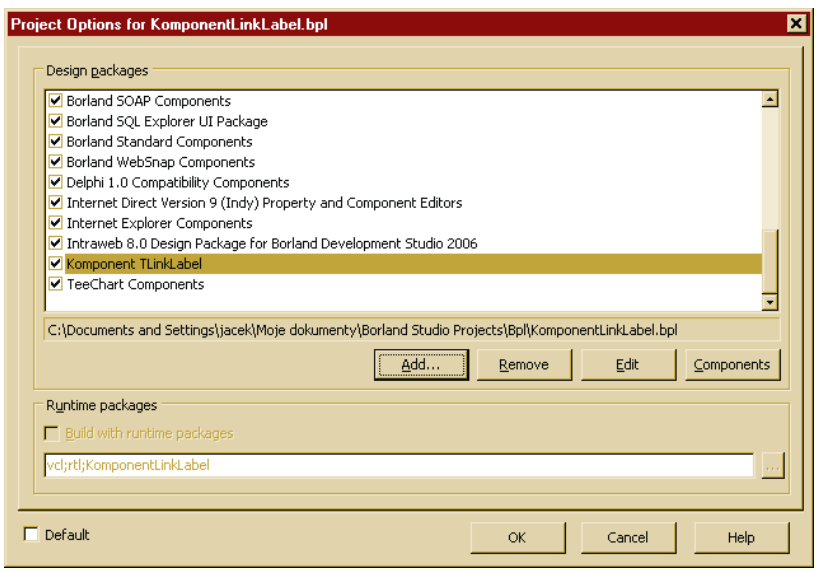
5. Z tego samego menu kontekstowego podokna projektu wybieramy polecenie *Compile*.

Skompilowany plik pakietu *KolorowyPasekPostepu.bpl* nie zostanie umieszczony w katalogu projektu, ale w osobnym katalogu *Moje dokumenty\Borland Studio Projects\Bpl*. Dzięki takiemu zabiegowi rozwiązany został problem wykorzystywania komponentu w wielu projektach.

Instalowanie komponentu VCL dla Win32

Instalowanie komponentu w środowisku Delphi to najprostszy krok w procesie projektowania komponentu. Będzie on musiał być wykonany również przez użytkownika naszego komponentu, jeżeli zdecydujemy się go udostępnić.

1. Jeżeli komponent instalujemy na innym komputerze, należy skopiować go do wybranego katalogu pliku *KomponentLinkLabel.bpl*, *KomponentLinkLabel.dcp* oraz *LinkLabel.dcu*[5] (ten ostatni znajdziemy nie w katalogu *Bpl*, ale w katalogu projektu). Pierwsze dwa będą konieczne do instalacji, a trzeci dopiero do kompilacji projektu wykorzystującego nasz komponent. Załóżmy, że umieszczamy powyższe pliki w katalogu *Moje dokumenty\Borland Studio Projects\Bpl*.
2. Z menu *Component* wybieramy polecenie *Install Packages…*.
3. W oknie widocznym na rysunku 14.6 klikamy przycisk *Add…* i w oknie wyboru pliku odnajdujemy plik *Moje dokumenty\Borland Studio Projects\Bpl\KomponentLinkLabel.bpl*.

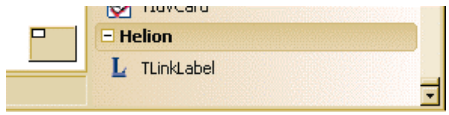


Rysunek 14.6. Lista zainstalowanych komponentów VCL

4. Klikamy *Ok*, zamykając okno z uzupełnioną listą komponentów.

[5] Co oznaczają rozszerzenia nazw plików? Skrót *bpl* oznacza *Borland package library* (biblioteka pakietu Borlanda), *dcp* to *Delphi compiled package* (skompilowany pakiet Delphi), *dpc* to *Delphi package collection* (zbiór pakietów Delphi), *dcu* — *Delphi compiled unit* (skompilowany moduł Delphi).

Komponent zostanie zainstalowany i od tej chwili będzie dostępny we wszystkich projektach aplikacji *VCL Forms Application — Delphi for Win32*. W palecie komponentów zobaczymy zakładkę *Helion* (lub inną, w zależności od nazwy podanej przez Czytelnika w kreatorze komponentu), na której będzie widoczny jest komponent `TLinkLabel` (rysunek 14.7). Możemy stworzyć nowy projekt i bez problemu umieścić komponent na jego powierzchni. W inspektorze obiektów zobaczymy zdefiniowane przez nas własności. Krótko mówiąc, możemy napawać się swoim sukcesem.



Rysunek 14.7. Po zainstalowaniu nasz komponent będzie widoczny na palecie komponentów we wszystkich projektach typu Delphi for Win32

Istnieje również możliwość automatycznej instalacji komponentu. Po skompilowaniu można to zrobić za pomocą polecenia *Install* z menu kontekstowego pozycji *KomponentLinkLabel.bpl* w oknie projektu. To jednak dotyczy wyłącznie sytuacji, w której dysponujemy projektem komponentu.

Aby stworzyć projekt pakietu przeznaczonego dla platformy .NET

Kod komponentu `TLinkLabel` można bez modyfikacji wykorzystać także w projektach *VCL Forms Application — Delphi for .NET*. Należy w tym celu stworzyć osobny pakiet, z którego może on być zainstalowany. Pakiet tworzymy niemal tak samo jak w przypadku platformy Win32. Najważniejsza różnica polega na tym, że po dodaniu do projektu pakietu pliku *LinkLabel.pas* konieczne będzie uwzględnienie w nim również referencji do biblioteki *Borland.Vcl*. Zawiera ona bowiem wykorzystywane w komponencie klasy VCL.NET, m.in. `TCustomLabel`.

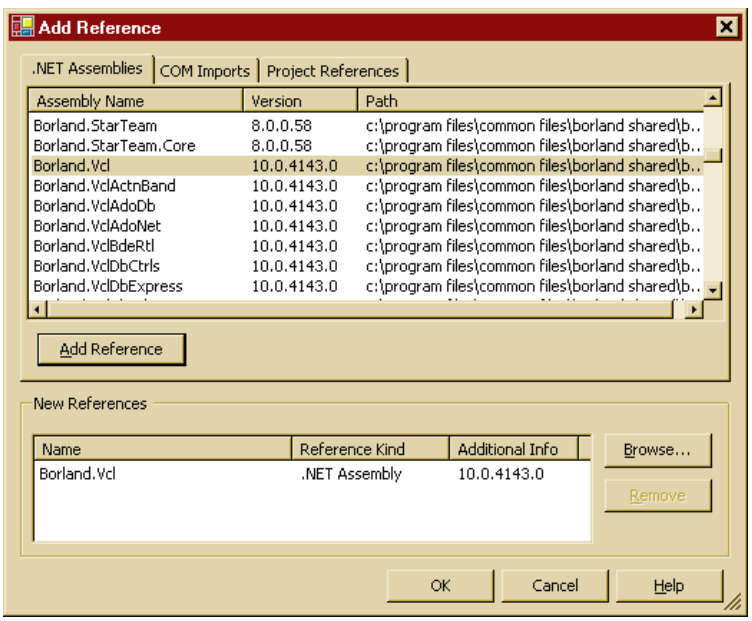
1. Z menu *File*, podmenu *New* wybieramy pozycję *Other…*.
2. W oknie *New Items* w drzewie *Item Categories* wybieramy *Delphi for .NET Projects*.
3. Zaznaczamy ikonę *Package* w prawej części okna i klikamy *OK*.
4. Zapisujemy nowy projekt do osobnego katalogu (klawisze *Ctrl+Shift+S*) pod nazwą *KomponentLinkLabel.bdsproj*.

5. Do projektu dodajemy plik *LinkLabel.pas*. W tym celu:

- z menu kontekstowego pozycji *KomponentLinkLabel.dll* w oknie projektu wybieramy polecenie *Add*;
- w oknie *Add* klikamy przycisk *Browse* i w oknie dialogowym odnajdujemy plik *LinkLabel.pas*, znajdujący się w katalogu projektu opisanego wyżej (ten sam, który dołączyliśmy do pakietu przeznaczonego dla Win32); możemy wcześniej skopiować pliki *LinkLabel.pas* i *LinkLabel.dcr* do katalogu projektu pakietu dla platformy .NET;
- po powrocie do okna *Add* klikamy *OK*;
- w oknie projektu powinniśmy zobaczyć dodany plik *.pas* oraz towarzyszący mu plik *.dcr*.

6. Aby do projektu dodać referencję do biblioteki *Borland.Vcl*:

- w oknie projektu z menu kontekstowego pozycji *KomponentLinkLabel.dll* wybieramy polecenie *Add Reference...*;
- w oknie *Add Reference* (rysunek 14.8) odnajdujemy bibliotekę *Borland.Vcl* i zaznaczamy ją;



Rysunek 14.8. Dodawanie referencji do bibliotek DLL zawierających komponenty VCL.NET

- klikamy przycisk *Add Reference*;
- zamykamy okno klikając przycisk *OK*;
- nazwa biblioteki pojawi się w gałęzi *Requires* okna projektu.

7. Warto jeszcze wejść do opcji projektu (pozycja *Options...* w menu kontekstowym) i wpisać opis pakietu (por. rysunek 14.5).
8. Teraz możemy bez przeszkód skompilować pakiet, wybierając polecenie *Compile* z menu kontekstowego pozycji *KomponentLinkLabel.dll* w oknie projektu.

Podczas kompilacji pojawi się seria ostrzeżeń o tym, że klasy biblioteki *Borland.Vcl.dll* są zadeklarowane jako *specific to a platform* — nie są przenaszalne na inne platformy systemowe. Oznacza to, że nie będziemy mogli użyć komponentu w przypadku platformy .NET, np. w Linuksie, bo biblioteka *Borland.Vcl.dll* związana jest wyłącznie z systemem Windows. Powodem jest wykorzystanie funkcji WinAPI poprzez mechanizm PInvoke w niektórych komponentach i klasach VCL.NET. Jeżeli nie zamierzamy używać naszych aplikacji pod Linuksem, a jest to nadal raczej egzotyczna możliwość, to nie musimy się tymi komunikatami w ogóle przejmować.

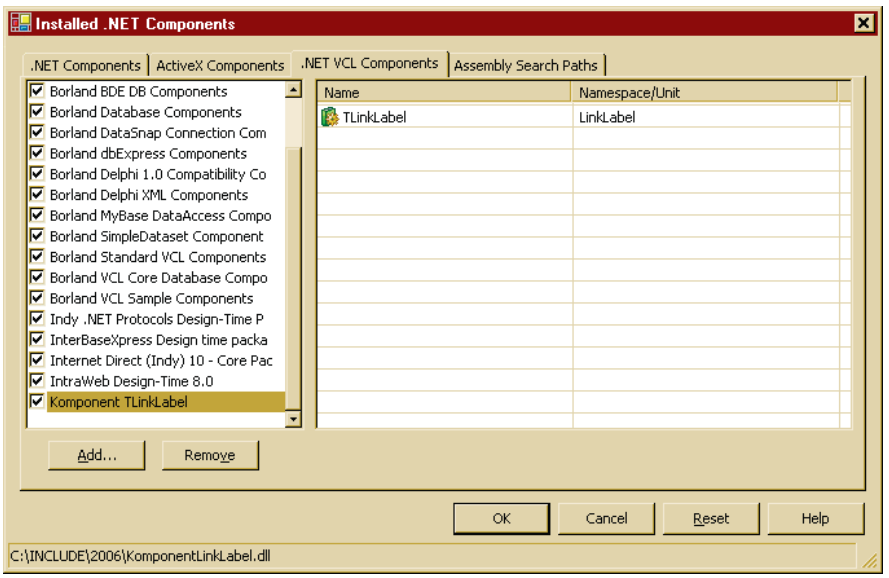
Tym razem skompilowany plik komponentu *KomponentLinkLabel.dll* zostanie umieszczony w katalogu projektu.

Instalacja komponentu .NET

Zacznijmy od zorganizowania sobie miejsca, na wzór stworzonego przez Delphi katalogu *Bpl* dla komponentów VCL, w którym będziemy przechowywać stworzone przez nas lub pobrane z sieci komponenty VCL.NET. Dzięki temu unikniemy dodawania do ścieżki przeszukiwania (zob. punkt 3. poniżej) osobnych katalogów dla każdego dodawanego komponentu.

1. Załóżmy, że Czytelnik moim wzorem założy katalog *C:\Include\2006*. Należy do niego skopiować bibliotekę *KomponentLinkLabel.dll* oraz plik *KomponentLinkLabel.dcpil*.
2. Następnie w środowisku BDS 2006 z menu *Component* wybieramy *Installed .NET Components...*.
3. Dodajemy katalog *C:\Include\2006* do ścieżki przeszukiwania:
   - przechodzimy na zakładkę *Assembly Search Paths* (ścieżki przeszukiwania zasobów);
   - pod listą katalogów wpisujemy w polu edycyjnym katalog zawierający komponent. Zgodnie z powyższymi uwagami może to być *C:\Include\2006*;
   - klikamy przycisk *Add*.

4. Teraz możemy zainstalować nowy komponent VCL.NET. W tym celu:
   - przechodzimy na zakładkę *.NET VCL Components* (rysunek 14.9);



Rysunek 14.9. Zestaw komponentów VCL.NET zainstalowanych w środowisku Delphi

   - klikamy przycisk *Add...* i w typowym oknie dialogowym wskazujemy plik *C:\Include\2006\KomponentLinkLabel.dll*.

5. Klikamy *OK*, aby zamknąć okno *Installed .NET Components*.

6. Aby zobaczyć nowy komponent na palecie, wystarczy stworzyć lub wczytać projekt *VCL Forms Application — Delphi for .NET*.

Ostateczne testowanie komponentu

To już pestka. Otwórzmy projekt naszej przeglądarki w wersji dla Win32 lub dla .NET. Przejdźmy na zakładkę *Unit1*. Dodajmy do aplikacji dodatkową formę i przygotujmy jej interfejs, np. według wzoru widocznego na rysunku 14.10. Jednym z komponentów widocznych na tym rysunku jest oczywiście nasz `TLinkLabel`. Jego własność `Caption` zawiera napis widoczny na formie, a własność `Adres` — adres strony WWW, do której chcemy odesłać użytkownika aplikacji. Zapisujemy nową formę na dysku i wracamy na zakładkę *Unit1*.

Rysunek 14.10. Najprostszy sposób wykorzystania komponentu TLinkLabel

Następnie do menu głównego aplikacji dodajmy podmenu *Pomoc*, a w nim pozycję *O...*. Klikamy dwukrotnie tę pozycję w edytorze menu, aby utworzyć metodę zdarzeniową, w której modalnie otwieramy zaprojektowaną przed chwilą formę (listing 14.11). Przy próbie pierwszej kompilacji (*F9*) Delphi poinformuje nas, że moduł *Unit2* nie jest widoczny z bieżącego modułu i zaproponuje, że może go dodać. Chętnie się na to zgadzamy, naciskając przycisk *Yes*, i ponownie uruchamiamy kompilację. I to by było na tyle.

Listing 14.11. Wywołujemy Form2 jako okno modalne

```
procedure TForm1.O1Click(Sender: TObject);
begin
Form2.ShowModal;
end;
```

W domu

Własności w klasie TRownanieKwadratowe

Do klasy `TRownanieKwadratowe` z rozdziału piątego proszę dodać własności pozwalające na przypisywanie wartości współczynnikom `a`, `b` i `c`, które powinny być zmienione na prywatne, oraz własności tylko do odczytu (nieposiadające w definicji części ze słowem kluczowym `write`), pozwalające na odczytanie rozwiązań.

Rozwijanie komponentu TLinkLabel

Można na wiele sposobów rozbudować powyższy komponent. Można go na przykład uatrakcyjnić, dodając możliwość pojawiania się i znikania podkreślenia pod jego etykietą w zależności od tego, czy kursor myszy znajduje się nad komponentem, zupełnie tak, jak robiliśmy z komponentem `TShape` w rozdziale 8. To będzie wymagało użycia zdarzeń `OnMouseEnter` i `OnMouseLeave`. Zamiast podkreślenia możemy zmieniać także choćby kolor komponentu.

Można również zmieniać kolor komponentu w trakcie jego działania. Na przykład niech Czytelnik spróbuje tak zmodyfikować komponent, aby w momencie naciśnięcia klawisza myszki zmienić jego kolor na czarny lub na kolor określony przez dodatkową własność komponentu, a po zwolnieniu klawisza przywrócić kolor pierwotny. Dokładnie tak samo zachowują się linki w przeglądarkach.

Metody wirtualne i klasa abstrakcyjna

Na koniec proponuję zmianę o znacznie poważniejszym charakterze, przeznaczoną raczej dla tych, którzy wiedzą już coś więcej o dziedziczeniu i polimorfizmie. Zauważmy, że nasz komponent `TLinkLabel` mógłby bez problemu służyć także do inicjowania edycji listów e-mail, gdyby metoda `SprawdzAdres`, zamiast sprawdzać obecność przedrostka zawierającego nazwę protokołu *http://*, umiała sprawdzać poprawność adresu poczty elektronicznej. Moglibyśmy zatem napisać klasę rozszerzającą naszą klasę `TLinkLabel`, w której zmodyfikowalibyśmy jedynie tę jedną metodę. Ja jednak proponuję coś więcej. Zmieńmy klasę `TLinkLabel` w klasę abstrakcyjną, to znaczy taką, której metoda `SprawdzAdres` będzie metodą abstrakcyjną (czyli metodą wirtualną bez implementacji) i zdefiniujmy jej dwie klasy potomne: `TLink` i `TMail` implementujące tę metodę w sposób właściwy dla protokołów *http* i *mailto*. Rozwiązanie tego zadania z komentarzami znajduje się w dołączonych do książki źródłach. Mam nadzieję, że zachęci ono Czytelnika do dalszego studiowania możliwości Object Pascala lub innych języków obiektowych dostępnych w Borland Developer Studio.