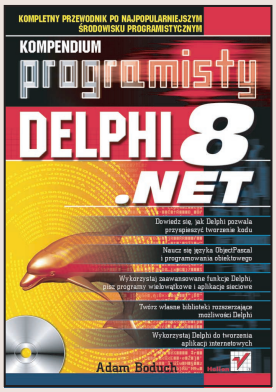
Delphi 8 .NET. Kompendium programisty



Autor: Adam Boduch
ISBN: 83-7361-485-0
Format: B5, stron: 840

Poznaj najnowszą wersję Delphi i stwórz aplikację dla platformy .NET

W dobie tak dynamicznego rozwoju technologii informatycznych możliwość szybkiego stworzenia aplikacji jest dużym atutem. Dlatego powstają coraz bardziej rozbudowane środowiska programistyczne, umożliwiające skonstruowanie aplikacji z „cegiełek", które można wykorzystywać wielokrotnie. Wśród tych środowisk programistycznych zasłużoną popularność zyskało Delphi, dostępne obecnie w wersji 8. Wygodny i czytelny interfejs, język programowania oparty na znanym i popularnym Pascalu, możliwość łatwego połączenia się z dowolną bazą danych oraz rozbudowany mechanizm tworzenia aplikacji wieloplatformowych to główne zalety tego środowiska.

Książka „Delphi 8 .NET. Kompendium programisty" to przewodnik po najnowszej wersji środowiska Delphi, uwzględniający jego nowe możliwości związane z tworzeniem aplikacji dla platformy .NET. Zawiera opis środowiska programistycznego i języka ObjectPascal. Przedstawia sposoby pisania aplikacji połączonych z bazami danych i korzystających z technologii XML. Opisuje również podstawowe zasady wykorzystywania w tworzonych programach protokołów sieciowych, takich jak HTTP i SMTP. Książka jest doskonała zarówno dla początkujących programistów Delphi, jak i dla tych, którzy chcą poszerzyć swoją wiedzę o zasady tworzenia aplikacji dla platformy .NET.

- Zasady tworzenia aplikacji dla platformy .NET
- Wizualne projektowanie aplikacji
- Mechanizm komponentów
- Projektowanie interfejsu użytkownika i menu
- Programowanie w języku ObjectPascal
- Korzystanie z procedur i funkcji
- Zasady projektowania obiektowego
- Programowanie oparte na zdarzeniach
- Wykrywanie i usuwanie błędów w aplikacjach
- Korzystanie z baz danych
- Język SQL
- Wykorzystywanie technologii XML w aplikacjach
- Programowanie sieciowe i ASP.NET
- Usługi sieciowe

Cennym źródłem informacji jest dodatek, zawierający praktyczne wskazówki dotyczące zasad pisania czytelnego i przejrzystego kodu.

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

Spis treści

4.

Delphi bardziej zaawansowane

Jak dotąd mówiliśmy o programowaniu strukturalnym, proceduralnym, ale nie wspomnieliśmy o obiektach. Idea programowania obiektowego jest teraz niezwykle istotna oraz bardzo popularna. Większość języków wysokiego poziomu, np. Delphi, C++, C#, Java, Python, PHP i inne, umożliwia tworzenie obiektów. Na przykład dobry skądinąd C nie umożliwiał programowania obiektowego. I to jest wielka wada tych języków w obecnych czasach.

Rozdział niniejszy poświęcony będzie przede wszystkim opisowi tworzenia klas w Delphi. Omówimy także nowości wprowadzone do Delphi za sprawą .NET, a także zwrócimy uwagę na wiele narzędzi i opcji Delphi związanych z edycją kodu źródłowego.

W tym rozdziale:

- ✧ nauczysz się korzystać z edytora kodu,
- ✧ poznasz opcje edytora kodu,
- ✧ dowiesz się, czym jest programowanie obiektowe i jak tworzyć klasy,
- ✧ poznasz nowe elementy języka programowania wprowadzone za sprawą .NET,
- ✧ zapoznasz się z podstawowymi ideami programowania na platformie .NET,
- ✧ dowiesz się, czym są wyjątki,
- ✧ zostaną Ci przedstawione podstawowe narzędzia dołączone do pakietu .NET Framework.

Na czym polega programowanie obiektowe?

Programy rozrastają się coraz bardziej i bardziej. Tak samo jak kiedyś nie wystarczała idea programowania proceduralnego, teraz nie wystarcza już programowanie strukturalne.

Koncepcja programowania obiektowego pojawiła się już w latach 60. za sprawą języka Simula 67 zaprojektowanego przez naukowców z Oslo w celu przeprowadzania symulacji zachowania się statków. Jednakże idea programowania obiektowego swoją popularyzację zawdzięcza językowi SmallTalk. Połowa lat 80. to czas, kiedy programowanie obiektowe stało się dominującą techniką — głównie za sprawą C++. Wtedy to też w wielu innych językach pojawiła się możliwość tworzenia obiektów.

Można powiedzieć, że *klasa* to „paczuszka", pewien element programu, który wykonuje jakieś funkcje. Klasa zawiera metody (procedury i funkcje) współdziałające ze sobą w celu wykonania jakiegoś zadania. Programowanie obiektowe przyczyniło się do tego, że takie „podprogramy" jak klasy mogą być wykorzystywane w wielu innych projektach — ułatwia to jeszcze bardziej zarządzanie i konserwację kodu.

Załóżmy, że napisałeś klasę do obsługi poczty (wysyłanie i odbieranie). Klasa może mieć procedury `Connect` (połącz), `SendMail` (wyślij e-mail), `Disconnect` (rozłącz). Z kolei procedura `Connect` może wywoływać inną, np. `Error` (która też jest procedurą znajdującą się w klasie), wyświetlającą błąd w razie niepowodzenia i zapisującą odpowiedni tekst w *dzienniku programu* (czyli, inaczej mówiąc, w *logach* — plikach z rozszerzeniem **.log*). Teraz taką klasę możesz wykorzystać w wielu swoich aplikacjach — wystarczy skopiować fragment kodu i już gotowa jest obsługa błędów, łączenie itp. Taką klasę możesz udostępnić innym użytkownikom lub swoim współpracownikom. Taki inny użytkownik nie musi wiedzieć, jak działa klasa — ważne jest dla niego, co ona robi (wysyła e-maile). Użytkownik musi jedynie wiedzieć, że istnieje metoda `Connect`, która połączy go z danym serwerem oraz musi mieć świadomość obecności kilku innych procedur. To wszystko — nie interesuje go obsługa błędów — nie musi nawet zdawać sobie sprawy z jej istnienia.

Można by oczywiście utworzyć nowy moduł, a w module umieścić także procedury `Connect`, `SendMail` oraz `Disconnect`, `Error` i resztę potrzebnego kodu. Jednak w takim przypadku metody i zmienne (zmienne także mogą być elementami danej klasy) nie oddziałują na siebie w takim stopniu. Przykładowo, użytkownik korzystający z takiego kodu będzie miał dostęp do tych zmiennych, do których nie powinien go mieć. Będzie mógł też wywołać swobodnie procedurę `Error` — a nie powinien, bo może to spowodować niepożądane skutki. Dzięki klasom możesz sprawić, iż taka procedura `Error` nie będzie dostępna **poza** klasą; jej elementy (zmienne) też będą nie do odczytania przez przyszłego użytkownika.

Biblioteki wizualne

W rzeczywistości biblioteki takie jak VCL.NET (ang. *Visual Component Library*) to szereg modułów i klas współpracujących ze sobą. Dzięki idei programowania obiektowego to wszystko działa bardzo sprawnie i jest nad wyraz czytelne. Głównie dzięki funkcji *dziedziczenia*, która

polega na rozszerzeniu funkcjonalności klas. Klasa potomna może dziedziczyć po klasie bazowej, przejmując od niej całą funkcjonalność, ale jednocześnie można rozszerzyć ją o nowe funkcje. Tak samo zbudowana jest biblioteka VCL — w oparciu o dziedziczenie szeregu klas.

Podstawowy kod formularza

Poprzednio zajmowaliśmy się jedynie *Windows Forms*, ale od strony wizualnej, czyli głównie projektantem formularzy. Nie zajmowaliśmy się szczegółami takimi jak kod źródłowy *formularza*. Po otwarciu nowego projektu *WinForms* na pierwszym planie pojawi się czysty formularz. Klawiszem *F12* przejdź do edytora kodu. Podstawowy kod modułu źródłowego przedstawiony został na listingu 4.1.

Listing 4.1. *Podstawowy kod modułu WinForms*

```
unit WinForm2;

interface

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

type
  TWinForm2 = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    /// <summary>
    /// Required designer variable.
    /// </summary>
    Components: System.ComponentModel.Container;
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    procedure InitializeComponent;
  {$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(TWinForm2))]

implementation
```

```
{$REGION 'Windows Form Designer generated code'}
/// <summary>
/// Required method for Designer support -- do not modify
/// the contents of this method with the code editor.
/// </summary>
procedure TWinForm2.InitializeComponent;
begin
  Self.Components := System.ComponentModel.Container.Create;
  Self.Size := System.Drawing.Size.Create(300, 300);
  Self.Text := 'WinForm2';
end;
{$ENDREGION}

procedure TWinForm2.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;

constructor TWinForm2.Create;
begin
  inherited Create;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent;
  //
  // TODO: Add any constructor code after InitializeComponent call
  //
end;

end.
```

Należy już na samym początku zaznaczyć, że składnia modułu *WinForms* znacząco różni się od standardowego VCL.NET (listing 4.2).

Listing 4.2. *Kod formularza w VCL.NET*

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.nfm}

end.
```

Na pierwszy rzut oka widać wyraźną różnicę pomiędzy kodem formularza *WinForms* a VCL.NET. Przede wszystkim ten drugi jest prostszy i krótszy.

Na razie omówimy podstawowe części kodu formularza w VCL.NET, gdyż jest on prostszy i może ułatwić użytkownikowi zrozumienie znaczenia poszczególnych elementów.

Formularz w VCL.NET

Moduły i ich budowa zostały szczegółowo omówione w poprzednim rozdziale poświęconym elementom języka Delphi. Zapamiętaj! Każdemu formularzowi odpowiada jeden moduł z odpowiednim kodem, lecz to stwierdzenie nie działa w drugą stronę — moduł niekoniecznie musi być formularzem.

Podczas projektowania wizualnego Delphi samodzielnie edytuje kod źródłowy, dodając lub zmieniając odpowiednie elementy — dzięki temu projektowanie staje się szybsze, a my nie musimy martwić się o szczegóły związane z dodaniem jakiegoś potrzebnego elementu.

Kod źródłowy formularza (modułu) składa się także ze standardowych sekcji i słów kluczowych typu: `unit`, `interface`, `uses`, `implementation`.

Sekcja uses

Podczas tworzenia nowego projektu Delphi doda do sekcji `uses` standardowe moduły konieczne do prawidłowego działania aplikacji. W tych modułach znajdują się deklaracje odpowiednich klas, dzięki czemu projekt może zostać skompilowany. Przykładowo, doprowadź sekcję `uses` do takiego wyglądu:

```
uses
  Windows;
```

Po próbie kompilacji Delphi już na początku wyświetli błąd: `[Error] Unit1.pas(9): Undeclared identifier: 'TForm'`. Komunikat mówi o tym, że nazwa `TForm` nie jest rozpoznawana dla kompilatora. Stało się tak dlatego, że usunęliśmy z sekcji `uses` moduł `Forms`, który zawierał deklarację typu `TForm`.

Klasa

Poniższy fragment w kodzie źródłowym jest deklaracją klasy, o której tyle mówiliśmy:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

W tym momencie zadeklarowany został nowy typ — `TForm1`, który jest klasą wywodzącą się z kolei z klasy bazowej `TForm`.

Wskazówka

W VCL.NET nazwy wszystkich klas zaczynają się od litery `T`.

Zmienna wskazująca na klasę

Poniżej deklaracji klasy, jeszcze w sekcji `interface`, zadeklarowana jest standardowo zmienna, która wskazuje na nasz typ — `TForm1`:

```
var
  Form1: TForm1;
```

Plik *.nfm

Komponenty umieszczane na formularzu mogą mieć różne właściwości. Te dane muszą być gdzieś przechowywane. W VCL.NET miejscem, gdzie zapisywane są te informacje, jest plik **.nfm* (w poprzednich wersjach Delphi plik miał rozszerzenie **.dfm*), którego zawartość została przedstawiona na listingu 4.3.

Listing 4.3. *Zawartość pliku *.nfm*

```
object Form1: TForm1
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  Height = 240
  Width = 347
```

```
  Left = 13
  Top = 13
  PixelsPerInch = 96
  TextHeight = 13
end
```

Praktycznie nigdy nie zachodzi potrzeba, aby ręcznie edytować ten plik. To zadanie należy już do Delphi, które dokonuje zapisu i odczytu danych właśnie z tego pliku. Pliki **.nfm* zawierają informacje na temat formularza, jego właściwości, komponentów oraz właściwości komponentów.

W pliku źródłowym formularza znajduje się dyrektywa każąca kompilatorowi dołączyć do projektu plik **.nfm*:

```
{$R *.nfm}
```

Jest to specjalna instrukcja — na pierwszy rzut oka wyglądająca jak komentarz, jednak jest informacją dla kompilatora o włączeniu do projektu wszystkich plików z rozszerzeniem **.nfm*.

Generowanie kodu

Powróć do projektanta formularzy. Odszukaj na palecie narzędzi komponent `DBGrid` i umieść go na formularzu. Z powrotem przejdź do edytora kodu — zauważ, że Delphi w sekcji `uses` dodało nowe moduły: `Borland.Vcl.Grids` oraz `Borland.Vcl.DBGrids`, które są wymagane do prawidłowego skompilowania aplikacji. Dodało również nową linijkę w klasie `Form1`:

```
DBGrid1: TDBGrid;
```

Dzięki temu oszczędzamy na czasie, nie musząc wpisywać tego ręcznie.

Powyższa linia w klasie `Form1` identyfikuje komponent *typu* `TDBGrid` o *nazwie* `DBGrid1`. Delphi automatycznie nadaje własne nazwy tworzonym komponentem, przypisując im kolejne numery: `DBGrid1`, `DBGrid2` itd. Jeżeli na formularzu umieścisz kilka komponentów typu `TButton`, to Delphi nazwie je odpowiednio: `Button1`, `Button2` itd.

Nazwy komponentów

Nazwa komponentu jest symboliczna. Z punktu widzenia kompilatora nie jest ważne, czy komponent nazywa się `Button1` czy `ToJestMojFajnyButtonikNumer1`. Ważne jest jedynie to, aby nazwa była unikalna, a więc nie mogą istnieć dwa komponenty o tej samej nazwie. W nazwie komponentu nie mogą być użyte polskie znaki; nie może także zaczynać się ona od cyfry. W nazewnictwie komponentu istnieją takie same reguły co w deklaracji zmiennej.

Nazwę komponentu można zmienić, korzystając z inspektora obiektu. Znajdź właściwość `Name`, której możesz nadać nową wartość będącą nazwą komponentu. Po zmianie nazwy Delphi uaktualni również wszelkie wpisy w kodzie źródłowym związane z tą nazwą.

Wskazówka

Nigdy nie pozostawiaj domyślnych nazw komponentów, jakie nadaje Delphi (tzn. `Button1`, `Button2` itd.). W późniejszym czasie łatwo jest się pogubić w takim nazewnictwie. Staraj się nazywać komponenty opisowo — to znaczy tak, aby opisywały rolę, jaką pełnią. Jeżeli, przykładowo, masz komponent, który wyświetla informacje o autorze programu, nazwij go `btnAbout`; w przypadku przycisku, który będzie zamykał program — niech będzie to nazwa `btnClose`.

Programowanie zdarzeniowe

Programy napisane w Delphi są programami *zdarzeniowymi*, które wykonują pewne czynności pod wpływem nakazu użytkownika. Co to oznacza? Kiedy program jest uruchamiany, następuje jego załadowanie do pamięci, rezerwowanie pamięci itp. Następnie program taki czeka na zadania. Projektując nasze aplikacje wizualne, programujemy pewne zdarzenia, na jakie ma reagować program — np. kliknięcie przyciskiem myszy w obszarze komponentu. Jeżeli aplikacja odbierze takie zdarzenie (użytkownik kliknie w obszarze komponentu), podejmowane jest pewne działanie. To samo tyczy się innych przypadków — poruszanie myszą, wprowadzanie tekstu z klawiatury itp.

Generowanie zdarzeń

W kodzie źródłowym modułu nie możesz ręcznieumieszczać instrukcji wykonujących pewne czynności (pomijamy tu sekcję `initialization` oraz `finalization`) — wszystko musi być zawarte w procedurach zdarzeniowych.

1. Utwórz nowy projekt VCL.NET.
2. Umieść na formularzu komponent `TButton`.
3. Kliknij dwukrotnie w przycisk (komponent `TButton`).

W tym momencie Delphi powinno wygenerować w kodzie źródłowym zdarzenie i przenieść Cię do edytora kodu, natomiast kursor ustawić w ciele procedury:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Właśnie we wnętrzu tej procedury należy umieścić kod, który będzie wykonywany po naciśnięciu w trakcie działania programu przycisku. W naszym przykładzie spowodujemy tym zmianę właściwości komponentu.

Przykładowy program

Jak dotąd zmieniałeś właściwości komponentu jedynie z poziomu inspektora obiektów. Powinieneś wiedzieć, że jest to możliwe także z poziomu kodu źródłowego za pomocą operatora odwołania (.).

Nasz przykładowy program zmieniał będzie pozycję przycisku na formularzu. Nowa pozycja będzie losowana i ustawiana za każdym razem, gdy użytkownik kliknie przycisk.

Każda kontrolka wizualna w VCL.NET posiada właściwość `Width` (szerokość), `Height` (wysokość), `Left` (pozioma pozycja na formularzu), `Top` (pionowa pozycja na formularzu). Tak więc nadanie nowej wartości dla właściwości z poziomu kodu wygląda następująco:

```
Button1.Width := 100; // szerokość komponentu
```

Do konkretnych właściwości odwołujemy się za pomocą operatora odwołania (kropka), a wartości nadajemy tak jak zwykłym zmiennym.

Rozwiązanie

Przy każdorazowym naciśnięciu przycisku program powinien losować nową wartość z zakresu i ustawiać komponent na tej wylosowanej wartości. Pytanie: jaki to będzie zakres? Szerokość i wysokość formularza!

Listing 4.4 przedstawia kod całego modułu programu.

Listing 4.4. *Kod modułu*

```
unit Unit1;

interface

uses
  Windows, Messages, Variants, SysUtils, Graphics, Forms,
  Dialogs, System.ComponentModel, Borland.Vcl.Controls,
  Borland.Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.nfm}
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FormWidth, FormHeight : Integer;
begin
{ przypisz wartości opisujące szerokość i wysokość formularza }
  FormWidth := Form1.Width - 100;
  FormHeight := Form1.Height - 50;

  Randomize; // losowanie

{ nadanie nowych wartości właściwościom Left i Top }
  Button1.Left := Random(FormWidth);
  Button1.Top := Random(FormHeight);
end;

end.
```

Szerokość i wysokość formularza odczytujemy z właściwości `Width` oraz `Height`, a następnie przypisujemy te wartości do zmiennych zadeklarowanych wcześniej: `FormWidth` oraz `FormHeight`:

```
  FormWidth := Form1.Width - 100;
  FormHeight := Form1.Height - 50;
```

Zwróć uwagę, że dodatkowo odjęliśmy od tych wartości odpowiednio liczby 100 i 50, tak aby zachować pewien margines podczas losowania liczb.

Następnie wylosowane wartości posłużą jako nowe współrzędne położenia komponentu `TButton`:

```
  Button1.Left := Random(FormWidth);
  Button1.Top := Random(FormHeight);
```

Wiesz już, dlaczego odjęliśmy 100 i 50 od wartości zmiennej `FormWidth` i `FormHeight`? Zakładając, że szerokość formularza to 500, program będzie losował jedynie z zakresu od 0 do 400, tak aby nie wylosował liczby 500 — gdyby tak się stało, to komponent nie zmieściłby się całkowicie w oknie formularza (jego krawędź wystawałaby poza obszar formularza). Wszystko dlatego, że właściwość `Left` określa pozycję lewej krawędzi komponentu względem formularza.

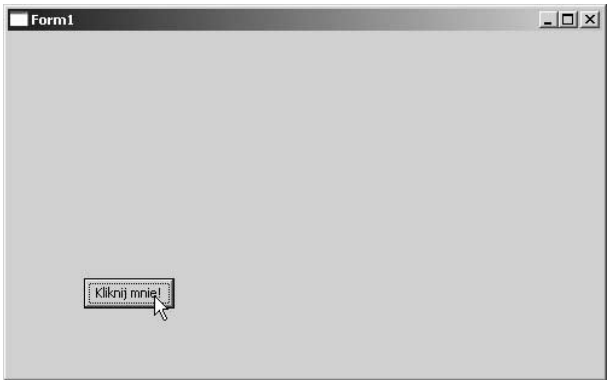
Zmiany kosmetyczne

1. W inspektorze obiektów zmień właściwość `Width` oraz `Height` formularza na `500`.
2. Zmień właściwość `Caption` przycisku (`Button1`) na `Kliknij mnie!`.

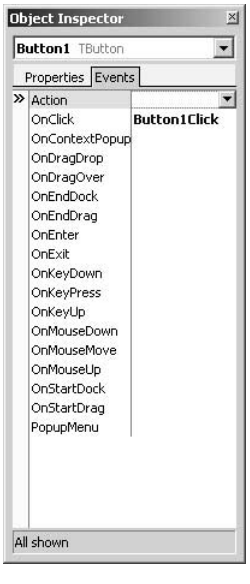
Działanie programu przedstawione zostało na rysunku 4.1.

Lista zdarzeń w inspektorze obiektów

Wspominaliśmy wcześniej o zakładce *Events* inspektora obiektów. Po wykonaniu wcześniejszego ćwiczenia na liście *Events* przy pozycji `OnClick` będzie widniał napis: `Button1Click` (rysunek 4.2).



Rysunek 4.1. *Program w trakcie działania*



Rysunek 4.2. *Inspektor obiektów z wygenerowanym zdarzeniem OnClick*

Po zaznaczeniu zdarzenia `OnClick` zostanie ono podświetlone na szaro. Po przeniesieniu kursora nad tekst, powinien on zmienić kształt. W tym miejscu po podwójnym kliknięciu zostaniesz przeniesiony do deklaracji zdarzenia `OnClick`. Tak samo jest z generowaniem innych zdarzeń — np. `OnKeyDown`.

Jedno zdarzenie dla kilku obiektów

Czasami może zajść potrzeba „podpięcia” tego samego zdarzenia `OnClick` (lub jakiegokolwiek innego) w kilku przyciskach lub innych komponentach.

W takim wypadku po zaznaczeniu wybranej pozycji, np. `OnClick`, po prawej stronie pojawia się strzałka, po naciśnięciu której rozwija się lista z dostępnymi w programie zdarzeniami. Możesz dla swojego komponentu wybrać dowolne zdarzenie.

Generowanie pozostałych zdarzeń

Możesz korzystać ze wszystkich zdarzeń, jakie w obrębie danego komponentu oferuje Ci Delphi. Przykładowo, zdarzenie `OnMouseMove` umożliwia Ci oprogramowanie zdarzenia przesuwania kursora nad obiektem. Innymi słowy, możesz odpowiednio na taką sytuację zareagować.

Jak odbywa się generowanie zdarzeń z poziomu inspektora obiektów? Zaznacz zdarzenie, niech to będzie `OnMouseMove`, klikając je. Po prawej stronie pojawi się lista rozwijalna, która na razie jest pusta. Przesuń kursor nad białe pole — kursor powinien zmienić swój kształt. W tym momencie kliknij dwukrotnie — spowoduje to wygenerowanie zdarzenia `OnMouseMove`.

```
procedure TMainForm.btnMainMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin

end;
```

Wskazówka

W niniejszej książce dla określenia nazw komponentów będę posługiwał się prawidłowym określeniem z literą *T* na początku — np. `TButton` czy `TLabel`. Są to prawidłowe nazwy dla komponentów VCL, chociaż wiele osób nadaje komponentom nazwy pozbawione tej litery — `Button`, `Label`.

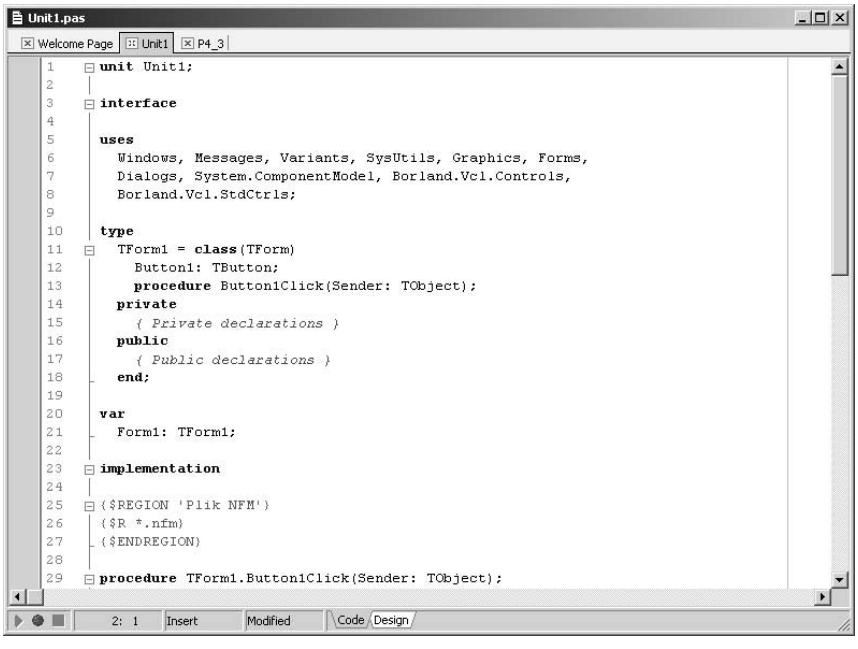
Jeżeli wygenerowane zdarzenie pozostanie puste, tj. nie będzie zawierało żadnego kodu, zostanie ono usunięte podczas kompilacji lub zapisu projektu.

Edytor kodu

Zanim przystąpimy do dalszej pracy, wypadałoby, abyś poznał główne narzędzie pracy używane w tym rozdziale — edytor kodu (rysunek 4.3).

Edytor kodu to zwykły edytor tekstów. Podzielony jest na zakładki, co daje możliwość jednoczesnej edycji wielu plików. W przypadku gdy mamy do czynienia z kodem źródłowym formularza, okno dodatkowo jest podzielone na dole na dwie zakładki — *Code* (pisanie kodu) oraz *Design* (projektowanie wizualne).

Trzeba przyznać, że w nowej wersji edytor kodu uległ zmianie — przede wszystkim dzięki dodaniu nowych opcji i możliwości.



Rysunek 4.3. *Edytor kodu*

Ukrywanie kodu

Jedną z nowości jest możliwość ukrywania fragmentu kodu. W takim przypadku po lewej stronie edytora wyświetlone zostaną małe ikonki, których kliknięcie spowoduje rozwinięcie ukrytego fragmentu. Jest to dość ciekawe rozwiązanie — umożliwia zwiększenie przejrzystości w kodzie i schowanie fragmentu, który nas nie interesuje.

Wszystko to odbywa się za sprawą dyrektywy {$REGION}, którą należy wstawić przed blokiem, który chcemy ukryć. Wygląda to następująco:

```
{$REGION 'Nazwa regionu'}
// kod, który chcemy ukryć
{$ENDREGION}
```

Po wpisaniu takich instrukcji możliwe będzie ukrycie danych zapisanych pomiędzy tymi dyrektywami.

Makra

Mówiąc ogólnie, makra (również wprowadzone jako nowość w Delphi 8) umożliwiają „nagrywanie" pewnych operacji dokonanych w kodzie źródłowym, a następnie ich „odtwarzanie". Wszystkie trzy przyciski z tym związane znajdują się w lewym dolnym rogu edytora kodu.

Przeprowadź pewien eksperyment potwierdzający moje słowa: kliknij przycisk *Record Macro* (środkowy przycisk) i wpisz w edytorze kodu tekst:

```
ShowMessage('Nagrywamy makro...');
```

Po wpisaniu kliknij przycisk *Stop Recording Macro*. Od tej pory aktywny powinien być pierwszy przycisk — *Playback Macro*. Po jego kliknięciu w edytorze kodu ponownie zostanie wstawiona nagrana instrukcja.