

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

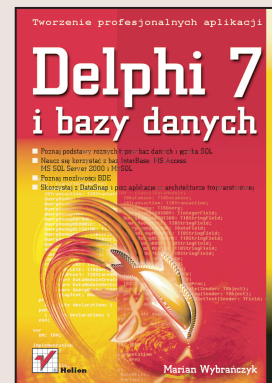
Delphi 7 i bazy danych

Autor: Marian Wybrańczyk

ISBN: 83-7361-129-0

Format: B5, stron: 240

Zawiera CD-ROM



Borland Delphi to jedno z najpopularniejszych narzędzi służących do szybkiego tworzenia aplikacji bazodanowych. Używając Delphi nie tylko w prosty sposób połączysz się z wieloma systemami zarządzania relacyjnymi bazami danych, ale także szybko stworzysz elegancki i wygodny interfejs, pozwalający końcowemu użytkownikowi na dostęp do danych. Właśnie stąd wzięła się ogromna popularność Delphi przy pisaniu aplikacji bazodanowych klient-serwer.

Książka przedstawia zarówno rozmaite systemy bazodanowe, z jakimi można spotkać się w praktyce programistycznej (w tym m.in. InterBase, MS Access, MS SQL Server 2000 i MySQL) jak też i podstawowe komponenty wspomagające z poziomu Delphi 7 zarządzanie danymi. Prześledzisz proces tworzenia bazy danych, modelowania jej struktury i sposobów korzystania z danych w niej zawartych z poziomu Delphi.

Poznasz:

- Podstawowe informacje na temat baz danych i języka SQL
- Narzędzia wspomagające tworzenie i modyfikację bazy danych
- MS Access i interfejs ODBC
- InterBase i interfejs IBX
- MS SQL Server 2000 i interfejs ADO
- MySQL i narzędzie dbExpress
- Metody korzystania z BDE
- DataSnap i tworzenie aplikacji w architekturze trójwarstwowej
- Zasady pisania własnych komponentów

Jeśli zamierzasz pisać w Delphi, wcześniej czy później staniesz przed koniecznością skorzystania z systemu bazodanowego. Kupując tę książkę możesz być pewien, że żaden z tych systemów nie zaskoczy Cię i nie przerośnie Twoich umiejętności.



Spis treści

Wstęp	7
Rozdział 1. Przykładowa baza danych.....	9
Analiza problemu	9
Model bazy danych	10
Uwagi na temat implementacji.....	14
Podsumowanie	16
Rozdział 2. Elementy SQL	17
SQL — co to jest?	17
Baza danych.....	18
Tabele.....	18
Select.....	21
Klucz główny (primary key).....	23
Klucz obcy (foreign key) i integralność referencyjna.....	24
Wartość NULL.....	26
Domena	27
Indeksy	29
Widoki (perspektywy)	30
Wyzwalacze i generatory.....	32
Procedury	34
Transakcje.....	35
Rozdział 3. Narzędzia wspomagające tworzenie i modyfikację bazy danych.....	37
Database Desktop.....	37
Datapump	41
Konfiguracja ODBC	41
Konfiguracja BDE	43
Rozdział 4. MS Access i ODBC	47
Tworzymy bazę danych w MS Access	47
Tabele.....	47
Relacje.....	50
Kwerendy.....	50
Formularze	52

ODBC i MS Access	53
Łączymy się z MS Access poprzez ODBC.....	57
ODBC i XBase	58
Podsumowanie	61
Rozdział 5. InterBase i IBX	63
IBConsole.....	64
Interactive SQL	72
Backup.....	77
Restore.....	79
Użytkownicy i uprawnienia	80
IBX	83
Połączenie z InterBase	84
Monitorowanie bazy danych InterBase.....	108
Odinstalowanie serwera InterBase.....	109
Podsumowanie	109
Rozdział 6. MS SQL Server 2000 i ADO.....	111
Wstęp.....	111
MS SQL Server 2000	112
Tworzymy bazę danych	112
Połączenie z bazą danych.....	115
ADOConnection	116
ADOCommand	118
ADOTable, ADOQuery, ADOStoredProc.....	120
ADODataset	121
ADO i Transakcje	124
Motor JET	126
Podsumowanie	128
Rozdział 7. MySQL i dbExpress	129
Wstęp.....	129
MySQL uruchomienie serwera	130
Użytkownicy i uprawnienia	132
Zmiana hasła administratora	132
Inni użytkownicy	133
Definiowanie użytkownika.....	133
Minimum uprawnień.....	133
Tworzenie bazy danych.....	135
Usuwanie bazy danych.....	135
Tworzenie tabel.....	136
dbExpress	137
SQLConnection.....	139
SQLDataSet	141
Transakcje	151
ClientDataSet	156
Komunikacja dwukierunkowa	161
Informacje na temat bazy danych	164
SQLMonitor	165
Podsumowanie	166
Rozdział 8. BDE	167
Wstęp.....	167
Database	170
Query.....	171

Table	174
UpdateSQL	187
StoredProc	191
Podsumowanie	193
Rozdział 9. DataSnap	195
Wstęp — architektura trójwarstwowa	195
DataSnap	196
Serwer aplikacji	197
Program klienta	199
Ograniczenia	200
Odświeżanie danych	204
Konflikt	206
Podsumowanie	207
Rozdział 10. Podstawy tworzenia komponentów	209
Wstęp	209
Podstawowe informacje	209
Podejście tradycyjne	213
Tworzymy pierwszy komponent	215
Komponenty bazodanowe	220
Kontrolka bazodanowa	223
Styl projektowania komponentów	224
Instalacja komponentu w środowisku Delphi	227
Wykorzystanie komponentu	228
Podsumowanie	230
Dodatek A Adresy Internetowe	231
Skorowidz	233

Rozdział 10.

Podstawy tworzenia komponentów

Wstęp

W tym rozdziale chciałbym przedstawić podstawy związane z tworzeniem komponentów bazodanowych. Korzystając ze środowiska Delphi, używamy przede wszystkim biblioteki VCL (ang. *Visual Components Library*). Jak sama nazwa sugeruje, VCL to biblioteka komponentów. Komponenty te mają nie tylko charakter komponentów wizualnych. Na bibliotekę składają się również komponenty niewizualne, kontrolki zbiorów danych, komponenty związane z Internetem oraz klasy. Komponenty można podzielić na grupy:

- ◆ Komponenty (wywodzą się od klasy TComponent);
- ◆ Kontrolki niewizualne (wywodzą się od klasy TComponent);
- ◆ Kontrolki wizualne (wywodzą się od klasy TControl), a w nich:
 - ◆ Kontrolki okienkowe (wywodzą się od klasy TWinControl),
 - ◆ Kontrolki nieokienkowe (wywodzą się od klasy TGraphicControl).

Podstawowe informacje

Podstawową klasą dla wszystkich komponentów jest klasa TComponent. Jednak klasa ta już dziedziczy po klasie TPersistent (nazwa klasy pochodzi od ang. *persistent* — trwałe). Celowo nie wspominam tutaj o klasie TObject, od której dziedziczą wszystkie klasy, w tym również klasa TPersistent (listing 10.1).

Listing 10.1. Definicja klasy *TPersistent*

```

TPersistent = class(TObject)
private
    procedure AssignError(Source: TPersistent);
protected
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFile); virtual;
    function GetOwner: TPersistent; dynamic;
public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function GetNamePath: string; dynamic;
end;

```

W ramach interfejsu publicznego klasa *TPersistent* udostępnia przede wszystkim metodę *Assign*, która kopiuje aktualną wartość obiektu.

Definicję klasy *TComponent* znajdziemy w pliku (dla domyślnej instalacji Delphi) *C:\program files\borland\delphi7\source\rtl\common\classes.pas*. Klasa ta jest dość rozbudowana, dlatego podam tutaj tylko jej fragment (głównie jej interfejs publiczny). Warto przyjrzeć się poszczególnym składnikom klasy, aby się przekonać, jakie podstawowe cechy posiada każdy komponent (listing 10.2):

Listing 10.2. Fragment definicji klasy *TComponent*

```

TComponent = class(TPersistent, IInterface, IInterfaceComponentReference)
private
    FOwner: TComponent;
    FName: TComponentName;
    FTag: Longint;
    FComponents: TList;
    FDesignInfo: Longint;
    FComponentState: TComponentState;
    ...
public
    constructor Create(AOwner: TComponent); virtual;
    destructor Destroy; override;
    procedure BeforeDestruction; override;
    procedure DestroyComponents;
    procedure Destroying;
    function ExecuteAction(Action: TBasicAction): Boolean; dynamic;
    function FindComponent(const AName: string): TComponent;
    procedure FreeNotification(AComponent: TComponent);
    procedure RemoveFreeNotification(AComponent: TComponent);
    procedure FreeOnRelease;
    function GetParentComponent: TComponent; dynamic;
    function GetNamePath: string; override;
    function HasParent: Boolean; dynamic;
    procedure InsertComponent(AComponent: TComponent);
    procedure RemoveComponent(AComponent: TComponent);
    procedure SetSubComponent(IsSubComponent: Boolean);
    function SafeCallException(ExceptObject: TObject;
                               ExceptAddr: Pointer): HRESULT; override;
    function UpdateAction(Action: TBasicAction): Boolean; dynamic;

```

```

function IsImplementorOf(const I: IInterface): Boolean;
function ReferenceInterface(const I: IInterface; Operation: TOperation): Boolean;

property ComObject: IUnknown read GetComObject;
property Components[Index: Integer]: TComponent read GetComponent;
property ComponentCount: Integer read GetComponentCount;
property ComponentIndex: Integer read GetComponentIndex write SetComponentIndex;
property ComponentState: TComponentState read FComponentState;
property ComponentStyle: TComponentStyle read FComponentStyle;
property DesignInfo: Longint read FDesignInfo write FDesignInfo;
property Owner: TComponent read FOwner;
property VCLComObject: Pointer read FVCLComObject write FVCLComObject;

published
  property Name: TComponentName read FName write SetName stored False;
  property Tag: Longint read FTag write FTag default 0;
end;

```

Klasa ta oferuje pewną cechę związaną z komponentami o nazwie `published` (opublikowane). Jak widać z definicji klasy, dyrektywa `published` jest wymieniona w identyczny sposób jak inne dyrektywy dostępu do pól klasy:

```

TNazwa_Klasy = class
Public
  ...
protected
  ...
private
  ...
published
  property ...
  ...
end;

```

Najprościej rzecz ujmując, można stwierdzić, że to, co znajdzie się po dyrektywie `published` (poprzedzone słowem `property`), widzimy jako właściwości komponentu, gdy podglądamy je w oknie *Object Inspector*. W podanej definicji klasy `TComponent` widzimy dwie opublikowane właściwości (ang. *properties*), którymi są `Name` oraz `Tag`.

```

published
  property Name: TComponentName read FName write SetName stored False;
  property Tag : Longint      read FTag  write FTag default 0;
end;

```

Proszę otworzyć dowolny projekt Delphi i wskazać dowolny komponent. W oknie *Object Inspector* każdego komponentu zobaczymy właściwości `Name` (nazwa komponentu) oraz `Tag`. Przy czym właściwość `Tag`, jak wynika z opisu dokumentacji pomocy środowiska Delphi (menu: *help*), została dodana dla wygody projektanta. Możemy w niej umieścić, co chcemy — w ramach zgodności z typem tej właściwości.

Przypatrując się nadal definicji klasy `TComponent`, możemy zauważyć, że komponent może mieć swojego właściciela:

```

FOwner: TComponent;

```

Komponent zostanie dodany do listy komponentów za pośrednictwem metody `Insert` `Component`; przy czym lista komponentów jest dostępna za pośrednictwem:

```
property Components[Index: Integer]: TComponent read GetComponent;
```

Komponent ma swoją pozycję `ComponentIndex` na liście komponentów:

```
property ComponentIndex: Integer read GetComponentIndex write SetComponentIndex;
```

Do ustawienia wartości `Component Index` służy metoda:

```
procedure SetComponentIndex(Value: Integer);
```

a do pobrania aktualnej wartości:

```
function GetComponentIndex: Integer;
```

Komponent może również być właścicielem innych komponentów. Na przykład komponent `Panel`, na którym możemy umieścić inne komponenty — wówczas `Panel` dla pozostałych komponentów będzie ich właścicielem. Komponenty umieszczone na komponencie `Panel` będą wówczas miały odpowiednio ustawioną właściwość `Parent`. Wówczas:

```
nazwa_komponentu.Parent.Name;
```

wyświetli nazwę komponentu rodzica (właściciela).

Możemy sprawdzić, czy komponent ma właściciela:

```
function HasParent: Boolean; dynamic;
```

Chcąc się odwołać do jednego z komponentów znajdujących się na takim panelu, możemy skorzystać z metody:

```
function FindComponent(const AName: string): TComponent;
```

Jeżeli chcemy, możemy również przejrzeć całą listę komponentów

```
var
  kk : Integer;
begin
  for kk := 0 to ComponentCount-1 do begin
    if Components[kk] is (TButton) then begin
      (Components[kk] as TButton).Font.Size := 12;
    end;
  end;
end;
```

W podanym przykładzie w ramach pętli (na przykład: wewnątrz formularza) przejrzymy wszystkie komponenty znajdujące się na formularzu. Jeżeli komponentem jest komponent klasy `TButton`, wówczas zostanie dla niego zmieniona wielkość aktualnie wykorzystywanej czcionki na rozmiar 12 punktów.

Wiemy już, że komponenty posiadają właściwości i metody. Domyślamy się również, że komponenty potrafią reagować na wystąpienie pewnych zdarzeń. Zakładka *events* (w oknie *Object Inspector*) większości komponentów zawiera bogatą listę zdarzeń, jakie można dla nich oprogramować. Jednocześnie, jak o tym wspomniałem wcześniej,

właściwości widoczne w oknie *Object Inspector* należą również do grupy właściwości (ang. *properties*). Dlatego żeby obsłużyć zdarzenie komponentu `Button`, jakim jest `OnClick`, należy przypisać właściwości tego zdarzenia odpowiednią metodę. Dzięki temu można różnym zdarzeniom przypisać tę samą metodę — nawet dla różnych komponentów.

Podejście tradycyjne

Aby wejść łagodnie w świat projektowania komponentów, zademonstruję, jak na pewnych etapach można osiągnąć pozornie skomplikowany cel, jakim jest utworzenie komponentu. W wielu programach korzystałem z komponentu klasy `TComboBox`, który służył mi m. in. do wyboru miesiąca (w zakresie 1 – 12). Najprostszym rozwiązaniem jest umieszczenie na formularzu komponentu `ComboBox` oraz wstępne wypełnienie jego właściwości `Items` wartościami od 1 do 12 reprezentujących kolejne miesiące. Wobec tego komponentu — nazwijmy go `cbMC` — miałem jeszcze jeden wymóg. Otóż potrzebowałem, aby początku swej pracy komponent na był ustawiony na wartość, jaką posiada aktualny miesiąc. Narzuca się naturalne i proste rozwiązanie, aby skorzystać z klas i odpowiednio oprogramować zachowanie się nowej klasy tak, aby spełniała moje oczekiwania. W tym celu utworzyłem nową klasę dziedziczącą wprost od klasy `TComboBox`. Oryginalny konstruktor klasy `TComboBox` ma postać:

```
constructor create(Aowner: TComponent); override;
```

Jego parametr `AOwner` jest identyfikowany z właścicielem komponentu. Postanowiłem nieco zmodyfikować konstruktor, aby móc na starcie określić właściciela komponentu, jego rodzica oraz położenie lewego górnego rogu, w jakim ma się znaleźć komponent. W efekcie, konstruktor przyjął postać:

```
Type
MyComboBox = class(TComboBox)
public
    constructor Create(AOwner: TComponent;
                    pParent : TWinControl;
                    pLeft, pTop: Integer);
    procedure wypełnij_i_ustaw_mc();
end;
```

Jego implementacja wygląda następująco:

```
constructor MyComboBox.Create(AOwner: TComponent;
                             pParent : TWinControl;
                             pLeft, pTop: Integer);
begin
    inherited Create(AOwner);

    Parent := pParent;
    Left   := pLeft;
    Top    := pTop;
    Width  := 40;

    wypełnij_i_ustaw_mc();
end;
```

Kolejno w ciele konstruktora wykonuje się wywołanie konstruktora klasy bazowej, przypisanie rodzica, ustawienie położenia lewego górnego rogu komponentu oraz wywoływana jest metoda, której zadaniem jest wypełnienie listy komponentu wartościami od 1 do 12 oraz ustawienie właściwości `ItemIndex` (aktywny wiersz komponentu klasy `MyComboBox`) na odpowiadającą numerowi bieżącego miesiąca (listing 10.3):

Listing 10.3. Kod realizujący ustawienie wewnętrznej zawartości kontrolki

```
procedure MyComboBox.Wypełnij_i_ustaw_mc();
var
  mc      : integer;
  Year, Month, Day, Hour, Min, Sec, MSec: Word;
  dzisiaj : TDateTime;
  poz     : Integer;
  seekmc  : Integer;
  ss      : String;

begin
  for mc := 1 to 12 do begin
    Items.Add(IntToStr(mc));
  end;

  dzisiaj := Now();
  DecodeDate(dzisiaj, Year, Month, Day);
  seekmc := month;
  ss     := IntToStr(seekmc);

  poz := Items.IndexOf(ss);

  if poz > -1 then begin
    ItemIndex := poz;
  end;
end;
```

Natomiast, aby użyć mechanizmu w postaci nowo zdefiniowanej klasy, wykonuje wywołanie:

```
procedure TForm1.FormActivate(Sender: TObject);
var
  ob : MyComboBox;
begin
  ob := MyComboBox.Create(Self, Form1, 56, 88);
end;
```

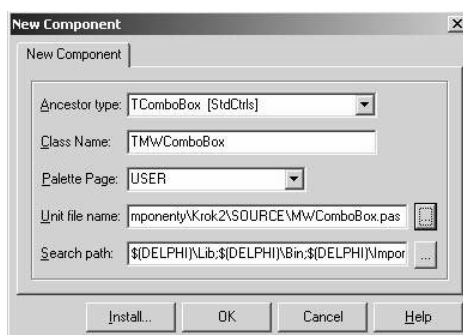
Takie jest tradycyjne podejście. Proste i skuteczne (pełny kod znajduje się w przykładowym projekcie o nazwie `Przyklad28` w c:\`helion\przyklady\komponenty\krok1\source\Przyklad28`). Ale ma swoją wadę. Przejawia się ona głównie tym, że jeżeli na przykład projektujemy okno dialogowe posiadające wiele obiektów kontrolnych w postaci komponentów, takich jak na przykład `Edit`, `MaskEdit`, `DateTimePicker`, `ComboBox` itd., to wygodnie jest albo wszystkie komponenty umieszczać na formularzu ściągając je z palety komponentów, albo wszystkie je tworzymy w locie, jak w przykładzie powyżej. Metoda polegająca na mieszaniu tych dwóch metod jest chyba najgorszym z rozwiązań. Rozwiązaniem najbardziej sensownym jest utworzenie komponentu realizującego te samo zadanie.

Tworzymy pierwszy komponent

Komponenty można tworzyć zupełnie od podstaw lub wykorzystując istniejące klasy wraz z mechanizmami, które klasy te posiadają. W dalszym ciągu postaram się omówić sposób utworzenia prostego komponentu powstałego na bazie komponentu `ComboBox`. Wykonuje on te same zadania, co utworzony w poprzednim punkcie obiekt klasy `MyComboBox` (omawiany przykład znajduje się w przykładowym projekcie o nazwie `Przyklad29` — `c:\helion\przyklady\komponenty\krok1\source\Przyklad29`).

Aby utworzyć nowy komponent, należy po uruchomieniu środowiska Delphi zamknąć ewentualnie otwarty projekt oraz z menu wybrać: *Component/New Component*. Na ekranie zobaczymy okno jak na rysunku 10.1.

Rysunek 10.1.
Okno tworzenia nowego komponentu



W oknie tym mamy do wyboru następujące parametry:

- ♦ *Ancestor Type* — należy wybrać klasę przodka;
- ♦ *Class Name* — należy podać naszą propozycję nazwy dla tworzonej klasy komponentu;
- ♦ *Palette Page* — należy podać nazwę istniejącej (lub nowej) palety, na której ma się znaleźć tworzony komponent;
- ♦ *Unit File Name* — należy podać położenie pliku z kodem źródłowym komponentu;
- ♦ *Search Path* (ścieżka poszukiwań) — najczęściej nie trzeba tutaj nic zmieniać.

Po podaniu wymaganych danych zatwierdzamy nasze dane, wybierając klawisz *OK*.

W efekcie, mechanizm tworzenia nowego komponentu utworzy nowy moduł w miejscu pliku jak w parametrze *Unit File Name*. Bedzie on zawierał definicje klasy o nazwie `ClassName` dziedziczącą od *Ancestor Type*. Oto zawartość pliku (listing 10.4):

Listing 10.4. *Początkowa zawartość modułu*

```
unit MWComboBox;  
interface  
uses  
  SysUtils, Classes, Controls, StdCtrls;
```

```

type
  TMWComboBox = class(TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;
implementation

procedure Register;
begin
  RegisterComponents('USER', [TMWComboBox]);
end;
end.

```

Nie ma tutaj specjalnych niespodzianek. Mamy w zasadzie szkielet klasy i jedną procedurę o nazwie Register. Deklaracja metody Register jest następująca:

```

procedure RegisterComponents(const Page: string;
                             const ComponentClasses: array of TComponentClass);

```

Procedura Register dokona rejestracji komponentu TMWComboBox na karcie o nazwie USER.

Aby tworzony komponent mógł zachowywać się jak jego przodek klasy TComboBox, trzeba do sekcji public dodać konstruktor, za pomocą którego wywołamy konstruktor przodka. Ja rozszerzyłem możliwości konstruktora o możliwość przyjęcia parametrów klasy podstawowej.

```

public
  constructor Create(AOwner: TComponent; pParent : TWinControl);

```

Chciałem również, aby nowy komponent posiadał właściwość Mc, która przechowywałaby aktualną wartość numeryczną miesiąca. W tym celu dodałem właściwość opublikowaną o takiej nazwie do sekcji published:

```

published
  property Mc : TMiesiace read GetMc write SetMc;

```

Nowa właściwość Mc pojawi się w oknie *Object Inspector* z taką samą nazwą. Aby móc ustawiać tę właściwość, trzeba zadeklarować odpowiednią zmienną w sekcji prywatnej. Zmienna ta powinna przyjmować wartości tylko z zakresu od 1 do 12. Aby tak się stało, utworzyłem nowy typ:

```

type
  TMiesiace = 1..12;

```

a w sekcji private podałem:

```

private
  fMc : TMiesiace;

```

Zmienna `fMc` oraz właściwość `Mc` są ze sobą nierozzerwalnie związane. Odczyt i zapis zmiennej jest realizowany za pośrednictwem obiektów wymienionych po słowach `read` i `write`. Proste zmienne można ustawiać i odczytywać za pomocą konstrukcji (dla zmiennej `fMc`):

```
property Mc : TMiesiace read fMc write fMc;
```

Przy czym odczyt (po słowie: „`read`”) odbywałby się ze zmiennej `fMc`, a zapis (po słowie: „`write`”) do tej samej zmiennej `fMc`. W podanym wcześniej fragmencie kodu podałem drugi sposób osiągnięcia tego samego celu:

```
property Mc : TMiesiace read GetMc write SetMc;
```

W tym przypadku odczyt będzie się odbywał za pomocą metody `GetMc`, a ustawianie właściwości `Mc` poprzez metodę `SetMc`. Skoro tak, to trzeba zadeklarować obie metody:

```
public
  procedure SetMc(const Value: TMiesiace);
  function GetMc: TMiesiace;
  procedure Wypełnij_i_ustaw_mc();
```

Do sekcji `public` dodałem również metodę, która wypełni i odpowiednio ustawi listę, z której będzie można wybierać odpowiednie wartości `wypełnij_i_ustaw_mc()`.

Do pełnej funkcjonalności komponentu przydałoby się jeszcze odpowiednie oprogramowanie zdarzeń, jakie zajdą, gdy użytkownik komponentu zmieni wartość miesiąca na inny. W tym celu dodamy obsługę zdarzenia. Oto fragment kodu związany z obsługą zdarzenia:

```
TMWComboBox = class(TComboBox)
private
  fOnChange : TNotifyEvent;
protected
  procedure Zmiana; dynamic;
published
  property OnChange : TNotifyEvent read fOnChange write FOnChange;
end;
```

Została zadeklarowana zmienna `fOnChange` typu `TNotifyEvent`, przy czym ten ostatni ma następującą deklarację:

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

Zdarzenie będzie również właściwością:

```
property OnChange : TNotifyEvent read fOnChange write FOnChange;
```

a odczyt i ustawianie będzie się odbywało poprzez zmienną `fOnChange`.

Istnieje jeszcze metoda:

```
procedure Zmiana; dynamic;
```

którą wywołam na koniec operacji związanych z ustawieniem zmiennej (`SetMc`).

Cały kod przygotowanego komponentu przedstawia się następująco (listing 10.5):

Listing 10.5. *Końcowa postać modułu realizującego obsługę kontrolki*

```

unit TryComboBox;
interface
uses
  SysUtils, Classes, Controls, StdCtrls;

type
  TMiesiace = 1..12;

  TMWComboBox = class(TComboBox)
  private
    fMc      : TMiesiace;
    fOnChange : TNotifyEvent;
  protected
    procedure Zmiana; dynamic;
  public
    procedure SetMc(const Value: TMiesiace);
    function  GetMc: TMiesiace;
    procedure Wypełnij_i_ustaw_mc();

    constructor Create(AOwner: TComponent; pParent : TWinControl);
  published
    property OnChange : TNotifyEvent read fOnChange write FOnChange;
    property Mc      : TMiesiace   read GetMc   write SetMc;
  end;
  procedure Register;

  implementation

  procedure Register;
  begin
    RegisterComponents('USER', [TMWComboBox]);
  end;

  constructor TMWComboBox.Create(AOwner: TComponent; pParent : TWinControl);
  begin
    inherited Create(AOwner);

    Parent := pParent;
    Width  := 40;

    Wypełnij_i_ustaw_mc();
  end;

  procedure TMWComboBox.Zmiana();
  begin
    if Assigned(fOnChange) then begin
      FOnChange(Self);
    end;
  end;

  procedure TMWComboBox.SetMc(const Value: TMiesiace);
  var
    poz : integer;
    ss  : String;

```

```
begin
    fMc := Value;
    ss := IntToStr(Value);

    poz := Items.IndexOf(ss);
    if poz > -1 then begin
        ItemIndex := poz;
    end
    else begin
        ItemIndex := -1;
    end;

    Zmiana();
end;

function TMWComboBox.GetMc(): TMiesiace;
begin
    Result := fMc;
end;

procedure TMWComboBox.Wypełnij_i_ustaw_mc();
var
    mc      : integer;
    Year, Month, Day, Hour, Min, Sec, MSec: Word;
    dzisiaj : TDateTime;
    poz      : Integer;
    seekmc   : Integer;
    ss       : String;

begin
    if not (csDesigning in ComponentState ) then begin
        for mc := 1 to 12 do begin
            Items.Add(IntToStr(mc));
        end;
    end;

    dzisiaj := Now();
    DecodeDate(dzisiaj, Year, Month, Day);
    seekmc := month;
    ss := IntToStr(seekmc);

    poz := Items.IndexOf(ss);

    if poz > -1 then begin
        ItemIndex := poz;
        fMc := Month;
    end;
end;
end.
```

Utworzony kod należy zapamiętać, po czym dobrze by było przetestować działanie powstałego kodu. W systemie pomocy środowiska Delphi pod hasłem *Testing uninstalled components* (zakładka *Znajdź*) znajduje się odpowiedź o tym, jak testować napisany komponent przed jego instalacją w zakładce komponentów. Operację należy wykonać w sześciu krokach.

- 1. Dodajemy do sekcji uses formularza głównego nazwę modułu zawierającego testowany komponent:**

```
Uses TryComboBox;
```

- 2. Dodajemy obiekt do sekcji public:**

```
public  
{ Public declarations }  
Ob : TMWCombobox;
```

- 3. Dodajemy obsługę zdarzenia OnCreate formularza:**

```
type  
TForm1 = class(TForm)  
    procedure FormCreate(Sender: TObject);  
end;
```

- 4. Tworzymy egzemplarz obiektu testowanej klasy:**

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Ob := TMWCombobox.Create(Self, Form1);  
    ...  
end;
```

- 5. Ustawiamy właściwość Parent, jeżeli komponent jest kontrolką. U mnie Parent to drugi parametr wywołania konstruktora (tutaj Form1):**

```
Ob := TMWCombobox.Create(Self, Form1);
```

- 6. Ustawiamy pozostałe (inne) parametry komponentu:**

```
Ob.Left := 56;  
Ob.Top := 88;
```

W następnych paragrafach przyjrzymy się bliżej problematyce związanej z podstawami projektowania komponentów bazodanowych.