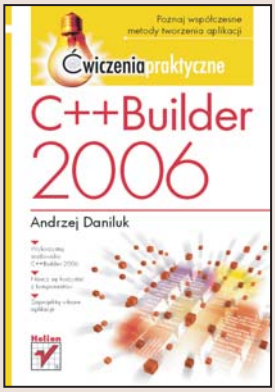
C++Builder 2006. Ćwiczenia praktyczne

Autor: Andrzej Daniluk
ISBN: 83-246-0518-5
Format: A5, stron: 192
Przykłady na ftp: 436 kB



Poznaj świat profesjonalnego programowania

C++Builder 2006 to środowisko programistyczne pozwalające na wizualne tworzenie aplikacji. Dzięki gotowym komponentom programista może skupić się na tym, co najważniejsze — na pisaniu kodu. Za pomocą C++Buildera bez problemu stworzy zarówno niewielką aplikację konsolową, jak i ogromny system informatyczny. Jednak każda wielka podróż, również ta w świat programowania, zaczyna się od małego kroku.

Dzięki książce „C++Builder 2006. Ćwiczenia praktyczne" uczynisz ten właśnie pierwszy krok. Poznasz środowisko C++Builder 2006 i podstawy języka C++. Przeczytasz o programowaniu obiektowym i obsłudze zdarzeń. Wykonując kolejne ćwiczenia, dowiesz się, jak korzystać z udostępnianych komponentów i kontrolować ich parametry. Wykorzystasz również oferowane przez C++Buildera narzędzia i stworzysz własne aplikacje.

- Elementy środowiska C++Builder 2006
- Tworzenie aplikacji konsolowych
- Podstawowe elementy języka C++
- Klasy i obiekty
- Projektowanie formularzy
- Korzystanie z komponentów VCL
- Projektowanie aplikacji z wykorzystaniem elementów biblioteki VCL

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

Spis treści

Środowisko programisty IDE C++Builder 2006

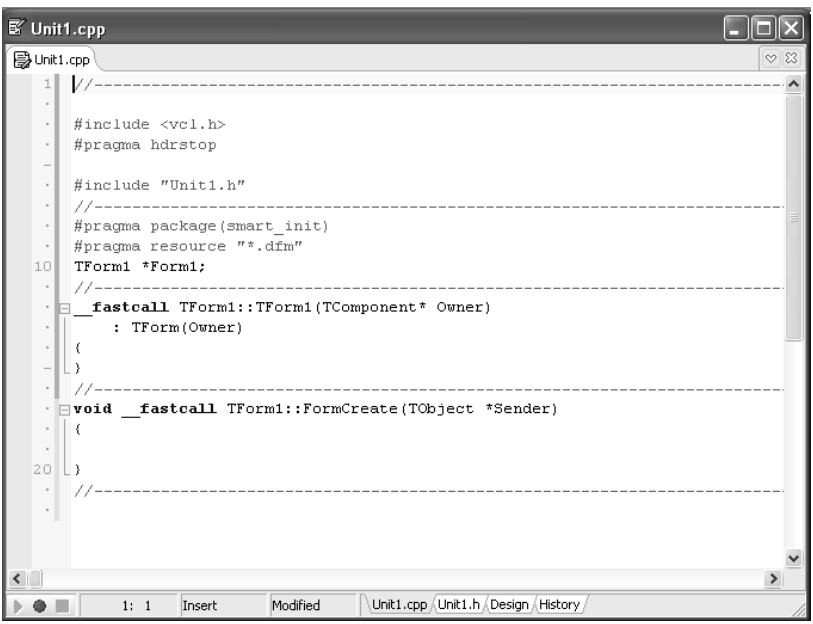
Rozdział ten poświęcony jest omówieniu praktycznych sposobów wykorzystania poznanych wcześniej elementów języka *C++* w graficznym środowisku *C++Builder 2006*. Zapoznamy się tutaj m. in. z pojęciem formularza oraz funkcji obsługi zdarzenia.

Ogólna postać programu środowiska graficznego pisanego w C++Builderze 2006

Formularz (ang. *form*) jest wyświetlanym na ekranie obiektem mogącym składać się z wielu pól, które można wypełniać podobnie jak tradycyjne dokumenty papierowe. Podczas wprowadzania danych do formularza można je poprawiać, ponieważ każde pole formularza zachowuje się jak miniaturowy edytor ekranowy.

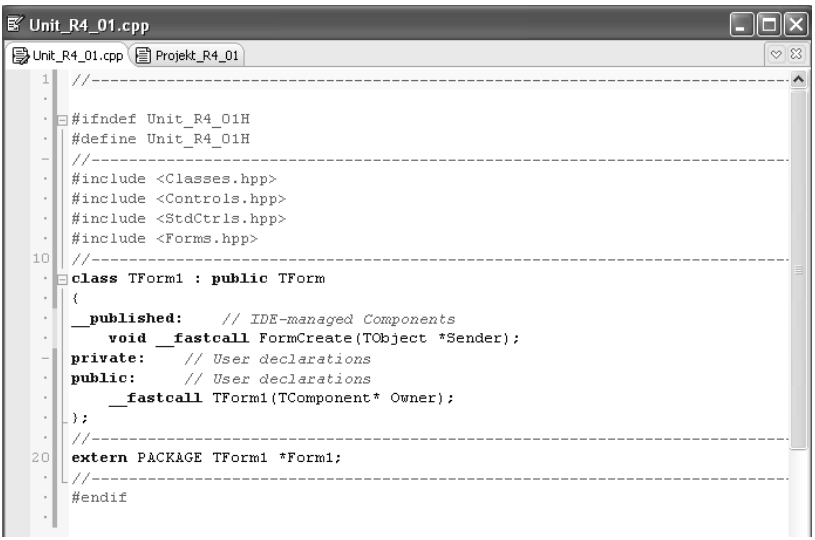
Formularz

Poleceniem menu *File\New\Other…\VCL Forms Application* stwórzmy na pulpicie szablon aplikacji opartej na formularzu. Formularz jest pierwszym obiektem, z którym się spotykamy, rozpoczynając pisanie aplikacji. Po dwukrotnym kliknięciu w obszarze formularza dostajemy się do okna edycji kodu modułu *Unit1.cpp*, który pokazany jest na rysunku 4.1.



Rysunek 4.1. *Okno edycji kodu głównego modułu aplikacji*

Jeżeli moduł tworzonego obecnie projektu za pomocą polecenia *File\Save As…* zapisaliśmy jako *R4\O1\Unit_R4_01.cpp*, to w tym samym katalogu *C++Builder* powinien wygenerować plik nagłówkowy *Unit_R4_01.h*. *C++Builder* oferuje nam bardzo wygodny sposób obejrzenia jego zawartości. Korzystając z zakładki noszącej taką samą nazwę jak plik nagłówkowy, zerknijmy do jego wnętrza. Od razu zauważymy, iż zawiera on definicję klasy naszego formularza (rysunek 4.2).



Rysunek 4.2. *Zawartość pliku nagłówkowego Unit_R4_01.h zawierającego definicję klasy formularza*

Przechodzenie pomiędzy plikiem *.cpp* i powiązanym z nim plikiem nagłówkowym *.h* możliwe jest również poprzez naciśnięcie kombinacji klawiszy *Ctrl+F6*. W celu przywołania formularza używamy zakładki *Design*.

Zdefiniowana klasa `TForm1` dziedziczy własności bazowej klasy formularza `TForm`, natomiast sam formularz, traktowany jako zmienna obiektowa, deklarowany jest jako:

```
TForm1 *Form1;
```

W definicji klasy formularza możemy zauważyć funkcję:

```
void __fastcall FormCreate(TObject *Sender);
```

Builder odpowiednio inicjuje formularz (tylko jeden raz), kiedy jest on tworzony po raz pierwszy. `Sender` jest pewnym wskaźnikiem wskazującym daną typu `TObject`. W rzeczywistości `Sender` reprezentuje pewną właściwość, polegającą na tym, iż każdy obiekt łącznie z formularzem (oraz każdy obiekt *VCL*) musi być w pewien sposób poinformowany o przyszłym przypisaniu mu pewnego zdarzenia (w przypadku formularza zdarzenie to polega na jego inicjalizacji).

TObject jest bezwzględnym przodkiem wszystkich komponentów oraz klas *VCL* i umieszczony jest na samym szczycie hierarchii klas.

Z rysunku 4.2 możemy odczytać, iż standardowa definicja klasy składa się z kilku części. Sekcja `public` służy do deklarowania funkcji i procedur (czyli metod lub operacji) oraz zmiennych (zwanych polami lub atrybutami), które w przyszłości mogą być udostępniane innym. Zasadniczą różnicą pomiędzy metodami a zwykłymi funkcjami czy procedurami jest to, że każda metoda posiada niejawny parametr `this`, wskazujący na obiekt będący przedmiotem wywołania tej metody. Sekcję `public` często nazywamy **interfejsem obiektu formularza**. Sekcja `private` przeznaczona jest dla pól i metod widzianych jedynie wewnątrz klasy.

Oprócz wymienionych elementów, definicja klasy może posiadać jeszcze sekcje `protected` oraz `__published`. W części `protected` można definiować pola i metody widoczne dla macierzystej klasy i klas po niej dziedziczących. Deklaracje zawarte w sekcji `__published` (publikowanej) pełnią taką samą rolę, jak deklaracje umieszczone w sekcji `public` (publicznej). Różnica pomiędzy nimi polega na tym, iż te pierwsze nie tworzą tzw. informacji czasu wykonania. Do zagadnień tych powrócimy w dalszej części Ćwiczeń.

Własności

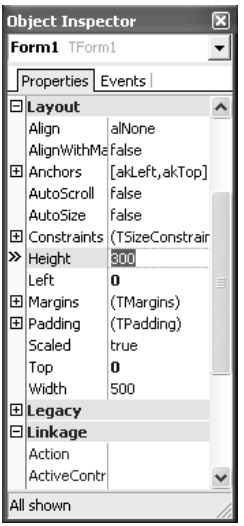
Własności pozwalają użytkownikowi na uzyskiwanie dostępu do elementów komponentów biblioteki *VCL* oraz na modyfikację niektórych ich atrybutów.

ĆWICZENIE

4.1 Poznawanie Inspektora obiektów — podstawowe operacje na formularzu

Kliknijmy zakładkę *Design* i przejdźmy do *Inspektora obiektów*. Rozmiary formularza ustalimy, korzystając z jego własności `Height` (wysokość) i `Width` (szerokość), znajdujących się w karcie właściwości (*Properties*) *Inspektora obiektów*, w zakładce *Layout* (rysunek 4.3). Jeżeli chcemy, aby po uruchomieniu formularz nie „rozpływał" się po ekranie w odpowiedzi na kliknięcie pola maksymalizacji, w *Inspektorze*

Rysunek 4.3. *Zakładka Layout Inspektora obiektów*



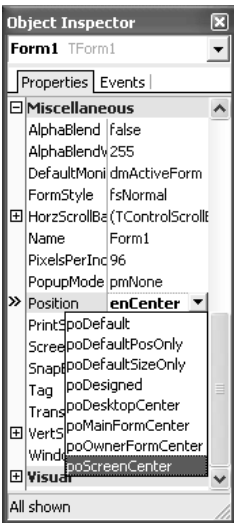
obiektów rozwińmy własność `Constraints` (ograniczenie) i we właściwe miejsca `MaxHight` oraz `MaxWidth` wpiszmy żądane rozmiary formularza (w pikselach).

Przejdźmy następnie do własności `Position` (zakładka *Miscellaneous*) i wybierzmy `poScreenCenter` (rysunek 4.4). Wybrane przypisanie spowoduje, że w momencie uruchomienia aplikacji formularz pozostanie w centrum ekranu (ale nie pulpitu `poDesktopCenter`) — jeżeli oczywiście w *Inspektorze obiektów*, w zakładce *Layout*, własności `Align` (zakotwiczenie) nie ustawiliśmy inaczej niż w pozycji `alNone` (patrz rysunek 4.3).

ĆWICZENIE

4.2 Alternatywny sposób ustalania położenia formularza działającej aplikacji

Warto pamiętać, iż graficzny interfejs użytkownika *C++Buildera 2006* został wyposażony w element umożliwiający określenie (w czasie projektowania) położenia na ekranie formularza uruchomionej aplikacji, bez konieczności posługiwania się *Inspektorem obiektów*. Mianowicie, w prawym dolnym rogu centralnej części GUI znajduje się niewielki piktogram *Form Screen Position* (rysunek 4.5), za którego pomocą można w przybliżeniu ustalić położenie formularza działającej aplikacji.



Rysunek 4.4. *Zakładka Miscellaneous*

Rysunek 4.5. *Określanie położenia formularza uruchomionego programu*

Zdarzenia

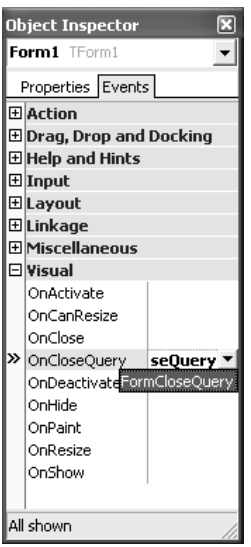
Zdarzenia (ang. *event*) powodują występowanie zmian stanu obiektu i są źródłem odpowiednich komunikatów, przekazywanych do aplikacji lub bezpośrednio do systemu. Reakcja obiektu na wystąpienie zdarzenia udostępniana jest aplikacji poprzez funkcję obsługi zdarzeń (ang. *event function*) będącą wydzieloną częścią kodu. Rolę zdarzeń w aplikacji najlepiej jest prześledzić, wykonując praktyczne ćwiczenie.

ĆWICZENIE

4.3 Programowanie zdarzenia OnClose

Może również zdarzyć się sytuacja, w której zechcemy zamknąć formularz, korzystając bezpośrednio z jego pola zamknięcia. Aby mieć pewność, że w momencie zamknięcia aplikacji wszystkie jej zasoby zostaną prawidłowo zwolnione, skorzystamy ze zdarzenia `OnClose`.

W celu zaprogramowania obsługi wybranego zdarzenia przejdźmy do karty zdarzeń (*Events*) *Inspektora obiektów*. Zdarzenie `OnClose` określmy jako `FormClose` (rysunek 4.6) i potwierdźmy klawiszem *Enter* lub podwójnym kliknięciem myszy.



Rysunek 4.6. *Zakładka Visual w karcie zdarzeń Inspektora obiektów*

W ten sposób *Builder* automatycznie wygeneruje funkcję obsługi zdarzenia:

```
void __fastcall TForm1::FormClose(TObject *Sender,
                                  TCloseAction &Action)
```

Zmiennej `Action` (akcja) można przypisać jeden z elementów typu wyliczeniowego:

```
enum TCloseAction {caNone, caHide, caFree, caMinimize};
```

gdzie:

- `caNone` oznacza, że formularz nie zostanie zamknięty;
- `caHide` oznacza, że formularz nie zostanie zamknięty, lecz ukryty;
- `caFree` oznacza, że formularz zostanie zamknięty z jednoczesnym zwolnieniem wszystkich zasobów pamięci, z których aktualnie korzysta;
- `caMinimize` oznacza, że formularz zostanie zminimalizowany.

Funkcję obsługi zdarzenia `FormClose()` wypełnimy następującym kodem:

```
//---------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                                  TCloseAction &Action)
{
  switch (MessageBox(0, "Zamknięcie aplikacji ?","Uwaga",
      MB_YESNOCANCEL || MB_ICONQUESTION)) {
    case ID_YES:
      Action = caFree;
    break;
    case ID_CANCEL:
      Action = caNone;
    break;
  }

}
//---------------------------------------------------------
```

Użycie w deklaracji funkcji konwencji `__fastcall` powoduje, że trzy pierwsze parametry funkcji mogą być umieszczane w rejestrach procesora. Rejestry nie będą używane, jeżeli parametrami funkcji będą dane zmiennopozycyjne lub struktury. Parametry tego typu odkładane są na stosie.

ĆWICZENIE

4.4 Programowanie zdarzenia OnCloseQuery

Odmianą `OnClose` jest zdarzenie `OnCloseQuery`, które tworzymy, również korzystając z karty zdarzeń *Inspektora obiektów* (rysunek 4.6).

Funkcję obsługi zdarzenia `FormCloseQuery()` wypełnimy następującym kodem:

```
//---------------------------------------------------------
void __fastcall TForm1::FormCloseQuery(TObject *Sender,
                 bool &CanClose)
{
  TMsgDlgButtons przyciski;
  przyciski <<mbYes <<mbNo <<mbCancel;
  AnsiString bufor = "Zakończyć działanie programu ?";
  switch(MessageDlg(bufor, mtConfirmation,przyciski,0)) {
   case mrYes:
      CanClose = true;
   break;
```

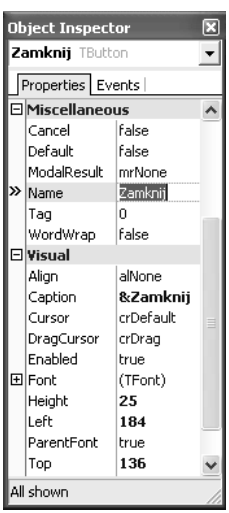
```
      case mrNo:
         CanClose = false;
      break;
      case mrCancel:
         CanClose = false;
      break;
     }
  }
  //---------------------------------------------------------
```

Należy pamiętać, iż jednoczesne używanie w programie dwóch zdarzeń typu `OnClose` nie jest celowe.

ĆWICZENIE

4.5 Programowanie zdarzeń dla komponentów umieszczanych na formularzu

Na tak przygotowanym formularzu umieśćmy jeden komponent reprezentujący klasę `TButton` z zakładki *Standard Palety narzędzi*. Korzystając z *Inspektora obiektów* oraz z karty własności, cechy `Name` (zakładka *Miscellaneous*) oraz `Caption` (zakładka *Visual*) przycisku `Button1` zmieńmy odpowiednio na `Zamknij` oraz `&Zamknij` (rysunek 4.7). Jeżeli jawnie nie zmienimy własności `Name` komponentu, w kodzie będzie występować jego nazwa domyślna (np. `Button1`).



Rysunek 4.7. *Zakładki Miscellaneous oraz Visual karty własności Inspektora obiektów*

Znak &, który występuje w opisie przycisku (ale nie w jego nazwie), spowoduje, że litera występująca bezpośrednio za nim stanowić będzie klawisz szybkiego dostępu do funkcji obsługi wybranego zdarzenia.

Dla naszego przycisku utworzymy funkcję obsługi odpowiedniego zdarzenia. Klikając dwukrotnie przycisk *Zamknij* lub w widoku struktury obiektów (*Structure*) odpowiednio oznaczony komponent, dostaniemy się do wnętrza właściwej funkcji obsługi zdarzenia:

```
void __fastcall TForm1:: ZamknijClick(TObject *Sender)
```

Już w tym miejscu możemy zauważyć, iż w definicji klasy *Builder* wygenerował automatycznie deklarację przycisku oraz deklarację funkcji obsługującego go zdarzenia.

Należy zawsze pamiętać, iż szkielety funkcji obsługi odpowiednich zdarzeń, takich jak `ZamknijClick()`, zostaną automatycznie wygenerowane przez *Buildera* w odpowiedzi na dwukrotne kliknięcie danego przycisku. W żadnym wypadku funkcji tych nie należy wpisywać samodzielnie.

Omawianą funkcję obsługi zdarzenia wypełnimy przedstawionym poniżej kodem, co spowoduje, że po naciśnięciu wybranego przycisku aplikacja zostanie zamknięta.

```
//---------------------------------------------------------
void __fastcall TForm1:: ZamknijClick(TObject *Sender)
{
   Application->Terminate();
}
//---------------------------------------------------------
```

Hierarchia własności komponentów VCL

Każdy komponent wykorzystywany w aplikacji posiada dwie podstawowe własności: może być właścicielem (ang. *Owner*) innych komponentów lub może być ich rodzicem (ang. *Parent*). Istnieje subtelna różnica pomiędzy tymi dwoma pojęciami, z której należy zdawać sobie sprawę, jeżeli chcemy zrozumieć idee rządzące zasadami programowania obiektowo-zdarzeniowego.

Wykorzystując graficzny interfejs użytkownika *GUI* (ang. *Graphical User Interface*), budujemy aplikacje, których głównym elementem jest formularz. Formularz jest właścicielem komponentów, które na nim umieszczamy. Jako przykład rozpatrzmy komponent `CheckBox1`, reprezentujący klasę `TCheckBox` i znajdujący się w obszarze określonym przez reprezentanta klasy `TGroupBox`. Jeżeli zechcemy teraz dowolnie zmienić położenie `CheckBox1`, napotkamy pewne trudności — nie będziemy mogli przesunąć go poza `GroupBox1`. Mówimy, że `GroupBox1`, czyli reprezentant klasy `TGroupBox`, stał się rodzicem dla `CheckBox1` reprezentującego klasę `TCheckBox`. Aby przeanalizować przykład hierarchii własności, można sprawdzić, kto jest właścicielem formularza. W tym celu wystarczy zaprojektować nowe zdarzenie (lub wykorzystać istniejące) oraz odpowiednio wykorzystać funkcję `ClassName()` zwracającą łańcuch znaków określający nazwę odpowiedniego egzemplarza klasy.

Pisząc:

```
ShowMessage(Form1->Owner->ClassName());
```

przekonamy się, że właścicielem formularza jest aplikacja. Jeżeli natomiast chcielibyśmy sprawdzić w ten sposób, czy formularz ma rodzica, wygenerujemy po prostu wyjątek. Formularz w prostej linii nie posiada rodzica. Następnie napiszmy:

```
ShowMessage(GroupBox1->Owner->ClassName());
ShowMessage(GroupBox1->Parent->ClassName());
```

Pojawiający się komunikat nie pozostawia cienia wątpliwości: zarówno właścicielem, jak i rodzicem komponentu `GroupBox1` umieszczonego bezpośrednio na formularzu jest `TForm1`. Przechodząc dalej, sprawdzimy cechy własności komponentu `CheckBox1`:

```
ShowMessage(CheckBox1->Parent->ClassName());
```

Stwierdzimy, że jego rodzicem jest klasa `TGroupBox`, zaś właścicielem:

```
ShowMessage(CheckBox1->Owner->ClassName());
```

Pozostanie dalej formularz, czyli `TForm1`.

Zdarzają się sytuacje, kiedy potrzebujemy, nawet w trakcie działania aplikacji, zmienić położenie jakiegoś komponentu umieszczonego uprzednio w obszarze takim jak `TGroupBox` czy `TPanel`. Aby to zrobić, wystarczy pamiętać o omówionych relacjach własności. Jeżeli chcemy, by np. `CheckBox1` znalazł się bezpośrednio w innym miejscu formularza, wystarczy przypisać mu `Form1` jako rodzica, a następnie podać nowe współrzędne:

```
CheckBox1->Parent = Form1;
CheckBox1->Top = 20;
CheckBox1->Left = 50;
```

Należy rozróżniać pojęcia właściciela (`Owner`) i rodzica (`Parent`). Rodzic nie jest tożsamy z właścicielem. Właściciela określa się tylko raz, podczas wywoływania jego konstruktora, i nie można już go zmienić bez zniszczenia obiektu. Rodzica obiektu możemy natomiast zmienić zawsze.

Dynamiczne tworzenie komponentów zarejestrowanych w bibliotece VCL

Formularz w momencie zamknięcia automatycznie (dzięki pewnym mechanizmom) niszczy umieszczone w jego obrębie komponenty. Ten sposób zwalniania komponentów nie ma oczywiście zastosowania w przypadku, gdy obiekty tworzymy dynamicznie za pomocą ich konstruktorów i operatora `new`. W takiej sytuacji obowiązkiem programisty jest jawne zwolnienie odpowiedniego wskaźnika za pomocą operatora `delete`.

Treść Ćwiczenia 4.6 obrazuje jeden ze sposobów samodzielnego tworzenia deklaracji obiektu oraz funkcji obsługi generowanego przez ten obiekt zdarzenia.

Jeżeli zdecydujemy się tworzyć komponenty dynamicznie, to wskaźniki do klas, w których są one wyrażane, nie mogą być umieszczane w sekcji `__published` deklaracji klasy formularza. Nie można też korzystać z usług *Inspektora obiektów*. Wszystkie własności i zdarzenia należy programować samodzielnie.

ĆWICZENIE

4.6 Dynamiczne umieszczanie komponentów na formularzu

Na listingach 4.1 oraz 4.2 zaprezentowano jeden ze sposobów dynamicznego konstruowania wraz z dołączaniem przykładowego zdarzenia `OnClick` do dynamicznie tworzonego obiektu `myButton` klasy `TButton`. Przycisk tworzony jest w ciele konstruktora klasy `TForm1`.

Listing 4.1. *Klasa TForm1 z deklaracją obiektu myButton z klasy TButton oraz funkcją obsługi zdarzenia myButtonClick()*

```
#ifndef Unit_R4_5H
#define Unit_R4_5H
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
   void __fastcall FormClose(TObject *Sender,
                             TCloseAction &Action);
   void __fastcall FormDestroy(TObject *Sender);
private:        // User declarations
   TButton *myButton;
   void __fastcall myButtonClick(TObject *Sender);
public:         // User declarations
   __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Listing 4.2. *Implementacja komponentu oraz funkcji obsługi generowanego przez ten komponent zdarzenia*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit_R4_5.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
{
  // utworzenie obiektu myButton
  myButton = new TButton(Form1);
  myButton->Parent = Form1;
  // opisanie myButton na formularzu
  myButton->Caption = "myButton";
  myButton->Top = 100;
  myButton->Left = 100;
  // przypisanie zdarzeniu OnClick odpowiedniej funkcji
  // obsługi zdarzenia
  myButton->OnClick = myButtonClick;
```

```
}
//---------------------------------------------------------
void __fastcall TForm1::myButtonClick(TObject *Sender)
{
  // ciało funkcji obsługi zdarzenia
  ShowMessage("Przycisk stworzony dynamicznie"
           " wywołał funkcję obsługi zdarzenia");
}
//---------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                 TCloseAction &Action)
{
  switch (MessageBox(0, "Zamknięcie aplikacji ?","Uwaga",
      MB_YESNOCANCEL || MB_ICONQUESTION)) {
    case ID_YES:
      Action = caFree;
    break;
    case ID_CANCEL:
      Action = caNone;
    break;
  }
}
//---------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
   // usunięcie obiektu myButton w momencie zamknięcia
   // formularza
   delete myButton;
}
//---------------------------------------------------------
```

W momencie niszczenia formularza wywoływana jest funkcja obsługi zdarzenia `FormDestroy()`, w której obiekt reprezentujący przycisk jest automatycznie niszczony. Zdarzenie `OnDestroy` dostępne jest w karcie zdarzeń *Inspektora obiektów*, w zakładce *Miscellaneous*.

ĆWICZENIE

4.7 Wielokrotne dynamiczne tworzenie i usuwanie komponentów

W poprzednim ćwiczeniu omówiono jedną z metod dynamicznego tworzenia komponentów. Komponent tworzony był w konstruktorze klasy formularza (wywoływanego tylko raz), dlatego też w takich sytuacjach nie mamy możliwości wielokrotnego tworzenia i niszczenia

wybranego komponentu. Na listingach 4.3 i 4.4 zaprezentowano jeden ze sposobów wielokrotnego tworzenia i niszczenia komponentu w trakcie działania aplikacji. Nowy komponent edycyjny tworzony jest w funkcji obsługi zdarzenia generowanego przez przycisk umieszczony na formularzu w sposób standardowy.

Listing 4.3. *Klasa TForm1 z deklaracją obiektu myEdit z klasy TEdit oraz funkcją obsługi zdarzenia myEditChange()*

```
#ifndef Unit_R4_6H
#define Unit_R4_6H
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
   TButton *Button1;
   TButton *Button2;
   void __fastcall FormClose(TObject *Sender,
                TCloseAction &Action);
   void __fastcall FormDestroy(TObject *Sender);
   void __fastcall Button1Click(TObject *Sender);
   void __fastcall Button2Click(TObject *Sender);
private:        // User declarations
   TEdit *myEdit;
   void __fastcall myEditChange(TObject *Sender);
public:         // User declarations
   __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Listing 4.4. *Implementacja komponentu oraz funkcji obsługi generowanego przezeń zdarzenia*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit_R4_6.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  if(myEdit == NULL) {
    // utworzenie obiektu myEdit
    myEdit = new TEdit(Form1);
    myEdit->Parent = Form1;
    // opisanie myEdit na formularzu
    myEdit->Width = 300;
    myEdit->Text = "Nowy komponent edycyjny. Wpisz tekst...";
    myEdit->Top = 100;
    myEdit->Left = 100;
    // przypisanie zdarzeniu OnChange odpowiedniej funkcji
    // obsługi zdarzenia
    myEdit->OnChange = myEditChange;
  }
}
//---------------------------------------------------------
void __fastcall TForm1::myEditChange(TObject *Sender)
{
  Button2->Caption = " Naciśnij aby zniszczyć komponent Edit";
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    // niszczy komponent
    delete myEdit;
    // przypisuje myEdit wskaźnik pusty
    myEdit = NULL;
    // zmienia opis komponentu Button2
    Button2->Caption = "Button2";
}
//---------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                 TCloseAction &Action)
{
  switch (MessageBox(0, "Zamknięcie aplikacji ?","Uwaga",
      MB_YESNOCANCEL || MB_ICONQUESTION)) {
    case ID_YES:
      Action = caFree;
    break;
    case ID_CANCEL:
      Action = caNone;
    break;
  }
}
```

```
//---------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
   // usunięcie obiektu myEdit w momencie zamknięcia
   // formularza
   if(myEdit != NULL)
    delete myEdit;
}
//---------------------------------------------------------
```