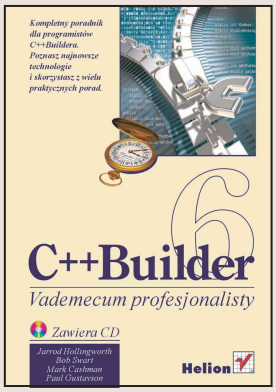
C++ Builder 6. Vademecum Profesjonalisty

Autorzy: Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson
Tłumaczenie: Jarosław Dobrzański, Rafał Jońca, Rafał Szpoton
ISBN: 83-7361-151-7
Tytuł oryginału: C++Builder 6 Developer's Guide
Format: B5, stron: 944



C++ Builder to zgodne ze standardem ANSI C++ środowisko szybkiego tworzenia aplikacji (RAD) w systemie Windows. Dzięki C++ Builder można tworzyć zarówno aplikacje typu „desktop", jak również aplikacje rozproszone i internetowe. „C++ Builder 6. Vademecum profesjonalisty" dostarczy Ci aktualnych informacji na temat najnowszych możliwości tego środowiska. Poznasz między innymi technologie DataSnap, C++ Mobile, XML, BizSnap, dbExpress.

Książka ta była pisana z myślą o aktualnych i przyszłych użytkownikach środowiska C++ Builder. Jest to poradnik dla programistów, a jego podstawowym zadaniem jest poszerzenie wiedzy na temat środowiska C++ Builder i związanych z nim technologii; opis najnowszych funkcji wprowadzonych w wersji 6 środowiska oraz ułatwienie tworzenia wydajnego i użytecznego oprogramowania. Choć większość rozdziałów adresowana jest do średnio zaawansowanych i zaawansowanych użytkowników, zostały one ułożone w taki sposób, że stopień trudności rośnie wraz z numerami rozdziałów, więc początkujący programiści nie powinni mieć większych problemów ze zrozumieniem opisywanych koncepcji.

- Naucz się tworzyć mobilne aplikacje korzystając z Borland C++Mobile Edition
- Poznaj sposoby przetwarzania i transformacji dokumentów XML
- Twórz usługi sieciowe wykorzystując BizSnap oraz WSDL i SOAP
- Pisz aplikacje rozproszone za pomocą DataSnap
- Uzyskaj dostęp do baz danych niezależnie od platformy wykorzystując dbExpress
- Poznaj sztuczki i chwyty stosowane przy pisaniu aplikacji graficznych i multimedialnych
- Poszerz możliwości środowiska programistycznego za pomocą OpenToolsAPI
- Poznaj tajniki Windows 32 API i wykorzystaj je w swoich aplikacjach

Doświadczenie i bogata wiedza autorów „C++Builder 6. Vademecum profesjonalisty" to gwarancja rzetelności tej książki. Jeśli programujesz w C++ Builder, jest Ci ona po prostu niezbędna.

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

Rzut oka na książkę

Spis treści

Rozdział 4.

Tworzenie własnych komponentów

Mark Cashman

W tym rozdziale:

- Tworzenie, kompilacja i instalacja pakietów
- Tworzenie własnych komponentów
- Mechanizm strumieniowania
- Dystrybucja komponentów

Ten rozdział omawia tworzenie i dystrybucję własnych komponentów. Komponenty C++Buildera to często pierwsze elementy umieszczane w aplikacji, ale jeśli zajmujemy się tylko projektowaniem, zapewne znamy sytuacje, w których utworzenie komponentów bazujących na innych komponentach prowadzi w dalszej perspektywie do oszczędności czasu i pieniędzy. Gdy zajmujemy się dużym projektem lub gdy udostępniamy jakąś funkcję wielu odbiorcom (na przykład jako sprzedawany lub udostępniany za darmo pakiet), tworzenie własnych komponentów to podstawa naszej działalności.

Tworzenie, kompilacja i instalacja pakietów

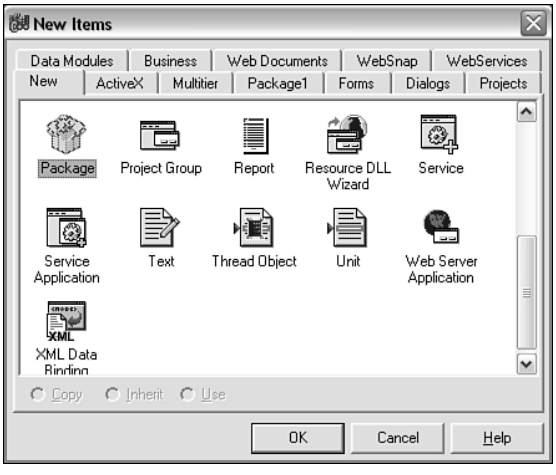
Możemy tworzyć trzy różne rodzaje komponentów: tylko do projektowania, tylko do uruchamiania i pakiety podwójne (do projektowania i uruchamiania). Do dystrybucji najlepiej wykonać dwa osobne pakiety: tylko do projektowania i tylko do uruchamiania. Gdy jednak ciągle udoskonalamy lub testujemy pakiet, najlepiej sprawdza się wersja podwójna.

Poza decyzją o strukturze pakietu należy zastosować sensowną konwencję nazw zarówno dla jednostek kompilacji wewnątrz pakietów, jak i dla samych komponentów. Poza tym musimy określić, w jakich wersjach kompilatora mogą działać komponenty.

Tworzenie pakietu komponentów

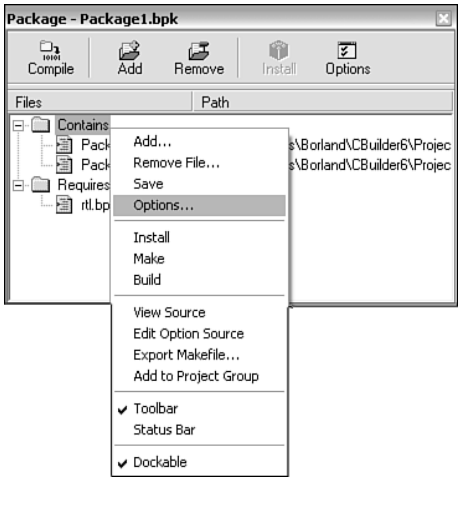
Przygotowanie do tworzenia komponentu lub komponentów wymaga wykreowania pakietu, w którym będą one kompilowane. Musimy utworzyć grupę projektu, aby przechowywać w nim pakiet, ponieważ każdy nowy pakiet zostanie automatycznie umieszczony w aktualnej grupie projektu, a przeważnie nie chcemy, aby pakiet stanowił część projektu aplikacji, nad którą właśnie pracujemy. Nową grupę projektu generujemy, wybierając polecenie *File/New* z menu, a następnie klikając ikonę *Project Group* z zakładki *New* okna, które się pojawi. Następnie jeszcze raz wybieramy polecenie *File/New*, ale tym razem klikamy ikonę *Package*.

Rysunek 4.1 przedstawia okno dialogowe, które pojawia się po wybraniu polecenia *File/New* z menu.



Rysunek 4.1. *Elementy okna wyświetlanego po wybraniu File/New z menu*

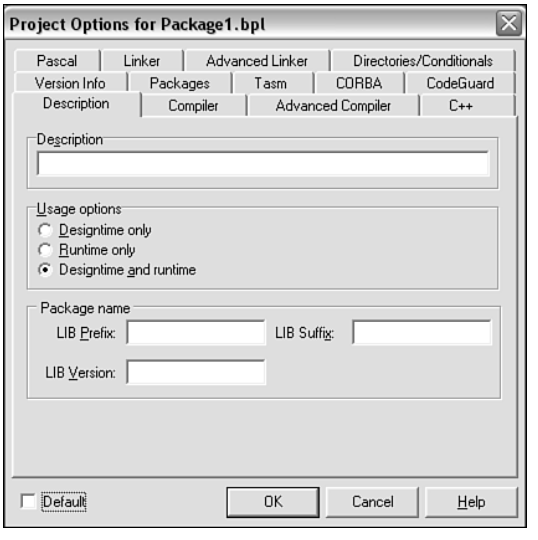
Następnie określamy rodzaj tworzonego pakietu. W tym celu wybieramy polecenie *Options* z menu podręcznego okna pakietu (patrz rysunek 4.2).



Rysunek 4.2. *Przejście do opcji rodzaju pakietu*

Spowoduje to otworzenie okna dialogowego, w którym będziemy mogli określić typ pakietu (patrz rysunek 4.3).

Rysunek 4.3. *Ustawianie typu pakietu*

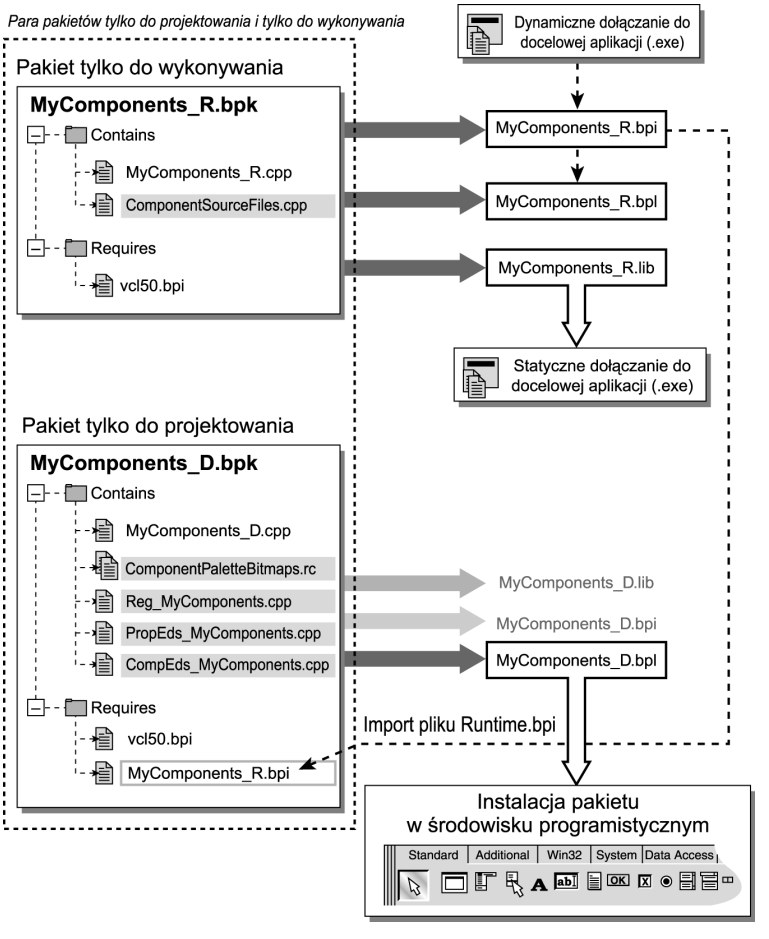


W trakcie prac nad pakietem najlepiej korzystać z pakietu podwójnego, projektowo-uruchomieniowego. Gdy jednak pakiet został już utworzony i przystępujemy do jego dystrybucji, umieszczamy kod w osobnych pakietach dotyczących projektowania i wykonywania. Takie podejście jest odpowiednie, gdy mamy do czynienia ze specjalnymi edytorami właściwości lub innymi funkcjami udostępnianymi osobom korzystającym z komponentu (patrz podrozdział o dystrybucji komponentów i rozdział 5., „Edytory komponentów i edytory właściwości").

Kiedy jesteśmy usatysfakcjonowani działaniem komponentów, powinniśmy zapewnić im poprawne umieszczenie w pakiecie. Najodpowiedniejsze podejście polega na rozdzieleniu pakietów projektowych od wykonywanych. Poprawne wykonanie tego zadania wymaga utworzenia przynajmniej dwóch pakietów: pakietu tylko do wykonywania i pakietu tylko do projektowania — czyli pary pakietów. Oznacza to wykonanie dwóch czynności.

1. Utworzenie pakietu wykonywania zawierającego *tylko* kod źródłowy komponentów. Nie umieszczamy tutaj żadnego kodu rejestrującego komponenty ani kodu odpowiadającego za interfejs komponentu w trakcie projektowania (na przykład kodu edytorów właściwości lub edytorów komponentów).
2. Utworzenie pakietu projektowego zawierającego *tylko* kod rejestracji i ewentualny kod odpowiadający za interfejs komponentu w trakcie projektowania. Nie umieszczamy tu żadnego kodu źródłowego komponentów, ale dodajemy bibliotekę importu (plik *.bpi*) z pakietu wykonywania do listy *Requires* pakietu.

Pakiet projektowy to pakiet instalowany w środowisku projektowym. Rysunek 4.4 obrazuje związek obydwu pakietów.

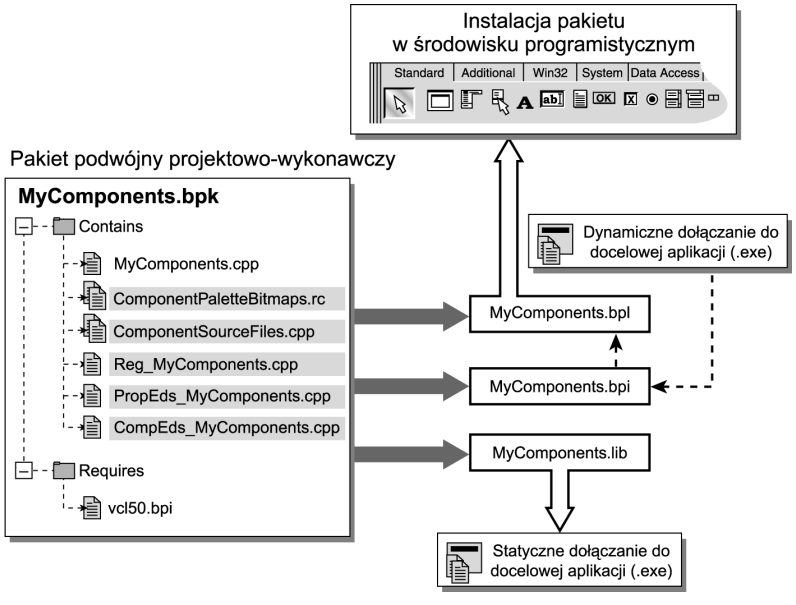


Rysunek 4.4. *Para pakietów tylko do projektowania i tylko do wykonywania*

Zauważmy, że na rysunku 4.4 pakiet projektowy wykorzystywany jest tylko do wygenerowania pliku *.bpl* używanego do instalacji pakietu w środowisku projektowym. Pakiety wykonywania wykorzystują już same aplikacje. W rzeczywistości jeden lub kilka pakietów wykonywania wykorzystywanych jest w połączeniu z pojedynczym pakietem projektowym (innymi słowy, pojawiają się one w sekcji *Requires* pakietu projektowego). Cały kod poza rejestrującym może zostać usunięty z pakietu projektowego, co powoduje, że taki pakiet służy jedynie do rejestracji komponentów.

Dla porównania pakiet podwójny (projektowy i wykonywania) przedstawia rysunek 4.5.

Kod wymagany do rejestracji oraz kod edytorów właściwości i komponentów zostanie niepotrzebnie dołączony do każdej aplikacji wykorzystującej pakiet podwójny, co wyraźnie obrazuje rysunek 4.5. W przypadku prostych pakietów bez edytorów komponentów i właściwości można to zaakceptować. Testowanie komponentów w pakiecie podwójnym jest na ogół wygodniejsze, ale istnieje możliwość uproszczenia przy stosowaniau osobnych pakietów wykonywania i projektowego, gdy umieścimy je w tej samej grupie programu. Należy pamiętać o zapisaniu grupy programu i projektów pakietu — na ogół w katalogu, w którym znajduje się kod komponentów.



Rysunek 4.5. *Pakiet podwójny projektowo-wykonywalny*

Pakiety podwójne niepotrzebnie zwiększają złożoność w trakcie projektowania.

Pierwszym tworzonym pakietem będzie pakiet wykonywania. Aby określić, że pakiet jest pakietem wykonywania, zaznaczamy opcję *Runtime Only* z zakładki *Description* okna dialogowego *Options*. Pakiet ten powinien zawierać tylko kod źródłowy komponentów.

Biblioteki importu (pliki *.bpi*) wymagane przez komponenty należy dodać do listy *Requires* projektu pakietu. Umieszczamy tam tylko te biblioteki, które są niezbędne do poprawnego wygenerowania pakietu. Pamiętajmy, że pakiety wymienione na liście *Requires* są dołączane w trakcie kompilacji do wszystkich aplikacji korzystających z tego pakietu i jednej lub kilku jego jednostek kompilacji.

Po udanym zbudowaniu pakietu wykonywania uzyskujemy trzy pliki (jeśli opcja *Generate .lib File* z zakładki *Linker* okna dialogowego opcji jest wyłączona, nie jest generowany plik *.lib*): *.bpl*, *.bpi* i *.lib*. Upewnijmy się, że zostały utworzone wszystkie trzy, ponieważ są one nam potrzebne. Plik *.lib* nie zawsze jest wymagany, ale powinien być dostępny dla tych, którzy zamierzają statycznie dołączać komponenty do aplikacji (wielu programistów preferuje uproszczoną dystrybucję wykorzystującą aplikacje ze statycznie dołączonymi bibliotekami).

Po utworzeniu pakietu wykonywania, który wygenerował odpowiedni plik importu, możemy rozpocząć pracę nad pakietem projektowym wykorzystującym wcześniejszy pakiet (jego komponenty). W pakiecie tym znajdzie się kod rejestracji, funkcje specjalne, a także edytory komponentów i właściwości wymagane przez komponenty. Sekcja *Requires* zawiera bibliotekę importu pakietu wykonywania. Jeśli pakiet jest na tyle prosty, że nie wymaga dodatkowych edytorów, musimy napisać tylko kod rejestracji (tworzenie edytorów właściwości i komponentów omawia rozdział 5.).

Kompilacja i instalacja pakietów

Projekty pakietów kompiluje się w tradycyjny sposób, generowane jest też typowe wyjście z kompilatora i konsolidatora. Pakiety to w zasadzie zmodyfikowane biblioteki DLL systemu Windows lub biblioteki dynamiczne systemu Linux.

Pakiet instaluje się w środowisku programistycznym na dwa sposoby. Pierwszy to kompilacja pakietu z wpisem *Install* z menu podręcznego projektu pakietu. Drugi sposób polega na tradycyjnej kompilacji, a następnie instalacji pakietu poleceniem *Install* z menu — pakiet zostanie dodany bez rekompilacji.

Istnieje jeszcze trzecia droga, z której na ogół będą korzystali użytkownicy pakietu. Polega ona na wybraniu polecenia *Install Packages* z menu *IDE Components* środowiska programistycznego.

Biblioteka VCL to bardzo użyteczne narzędzie, ponieważ wykonanie aplikacji z zapewnianych przez nią komponentów, klas i metod jest bardzo proste. Czasem jednak okazuje się, że komponenty te nie zapewniają tego, co jest nam potrzebne. To właśnie możliwość pisania i modyfikacji komponentów daje przewagę temu językowi programowania i zapewnia, że C++Builder jest środowiskiem często wykorzystywanym przez programistów na całym świecie. Tworzenie własnych komponentów pozwala poznać działanie VCL i zwiększyć produktywność C++Buildera. Poza tym warto zmierzyć się z komercyjnymi komponentami oferowanymi przez wiele stron internetowych.

Tworzenie własnych komponentów

Początkowo zadanie tworzenia własnych komponentów wydaje się trudne. Po przeczytaniu kilku artykułów i ćwiczeń na ten temat zapewne każdy zastanawia się, od czego zacząć. Najprościej wykorzystać już istniejący komponent, dodając do niego nowe funkcje.

Choć wydaje się to oczywiste, możemy po prostu dostosowywać lub rozszerzać standardowe komponenty VCL, aby sprostały one wymaganiom stawianym naszej aplikacji. Gdy piszemy aplikację bazodanową, umieszczamy komponent `TDBGrid` na formularzu, a następnie modyfikujemy jego właściwości zawsze na te same wartości. Podobnie w pewnych narzędziach firmowych zawsze umieszczamy na formularzu pasek stanu, dodajemy kilka paneli i usuwamy uchwyt zmiany rozmiaru. Zamiast wykonywać te monotonne czynności w każdym nowym projekcie, kreujemy własny komponent, który automatycznie ustawia wszystko za nas. W ten sposób nie tylko szybciej rozpoczynamy pracę nad nowymi aplikacjami, ale dodatkowo mamy pewność, że są one pozbawione błędów. Jeśli jednak znajdzie się gdzieś jakiś błąd, wystarczy poprawić kod komponentu i ponownie skompilować pakiet, a zmiany zostaną uwzględnione w wykorzystujących go aplikacjach.

Pisanie komponentów

Istnieją różne rodzaje komponentów, więc to przodek naszego komponentu określi jego typ.

Komponenty *niewizualne* dziedziczą po klasie `TComponent`. Klasa ta zawiera minimum, jakie musi posiadać każdy komponent, ponieważ zapewnia podstawową integrację ze środowiskiem projektowym i umożliwia strumieniowanie właściwości.

Komponenty niewizualne to na ogół otoczenie bardziej złożonego kodu, który nie ma związku z interfejsem użytkownika. Przykładem może być komponent przyjmujący komunikat dziennika zdarzeń i automatycznie wysyłający go do komponentu listy tekstowej lub zapisujący go na dysk twardy. Sam komponent jest niewidoczny dla użytkownika aplikacji, ale wykonuje swoje działania w tle, zapewniając poprawną pracę aplikacji.

Komponenty *okienkowe* dziedziczą po klasie `TWinControl`. Obiekty te pojawiają się w interfejsie użytkownika i są interaktywne (na przykład umożliwiają wybranie pliku z listy). Choć możliwe jest kreowanie komponentów wywodzących się od `TWinControl`, C++Builder zapewnia komponent `TCustomControl`, który czyni to zadanie prostszym.

Komponenty *graficzne* przypominają komponenty okienkowe. Główna różnica między nimi polega na tym, że te pierwsze nie posiadają uchwytu okna, a co za tym idzie, nie mogą wchodzić w interakcję z użytkownikiem. Brak uchwytu oznacza także mniej zajmowanych zasobów. Choć komponenty te nie wchodzą w interakcję z użytkownikiem, mogą otrzymywać pewne komunikaty okna, na przykład te związane ze zdarzeniami myszy. Komponenty te dziedziczą po klasie `TGraphicsControl`.

Dlaczego warto budować na podstawie istniejących komponentów?

Największą zaletą budowania nowych komponentów wykorzystujących już istniejące jest zmniejszenie czasu przeznaczanego na programowanie. Nie bez znaczenia jest też to, że zakładamy, iż wszystkie komponenty, których używamy, są pozbawione błędów.

Przykładem może być komponent `TLabel`, z którego korzysta w zasadzie każdy projekt. Jeśli kilka tworzonych przez nas projektów współdzieli szatę graficzną, która wymaga dodania wielu etykiet i zmiany ich właściwości na te same wartości, warto zastanowić się nad wykreowaniem własnego komponentu, który modyfikuje właściwości etykiet, a my tylko zajmujemy się ustawieniem znajdujących się w nich tekstów i określeniem ich położenia.

Aby przedstawić, jak łatwo można to zrobić, wykreujemy komponent w kilka minut i napiszemy tylko trzy wiersze kodu. Z menu C++Buildera wybieramy *Component/New Component*. Po ukazaniu się okna dialogowego *New Component* wybieramy *TLabel* jako przodka komponentu, a jako nazwę klasy wpisujemy `TStyleLabel`. Dla komponentu instalowanego na palecie komponentów lub używanego w aplikacjach wybralibyśmy bardziej opisową nazwę klasy. W tym przykładzie pozostałe pola pozostawimy z wartościami domyślnymi. Klikamy przycisk *OK*. C++Builder utworzył dla nas pliki jednostki. Musimy jeszcze tylko dodać wiersze ustawiające właściwości etykiety. Po dokonaniu zmian zapisujemy plik i z menu *Component* wybieramy polecenie *Install Component*. Jeśli plik jest otwarty w środowisku programistycznym, jego nazwa pojawi się w polu *Unit file name*. Klikamy przycisk *OK*, aby zainstalować komponent na palecie komponentów. Listingi 4.1 i 4.2 przedstawiają cały kod przykładu.

Listing 4.1. *Plik nagłówkowy TStyleLabel, StyleLabel.h*

```
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------------------------------
class PACKAGE TStyleLabel : public TLabel
{
private:
protected:
public:
  __fastcall TStyleLabel(TComponent* Owner);
__published:
};
//---------------------------------------------------------------------------
#endif
```

Listing 4.2. *Kod źródłowy TStyleLabel, plik StyleLabel.cpp*

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "StyleLabel.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//

static inline void ValidCtrCheck(TStyleLabel *)
{
  new TStyleLabel(NULL);
}
//---------------------------------------------------------------------------
__fastcall TStyleLabel::TStyleLabel(TComponent* Owner) : TLabel(Owner)
{
  Font->Name = "Verdana";
  Font->Size = 12;
  Font->Style = Font->Style << fsBold;
}
//---------------------------------------------------------------------------
namespace Stylelabel
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TStyleLabel)};
    RegisterComponents("TestPack", classes, 0);
  }
}
//---------------------------------------------------------------------------
```

Inną zaletą tworzenia nowych komponentów na podstawie już istniejących jest możliwość kreowania klas bazowych ze wszystkimi funkcjami, ale bez upubliczniania właściwości. Przykładem może być typ `TListBox` komponentu, który nie udostępnia użytkownikom właściwości `Items`. Przy dziedziczeniu po `TCustomListBox` możliwe jest upublicznienie właściwości, do których ma mieć dostęp użytkownik (w trakcie projektowania) i udostępnienie pozostałych (na przykład właściwości `Items`) tylko w trakcie działania programu.

Poza tym właściwości i zdarzenia dodawane do już istniejących komponentów oznaczają zdecydowanie mniej kodu pisanego od podstaw.

Projektowanie własnych komponentów

Choć pewnie wydaje się to oczywiste, trzeba podkreślić, że dla własnych komponentów wykorzystuje się te same zasady projektowania, co dla całej aplikacji. Warto wcześniej pomyśleć o kierunku, w jakim może podążyć rozwijanie komponentu. Wcześniej wspomniane komponenty zapewniające listę informacji z baz danych nie dziedziczą po prostu po `TListBox`. Zamiast tego staramy się wykonać własną wersję klasy `TCustomListBox`, która zawrze nowe właściwości wspólne dla wszystkich kreowanych komponentów list baz danych. Następnie każdy z nowych komponentów korzysta z tej wspólnej klasy, co eliminuje potrzebę kilkakrotnego pisania tego samego kodu. Końcowe wersje każdego komponentu zawierają tylko unikalny dla nich kod (właściwości, metody i zdarzenia).

Korzystanie ze schematów VCL

Aby uzyskać pełną zgodność z architekturą VCL C++Buildera, warto poświęcić chwilę czasu na przejrzenie schematów VCL dostarczonych ze środowiskiem projektowym. Pozwoli to nie tylko dowiedzieć się, jakie komponenty są dostępne, ale także — od jakich klas się wywodzą.

W trakcie nauki projektowania i tworzenia komponentów należy starać się tak modelować klasy, by przyjęły one bardzo dobry styl obiektowy, czyli posiadały bardzo elastyczne klasy bazowe, od których generowane są komponenty. Choć kod źródłowy komponentów z C++Buildera napisany jest w języku Pascal, warto przyjrzeć się klasom bazowym niektórych komponentów, by dobrze poznać sposób ich konstrukcji, zobaczyć, w jaki sposób komponenty współdzielą właściwości tej samej klasy bazowej lub rodzicielskiej.

Poza tym schematy VCL bardzo dobrze uwidaczniają, które klasy bazowe spełniają wymagania kreowanego przez nas komponentu. W połączeniu z plikami pomocy VCL łatwo zidentyfikować najbardziej odpowiednią klasę rodzica. Wspomnieliśmy już, że minimalną klasą bazową dla komponentów jest klasa `TComponent`, `TWinControl` lub `TGraphicsControl`, w zależności od tego, jaki komponent kreujemy.

Pisanie komponentów niewizualnych

Świat komponentów opiera się na trzech filarach: właściwościach, zdarzeniach i metodach. W tej części przyjrzymy się każdemu z nich, aby poznać zasady tworzenia komponentów i ich wspólnej pracy w aplikacjach C++Buildera.

Właściwości

Właściwości można podzielić na dwie kategorie, publikowanie i niepublikowane. Właściwości *publikowane* dostępne są w zintegrowanym środowisku projektowym w trakcie projektowania (oczywiście w trakcie działania programu również). Właściwości *niepublikowane* wykorzystywane są przez aplikację w czasie jej pracy. Najpierw przyjrzymy się właściwościom niepublikowanym.

Właściwości niepublikowane

Komponent to klasa znajdująca się w pakiecie i posiadająca kilka dodatkowych funkcji. Przyjrzyjmy się przykładowej klasie z listingu 4.3.

Listing 4.3. *Ustawianie i pobieranie zmiennych prywatnych*

```
class LengthClass
{
private:
  int FLength;

public:
  LengthClass(void){}
  ~LengthClass(void){}
  int GetLength(void);
  void SetLength(int pLength);
  void LengthFunction (void);
}
```

Listing 4.3 przedstawia prywatną zmienną wykorzystywaną wewnętrznie przez klasę i metody używane do jej ustawiania i pobierania. Bardzo łatwo może to prowadzić do powstania nieczytelnego kodu. Przyjrzyjmy się listingowi 4.4 zawierającemu przykład użycia metod.

Listing 4.4. *Wykorzystanie metod pobierania i ustawiania*

```
LengthClass Rope;
Rope.SetLength(15);
// pozostały kod
int NewLength = Rope.GetLength();
```

Powyższy kod nie jest złożony, ale bardzo szybko może się stać mało czytelny w bardziej wyrafinowanej aplikacji. Czy nie lepiej byłoby, gdybyśmy mogli się odnosić do zmiennej jako właściwości klasy? To właśnie umożliwia C++Builder. Przebudowaną klasę przedstawia listing 4.5.

Listing 4.5. *Wykorzystanie właściwości do ustawiania i pobierania zmiennej prywatnej*

```
class LengthClass2
{
private:
  int FLength;
```

```
public:
  LengthClass2(void){}
  ~LengthClass2(void){}
  void LengthFunction(void);
  __property int Length = {read = FLength, write = FLength};
}
```

Przykładowy kod z listingu 4.4 przyjmie teraz postać zaprezentowaną na listingu 4.6.

Listing 4.6. *Ustawianie i pobieranie właściwości*

```
LengthClass Rope;
Rope.Length = 15;
// pozostały kod
int NewLength = Rope.Length;
```

W deklaracji klasy pojawiło się słowo kluczowe `__property` (rozszerzenie języka C++ wprowadzone w C++Builderze). Właściwość zawiera dwa inne słowa kluczowe, `read` i `write`. Gdy w listingu 4.6 odczytujemy wartość właściwości `Length`, zwracana jest wartość `FLength`. Gdy ustawiamy właściwość `Length`, w rzeczywistości modyfikujemy `FLength`.

Dlaczego mamy postępować w sposób tak skomplikowany, gdy można po prostu upublicznić zmienną `FLength`? Właściwości umożliwiają wykonanie następujących zadań:

- Ustalenie, że właściwość `Length` jest tylko do odczytu, bez stosowania słowa kluczowego `write`.
- Zapewnienie aplikacji publicznego dostępu do prywatnych danych klasy bez wpływania na implementację samej klasy. Jest to bardziej istotne, gdy wartość właściwości jest dziedziczona lub należy wykonać pewne działania w trakcie zmiany właściwości.
- Korzystanie z efektów ubocznych w trakcie przypisywania wartości do właściwości. Te efekty uboczne mogą powodować zachowanie poprzedniego stanu obiektu, zapisanie informacji w innym miejscu lub przygotowanie wartości innych właściwości do pobrania przez kod wywołujący (obliczanie wyprzedzające).
- Obliczanie wartości tylko wtedy, gdy zostaniemy o nią poproszeni (obliczanie na żądanie). Jest to szczególnie przydatne dla nieskończonych ciągów liczb (na przykład liczb pierwszych) lub też złożonych obliczeń, których nie chcemy wykonywać, o ile nie zostaniemy do tego zmuszeni.

Listing 4.7 przedstawia odmianę poprzedniego przykładu.

Listing 4.7. *Połączenie metod pobierania i ustawiania z właściwością*

```
class LengthClass3
{
private:
  int FLength;
  int GetLength(void);
  void SetLength(int pLength);
```

```
public:
  LengthClass3(void){}
  ~LengthClass3(void){}
  void LengthFunction(void);
  __property int Length = {read = GetLength, write = SetLength};
}
```

Przykład z listingu 4.7 obrazuje ogromną użyteczność właściwości. Deklaracja właściwości informuje, że wartość zwracana jest przez metodę `GetLength()`, gdy odczytujemy `FLength`. Metoda `SetLength()` jest wywoływana, gdy przypisujemy nową wartość do `FLength`.

Metoda `GetLength()` może przeprowadzić pewne obliczenia bazujące na innych prywatnych członkach klasy. Metoda `SetLength()` może przeprowadzić sprawdzenie przekazanej wartości, a także wykonać inne zadania przed rzeczywistym ustawieniem `FLength`.

Przykładem takiego działania w C++Builderze jest zmiana nazwy aliasu dla połączenia ze źródłem bazy danych. Jako programiści modyfikujemy nazwę aliasu. W tle komponent odłącza się od aktualnej bazy danych (jeśli takowa istnieje), zanim połączy się z nowym źródłem. Implementacja jest ukryta przed użytkownikiem i korzysta z niej tylko właściwość.

Rodzaje właściwości

Właściwości mogą być dowolnego rodzaju, od prostych typów danych, jak `int`, `bool`, `short`, aż po własne klasy. Należy jednak zwrócić uwagę na dwa zagadnienia, gdy korzysta się z własnych klas jako typów właściwości. Po pierwsze klasa musi się wywodzić przynajmniej od klasy `TPersistent`, jeśli ma być wykorzystywana strumieniowo w formularzu. Po drugie, jeśli chcemy zadeklarować klasę z wyprzedzeniem, musimy użyć słowa kluczowego `__declspec(delphiclass)`.

Kod z listingu 4.8 skompiluje się, gdy skorzystamy z typowej deklaracji klasy z wyprzedzeniem. Zauważmy, że jeszcze nie zdefiniowaliśmy właściwości.

Listing 4.8. *Deklaracja z wyprzedzeniem*

```
class MyClass;
class PACKAGE MyComponent : public TComponent
{
private:
  MyClass *FMyClass;
// ...
};

class MyClass : public TPersistent
{
public:
  __fastcall MyClass (void){}
};
```

Słowo kluczowe PACKAGE między nazwą klasy a słowem kluczowym class to makro, które zostanie rozwinięte do kodu, umożliwiającego eksport komponentu z biblioteki pakietu (*.BPL*). Biblioteka pakietu to specjalny rodzaj biblioteki DLL, która pozwala na współdzielenie kodu przez aplikacje. Więcej informacji na temat bibliotek pakietów i makra PACKAGE znajduje się w pomocy środowiska C++Builder.

Jeśli jednak chcemy dodać właściwość typu MyClass, musimy zmodyfikować deklarację wyprzedzającą w sposób przedstawiony na listingu 4.9.

Listing 4.9. *Właściwość typu własnej klasy*

```
class __declspec(delphiclass) MyClass;

class PACKAGE MyComponent : public TComponent
{
private:
  MyClass *FMyClass;
// ...

__published:
  __property MyClass *Class1 = {read = FMyClass, write = FMyClass};
};

class MyClass : public TPersistent
{
public:
  __fastcall MyClass (void){}
};
```

Właściwości publikowane

Publikacja właściwości umożliwia dostęp do nich z poziomu komponentu w trakcie projektowania w środowisku projektowym. Właściwości wyświetlane są w oknie *Object Inspector*, co umożliwia użytkownikowi zobaczenie lub modyfikację aktualnej wartości właściwości. Oczywiście właściwości te dostępne są także w trakcie działania programu, ale ich główne zastosowanie to zapewnienie ich łatwej modyfikacji w sposób graficzny bez potrzeby pisania jakiegokolwiek kodu. Dodatkowo opublikowane właściwości są strumieniowane do formularza, więc ich wartości przechowuje się między kolejnymi otwarciami projektu.

Właściwości publikowane definiuje się w ten sam sposób, co pozostałe, ale umieszcza się je w sekcji __published deklaracji klasy. Przykład ich zastosowania przedstawia listing 4.10.

Listing 4.10. *Publikacja właściwości*

```
class PACKAGE LengthClass : public TComponent
{
private:
  int FLength;
  int GetLength(void);
  void SetLength(int pLength);
```

```
public:
  __fastcall LengthClass(TObject *Owner) : TComponent(Owner) {}
  __fastcall ~LengthClass(void){}
  void LengthFunction(void);

__published:
  __property int Length = {read = Getlength, write = Setlength};
}
```

Poprzednia klasa przypomina tę z listingu 4.7, ale właściwość `Length` została przeniesiona do sekcji `__published`. Właściwości publikowane przedstawione w oknie *Object Inspector* mogą być odczytywane i zapisywane, ale istnieje możliwość uczynienia z nich właściwości tylko do odczytu, dzięki czemu powstaje atrapa metody zapisu. Listing 4.11 przedstawia sposób dodania właściwości publikowanej w poprzednim komponencie, aby pokazywała aktualną wersję komponentu.

Listing 4.11. *Wersja właściwości*

```
const int MajorVersion = 1;
const int MinorVersion = 0;

class PACKAGE LengthClass : public TComponent
{
private:
  AnsiString FVersion;
  int FLength;
  int GetLength(void);
  void SetLength(int pLength);
  void SetVersion(AnsiString /* pVersion */ )
    {FVersion = AnsiString(MajorVersion) + "." + AnsiString(MinorVersion);}

public:
  __fastcall LengthClass(TObject *Owner) : TComponent(Owner) {SetVersion("");}
  __fastcall ~LengthClass(void){}
  void LengthFunction(void);

__published:
  __property int Length = {read = Getlength, write = Setlength};
  __property AnsiString Version = {read = FVersion, write = SetVersion};
}
```

Zdefiniowaliśmy prywatną zmienną `FVersion`, której wartość ustawiamy w konstruktorze klasy. Następnie dodaliśmy właściwość `Version` do sekcji `__published` i przypisaliśmy jej słowa kluczowe `read` i `write`. Słowo kluczowe `read` powoduje zwrócenie wartości `FVersion`, a metoda `write` ustawia właściwość na jej wartość początkową. Nazwa zmiennej z listy parametrów `SetVersion` została oznaczona komentarzem, aby nie otrzymywać komunikatu o deklaracji zmiennej, która nie jest wykorzystywana. Ponieważ właściwość jest typu `AnsiString`, metoda `SetVersion()` musi przyjmować właśnie taki parametr w swojej deklaracji.

Właściwości tablicowe

Pewne właściwości to tablice, a nie typy proste, jak `bool`, `int`, a nawet `AnsiString`. Ich stosowanie nie jest zbyt dobrze opisane w dokumentacji. Na przykład właściwością tablicową jest `Lines` komponentu `TMemo`. Właściwość ta umożliwia użytkownikowi dostęp do poszczególnych wierszy komponentu.

Właściwości tablicowe deklaruje się w ten sam sposób, co pozostałe. Istnieją jednak dwie różnice: deklaracja zawiera odpowiednie indeksy z wymaganymi typami, a indeksy nie są ograniczone do wartości całkowitych. Listingi od 4.12 do 4.15 obrazują sposób wykorzystania dwóch właściwości. Jedna jako indeks przyjmuje tekst, a druga — wartość całkowitą.

Listing 4.12. *Wykorzystanie tekstu jako indeksu*

```
class PACKAGE TStringAliasComponent : public TComponent
{
private:
  TStringList RealList;
  TStringList AliasList;
  __AnsiString __fastcall GetStringAlias(AnsiString RawString);
  AnsiString __fastcall GetRealString(int Index);
  void __fastcall SetRealString(int Index, AnsiString Value);
public:
  __property AnsiString AliasString[AnsiString RawString] = {read = GetStringAlias};
  __property AnsiString RealString[int Index] = {read=GetRealString, write=SetRealString};
}
```

Podany przykład może stać się częścią komponentu, który wewnętrznie przechowuje listę tekstów i dodatkową listę aliasów tekstów. Właściwość `AliasString` przyjmuje wartość `RawString`, a zwraca alias przez metodę `GetStringAlias()`. Wiele osób piszących komponenty przy pierwszym stosowaniu właściwości tablicowych dziwi to, że deklaracja wykorzystuje notację indeksową (czyli `[]`), ale kod używa takiej samej notacji, jak w przypadku wywoływania innej metody. Przyjrzyjmy się właściwości `RealString`. Zauważymy, że nie tylko zwraca ona typ `AnsiString`, ale przyjmuje jako indeks liczbę całkowitą. W celu pobrania konkretnego tekstu z listy bazującej na indeksie zostania użyta metoda `GetRealString()` (patrz listing 4.13).

Listing 4.13. *Metoda odczytu właściwości tablicowej*

```
AnsiString __fastcall TStringAliasComponent::GetRealString(int Index)
{
  if(Index > (RealList->Count -1))
    return "";
  return RealList->Strings[Index];
}
```

W kodzie właściwość będzie używana w następujący sposób:

```
AnsiString str = StringAlias1->RealString[0];
```

Teraz przyjrzyjmy się metodzie `SetRealString()`. Może ona wydawać się nieco dziwna, jeśli nigdy wcześniej nie używaliśmy tablic jako właściwości. Pierwszy parametr przyjmuje indeks tablicy jako tekst, a drugi — wartość `AnsiString`. Zmienna `RealList TStringList` umieści `AnsiString` na liście w położeniu podanym w parametrze indeksu. Listing 4.14 przedstawia definicję metody `SetRealString()`.

Listing 4.14. *Metoda zapisu właściwości tablicowej*

```
void __fastcall TStringAliasComponent::SetRealString(int Index, AnsiString Value)
{
  if((RealList->Count - 1) < Index)
    RealList->Add(Value);
  else
    RealList->Insert(Index, Value);
}
```

W listingu 4.14 wartość parametru `Index` jest porównywana z ilością tekstów aktualnie znajdujących się na liście. Jeśli `Index` jest większy, tekst określony w `Value` jest po prostu dodawany do końca listy. W przeciwnym razie wykorzystuje się metodę `Insert()` z `TStringList`, aby wstawić tekst w miejsce wskazane indeksem. Teraz możemy przypisać tekst do listy, używając następującego kodu:

```
StringAlias1->RealString[1] = "Tekst";
```

Teraz ciekawostka: metoda `GetStringAlias()` jest metodą odczytu dla właściwości `AliasString`, która przyjmuje tekst jako indeks. Wiemy, że listy tekstów to tablice, więc każdy łańcuch tekstu ma indeks, czyli położenie na liście. Możemy użyć metody `IndexOf()` w celu porównania tekstu przekazanego jako indeks z tekstem zawartym na liście. W ten sposób uzyskamy wartość całkowitą stanowiącą indeks tekstu lub –1, jeśli tekst nie istnieje na liście. Teraz wystarczy już tylko zwrócić tekst wskazywany indeksem uzyskanym od `IndexOf()` z listy aliasów. Obrazuje to listing 4.15.

Listing 4.15. *Metoda GetStringAlias()*

```
AnsiString __fastcall TStringAliasComponent::GetStringAlias(AnsiString RawString)
{
  int Index;
  Index = RealList->IndexOf(RawString);
  if((Index == -1) || (Index > (AliasList->Count-1)))
    return RawString;
  return AliasList->Strings [Index];
}
```

Aby wykorzystać właściwość, stosujemy następujący kod:

```
AnsiString MyAliasString = StringAlias1->AliasString("Tekst");
```

Nie tylko odczyt i zapis

Przykładowe kody z listingów od 4.5 do 4.15 przedstawiały właściwości ze słowami kluczowymi `read` i `write` stanowiącymi część deklaracji. C++Builder umożliwia także stosowanie trzech dodatkowych opcji: `default`, `nodefault` i `stored`.

Słowo kluczowe `default` nie powoduje ustawienia domyślnej wartości właściwości. Informuje tylko C++Buildera, jaka wartość domyślna zostanie przypisana do właściwości (przez programistę) w konstruktorze klasy. Środowisko projektowe wykorzystuje tę informację, by stwierdzić, czy konieczne jest strumieniowane właściwości do formularza. Jeśli aktualna wartość właściwości jest równa wartości domyślnej, właściwość nie zostanie zapisana jako część formularza. Oto przykład.

```
__property int IntegerProperty = {read = Finteger, write = Finteger, default = 10};
```

Słowo kluczowe `nodefault` informuje środowisko projektowe, że właściwość nie ma związanej z nią wartości domyślnej. Gdy pierwszy raz deklarujemy właściwość, nie musimy dodawać słowa kluczowego `nodefault`, ponieważ brak słowa kluczowego `default` oznacza brak wartości domyślnej. Słowo kluczowe `nodefault` wykorzystuje się głównie w zmianach definicji dziedziczonych właściwości. Oto przykład.

```
__property int DescendantInteger = {read = Finteger, write = Finteger, nodefault};
```

Pamiętajmy o tym, że wartość właściwości ze słowem kluczowym `nodefault` będzie strumieniowana do formularza tylko wtedy, gdy przypiszemy do niej (lub do dotyczących jej zmiennych członkowskich) wartość za pomocą inspektora obiektów lub odpowiednich metod.

Słowo kluczowe `stored` wykorzystywane jest do sterowania zapamiętywaniem właściwości. Wszystkie publikowane właściwości są domyślnie przechowywane. Możemy zmienić to zachowanie, ustawiając słowo kluczowe `stored` na `false` lub podając nazwę funkcji, która zwraca wartość logiczną. Listing 4.16 przedstawia sposób wykorzystania słowa kluczowego `stored`.

Listing 4.16. *Stosowanie słowa kluczowego stored*

```
class PACKAGE LengthClass : public TComponent
{
protected:
  int FProp;
  bool StoreProperty(void);

__published:
  __property int AlwaysStore = {read = FProp, write = FProp, stored = true};
  __property int NeverStore = {read = FProp, write = FProp, stored = false};
  __property int SometimesStore = {read = FProp, write = FProp, stored = StoreProperty};
}
```

Kolejność tworzenia

Jeśli komponent zawiera właściwości, których wartości zależą od innych właściwości w trakcie strumieniowania, możemy sterować kolejnością ich wczytywania (a co za tym idzie — inicjalizacji), deklarując je w wymaganym porządku w nagłówku klasy. Na przykład kod z listingu 4.17 wczytuje właściwości w następującej kolejności: `PropA`, `PropB`, `PropC`.

Listing 4.17. *Zależności właściwości*

```
class PACKAGE SampleComponent : public TComponent
{
private:
  int FPropA;
  bool FPropB;
  String FProC;
  void __fastcall SetPropB(bool pPropB);
  void __fastcall SetPropC(String pPropC);

public:
  __property int PropA = {read = FPropA, write = FPropA};
  __property bool PropB = {read = FPropB, write = SetPropB};
  __property String PropC = {read = FPropC, write = SetPropC};
}
```

Jeśli mamy właściwości z zależnościami i nie potrafimy ich poprawnie zainicjalizować, upewnijmy się, że kolejność ich deklaracji jest właściwa.

Zdarzenia

Zdarzenie komponentu to wywołanie opcjonalnej metody w odpowiedzi na pewne wydarzenie. Wydarzenie może być informacją dla użytkownika, aby wykonał pewne działanie, zanim komponent wyłapie wyjątek lub system przechwyci komunikat.

Załóżmy, że mamy komponent, który przechodzi przez katalogi od korzenia. Jeśli został on tak zaprojektowany, że informuje użytkownika o zmianie katalogu, działanie to nazwiemy zdarzeniem. Gdy zajdzie zdarzenie, komponent sprawdza, czy użytkownik zapewnił procedurę jego obsługi (metoda dołączona do zdarzenia). Jeśli tak, wywołuje ją. Jeśli opis ten wydaje się niezrozumiały, warto prześledzić kod listingu 4.18.

Listing 4.18. *Deklaracja właściwości zdarzenia*

```
class PACKAGE TTraverseDir : public TComponent
{
private:
  AnsiString FCurrentDir;
  TNotifyEvent *FOnDirChanged;

public:
  __fastcall TTraverseDir(TObject *Owner) : TComponent(Owner){FOnDirChanged = 0;}
  __fastcall ~TTraverseDir(void){}
  __fastcall Execute();

__published:
  __property AnsiString CurrentDir = {read = FCurrentDir};
  __property TNotifyEvent OnDirChanged = {read = FOnDirChanged, write = FOnDirChanged};
}
```

Listing 4.18 obrazuje interesujący nas fragment kodu zawierający deklarację właściwości tylko do odczytu i standardowego zdarzenia. Gdy komponent jest wykonywany, pojawią się sytuacje, w których nastąpi zmiana aktualnego katalogu. Przyjrzyjmy się przykładowemu kodowi.

```
void __fastcall TTraverseDir::Execute(void)
{
  // przeprowadzanie przejścia przez katalog

  // Tu właśnie zmieniamy katalog, więc
  // wywołujemy zdarzenie DirChanged, jeśli takowe istnieje.

if(FOnDirChanged)
  FOnDirChanged(this);

// pozostały kod komponentu
}
```

Zmienna `FOnDirChange` z tego kodu to wskaźnik na `TNotifyEvent` zadeklarowany w następujący sposób:

```
typedef void __fastcall (__closure *TNotifyEvent)(System::TObject* Sender)
```

Deklaracja wymaga przekazania pojedynczego parametru typu `TObject*`. Gdy tworzymy zdarzenie (dwukrotne kliknięcie zdarzenia w oknie inspektora obiektów), środowisko projektowe generuje następujący kod:

```
void __fastcall TTraverseDir::Traverse1DirChanged(TObject *Sender)
{
}
```

Wewnątrz tej metody użytkownik dodaje kod wykonywany po zgłoszeniu zdarzenia. W tym przypadku mamy do czynienia ze standardowym zdarzeniem przekazującym wskaźnik do obiektu generującego zdarzenie. Wskaźnik ten umożliwia rozróżnienie w projekcie kilku komponentów tego samego typu.

```
void __fastcall TTraverseDir::Traverse1DirChanged(TObject *Sender)
{
  if(Sender == Traverse1)
    // kod obsługi komponentu o nazwie Traverse1
  else
    // obsługa pozostałych
}
```

Tworzenie zdarzeń zawierających dodatkowe parametry

Przypomnijmy, że typowe zdarzenia definiowane są w następujący sposób.

```
typedef void __fastcall (__closure *TNotifyEvent)(System::TObject* Sender)
```

Podany kod przedstawia sposób definiowania własnych zdarzeń.

```
typedef void __fastcall (__closure *TDirChangedEvent) (System::TObject* Sender,
➥bool &Abort)
```

W kodzie tym wykonaliśmy dwa zadania:

- utworzyliśmy unikalną definicję typu `typedef`. `TNotifyEvent` to teraz `TDirChangedEvent`;
- dodaliśmy wymagane parametry do listy parametrów.

Możemy teraz zmodyfikować deklarację klasy. Zmianę przedstawia listing 4.19.

Listing 4.19. *Właściwości własnych zdarzeń*

```
typedef void __fastcall (__closure *TDirChangedEvent) (System::TObject* Sender,
➥bool &Abort)

class PACKAGE TTraverseDir : public TComponent
{
private:
  TDirChangedEvent *FOnDirChanged;

__published:
  __property TDirChangedEvent OnDirChanged = {read = FOnDirChanged, write = FOnDirChanged};
}
```

Gdy teraz użytkownik utworzy nowe zdarzenie, środowisko programistyczne doda następujący kod.

```
void __fastcall TTraverseDir::Traverse1DirChanged(TObject *Sender, bool &Abort)
{
}
```

Należy wykonać jeszcze tylko jedną modyfikację: zmienić kod źródłowy wywołujący zdarzenie (patrz listing 4.20).

Listing 4.20. *Wywołanie zdarzenia*

```
void __fastcall TTraverseDir::Execute(void)
{
  // przeprowadzanie przejścia przez katalog

  bool &Abort = false;

  // Tu właśnie zmieniamy katalog, więc
  // wywołujemy zdarzenie DirChanged, jeśli takowe istnieje.

if(FOnDirChanged)
  FOnDirChanged(this, Abort);

if (Abort)
  // obsługa zaprzestania zmian katalogu

// pozostały kod komponentu
}
```

Komponent został tak zmodyfikowany, by umożliwić użytkownikowi przerwanie procesu wchodzenia do kolejnych katalogów.

Metody

Metody komponentu zawierają kod wymagany do wykonywania różnych zadań. Metody te nie różnią się niczym od metod typowej klasy. W trakcie pisania komponentu podstawowym celem jest minimalizacja liczby metod, które musi wywołać aplikacja. Dalej przedstawiamy kilka prostych zasad dotyczących projektowania komponentów.

- Użytkownik nie musi wywoływać żadnych metod, aby komponent zachowywał się tak, jak tego oczekuje. Na przykład komponent sam musi zająć się całą inicjalizacją.
- Nie mogą istnieć żadne zależności co do kolejności wywoływania metod. Należy tak zaprojektować komponenty, by zdarzenia mogły zachodzić w dowolnej kolejności. Jeśli użytkownik wywoła metodę uzależnioną od stanu (na przykład próba odpytania bazy danych, gdy nie ma aktywnego połączenia), komponent musi sobie poradzić z taką sytuacją. Od projektanta (i rodzaju metody) zależy, czy w takiej sytuacji komponent postara się połączyć z bazą danych, czy zgłosi wyjątek.
- Użytkownik nie może wywołać metody zmieniającej stan komponentu, gdy ten wykonuje inne zadanie.
- Ogólnie nie należy używać metod do pobierania i ustawiania wartości komponentów. W tym celu używa się właściwości.

Metody piszemy tak, by sprawdzały aktualny stan komponentu. Jeśli nie spełnia on wszystkich założeń, komponent powinien zostać poprawiony, o ile to możliwe. W przeciwnym razie należy zgłosić wyjątek. Gdy jest to wskazane, kreujemy własne wyjątki, aby użytkownik po ich typie mógł określić, od jakiego komponentu pochodzą.

Starajmy się pisać właściwości, a nie metody. Właściwości umożliwiają ukrycie przez użytkownikiem rzeczywistej implementacji, więc łatwiej zrozumie działanie komponentu.

Na ogół metody komponentów znajdują się w sekcjach publicznych lub chronionych. Metody prywatne piszemy tylko wtedy, gdy zamierzamy ukryć konkretną implementację komponentu nawet przed komponentami pochodnymi.

Metody publiczne

Metody *publiczne* to metody wywoływane przez użytkownika, gdy chce, aby komponent wykonał pewne zadanie.

Jeśli metoda będzie działała dłuższy czas, warto rozważyć wykonanie zdarzenia, które będzie informowało użytkownika o postępie prac. Poza tym możemy zapewnić przerwanie zadania po zwróceniu przez zdarzenie odpowiedniej wartości.

Wyobraźmy sobie komponent wyszukujący w drzewie katalogów konkretny plik. W zależności od przeszukiwanego systemu może to zająć znaczną ilość czasu. Aby użytkownik nie musiał zastanawiać się, czy aplikacja przestała funkcjonować, warto utworzyć zdarzenie wywoływane w metodzie analizującej. Zdarzenie może zapewnić sprzężenie zwrotne, wyświetlając na przykład nazwę aktualnie przeszukiwanego katalogu.

Metody chronione

Jeśli komponent posiada metody, które mają być wywoływane przez komponenty pochodne, a nie użytkownika, deklarujmy je jako metody *chronione*. W ten sposób zabezpieczamy się przed wywołaniem metody w nieodpowiednim momencie. Najbezpieczniej jest tworzyć metody publiczne wywołujące metody chronione tylko wtedy, gdy spełnione zostaną wszystkie wymagania.

Jeśli metodę tworzymy z myślą o implementacji właściwości, powinniśmy ją zadeklarować jako *wirtualną metodę chronioną*. Umożliwi to komponentom pochodnym zmianę jej implementacji.

Przykładem wirtualnej metody chronionej jest metoda `Loaded()` komponentów. Jest ona wywoływana, gdy komponent zostanie w pełni wczytany (zakończy się strumieniowanie z formularza).

W pewnych przypadkach komponent pochodny musi wiedzieć, kiedy wczytane zostały wszystkie właściwości, aby przeprowadzić jeszcze dodatkowe ustawienia. Przykładem może być komponent przeprowadzający sprawdzenie w metodzie ustawiającej, który nie może zapewnić tego sprawdzenia, zanim nie zostaną wczytane wszystkie właściwości. W takim przypadku kreujemy zmienną członkowską `IsLoaded` i ustawiamy ją w konstruktorze na `false` (wykonywane jest to domyślnie, ale wykonanie tego jawnie uczyni kod czytelniejszym). Następnie przysłaniamy metodę `Loaded()`, aby ustawić w niej `IsLoaded` na `true`. Następnie korzystamy z tej zmiennej w metodach implementujących właściwości, aby dokonać odpowiedniego sprawdzenia.

Listingi 4.21 i 4.22 pochodzą od własnego komponentu `TAliasComboBox`. Komponent ten wchodzi w skład pakietu MJFPack, który można pobrać z witryny *http://www.mj-freelancing.com*. Pakiet zawiera więcej komponentów umożliwiających łączenie w przedstawiony sposób.

Listing 4.21. *Plik nagłówkowy dla TAliasComboBox*

```
class PACKAGE TAliasComboBox : public TSmartComboBox
{
private:
  bool IsLoaded;

protected:
  virtual void __fastcall Loaded(void);
}
```

Listing 4.22. *Plik źródłowy TAliasComboBox*

```
void __fastcall TAliasComboBox::Loaded(void)
{
  TComponent::Loaded();

  if(!ComponentState.Contains(csDesigning))
  {
```

```
    IsLoaded = true;
    GetAliases();
  }
}
```

W zaprezentowanym kodzie metoda `Loaded()` została przysłonięta w deklaracji klasy. W pliku źródłowym zaczynamy od wywołania metody `Loaded()` przodka, a następnie dodajemy własny kod. Listing 4.22 przedstawia weryfikację komponentu, której nie ma w trybie projektowania, zanim nie zostaną pobrane informacje o aliasach. Ponieważ stan pewnych właściwości może zależeć od stanu innych właściwości, inne metody tego komponentu sprawdzają zmienną `IsLoaded` przed przeprowadzeniem przetwarzania wykorzystującego pozostałe właściwości. Oczywiście większość przetwarzania dla tego komponentu odbywa się w trakcie pracy programu.

Tworzenie wyjątków dla komponentów

Czasem istnieje możliwość ponowienia wyjątku uzyskanego w komponencie w celu obsłużenia go przez użytkownika. Zapewne bez problemu wykonamy wymagane zwalnianie pamięci zajmowane przez komponent, gdy wystąpi wyjątek. Po przeprowadzeniu zwalniania musimy wykonać jedno z dwóch zadań.

Możemy ponowić zgłoszenie wyjątku. Takie zachowanie jest odpowiednie dla standardowych błędów, jak na przykład dzielenie przez zero. Istnieją jednak sytuacje, w których wyjątek lepiej zamienić na zdarzenie, co zapewni bardzo wygodny sposób obsłużenia błędu przez użytkownika. Nie należy jednak zamieniać wszystkich wyjątków na zdarzenia, ponieważ utrudni to użytkownikowi wykreowanie aplikacji.

Przedstawimy prosty przykład, który powinien wyjaśnić sytuację. Wyobraźmy sobie komponent wykonujący pewną liczbę sekwencyjnych zapytań do bazy danych. Komponent składa się z właściwości `TStrings` zawierającej wszystkie zapytania z metody `Execute()`, która je wykonuje. W jaki sposób użytkownik korzysta z komponentu? Obrazują to dwa podane dalej wiersze kodu.

```
MultiQuery1->Queries->Assign(Memo1->Lines);
MultiQuery1->Execute();
```

Implementacja kodu przez użytkownika jest prosta, ale co z możliwymi zgłoszeniami wyjątków? Czy użytkownik sam powinien zająć się ich obsługą? Nie jest to najlepsze rozwiązanie. Lepsze podejście to wykreowanie zdarzenia, gdy wystąpi wyjątek. W procedurze obsługi zdarzenia użytkownik będzie miał możliwość przerwania procesu.

Utwórzmy własny wyjątek, który będzie zgłaszany, gdy użytkownik zechce wykonać zapytanie znajdujące się poza dopuszczalnym zakresem indeksów. Na razie załóżmy, że istnieje metoda `ExecuteItem()`, która przyjmuje indeks zapytania do wykonania.

Najpierw musimy napisać wyjątek w pliku nagłówkowym. Jest to proste, ponieważ wymaga jedynie utworzenia nowej klasy wyjątku dziedziczącej po klasie `Exception` (patrz listing 4.23).

Listing 4.23. *Własna klasa wyjątku*

```
class EMultiQueryIndexOutOfBounds : public Exception
{
public:
  __fastcll EMultiQueryIndexOutOfBounds(const AnsiString Msg) : Exception(Msg){}
};
```

To wszystko. Jeśli teraz użytkownik spróbuje wykonać zapytanie (identyfikowane indeksem), podając indeks spoza dopuszczalnego zakresu, zgłaszamy wyjątek.

Kod zgłaszający wyjątek przedstawia listing 4.24.

Listing 4.24. *Zgłaszanie własnego wyjątku*

```
void __fastcall TMultiQuery::ExecuteItem(int Index)
{
  if(Index < 0 || Index > Queries->Count)
    throw EmultiQueryIndexOutOfBounds;

  // przeprowadzenie zapytania
}
```

Na listingach 4.23 i 4.24 pokazaliśmy, że utworzenie własnego wyjątku jest bardzo łatwe w implementacji. Jeśli komponent ma przeprowadzić zapytanie w trakcie projektowania, musimy zapewnić użytkownikowi komunikat (a nie zgłoszenie wyjątku przez środowisko). Zmodyfikowany w tym celu kod przedstawia listing 4.25.

Listing 4.25. *Zgłaszanie wyjątku w czasie projektowania*

```
void __fastcall TMultiQuery::ExecuteItem(int Index)
{
  if(Index < 0 || Index > Queries->Count)
  {
    if(ComponentState.Contains(csDesigning))
      throw EmultiQueryIndexOutOfBounds("Indeks zapytania znajduje się poza zakresem");
    else
      throw EmultiQueryIndexOutOfBounds;
  }

  // przeprowadzenie zapytania
}
```

Przestrzeń nazw

Gdy kreujemy i nazywamy komponenty, mogą pojawić się inni programiści, którzy przypadkiem stosują te same nazwy dla własnych komponentów. Spowoduje to powstanie konfliktu. Można jednak temu zapobiec, stosując słowo kluczowe `namespace`.

Gdy nowy komponent tworzymy za pomocą kreatora, środowisko programistyczne generuje kod podobny do tego z listingu 4.26.

Listing 4.26. *Kod w przestrzeni nazw*

```
namespace Aliascombobox
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TAliasComboBox)};
    RegisterComponents("MJF Pack", classes, 0);
  }
}
```

Słowo kluczowe `namespace` zapewnia tworzenie komponentu we własnym podsystemie. Pamiętajmy o tym, że musimy w reszcie kodu komponentu stosować tę samą przestrzeń nazw.

Przypuśćmy, że dwóch programistów opracowało komponent zegara i tak samo nazwali zmienną `const` określającą domyślny tryb pracy. Jeśli obydwu zegarów użyjemy w tej samej aplikacji, kompilator zgłosi błąd z powodu podwójnej deklaracji.

```
// pierwszy programista
const bool Mode12; // domyślnie tryb 12-godzinny
class PACKAGE TClock1 : public TComponent
  {
  }
// drugi programista
const bool Mode12; // domyślnie tryb 12-godzinny
class PACKAGE TClock2 : public TComponent
  {
  }
```

Z tego właśnie powodu musimy pamiętać o przestrzeniach nazw w trakcie kreowania komponentów. Po wszystkich instrukcjach `#include` z pliku nagłówkowego otaczamy kod w sposób przedstawiony na listingu 4.27.

Listing 4.27. *Otaczanie kodu*

```
namespace NClock1
{
  class PACKAGE TClock1 : public
  {
  }
}
```

Stosujemy jedną konwencję nazw dla wszystkich naszych komponentów. Na przykład nazwę przestrzeni rozpoczynamy od litery `N`, po której następuje nazwa komponentu. Jeśli istnieje możliwość stosowania tej nazwy przez kogoś innego, poprzedzamy nazwę przestrzeni inicjałami nazwy firmy. Korzystanie z przestrzeni nazw zapewni poprawne współdziałanie naszych komponentów z innymi, pisanymi przez różne firmy.

Odpowiadanie na komunikaty

VCL obsługuje prawie wszystkie komunikaty okien, jakie kiedykolwiek będziemy chcieli wykorzystać. Istnieją jednak sytuacje, w których sami musimy dodać obsługę odpowiedzi na specyficzny komunikat.

Należy pamiętać o tym, że jawne korzystanie z komunikatów systemu Windows uniemożliwi przeniesienie komponentów do innych systemów operacyjnych. Komponenty CLX nigdy nie powinny jawnie korzystać z komunikatów Windows.

Przykładem sytuacji wymagającej bezpośredniej obsługi komunikatów jest na przykład dodanie obsługi przeciągania plików z *Eksploratora Windows* do komponentu siatki tekstowej. Możemy utworzyć taki komponent, nazywany `TSuperStringGrid`, kreując go z klasy rodzicielskiej `TStringGrid` i dodając nowe funkcje.

Operacja przeciągnij i upuść jest obsługiwana przez komunikat API `WM_DROPFILES`. Informacje dotyczące operacji przechowywane są w strukturze `TWMDropFiles`.

Przechwytywanie komunikatów okien w komponentach jest takie samo jak w pozostałych projektach. Jedyna różnica polega na tym, że pracujemy z komponentem, a nie formularzem projektu. Z tego powodu mapę komunikatów ustawiamy w sposób przedstawiony na listingu 4.28.

Listing 4.28. *Przechwytywanie komunikatów*

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_DROPFILES, TWMDropFiles, WmDropFiles)
END_MESSAGE_MAP(TStringGrid)
```

W deklaracji mapy komunikatów nie stosuje się średników, ponieważ `BEGIN_MESSAGE_MAP`, `MESSAGE_HANDLER` i `END_MESSAGE_MAP` to makra rozwijane w trakcie kompilacji, które same je dodają.

Kod z listingu 4.28 tworzy mapę komunikatów dla komponentu (zauważ `TSringGrid` na końcu makra `END_MESSAGE_MAP`). Obsługa komunikatów przekaże wszystkie przechwycone komunikaty do metody `WmDropFiles` (za chwilę się nią zajmiemy). Informacje do metody przekazywane są w postaci struktury `TWMDropFiles` zdefiniowanej w systemie Windows.

Teraz zajmiemy się utworzeniem metody, która obsłuży komunikat. W sekcji `protected` komponentu definiujemy metodę, używając następującego kodu:

```
protected:
  void __fastcall WmDropFiles(TWMDropFiles &Message);
```

Zauważmy, że dodajemy referencję na wymaganą strukturę jako parametr metody.

Zanim komponent zacznie działać, musimy jeszcze zarejestrować go w systemie Windows, informując o tym, iż może przyjmować upuszczane pliki. Wykonujemy to poleceniem `DragAcceptFiles()` w trakcie wczytywania komponentu.

```
DragAcceptFiles(Handle, FCanDropFiles);
```

W tym kodzie zmienna `FCanDropFiles` wykorzystywana jest przez komponent do wskazania, czy możliwa jest obsługa upuszczania plików.

Teraz metoda przyjmie nazwy plików, gdy komponent otrzyma komunikat Windows dotyczący techniki *przeciągnij i upuść*. Kod z listingu 4.29 to fragment pełnej wersji klasy komponentu.

Listing 4.29. *Przyjmowanie upuszczonych plików*

```
void __fastcall TSuperStringGrid::WmDropFiles(TWMDropFiles &Message)
{
  char buff[MAX_PATH];
  HDROP hDrop = (HDROP)Message.Drop;
  POINT Point;
  int NumFiles = DragQueryFile(hDrop, -1, NULL, NULL);
  TStringList *DFiles = new TStringList;
  DFiles->Clear();
  DragQueryPoint(hDrop, &Point);
  for(int you = 0; you < NumFiles; i++)
  {
    DragQueryFile(hDrop, i, buff, sizeof(buff));
    DFiles->Add(buff);
  }
  DragFinish(hDrop);
  // wykonujemy zadania związane z listą plików zawartą w DFiles
  delete DFiles;
}
```

Wyjaśnienie tego kodu wykracza poza ramy niniejszego rozdziału. Pomoc środowiska C++Builder stanowi bardzo dobre źródło informacji o zastosowanych metodach.

Możemy się przekonać, że przechwytywanie komunikatów nie jest trudne, gdy zna się zasady ich ustawiania oraz pewne funkcje API systemu Windows. Listę dostępnych struktur komunikatów znajdziemy w pliku *messages.hpp* instalowanym wraz ze środowiskiem programistycznym.

W trakcie projektowania a w trakcie działania programu

Przeprowadziliśmy już kilka sprawdzeń dotyczących tego, czy komponent działa w trybie projektowym, czy wykonywania. Operacje *trybu projektowania* dotyczą zachowania komponentu w trakcie tworzenia projektu w środowisku projektowym. Operacje *wykonywania* dotyczą działania komponentu po uruchomieniu aplikacji.

Obiekt `TComponent` posiada właściwość (zbiór) o nazwie `ComponentState`, składającą się z następujących stałych: `csAncestor`, `csDesigning`, `csDesignInstance`, `csDestroying`, `csFixups`, `csFreeNotification`, `csInline`, `csLoading`, `csReading`, `csWriting` i `csUpdating`. Tabela 4.1 opisuje każdą z nich.

Najbardziej jesteśmy zainteresowani zawartością znacznika `csDesigning`. Jeśli komponent istnieje w środowisku programistycznym (jako część projektu), komponent będzie zawierał tę wartość ustawioną przez cały etap projektowania. Aby sprawdzić, czy komponent wykorzystywany jest w trakcie projektowania, używamy następującego kodu:

```
if(ComponentState.Contains(csDesigning))
  // tutaj znajduje się kod dotyczący etapu projektowania
else
  // tutaj znajduje się kod dotyczący etapu wykonywania
```

Tabela 4.1. *Znaczniki ComponentState*

Znacznik	Zastosowanie
csAncestor	Wskazuje, że komponent został wprowadzony w formularzu-przodku. Ustawiany tylko wtedy, gdy jest ustawiony csDesigning. Ustawiany lub zerowany w metodzie TComponent::SetAncestor().
csDesigning	Wskazuje, że komponent jest modyfikowany w trakcie projektowania. Używany do rozróżnienia między trybem projektowym a wykonywania. Ustawiany lub zerowany w metodzie TComponent::SetDesigning().
csDesignInstance	Wskazuje, że w edytorze projektu komponent jest korzeniem. Na przykład został ustawiony dla ramki w trakcie projektowania, ale nie na ramce zachowującej się jak komponent. Znacznik ten występuje zawsze w połączeniu z csDesigning. Ustawiany lub zerowany w metodzie TComponent::SetDesignInstance().
cDestroying	Wskazuje na usuwanie obiektu. Ustawiany lub zerowany w metodzie TComponent::Destroying().
csFixups	Wskazuje, że komponent jest połączony z komponentem znajdującym się na innym formularzu, który nie został jeszcze wczytany. Znacznik jest zerowany, gdy wszystkie zależności zostaną rozwiązane. Zerowanie odbywa się w funkcji globalnej GlobalFixupReferences().
csFreeNotification	Wskazuje, że komponent wysłał powiadomienie do innych formularzy, że jest usuwany, ale jeszcze nie został zniszczony. Ustawiany metodą TComponent::FreeNotification().
csInline	Wskazuje, że komponent jest komponentem najwyższego poziomu, który może być modyfikowany w trakcie projektowania, a także osadzany w formularzu. Znacznika używamy do identyfikacji zagnieżdżonych ramek w trakcie zapisu lub odczytu. Ustawiany lub zerowany w metodzie SetInline() komponentu. Ustawiany także w metodzie TReader::ReadComponent().
csLoading	Wskazuje, że obiekt wypełniający właśnie wczytuje komponent. Znacznik jest ustawiany, gdy komponent jest tworzony po raz pierwszy, ale jest zerowany po zakończeniu procesu ładowania komponentu i jego potomków (w momencie wywoływania metody Loaded()). Ustawiany w metodach TReader::ReadComponent() i TReader::ReadRootComponent(). Zerowany w metodzie TComponent::Loaded() (więcej informacji na temat obiektów wypełniających w pomocy online środowiska).
csReading	Wskazuje, że komponent odczytuje wartości właściwości ze strumienia. Znacznik csLoading jest zawsze ustawiony, gdy jest ustawiony csReading. Innymi słowy, znacznik csReading jest ustawiony w trakcie pobierania wartości właściwości w czasie wczytywania komponentu. Ustawiany i zerowany w metodach TReader::ReadComponent() i TReader::ReadRootComponent().
csWriting	Wskazuje, że komponent zapisuje wartości właściwości do strumienia. Ustawiany i zerowany w metodzie TWriter::WriteComponent().
csUpdating	Wskazuje, że komponent został zmodyfikowany w odpowiedzi na zmiany wprowadzone w formularzu-przodku. Ustawiony tylko wtedy, gdy jest ustawiony csAncestor. Ustawiany w metodzie TComponent::Updating(), a zerowany w metodzie TComponent::Updated().

Dlaczego chcielibyśmy wykonywać pewien kod komponentu tylko w samej aplikacji? Ponieważ istnieje wiele przypadków, w których jest to potrzebne. Oto kilka z nich.

- Sprawdzenie poprawności właściwości, która posiada zależności dostępne tylko na etapie wykonywania.
- Wyświetlenie komunikatu użytkownikowi, gdy wpisze błędną wartość właściwości.
- Wyświetlenie okna wyboru lub edytora właściwości po uzyskaniu nieodpowiedniej wartości właściwości.

Wiele osób piszących komponenty nie zapewnia użytkownikom takich okien dialogowych ani komunikatów. Jednak takie dodatkowe funkcje mogą uczynić komponent prostszym i bardziej przyjaznym dla użytkownika.

Łączenie komponentów

Łączenie komponentów polega na tym, że komponent ma możliwość odniesienia się do innego komponentu z tego samego projektu lub zmiany tego komponentu. Przykładem może być komponent `TDriveComboBox` z C++Buildera. Ma on właściwość `DirList`, która umożliwia programiście wybranie komponentu `TDirectoryListBox`, dostępnego na tym samym formularzu. Tego rodzaju łączenie umożliwia szybką (automatyczną) zmianę listy katalogów po każdej zmianie napędu. Utworzenie projektu, który wyświetla listę katalogów i plików, wymaga tylko umieszczenia na formularzu trzech komponentów (`TDriveComboBox`, `TDirectoryListBox` i `TFileListBox`) i ustawienia dwóch właściwości. Oczywiście nadal sami musimy napisać kod zdarzeń, aby projekt wykonywał coś sensownego, ale do tego momentu nie musimy napisać ani jednego wiersza kodu.

Zapewnienie łącza do innych komponentów rozpoczynamy od utworzenia właściwości odpowiedniego typu. Jeśli utworzymy właściwość typu `TLabel`, inspektor obiektów wyświetli wszystkie komponenty formularza tego typu. Aby pokazać, jak to działa dla elementów pochodnych, utworzymy prosty komponent umożliwiający dołączenie komponentów `TMemo` lub `TRichEdit`. By to wykonać, musimy najpierw zdać sobie sprawę z tego, że oba komponenty pochodzą od klasy `TCustomMemo`.

Zacznijmy do utworzenia komponentu dziedziczącego po `TComponent` i zawierającego właściwość o nazwie `LinkedEdit` (patrz listing 4.30).

Listing 4.30. *Łączenie komponentów*

```
class PACKAGE TMsgLog : public TComponent
{
private:
  // może to być TMemo lub TRichEdit albo inny komponent potomny
  TCustomMemo *FLinkedEdit;

public:
  __fastcall TMsgLog(TComponent* Owner);
  __fastcall ~TMsgLog(void);
  void __fastcall OutputMsg(const AnsiString Message);

protected:
  virtual void __fastcall Notification(TComponent *AComponent, TOperation Operation);

__published:
  __property TCustomMemo *LinkedEdit = {read = FLinkedEdit, write = FLinkedEdit};
};
```

Kod z listingu 4.30 tworzy komponent z jedną właściwością o nazwie `LinkedEdit`. Musimy jeszcze zadbać o dwie sprawy. Musimy wysyłać komunikaty do komponentu `TMemo` lub `TRichEdit` (jeśli takowy istnieje), a także pamiętać o tym, że istnieje możliwość usunięcia dołączenia przez użytkownika. Metoda `OutputMsg()` służy do wysłania komunikatu do dołączonego komponentu, a `Notification()` do powiadomienia, jeśli został on usunięty.

Podany kod dotyczy wyjścia.

```
void __fastcall TMsgLog::OutputMsg(const AnsiString Message)
{
  if(FLinkedEdit)
    FLinkedEdit->Lines->Add(Message);
}
```

Ponieważ komponenty `TMemo` i `TRichEdit` mają właściwość `Lines`, nie ma potrzeby przeprowadzania żadnego rzutowania. Aby przeprowadzić zadania zależne od komponentu (lub inaczej obsługiwane), korzystamy z kodu przedstawionego na listingu 4.31.

Listing 4.31. *Metoda OutputMsg()*

```
void __fastcall TMsgLog::OutputMsg(const AnsiString Message)
{
  TMemo *LinkedMemo = 0;
  TRichEdit *LinkedRichEdit = 0;

  LinkedMemo = dynamic_cast<TMemo *>(FLinkedEdit);
  LinkedRichEdit = dynamic_cast<TRichEdit *>(FLinkedEdit);

  if(FLinkedMemo)
    FLinkedMemo->Lines->Add(Message);
  else
  {
    FLinkedRichEdit->Font->Color = clRed;
    FLinkedRichEdit->Lines->Add(Message);
  }
}
```

Ostatnie sprawdzenie dotyczy usuwania dołączonego elementu. Wykonujemy to, przeciążając metodę `Notification()` z `TComponent`, co przedstawia listing 4.32.

Listing 4.32. *Metoda Notification()*

```
void __fastcall TMsgLog::Notification(TComponent *AComponent,
TOperation Operation)
{
  // Nie interesuje nas dodawanie sterowania.
  if(Operation != opRemove)
    return;

  // Musimy sprawdzić wszystkie, na wypadek gdyby użytkownik wykonał coś takiego,
  // jak posiadanie tej samej etykiety przypisywanej do kilku właściwości.
  if(AComponent == FLinkedEdit)
    FLinkedEdit = 0;
}
```

Kod z listingu 4.32 przedstawia sposób radzenia sobie z usunięciem innego komponentu. Dwa pierwsze wiersze obrazują wykorzystanie parametru `Operation`.

Najważniejsze są jednak dwa ostatnie wiersze, które porównują wskaźnik `AComponent` z właściwością `LinkedEdit` (wskaźnik na komponent wywodzący się od `TCustomMemo`). Jeśli wskaźniki są takie same, zerujemy wskaźnik `LinkedEdit`. Powoduje to usunięcie referencji z inspektora obiektów, a kod przestaje wskazywać na adres w pamięci, który za chwilę zostanie utracony (w momencie rzeczywistego usunięcia komponentu edycji). Pamiętajmy o tym, że `LinkedEdit = 0` jest równoznaczne `LinkedEdit = NULL`.

Jeśli dołączymy nasz komponent do innego, który posiada pewne zależności (na przykład `TDBDataSet` wymaga połączenia z bazą danych), to na nas spoczywa obowiązek zapewnienia spełnienia tych zależności i ich odpowiedniej obsługi. Dobrze zaprojektowany komponent można poznać po tym, że użytkownik wykonuje minimum zadań, by działał on tak, jak się tego spodziewa.

Łączenie zdarzeń między komponentami

Opisaliśmy łączenie komponentów przez właściwości. Na przykładzie zobrazowaliśmy sposób dołączania właściwości komponentu `TMsgLog` do innego komponentu w taki sposób, aby komunikat został przekazany automatycznie bez potrzeby pisania dodatkowego kodu przez użytkownika.

Teraz przyjrzymy się łączeniu zdarzeń między komponentami. Kontynuując poprzedni przykład, zobrazujemy sposób przechwytywania zdarzenia `OnExit` z dołączonego komponentu (pamiętajmy, że `TMemo` i `TRichEdit` posiadają zdarzenie `OnExit` o typie `TNotifyEvent`), aby wykonać dodatkowe przetwarzanie przed przejściem do kodu użytkownika. Załóżmy, że dołączony komponent nie jest tylko do odczytu, więc użytkownik może coś wpisać w dzienniku zdarzeń; taka sytuacja musi zostać zanotowana jako wpis użytkownika. Przedstawimy sposób przeprowadzenia przechwycenia; napisanie dalszego kodu pozostawiamy czytelnikowi.

Zdarzenia komponentów można zaimplementować różnie w zależności od natury samego zdarzenia. Jeśli komponent zawiera proces iteracyjny, kod może po prostu wywołać procedurę obsługi zdarzenia, gdy opuszcza pętlę. Taką sytuację przedstawia następujący kod.

```
// początek pętli
if(FOnExit)
  FOnExit(this);
endif;
// ...
// koniec pętli
```

Inne zdarzenia mogą być wynikiem odebrania komunikatu. Listing 4.26 przedstawia makro mapy komunikatów dotyczące przyjmowania upuszczonych plików z *Eksploratora Windows*.

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_DROPFILES, TWMDropFiles, WmDropFiles)
END_MESSAGE_MAP(TStringGrid)
```

Jeśli komponent posiada zdarzenie OnDrop, możemy napisać implementację w przedstawiony dalej sposób.

```
void __fastcall TSuperStringGrid::WmDropFiles(TWMDropFiles &Message)
{
if(FOnDrop)
  FOnDrop(this);
endif;

// pozostała część kodu
}
```

Zauważmy, że komponent przechowuje wskaźnik na procedurę obsługi zdarzenia, na przykład FOnExit lub FOnDrop z poprzednich przykładów. Ułatwia nam to utworzenie własnego wskaźnika wskazującego miejsce znajdowania się procedury obsługi, a następnie przekierowanie zdarzenia użytkownika w taki sposób, aby została wywołana metoda wewnętrzna. Ta metoda wewnętrzna wykona najpierw kod użytkownika, a dopiero później kod komponentu (lub na odwrót).

Musimy jednak pamiętać o jednej sprawie w trakcie przekierowywania wskaźników. Logicznym miejscem do wykonywania przekierowań jest metoda Loaded() komponentu, ponieważ jest ona wywoływana, gdy cały komponent został odczytany strumieniowo z formularza; oznacza to, że wszystkie procedury obsługi zdarzeń zostały już przypisane.

Definiujemy metodę Loaded() i wskaźnik na standardowe zdarzenie klasy (zdarzenie jest tego samego typu, jaki zamierzamy przechwycić — w naszym przykładzie dla zdarzenia OnExit jest to typ TNotifyEvent). Potrzebujemy także metody wewnętrznej o takiej samej deklaracji jak procedura obsługi zdarzenia. W przykładzie nazwaliśmy tę metodę MsgLogOnExit(). Metoda ta zostanie wywołana przed zdarzeniem OnExit dołączonego komponentu. Na listingu 4.33 umieszczamy typedef typu TComponent o nazwie Inherited. Powód takiego działania stanie się oczywisty, gdy przyjrzymy się kodowi źródłowemu.

Listing 4.33. *Plik nagłówkowy klasy TMsgLog*

```
class PACKAGE TMsgLog : public TComponent
{
typedef TComponent Inherited;

private:
  TNotifyEvent *FonUsersExit;
  void __fastcall MsgLogOnExit(TObject *Sender);

protected:
  virtual void __fastcall Loaded(void);

// pozostała część kodu
}
```

Przykładowy kod źródłowy przedstawia listing 4.34.

Listing 4.34. *Plik źródłowy klasy TMsgLog*

```
void __fastcall TMsgLog::TMsgLog(TComponent *Owner)
{
  FOnUsersExit = 0;
}

void __fastcall TMsgLog::Loaded(void)
{
  Inherited::Loaded();

  if(!ComponentState.Contains(csDesigning))
  {
    if(FLinkedEdit)
    {
      if(FLinkedEdit->OnExit)
        FOnUsersExit = FLinkedEdit->OnExit;

      FLinkedEdit->OnExit = MsgLogOnExit;
    }
  }
}

void __fastcall TMsgLog::MsgLogOnExit(TObject *Sender)
{
  if(FOnUsersExit)
    FOnUsersExit(this);

//  tutaj umieszczamy dodatkowy kod do wykonania
}
```

W trakcie tworzenia komponentu konstruktor inicjalizuje `FOnUsersExit` na `NULL`. Po pełnym przesłaniu strumieniowym komponentu wywoływane jest zdarzenie `OnLoaded`. Rozpoczyna się ono od wywołania metody nadrzędnej (`typedef` pozwoliło nam zwiększyć czytelność kodu). Następnie sprawdza, czy komponent nie znajduje się w trybie projektowym. Jeśli aplikacja znajduje się w trybie wykonywania, dowiaduje się, czy komponent posiada dołączony do siebie komponent edycyjny. Jeśli tak, sprawdza, czy użytkownik przypisał do tego komponentu procedurę obsługi zdarzenia `OnExit`. Jeśli wszystkie te testy dadzą odpowiedź twierdzącą, ustawiamy wewnętrzny wskaźnik `FOnUsersExit` na adres procedury obsługi zdarzenia napisanej przez użytkownika. Następnie przypisujemy zdarzenie komponentu do naszej metody `MsgLogOnExit()`. Spowoduje to wywoływanie własnej metody za każdym razem, gdy kursor opuści obszar komponentu edycyjnego, nawet jeśli użytkownik nie przypisał żadnej procedury obsługi.

Metoda `MsgLogOnExit()` zaczyna się od określenia, czy użytkownik przypisał procedurę obsługi zdarzenia. Jeśli tak, jest ona wykonywana. Następnie implementujemy własne działania, które chcemy wykonać. Decyzja dotycząca tego, czy kod użytkownika wywołać po czy przed własnym, zależy od rodzaju zdarzenia, na przykład sposobu szyfrowania lub testowania danych.

Tworzenie komponentów widocznych

Dowiedzieliśmy się już, że komponenty stanowią część programu, z którą programista może wchodzić w interakcję. Komponenty mogą być niewizualne (`TOpenDialog` albo `TTable`) lub wizualne (`TListBox` albo `TButton`). Główna różnica między tymi komponentami polega na tym, że komponenty wizualne wyglądają tak samo w trakcie projektowania, jak i działania programu. Gdy dla takiego komponentu zmodyfikujemy właściwości w inspektorze obiektów, musi on zostać ponownie narysowany, by odzwierciedlić zmianę. Kontrolki okien otaczają podstawowe kontrolki Windows, więc często to sam system zajmuje się ich przerysowywaniem. Jeśli jednak komponent nie jest związany z żadną istniejącą kontrolką, sami musimy zadbać o jego rysowanie. W każdym z tych przypadków warto wiedzieć, jakie funkcje wspomagające rysowanie na ekranie udostępnia C++Builder.

Od czego zacząć?

Jednym z najważniejszych kroków dotyczących pisania komponentów jest określenie klasy rodzica, po której będziemy dziedziczyć. Warto w tym celu przejrzeć pliki pomocy lub kod VCL, jeśli go posiadamy. Na pewno nie będzie to czas zmarnowany, ponieważ nie ma nic gorszego od spędzenia kilku godzin, a nawet dni na pracy nad komponentem i stwierdzeniu, że nie posiada on potrzebnych nam funkcji. Jeśli piszemy komponent okienkowy (może przyjmować wejście i posiada uchwyt okna), dziedziczymy po klasie `TCustomControl` lub `TWinControl`. Jeśli kreujemy komponent czysto graficzny, na przykład `TSpeedButton`, dziedziczymy po `TGraphicsControl`. W przypadku pisania komponentów wizualnych istnieje co najwyżej tylko kilka ograniczeń, a przykłady różnych komponentów wraz z ich kodami źródłowymi można znaleźć na witrynie *http://www.torry.net* (największy zbiór) lub na stronach wchodzących w skład C++Builder Programmer's Webring — początek pod adresem *http://www.temporaldoorway.com/programming/cbuilder/index.htm*.

Klasa TCanvas

Obiekt `TCanvas` to otoczenie kontekstu urządzenia. Zawiera różne narzędzia dotyczące rysowania złożonych kształtów i grafiki na ekranie. Obiekt ten możemy uzyskać przez właściwość `Canvas` większości komponentów. Niestety, niektóre komponenty rysuje system, więc właściwość ta nie jest dla nich dostępna. Istnieją jednak sposoby obejścia tego problemu, co wkrótce pokażemy. Obiekt `TCanvas` zawiera kilka metod dotyczących rysowania linii, kształtów i grafiki.

Listing 4.35 przedstawia przykład kodu, który rysuje przekątną z lewego górnego do prawego dolnego narożnika obszaru okna. Metoda `LineTo()` rysuje linię od aktualnego położenia kursora rysowania do współrzędnych określonych w zmiennych `X` i `Y`. Na początku metodą `MoveTo()` określamy początek linii.

Listing 4.35. *Rysowanie linii za pomocą MoveTo() i LineTo()*

```
Canvas->MoveTo(0, 0);
int X = ClientRect.Right;
int Y = ClientRect.Bottom;
Canvas->LineTo (X, Y);
```

Listing 4.36 używa metody `Frame3D()`, aby narysować ramkę naokoło obszaru okna, co nadaje mu wygląd przycisku.

Listing 4.36. *Tworzenie wyglądu przycisku*

```
int PenWidth = 2;
TColor Top = clBtnHighlight;
TColor Bottom = clBtnShadow;
Frame3D(Canvas, ClientRect, Top, Bottom, PenWidth);
```

Możliwe jest też stosowanie funkcji API w połączeniu z `TCanvas` w celu uzyskania pewnych efektów. Niektóre metody używają kontekstu urządzenia kontrolki, choć nie zawsze konieczne jest jego pobranie w celu wywołania funkcji API, która go potrzebuje. Aby uzyskać kontekst kontrolki, stosujemy metodę `GetDC()`.

`HDC` to typ danej zwracanej przez wywołanie `GetDC()`. Jest to po prostu uchwyt kontekstu urządzenia równoważny właściwości `Handle` z `TCanvas`.

Listing 4.37 używa formularza z `TPaintBox` (korzystamy z `TPaintBox`, ponieważ jego właściwość `Canvas` jest dostępna publicznie) i rysuje elipsę, używając funkcji API `RoundRect()`. Komponent `TPaintBox` umieszczamy w dowolnym miejscu formularza. Kod zostanie umieszczony w procedurze obsługi zdarzenia `OnPaint` dla `TPaintBox`. Cały projekt znajduje się w katalogu *Rozdzial4\PaintBox1* na płycie dołączonej do książki. Plik projektu nosi nazwę *Project1.bpr*.

Listing 4.37. *Korzystanie z funkcji API dotyczących rysowania*

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{

  TRect Rect = PaintBox1->ClientRect;

  int nLeftRect, nTopRect, nRightRect, nBottomRect, nWidth, nHeight;

  nLeftRect = Rect.Left;
  nTopRect = Rect.Top;
  nRightRect = Rect.Right;
  nBottomRect = Rect.Bottom;
  nWidth = Rect.Right - Rect.Left + 50;
  nHeight = Rect.Bottom - Rect.Top;

  if(RoundRect(
      PaintBox1->Canvas->Handle, // uchwyt kontekstu urządzenia
      nLeftRect,  // wsp. x prostokąta otaczającego lewy górny narożnik
      nTopRect,   // wsp. y prostokąta otaczającego lewy górny narożnik
      nRightRect, // wsp. x prostokąta otaczającego prawy dolny narożnik
      nBottomRect,// wsp. y prostokąta otaczającego prawy dolny narożnik
      nWidth,     // szerokość elipsy używanej do narysowania zaokrąglonych narożników
      nHeight     // wysokość elipsy używanej do narysowania zaokrąglonych narożników
      ) == 0)
    ShowMessage("Błąd RoundRect...");
}
```

Spróbujmy zmienić wartości zmiennych `nWidth` i `nHeight`. Zaczynamy od zera; prostokąt będzie miał ostre narożniki. Po zwiększeniu wartości tych dwóch zmiennych narożnik prostokąta zaczną się zaokrąglać. Przy użyciu tej funkcji (lub jej podobnych) możemy kreować zaokrąglone lub eliptyczne przyciski. W dalszej części rozdziału przedstawimy kolejne przykłady. Więcej informacji na temat funkcji API znajduje się w pliku pomocy Win32 (plik *win32.hlp*).

Grafika w komponentach

Grafika już na stałe zagościła w komponentach. Przypomnijmy sobie `TSpeedButton` lub `TBitButton`, a przecież istnieje jeszcze wiele darmowych i płatnych komponentów udostępniających różnego rodzaju grafikę. Grafika zwiększa wizualną atrakcyjność komponentów. Na szczęście C+Builder zapewnia kilka klas obsługujących popularne formaty: bitmapy, ikony, pliki JPEG i GIF. Tradycyjnie w grafice komponentów korzysta się z bitmapy pozaekranowej, na której odbywa się rysowanie, a następnie kopiuje się tę bitmapę na ekran. Redukuje to efekt migotania, ponieważ bitmapa jest rysowana tylko raz. Ma to znaczenie szczególnie wtedy, gdy pracujemy ze złożonymi kształtami lub obrazami. Klasa `TBitmap` ma właściwość `Canvas`, będącą obiektem `TCanvas`. Umożliwia to rysowanie na ekranie dowolnych kształtów i grafiki.

Podany dalej przykład korzysta z formularza z komponentem `TPaintBox`. Do narysowania obrazu podobnego do `TSpeedButton` z właściwością `Flat` ustawioną na `true` używamy obiektu `TBitmap`. Następnie w jednym kroku kopiujemy bitmapę na ekran. W tym przykładzie dodajemy `TButton`, który zmienia wygląd obrazu z wyciśniętego na wciśnięty. Pełny projekt znajduje się w katalogu *Rozdzial4\PaintBox2* na płycie dołączonej do książki. Plik projektu nosi nazwę *Project1.bpr*. Najpierw przyjrzyjmy się plikowi nagłówkowemu z listingu 4.38.

Listing 4.38. *Tworzenie wyglądu dla wciśnięcia i wyciśnięcia*

```
class TForm1 : public TForm
{
__published:    // IDE-managed Components
  TPaintBox *PaintBox1;
  TButton *Button1;
private:        // User declarations
  bool IsUp;
public:         // User declarations
  __fastcall TForm1(TComponent* Owner);
};
```

Deklarujemy zmienną logiczną `IsUp`, której będziemy używać do zmiany kolorów rozjaśnienia i cienia oraz tekstu przycisku. Jeśli `IsUp` wynosi `true`, obraz jest w stanie wyciśniętym. Jeśli `IsUp` wynosi `false`, obraz jest wciśnięty. Ponieważ `IsUp` to pole składowe klasy, zostanie zainicjalizowana na `false` w trakcie tworzenia formularza. Właściwość `Caption` dla `Button1` powinna zostać ustawiona na stan `Góra` w inspektorze obiektów.

Zdarzenie `OnClick` dla przycisku jest proste. Zmienia wartość zmiennej `IsUp`, modyfikuje właściwość `Caption` przycisku i wywołuje metodę `Repaint()` dla `TPaintBox`, aby ponownie narysować obraz. Procedurę obsługi zdarzenia przedstawia listing 4.39.

Listing 4.39. *Metoda Button1Click()*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  IsUp = !IsUp;
  Button1->Caption = (IsUp) ? "Dół" : "Góra";
  PaintBox1->Repaint();
}
```

Metoda prywatna `SwapColors()` odpowiada za zmianę kolorów rozjaśnienia i cienia w zależności od wartości zmiennej `IsUp` (patrz listing 4.40).

Listing 4.40. *Metoda SwapColors()*

```
void __fastcall TForm1::SwapColors(TColor &Top, TColor &Bottom)
{
  Top = (IsUp) ? clBtnHighlight : clBtnShadow;
  Bottom = (IsUp) ? clBtnShadow : clBtnHighlight;
}
```

Ostatni krok to utworzenie procedury obsługi zdarzenia `OnPaint` dla `TPaintBox`. Przedstawia to listing 4.41.

Listing 4.41. *Rysowanie przycisku*

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
  TColor TopColor, BottomColor;
  TRect Rect;

  Rect = PaintBox1->ClientRect;

  Graphics::TBitmap *bit = new Graphics::TBitmap;
  bit->Width = PaintBox1->Width;
  bit->Height = PaintBox1->Height;
  bit->Canvas->Brush->Color = clBtnFace;
  bit->Canvas->FillRect(Rect);
  SwapColors(TopColor, BottomColor);
  Frame3D(bit->Canvas, Rect, TopColor, BottomColor, 2);
  PaintBox1->Canvas->Draw(0, 0, bit);
  delete bit;
}
```

W listingu 4.42 pójdziemy o krok dalej i zobrazujemy sposób korzystania z plików bitmap i rysowania linii w oknie. Wiele komponentów przyciskowych zawiera nie tylko linie i zarysy określające kształt, ale także ikony, bitmapy oraz tekst. Przykład ten będzie nieco bardziej złożony, ponieważ wymaga drugiego obiektu `TBitmap`, aby wczytać plik grafiki. Ustalimy położenie grafiki, skopiujemy ją do pierwszej bitmapy, a następnie wszystko przeniesiemy na obszar okna. Cały projekt znajduje się w katalogu *Rozdzial4\PaintBox2* na płycie dołączonej do książki. Plik projektu nosi nazwę *Project1.bpr*.

Listing 4.42. *Korzystanie z bitmap i linii*

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
  TColor TopColor, BottomColor;
  TRect Rect, gRect;

  Rect = PaintBox1->ClientRect;

  Graphics::TBitmap *bit = new Graphics::TBitmap;
  Graphics::TBitmap *bitFile = new Graphics::TBitmap;

  bitFile->LoadFromFile("geom1b.bmp");
  // rozmiar bitmapy pozaekranowej to rozmiar obszaru okna
  bit->Width = PaintBox1->Width;
  bit->Height = PaintBox1->Height;

  // wypełnianie płótna kolorem pędzla
  bit->Canvas->Brush->Color = clBtnFace;
  bit->Canvas->FillRect(Rect);

  // umieszczenie drugiej struktury TRect wyśrodkowanej w Rect
  gRect.Left = ((Rect.Right - Rect.Left) / 2) - (bitFile->Width / 2);
  gRect.Top = ((Rect.Bottom - Rect.Top) / 2) - (bitFile->Height / 2);

  // przesuwanie wewnętrznego prostokąta o jeden piksel w górę i w lewo,
  // aby uzyskać efekt góra-dół
  gRect.Top += (IsUp) ? 0 : 1;
  gRect.Left += (IsUp) ? 0 : 1;

  gRect.Right = bitFile->Width + gRect.Left;;
  gRect.Bottom = bitFile->Height + gRect.Top;

  // kopiowanie bitmapy do obiektu bitmapy pozaekranowej z wykorzystaniem przezroczystości
  bit->Canvas->BrushCopy(gRect, bitFile,
    TRect(0,0,bitFile->Width, bitFile->Height), bitFile->TransparentColor);

  // rysowanie krawędzi
  SwapColors(TopColor, BottomColor);
  Frame3D(bit->Canvas, Rect, TopColor, BottomColor, 2);

  // kopiowanie bitmapy pozaekranowej do obszaru okna
  BitBlt(PaintBox1->Canvas->Handle, 0, 0, PaintBox1->ClientWidth,
    PaintBox1->ClientHeight, bit->Canvas->Handle, 0, 0, SRCCOPY);

  delete bitFile;
  delete bit;
}
```

Odpowiedzi na komunikaty myszy

Na ogół komponenty graficzne dziedziczą po klasie `TGraphicsControl`, która zapewnia dostęp do obszaru okna i obsługę komunikatów `WM_PAINT`. Pamiętajmy, że komponenty nieokienkowe nie mają potrzeby przejmowania wejścia ani uzyskiwania uchwytu okna.

Choć tego rodzaju komponenty nie mogą przyjmować wejścia, VCL umożliwia przechwytywanie dla nich komunikatów zdarzeń myszy.

Jeśli na przykład właściwość `Flat` z `TSpeedButton` jest ustawiona na `true`, przycisk pokazuje brzegi, gdy znajdzie się nad nim kursor myszy, a przestaje je pokazywać, gdy kursor się oddali. Takie działanie to odpowiedź na dwa komunikaty — odpowiednio `CM_ONMOUSEENTER` i `CM_ONMOUSELEAVE`. Komunikaty te przedstawia listing 4.43.

Listing 4.43. *Komunikaty CM_ONMOUSEENTER i CM_ONMOUSELEAVE*

```
void __fastcall CMMouseEnter(TMessage &Msg); // CM_MOUSEENTER
void __fastcall CMMouseLeave(TMessage &Msg); // CM_MOUSELEAVE

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_MOUSEENTER, TMessage, CMMouseEnter)
  MESSAGE_HANDLER(CM_MOUSELEAVE, TMessage, CMMouseLeave)
END_MESSAGE_MAP(TNazwaKomponentuBazowego)
```

Innym bardzo ważnym komunikatem jest `CM_ENABLEDCHANGED`. Właściwość `Enabled` z `TGraphicsControl` zadeklarowana jest jako publiczna, a metoda odpowiedzialna za jej ustawienie po prostu wysyła komunikat `CM_ENABLEDCHANGED`, aby zostały przedsięwzięte odpowiednie działania, na przykład wyświetlenie tekstu w sposób wyszarzony lub niezgłaszanie zdarzeń. Jeśli nasz komponent ma posiadać możliwość włączania i wyłączania, powinniśmy ponownie zadeklarować właściwość jako publiczną w pliku nagłówkowym komponentu, a następnie zdefiniować metodę i procedury obsługi komunikatu. Jeśli tego nie zrobimy, użytkownik będzie mógł co prawda zmieniać wartość właściwości w trakcie działania aplikacji, ale nie spowoduje to żadnych zmian komponentu. Komunikat `CM_ENABLEDCHANGED` przedstawia listing 4.44.

Listing 4.44. *Komunikat CM_ENABLEDCHANGED*

```
void __fastcall CMEnabledChanged(TMessage &Msg);
__published:
  __property Enabled;

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_ENABLEDCHANGED, TMessage, CMEnabledChanged)
END_MESSAGE_MAP(TNazwaKomponentu)
```

Pozostałe zdarzenia myszy, na przykład `OnMouseUp`, `OnMouseDown` i `OnMouseOver`, są zadeklarowane w sekcji chronionej z `TControl`, więc skorzystanie z nich wymaga przesłonięcia odpowiedniej metody. Pamiętajmy jednak o tym, by w trakcie przysłaniania tych zdarzeń zadeklarować je w sekcji chronionej pliku nagłówkowego komponentu, co umożliwi ich zmianę w klasach pochodnych. Odpowiedni kod przedstawia listing 4.45.

Listing 4.45. *Przysłanianie zdarzeń myszy z klasy TControl*

```
private:
  TMmouseEvent FOnMouseUp;
  TMouseEvent FOnMouseDown;
  TMouseMoveEvent FOnMouseMove;
```

```
protected:
  void __fastcall MouseDown(TMouseButton Button, TShiftState Shift, int X, int Y);
  void __fastcall MouseMove(TshiftState Shift, int X, int Y);
  void __fastcall MouseUp(TMouseButton Button, TShiftState Shift, int X, int Y);

__published:
  __property TMouseEvent OnMouseUp = {read=FOnMouseUp, write=FOnMouseUp};
  __property TMouseEvent OnMouseDown = {read=FOnMouseDown, write=FOnMouseDown};
  __property TMouseMoveEvent OnMouseMove = {read=FOnMouseMove, write=FOnMouseMove};
```

W przedstawionych wcześniej przykładowych projektach procedura obsługi zdarzenia została utworzona dla zdarzenia `OnPaint` z `TPaintBox`. Zdarzenie to jest wywoływane po otrzymaniu komunikatu `WM_PAINT`. `TGrapicsControl` przechwytuje komunikat i zapewnia metodę wirtualną `Paint()`, którą można przesłonić w komponentach pochodnych w celu rysowania na ekranie. Ewentualnie można skorzystać ze zdarzenia `OnPaint`, jak w `TPaintBox`.

Te i inne komunikaty są zdefiniowane w pliku nagłówkowym *messages.hpp*. Jeśli mamy kod źródłowy VCL, warto poświęcić trochę czasu, aby dowiedzieć się, które komunikaty i zdarzenia są dostępne oraz jakie metody można przesłonić.

Łączymy wszystko razem

W tym podrozdziale połączymy omówione wcześniej techniki, kreując podstawowy komponent, który można rozszerzać i wzbogacać. Choć komponent ten nie jest kompletny, można go zainstalować na palecie komponentów i używać w aplikacjach. Jako twórcy komponentów nigdy nie powinniśmy niczego pozostawiać przypadkowi. Im prostszy w użyciu będzie komponent, tym częściej inni będą z niego korzystali. Listingi 4.46 i 4.47 przedstawiają kod komponentu przycisku, który działa w sposób podobny do `TSpeedButton` i zawiera grafikę oraz tekst. Po opisaniu kodu zajmiemy się omówieniem kilku oczywistych usprawnień komponentu. Kod źródłowy jest także dostępny na dołączonej płycie CD-ROM w katalogu *Rozdzial4\ExampleButton*.

Listing 4.46. *Plik nagłówkowy TExampleButton, plik ExampleButton.h*

```
#ifndef ExampleButtonH
#define ExampleButtonH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------

enum TExButtonState {esUp, esDown, esFlat, esDisabled};

class PACKAGE TExampleButton : public TGraphicControl
{
private:

  Graphics::TBitmap *FGlyph;
  AnsiString FCaption;
  TImageList *FImage;
```

```
  TExButtonState FState;
  bool FMouseInControl;
  TNotifyEvent FOnClick;
  void __fastcall SetGlyph(Graphics::TBitmap *Value);
  void __fastcall SetCaption(AnsiString Value);
  void __fastcall BeforeDestruction(void);
  void __fastcall SwapColors(TColor &Top, TColor &Bottom);
  void __fastcall CalcGlyphLayout(TRect &r);
  void __fastcall CalcTextLayout(TRect &r);
  MESSAGE void __fastcall CMMouseEnter(TMessage &Msg);
  MESSAGE void __fastcall CMMouseLeave(TMessage &Msg);
  MESSAGE void __fastcall CMEnabledChanged(TMessage &Msg);
protected:

  void __fastcall Paint(void);
  DYNAMIC void __fastcall MouseDown(TMouseButton Button, TShiftState Shift,
    int X, int Y);
  DYNAMIC void __fastcall MouseUp(TMouseButton Button, TShiftState Shift,
    int X, int Y);
public:
  __fastcall TExampleButton(TComponent* Owner);
__published:

  __property AnsiString Caption = {read=FCaption, write=SetCaption};
  __property Graphics::TBitmap * Glyph = {read=FGlyph, write=SetGlyph};
  __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_MOUSEENTER, TMessage, CMMouseEnter)
  MESSAGE_HANDLER(CM_MOUSELEAVE, TMessage, CMMouseLeave)
  MESSAGE_HANDLER(CM_ENABLEDCHANGED, TMessage, CMEnabledChanged)
END_MESSAGE_MAP(TGraphicControl)
};
//---------------------------------------------------------------------------
#endif
```

Listing 4.47. *Plik źródłowy TExampleButton, plik ExampleButton.cpp*

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "ExampleButton.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//

static inline void ValidCtrCheck(TExampleButton *)
{
  new TExampleButton(NULL);
}
//---------------------------------------------------------------------------
__fastcall TExampleButton::TExampleButton(TComponent* Owner)
  : TGraphicControl(Owner)
```

```
{
  SetBounds(0,0,50,50);
  ControlStyle = ControlStyle << csReplicatable;
  FState = esFlat;

}
//---------------------------------------------------------------------------
namespace Examplebutton
{
  void __fastcall PACKAGE Register()
  {
     TComponentClass classes[1] = {__classid(TExampleButton)};
     RegisterComponents("Samples", classes, 0);
  }
}
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::CMMouseEnter(TMessage &Msg)
{
  if(Enabled)
    {
    FState = esUp;
    FMouseInControl = true;
    Invalidate();
    }
}
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::CMMouseLeave(TMessage &Msg)
{
  if(Enabled)
    {
    FState = esFlat;
    FMouseInControl = false;
    Invalidate();
    }
}
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::CMEnabledChanged(TMessage &Msg)
{
  FState = (Enabled) ? esFlat : esDisabled;
  Invalidate();
}

// ---------------------------------------------------------------------------
void __fastcall TExampleButton::MouseDown(TMouseButton Button, TShiftState
  Shift, int X, int Y)
{
  if(Button == mbLeft)
    {
    if(Enabled && FMouseInControl)
      {
      FState = esDown;
      Invalidate();
      }
    }
}
```

```
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::MouseUp(TMouseButton Button, TShiftState
  Shift, int X, int Y)
{
  if(Button == mbLeft)
    {
    if(Enabled && FMouseInControl)
      {
      FState = esUp;
      Invalidate();
      if(FOnClick)
        FOnClick(this);
      }
    }
}
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::SetGlyph(Graphics::TBitmap * Value)
{
  if(Value == NULL)
    return;

  if(!FGlyph)
    FGlyph = new Graphics::TBitmap;
  FGlyph->Assign(Value);
  Invalidate();
}

// ---------------------------------------------------------------------------
void __fastcall TExampleButton::SetCaption(AnsiString Value)
{
  FCaption = Value;
  Invalidate();
}
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::SwapColors(TColor &Top, TColor &Bottom)
{
  if(ComponentState.Contains(csDesigning))
    {
    FState = esUp;
    }

  Top = (FState == esUp) ? clBtnHighlight : clBtnShadow;
  Bottom = (FState == esDown) ? clBtnHighlight : clBtnShadow;
}

// ---------------------------------------------------------------------------
void __fastcall TExampleButton::BeforeDestruction(void)
{
  if(FImage)
    delete FImage;

  if(FGlyph)
    delete FGlyph;
}
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::Paint(void)
{
```

```
  TRect cRect, tRect, gRect;
  TColor TopColor, BottomColor;

  Canvas->Brush->Color = clBtnFace;
  Canvas->FillRect(ClientRect);
  cRect = ClientRect;
  Graphics::TBitmap *bit = new Graphics::TBitmap;
  bit->Width = ClientWidth;
  bit->Height = ClientHeight;
  bit->Canvas->Brush->Color = clBtnFace;
  bit->Canvas->FillRect(cRect);

  if(FGlyph)
    if(!FGlyph->Empty)
      {
      CalcGlyphLayout(gRect);
      bit->Canvas->BrushCopy(gRect, FGlyph,
        Rect(0,0,FGlyph->Width,FGlyph->Height), FGlyph->TransparentColor);
      }
  if(!FCaption.IsEmpty())
    {
    CalcTextLayout(tRect);
    bit->Canvas->TextRect(tRect, tRect.Left,tRect.Top, FCaption);
    }

  if(FState == esUp || FState == esDown)
    {
    SwapColors(TopColor, BottomColor);
    Frame3D(bit->Canvas, cRect, TopColor, BottomColor, 1);
    }


  BitBlt(Canvas->Handle, 0, 0, ClientWidth, ClientHeight, bit->Canvas->Handle, 0, 0,
      SRCCOPY);

  delete bit;

}
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::CalcGlyphLayout(TRect &r)
{

  int TotalHeight=0;
  int TextHeight=0;

  if(!FCaption.IsEmpty())
    TextHeight = Canvas->TextHeight(FCaption);

  // Dodawane 5 może być właściwością odstępu, ale dla uproszczenia po prostu
  // dodajemy 5.
  TotalHeight = FGlyph->Height + TextHeight + 5;

  r = Rect((ClientWidth/2)-(FGlyph->Width/2),
          ((ClientHeight/2)-(TotalHeight/2)), FGlyph->Width +
          (ClientWidth/2)-(FGlyph->Width/2), FGlyph->Height +
          ((ClientHeight/2)-(TotalHeight/2)));
}
```

```
// ---------------------------------------------------------------------------
void __fastcall TExampleButton::CalcTextLayout(TRect &r)
{
  int TotalHeight=0;
  int TextHeight=0;
  int TextWidth=0;
  TRect temp;

  if(FGlyph)
    TotalHeight = FGlyph->Height;

  TextHeight = Canvas->TextHeight(FCaption);
  TextWidth = Canvas->TextWidth(FCaption);

  TotalHeight += TextHeight + 5;

  temp.Left = 0;
  temp.Top = (ClientHeight/2)-(TotalHeight/2);
  temp.Bottom = temp.Top + TotalHeight;
  temp.Right = ClientWidth;

  r = Rect(((ClientWidth/2) - (TextWidth/2)), temp.Bottom-TextHeight,
      ((ClientWidth/2)-(TextWidth/2))+TextWidth, temp.Bottom);
}
```

Prezentowany kod publikuje tylko zdarzenie `OnClick`. W rzeczywistym komponencie zapewne zostałyby także opublikowane zdarzenia `OnMouseUp`, `OnMouseDown` i `OnMouseMove`. Publikowane są dwie właściwości, `Caption` i `Glyph`, ale powinniśmy jeszcze posiadać właściwość `Font`, aby umożliwić użytkownikom zmianę czcionki tekstu.

Dobrym pomysłem jest też przechwytywanie komunikatu `CM_FONTCHANGED`, aby położenie wyglądu przycisku i napisu zostało ponownie odrysowane po zmianie czcionki. W obliczaniu położenia obrazu i tekstu używamy wartości 5 pikseli jako odległości między tymi dwoma elementami. Warto utworzyć osobną właściwość, która umożliwi użytkownikowi dobranie tej wartości.

Na listingu 4.47 przyjrzyjmy się metodzie `write` właściwości `Glyph`, czyli `SetGlyph()`. Jeśli przypiszemy do niej wskaźnik równy `NULL`, metoda zakończy działanie bez wykonywania żadnych zadań. Wydaje się to logicznym zachowaniem dla tego rodzaju właściwości, ale gdy przypiszemy obraz, nie istnieje żaden sposób na jego usunięcie. Innymi słowy, aby wyświetlić tylko tekst, musimy usunąć aktualny komponent i utworzyć nowy.

Przyjrzyjmy się jeszcze zmiennej logicznej `FMouseInControl`. Ponieważ kontrolka odpowiada na zdarzenia myszy, warto je śledzić. Zmienna określa, czy kursor myszy znajduje się w obrębie kontrolki. Bez tej zmiennej pewne metody mogłyby działać błędnie, ponieważ komponent otrzymuje zdarzenia myszy nawet wtedy, gdy kursor nie znajduje się bezpośredno nad nim. Jeśli na przykład użytkownik nacisnął i przytrzymał przycisk myszy, a następnie przesunął kursor nad komponent i zwolnił przycisk, zostanie wywołana metoda `CMMouseUp()`, ale komponent nie będzie wiedział, że kursor znalazł się nad nim. W efekcie spowoduje to ponowne narysowanie przycisku w stanie uniesienia, a nie spowoduje odrysowania, jeśli przeniesiemy mysz w inne miejsce, a następnie wrócimy lub klikniemy przycisk. Przed takim działaniem zabezpiecza zmienna `FMouseInControl`.

Na listingu 4.47 rysowanie kształtu przycisku odbywa się przy użyciu metody `Frame3D()`. Jeśli do kodu źródłowego dołączymy plik nagłówkowy *Buttons.hpp*, będziemy mogli korzystać z innej metody rysowania kształtów, `DrawButtonFace()`, przedstawionej na listingu 4.48.

Listing 4.48. *Metoda DrawButtonFace()*

```
TRect DrawButtonFace(TCanvas *Canvas, const TRect Client,
int BevelWidth, TButtonStyle Style, bool IsRounded, bool IsDown,
bool IsFocused);
```

Metoda `DrawButtonFace()` rysuje kształt przycisku o wymiarach `Client` na obszarze określonym parametrem `Canvas`. Działanie pewnych właściwości zależy od wartości innych właściwości. Na przykład parametry `BevelWidth` i `IsRounded` wydają się działać tylko dla parametru `Style` ustawionego na `bsWin31`. `IsFocused` nie wpływa na wygląd graficzny przycisku.

Metoda `DrawButtonFace()` korzysta z funkcji API `DrawEdge()` (patrz pomoc Win32 dołączona do C++Buildera). Można jej używać także we własnych metodach rysowania.

Modyfikacja komponentów okienkowych

Wspomnieliśmy już, że komponenty okienkowe to otoczenia standardowych kontrolek Windows. Takie komponenty wiedzą, jak mają się rysować, więc nie musimy się o to martwić. W przypadku takich kontrolek będziemy przede wszystkim zmieniać ich zachowanie, a nie wygląd. Na szczęście VCL udostępnia metody chronione tych komponentów, więc zadanie to nie stanowi większego problemu — wystarczy przysłonić te metody.

W tym ostatnim przykładzie wykorzystamy kilka opisanych wcześniej technik, aby utworzyć wersję komponentu `TFileListBox` bardziej przyjazną od dostarczanej z C++Builderem. Zanim zaczniemy pisać kod, warto zapisać sobie to, co zamierzamy wykonać. Pamiętajmy, że zależy nam na jak największym ułatwieniu korzystania z komponentu i odciążeniu użytkownika od pisania kodu wspólnego dla wszystkich komponentów zawierających listy plików. Dalej przedstawiamy listę zmian, których dokonamy w komponencie.

- Wyświetlanie odpowiedniej ikony dla każdego pliku.
- Umożliwienie uruchomienia aplikacji lub otwarcia dokumentu po dwukrotnym kliknięciu na jego nazwie.
- Umożliwienie użytkownikowi dodawania konkretnego elementu do listy.
- Zaznaczanie elementu po kliknięciu na nim prawym klawiszem myszy.
- Wyświetlenie poziomego paska przewijania, gdy nazwa elementu jest dłuższa niż wymiary listy.
- Zachowanie zgodności z `TDirectoryListBox`.

Skoro wiemy już, co komponent powinien robić, musimy zdecydować, po której klasie bazowej będziemy dziedziczyć. Wspomnieliśmy wcześniej, że C++Builder zapewnia klasy,

z których można kreować nowe komponenty, ale w przypadku naszego komponentu musimy obrać inną strategię. `TDirectoryListBox` i `TFileListBox` są połączone ze sobą właściwością `FileList` z `TDirectoryListBox`. Właściwość ta jest zadeklarowana jako wskaźnik na `TFileListBox`, więc komponenty dziedziczące po `TCustomListBox` lub `TListBox` nie będą przez nią widziane. Aby zachować zgodność z `TDirectoryListBox`, musimy dziedziczyć po `TFileListBox`. Na szczęście metody wykorzystywane do odczytu nazw plików są metodami chronionymi, więc możemy je przysłonić we własnym komponencie.

Teraz rozważymy zmiany, jakich chcemy dokonać w komponencie, i zadeklarujemy nowe właściwości, metody i zdarzenia. Po pierwsze, chcemy użytkownikom umożliwić uruchomienie aplikacji lub otwarcie dokumentu po dwukrotnym kliknięciu elementu. Deklarujemy właściwość logiczną, która umożliwi użytkownikowi włączenie lub wyłączenie tej funkcji. Jej kod przedstawia podany wiersz.

```
__property bool CanLaunch = {read=FCanLaunch, write=FCanLaunch, default=true};
```

Gdy użytkownik dwukrotnie kliknie listę, wysyłany jest komunikat `WM_LBUTTONDBLCLK`. `TCustomListBox` zawiera metodę chronioną wywoływaną w odpowiedzi na ten komunikat. Listing 4.49 przedstawia sposób przysłonięcia metody `DblClick()` w celu uruchomienia aplikacji lub otwarcia dokumentu.

Listing 4.49. *Metoda DblClick()*

```
void __fastcall TSHFileListBox::DblClick(void)
{
  if(FCanLaunch)
  {
    int ii=0;
    // przejście przez listę i sprawdzenie, który element został zaznaczony
    for(ii=0; ii < Items->Count; ii++)
    {
      if(Selected[ii])
      {
        AnsiString str = Items->Strings[ii];
        ShellExecute(Handle, "open", str.c_str(), 0, 0, SW_SHOWDEFAULT);
      }
    }
  }
  // zgłoszenie zdarzenia OnDblClick
  if(FOnDblClick)
    FOnDblClick(this);
}
```

Jeśli zmienna `FCanLaunch` wynosi `true`, musimy najpierw znaleźć zaznaczony element, a następnie użyć funkcji API `ShellExecute()`, aby uruchomić aplikację. Metoda ta zgłasza również zdarzenie `OnDblClick` deklarowane w następującym kodzie.

```
private:
  TNotifyEvent FOnDblClick;

__published:
  __property TNotifyEvent OnDblClick = {read=FOnDblClick, write=FOnDblClick};
```

Ponieważ zdarzenie `OnDblClick` nie musi przekazywać żadnych informacji, deklarujemy je jako zmienną typu `TNotifyEvent`. Oczywiście zachowanie to można zmienić, gdy zajdzie taka potrzeba, ale na razie zaproponowane działanie jest wystarczające. Teraz zajmijmy się problemem zaznaczania elementów prawym klawiszem myszy. Najpierw musimy zadeklarować nową właściwość, używając następującego kodu:

```
__property bool RightBtnClick = {read=FRightBtnSelect, write=FRightBtnSelect,
default=true};
```

Zauważmy, że właściwość odnosi się bezpośrednio do pola klasy — nie istnieją metody ustawiania i pobierania. Wykonujemy to w ten sposób, gdyż pola użyjemy w zdarzeniu `MouseUp()` w celu określenia, czy należy zaznaczyć element. Listing 4.50 przedstawia kod tej metody.

Listing 4.50. *Metoda MouseUp()*

```
//---------------------------------------------------------------------------
void __fastcall TSHFileListBox::MouseUp(TMouseButton Button, TShiftState Shift,
  int X, int Y)
{
  if(!FRightBtnSel)
    return;

  TPoint ItemPos = Point(X,Y);
  // czy mysz znajduje się nad elementem?
  int Index = ItemAtPos(ItemPos, true);
  // jeśli nie, wracamy
  if(Index == -1)
    return;
  // w przeciwnym razie zaznaczamy element
  Perform(LB_SETCURSEL, (WPARAM)Index, 0);
}
```

Kod z listingu 4.50 jest bardzo prosty. Najpierw sprawdzamy zmienną `FRightBtnSel`, aby określić, czy możemy zaznaczać elementy. Następnie konwertujemy współrzędne myszy na strukturę `TPoint`. Aby dowiedzieć się, nad którym elementem znajduje się kursor myszy, korzystamy z metody `ItemAtPos()` z `TCustomListBox`, która przyjmuje utworzoną strukturę `TPoint` i wartość logiczną określającą, czy w przypadku znajdowania się kursora poza elementami listy zwracaną wartością ma być –1, czy może ostatni element listy. Przekazujemy `true`, więc zwracaną wartością dla takiej sytuacji jest –1. Powoduje to zakończenie działania metody. Można też przekazać `false` i usunąć warunek `if` sprawdzający wartość –1. Następnie metodą `Perform()` wymuszamy takie działanie kontrolki, jakby otrzymała komunikat okna. Pierwszy parametr to symulowany komunikat. `LB_SETCURSEL` informuje listę, że mysz spowodowała zmianę zaznaczenia. Drugi parametr to indeks zaznaczanego elementu. Ostatniego parametru nie używamy, więc ustawiamy go na 0.

Teraz zajmiemy się możliwością dodania nowego elementu przez użytkownika. `TFileListBox` zawiera właściwość `Mask`, która pozwala określić rozszerzenia nazw plików możliwe do umieszczenia na liście. Możliwa jest ponowna deklaracja właściwości i zapewnienie metod odczytu i zapisu, które zajmą się filtrowaniem nazw plików w zależności od wartości maski. Można też pozwolić użytkownikowi na napisanie

własnego zdarzenia zajmującego się filtrowaniem nazw plików. Zachowanie tego zdarzenia i dodatkowe zapewnienie właściwości `Mask` zdecydowanie zwiększy dostępną funkcjonalność.

Zadeklarujemy nowe zdarzenie.

```
typedef void __fastcall (__closure *TAddItemEvent)(TObject *Sender, AnsiString Item,
bool &CanAdd);
```

Zauważmy, że zdarzenie zapewnia trzy parametry. `Sender` to lista, `Item` to `AnsiString` zawierający nazwę pliku, a `CanAdd` to wartość logiczna wskazująca, czy element można dodać. Zauważmy, że ostatni parametr przekazujemy przez referencję, czyli umożliwiamy użytkownikowi zmianę jego wartości na `false`, co zapobiegnie dodaniu `Item` do listy.

Zanim przyjrzymy się sposobowi pobierania nazw plików i dodawania ich do listy, zajmijmy się kodem z listingu 4.51, który umożliwia wyświetlanie takich samych ikon, co w *Eksploratorze Windows*.

Listing 4.51. *Pobieranie listy ikon systemowych*

```
SHFILEINFO shfi;
DWORD iHnd;
TImageList *Images;
Images = new TImageList(this);
Images->ShareImages = true;
iHnd = SHGetFileInfo("", 0, &shfi, sizeof(shfi), SHGFI_SYSICONINDEX |
➥SHGFI_SHELLICONSIZE | SHGFI_SMALLICON);
if(iHnd != 0)
  Images->Handle = iHnd;
```

Zauważmy, że w listingu 4.51 ustawiamy właściwość `ShareImages` z `FImages` na `true`. Jest to bardzo ważne. Informujemy w ten sposób listę obrazów, by nie usuwała uchwytu w trakcie usuwania komponentu. Uchwyt systemowej listy obrazów należy do systemu i jeśli zostałby zniszczony w momencie usuwania komponentu, system Windows nie potrafiłby wyświetlać ikon w menu i skrótach klawiszowych. Aby powrócić do poprzedniego stanu, musielibyśmy zrestartować system.

W tym miejscu możemy przesłonić metodę `ReadFileNames()` z `TFileListBox`, aby pobierać nazwy plików w sposób nieco inny od domyślnego. Nasza wersja wykorzysta powłokę do pobrania nazw plików, używając interfejsów COM. Ponieważ przechodzenie przez listę `itemid` jest dosyć trudne i omówienie tego zagadnienia wykracza poza ramy tego rozdziału, nie będziemy zagłębiać się w szczegóły. Tworzymy nową metodę, `AddItem()`, przedstawioną na listingu 4.52. Pobierze ona nazwę pliku i jego ikonę z systemowej listy obrazów, a następnie zgłosi opisane wcześniej zdarzenie `OnAddItem`.

`itemid` to skrótowa nazwa identyfikatora elementu bądź listy idetyfikatorów. Więcej informacji na ten temat znajduje się w pliku pomocy Win32 pod tematem „Item Identifiers and Identifier Lists”.

Listing 4.52. *Metoda AddItem()*

```
int __fastcall TSHFileListBox::AddItem(LPITEMIDLIST pidl)
{
  SHFILEINFO shfi;
  int Index;
  SHGetFileInfo((char*)pidl, 0, &shfi, sizeof(shfi), SHGFI_PIDL | SHGFI_SYSICONINDEX |
  ➥SHGFI_SMALLICON | SHGFI_DISPLAYNAME | SHGFI_USEFILEATTRIBUTES);

  // zgłoszenie zdarzenia OnAddItem, co pozowoli użytkownikowi na określenie,
  // czy dodawać nazwę plików, czy też jej nie umieszczać na liście
  bool FCanAdd = true;
  if(FOnAddItem)
    FOnAddItem(this, AnsiString(shfi.szDisplayName), FCanAdd);

  if(FCanAdd)
  {
    TShellFileListItem *ShellInfo = new TShellFileListItem(pidl, shfi.iIcon);
    Index = Items->AddObject(AnsiString(shfi.szDisplayName),(TObject*)ShellInfo);
    // zwracamy długość nazwy pliku
    return Canvas->TextWidth(Items->Strings[Index]);
  }
  // zwracamy zero, ponieważ nie dodano pliku do listy
  return 0;
}
```

Metoda `AddItem()` jako jedyny parametr przyjmuje `itemid` i zwraca wartość całkowitą. W listingu 4.52 korzystamy z funkcji API `SHGetFileInfo()`, aby pobrać nazwę pliku i indeks ikony. Po otrzymaniu nazwy pliku tworzymy zmienną logiczną `CanAdd` określającą możliwość dodania elementu do listy, a następnie zgłaszamy zdarzenie `OnAddItem`. Po jego wykonaniu sprawdzamy zawartość `CanAdd`. Jeśli wynosi `true`, dodajemy nowy element do listy. Po dodaniu korzystamy z metody `TextWidth()` klasy `TCanvas`, aby pobrać szerokość napisu w pikselach. Wartość tę zwracamy, gdy dodano element. W przeciwnym razie zwracamy 0. Wkrótce przekonamy się, czemu warto stosować takie podejście.

Do tej pory nie zajęliśmy się jeszcze klasą `TShellFileListItem`. Ponieważ musimy zająć się rzeczywistym rysowaniem ikon i tekstu na liście, musimy w jakiś sposób przechowywać wszystkie indeksy ikon elementów. Dla każdego elementu dodawanego do listy tworzymy obiekt `TShellFileListItem` i przypisujemy go do właściwości `Object` z właściwości `Items` listy. Tym sposobem łatwo ją pobierzemy, gdy zajdzie potrzeba rysowania ikony elementu. `TShellFileListItem` przechowuje także kopię `itemid` elementu. Umożliwia to kreowanie dalszych rozszerzeń — na przykład możemy utworzyć potomka `TSHFileListBox` i przysłonić metodę `MouseUp()` w celu wyświetlania menu podręcznego dla pliku.

Jeśli korzystamy z właściwości `Object`, musimy pamiętać o tym, że pamięć wykorzystywana przez instancję `TShellFileListItem` musi zostać zwolniona w trakcie usuwania listy. Wykonamy to, przysłaniając metodę `DeleteString()` (patrz listing 4.55).

Wspomnieliśmy o tym, że wartość zwracana przez metodę `AddItem()` to długość w pikselach elementu, który właśnie został dodany do listy. W ten sposób potrafimy określić najdłuższy element i wyświetlić pionowy pasek przewijania, jeśli nazwa jest dłuższa od szerokości listy. Przyjrzyjmy się następującemu kodowi:

```
while(Fetched > 0)
{
  // dodajemy element do listy
  int l = AddItem(rgelt);
  if(l > hExtent)
    hExtent = l;
  ppenumIDList->Next(celt, &rgelt, &Fetched);
}
```

Jest to fragment kodu z metody `ReadFileNames()`. Przechodzi on iteracyjnie przez listę `itemid` folderu i pobiera `itemid` dla każdego pliku. Metoda `AddItem()` zwraca długość elementu. Następnie porównujemy tę długość z największą zanotowaną wcześniej. Jeśli długość nowego elementu jest większa, przypisujemy ją do zmiennej `l`. Cały proces powtarza się tak długo, aż zostaną przetworzone wszystkie pliki. Na końcu pętli `l` przechowuje długość największego elementu. Później wywołujemy metodę `DoHorizontalScrollBar()`, aby sprawiedzić, czy konieczne jest wyświetlenie poziomego paska przewijania.

Metoda `DoHorizontalScrollBar()`, przedstawiona na listingu 4.53, przyjmuje jako parametr wartość całkowitą. Jest to długość w pikselach właśnie dodanego elementu. Wartość zwiększamy o 2 piksele dla lewego marginesu, a jeśli jest włączona właściwość `ShowGlyph` (wartość `true`), dodajemy jeszcze 18 pikseli, by skompensować szerokość obrazu i odstęp między obrazem a tekstem. Na końcu wywołujemy metodę `Perform()`, aby wyświetlić poziomy pasek przewijania, gdy wartość z parametru `WPARAM` będzie większa od szerokości kontrolki.

Listing 4.53. *Dodanie poziomego paska przewijania*

```
void __fastcall TSHFileListBox::DoHorizontalScrollBar(int he)
{
  he += 2;
  if(ShowGlyphs)
    he += 18;
  Perform(LB_SETHORIZONTALEXTENT, he, 0);
}
```

Listingi 4.54 i 4.55 przedstawiają pełny kod źródłowy komponentu `TSHFileListBox`, który można znaleźć w katalogu *Rozdzial4\TSHFileListBox* na dołączonej płycie CD-ROM.

Listing 4.54. *Plik nagłówkowy THFileListBox, plik SHFileListBox.h*

```
//---------------------------------------------------------------------------
#ifndef SHFileListBoxH
#define SHFileListBoxH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <FileCtrl.hpp>
#include <StdCtrls.hpp>
#include "ShlObj.h"
```

```
//---------------------------------------------------------------------------

class TShellFileListItem : public TObject
{
private:
  LPITEMIDLIST Fpidl;
  int FImageIndex;
public:
  __fastcall TShellFileListItem(LPITEMIDLIST lpidl, int Index);
  __fastcall ~TShellFileListItem(void);
  __property LPITEMIDLIST pidl = {read=Fpidl};
  __property int ImageIndex = {read=FImageIndex};
};

typedef void __fastcall (__closure *TAddItemEvent)(TObject *Sender, AnsiString Item,
➥bool &CanAdd);

class PACKAGE TSHFileListBox : public TFileListBox
{
private:
  TImageList *FImages;
  TNotifyEvent FOnDblClick;
  bool FCanLaunch;
  bool FRightBtnSel;
  TAddItemEvent FOnAddItem;
  void __fastcall GetSysImages(void);

protected:
  DYNAMIC void __fastcall DblClick(void);
  void __fastcall ReadFileNames(void);
  DYNAMIC void __fastcall MouseUp(TMouseButton Button, TShiftState Shift, int X, int Y);
  void __fastcall DrawItem(int Index, const TRect &Rect, TOwnerDrawState State);
  void __fastcall DoHorizontalScrollBar(int he);
  DYNAMIC void __fastcall DeleteString(int Index);

public:
  __fastcall TSHFileListBox(TComponent* Owner);
  __fastcall ~TSHFileListBox(void);
  int __fastcall AddItem(LPITEMIDLIST pidl);

__published:
  __property bool CanLaunch = {read=FCanLaunch, write=FCanLaunch, default=true};
  __property bool RightBtnSel = {read=FRightBtnSel, write=FRightBtnSel,
    default=true};
  __property TNotifyEvent OnDblClick = {read=FOnDblClick, write=FOnDblClick};
  __property TAddItemEvent OnAddItem = {read=FOnAddItem, write=FOnAddItem};
};
#endif
```

Listing 4.55. *Plik źródłowy TSHFileListBox, plik SHFileListBox.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "SHFileListBox.h"
#pragma package(smart_init)
```

```
//---------------------------------------------------------------------------
__fastcall TShellFileListItem::TShellFileListItem(LPITEMIDLIST lpidl, int Index)
  : TObject()
{
  // zachowaj kopię pidl pliku
  Fpidl = CopyPIDL(lpidl);
  // zapamiętaj też indeks ikony
  FImageIndex = Index;
}
//---------------------------------------------------------------------------
__fastcall TShellFileListItem::~TShellFileListItem(void)
{
  LPMALLOC lpMalloc=NULL;
  if(SUCCEEDED(SHGetMalloc(&lpMalloc)))
    {
    // zwolnij pamięć związaną z pidl
    lpMalloc->Free(Fpidl);
    lpMalloc->Release();
    }
}

//---------------------------------------------------------------------------
__fastcall TSHFileListBox::TSHFileListBox(TComponent* Owner)
  : TFileListBox(Owner)
{
  ItemHeight = 18;
  ShowGlyphs = true;
  FCanLaunch = true;
  FRightBtnSel = true;
}
//---------------------------------------------------------------------------
__fastcall TSHFileListBox::~TSHFileListBox(void)
{
  // zwolnij obrazy
  if(FImages)
    delete FImages;
  FImages = NULL;
}
//---------------------------------------------------------------------------
void __fastcall TSHFileListBox::DeleteString(int Index)
{
  // ta metoda wywoływana jest w odpowiedzi na komunikat LB_DELETESTRING
  // najpierw usuń TShellFileListItem wskazywany przez właściwość Object tekstu
  TShellFileListItem *ShellItem = reinterpret_cast<TShellFileListItem*>
    (Items->Objects[Index]);
  delete ShellItem;
  ShellItem = NULL;
  // teraz usuń element
  Items->Delete(Index);
}
//---------------------------------------------------------------------------
namespace Shfilelistbox
{
  void __fastcall PACKAGE Register()
  {
     TComponentClass classes[1] = {__classid(TSHFileListBox)};
     RegisterComponents("Samples", classes, 0);
  }
```

```
}
//---------------------------------------------------------------------------
void __fastcall TSHFileListBox::ReadFileNames(void)
{

  LPMALLOC g_pMalloc;
  LPSHELLFOLDER pisf;
  LPSHELLFOLDER sfChild;
  LPITEMIDLIST pidlDirectory;
  LPITEMIDLIST rgelt;
  LPENUMIDLIST ppenumIDList;
  int hExtent;

  try
    {
    try
      {
      if(HandleAllocated())
        {
        GetSysImages();
        // wyłącz aktualizację ekranu
        Items->BeginUpdate();
        // usuń element znajdujący się już na liście
        Items->Clear();
        // pobierz globalny alokator powłoki
        if(SHGetMalloc(&g_pMalloc) != NOERROR)
          {
          return;
          }
        // pobierz interfejs IShellFolder pulpitu
        if(SHGetDesktopFolder(&pisf) != NOERROR)
          {
          return;
          }

        // konwertuj nazwę folderu na WideChar
        WideChar oleStr[MAX_PATH];
        FDirectory.WideChar(oleStr, MAX_PATH);
        unsigned long pchEaten;
        unsigned long pdwAttributes;
        // pobierz pidl aktualnego katalogu
        pisf->ParseDisplayName(Handle, 0, oleStr, &pchEaten,
                &pidlDirectory, &pdwAttributes);

        // pobierz interfejs IShellFolder aktualnego katalogu
        if(pisf->BindToObject(pidlDirectory,NULL,
            IID_IShellFolder, (void**)&sfChild) != NOERROR)
          {
          return;
          }
        // wylicz obiekty znajdujące się w folderze
        sfChild->EnumObjects(Handle, SHCONTF_NONFOLDERS |
          SHCONTF_INCLUDEHIDDEN, &ppenumIDList);

        // przejdź przez listę numerowaną
        ULONG celt = 1;
        ULONG Fetched = 0;
```

```
        ppenumIDList->Next(celt, &rgelt, &Fetched);
        hExtent = 0;
        while(Fetched > 0)
          {
          // dodaj element do listy
          int l = AddItem(rgelt);
          if(l > hExtent)
            hExtent = l;
          ppenumIDList->Next(celt, &rgelt, &Fetched);
          }

        }
      }
    catch(Exception &E)
      {
      throw(E); // ponów zgłoszenia uzyskanych wyjątków
      }
    }
  __finally
    { // upewniamy się, czy wykonujemy to bez strażników
    g_pMalloc->Free(rgelt);
    g_pMalloc->Free(ppenumIDList);
    g_pMalloc->Free(pidlDirectory);
    pisf->Release();
    sfChild->Release();
    g_pMalloc->Release();
    Items->EndUpdate();
    }
  // wyświetl poziomy pasek przewijania, jeśli to konieczne
  DoHorizontalScrollBar(hExtent);
}
// ---------------------------------------------------------------------------
void __fastcall TSHFileListBox::DoHorizontalScrollBar(int he)
{
  // dodaj nieco miejsca na margines
  he += 2;
  // jeśli wyświetlamy ikonę, uwzględnij jeszcze miejsce na nią
  // i odstęp między obrazem a tekstem
  if(ShowGlyphs)
    he += 18;

  Perform(LB_SETHORIZONTALEXTENT, he, 0);
}

// ---------------------------------------------------------------------------
void __fastcall TSHFileListBox::GetSysImages(void)
{
  SHFILEINFO shfi;
  DWORD iHnd;
  if(!FImages)
    {
    FImages = new TImageList(this);
    FImages->ShareImages = true;
    FImages->Height = 16;
    FImages->Width = 16;
    iHnd = SHGetFileInfo("", 0, &shfi, sizeof(shfi), SHGFI_SYSICONINDEX |
      SHGFI_SHELLICONSIZE | SHGFI_SMALLICON);
```

```
    if(iHnd != 0)
      FImages->Handle = iHnd;
    }
}
// ---------------------------------------------------------------------------
int __fastcall TSHFileListBox::AddItem(LPITEMIDLIST pidl)
{
  SHFILEINFO shfi;
  int Index;

  SHGetFileInfo((char*)pidl, 0, &shfi, sizeof(shfi), SHGFI_PIDL | SHGFI_SYSICONINDEX |
    SHGFI_SMALLICON | SHGFI_DISPLAYNAME | SHGFI_USEFILEATTRIBUTES);

  // zgłoś zdarzenie OnAddItem, aby użytkownik zdecydował, czy rzeczywiście chce
  // dodać ten plik do listy
  bool FCanAdd = true;
  if(FOnAddItem)
    FOnAddItem(this, AnsiString(shfi.szDisplayName), FCanAdd);

  if(FCanAdd)
    {
    TShellFileListItem *ShellInfo = new TShellFileListItem(pidl, shfi.iIcon);
    Index = Items->AddObject(AnsiString(shfi.szDisplayName), (TObject*)ShellInfo);
    // zwróć długość nazwy pliku
    return Canvas->TextWidth(Items->Strings[Index]);
    }
  // jeśli nie dodano pliku do listy, zwróć 0
  return 0;
}
// ---------------------------------------------------------------------------
void __fastcall TSHFileListBox::DrawItem(int Index, const TRect &Rect,
  TOwnerDrawState State)
{
  int Offset;

  Canvas->FillRect(Rect);
  Offset = 2;
  if(ShowGlyphs)
    {
    TShellFileListItem *ShellItem = reinterpret_cast<TShellFileListItem*>
      (Items->Objects[Index]);
    // rysuj ikony pliku na liście
    FImages->Draw(Canvas, Rect.Left+2, Rect.Top+2, ShellItem->ImageIndex, true);
    Offset += 18;
    }
  int Texty = Canvas->TextHeight(Items->Strings[Index]);
  Texty = ((ItemHeight - Texty) / 2) + 1;
  // rysuj tekst
  Canvas->TextOut(Rect.Left + Offset, Rect.Top + Texty, Items->Strings[Index]);
}
//---------------------------------------------------------------------------
void __fastcall TSHFileListBox::DblClick(void)
{
  if(FCanLaunch)
    {
    int ii=0;
    // przejdź przez listę, by dowiedzieć się, który element zaznaczono
```

```
    for(ii=0; ii < Items->Count; ii++)
      {
      if(Selected[ii])
        {
        AnsiString str = Items->Strings[ii];
        ShellExecute(Handle, "open", str.c_str(), 0, 0, SW_SHOWDEFAULT);
        }
      }
    }
  // zgłoś zdarzenie OnDblClick
  if(FOnDblClick)
    FOnDblClick(this);
}
//---------------------------------------------------------------------------
void __fastcall TSHFileListBox::MouseUp(TMouseButton Button, TShiftState Shift,
  int X, int Y)
{
  if(!FRightBtnSel)
    return;

  TPoint ItemPos = Point(X,Y);
  // czy mysz znajduje się nad elementem?
  int Index = ItemAtPos(ItemPos, true);
  // jeśli nie, powrót
  if(Index == -1)
    return;
  // w przeciwnym razie zaznacz element
  Perform(LB_SETCURSEL, (WPARAM)Index, 0);
}
//---------------------------------------------------------------------------
// ValidCtrCheck zapewnia, że tworzone komponenty nie posiadają żadnych
// funkcji czysto wirtualnych
//
static inline void ValidCtrCheck(TSHFileListBox *)
{
  new TSHFileListBox(NULL);
}
```

Tworzenie własnych komponentów związanych z danymi

Podobnie jak w przypadku pozostałych komponentów na początku stajemy przed wyborem, po której klasie dziedziczyć, kreując komponent związany z danymi. W tym podrozdziale zamierzamy rozszerzyć komponent `TEditMask`, aby odczytywał on dane ze źródła danych i wyświetlał je z określoną maską. Tego rodzaju kontrolki nazywa się kontrolkami przeglądania danych. Następnie rozszerzymy tę kontrolkę, by uczynić z niej komponent związany z danymi, co umożliwi dwukierunkową zmianę wartości zarówno w polu tekstowym, jak i bazie danych.

Wykonanie kontrolki tylko do odczytu

Kontrolka, którą zamierzamy tworzyć, posiada już właściwość `ReadOnly`, więc nie musimy jej kreować. Jeśli komponent nie posiada takiej właściwości, tworzymy ją jak każdą inną właściwość.

Jeżeli komponent nie posiada właściwości `ReadOnly`, sposób jej utworzenia przedstawia listing 4.56 (zauważmy, że przedstawiony kod nie jest wymagany dla tego komponentu).

Listing 4.56. *Tworzenie właściwości ReadOnly*

```
class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
  bool FReadOnly;
protected:
public:
  __fastcall TDBMaskEdit(TComponent* Owner);
__published:
  __property ReadOnly = {read = FReadOnly, write = FReadOnly, default = true};
};
```

W konstruktorze ustawiamy domyślną wartość właściwości.

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent* Owner)
: TMaskEdit(Owner)
{
  FReadOnly = true;
}
```

Teraz musimy zatroszczyć się o to, by komponent zachowywał się jak kontrolka tylko do odczytu. Oznacza to konieczność przesłonięcia metody odpowiedzialnej za dostęp użytkownika do kontrolki. Jeśli tworzymy siatkę związaną z danymi, wartość właściwości `ReadOnly` sprawdzalibyśmy w metodzie `SelectCell()`, a następnie odpowiednio reagowali. Jeśli wartość `ReadOnly` wynosi `false`, wywołujemy metodę nadrzędną, a w przeciwnym razie wracamy.

Jeśli komponent `TMaskEdit` posiadałby metodę `SelectEdit()`, kod mógłby mieć następującą postać:

```
bool __fastcall TDBMaskEdit::SelectEdit(void)
{
  if(FReadOnly)
    return(false);
  else
    return(TMaskEdit::SelectEdit());
}
```

W naszym przypadku nie musimy się martwić o właściwość `ReadOnly`, ponieważ `TMaskEdit` już ją posiada.

Nawiązywanie połączenia

Aby nasz komponent był związany z danymi, musimy zapewnić łącze danych wymagane do komunikacji z daną należącą do bazy danych. Łącze to nazwiemy `TFieldDataLink`.

Kontrolka związana z danymi ma własną klasę łącza danych. To na niej spoczywa odpowiedzialność związana z utworzeniem, inicjalizacją oraz usunięciem łącza danych.

Ustanowienie połączenia wymaga wykonania trzech kroków.

1. Deklaracji klasy łącza danych jako pole komponentu.
2. Deklaracji odpowiednich metod ustawiania i pobierania.
3. Inicjalizacji łącza danych.

Deklaracja łącza danych

Łącze danych to klasa typu `TFieldDataLink` wymagająca dołączenia pliku nagłówkowego *DBCTRLS.HPP*.

```
#include <DBCtrls.hpp>

class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
  TFieldDataLink *FDataLink;
  ...
};
```

Komponent związany z danymi wymaga jeszcze właściwości `DataField` i `DataSource` (podobnie jak wszystkie inne komponenty tego rodzaju). Właściwości te wykorzystują metody przejścia, by uzyskać dostęp do właściwości klasy łącza danych. Umożliwia to komponentowi i łączu danych współdzielenie tego samego pola i źródła danych.

Deklaracja metod odczytu i zapisu

Dostęp, jaki zapewniamy kontrolce, zależy od deklaracji samych właściwości. Zamierzamy nadać komponentowi pełny dostęp. Posiadamy właściwość `ReadOnly`, która automatycznie zadba o to, by użytkownik nie mógł modyfikować kontrolki. Zauważmy, że nie blokuje to programiście możliwości napisania kodu, który zapisuje bezpośrednio do dołączonego pola bazy danych przez kontrolkę. Aby wymusić dostęp tylko do odczytu, nie korzystamy ze słowa kluczowego `write`.

Kod z listingów 4.57 i 4.58 przedstawia deklaracje właściwości i odpowiadające im implementacje metod odczytu i zapisu.

Listing 4.57. *Deklaracja klasy TDBMaskEdit z pliku nagłówkowego*

```
class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
  ...
  AnsiString __fastcall GetDataField(void);
  TDataSource* __fastcall GetDataSource(void);
  void __fastcall SetDataField(AnsiString pDataField);
  void __fastcall SetDataSource(TDataSource *pDataSource);
  ...
__published:
  __property AnsiString DataField = {read = GetDataField, write = SetDataField, nodefault};
  __property TDataSource *DataSource = {read = GetDataSource, write = SetDataSource,
  ➥nodefault};
};
```

Listing 4.58. *Metody TDBMaskEdit z pliku źródłowego*

```
AnsiString __fastcall TDBMaskEdit::GetDataField(void)
{
  return(FDataLink->FieldName);
}

TDataSource * __fastcall TDBMaskEdit::GetDataSource(void)
{
  return(FDataLink->DataSource);
}

void __fastcall TDBMaskEdit::SetDataField(AnsiString pDataField)
{
  FDataLink->FieldName = pDataField;
}

void __fastcall TDBMaskEdit::SetDataSource(TDataSource *pDataSource)
{
  if(pDataSource != NULL)
    pDataSource->FreeNotification(this);
  FDataLink->DataSource = pDataSource;
}
```

Jedyny kod wymagający dodatkowego wyjaśnienia to metoda `FreeNotification()` dla `pDataSource`. C++Builder przechowuje wewnętrzną listę obiektów, więc istnieje możliwość powiadomienia innych obiektów o tym, że aktualny obiekt jest usuwany. Metoda `FreeNotification()` jest wywoływana automatycznie dla komponentów tego samego formularza, ale w tym przypadku istnieje możliwość, iż komponent innego formularza (na przykład modułu danych) posiada do niej referencję. W związku z tym musimy wywołać `FreeNotification()`, aby obiekt został dodany do wewnętrznej listy wszystkich formularzy.

Inicjalizacja łącza danych

Może się wydawać, że wszystko zostało już zrobione, ale jeśli spróbujemy skompilować komponent i dodać go do formularza, uzyskamy błąd ochrony zgłaszany przez inspektora obiektów dla właściwości `DataField` i `DataSource`. Spowodowane jest to brakiem instancji wewnętrznego obiektu `FieldDataLink`.

Do sekcji publicznej pliku nagłówkowego klasy dodajemy następującą deklarację:

```
__fastcall ~TDBMaskEdit(void);
```

Do konstruktora i destruktora komponentu dodajemy następujący kod:

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent* Owner) : TMaskEdit(Owner)
{
  FDataLink = new TFieldDataLink();
  FDataLink->Control = this;
}

__fastcall TDBMaskEdit::~TDBMaskEdit(void)
{
  if(FDataLink)
```

```
  {
    FDataLink->Control = 0;
    FDataLink->OnUpdateData = 0;
    delete FDataLink;
  }
}
```

Właściwość `Control` z `FDataLink` jest typu `TComponent`. Musi być ona ustawiona na komponent, który korzysta z obiektu `TFieldDataLink`, aby móc zarządzać łączem do obiektu `TField`. Właściwość `Control` ustawiamy na `this`, aby zaznaczyć, że łącze dotyczy komponentu.

Dostęp do `TObject` uzyskujemy, dodając właściwość tylko do odczytu. Podany wiersz umieszczamy w sekcji publicznej definicji klasy.

```
__property TField *Field = {read = GetField};
```

Deklarację `GetField` dodajemy do sekcji prywatnej.

```
TField * __fastcall GetField(void);
```

Podany kod umieszczamy w pliku kodu źródłowego.

```
TField * __fastcall TDBMaskEdit::GetField(void)
{
  return(FDataLink->Field);
}
```

Korzystanie ze zdarzenia OnDataChange

Wykonaliśmy komponent łączący się ze źródłem danych, ale na razie nie reaguje on na zmiany danych. Teraz napiszemy kod, który umożliwi reakcję komponentu na zmiany pól, na przykład przejście do nowego rekordu.

Klasy łączy danych posiadają zdarzenie `OnDataChange` zgłaszane w momencie, w którym źródło danych wykryje zmianę w danych. Aby komponent mógł odpowiedzieć na tę zmianę, dodajemy metodę i przypisujemy ją do zdarzenia `OnDataChange`.

`TDataLink` to klasa pomocnicza wykorzystywana w obiektach związanych z danymi. Listę jej metod, zdarzeń i właściwości zawiera pomoc środowiska programistycznego C++Builder.

Zdarzenie `OnDataChange` jest typu `TNotifyEvent`, więc przy tworzeniu własnej metody musimy korzystać z tego samego prototypu. Poniższy fragment kodu dodajemy do sekcji prywatnej nagłówka komponentu.

```
class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
  // ...
  void __fastcall DataChange(TObject *Sender);
}
```

W konstruktorze przypisujemy metodę `DataChange()` do zdarzenia `OnDataChange`. Przypisanie to usuwamy w destruktorze.

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent* Owner) : TMaskEdit(Owner)
{
  FDataLink = new TFieldDataLink();
  FDataLink->Control = this;
  FDataLink->OnDataChange = DataChange;
}

__fastcall TDBMaskEdit::~TDBMaskEdit(void)
{
  if(FDataLink)
  {
    FDataLink->Control = 0;
    FDataLink->OnUpdateData = 0;
    FDataLink->OnDataChange = 0;
    delete FDataLink;
  }
}
```

Na końcu w następujący sposób definiujemy metodę `DataChange()`.

```
void __fastcall TDBMaskEdit::DataChange(TObject *Sender)
{
  if(!FDataLink->Field)
  {
  if(ComponentState.Contains(csDesigning))
    Text = Name;
  else
    Text = "";
  }
  else
    Text = FDataLink->Field->AsString;
}
```

Metoda `DataChange()` sprawdza najpierw, czy łącze danych wskazuje na źródło (i pole) danych. Jeśli nie jest to poprawny wskaźnik, ustawia właściwość `Text` (pole komponentu rodzica) na pusty tekst (w trakcie wykonywania programu) lub na nazwę kontrolki (w trybie projektowania). Jeśli jest to poprawne pole, ustawia właściwość `Text` na zawartość pola, używając właściwości `AsString` obiektu `TField`.

Tym sposobem uzyskaliśmy kontrolkę przeglądania danych. Jest tak nazywana, ponieważ umożliwia tylko wyświetlanie zmian źródła danych. Zamieńmy teraz komponent na kontrolkę edycji danych.

Przeróbka na kontrolkę edycji danych

Zamiana kontrolki przeglądania danych na kontrolkę edycji danych wymaga dodania kodu reagującego na zdarzenia myszy i klawiatury. Umożliwi to odzwierciedlenie zmian dokonanych w kontrolce w polu dołączonej bazy danych.

Właściwość ReadOnly

Gdy użytkownik umieszcza kontrolkę edycji danych w projekcie, oczekuje, że kontrolka taka *nie będzie* tylko do odczytu. Domyślna wartość właściwości `ReadOnly` klasy `TMaskEdit` (klasa rodzicielska) to `false`, więc nie musimy niczego modyfikować. Jeśli

kreujemy komponent zawierający własną właściwość `ReadOnly`, upewnijmy się, że jest ona inicjalizowana na `false`.

Zdarzenia myszy i klawiatury

Jeśli zajrzymy do pliku *controls.hpp*, dowiemy się, że `TMaskEdit` zawiera dwie metody chronione, `KeyDown()` i `MouseDown()`. Metody te są wywoływane w odpowiedzi na komunikaty okien (`WM_KEYDOWN`, `WM_LBUTTONDOWN`, `WM_MBUTTONDOWN` i `WM_RBUTTONDOWN`), a same mogą wywoływać odpowiednie zdarzenia, jeśli użytkownik takowe zdefiniuje.

Aby przesłonić te metody, dodajemy własne metody `KeyDown()` i `MouseDown()` do klasy `TDBMaskEdit`. Wykorzystujemy deklaracje klas z pliku *controls.hpp*.

```
virtual void __fastcall MouseDown(TMouseButton, TShiftState Shift, int X, int Y);
virtual void __fastcall KeyDown(unsigned short &Key, TShiftState Shift);
```

Opis tych klas znajdziemy także w pliku pomocy środowiska C++Builder.

Dodajemy kod źródłowy przedstawiony na listingu 4.59.

Listing 4.59. *Metody MouseDown() i KeyDown()*

```
void __fastcall TDBMaskEdit::MouseDown(TMouseButton Button, TShiftState Shift, int X,
➥int Y)
{
  if(!ReadOnly && FDataLink->Edit())
    TMaskEdit::MouseDown(Button, Shift, X, Y);
  else
  {
    if(OnMouseDown)
      OnMouseDown(this, Button, Shift, X, Y);
  }
}

void __fastcall TDBMaskEdit::KeyDown(unsigned short &Key, TShiftState Shift)
{
  Set<unsigned short, VK_PRIOR, VK_DOWN> Keys;
  Keys = Keys << VK_PRIOR << VK_NEXT << VK_END << VK_HOME << VK_LEFT << VK_UP <<
  ➥VK_RIGHT << VK_DOWN;
  if(!ReadOnly && (Keys.Contains(Key)) && FDataLink->Edit())
    TMaskEdit::KeyDown(Key, Shift);
  else
  {
    if(OnKeyDown)
      OnKeyDown(this, Key, Shift);
  }
}
```

W obydwu przypadkach sprawdzamy, czy komponent nie jest tylko do odczytu oraz czy `FieldDataLink` jest w trybie edycji. Metoda `KeyDown()` sprawdza jeszcze wszystkie klawisze kursorów (zdefiniowane w pliku *winuser.h*). Jeśli wszystkie testy przejdą pomyślnie, pole można edytować, więc wywoływana jest metoda klasy rodzica. Wywołuje ona automatycznie kod użytkownika przypisany do zdarzenia, jeśli istnieje. Jeśli pola nie można edytować, wywołujemy tylko kod użytkownika dotyczący zdarzenia.

Radzenie sobie z aktualizacją danych

Jeśli użytkownik zmodyfikuje zawartość kontrolki związanej z danymi, zmiana musi zostać odzwierciedlona w polu. Podobnie, jeśli zostanie zmodyfikowana wartość pola, kontrolka powinna uaktualnić swą wartość.

Komponent `TDBMaskEdit` zawiera metodę `DataChange()` wywoływaną przez zdarzenie `OnDataChange` z `TFieldDataLink`. Metoda ta odpowiada za aktualizację komponentu, gdy zmieniona została zawartość pola. Oznacza to, że drugi przypadek aktualizacji danych jest już wykonany.

Zajmiemy się teraz aktualizacją wartości pola, gdy użytkownik zmodyfikował zawartość komponentu. Klasa `TFieldDataLink` zawiera zdarzenie `OnUpdateData`, w którym to kontrolka może umieścić zmiany rekordu bazy danych. Możemy teraz wykreować metodę `UpdateData()` dla `TDBMaskEdit` i przypisać ją do zdarzenia `OnUpdateData` klasy `TFieldDataLink`.

Dodajemy deklarację metody `UpdateData()` do komponentu `TDBMaskEdit`, używając przedstawionego dalej wiersza kodu.

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

W konstruktorze przypisujemy metodę do zdarzenia `OnUpdateData` klasy `TFieldDataLink`.

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent* Owner) : TMaskEdit(Owner)
{
  FDataLink = new TFieldDataLink();
  FDataLink->Control = this;
  FDataLink->OnUpdateData = UpdateData;
  FDataLink->OnDataChange = DataChange;
}
```

Ustawiamy wartość pola na aktualną zawartość komponentu `TDBMaskEdit`.

```
void __fastcall TDBMaskEdit::UpdateData(TObject *Sender)
{
  if(FDataLink->CanModify)
    FDataLink->Field->AsString = Text;
}
```

Komponent `TDBMaskEdit` to potomek komponentu `TMaskEdit`, który wywodzi się od klasy `TCustomEdit`. Klasa ta zawiera metodę chronioną `Change()` wywoływaną przez zdarzenia systemu. Metoda ta wywołuje zdarzenie `OnChange`.

Zamierzamy przysłonić metodę `Change()`, aby aktualizowała bazę danych przed wywołaniem metody nadrzędnej. W sekcji chronionej klasy `TDBMaskEdit` dodajemy poniższą metodę.

```
DYNAMIC void __fastcall Change(void);
```

Kod źródłowy metody `Change()` przedstawia listing 4.60.

Listing 4.60. *Metoda Change()*

```
void __fastcall TDBMaskEdit::Change(void)
{
if(FDataLink)
  {
  // sprawdzamy, czy źródło danych znajduje się w trybie edycji
  // jeśli nie, musimy zapamiętać aktualną wartość, ponieważ
  // przełączenie źródła danych w tryb edycji spowoduje zmianę
  // aktualnej wartości na tę występującą w tabeli
  AnsiString ChangedValue = Text;

  // pobieramy także kursor położenia
  int CursorPosition = SelStart;

  if(FDataLink->CanModify && FDataLink->Edit())  // musi się znajdować w trybie edycji
    {
    Text = ChangedValue;   // na wypadek, gdyby jednak nie był to tryb edycji
    SelStart = CursorPosition;
    FDataLink->Modified(); // wysłanie zmiany (źródło danych nie wraca do trybu edycji)
    }
  }

  TMaskEdit::Change();
}
```

Taka modyfikacja metody powoduje poinformowanie klasy `TFieldDataLink`, że nastąpiły zmiany. Na końcu metody wywołujemy metodę nadrzędną `Change()`.

Ostatni krok polega na zapewnieniu uaktualnienia, gdy kontrolka traci aktywność. Klasa `TWinControl` reaguje na komunikat `CM_EXIT` generowaniem zdarzenia `OnExit`.

Możemy jeszcze zareagować na ten komunikat, aktualizując rekord z dołączonej bazy danych. Aby to wykonać, tworzymy mapę komunikatów dla klasy `TDBMaskEdit`. Podany kod dodajemy do sekcji prywatnej.

```
void __fastcall CMExit(TWMNoParams Message);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
END_MESSAGE_MAP(TMaskEdit)
```

W mapie komunikatów określamy, że po uzyskaniu komunikatu `CM_EXIT` ma zostać wywołana metoda `CMExit()`, której nie interesują parametry komunikatu.

```
void __fastcall TDBMaskEdit::CMExit(void)
{
  try
  {
    ValidateEdit();
    if(FDataLink && FDataLink->CanModify)
      FDataLink->UpdateRecord();
  }
  catch(...)
  {
    SetFocus();
```

```
    throw;
  }
}
```

Najpierw sprawdzamy zgodność zawartości pola z maską. Jeśli źródło danych można modyfikować, aktualizujemy rekord w bazie danych. Jeśli w tym czasie został zgłoszony wyjątek, ustawiamy kursor na kontrolce, aby wskazać pojawienie się problemu, a sam wyjątek ponawiamy, aby zajęła się nim aplikacja.

Dodajemy ostatni komunikat

C++Builder zawiera komponent o nazwie `TDBCtrlGrid`. Kontrolka ta wyświetla rekord ze źródła danych w dowolnym układzie graficznym. Gdy komponent aktualizuje swoje źródło danych, wysyła komunikat `CM_GETDATALINK`. Jeśli poszukamy takiego komunikatu w plikach źródłowych C++Buildera, znajdziemy go w mapach komunikatów wszystkich komponentów związanych z bazami danych. Jeśli zajrzymy do plików *.pas*, ujrzymy następującą procedurę obsługi komunikatu.

```
procedure TDBEdit.CMGetDataLink(var Message: TMessage);
begin
  Message.Result := Integer(FDataLink);
end;
```

Dodamy obsługę tego komunikatu do naszego komponentu, rozszerzając mapę komunikatów, deklarując metodę i implementując procedurę obsługi komunikatu.

W sekcji prywatnej dodajemy podany wiersz.

```
void __fastcall CMGetDataLink(TMessage Message);
```

Modyfikujemy mapę komunikatów, aby przyjęła ona następującą postać.

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
  MESSAGE_HANDLER(CM_GETDATALINK, TMessage, CMGetDataLink)
END_MESSAGE_MAP(TMaskEdit)
```

Na końcu implementujemy metodę w pliku źródłowym.

```
void __fastcall TDBMaskEdit::CMGetDataLink(TMessage Message)
{
  Message.Result = (int)FDataLink;
}
```

To wszystko. Wykonaliśmy komponent związany z danymi, który zachowuje się tak samo, jak wszystkie pozostałe komponenty tego typu.

Rejestracja komponentów

Rejestracja komponentów to prosta, choć wieloetapowa procedura. Pierwszy etap jest bardzo łatwy. Musimy się upewnić, że dowolny komponent, który chcemy zainstalować na palecie komponentów, nie zawiera żadnych czysto wirtualnych funkcji (lub niezaimplementowanych funkcji `DYNAMIC`) — innymi słowy, funkcji o następującej postaci:

```
virtual ZwracanyTyp __fastcall NazwaFunkcji(ListaParametrow) = 0;
```

Słowo kluczowe `__fastcall` nie jest wymagane do zadeklarowania czysto wirtualnych funkcji (metod abstrakcyjnych), ale musi istnieć w metodach komponentów, więc zostało tu uwzględnione.

Sprawdzenia możemy dokonać ręcznie, przeglądając definicje klasy komponentu, lub automatycznie, wywołując funkcję `ValidCtrCheck()` i przekazując jako jej argument wskaźnik na komponent. Funkcja `ValidCtrCheck()` znajduje się w dowolnym miejscu pliku implementacyjnego. Dla komponentu o nazwie `TCustomComponent` przyjmie postać:

```
static inline void ValidCtrCheck(TCustomComponent *)
{
  new TCustomComponent(NULL);
}
```

Jedynym zadaniem tej funkcji jest utworzenie obiektu `TCustomComponent`. Ponieważ nie jest możliwe utworzenie obiektu klasy posiadającej choćby jedną czysto wirtualną metodę, kompilator zgłosi następujące błędy kompilacji:

```
E2352 Cannot create instance of abstract class 'TCustomComponent'
E2353 Class 'TCustomComponent' is abstract because of 'funkcja'
```

Drugi błąd wskaże funkcję, która nadal jest abstrakcyjna. Obydwa błędy będą dotyczyły wiersza

```
new TCustomComponent(NULL);
```

Korzystanie z tej metody rzadko kiedy jest potrzebne, ponieważ w zasadzie nie istnieje możliwość przypadkowego utworzenia metod abstrakcyjnych. Jeśli jednak wykorzystujemy środowisko projektowe do tworzenia komponentów, funkcja `ValidCtrCheck()` dodawana jest automatycznie do pliku z implementacją. Warto ją wtedy pozostawić, gdyż może się kiedyś przydać.

Po stwierdzeniu, że komponent nie jest abstrakcyjną klasą bazową i można tworzyć jego obiekty, rozpoczynamy pisanie właściwego kodu rejestrującego. W tym celu piszemy funkcję `Register()`. Musi się ona znaleźć w przestrzeni nazw o takiej samej nazwie jak plik, w którym się znajduje. Należy też spełnić dodatkowy warunek: pierwsza litera nazwy przestrzeni nazw musi być pisana wielką literą, a pozostałe — małą. Z tego powodu funkcja rejestracji przyjmie następującą postać:

```
namespace Nazwaaktualnegopliku
{
  void __fastcall PACKAGE Register()
  {
    // tutaj znajduje się kod rejestrujący
  }
}
```

Należy pamiętać o dodaniu makra `PACKAGE` przed funkcją `Register()`. Skoro funkcja jest już na miejscu, musimy jeszcze zarejestrować komponent (lub komponenty). W tym celu korzystamy z funkcji `RegisterComponents()`. Jest ona zadeklarowana w pliku *$(BCB)\Include\Vcl\Classes.hpp* w następującej postaci.

```
extern PACKAGE void __fastcall RegisterComponents(const AnsiString Page,
                                TMetaClass* const * ComponentClasses,
                                const int ComponentClasses_Size);
```

`RegisterComponent()` oczekuje przekazania dwóch elementów, obiektu `AnsiString` reprezentującego nazwę zakładki palety, na której znajdzie się komponent, a także wskaźnika na otwartą tablicę wskaźników `TMetaClass` komponentów do zainstalowania. Jeśli tekst przekazywany w `AnsiString` nie odpowiada żadnej z zakładek palety, tworzona jest nowa zakładka o nazwie przekazanej w obiekcie. Wartość tego argumentu może być pobierana z zasobów tekstowych, co umożliwia stosowanie różnych tekstów dla różnych języków.

Otwarta tablica `TMetaClass*` wymaga więcej uwagi. Można ją utworzyć na dwa sposoby — wykorzystując makro `OPENARRAY` lub kreując ją ręcznie. Przyjrzyjmy się przykładowi obrazującemu obydwa podejścia.

Używając `OPENARRAY`, możemy napisać funkcję `RegisterComponents()` w następujący sposób.

```
RegisterComponents("MyCustomComponents", OPENARRAY( TMetaClass*,
                                          ( __classid(TCustomComponent1),
                                            __classid(TCustomComponent2),
                                            __classid(TCustomComponent3) ) ) );
```

Moglibyśmy skorzystać z `TComponentClass` zamiast `TMetaClass*`, ponieważ jest to definicja typu dla `TMetaClass*` zadeklarowana w pliku *$(BCB)\Include\Vcl\Classes.hpp* w następujący sposób.

```
typedef TMetaClass* TComponentClass;
```

Z powodu ograniczeń makra `OPENARRAY` jesteśmy ograniczeni do 19 argumentów (komponentów) w jednym wywołaniu `RegisterComponents()`. W większości przypadków nie stanowi to żadnego problemu.

Drugie podejście to ręczna deklaracja i inicjalizacja tablicy `TMetaClass*` (lub `TComponentClass`).

```
TMetaClass Components[3] = { __classid(TCustomComponent1),
                             __classid(TCustomComponent2),
                             __classid(TCustomComponent3) };
```

Przekazujemy tę tablicę do funkcji `RegisterComponents()`, ale poza tym musimy jeszcze przekazać dodatkowy argument zawierający indeks ostatniego poprawnego elementu tablicy (w tym przypadku 2).

Wywołanie funkcji jest bardzo proste, ale istnieje większa możliwość przekazania błędnej wartości ostatniego parametru.

Pełna funkcja `Register()` ma więc postać przedstawioną dalej.

```
namespace Nazwaaktualnegopliku
{
  void __fastcall PACKAGE Register()
  {
```

```
    RegisterComponents("MyCustomComponents", OPENARRAY( TMetaClass*,
                                          ( __classid(TCustomComponent1),
                                            __classid(TCustomComponent2),
                                            __classid(TCustomComponent3) ) ) );
  }
}
```

Pamiętajmy, że w funkcji `Register()` może się znajdować dowolna ilość funkcji `RegisterComponents()`. Umieszczamy też w niej inne rejestracje, na przykład edytorów właściwości lub komponentów. Tym tematem zajmiemy się w następnym rozdziale. Możliwe jest umieszczenie rejestracji komponentu w pliku implementacyjnym komponentu, ale na ogół umieszcza się ją w osobnym pliku.

Mechanizm strumieniowania

C++Builder jest nazywany środowiskiem do szybkiego tworzenia aplikacji (RAD). Dzieje się tak częściowo dlatego, że programista wiele czynności wykonuje w środowisku graficznym, a sam język programowania jest zorientowany obiektowo. Poza tym C++Builder wykorzystuje mechanizm strumieniowania (odczyt i zapis) do przechowywania ustawień właściwości.

W trakcie projektowania programista ustawia wiele właściwości komponentu. Środowisko programistyczne przechowuje te właściwości (dokładny opis działania mechanizmu znajduje się w dalszej części rozdziału) jako część formularza, do którego przynależy komponent. Formularze zapisywane są jako części projektu w pliku o rozszerzeniu *.dmf*. Wartości przechowywane w pliku formularza są wczytywane także w czasie uruchamiania aplikacji. Można powiedzieć, że właściwości są nieulotne.

Sekcja `__published` deklaracji klasy zawiera właściwości, które mają być nieulotne. Tego rodzaju właściwości są zapisywane w pliku formularza w czasie zapisu projektu.

Gdy tworzymy komponent, nadajemy mu zbiór domyślnych wartości dla publikowanych właściwości. Te wartości domyślne są przypisywane w konstruktorze komponentu. Użytkownik korzystający z komponentu modyfikuje właściwości w inspektorze obiektów. W rzeczywistości wszystkie właściwości wyświetlane w oknie *Object Inspector* są właściwościami typu `published`. Właściwości można także deklarować w sekcjach `public` klas komponentów, ale wtedy są one dostępne tylko w trakcie wykonywania programu.

Jednak nie wszystkie publikowane właściwości muszą być przechowywane. Wyobraźmy sobie dwie właściwości: pierwsza ma wartość domyślną równą 10, a druga zawiera ustawienie wyłączające jej przechowywanie.

```
__property int SomeProperty1 = {read=FProp1, write=FProp1, default=10};
__property AnsiString SomeProperty2 = {read=FProp2, stored=false};
```

Deklaracja `SomeProperty1` nie ustawia wartości na 10. Tradycyjnie jest to wykonywane w konstruktorze. Słowo kluczowe `default` informuje środowisko programistyczne, aby przechowywało wartość właściwości tylko wtedy, gdy jest ona różna od 10.

Druga właściwość, `SomeProperty2`, została zadeklarowana z opcją wyłączającą jej przechowywanie w pliku formularza. Przykładem takiej właściwości może być numer wersji aktualnego komponentu. Ponieważ wartość wersji nie ulegnie zmianie, nie ma sensu przechowywać takiej właściwości w formularzu.

W trakcie zapisywania komponentu właściwości, które różnią się od swych wartości domyślnych, są zapamiętywane w pliku formularza. W trakcie ponownego otwierania projektu i tworzenia komponentu właściwościom są przypisywane ich wartości domyślne, a następnie wartości odczytane z pliku formularza.

Konstrukcja komponentu oraz sprawy związane z procesem strumieniowania właściwości w większości przypadków nie interesują programistów.

Dodatkowe wymagania dotyczące strumieniowania

Właściwości komponentu mogą być typu liczbowego, znakowego, łańcuchowego, wyliczeniowego (także wartością logiczną), zbiorem lub bardziej złożonym elementem, jak własna struktura lub klasa. C++Builder ma wbudowaną obsługę typów podstawowych. Obsługa strumieniowania własnych obiektów (klas właściwości) zapewniania jest tylko wtedy, gdy klasa ta dziedziczy po `TPersistent`.

Klasa `TPersistent` zapewnia możliwość przypisywania obiektów do innych obiektów, a także zapewnia odczyt i zapis ich właściwości z i do strumienia. Dodatkowe informacje na temat tej klasy znajdują się w pliku pomocy C++Buildera pod hasłem „TPersistent”.

Pewne typy właściwości, na przykład tablice, wymagają własnych edytorów właściwości. Bez edytora inspektor obiektów nie jest w stanie zapewnić programiście poprawnego interfejsu edycji wartości właściwości. Więcej informacji na temat edytorów właściwości znajduje się w rozdziale 5.

Strumieniowanie niepublikowanych właściwości

Dowiedzieliśmy się, że inspektor obiektów zapewnia programiście interfejs dla opublikowanych właściwości komponentu. Jest to domyślne zachowanie, ale nie musimy się do niego ograniczać. Dowiedzieliśmy się także, że strumieniowane mogą być właściwości klas pochodzących od `TPersistent`. Oznacza to możliwość kreowania strumieniowanych właściwości, które nie pojawiają się w oknie *Object Inspector*. Poza tym jesteśmy w stanie napisać metody strumieniowania dla właściwości, dla których C++Builder nie zna sposobu ich odczytu oraz zapisu.

Zapis niepublikowanych właściwości uzyskujemy, dodając kod informujący C++Buildera o sposobie zapisu i odczytu wartości. Proces ten polega na wykonaniu dwóch kroków.

- Przysłaniamy metody `DefineProperties()`. Wcześniej zdefiniowane metody przekazujemy do obiektu nazywanego *obiektem wypełniającym*.
- Tworzymy metody odczytu i zapisu wartości właściwości.

Wielokrotnie wspominaliśmy, że publikowane właściwości są automatycznie wysyłane do pliku formularza. Zajmują się tym metody odczytu i zapisu zdefiniowane dla konkretnych typów właściwości. Nazwy właściwości i metod wykorzystywanych do strumieniowania są definiowane przez metodę `DefineProperties()`. Gdy chcemy strumieniować niepublikowaną właściwość, musimy o tym poinformować C++Buildera. Wykonujemy to, przysłaniając metodę `DefineProperties()`.

Listing 4.61 zawiera przykładowy komponent, `TSampleComp`, posiadający trzy niepublikowane właściwości. Komponent zapewnia ich strumieniowanie za pomocą dodatkowych metod. W trakcie wykonywania programu tworzy drugi komponent o nazwie `TComp`, na który wskazuje właściwość `Comp3`. Ponieważ komponent ten nie jest upuszczany w formularzu przez programistę, jego właściwości nie są automatycznie strumieniowane do pliku. Zapewniamy więc kod konieczny do zapisania tych danych.

Listing 4.61. *Komponent strumieniujący niepublikowane właściwości*

```
// minimalna deklaracja klasy zapewniająca kompilację przykładu
class TComp : public TComponent
{
public:
  __fastcall TComp::TComp(TComponent *Owner) : TComponent(OWner) {}
};

class TSampleComp : public TComponent
{
private:
  int FProp1;
  AnsiString FProp2;
  TComp *FComp3;
  void __fastcall ReadProp1(TReader *Reader);
  void __fastcall WriteProp1(TWriter *Writer);
  void __fastcall ReadProp2(TReader *Reader);
  void __fastcall WriteProp2(TWriter *Writer);
  void __fastcall ReadComp3(TReader *Reader);
  void __fastcall WriteComp3(TWriter *Writer);

protected:
  void __fastcall DefineProperties(TFiler *Filer);

public:
  __fastcall TSampleComp(TComponent* Owner);
  __fastcall ~TSampleComp(void);

  __property int Prop1 = {read = FProp1, write = FProp1, default = 10};
  __property AnsiString Prop2 = {read = FProp2, write = FProp2, nodefault};
  __property TComp *Comp3 = {read = FComp3, write = FComp3};
};

void __fastcall TSampleComp::TSampleComp(TComponent* Owner) : TComponent(Owner)
{
  FProp1 = 10;              // domyślna
  FComp3 = new TComp(NULL); // musimy to strumieniować
}

void __fastcall TSampleComp::~TSampleComp (void)
```

```
{
  if(FComp3)
    delete FComp3;
}

void __fastcall TSampleComp::DefineProperties(TFiler *Filer)
{
  // najpierw wywołujemy metodę bazową
  TComponent::DefineProperties(Filer);
  Filer->DefineProperty("Prop1", ReadProp1, WriteProp1, (FProp1 != 10));
  Filer->DefineProperty("Prop2", ReadProp2, WriteProp2, (FProp2 != ""));

  // sprawdzamy, czy właściwości Comp3 muszą zostać zapisane
  bool WriteValue;
    if(Filer->Ancestor) // sprawdzamy dziedziczoną wartość
    {
    TSampleComp *FilerComp = dynamic_cast<TSampleComp *>(Filer->Ancestor);
    if(FilerComp->Comp3 == NULL)
      WriteValue = (Comp3 != NULL);
    else
    {
      if((Comp3 == NULL) || (FilerComp->Comp3->Name != Comp3->Name))
        WriteValue = true;
      else
        WriteValue = false;
    }
  }
  else // nie jest to wartość dziedziczona, zapisz właściwość, jeśli różna od null
    WriteValue = (Comp3 != NULL);

  Filer->DefineProperty("Comp3", ReadComp3, WriteComp3, WriteValue);
}

void __fastcall TSampleComp::ReadProp1(TReader *Reader)
{
  Prop1 = Reader->ReadInteger();
}

void __fastcall TSampleComp::WriteProp1(TWriter *Writer)
{
  Writer->WriteInteger(FProp1);
}

void __fastcall TSampleComp::ReadProp2(TReader *Reader)
{
  FProp2 = Reader->ReadString();
}

void __fastcall TSampleComp::WriteProp2(TWriter *Writer)
{
  Writer->WriteString(FProp2);
}

void __fastcall TSampleComp::ReadComp3(TReader *Reader)
{
  if(Reader->ReadBoolean())
    FComp3 = (TComp *)Reader->ReadComponent(NULL);
}
```

```
void __fastcall TSampleComp::WriteComp3(TWriter *Writer)
{
  if(FComp3)
  {
    Writer->WriteBoolean(true);
    Writer->WriteComponent(Comp3);
  }
  else
    Writer->WriteBoolean (false);
}
```

Metoda `DefineProperties()` zawiera dwa wiersze rejestrujące dwie pierwsze właściwości.

```
Filer->DefineProperty("Prop1", ReadProp1, WriteProp1, (FProp1 != 10));
Filer->DefineProperty("Prop2", ReadProp2, WriteProp2, (FProp2 != ""));
```

Informujemy w nich C++Buildera, aby korzystał z dostarczonych metod zapisu i odczytu w trakcie strumieniowania tych właściwości. Ostatni parametr to znacznik określający, czy należy zapisywać dane. Właściwości mają być strumieniowane tylko wtedy, gdy ich aktualne wartości różnią się od wartości domyślnych.

Właściwość `Comp3` jest nieco inna i wymaga dodatkowego wyjaśnienia. Różnica polega głównie na tym, że właściwość jest komponentem (instancja tworzona jest w trakcie działania programu), a nie typem danych. Listing 4.62 przedstawia fragment kodu odpowiedzialny za stwierdzenie, czy należy ją strumieniować.

Listing 4.62. *Określenie, czy właściwość będąca komponentem wymaga strumieniowania*

```
bool WriteValue;
if(Filer->Ancestor) // sprawdzamy dziedziczoną wartość
{
TSampleComp *FilerComp = dynamic_cast<TSampleComp *>(Filer->Ancestor);
if(FilerComp->Comp3 == NULL)
  WriteValue = (Comp3 != NULL);
else
  {
  if((Comp3 == NULL) || (FilerComp->Comp3->Name != Comp3->Name))
    WriteValue = true;
  else
    WriteValue = false;
  }
}
else // nie jest to wartość dziedziczona, zapisz właściwość, jeśli różna od null
  WriteValue = (Comp3 != NULL);

Filer->DefineProperty("Comp3", ReadComp3, WriteComp3, WriteValue);
```

Właściwość reprezentuje komponent tworzony w trakcie wykonywania programu. Ponieważ komponent nie jest upuszczany do formularza, nie jest przeprowadzany domyślny mechanizm strumieniowania. Zajmuje się tym napisana przez nas metoda `DefineProperties()`.

Najpierw musimy stwierdzić, czy właściwość `Ancestor` obiektu `Filer` wynosi `true`, aby zapobiec zapisywaniu wartości właściwości w formularzach potomnych. Skoro nie istnieje wartość dziedziczenia, strumieniujemy właściwość (komponent), gdy `Comp3` jest różne od `NULL`.

Jeśli właściwość `Ancestor` wynosi `true`, analizujemy właściwość `Comp3` przodka. Jeśli wynosi `NULL`, strumieniujemy właściwość (`TSampleComp->Comp3`), gdy jest różna od `NULL`. Jeśli właściwość `Comp3` przodka nie jest równa `NULL`, dokonujemy ostatniego sprawdzenia; jeśli właściwość (`TSampleComp->Comp3`) wynosi `NULL` lub nazwa właściwości `Comp3` jest różna od przodka, strumieniujemy właściwość (komponent).

Na końcu definiujemy właściwość, używając opisanej wcześniej metody `DefineProperty()`.

Poznaliśmy sposoby wykorzystania metody `DefineProperty()`. Dotyczy ona strumieniowania danych typu liczbowego, tekstowego, znakowego lub wyliczeniowego. Istnieje też inna metoda, `DefineBinaryProperty()`, zaprojektowana do strumieniowania danych binarnych, jak obrazy lub dźwięki. Więcej informacji na ten temat znajduje się w sekcji „DefineBinaryProperty" tematu „TWriter" z pomocy C++Buildera.

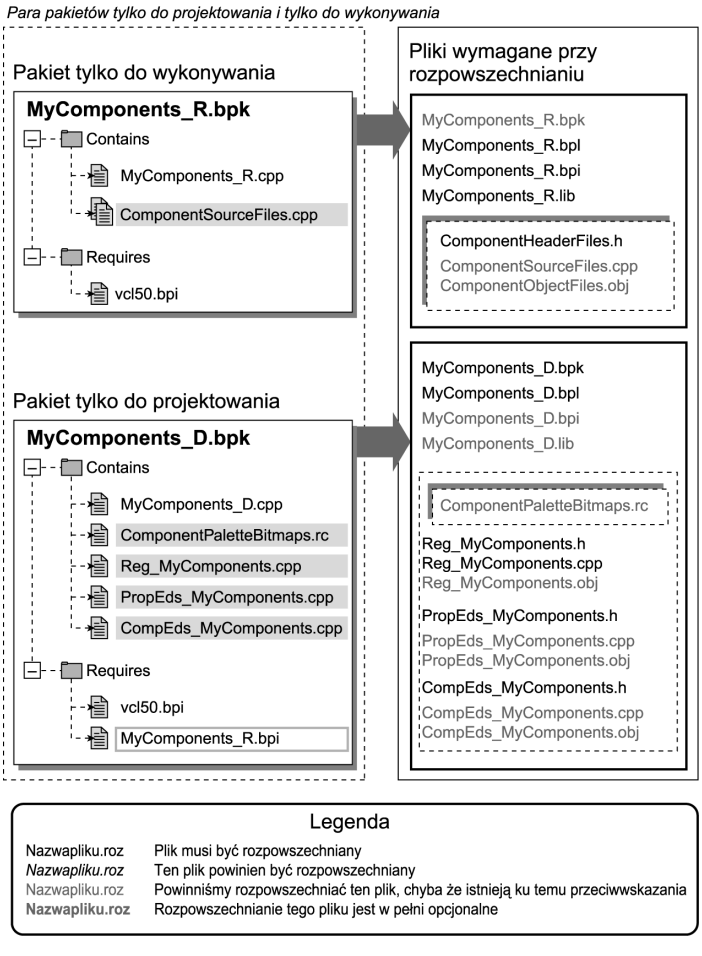
Dystrybucja komponentów

Gdy zamierzamy rozprowadzać komponenty, najważniejszym dla nas plikiem jest plik z rozszerzeniem *.bpl*. Jest to biblioteka pakietu umożliwiająca instalację komponentów w środowisku programistycznym, aby były one dostępne w trakcie projektowania.

Gdy budujemy pakiety, warto przejrzeć plik źródłowy opcji (polecenie *Project/Edit option source*) pakietu i usunąć wszystkie niepotrzebne wpisy z sekcji `<LIBRARIES>` i `<SPARELIBS>`. Jeśli tego nie zrobimy, zapewne zgłosi się wielu zdenerwowanych użytkowników szukających pewnej biblioteki, która nie jest im do niczego potrzebna, ale wymaga jej komponent. Pakiet powinien wymagać tylko tych bibliotek, których naprawdę potrzebuje.

Pliki, które należy udostępnić, aby pakietu mogli używać użytkownicy, przedstawia rysunek 4.6.

Na rysunku 4.6 uwidoczniono, że oprócz wspomnianych wcześniej plików *.bpl*, *.bpi* i *.lib*, należy także udostępnić pliki nagłówkowe (*.h*) wszystkich jednostek kompilacji występujących w sekcji *Contain* pakietu tylko do wykonywania. Powinniśmy również rozpowszechnić pliku nagłówkowe jednostek używanych w pakiecie projektowym. Wyjątkiem od tej zasady jest plik nagłówkowy jednostki rejestracji; ponieważ plik ten jest na ogół pusty, więc jego dystrybucja nie ma sensu. Warto jednak rozpowszechniać plik źródłowy rejestracji, aby użytkownicy mogli sprawdzić, jakie zmiany w środowisku projektowym wprowadza instalacja pakietu. Zauważmy, że na rysunku pojawiły się pliki *.rc* obrazujące pliki zasobów przechowujące bitmapy dla palety komponentów. Zamiast plików *.rc* możemy skorzystać z plików *.res* lub *.dcr*. Omawiamy to dokładniej w dalszej części rozdziału. Pamiętajmy o tym, by użytkownicy mogli kreować własne edytory potomne, gdy w pakiecie dostarczamy edytory właściwości lub komponentów.

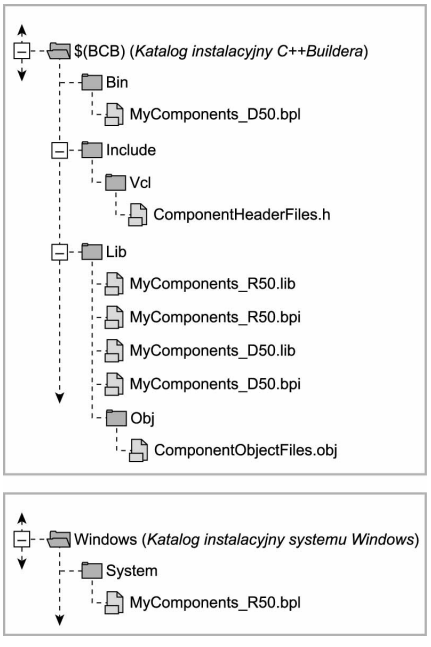


Rysunek 4.6. *Pliki wymagane przy rozpowszechnianiu pakietu*

Pamiętamy zapewne z rozdziału 2. „Projekty i środowisko projektowe C++Buildera", że plik *.lib* to w zasadzie zbiór plików *.obj* jednostek wchodzących w skład pakietu. W zależności od potrzeb użytkownika, udostępniamy plik *.lib* lub pliki *.obj* pakietu wykonywalnego, co umożliwi generowanie aplikacji ze statycznie dołączanymi bibliotekami. Plik *.lib* ma tę przewagę nad plikami *.obj*, że wszystkie obiekty znajdują się w jednym pliku, co ułatwia zachowanie porządku.

Miejsca umieszczania rozpowszechnianych plików

Od tego, gdzie zostaną umieszczone rozpowszechniane pliki na komputerze użytkownika, zależy łatwość korzystania z pakietu. Korzystanie z domyślnych katalogów środowiska C++Builder przedstawionych na rysunku 4.7 gwarantuje, że użytkownik nie będzie zmuszony do modyfikacji ustawień katalogów projektu. Z drugiej strony, wiele osób nie życzy sobie, aby komponenty niezależnych firm były umieszczane w tych samych katalogach, co pakiety Borlanda. Warto wtedy przynajmniej zastosować tę samą strukturę katalogów, co C++Builder.



Rysunek 4.7. *Domyślne katalogi pakietu*

Konwencje nazw użyte na rysunku 4.7 zostały wykorzystane w celu zachowania konsekwencji. Nie pokazujemy modułów źródłowych, ponieważ ich rozmieszczenie ma znaczenie tylko w czasie kompilacji pakietu. Najprostsza alternatywa dla rozmieszczenia plików w katalogach z rysunku 4.7 to umieszczenie wszystkich plików pakietu w jednym katalogu; nie dotyczy to pliku *.bpl*, który powinien znaleźć się w pliku *Windows\System* lub równoważnym. Ułatwi to dodawanie ścieżek pakietu do projektu. Program konsolidujący musi odnaleźć pliki *.bpi* i *.lib* pakietu wykonywalnego (przy założeniu istnienia pary pakietów), więc katalog je zawierający powinien zostać dodany do ustawienia globalnej ścieżki bibliotek (pole *Library Path* z zakładki *Library* okna opcji projektu). Można także zmodyfikować rejestr systemowy, a dokładniej klucz *HKEY_CURRENT_USER\Software\Borland\C++Builder\6.0\Library\Search Path*, gdzie *6.0* to numer wersji (może to być także 1.0, 3.0 lub 5.0). Jeśli katalogi zawierające komponenty znajdują się wewnątrz katalogu instalacyjnego systemu, zamiast nazwy katalogu instalacyjnego stosujemy tekst *$(BCB)*.

Nazwy pakietów i ich jednostek

Przyjęty sposób nazewnictwa pakietów i jednostek jest bardzo ważny, gdy zamierzamy rozpowszechniać pakiet. Zakładając stosowanie pary pakietów (projektowania i wykonywania), musimy tak nazwać pakiet (lub pakiety) wykonywania, aby można go było łatwo odróżnić od pakietu projektowego. Na ogół do nazwy pakietu wykonywania dodaje się litery *_R* lub *_RTP*, a do pakietu projektowego litery *_D* lub *_DTP*. Znak podkreślenia jest opcjonalny.

Powinniśmy także dodawać numer reprezentujący wersję VCL wykorzystywaną do tworzenia pakietu. W przypadku pakietów wykonywanych w C++Builderze 6, będzie to 60 (wersja VCL dla tej wersji środowiska). W ten sposób staje się oczywiste, dla jakiej

wersji C++Buildera przewidziany jest pakiet. Na przykład pakiet wykonywania dla wersji 6. powinien zawierać w nazwie tekst *_R60*, a pakiet projektowy dla tej wersji — *_D60* (tutaj także znak podkreślenia jest opcjonalny). Dokładny sposób opisania numeru wersji nie ma znaczenia, ważne, by od razu rzucał się w oczy. Na przykład konwencja wykorzystywana w pakietach VCL to zamiana V na D dla VCL (powstaje DCL) w pakietach tylko do projektowania. W obydwu rodzajach pakietów numer wersji umieszczany jest na końcu nazwy pliku. Na przykład nazwy pakietu dla komponentów dostępu do danych to *VCLDB50.bpl* i *DCLDB50.bpl* odpowiednio dla pakietu wykonywania i projektowego.

Oczywiście niezależnie od stosowania konwencji nazewnictwa umożliwiających rozróżnianie pakietów projektowych od wykonywanych wszystkie pakiety muszą posiadać unikalne nazwy. Nie jest możliwa instalacja lub użycie pakietu, którego nazwa koliduje z nazwą pakietu już istniejącą w tej samej aplikacji.

Nazwy jednostek pakietu są równie ważne, jak nazwa samego pakietu. Zanim zajmiemy się stosowanymi konwencjami, musimy ostrzec, że jednostki eksportowane z pakietu *muszą być unikalne* dla każdej korzystającej z nich aplikacji (nie jest możliwe posiadanie jednostek o takich samych nazwach w dwóch pakietach używanych jednocześnie przez tę samą aplikację), a także dla środowiska projektowego.

Należy rozważyć dwa przypadki. Pierwszy dotyczy nazw jednostek występujących tylko w pakiecie projektowym (jednostki zawierające kod rejestracji oraz jednostki edytorów właściwości i komponentów). Drugi dotyczy nazw jednostek znajdujących się w pakiecie wykonywania (jednostki zawierające kod źródłowy komponentów).

W nazwach jednostek pakietu projektowego powinniśmy umieszczać nazwę pakietu. Zapewni to unikalność nazwy jednostki. Na przykład jednostka zawierająca kod rejestracji może uzyskać nazwę *Registration_NazwaPakietu.cpp*. Zmiana kolejności słów, zastąpienie *Registration* przez *Reg* (i tym podobne), a także stosowanie znaków podkreślenia nie ma znaczenia. Ważne, by nazwa była unikalna i jednocześnie mówiła coś o zastosowaniu jednostki. Ponieważ nazwa pakietu musi być unikalna, dołączanie jej do nazwy jednostki zapewnia, że ona także będzie unikalna. Z tego powodu dla pakietu projektowego o nazwie *NewComponentD60* odpowiednie są następujące nazwy jednostki z kodem rejestracji:

```
Reg_ NewComponentD60,
RegNewComponentD60,
Registration_ NewComponentD60,
RegistrationNewComponentD60.
```

Możliwych jest wiele rozwiązań. Wystarczy wybrać jedno z nich i konsekwentnie się do niego stosować. Wymóg unikalności nazw wyklucza stosowanie oczywistych nazw dla jednostek edytorów komponentów lub właściwości (na przykład *PropertyEditors.cpp* lub *ComponentEditors.cpp*). Najprostszy sposób zapewnienia niepowtarzalności nazwy to stosowanie się do podanych wcześniej wskazówek; należy dołączać nazwę pakietu do nazwy jednostki. Istnieje jednak alternatywny sposób uzyskania unikalności nazw jednostek zawierających edytory właściwości i komponentów. W tych jednostkach można zastosować dyrektywę `#pragma package(smart_init, weak)` zamiast `#pragma package (smart_init)`. Spowoduje to „słabe” dołączenie jednostek do pakietu projektowego. Wtedy

nazwa jednostki staje się nieistotna, ponieważ nie jest dołączana do pakietu projektowego. Zamiast tego kod jednostki dostępny jest bezpośrednio, gdy jest wymagany. Wszystko będzie działało poprawnie, o ile nazwy klas edytorów właściwości i komponentów są unikalne.

Drugi przypadek rozważamy, gdy dobieramy nazwy jednostek należących do pakietu wykonywania. Jeśli cały komponent znajduje się w jednej jednostce kompilacji, rozsądne wydaje się zastosowanie nazwy komponentu jako nazwy pliku (bez początkowego `T`). Ponieważ nazwa komponentu musi być unikalna (patrz następny podrozdział), stosowanie tej konwencji w zasadzie zapewnia niepowtarzalność nazw. Dla jednostek zawierających kilka komponentów sensowna konwencja nazw polega na wybraniu nazwy opisującej komponenty i dodaniu nazwy pakietu — podobną metodę stosujemy dla nazw jednostek pakietu projektowego. Jeśli zastosujemy się do przedstawionych wskazówek, ryzyko wywołania konfliktu nazw zdecydowanie maleje.

Nazwy komponentów

Dobór odpowiedniej nazwy komponentu jest bardzo ważny. Należy wziąć pod uwagę dwie sprawy — nazwa komponentu musi być unikalna, a po drugie, musi jednoznacznie określać zastosowanie komponentu.

Zapewnienie unikalnej nazwy komponentu nie jest tak proste, jak się początkowo wydaje. Programiści piszą naprawdę wiele komponentów wykonujących to samo zadanie, na przykład otaczających port szeregowy komputera. Możliwych kombinacji `TComport` jest tyle, ile zapragniemy. Z tego powodu programiści zajmujący się dystrybucją komponentów umieszczają swoje inicjały między literą `T` a właściwą nazwą komponentu. Napisanie inicjałów tylko wielką lub tylko małą literą ułatwia ich ignorowanie. Niektórzy twierdzą, że wielkie litery łatwiej się ignoruje niż małe z powodu symbolicznej natury tych pierwszych. Stosowanie inicjałów przypomina konwencje nazw stosowane dla wyliczeń właściwości komponentów. Dobór komunikatów powinien być dowolny lub składać się z inicjałów firmy. Dla firmy o nazwie *Komponenty dla Buildera* inicjały będą następujące: kdb. Jeśli kreujemy komponent portu szeregowego, nadamy komponentowi nazwę `TkdbComport`. Może nie jest to zbyt ładne, ale gdy użytkownik nie ma dostępu do kodu źródłowego komponentu, jest to konieczne. Prawdopodobieństwo, że inna firma tworzy takie same komponenty i posiada takie same inicjały, jest niewielkie. Wspomnieliśmy o dostępie użytkownika do kodu źródłowego komponentu. Jeśli użytkownik ma pełny kod źródłowy, nazwa komponentu ma mniejsze znaczenie, gdyż można ją zmienić w dowolnym momencie.

Dobór nazwy, która krótko i jednoznacznie charakteryzuje działanie komponentu, wymaga czasem długiego myślenia. Należy stosować najbardziej typowe określenia, a także uważać na różne znaczenia stosowanych słów. Na przykład dla komponentu otaczającego port szeregowy należy unikać stosowania nazwy `TCOM`, ponieważ kojarzy się ona bardziej z programowaniem obiektów COM. Przykład jest stosunkowo prosty, ale bardzo dobrze ilustruje, o co chodzi. Jeśli mamy trudności z doborem odpowiedniej nazwy dla komponentu, może to świadczyć o jego błędnym zaprojektowaniu. Zapewne przydatne okaże się ponowne przemyślenie zagadnienia.

Dystrybucja wyłącznie pakietu projektowego

Do tej pory mówiliśmy o rozprowadzaniu zestawu pakietów — jednego pakietu projektowego wraz z jednym lub kilkoma dotyczącymi go pakietami wykonywania. W tym podrozdziale przyjrzymy się pokrótce dystrybucji komponentów za pomocą modelu wykorzystującego tylko pakiet projektowy. Jeśli tak zrobimy, użytkownik będzie zmuszony do statycznego dołączania plików obiektów (*.obj*), aby korzystać z komponentów w aplikacji. Może się wydawać, że utrudnia to użytkownikowi korzystanie z komponentów, ale w rzeczywistości tak nie jest. Gdy dodajemy do formularza aplikacji komponent nie będący VCL, środowisko projektowe wykonuje dwa zadania: dołącza plik nagłówkowy zawierający definicję komponentu i dodaje instrukcję `#pragma link ("nazwajednostki")` do pliku implementacyjnego formularza (*nazwajednostki* to nazwa jednostki implementującej komponent). Powoduje to statyczne dołączenie komponentu do aplikacji. Użytkownik pisze tylko dodatkowy kod. Przy założeniu, że program konsolidujący odnajdzie pliki *.obj*, wszystko będzie działało poprawnie. Rysunek 4.8 przedstawia strukturę dystrybucji zawierającej tylko pakiet projektowy. Dodatkowo przedstawione są pliki, które należy rozpowszechniać.

Oczywiście zamiast kilkunastu plików *.obj* równie dobrze można by rozpowszechniać bibliotekę statyczną (*.lib*), która przecież zawiera pliki obiektów, ale w bardziej poręcznej formie. Zadanie to da się osiągnąć na dwa sposoby. Można utworzyć pakiet wykonywania zawierający tylko i wyłącznie komponenty. Po jego zbudowaniu plik *.lib* będzie zawierał wszystkie wymagane pliki *.obj*. Pozostałe pliki pakietu, czyli *.bpi* i *.bpl*, nie są potrzebne. W ten sposób będziemy mogli rozpowszechniać wygodniejszy plik *.lib* zamiast wielu plików *.obj*. Inna możliwość to bezpośrednie dodanie plików *.obj* do pakietu projektowego za pomocą makra `USEOBJ`. Wtedy rozpowszechniamy plik *.lib* wygenerowany dla pakietu projektowego. Jeśli program konsolidujący nie potrafi odnaleźć pliku *.lib*, musimy poddać edycji plik źródłowy opcji i dodać nazwę biblioteki statycznej do wiersza `<SPARELIB>` w sposób przedstawiony dalej.

```
<SPARELIBS value="VCL50.lib MyStaticLibrary.lib"/>
```

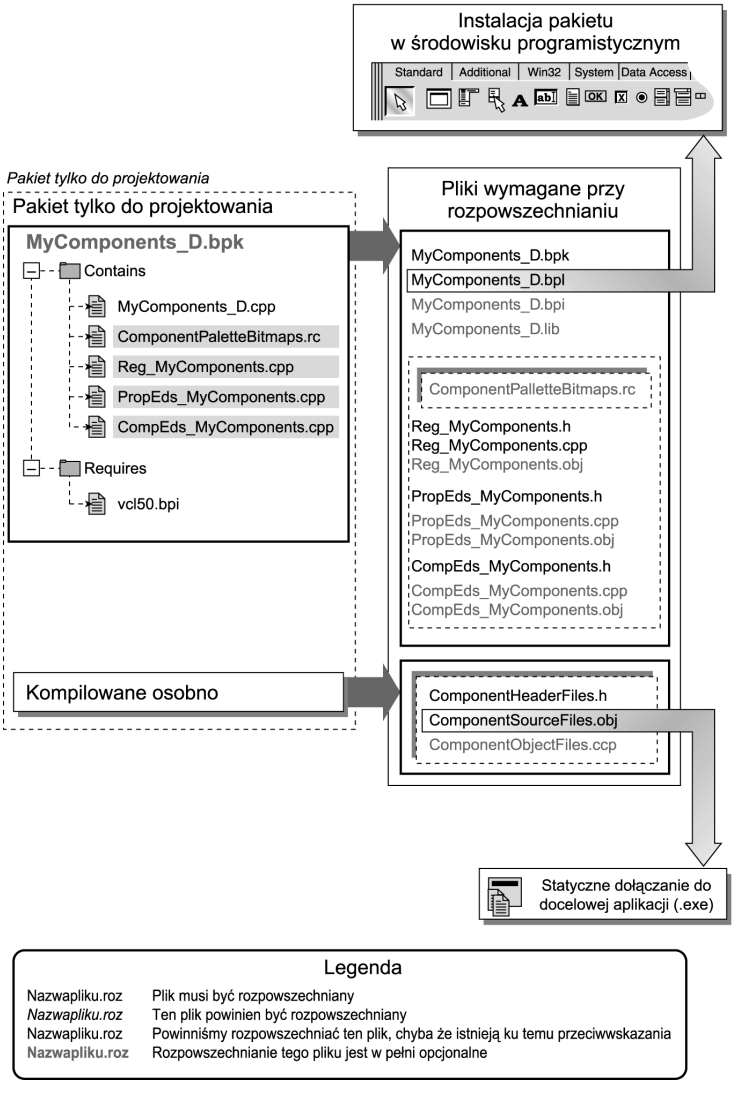
Jeśli plik ten znajduje się w ścieżce wyszukiwania bibliotek danego projektu, zostanie poprawnie dołączony.

Zaletą korzystania tylko z pakietu projektowanego jest to, że zajmujemy się wyłącznie jednym pakietem. W trakcie tworzenia pakietu dla różnych wersji C++Buildera (temat następnego podrozdziału) takie podejście znacznie ułatwia pracę. Pliki obiektów dla komponentów generujemy, kompilując komponenty w różnych wersjach C++Buildera, oczywiście przy założeniu, że kod napisany jest w taki sposób, iż może być kompilowany dla każdej z docelowych wersji. Z tego modelu pakietu korzystamy też wtedy, gdy z jakiegoś powodu nie chcemy rozpowszechniać biblioteki dołączanej dynamicznie. Ogólnie jednak preferowane jest wykorzystywanie pary pakietów (projektowego i wykonywania) i takie rozwiązanie powinno mieć pierwszeństwo.

Rozpowszechnianie komponentów dla różnych wersji C++Buildera

Gdy chcemy rozpowszechniać komponenty jako ich producent, zapewne będziemy zamierzali rozpowszechniać je dla jak największej liczby wersji C++Buildera (aktualnie istnieje pięć wersji: 1., 3., 4., 5. i 6.). Oczywiście na pewno będą istniały przypadki,

Rysunek 4.8.
Model tylko z pakietem projektowym



w których nie będzie to możliwe, ponieważ kod komponentu będzie korzystał z pewnych funkcji dostępnych tylko w nowszych wersjach kompilatora. Jeśli jednak założymy, że komponent może działać w więcej niż jednej wersji środowiska (bardzo częsta sytuacja), musimy wykonać kilka wersji komponentu. Dystrybucja dla różnych wersji wymaga kompilacji komponentu w tych wersjach C++Buildera, dla których jest on przeznaczony. Poza tym musimy wykreować pakiet dla komponentów instalowanych w wersjach 3., 4., 5. i 6. (w wersji 1. komponenty były instalowane bezpośrednio w *CMPLIB32.CCL* i nie istniały pakiety). Z tego powodu warto rozważyć opisane wcześniej modele. Korzystamy z odpowiednich bibliotek importu dla każdej wersji C++Buildera, a pakiety kreujemy, jak poprzednio. Wszystko będzie działało przy założeniu, że źródła komponentu będą poprawnie kompilowane w każdej wersji kompilatora. Jest to mało prawdopodobne, więc właśnie na tym etapie pojawia się najwięcej problemów. Stosowanie osobnych kodów źródłowych dla każdej wersji kompilatora jest niepraktyczne, dlatego

trzeba stosować inne podejście. Wykorzystuje ono te same pliki źródłowe jednostek dla wszystkich wersji kompilatora. Wykrywana jest wersja kompilatora, a preprocesor służy do wybrania odpowiedniego kodu. Podejście to, a także inne zagadnienia związane z wersjami kompilatora omawiamy w kolejnych podrozdziałach.

Wykrywanie wersji kompilatora w trakcie kompilacji

Każda wersja C++Buildera definiuje własny numer wersji. Sprawdzając jego wartość, można wykonywać różne fragmenty kodu w zależności od wersji kompilatora. Listing 4.63 przedstawia sposób ustawiania wartości `#define` w zależności od wersji kompilatora.

Listing 4.63. *Ustawianie #define w zależności od wersji kompilatora*

```
#ifndef VERSION_DEFINES
#define VERSION_DEFINES

  #if(__TURBOC__ == 0x550) // C++Builder 5
  #define CPPBUILDER_VERSION_5
  #endif

  #if(__TURBOC__ == 0x540) // C++Builder 4
  #define CPPBUILDER_VERSION_4
  #endif

  #if(__TURBOC__ == 0x530) // C++Builder 3
  #define CPPBUILDER_VERSION_3
  #endif

  #if(__TURBOC__ == 0x520) // C++Builder 1
  #define CPPBUILDER_VERSION_1
  #endif
#endif
```

Jeśli umieścimy ten kod w pliku nagłówkowym komponentu lub w odrębnym pliku nagłówkowym dołączanym do komponentu, będzie możliwe stosowanie następującego kodu:

```
#ifdef CPPBUILDER_VERSION_5
//  rejestracja filtrów właściwości obsługiwana tylko przez wersję 5. i 6.
#endif
```

Stosując konstrukcje `#ifdef/#endif` oraz `#ifndef/#endif`, można wybiórczo stosować kod w zależności od wersji C++Buildera. W rozdziale 3. odradzaliśmy korzystanie z preprocesora, o ile nie jest to niezbędne. To jest akurat jeden z przypadków, kiedy preprocesor pozwala nam uniknąć mnóstwa problemów.

Korzystanie z funkcji ValidCtrCheck()

Tej funkcji używamy do sprawdzenia, czy któryś z komponentów, które zamierzamy instalować, nie zawiera metod abstrakcyjnych. Wykrywanie odbywa się w trakcie kompilacji. Komponenty posiadające metody abstrakcyjne nie mogą być instalowane w środowisku projektowym. Dla wersji 1. C++Buildera obowiązuje jednak nieco inna wersja funkcji. Przyjmuje ona taką postać:

```
static inline TNazwaKomponentu *ValidCtrCheck()
{
  return new TNazwaKomponentu(NULL);
}
```

W wersjach 3., 4. 5. i 6. C++Buildera poprawna jest natomiast następująca postać funkcji `ValidCtrCheck()`:

```
static inline void ValidCtrCheck(TNazwaKomponentu *)
{
  new TNazwaKomponentu(NULL);
}
```

Pakiety a C++Builder 1

Pierwsza wersja kompilatora nie korzysta z pakietów, więc nie należy stosować makra `PACKAGE` obok słowa kluczowego `class` w definicji komponentu ani przed funkcją rejestrującą `Register()`. W pliku źródłowym komponentu nie może się znaleźć dyrektywa `#pragma package(smart_init)`.

Ponieważ 1. wersja kompilatora nie używa pakietów, rozpowszechniane powinny być tylko pliki nagłówkowe i obiektów. Opcjonalnie można udostępniać osobną jednostkę z kodem rejestrującym.

Korzystanie ze zbiorów w komponentach

Podany opis dotyczy implementacji zbiorów w różnych wersjach C++Buildera. Ogólnie problem polega na tym, że ich stosowanie w wersji 1. i 3. różni się od podejścia, którego użyto w kolejnych wersjach kompilatora. Dalej wyjaśniamy całe zagadnienie.

Pakiet MJFSecurity dla C++Buildera 3 autorstwa Malcolma Smitha (dostępny na stronie *http://www.mjfreelancing.com*) zawiera w pliku nagłówkowym następujący kod.

```
#include <sysdefs.h>

enum TFailedShareRegKey { fsrNone, fsrInstalledDate, fsrRegUser, fsrRegOrgn,
                          fsrRegCode, fsrRunCount, fsrUserDefined };

typedef Set<TFailedShareRegKey, fsrNone, fsrUserDefined> TFailedShareRegKeys;

typedef void __fastcall (__closure *TLoadErrorEvent) (TObject *Sender,
                         TFailedShareRegKeys FailedKeys, bool &Terminate);
```

W pliku implementacji znajduje się kod podobny do tego przedstawionego na listingu 4.64.

Listing 4.64. *Kod źródłowy obrazujący korzystanie ze zbiorów*

```
TFailedShareRegKeys FailedKeys;
FailedKeys << fsrNone;

// ... przykładowy kod dotyczący odczytu rejestru:

if(MyReg->ValueExists(KeyNames->InstalledDate))
  FInstalledDate = MyReg->ReadDate(KeyNames->InstalledDate);
```

```
else
{
  FailedKeys >> fsrNone;
  FailedKeys << fsrInstalledDate;
}

if(MyReg->ValueExists(KeyNames->Username))
  FRegisteredName = MyReg->ReadString(KeyNames->Username);
else
{
  FailedKeys >> fsrNone;
  FailedKeys << fsrRegUser;
}

// ... i tak dalej
// ... a teraz kod wywołujący zdarzenie:

if(FOnLoadError)
{
  bool Terminate = TerminateOnLoadError;
  FOnLoadError(this, FailedKeys, Terminate);
}
```

Ten kod powinien umożliwiać komponentowi konstrukcję zbioru definiującego wszystkie możliwe powody błędu wczytywania aplikacji. Zbiór jest przekazywany do zdarzenia `OnLoadError` jako parametr, aby użytkownik mógł sprawdzić informację i odpowiednio na nią zareagować.

Przedstawiony kod będzie działał poprawnie w C++Builderze 4 i 5, ale nie w wersjach 1. ani 3. Dla tych wersji wymagana jest taka deklaracja:

```
template class TFailedShareRegKeys;
```

Ta jawna deklaracja wymusza na kompilatorze kompilację wszystkich metod klasy `Set`.

C++Builder 4 i 5 zawiera już taką jawną deklarację w pliku *$(BCB)\Include\Vcl\sysmac.h* dołączanym pośrednio przez wiersz `#include <system.hpp>`. Kod tej deklaracji jest następujący.

```
template<class T, unsigned char minEl, unsigned char maxEl>
  class RTL_DELPHIRETURN Set;
```

Otaczając wiersz

```
template class TFailedShareRegKeys;
```

dyrektywą preprocesora, możemy powodować jej kompilację w wersjach 1. i 3., a ignorowanie w wersjach późniejszych.

Tworzenie bitmap dla palety komponentów

Przy omawianiu pakietów nie można pominąć bitmap palety komponentów określanych w plikach *.rc* (skrypty zasobów). Korzystanie z tych plików zasobów (lub jednego pliku z wpisami dotyczącymi wszystkich bitmap, co jest preferowanym sposobem) daje nam

większą kontrolę nad tworzeniem bitmap palety. Tworzenie bitmap palety komponentów w odpowiednich narzędziach, a następnie ręczne dołączanie ich do skryptu zasobów umożliwia wydajniejsze wykorzystanie własnych palet. Tym sposobem możemy uatrakcyjnić wygląd bitmap komponentów na palecie.

Wskazówki dotyczące projektowania komponentów przeznaczonych do rozpowszechniania

Pisanie komponentów do użytku w domu lub w firmie to nie to samo co pisanie komponentów wykorzystywanych przez nieznanych nam użytkowników lub firmy. Największym problemem jest to, że nie wiemy, w jaki sposób będzie wykorzystywany komponent. Z tego powodu w trakcie jego projektowania należy pamiętać o następujących wskazówkach.

- Nie ukrywajmy działań, z których użytkownik prawdopodobnie nie będzie korzystał. Jeśli udostępnienie pewnej funkcji nie jest zagrożeniem dla projektu komponentu, warto pozostawić ją jako publiczną. Zawsze znajdzie się ktoś potrzebujący danej funkcji.
- W trakcie dodawania zdarzeń należy rozważyć wszystkie możliwe sytuacje, w których użytkownik chciałby otrzymać zdarzenie. Uniemożliwienie użytkownikowi odpowiedzi na kilka potrzebnych mu zdarzeń zmniejszy użyteczność komponentu.
- Nie należy zmuszać użytkownika do łączenia kilku komponentów w celu uzyskania określonych działań. Gdy na przykład piszemy komponent przechwytujący dane z karty dźwiękowej i komponent wyświetlający te dane, sensowne wydaje się umożliwienie ich połączenia. Jeśli jednak jest to jedyny sposób korzystania z komponentów, ich zastosowanie staje się bardzo ograniczone — na przykład gdy jesteśmy zmuszeni do wyświetlenia przechwyconych danych, by móc dokonać jakiejkolwiek ich modyfikacji.
- Interfejs komponentów powinien być intuicyjny, szczególnie w trakcie projektowania, gdyż wtedy to użytkownicy po raz pierwszy zetkną się z komponentem. Gdy jest to właściwe, inteligentnie korzystajmy z edytorów właściwości i komponentów. Sensowne wykorzystanie filtrów właściwości (kategorii) zapewne ułatwi nawigację po wielu właściwościach.
- Ewentualnie możemy wykonać abstrakcyjną klasę bazową dla komponentu, podobnie jak C++Builder zawiera klasę `TCustom`. Nie zawsze będzie to miało sens, ale w pewnych sytuacjach może się okazać bardzo użyteczne.

Inne zagadnienia związane z dystrybucją komponentów

Pozostałe problemy związane z dystrybucją komponentów dotyczą takiego dostosowywania edytorów komponentów, aby było dostępne łącze do strony producenta, wyświetlane logo firmy lub inne informacje (na przykład wersja komponentu).

Musimy się też zastanowić, czy udostępniać komponent bezpłatnie (*freeware*), czy za pewną opłatą (na przykład *shareware*). Czy chcemy dołączać kod źródłowy? Czy umieścić komponenty w programie instalacyjnym, aby instalacja komponentu była możliwa dopiero po zaakceptowaniu warunków licencji? Takich pytań jest więcej. Te zagadnienia wykraczają poza ramy tego rozdziału. Warto jednak ustalić sposób dystrybucji, zanim przystąpi się do rozpowszechniania komponentu.

Ostatnia uwaga: do wszystkich komponentów poza naprawdę banalnymi powinniśmy dołączać dokumentację. Może to być plik pomocy, dokument elektroniczny, a nawet wyczerpująca instrukcja i komentarze w plikach nagłówkowych komponentów. Zaleca się napisanie zarówno pliku pomocy, jak i pewnego dokumentu elektronicznego do druku. Niezależnie od tego, jak dobry byłby komponent, jego stosowanie bez dokumentacji jest mocno ograniczone. Powinniśmy poza tym dostarczyć przykładowy kod wykorzystujący komponent (na przykład prostą aplikację prezentującą jego możliwości).

Podsumowanie

W tym rozdziale przedstawiliśmy zagadnienia związane z tworzeniem komponentów różnego typu, a także sposoby ich dystrybucji i projektowania.