Jacek Matulewski

e-mail: jacek@fizyka.umk.pl

WWW: http://www.fizyka.umk.pl/~jacek

Ostatnia aktualizacja: 19 kwietnia 2008

Krótkie wprowadzenie do projektowania interfejsu aplikacji przy użyciu biblioteki MFC

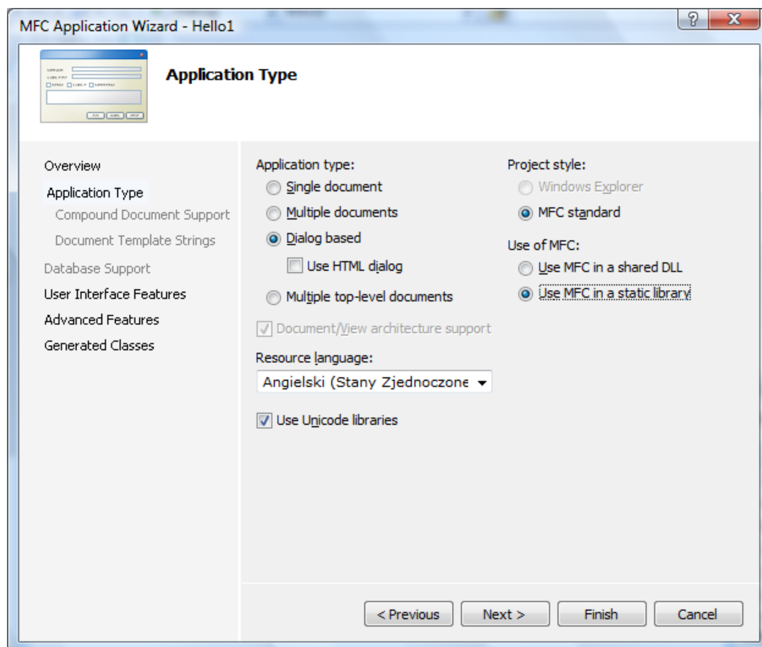
WinAPI

Zacznijmy od wyjaśnienia ważnego pojęcia, które będzie się przewijać przez całą książkę. Co to jest WinAPI? Otóż jest to interfejs programistyczny Windows (*Windows Application Programming Interface*). A w praktyce zbiór bibliotek DLL znajdujących się w katalogu systemowym Windows udostępniających zestaw funkcji pozwalających na kontrolę systemu, budowanie graficznego interfejsu użytkownika i tego typu rzeczy. Więcej konkretnych informacji o WinAPI znajdzie Czytelnik w kolejnych rozdziałach, a także na stronie MSDN: *http://msdn2.microsoft.com/en-us/library/aa383750.aspx*.

Tworzenie projektu

Od czego zaczniemy? Proponuję utworzenie projektu prostej aplikacji, której okno pozbawione będzie wszelkich dodatków typu menu, paska narzędzi itp. Do okna aplikacji dodamy tylko jeden przycisk, którego kliknięcie pokaże komunikat o treści „Hello World!”. Wszystko to uzyskamy minimalnym nakładem pracy.

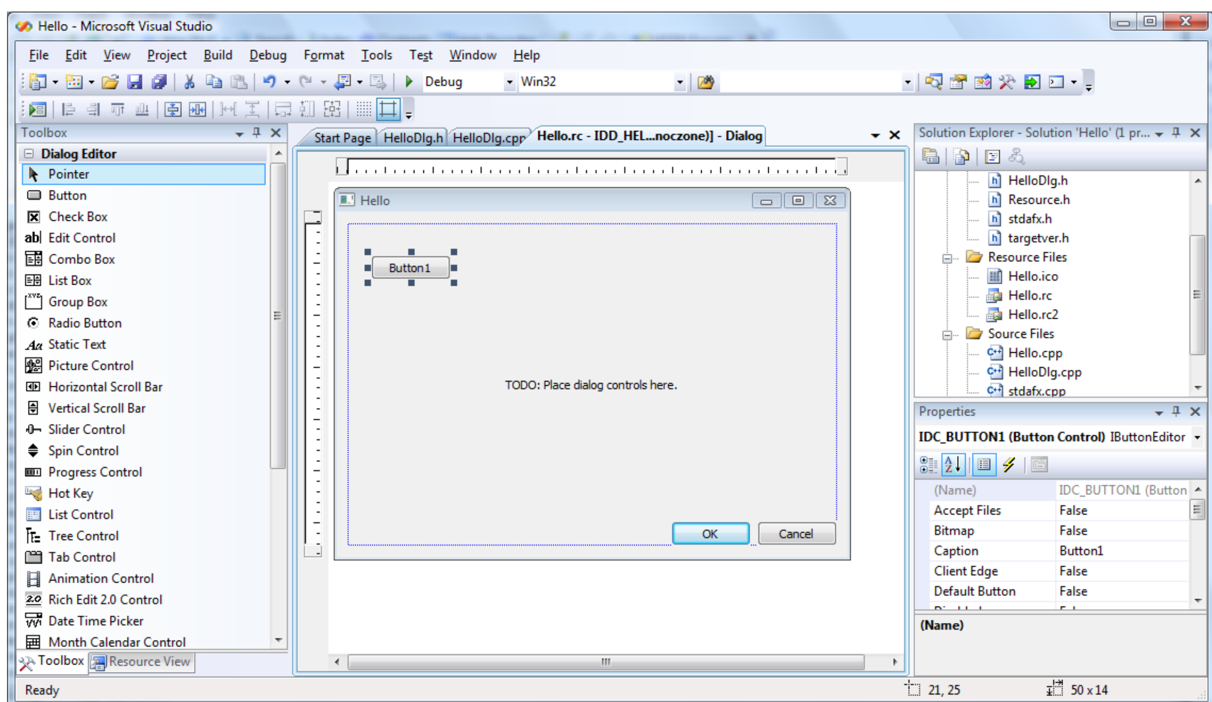
1. Aby utworzyć nowy projekt naciskamy kombinację klawiszy *Ctrl+Shift+N* lub z menu *File*, *New*, wybieramy pozycję *Project*. Następnie:
    a. w panelu *Project types* wybieramy *Other languages*, *Visual C++*, *MFC*,
    b. wówczas w panelu *Templates* wybieramy *MFC Application*,
    c. w polu edycyjnym *Name* wpisujemy nazwę projektu np. „Hello”,
    d. klikamy OK.
2. Pojawi się kreator aplikacji *MFC Application Wizard*. Klikamy *Next >*.
3. W drugim kroku kreatora wybieramy (rysunek 1):
    a. *Application type*: *Dialog based* (to ważne ustawienie, proszę go nie przeoczyć!)
    b. *Use of MFC*: *Use MFC in a static library*.
    c. zaznaczona niech zostanie opcja *Use Unicode libraries*.



Rysunek 1. Kreator aplikacji MFC w VC++ 2008

4. W trzecim kroku (*User Interfaces Features*):
   a. usuwamy zaznaczenie przy pozycji *About box*,
   b. natomiast pozostawiamy zaznaczone *System menu*,
   c. zaznaczamy także *Minimize box* i *Maximize box*,
   d. tytuł okna (pole *Dialog title*) ustawiamy na „Hello World!".
5. Klikamy *Finish*.

Typ aplikacji, jaki wybraliśmy (punkt 3a) ma zasadnicze znaczenie dla wygody programowania. Ponieważ VC++ wyposażony jest tylko w narzędzia wizualne przeznaczone do projektowania okien dialogowych (rysunek 2), wybranie innego typu aplikacji pozbawiłoby nas możliwości projektowania interfejsu aplikacji myszką.



Rysunek 2. Widok projektowania okna dialogowego

Proszę także zwrócić uwagę na opcję *Use Unicode libraries* (punkt 3c). Zaznaczenie tej opcji oznacza, że w przypadku funkcji pobierających łańcuchy (dotyczy to także funkcji WinAPI) będziemy korzystać z funkcji WinAPI z postfixem `W`. Ich argumentami nie mogą być zwykłe tablice jednobajtowych znaków `char`, a tablice znaków `wchar_t`. Ale o tym więcej za chwilę.

Uzyskamy projekt aplikacji, której okno pozbawione będzie wszelkich dodatkowych elementów takich jak pasek narzędzi, menu itp. Wyposażone jest jednak w dwa domyślne przyciski o etykietach *OK* i *Cancel*, których zresztą się niebawem pozbędziemy. Projekt można natychmiast skompilować naciskając klawisz *F5*. Pojawi się co prawda okno dialogowe informujące nas o braku plików binarnych (to chyba nie dziwi), ale wystarczy kliknąć *Yes*, aby uzyskać plik *.exe*. Jeżeli nie chcemy, aby to okno pojawiało się przy każdej kompilacji, zaznaczmy pole opcji *Do not show this dialog again* przed kliknięciem przycisku *Yes*.

Cały projekt składa się z dwóch zasadniczych klas. Pierwsza, to `CHelloApp`, która jest klasą całej aplikacji. Druga to klasa `CHelloDlg`, która implementuje okno aplikacji. Zanim przejdziemy do ich omówienia chciałbym je nieco rozbudować. Używać będziemy do tego niemal wyłącznie widocznego po utworzeniu edytora wizualnego z podglądem okna oraz okna własności (*Properties*).

Dodawanie kontrolki

Umieśćmy teraz w oknie dodatkowy przycisk o etykiecie *Hello*.

1. Zmniejszamy rozmiar okna w podglądzie widocznym na rysunku 2 (poniższy przepis przeznaczony jest tylko dla zupełnych laików Windows ;-)):
    a. klikamy okno (pojawią się „łapki” w rogach okna i na krawędziach)
    b. łapiąc „łapkę” w prawym dolnym rogu i po prostu przesuwamy tak, aby okno uzyskało pożądany rozmiar.
2. Następnie w panelu *Toolbox* zaznaczamy pozycję `Button`.
3. Następnie klikając na podglądzie okna umieszczamy go w wyznaczonej pozycji.
4. Możemy zmienić rozmiar przycisku w identyczny sposób, w jaki zmienialiśmy rozmiar okna.

Cały „przepis” na okno dialogowe wraz z jego kontrolkami zapisywany jest w plikach zasobów *.rc*. Jego reprezentacją w kodzie C++ jest klasa `CHelloDlg` zapisana w plikach *HelloDlg.h*/*HelloDlg.cpp*. Dodany przed chwilą przycisk nie posiada jednak odpowiednika w jakimkolwiek polu tej klasy. Takie stworzymy dopiero sami nieco później.

Jak otworzyć widok projektowania jeżeli go przypadkiem zamknęliśmy? Należy w podoknie *Solution Explorer* dwukrotnie kliknąć plik zasobów *Hello.rc*. Z lewej strony okna Visual Studio pojawi się podokno *Recource View – Hello* z drzewem zasobów. Należy przejść do gałęzi *Dialog*, a w niej kliknąć dwukrotnie pozycję *IDD_HELLO_DIALOG [Angielski (Stany Zjednoczone)]*.

Wiązanie metody z komunikatem domyślnym kontrolki

Z przyciskiem nie będziemy wiązać osobnej klasy. Z jego zdarzeniami, a konkretnie z jednym zdarzeniem związanym z kliknięciem przycisku, zwiążemy jedynie metodę klasy `CHelloDlg` reprezentującej całe okno. Metodę taką możemy utworzyć na dwa sposoby. Sposób łatwiejszy, to dwukrotne kliknięcie przycisku w widoku projektowania (tj. na zakładce *Hello.rc – IDD_HELLO_DIALOG [Angielski (Stany Zjednoczone)] – Dialog*. Powstanie wówczas metoda o sygnaturze `void CHelloDlg::OnBnClickedButton1`. Skąd taka nazwa metody. `CHelloDlg`, jak już wiemy, to klasa okna dialogowego. Identyfikator przycisku to `IDC_BUTTON1`. Natomiast nazwa zdarzenia to `BN_CLICKED`. Po przerobieniu na styl wielbłądzi otrzymamy `OnBnClickedButton1`, gdzie „On” w nazwie sygnalizuje, że mamy do czynienia z metodą związaną ze zdarzeniem. W ten sposób możemy utworzyć metody związane ze zdarzeniami domyślnymi tj. zazwyczaj z tymi, które najbardziej kojarzą się z daną kontrolką np. kliknięcie z przyciskiem. Drugi sposób wymaga użycia podokna *Properties* i omówiony zostanie nieco niżej.

Nowa metoda została zadeklarowana w sekcji publicznej klasy `CHelloDlg` w pliku *Hello.h* w następujący sposób:

```
afx_msg void OnBnClickedButton1();
```

Modyfikator afx_msg sygnalizuje, że metoda jest związana z jakimś komunikatem. Akronim AFX sygnalizuje, że mamy do czynienia z biblioteką MFC (*Microsoft Foundation Classes*), która we wczesnych fazach rozwoju nazywana była *Application Framework Extensions*. Stąd AFX. Tak naprawdę afx_msg nie jest prawdziwym modyfikatorem, a jedynie pomocą dla narzędzi projektowania klasy (ang. *Class Wizard*) – pomaga mu zidentyfikować deklaracje metod związanych z komunikatami (ang. *message map handler declarations*).

Definicja nowoutworzonej metody znajduje się w pliku *Hello.cpp*. Dodajmy do niej polecenia wyświetlające komunikaty o treści „Hello World!" i „Witaj świecie!". Pokazuje to listing 1.

Listing 1. Metoda zdarzeniowa przycisku z poleceniem wyświetlającym komunikat

```
void CHelloDlg::OnBnClickedButton1()
{
   MessageBox(L"Hello World");
   MessageBox(L"Witaj świecie!");
}
```

Łańcuch "Hello World" jest typu const char[12] (uwzględnia także znak \0 na końcu). Metoda MessageBox zdefiniowana w klasie CDialog, jeżeli korzystamy z bibliotek w wersji Unicode (zob. pkt 3c w przepisie na tworzenie projektu), żąda od nas parametru LPCTSTR tj. wskaźnika do wchar_t. Odpowiedni typ uzyskujemy stawiając literę L przed łańcuchem.

IntelliSense

TO DO: Informacja o podpowiadaniu argumentów i składowych klas i obiektów (uruchamiana po znakach (, ., ->, :: lub przez naciśnięcie *Ctrl+Space* i *Ctrl+Shift+Space*).

Wiązanie komunikatów

Warto zajrzeć do pliku *HelloDlg.cpp* i odnaleźć fragment kodu, w którym metoda OnBnClickedButton1 wiązana jest z komunikatem za pomocą makra ON_BN_CLICKED. Nazywa się to mapowaniem lub wiązaniem komunikatów (ang. *message map*). Pokazuje to poniższy listing:

Listing 2. Sekcja mapowania komunikatów w pliku HelloDlg.cp

```
BEGIN_MESSAGE_MAP(CHelloDlg, CDialog)
   ON_WM_PAINT()
   ON_WM_QUERYDRAGICON()
   //}}AFX_MSG_MAP
   ON_BN_CLICKED(IDC_BUTTON1, &CHelloDlg::OnBnClickedButton1)
   ON_BN_CLICKED(IDOK, &CHelloDlg::OnBnClickedOk)
END_MESSAGE_MAP()
```

Sekcja mapowania otwierana jest przez makro BEGIN_MESSAGE_MAP pobierające jako argumenty klasę, do których należą metody oraz jej klasę bazową. Zwykle tego kodu nie trzeba edytować, robi to za nas VC++, ale warto zdawać sobie sprawę z jego istnienia i roli.

Dodajmy do okna jeszcze jeden przycisk. Jego identyfikatorem będzie IDC_BUTTON2. Aby jego kliknięcie również powodowało uruchomienie istniejącej już metody OnBnClickedButton1 wystarczy do kodu z listingu 2 dodać instrukcję:

```
ON_BN_CLICKED(IDC_BUTTON2, &CHelloDlg::OnBnClickedButton1)
```

W jej efekcie w momencie otrzymania przez kontrolkę identyfikowaną przez IDC_BUTTON2 komunikatu powiadamiającego (ang. *notification message*) BN_CLICKED (właściwie okno otrzymuje komunikat WM_COMMAND, ale nie chcę tego na razie zbyt szczegółowo wyjaśniać) uruchomiona zostanie metoda CHelloDlg::OnBnClickedButton1.

Metoda MessageBox – trochę filozofii MFC

Metoda MessageBox zdefiniowana została w klasie CWnd, bazowej dla klasy CDialog, która z kolei jest bazową dla CHelloDlg, jako opakowanie do funkcją WinAPI o tej samej nazwie. Oryginalna funkcja przyjmuje jednak trzy dodatkowe parametry. Musimy w niej wskazać uchwyt do okna, z którym ma być związane okno komunikatu, a także określić tytuł okna z komunikatem oraz rodzaj przycisków widocznych na oknie komunikatu oraz rodzaj ikony widocznej z lewej strony tekstu. Wszystkie parametry funkcji MessageBox[1], poza uchwytem do okna, mogą być użyte także w przypadku metody – dla wszystkich poza tekstem komunikatu zdefiniowane są wartości domyślne. I tak na przykład możemy użyć okna komunikatu do zadania użytkownikowi pytania, w odpowiedzi na które może kliknąć przycisk *Tak* lub *Nie*. Co więcej możemy także sprawdzić jaką wybrał odpowiedź. Pozwala na to zwracana przez metodę i funkcję wartość odpowiadająca klikniętemu przyciskowi (IDOK, IDYES, IDNO, IDCANCEL itd.). Pokazuje to listing 3.

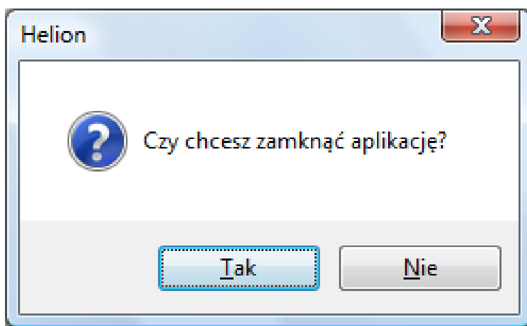
Listing 3. Pełniejsze wykorzystanie możliwości funkcji MessageBox

```
if (MessageBox(L"Czy chcesz zamknąć aplikację?",
               L"Helion",MB_ICONQUESTION | MB_YESNO)==IDYES)

   this->PostMessage(WM_CLOSE);

else this->SetWindowText(L"Zamknięcie anulowane");
```

W bibliotece MFC zdefiniowana jest także funkcja AfxMessageBox przyjmująca argumenty analogiczne, jak metoda CWnd::MessageBox.

W listing 3 pojawiły się kolejne metody klasy CHelloDlg: PostMessage i SetWindowText. Podobnie, jak MessageBox obie zostały zdefiniowane w klasie bazowej CWnd (*Wnd* to oczywiście skrót od *Window* – okno). I podobnie jak MessageBox obie są opakowaniami dla funkcji WinAPI o tych samych nazwach. Pierwsza wysyła komunikat do bieżącego okna (o tym czym są komunikaty zob. w rozdziale ???), druga zmienia tekst widoczny na pasku tytułu okna. Po systematyczny przegląd metod klasy CWnd odsyłam do MSDN: *http://msdn2.microsoft.com/en-us/library/1xb05f0h(VS.80).aspx*.

Jak widać na rysunku 3 komunikat pokazywany instrukcją z listingu 3 zadaje użytkownikowi pytanie o to, czy zamknąć aplikację. Jeżeli ten kliknie przycisk z etykietą *Tak* tj. jeżeli metoda MessageBox zwróci wartość IDYES, wyślemy do okna komunikat WM_CLOSE. W przeciwnym razie na pasku tytułu wyświetlimy informację o anulowaniu zamknięcia (niezbyt to eleganckie, ale przynajmniej demonstruje sposób zmiany tytułu okna). Wysłanie komunikatu WM_CLOSE jest dla okna sygnałem do zamknięcia nieodróżnialnym od sytuacji, w której użytkownik kliknie myszką przycisk z „x" na pasku tytułu lub naciśnie kombinację klawiszy *Alt+F4*. Zamknięcie okna dialogowego spowoduje automatyczne zakończenie działania metody InitInstance obiektu aplikacji CHelloApp, a tym samym zakończenie jej procesu. Użycie do tego celu metod klasy CHelloDlg np. narzucającej się CloseWindow nie przyniesie spodziewanego efektu. CloseWindow, wbrew nazwie, jedynie zminimalizuje okno, a nie usunie a go z pamięci.



Rysunek 3. Komunikat wyświetlany metodą CWnd::MessageBox z listingu 3.

---

[1] Ściśle rzecz ujmując są dwie wersje funkcji MessageBox: MessageBoxA i MessageBoxW. Pierwsza oczekuje tekstu komunikatów zbudowanego ze znaków ASCII, druga akceptuje wielobajtowe znaki unicode. Jeżeli chcemy wywołać funkcję WinAPI, a nie metodę klasy CWnd należy poprzedzić nazwę funkcji nazwą przestrzeni nazw (może być domyślna). Niezbędny w tej funkcji uchwyt można pobrać z pola obiektu okna o nazwie m_hWnd:

```
::MessageBox(this->m_hWnd,L"Hello World!",L"Helion",MB_ICONINFORMATION | MB_OK);
```

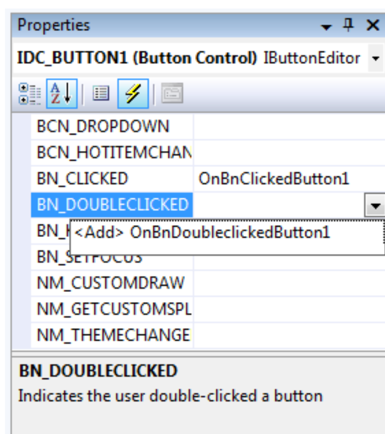
Problem zamknięcia okna i tym samym zakończenia działania aplikacji można rozwiązać jeszcze prościej, choć pewnie nieco mniej elegancko. Ponieważ utworzyliśmy projekt na bazie okna dialogowego (klasa `CHelloDlg` dziedzicząca ze zwykłej klasy okna dialogowego `CDialog` z MFC), to dysponuje ono metodą `OnOK`. Zasadniczym zadaniem tej metody jest zamknięcie okna dialogowego i zwrócenie wartości `IDOK`. Wartość ta jest przekazywana do metody `CHelloDlg::DoModal`, którą tworzone jest okno w metodzie `CHelloApp::InitInstance` (klasę `CHelloApp` znajdziemy w plikach *Hello.h*/*Hello.cpp*). Skoro zwracana wartość nie jest dla nas w tym projekcie interesująca, to możemy bez wyrzutów sumienia wykorzystać `OnOK` do zamknięcia okna.

W WinAPI stosowana jest notacja węgierska. W skrócie oznacza to poprzedzanie nazw typów i zmiennych prefixami określającymi typ (np. *b* dla `bool`, *h* dla uchwytu). Byłoby to szczególnie przydatne, gdybyśmy do programowania stosowali języka C. W C++, w którym kontrola typów jest ściślejsza ta dodatkowa kontrola nie jest niezbędna. W MFC stosuje się jednak inną (niezależną od węgierskiej) konwencję, w której wszystkie nazwy klas rozpoczynają się od *C*, natomiast nazwy pól od *m_*.

Okno Properties: własności i zdarzenia

Jeżeli chcemy utworzyć metodę, która nie jest związana ze zdarzeniem domyślnym należy użyć podokna *Properties*. W tym celu:

1. Zaznaczamy kontrolkę – najlepiej drugi dodany przez nas przycisk.
2. W oknie *Properties* zmieniamy zakładkę na *Control Events* (ikona żółtej błyskawicy).
3. Zaznaczamy pozycję odpowiadającą zdarzeniu `BN_DOUBLECLICKED`.
4. Rozwijamy listę w polu z prawej strony nazwy zdarzenia (rysunek 4). Wybieramy pozycję z *<Add>* na początku, co spowoduje dodanie nowej metody o nazwie `OnBnDoubleclickedButton1` do klasy `CHelloDlg` i związanie jej z odpowiednim komunikatem.



Rysunek 4. Dodawanie metody zdarzeniowej do kontrolki w zasobach aplikacji

Niestety obsługa podwójnego kliknięcia w przypadku przycisku to nie jest dobry pomysł. Już pierwsze kliknięcie z dwóch spowoduje wyświetlenie okna z komunikatem „Hello World!". Dlatego bez zwłoki usuńmy to wiązanie i samą metodę. Możemy to prosto zrobić wybierając z rozwijanej listy przy `BN_DOUBLECLICKED` pozycję *<Delete> OnBnDoubleclickedButton1*. Jest tam również pozycja *<Edit code>*, która przenosi do metody i pozwala na jej edycję.

Wszystkie zmiany w kodzie, także te wprowadzane za pomocą kreatorów i za pomocą narzędzi projektowania wizualnego można odwoływać kombinacją klawiszy *Ctrl+Z*.

Okno *Properties* służy przede wszystkim do konfigurowania przycisku. Zmieńmy dla przykładu etykietę przycisku:

1. Zaznaczmy odpowiednią kontrolkę.
2. Zmieńmy zakładkę w oknie *Properties* na *Properties* (ikona z czymś podobnym do szarej tabelki).

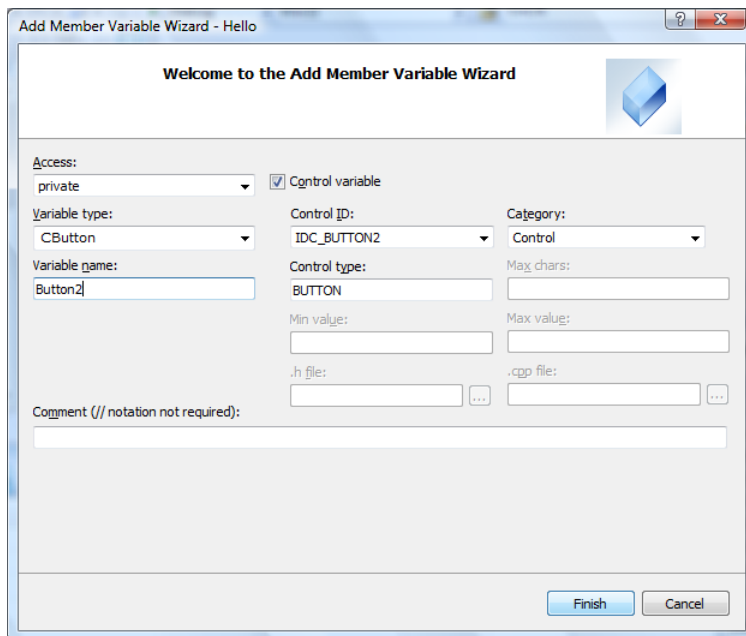
3. W polu przy nazwie własności *Caption* wpiszmy dowolny tekst, chociażby *Hello World!*.

Z pewnością warto poeksperymentować z pozostałymi własnościami. Odpowiadają one za wygląd i funkcjonowanie przycisku w aplikacji.

Wiązanie zmiennej z kontrolką

Do tej pory użyliśmy kontrolki umieszczonej w zasobach, która na dobrą sprawę nie miała swojej reprezentacji w kodzie C++ (ściślej w klasie `CHelloDlg`). Jeżeli chcemy mieć możliwość programowej zmiany własności kontrolki, warto związać z nią zmienną (ściślej zmienną składową[2] klasy `CHelloDlg`). Kontrolki reprezentowane są przez pola, których typem są odpowiadające kontrolkom klasy MFC. W przypadku przycisku będzie to klasa `CButton`. Na szczęście wiązania nie musimy robić ręcznie. Służy do tego narzędzie o nazwie *Class Wizard*, a dokładniej jeden z jego kreatorów:

1. W widoku projektowania zaznaczmy przycisk na podglądzie okna (np. ten z identyfikatorem `IDC_BUTTON2`).
2. Z menu kontekstowego rozwijanego prawym klawiszem myszy wybierzmy polecenie *Add Variable...*.
3. Pojawi się okno kreatora dodawania zmiennej składowej (rysunek 5):
   a. zmieniamy w nim zakres dostępności nowej zmiennej na *private*,
   b. nadajemy zmiennej nazwę np. `Button2` (pole *Variable name*),
   c. klikamy *Finish*.



Rysunek 5. Kreator zmiennej składowej

W efekcie w klasie `CHelloDlg` pojawi się nowa zmienna składowa (lub jak kto woli pole) o nazwie `Button2` typu `CButton` (plik *HelloDlg.h*). Jej deklaracja w pliku nagłówkowym nie jest jednak tym, co najciekawsze. Ciekawsza zmiana w kodzie pojawiła się bowiem w pliku *HelloDlg.cpp*. W metodzie `DoDataExchange` pojawiła się nowa instrukcja:

```
DDX_Control(pDX, IDC_BUTTON2, Button2);
```

Odpowiada ona za wymianę danych między nową kontrolką z zasobów aplikacji, a klasą reprezentującą okno dialogowe w kodzie C++. Dzięki mechanizmowi DDX (ang. *Dialog Data Exchange*) odpowiada za zgodność wartości wyświetlanej w kontrolce MFC widocznej w oknie i stanu związanej z nią zmiennej. Powyższe polecenie wiąże zatem kontrolkę o identyfikatorze `IDC_BUTTON2` ze zmienną `Button2`. Możemy dzięki temu

[2] W całym tekście będę zamiennie stosowane pojęcia pola (ang. *field*) lub zmiennej składowej czy danej składowej (ang. *data member*).

utożsamiać oba twory. Oprócz DDX istnieje jeszcze mechanizm DDV (ang. *Dialog Data Validation*), który może przejąć kontrolę nad danymi wprowadzanymi do kontrolek. Możemy na przykład ograniczyć maksymalną liczbę znaków wprowadzanych do pola edycyjnego itp.

Teraz możemy odwoływać się do przycisku korzystając z pola `Button2`. Ułatwia to wywoływanie metod lub zmianę własności tej kontrolki np.

```
Button2.SetWindowText(L"Helion");
```

lub np.

```
Button2.SetButtonStyle(BS_CHECKBOX,TRUE);
Button2.SetCheck(TRUE);
```

Usuwanie zbędnych kontrolek

Poza dodanymi przez nas przyciskami na oknie znajdują się umieszczone przez kreator aplikacji dwa przyciski z etykietami *OK* i *Cancel* oraz kontrolka opisana w *Toolbox* jako *Static Text*. Wszystkie je możemy po prostu zaznaczyć w widoku projektowania i usunąć naciskając klawisz *Del*. Nie są z nimi związane ani żadne zmienne, ani metody zdarzeniowe. Jeżeli po skasowaniu rozmyślimy się i uznamy, że potrzebujemy obecności przycisku funkcjonującego jak przycisk *OK*, to po umieszczeniu przycisku z *Toolboxa* na podglądzie okna zmieńmy jego identyfikator na `IDOK` (pozycja *ID* w podoknie *Properties*). Wówczas „automatycznie" wywoływać on będzie metodę `OnOK`.

Analiza kodu aplikacji

Osoba znająca C++ nie powinna mieć problemu ze zrozumieniem kodu aplikacji *Hello*. W plikach *Hello.h*/*Hello.cpp* zdefiniowana jest klasa `CHelloApp` oraz jej instancja `theApp`. Klasa ta rozszerza klasę MFC `CWinApp` i wyposażona została w nadpisany domyślny konstruktor (jednak bez żadnej linii kodu) oraz metodę `InitInstance`. To w tej drugiej metodzie należy umieszczać wszystkie instrukcje inicjujące obiekt aplikacji, szczególnie jeżeli związane są one z kontrolkami MFC. W metodzie `CHelloApp::InitInstance` tworzona jest instancja klasa `CHelloDlg` i wywoływana jest metoda `DoModal`. To powoduje utworzenie okna dialogowego i wstrzymanie wykonywania metody `CHelloApp::InitInstance` aż do zamknięcia okna dialogowego. Klasa `CHelloDlg` zdefiniowana jest w plikach *HelloDlg.h*/*HelloDlg.cpp*. Jej zawartość kontrolowana jest przez wizualny edytor pozwalający na projektowanie okna, podokno *Properties* pozwalające na dodawanie metod związanych z komunikatami oraz *Class Wizard*, którym dodajemy zmienne związane z kontrolkami.

W kilku miejscach kodu znajduje się dyrektywa preprocesora dołączająca pliki nagłówkowe *afxwin.h* oraz inne nagłówki rozpoczynające się od *afx*. To są pliki nagłówkowe biblioteki MFC.

Nigdzie nie znajdziemy jednak funkcji obowiązkowych dla aplikacji Win32, a więc `WinMain` i `WindowProc`. Funkcja `WinMain` to główne wejście do aplikacji (ang. *entry point*), a więc funkcja, która jest wywoływana przez system w momencie uruchamiania aplikacji – odpowiada funkcji `main` znanej z aplikacji konsolowych C++. Jej zakończenie oznacza jednoczesne zakończenie programu. Z kolei `WindowProc` wywoływana jest już w trakcie działania aplikacji za każdym razem, gdy otrzymuje ona komunikat. Jak to zatem jest możliwe, że ich nie ma w kodzie naszej aplikacji? Obie funkcje zaszyte są gdzieś w plikach MFC. W zasadzie klasa `CWinApp` to jedno duże opakowanie dla `WinMain`, a jej metoda `InitInstance` pełni rolę nowego punktu wejścia do aplikacji. Jak to możliwe? Prawdziwa funkcja `WinMain` wywołuje jedynie funkcję `AfxWinMain` zdefiniowaną w bibliotece MFC. Ta natomiast wywołuje po kolei wirtualne metody instancji klasy `CWinApp`, a więc `InitApplication`, `InitInstance`, a wreszcie `Run`[3]. W naszej aplikacji tylko ta druga nadpisana jest w klasie `CHelloApp` i pełni rolę nowego punktu wejścia. Z kolei metoda `Run` zawiera pętle komunikatów (ang. *messager loop*), która przerywana jest dopiero w momencie otrzymania komunikatu `WM_QUIT`. Wówczas wywoływana jest kolejna z funkcji `CWinApp`, a mianowicie `ExitInstance`. Zwracana przez nią wartość przekazywana jest do `AfxWinMain` i dalej do `WinMain`. Z kolei `WindowProc`, której wersja MFC nosi nazwę `AxfWndProc`, zajmuje się jedynie przekazywaniem komunikatów do metod `WndProc` klas okien (pochodnych względem `CWnd` np.

[3] Klasa `CWinApp` lub jej klasa pochodna może mieć tylko jedną instancję w projekcie MFC.

CHelloDlg). Tam obsługiwane są komunikaty WM_NOTIFY, WM_COMMAND i wszystkie pozostałe. Użytkownik zwolniony jest z przygotowywania instrukcji switch charakterystycznej dla typowej metody WndProc, a w zamian definiuje mapę komunikatów, w której za pomocą makr wiążemy metody z poszczególnymi rodzajami komunikatów. To już jednak dobrze wiemy.

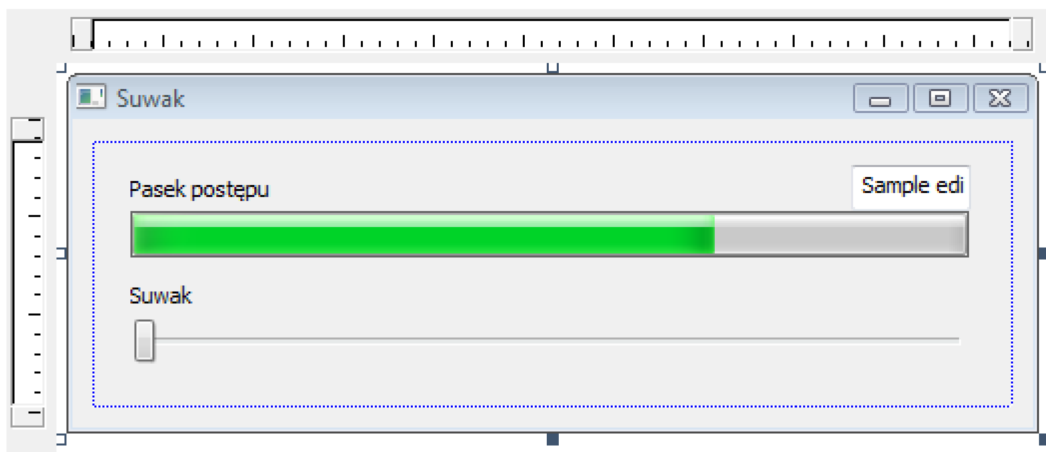
Więcej informacji można znaleźć w artykułach dostępnych w portalu The Code Project: *http://www.codeproject.com/KB/cpp/mfc_architecture.aspx*, *http://www.codeproject.com/KB/cpp/mfc_architecture2.aspx* i *http://www.codeproject.com/KB/cpp/mfcprogflow.aspx*.

Więcej kontrolek

Stwórzmy nowy projekt aplikacji, w której na oknie utworzonym ponownie na bazie okna dialogowego umieścimy suwak (*Slider Control*) oraz pasek postępu (*Progress Control*). Pozycja paska postępu powinna odpowiadać pozycji suwaka. Ta ostatnia może być kontrolowana przez użytkownika. Do obu kontrolek należy dodać opisy (*Static Text*). Ponadto wartość procentową paska postępu należy wyświetlić w postaci liczby w polu edycyjnym (*Edit Control*).

Powyższe zadanie proponuję rozwiązać w następujący sposób:

1. Tworzymy projekt postępując zgodnie z instrukcjami z paragrafu *Tworzenie projektu* zmieniając jedynie nazwę na *Suwak*. Powstaną klasy CSuwakApp i CSuwakDlg.
2. Usuwamy domyślnie umieszczone w oknie dwa przyciski i etykietę.
3. Na formie umieszczamy kontrolki *Slider Control* i *Progress Control*. Nad każdą z nich – etykietę w postaci kontrolki *Static Text*. Całość uzupełniamy polem edycyjnym *Edit Control* (rysunek 6).



Rysunek 6. Projekt okna aplikacji Suwak

4. Za pomocą okna *Properties*:
   a. Zmieniamy zawartość etykiet (własność *Caption*) zgodnie ze wzorem na rysunku 6.
   b. Przełączamy również własność *Read Only* pola edycyjnego na *True*.
   c. Przełączamy własności *Auto Ticks* i *Tick Marks* na *True*.
5. Tworzymy zmienne dla suwaka, paska postępu i pola edycyjnego. Proponuję nadać im nazwy odpowiednio Slider1, Progress1 i Edit1.
6. Korzystając z tych zmiennych inicjujemy własności kontrolek niedostępne w oknie *Properties*. W tym celu do metody OnInitDialog klasy CSuwakDlg dodajemy polecenia wyróżnione w poniższym listingu:

Listing 4. Metoda odpowiedzialna za inicjowanie okna dialogowego i umieszczonych na nim kontrolek

```
BOOL CSuwakDlg::OnInitDialog()
{
```

```
    CDialog::OnInitDialog();

    // Set the icon for this dialog. The framework does this automatically
    // when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);        // Set big icon
    SetIcon(m_hIcon, FALSE);       // Set small icon

    // TODO: Add extra initialization here
    Progress1.SetRange(0,255);
    Progress1.SetPos(0);
    Slider1.SetRangeMin(0);
    Slider1.SetRangeMax(255);
    Slider1.SetTicFreq(15);
    Slider1.SetPos(0);
    Edit1.SetWindowText(L"0%");

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

7. Tworzymy metodę uruchamianą po zmianie pozycji suwaka:
   a. Zaznaczamy suwak (`IDC_SLIDER1`).
   b. W podoknie *Properties* zmieniamy zakładkę na *Events*.
   c. Z rozwijanej listy przy pozycji `TRBN_THUMPOSCHANGING` wybieramy polecenie *<Add>...* tworząc metodę `OnTRBNThumbPosChangingSlider1` w pliku *SuwakDlg.cpp*.
   d. Na końcu metody dodajemy instrukcję zmieniającą pozycję paska postępu:
      `Progress1.SetPos(Slider1.GetPos());`
   e. Dodajmy również polecenia konwertującą pozycję paska postępu na łańcuch zawierający procentową pozycję i umieszczamy go w polu edycyjnym:

Listing 5. Definicja metody związanej ze zmianą pozycji suwaka

```
void CSuwakDlg::OnTRBNThumbPosChangingSlider1(NMHDR *pNMHDR, LRESULT *pResult)
{
    // This feature requires Windows Vista or greater.
    // The symbol _WIN32_WINNT must be >= 0x0600.
    NMTRBTHUMBPOSCHANGING *pNMTPC = reinterpret_cast<NMTRBTHUMBPOSCHANGING *>(pNMHDR);
    // TODO: Add your control notification handler code here
    *pResult = 0;

    Progress1.SetPos(Slider1.GetPos());

    wchar_t bufor[10]=L"";
    int minimum,maximum;
    Progress1.GetRange(minimum,maximum);
    int procent=100*(Progress1.GetPos()-minimum)/(maximum-minimum);
    _itow_s(procent,bufor,10,10);
    wcscat_s(bufor,10,L"%");
    Edit1.SetWindowText(bufor);
}
```

8. Aby powyższa metoda była rzeczywiście wywoływana przy zmianie pozycji suwaka należy w oknie *Properties* zmienić jeszcze własność *Notify Before Move* na *True*!

Rezultat, jakiego się spodziewamy po uruchomieniu aplikacji, to oczywiście zmiana pozycji paska postępu przy zmianie myszką pozycji suwaka. I taki efekt zobaczymy o ile… dysponujemy Windows Vista (lub nowszym). Tylko w Windows Vista przesyłany jest bowiem komunikat `TRBN_THUMPOSCHANGING`, który wykorzystaliśmy. W starszych wersjach Windows należy użyć komunikatu `NM_CUSTOMDRAW`. Zatem jeżeli dysponujemy starszym systemem, kod z listingu 5 należy umieścić w metodzie związanej z tym ostatnim komunikatem lub wywołać niej metodę `OnTRBNThumbPosChangingSlider1`.

Uwaga! Komunikat `TRBN_THUMPOSCHANGING` wykorzystany w punkcie 7 działa tylko w systemi Windows Vista.

Należy jednak pamiętać, że użycie komunikatu `NM_CUSTOMDRAW` zadziała również w Viście, a więc należy zabezpieczyć się, żeby metoda `OnTRBNThumbPosChangingSlider1` wywoływana była tylko w starszych niż Vista wersjach Windows (listing 6). W ten sposób unikniemy dublowania metod.

Listing 6. W Windows XP i starszych metoda poniższa wywoła metodę `OnTRBNThumbPosChangingSlider1`. W Windows Vista nie zrobi w zasadzie niczego.

```
void CSuwakDlg::OnNMCustomdrawSlider1(NMHDR *pNMHDR, LRESULT *pResult)
{
   LPNMCUSTOMDRAW pNMCD = reinterpret_cast<LPNMCUSTOMDRAW>(pNMHDR);
   // TODO: Add your control notification handler code here
   *pResult = 0;

   OSVERSIONINFO wersja;
   ZeroMemory(&wersja, sizeof(OSVERSIONINFO));
   wersja.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
   GetVersionEx(&wersja);
   if (wersja.dwMajorVersion<6)
   {
      OnTRBNThumbPosChangingSlider1(pNMHDR,pResult);
   }
}
```

Kolory

Projekt można nieco uatrakcyjnić, choć i w tym przypadku nie obędzie się bez problemów. Jeżeli do obecnego suwaka dodamy jeszcze dwa, zwiążemy z nimi zmienne `Slider2` i `Slider3`, uzupełnimy metodę `OnInitDialog` poleceniami konfigurującymi nowe suwaki analogicznie, jak wcześniejszy, a następnie zmienimy kod metody `OnTRBNThumbPosChangingSlider1` na ten widoczny w listingu 7, to kolor paska postępu będziemy kontrolować suwakami pozwalającymi na zmiany składowych R, G i B.

Listing 7. Wyróżnione zostały zmiany względem kodu z listingu 5

```
void CSuwakDlg::OnTRBNThumbPosChangingSlider1(NMHDR *pNMHDR, LRESULT *pResult)
{
   // This feature requires Windows Vista or greater.
   // The symbol _WIN32_WINNT must be >= 0x0600.
   NMTRBTHUMBPOSCHANGING *pNMTPC = reinterpret_cast<NMTRBTHUMBPOSCHANGING *>(pNMHDR);
   // TODO: Add your control notification handler code here
   *pResult = 0;
```

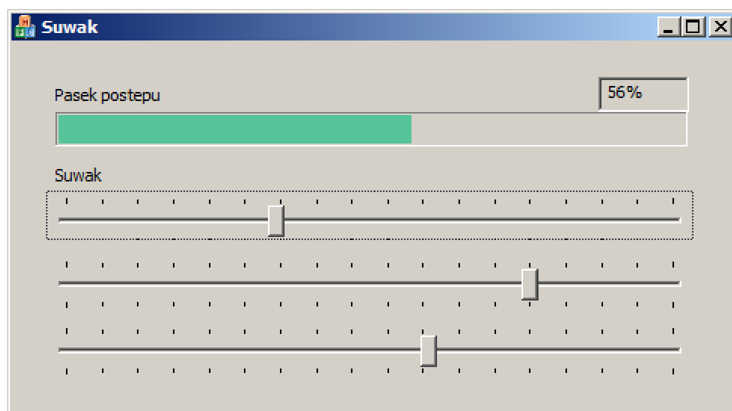
```
        int jasnosc=(Slider1.GetPos()+Slider2.GetPos()+Slider3.GetPos())/3;
        Progress1.SetPos(jasnosc);

        wchar_t bufor[10]=L"";
        int minimum,maximum;
        Progress1.GetRange(minimum,maximum);
        int procent=100*(Progress1.GetPos()-minimum)/(maximum-minimum);
        _itow_s(procent,bufor,10,10);
        wcscat_s(bufor,10,L"%");
        Edit1.SetWindowText(bufor);

        Progress1.SetBarColor(RGB(Slider1.GetPos(),Slider2.GetPos(),Slider3.GetPos()));
    }
```

Teraz wystarczy związać nowe suwaki z powyższą metodą dodając wywołania makr w sekcji mapowania (listing 8). Oczywiście należy także zmienić ich własności *Notify Before Move* na *True*.

Listing 8. Wiązanie istniejących metod z komunikatami adresowanymi do nowych kontrolek

```
    BEGIN_MESSAGE_MAP(CSuwakDlg, CDialog)
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
        //}}AFX_MSG_MAP
        ON_NOTIFY(TRBN_THUMBPOSCHANGING, IDC_SLIDER1,
    &CSuwakDlg::OnTRBNThumbPosChangingSlider1)
        ON_NOTIFY(TRBN_THUMBPOSCHANGING, IDC_SLIDER2,
    &CSuwakDlg::OnTRBNThumbPosChangingSlider1)
        ON_NOTIFY(TRBN_THUMBPOSCHANGING, IDC_SLIDER3,
    &CSuwakDlg::OnTRBNThumbPosChangingSlider1)
        ON_NOTIFY(NM_CUSTOMDRAW, IDC_SLIDER1, &CSuwakDlg::OnNMCustomdrawSlider1)
        ON_NOTIFY(NM_CUSTOMDRAW, IDC_SLIDER2, &CSuwakDlg::OnNMCustomdrawSlider1)
        ON_NOTIFY(NM_CUSTOMDRAW, IDC_SLIDER3, &CSuwakDlg::OnNMCustomdrawSlider1)
    END_MESSAGE_MAP()
```

Teraz możemy uruchomić aplikację i spróbować zmienić kolor paska postępu. Zawiodą się jednak wszyscy Ci, którzy używają „stylu wizualnego" bowiem zmiana koloru paska postępu widoczna jest jedynie w kompozycji „klasyczna Windows" (rysunek 7). Nie jest to zatem zbyt praktyczne rozwiązanie.



Rysunek 7. Tylko w widoku klasycznym kolor suwaka może być dowolnie zmieniany

Użycie kontrolki ActiveX

Zamiast zmieniać kolor suwaka możemy jednak wybrać inną kontrolkę. Niestety żadna z kontrolek MFC oferowanych w *Toolbox* nie nadaje się do tego celu bez konieczności obsługi komunikatu `WM_CTLCOLOR`[4]. Na szczęście istnieje możliwość użycia kontrolek ActiveX zarejestrowanych w systemie Windows.

1. Przejdźmy do widoku projektowania okna dialogowego.
2. Prawym klawiszem myszy rozwińmy menu kontekstowe.
3. Wybierzmy z niego polecenie *Insert ActiveX Control…*.
4. Z listy widocznej w oknie dialogowym wybierzmy *Microsoft Forms 2.0 Image* i kliknijmy *OK*.
5. Następnie przesuńmy nową kontrolkę w miejsce, które chcemy, aby zajmowała w oknie.
6. Zwiążmy z nią zmienną o nazwie `Image1`. Zwróćmy uwagę, że do projektu dodane zostaną pliki *image1.h*/*image1.cpp* zawierające definicję nowej klasy `CImage1` dziedziczącej wprost z klasy `CWnd`. Zmienna składowa – instancja tej klasy dodana do klasy `CSuwakDlg` nazywa się `Image1`.
7. Zaznaczmy nową kontrolkę w widoku projektowania i zmieńmy jej własność `BackColor` na *Black* (zakładka *Web* edytora kolorów).
8. Wreszcie zmodyfikujmy metodę zdarzeniową, aby zmieniać kolor nowej kontrolki (listing 9).

Listing 9. Zmiany w metodzie względem wersji z listingu 7

```
void CSuwakDlg::OnTRBNThumbPosChangingSlider1(NMHDR *pNMHDR, LRESULT *pResult)
{
   ...

   int kolor=RGB(Slider1.GetPos(),Slider2.GetPos(),Slider3.GetPos());
   Progress1.SetBarColor(kolor);
   Image1.put_BackColor(kolor);
}
```

Jeżeli znudziło nam się przesuwanie suwaków możemy w nowej kontrolce pokazać obraz z pliku. To jest w końcu zasadnicze zastosowanie tej kontroli. Wystarczy użyć edytora własności `Picture`, w którym wskazujemy plik BMP, JPEG, GIF, PNG lub ICO. Jeżeli rozmiar obrazu nie pasuje do rozmiaru kontrolki możemy zmienić `PictureSizeMode` na `Stretch`, aby go ścisnąć lub rozciągnąć. Kontrolka ActiveX posiada również zdarzenia, używa się ich identycznie, jak zdarzenia kontrolek MFC. Nasz obraz będzie jednak raczej „biernym” elementem okna.

[4] Zob. *http://msdn2.microsoft.com/en-us/library/6skhh669(VS.80).aspx*