Jacek Matulewski
http://www.phys.uni.torun.pl/~jacek/

# Microsoft Visual Basic

## Projektowanie RAD w VB
## Współpraca VB i Microsoft Office

# **Ćwiczenia**

Toruń, 6 września 2002

Najnowsza wersja tego dokumentu znajduje się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad3.pdf

Źródła opisanych w tym dokumencie programów znajdują się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad3.zip

# I. Spis treści


I. Spis treści ........ 2
II. Od Delphi do Visual Basica ........ 3
  1. Podobieństwa „OCX RAD” i „VCL RAD” ........ 3
  2. Image i CommonDialog (ShowOpen) ........ 3
  3. Notatnik (TextBox, CommonDialog.ShowFont, obsługa plików, schowka) ........ 4
  4. Dynamiczne tworzenie obiektów ........ 8
  5. Odtwarzacz plików audio ........ 9
  6. Odtwarzacz plików wideo ........ 9
  7. Prosta aplikacja bazodanowa ........ 10
  8. Łączenie z bazą danych za pomocą ADO (OLE DB), wykresy ........ 11
  9. Pliki projektu: ........ 11
  10. Słówko na temat VBScript ........ 11
III. Gdzie szukać pomocy? ........ 13
IV. Współpraca Visual Basica i Microsoft Office? ........ 14
  1. Osadzanie obiektu OLE2 w aplikacji (przy okazji Menu Editor) ........ 14
  2. Automation - wykorzystanie obiektów Microsoft Excel 8.0/97 ........ 16
    a) Współpraca aplikacji z uruchomioną aplikacją Excela ........ 17
    b) Edycja komórek Excela z poziomu aplikacji ........ 18
    c) Wykorzystanie funkcji Excela do wykonania obliczeń ........ 21
  3. Automation - wykorzystanie obiektów Microsoft Word 97 ........ 25
  4. Visual Basic vs. Visual Basic for Applications ........ 27
  5. Kalkulator – makro VBA osadzone w Excelu ........ 28

# II. Od Delphi do Visual Basica

Celem tego rozdziału nie jest gruntowne uczenie Visual Basica. Nie jest nim również nauka programowania RAD. Cel jest znacznie skromniejszy, a mianowicie uświadomienie programistom znającym Borland C++ Buildera lub Delphi, że umieją także pisać aplikację za pomocą Microsoft Visual Basica (o ile znają BASIC, ale to nie jest też warunek konieczny, tak jak nie jest konieczna gruntowna znajomość C++ do programowania RAD w Builderze).

Dokument składa się w istocie z przykładów, których pomysły są zaczerpnięte z *Podstaw programowania RAD* (http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad1.pdf).

## 1. Podobieństwa „OCX RAD" i „VCL RAD"

Po uruchomieniu Visual Basica mamy możliwość wyboru typu aplikacji, którą chcemy tworzyć – początkowo będzie to zawsze „Standard EXE".

**Uwaga!** W przeciwieństwie do Delphi nie wystarczy wybrać kontrolkę i kliknąć na formę. Aby skutecznie umieścić ją na formie trzeba przeciągnąć myszką zaznaczając obszar, który ma zajmować. Można również kliknąć dwukrotnie kontrolkę na panelu.

**Uwaga!** Przełączanie między projektem formy i edytorem kodu możliwe jest w Visual Basicu z poziomu klawiatury za pomocą (F7 i Shift+F7).

**Label:** Dodać do projektu formy kontrolkę Label1 i edytować jego własności (Caption, Caption z accel char np. &Zamknij, ForeColor, Font, Visible). Zmiana własności Label1 podczas działania programu.
**CommandButton:** Analogicznie umieścić na formie Command1 i edytować własności oraz metodę zdarzeniową związaną ze zdarzeniem Click. Napisać procedurę zdarzeniową zamykającą aplikację (przez zamknięcie formy głównej poleceniem `UnLoad Me` – Me jest odpowiednikiem `this` w C++ i `Self` w Delphi, czyli jest wskaźnikiem do bieżącego obiektu). Z kolei UnLoad jest odpowiednikiem free w C++.

## 2. Image i CommonDialog (ShowOpen)

W Visual Basicu wszystkie okna dialogowe schowane są w kontrolce CommonDialog, która standardowo nie jest udostępniona w Projekcie. Typ okna zależy od metody, którą wywołamy do pokazania okna (ShowOpen, ShowSave, ShowColor, ShowFont itp.) Aby udostępnić ten komponent w bieżącym projekcie należy w **Project, Components ...** zaznaczyć pozycję **Microsoft Common Dialog Control 6.0**, a na panelu kontrolek pojawi się ikona CommonDialog. Komponent ten jest interfacem do biblioteki commdlg.dll znajdującej się w katalogu systemowym Windows.

W przeciwieństwie do Delphi/C++ Buildera LoadPicture nie jest własnością Image1.Picture, a jest „ogólną" funkcją Visual Basica. Na formie umieszczamy Image, CommonDialog i CommandButton. Naszym celem jest napisanie prostej aplikacji, która umożliwi użytkownikowi wybór nazwy obrazka po naciśnięciu klawisza i załadowanie go do Image1.Picture. W przedstawionej poniżej procedurze zdarzeniowej (związanej z Command1.Click) ignoruję możliwość sytuacji, w której użytkownik naciska Anuluj w oknie dialogowym (opis obsługi takiego zdarzenia znajduje się w następnym przykładzie).

```
Private Sub Command1_Click()
    CommonDialog1.ShowOpen 'Wywolanie okna dialogowego
    Image1.Picture = LoadPicture(CommonDialog1.FileName) 'Wczytanie obrazu
End Sub
```

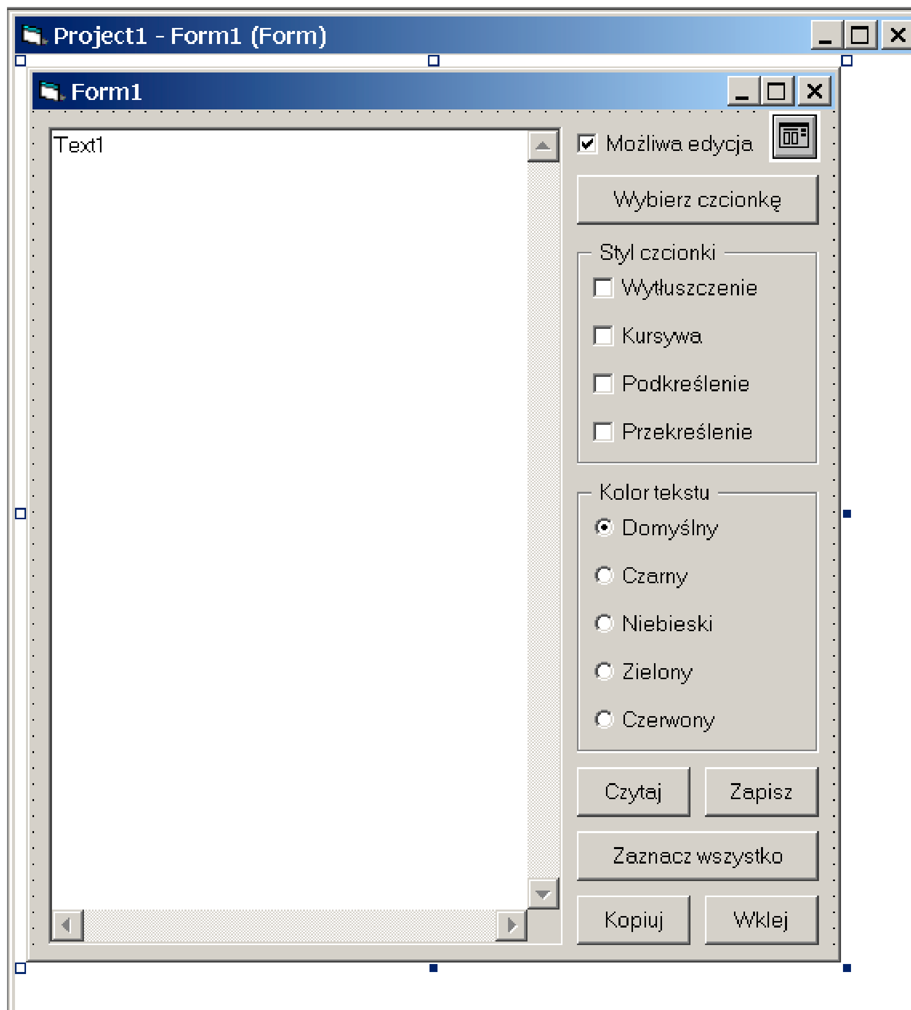
W przeciwieństwie do Delphi/C++ Buildera Visual Basic obługuje pliki JPEG i GIF.

## 3. **Notatnik** (TextBox, CommonDialog.ShowFont, obsługa plików, schowka)

Stworzymy aplikację pełniącą formę notesu, w którym (w bardzo ograniczonym zakresie) będziemy mogli modyfikować sposób wyświetlania tekstu na ekranie. Wśród standardowo dostępnych kontrolkach / komponentach odpowiednikiem Borlandowskiego TMemo jest Text (ikona w drugim wierszu). Dokładniej rzecz biorąc Text może pełnić funkcję zarówno TEdit jak i TMemo w zależności od ustawienia własności **MultiLine**. W naszym przypadku należy umieścić ją na formie nadając odpowiednią wielkość. Następnie do formy dodajmy przyciski, checkboxy i radioboxy podobnie jak na rysunku.

Komponentem, który jest nam potrzebny, ale domyślnie nie jest udostępniony, jest okno dialogowe wyboru czcionek. Kryje się ono w kontrolce **CommonDialog**, której metoda pozwala **ShowFont** wywołać potrzebne nam okno.

W Text1 za pomocą okienka Properties ustalono opcję **ScrollBars** na rtBoth.
Zaznaczono też domyślny kolor czcionki **OptionButton1.Value** ustawiając na True.




Następnie będziemy kolejno oprogramowywali widoczne na formie komponenty kontroli.

**a) „Możliwa edycja"**
Ten CheckBox ma kontrolować możliwość zmieniania przez użytkownika zawartości Text1.

```
Private Sub Check1_Click()
    'w instrukcji przypisania pominieta jest opcjonalna instrukcja Let
    Text1.Enabled = Check1.Value
End Sub
```

Check1.Value nie jest zmienną logiczną – przyjmuje trzy możliwe wartości: 0 (vbUnchecked), 1 (vbChecked) i 2 (vbGrayed), ale podobnie jak w C++ konwersja z liczb naturalnych do wartości logicznej jest automatyczna.

**b) „Wybierz czcionkę"**
Tym przyciskiem uruchomimy standardowe okno wyboru czcionki i jej stylu oraz zastosujemy nowo wybraną czcionkę do sformatowania tekstu w oknie Text1 (podobnie jak w Delphi/C++ Builderze, aby stworzyć szkielet domyślnej procedury reagującej na kliknięcie przycisku – wystarczy dwukrotnie kliknąć na niego w projekcie. Ponadto można wybrać obsługiwane zdarzenie w rozwijalnej liście na górze okna edycyjnego).

```
Private Sub Command1_Click()
    'Obsluga nacisniecia Cancel - przejscie do konca procedury
    CommonDialog1.CancelError = True
    On Error GoTo NacisnietoCancel

    'nalezy ustalic jakie typy czcionki maja byc widoczne dla okna
    'dialogowego (cdlCFScreenFonts lub cdlCFPrinterFonts lub jak tu obie)
    'inaczej pojawi sie komunikat bledu, informujący ze nie ma
    'zainstalowanych czcionek;
    'udostepniamy rowniez dodatkowe formatowania: cdlCFEffects
    CommonDialog1.Flags = cdlCFBoth Or cdlCFEffects
    CommonDialog1.ShowFont

    'przypisanie wybranej czcionki do Text1
    'wykorzystano konstrukcje With analogicznie jak w Delphi
    With Text1
        .Font.Bold = CommonDialog1.FontBold
        .Font.Name = CommonDialog1.FontName
        .Font.Size = CommonDialog1.FontSize
        .Font.Bold = CommonDialog1.FontBold
        .Font.Italic = CommonDialog1.FontItalic
        .Font.Underline = CommonDialog1.FontUnderline
        .Font.Strikethrough = CommonDialog1.FontStrikethru
        .ForeColor = CommonDialog1.Color
    End With

    'Wyjscie z procedury - zapobiega wykonaniu sekcji NacisnietoCancel
    Exit Sub

'Czesc obslugujaca blad - nic nie robi
NacisnietoCancel:

End Sub
```

Aby sprawdzić czy użytkownik po wybraniu czcionki nacisnął OK czy Cancel należy włączyć własność **CancelError**, która powoduje, ze naciśnięcie Cancel w oknie dialogowemu wywołuje błąd aplikacji. Poleceniem **On Error** ustalamy reakcję na ten błąd – komenda przejścia do końca etykiety NacisnietoCancel, czyli do końca procedury i ominięcie w ten sposób przypisania czcionki do Text1.

**c) „Styl czcionki"**
Ręczny wybór stylu czcionki – w przeciwieństwie do TFont z VCL formatowania (Bold, Italic itd.) stanowią oddzielne zmienne logiczne. Chcąc zmienić formatowanie czcionki nie modyfikujemy zbioru jak było w obiekcie TFont w VCL, a jedynie ustawiamy wartość logiczną poszczególnych własności.
Obsługa ComboBoxów zgrupowanych w `Frame1` polegać więc będzie na ustawianiu wartości logicznej odpowiednich własności `Text1.Font` zgodnie z wartością własności Value odpowiednich ComboBoxów:

```
Private Sub Check2_Click()
    Text1.Font.Bold = Check2.Value
End Sub
```

```
Private Sub Check3_Click()
    Text1.Font.Italic = Check3.Value
End Sub

Private Sub Check4_Click()
    Text1.Font.Underline = Check4.Value
End Sub

Private Sub Check5_Click()
    Text1.Font.Strikethrough = Check5.Value
End Sub
```

**d) „Kolor tekstu"**

Stworzenie odpowiednika RadioGroup w Visual Basicu polega na zrzuceniu kilku OptionButtonów do pojemnika (container) np. Frame. Każdy OptionButton jest osobnym obiektem z dostępnymi dla programisty własnościami i metodami. Tworzymy procedury dla zdarzenia Click każdego z OptionButtonów:

```
Private Sub Option1_Click()
    Text1.ForeColor = vbWindowText
End Sub

Private Sub Option2_Click()
    Text1.ForeColor = vbBlack
End Sub

Private Sub Option3_Click()
    Text1.ForeColor = vbBlue
End Sub

Private Sub Option4_Click()
    Text1.ForeColor = vbGreen
End Sub

Private Sub Option5_Click()
    Text1.ForeColor = vbRed
End Sub
```

**e) „Czytaj" i „Zapisz"**

**Uwaga!** W trakcie projektowania aplikacji zapisany przez nas kod jest interpretowany, a nie kompilowany i dlatego nie powstaje plik .exe. W konsekwencji katalogiem roboczym jest katalog Visual Basica (np. C:\Program Files\Microsoft Visual Studio\VB98), a nie katalog projektu. Dlatego tam należy szukać stworzonego przez aplikację pliku. Aby tego uniknąć należy skompilować projekt poleceniem z Menu **File, Make Project1.exe ...** i uruchamiać plik exe.

TextBox w Visual Basicu, w przeciwieństwie do TMemo nie posiada metody pozwalającej zapisać jego zawartość do pliku. Konieczne jest więc samodzielne napisanie procedur obsługi czytania z i zapisu do plików. W poniższych listingach realizujących to zadanie zaprezentowano dwa możliwe podejścia. W procedurze czytającej z pliku tekstowego wykorzystano standardowe instrukcje BASICa obsługi plików, nastomiast procedura zapisu do pliku wykorzystuje obiekt FileSystemObject, którego metoda pozwala na stworzenie strumienia – pliku tekstowego.

```
Private Sub Command2_Click()
    Text1.Text = ""
    Dim NumerPliku As Integer
    Dim TekstNotesu As String
    NumerPliku = FreeFile 'Pobranie wolnego numeru pliku z zakresu 1 - 511
    Open "notes.txt" For Input Access Read As #NumerPliku
    Do While Not EOF(NumerPliku) 'Sprawdzamy czy nie doszlismy do konca
```

```
        Line Input #NumerPliku, TekstNotesu
        'Line Input czyta linię bez CR LF
        'przy kopiowaniu do Text1.Text należy je dodać
        Text1.Text = Text1.Text + TekstNotesu + Chr(13) + Chr(10)
    Loop
    Close #NumerPliku
End Sub
```

Przy zapisie do pliku skorzystamy z obiektów zdefiniowanych w VB. Obiekt typu **FileSystemObject** jest wykorzystany do powołania strumienia **TextStream**, z którego można czytać i do którego można pisać:

```
Private Sub Command3_Click()
    'CreateObject tworzy obiekt klasy podanej jako argument
    'Instrukcja BASICa Set przypisuje nowostworzony obiekt
    'do zmiennej file (odpowiednik słowa kluczowego new w C++)
    Set file = CreateObject("Scripting.FileSystemObject")
    'Korzystajac z metody FileSystemObject tworzymy TextStream
    'i przypisujemy go do zmiennej textfile
    Set textfile = file.CreateTextFile("notes.txt", True)
    'Zapis zawartosci Text1.Text do strumienia textfile
    textfile.Write Text1.Text
    'Zamkniecie strumienia
    textfile.Close
End Sub
```

Próba czytania z pliku 'notes.txt' przed jego stworzeniem skończy się błędem. W przeciwieństwie do obsługi błędów.w Delphi – spowoduje to zakończenie działania aplikacji.

**Uwaga!** Visual Basic dopuszcza użycie zmiennych bez ich deklaracji (w powyższej procedurze wartość zmiennych file i textfile jest przypisana instrukcją Set bez ich wcześniejszej deklaracji). Można tę możliwość w obrębie modułu wyłączyć ustawiając `Option Explicit` w pierwszej linii modułu.

**f) „Zaznacz wszystko"**

W kontrolce TextBox nie ma znanej z TMemo metody SelectAll. Można jednak wyznaczyć zakres zaznaczanego tekstu (zresztą niemal identycznie można to zrobić w produktach Borlanda) za pomocą własności **SelStart** i **SelLength**:

```
Private Sub Command4_Click()
    Text1.SelStart = 0
    Text1.SelLength = Len(Text1.Text)
    Text1.SetFocus
End Sub
```

Trzecia linia metody zwraca „focus" do okna edycyjnego. Inaczej nie widać byłoby zaznaczenia tekstu (chyba, że opcja Text1.HideSelection ustawiona jest na False).

**g) „Kopiuj" i „Wklej" – współpraca ze schowkiem**

Dostęp do informacji przechowywanej w schowku umożliwia komponent Clipboard. Obsługuje format tekstowy i kopiowanie obrazów (format DIB). TextBox nie posiada  żadnych metod związanych ze schowkiem, ale nie jest to wielkim utrudnieniem

```
Private Sub Command5_Click()
    Clipboard.Clear
    Clipboard.SetText (Text1.SelText)
End Sub

Private Sub Command6_Click()
    Text1.SelText = Clipboard.GetText(vbCFText)
End Sub
```

W istocie jest to dublowanie funkcji dostępnych już w Memo pod typowymi skrótami klawiszy (Ctrl+C, Ctrl+X, Ctrl+V).

**Zadanie 1**
Korzystając z istniejącego już na formie komponentu CommonDialog1 wywołać okno dialogowe otwarcia i zapisu pliku pozwalające użytkownikowi wybór nazwy pliku przy czytaniu (**CommonDialog1.ShowOpen**) i zapisie (**ShowSave**). W oknie dialogowym otwarcia pliku tak wybrać opcje, aby niemożliwe było czytanie pliku, który nie istnieje.

**Zadanie 2**
W przypadku braku zaznaczonego fragmentu tekstu przy próbie skopiowania zgłoś odpowiedni komunikat funkcją **MsgBox**.

**Zadanie 3**
Zastąpić komponent TextBox komponentem **RichTextBox** (należy wpierw udostępnić komponent Microsoft Rich TextBox Control 6.0) i tak zmodyfikować program, aby możliwe było formatowanie wybranego fragmentu tekstu (należy wykorzystać własności SelFontName, SelFontSize, SelColor, SelBold, SelItalic itp.).
Klasa RichTextBox zawiera metody **SaveFile** i **LoadFile** znakomicie ułatwiające operacje na plikach. Wadą tego komponentu jest konieczność dystrybucji i rejestracji Richtx32.ocx.

# 4. Dynamiczne tworzenie obiektów

Poniższy przykład demonstruje jak stworzyć kontrolkę Label w czasie działania programu w reakcji na kliknięcie formy. Aby stworzyć procedurę zdarzeniową klikamy formę (w trakcie projektowania) i domyślne zdarzenie Load zmieniamy na Click.

```
Private Sub Form_Click()
    Set DynamicButton = Controls.Add("VB.CommandButton", "cmdDynamic")
    With DynamicButton
        .Visible = True
        .Width = 3000
        .Caption = "Dynamic Button"
        .Top = 0
        .Left = 0
    End With
End Sub
```

Nowy obiekt/kontrolkę tworzymy za pomocą metody własności Controls formy **Form1.Controls.Add(*nazwa klasy*, *nazwa obiektu*)**. Controls jest zbiorem kontrolek znajdujących się na formie (por. z Components i Controls w VCL). **Ustalenie właściwej nazwy klasy i jej biblioteki jest wygodne dzięki Object Browserowi**. Jeżeli chcemy tworzyć obiekt typu CommandButton, który jest elementem biblioteki VB (w dolnym panelu Object Browsera biblioteka do której należy dana kontrolka jest zaznaczona na zielono) musimy jako nazwę klasy wpisać „VB.CommandButton" (razem z cudzysłowami, gdyż argumentem jest łańcuch). Drugim parametrem jest unikalna nazwa obiektu (niezależnie od zmiennej, do której będzie on przypisany poleceniem Set). W naszym przypadku otrzymujemy `Set DynamicButton = Controls.Add("VB.CommandButton", "cmdDynamic")`, co oznacza, że w zmiennej DynamicButton jest przechowywany obiekt/kontrolka typu CommandButton o nazwie cmdDynamic. Dwukrotne wykonanie tej procedury spowoduje błąd ze względu na próbę utworzenia drugiego obiektu nazwie „cmdDynamic". Można to oczywiście łatwo obejść modyfikując każdorazowo nazwę.

Teraz będziemy chcieli ze zdarzeniem Click dynamicznie tworzonego przycisku połączyć procedurę usuwającą go z formy. W tym celu musimy połączyć procedurę ze zdarzeniem Click dynamicznie utworzonego przez nas przycisku. Można to zrobić, ale warunkiem jest jawne zadeklarowanie używanej w programie zmiennej DynamicButton ze słowem kluczowym **WithEvents**, które powoduje, że procedury o nazwach DynamicButton_*zdarzenie* będą interpretowane jako odpowiednie procedury zdarzeniowe:

```
Private WithEvents DynamicButton As CommandButton
```

**Private** jest słowem kluczowym podobnym w działaniu do **Dim**, z tym, że zadeklarowana zmienna jest prywatna. Jak było powiedziane wyżej słowo kluczowe **WithEvents** powoduje, że zadeklarowana nazwa zmiennej DynamicButton jest **zmienną obiektową używaną do reagowania na zdarzenia wywoływane przez kontrolki** ActiveX[1]. Wówczas dopisana poniżej procedura o nazwie **DynamicButton_Click()** będzie rozumiana jako procedura zdarzeniowa zdarzenia Click obiektu związanego ze zmienną DynamicButton (obiekt i zdarzenie rozpoznawane jest po nazwie procedury – efekt widać na rozwijalnych listach na górze edytora).

```
Private Sub DynamicButton_Click()
    Form1.Controls.Remove "cmdDynamic"
End Sub
```

Procedura ta korzysta z kolejnej metody Form1.Controls (można użyć także `this.Controls` lub po prostu `Controls`) usuwając obiekt o podanej nazwie. Po jej użyciu, a więc po kliknięciu nowopowstałego przycisku można jeszcze raz powołać obiekt klikając formę.

# 5. Odtwarzacz plików audio

Celem tego ćwiczenia jest przygotowanie prostego odtwarzacza plików audio. Na formie umieszczamy **MMControl** (należy go wcześniej udostępnić w Project, Components ..., pozycja **Microsoft Multimedia Control 6.0**). Kreator ustawień MMControl znajduje się pod pozycją (Custom), ale dla naszych celów wystarczy edycja pól własności.

Komponent MMControl korzysta z systemowych bibliotek Windows. Odtwarza nie tylko pliki typu wave, midi i avi, ale również wszystkie zarejestrowane typy kompresji (np. mpeg, w tym mp3). Informacje o koderach można znaleźć w Panelu sterowania\Multimedia\zakładka Urządzenia\Kodery-dekodery kompresujące audio i wideo.

Korzystając z okna Properties ustalmy następujące własności:

1. Wartość **MediaPlayer1 . FileName** (korzystając z okna dialogowego) – należy wskazać plik audio wav. Coś na pewno powinno być w katalogu WINDOWS\MEDIA.
2. We własności **DeviceType** (w oknie Properties) należy wpisać WaveAudio[2].
3. Uruchomienie/uaktywnienie kontrolki możliwe jest dzięki wydaniu komendy Open za pomocą polecenia `MMControl1.Command = „Open”`. Najlepiej umieścić ją w Form_Load() – czyli w procedurze edytowanej po kliknięciu formy w trakcie projektowania.
4. Można zażądać, aby MediaPlayer miał wyłączność na korzystanie z danego urządzenia ustalając **Sharable** na False (wartość domyślna).
5. Kontrola pliku, w tym jego odtworzenie, możliwe jest za pomocą przycisków MMControl lub za pomocą odpowiadających im komend:
   ```
   MMControl1.Command = „Play”
   MMControl1.Command = „Stop”
   ```
   itd.
6. W zdarzeniu Form1.UnLoad należy umieścić komendę zamykającą odtwarzany plik
   ```
   MMControl1.Command = „Close”
   ```

Uwaga! Możliwości komponentu MMControl były zwiększane w kolejnych łatkach do VB i Visual Studio. Najświeższy Service Pack można ściągnąć ze strony Microsoft.

# 6. Odtwarzacz plików wideo

[1] Po zadeklarowaniu obiektu ze słowem WithEvents w rozwijalnych listach edytora pojawi się ten obiekt i jego zdarzenia ułatwiając tworzenie związanych z nim procedur zdarzeniowych.
[2] Inne możliwe wartości to AVIVideo, CDAudio, DAT, DigitalVideo, MMMovie, Other, Overlay, Scanner, Sequencer, VCR i Videodisc.

Zmodyfikujemy projekt z poprzedniego paragrafu tak, żeby odtwarzał podany plik AVI. W tym celu należy ustalić **MMControl1.DeviceType** na AVIVideo, do **FileName** przypisać plik AVI obsługiwany przez system. Po uruchomieniu aplikacji będzie on odtwarzany w osobnym oknie. Można posłużyć się kontrolkami posiadającymi uchwyt (ang. *handle*, najczęściej przechowywany we własności hWnd; w Delphi były to własności Handle) takie jak forma lub PictureBox jako ekranem. W tym celu do formy należy dodać np. PictureBox i przypisać wartość **MMControl1.hWndDisplay** na ten obiekt.
Jeżeli wszystkie powyższe czynności umieścić w procedurze zdarzeniowej Form_Load oraz dodać komunikaty o błędach, to będzie ona podobna do poniższej:

```
Private Sub Form_Load()
    With MMControl1
        .DeviceType = AVIVideo
        .FileName = "GLOBE.AVI"
        .hWndDisplay = Picture1.hWnd
        .Command = "Open"
        If .Error Then
            MsgBox .ErrorMessage 'Wyswietlenie systemowego komunikatu bledu
            Else
            .Command = "Play"
            If .Error Then 'Wyswietlenie systemowego komunikatu bledu
                MsgBox .ErrorMessage
            End If
        End If
    End With
End Sub
```

**Uwaga!** Warto zadbać o zainstalowanie najnowszych uaktualnień (Service Packów) do Visual Basica. Poprawiono między innymi działanie komponentu MMControl.

# 7. Prosta aplikacja bazodanowa

Podobnie jak biblioteka VCL, w Visual Basic zawiera komponenty wyspecjalizowane w łączeniu aplikacji z bazami danych. W VB jest to domyślnie dostępna kontrolka **Data**. Konfigurujemy ją podobnie jak TDatabase w VCL. Za pomocą okna Properties ustalamy wartości:

1) **Connect = Access** (typ bazy danych)
2) **DefaultType = 2 – UseJet** (domyślna wartość oznaczająca wykorzystanie mechanizmu bezpośredniego dostępu do baz danych MS Access)
3) **DatabaseName =** *nazwa pliku bazy danych* (korzystając z okienka dialogowego szukamy pliku Northwind.mbd znajdującego się prawdopodobnie w C:\Program Files\Microsoft Office\Office\Przykldy\
4) **RecordSource =** *nazwa tabeli* (np. Produkty)

W VB większość komponentów jest data aware (nie ma charakterystycznego dla VCL podziału na TEdit i TDBEdit). Umieśćmy więc **Text** i ustawmy jego właściwości:

1) **DataSource** = Data1 ( w przeciwieństwie do Delphi kontolki nie wymagają pośrednictwa komponentu typu TDataSource z VCL)
2) **DataField** = NazwaProduktu

Po skompilowaniu dane powinny być już widoczne. Warto oczywiście dodać więcej kontrolek związanych z innymi polami tabeli lub obejrzeć całość w siatce.

Aby komponent DBGrid był dostępny należy dodać **Microsoft Data Bound Grid Control 5.0** w Project, Components. Ważne, żeby użyć komponentu w wersji 5.0, bo nowszy (korzystający ze standardu OLE DB) nie współpracuje z kontrolką Data. Po zrzuceniu go na formę łączymy go z Data1 podobnie jak w przypadku okienka edycyjnego ustalając **DataSource** na Data1. Warto zajrzeć do edytora własności (Custom), który pozwala zmienić wygląd siatki.

# 8. Łączenie z bazą danych za pomocą ADO (OLE DB), wykresy

Bardzo podobnie odbywa się łączenie z bazami danych za pomocą mechanizmu ADO. Do projektu należy dodać komponenty **Microsoft ADO Data Control 6.0** oraz **Microsoft DataGrid Control 6.0** i **Microsoft Chart Control 6.0**. Są to komponenty korzystające z techniki OLE DB (wyjaśnienie tajemniczych skrótów związanych z bazami danych znaleźć można w VB FAQ rozdział 13, pytanie 7).

**Adodc** – wykorzystamy edytor (Custom). Budowanie Connection String odbywa się analogicznie jak w Delphi/C++ Builderze przy TADOTable (wykorzystywane są nawet te same systemowe okienka dialogowe):
1) **Dostawca:** Microsoft Jet 4.0 OLE DB Provider
2) Wybieramy nazwę bazy danych (np. Northwind.mdb)
W efekcie łańcuch połączenia powinien być następujący:
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Microsoft Office\Office\Przykldy\Northwind.mdb;Persist Security Info=False
Należy jeszcze wybrać tabelę. Posłużymy się edytorem związanym z własnością **RecordSource**. Jej wartość ma różny sens w zależności od wartości ustawionej we własności **CommandType** (dostępnej również z tego edytora). Najważniejsze są dwie możliwości:

| **CommandType** | Sens **RecordSource** | Wartość **RecordSource** |
|---|---|---|
| 1 – adCmdText | komenda SQLa | SELECT * FROM Produkty |
| 2 – adCmdTable | nazwa tabeli | Produkty |

Skonfigurowanie **DataGrid1** jest już przyjemnością – wystarczy w polu **DataSource** wybrać Adodc1.

Równie nieskomplikowane jest konfigurowanie opcji **MSChart1**. Należy ustawić wartość **DataSource**, a następnie posłużyć się edytorem (Custom).

# 9. Pliki projektu:

| **Visual Basic** | **Opis** |
|---|---|
| *.vbp | Główny plik projektu |
| *.vbw | Informacje o opcjach środowiska programistycznego (workspace) |
| *.bas | Typowe rozszerzenie nazwy pliku z kodem BASICa. |
| *.cls | Moduł z definicją klasy |
| *.frm | Pliki, w których VB przechowuje informacje o zaprojektowanych przez użytkownika elementach formy. Tu znajduje się też kod procedur zdarzeniowych |
| *.dcu | Skompilowane formy i inne objekty tworzone przez użytkownika w oddzielnych plikach; można je skasować jeżeli posiadamy źródła |
| *.res | Plik *.res przechowuje zasoby (bitmapy, kursory), których wymaga tworzony przez użytkownika komponent – np. ikonę jaka pojawi się na palecie. |

# 10. Słówko na temat VBScript

VBScript jest alternatywnym dla JavaScriptu skryptowym językiem umieszczanym w kodzie stron HTML. Podstawowe zasady składni języka są identyczne jak w Visual Basicu: skrypt składa się z procedur, które mogą odwoływać się do zdefiniowanych na stronie obiektów i być powiązane z ich zdarzeniami.
Umieszczanie skryptów w kodzie HTML odbywa się analogicznie do przypadku skryptów JavaScript. Służy do tego znacznik `<SCRIPT>` z opcją `LANGUAGE="VBScript"`. Ponieważ każdy element HTML może mieć

nazwę określoną przez wartość parametru NAME, to procedury mogą być związane ze zdarzeniami przez użycie typowej dla VB konwencji nazw procedur zdarzeniowych (*nazwaobiektu_zdarzenie*).

Poniżej znajduje się przykład procedury zdarzeniowej reagującej na naciśnięcie przycisku Command1.

```
<HTML>
<HEADER>
<TITLE>Pierwszy VBScript</TITLE>

<SCRIPT LANGUAGE="VBScript">

    'W VB analogiczne zdarzenie nazywa sie Click, tu OnClick
    Sub Command1_OnClick()
        'Prosciej juz byc nie moze
        MsgBox "Hello World!", 0, "Okienko wywołane przez VBScript"
    End Sub

</SCRIPT>

</HEADER>
<BODY>

<INPUT TYPE="SUBMIT"
       NAME="Command1"
       VALUE="Uruchom skrypt Command1_OnClick">

</BODY>
</HTML>
```

Jak skrypt jest bardzo prosty – skorzystaliśmy z typowej funkcji MsgBox wyświetlającej okienko komunikatu.

**Uwaga!** Aby wywołać skrypt nie trzeba korzystać z przycisku. Może być to dowolny obiekt HTML reagujący na działanie użytkownika np. łącze `<A NAME="Command1">Obiekt uruchamiajacy skrypt</A>` lub obrazek `<IMG SRC="..." NAME="Command1">`. Ważne, żeby nazwa skryptu składała się z nazwy tego obiektu i nazwy zdarzenia w konwencji nazywania procedur zdarzeniowych w VB.

# III. Gdzie szukać pomocy?

W kolejności:

1. Uruchomić help (termin z którym mamy problem + F1) – działa jedynie w przypadki gdy zainstalowaliśmy z osobnego CD-ROMu MSDN (*Microsoft Data Network*) pełniący od wersji 5.0 rolę helpa, względnie zajrzeć do jego wersji on-line http://msdn.microsoft.com/
2. Jeżeli problem dotyczy np. WinAPI należy zajrzeć do MS Help lub któregoś z licznych FAQ dotyczących WinAPI i VB.
3. Zajrzeć do książek, stron poświęconych VB i tutoriali (np. http://vb4all.canpol.pl/ lub http://physinfo.ulb.ac.be/cit_courseware/vb6/default.htm)
4. Sprawdzić czy problem jest opisany w którymś FAQ (np. na stronie grupy dyskusyjnej pl.comp.lang.vbasic, czyli http://www.vbfaq.pl/) lub w na serwerze MSDN http://msdn.microsoft.com/vbasic/
5. Wysłać pytanie na pl.comp.lang.vbasic

# IV. Współpraca Visual Basica i Microsoft Office?

## 1. Osadzanie obiektu OLE2 w aplikacji

Systemowy mechanizm łączenia i osadzania obiektów OLE2 (*Object Linking and Embeding*) [3] pozwala na łatwe umieszczanie obiektów zdefiniowanych w zainstalowanych w systemie aplikacjach, np. arkusza Excela, w naszej aplikacji.

Jest to zadanie tym prostsze, że w Visual Basicu znajduje się domyślnie dostępna na palecie komponentów kontrolka o nazwie OLE. Analogiczny komponent o nazwie TOLEContainer dostępny jest także w VCL. Po położeniu kontrolki na formie otwierane jest systemowe okienko „Wstawianie obiektu" pozwalające wybrać obiekt spośród zarejestrowanych w systemie (potem funkcja wstawiania obiektu jest dostępna w menu kontekstowym). Wybierzmy „Arkusz Microsoft Excel". Można stworzyć pusty dokument lub skorzystać z istniejącego. Wybierzmy „Utwórz nowy". Po zaakceptowaniu zobaczymy arkusz Excela wewnątrz naszej aplikacji.

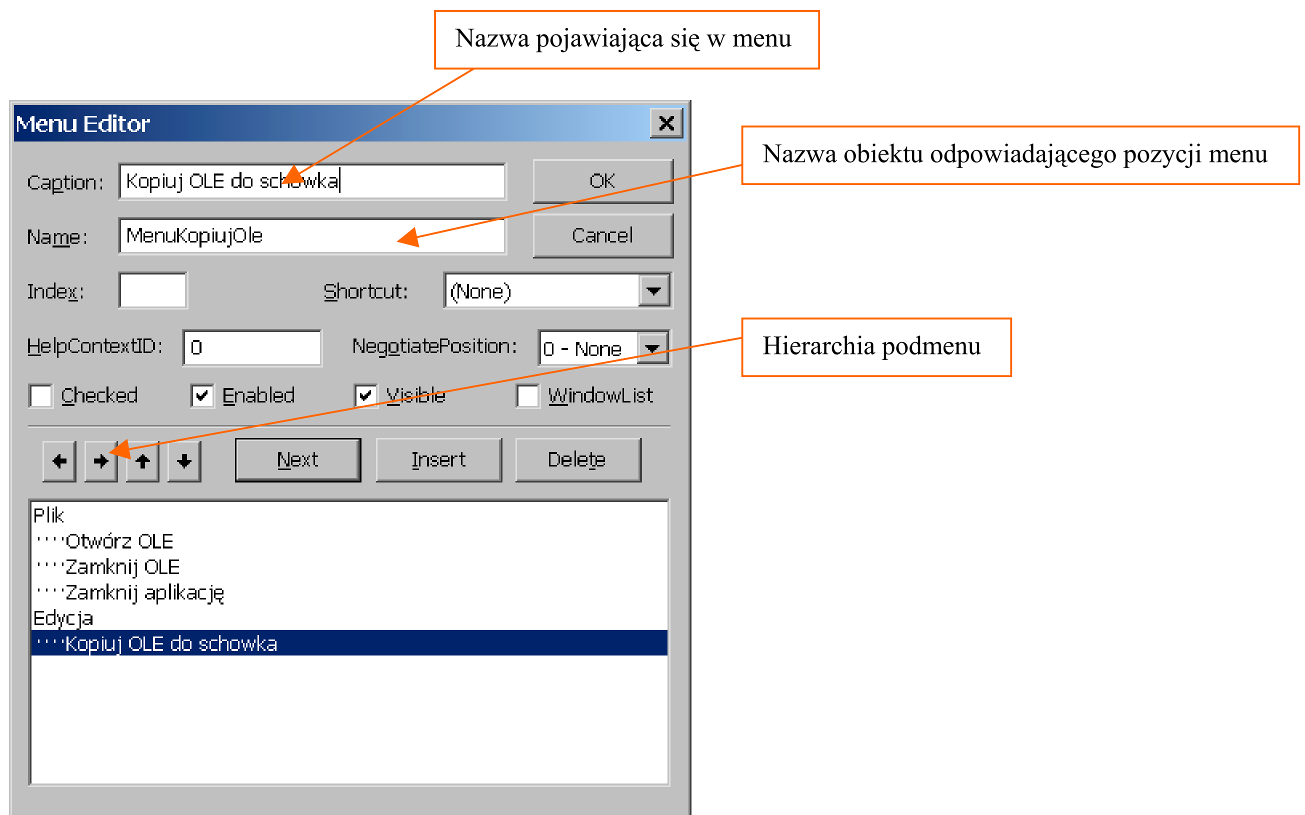
Za typ obiektu odpowiada własność **Class**, którą można oczywiście modyfikować.
Dopuszczalne formy połączenia (osadzenie i/lub połączenie) ustalane są przez zmienną **OLETypeAllowed**. Obiekt osadzony jest otwierany wewnątrz okna macierzystej aplikacji, integruje się z nią (np. menu serwera OLE i klienta OLE są łączone). Natomiast obiekt połączony powoduje otwarcie niezależnego okna serwera OLE i edycję dokumentu „na zewnątrz" naszej aplikacji.

Po uruchomieniu aplikacji i dwukrotnym kliknięciu na pole zajmowane przez OLE pojawi się arkusz Excela. Akcję wywołującą arkusz można modyfikować wykorzystując własność OLE1.**AutoActive**, można oczywiście zaktywować połączenie programowo. W tym celu stworzymy menu główne aplikacji i umieścimy tam odpowiedzialne za to pozycje.

Ważną własnością mechanizmu OLE jest łączenie menu osadzanego obiektu i menu formy (w przypadku MDI będzie to menu formy macierzystej). Warunkiem jest, żeby menu główne (nawet puste) w aplikacji istniało. Nie całe menu jest dołączane, pomijane są pozycje odpowiedzialne za obsługę plików, pozostawiając to zadanie klientowi OLE (nie będzie więc zazwyczaj menu Plik serwera OLE). Dołączanie menu można oczywiście kontrolować – warunkiem jest ustawienie **Form1.NegotiateMenus = True**, a pozycje menu, które chcemy zachować po połączeniu muszą mieć własność **NegotiatePosition** różną od 0 – None (zob. Menu Editor...). **Uwaga!** Łączenie menu dotyczy tylko głównych pozycji. Podmenu o identycznych nazwach zostaną powtórzone, a nie połączone.

**Tworzenie menu głównego aplikacji**
Stwórzmy menu główne zawierające menu Plik (Otwórz OLE, Zamknij OLE, Zapisz OLE do pliku, Zamknij aplikację) i Edycja (Kopiuj z OLE do schowka). Aby stworzyć menu klikamy prawym klawiszem myszy na projektowaną formę i z menu kontekstowego wybieramy **Menu Editor ...**. Jego wykorzystanie jest mniej intuicyjne niż w Delphi/Builderze, ale nie jest najgorzej (zob. rysunek poniżej).

[3] **OLE 2.0** jest 32-bitową, znacznie poprawioną wersją znanego z Windows 3.1 mechanizmu OLE 1.0. Na tym nie koniec rozwoju mechanizmu osadzania obiektów, gdyż OLE2 stał się bazą mechanizmu COM (*Component Object Model*) wykorzystanego w rozwinięciu OLE2 – **OCX**, które udoskonaliło możliwość definiowania własnych osadzalnych obiektów. OCX z powodów marketingowych zostało przemianowane na **ActiveX**. Wreszcie najnowszym wynalazkiem jest DCOM (Distributed COM) pozwalające na komunikację komponentów ActiveX przez sieć.




Następnie oprogramowujemy pozycje menu:

Plik, Otwórz OLE:
```
Private Sub MenuOtworzOLE_Click()
    'Polecenia połączenia z obiektem OLE 2.0
    OLE1.DoVerb
    'Pozostale polecenia steruja dostepnoscia elementow menu
    MenuOtworzOLE.Enabled = False
    MenuZamknijOLE.Enabled = True
    MenuZapiszOLE.Enabled = True
    MenuKopiujOle.Enabled = True
End Sub
```

Aby nie martwić się o dostępność pozycji menu w sytuacji, gdy użytkownik połączy się z obiektem klikając dwukrotnie na obszar OLE1 zamiast używając menu zablokujemy pierwszą możliwość ustawiając **OLE1.AutoActivate = 0 – Manual** za pomocą okna Properties.

Plik, Zamknij OLE:
```
Private Sub MenuZamknijOLE_Click()
    'Polecenia zamknięcia połączenia z obiektem OLE 2.0
    OLE1.Close
    'Pozostale polecenia steruja dostepnoscia elementow menu
    MenuOtworzOLE.Enabled = True
    MenuZamknijOLE.Enabled = False
    MenuZapiszOLE.Enabled = False
    MenuKopiujOle.Enabled = False
End Sub
```

Plik, Zamknij aplikację:
```
Private Sub MenuZamknijAplikacje_Click()
    Unload Me
End Sub
```

Edycja, Kopiuj OLE do schowka:
```
Private Sub MenuKopiujOle_Click()
    OLE1.Copy
End Sub
```

Kopiowanie do schowka w przypadku OLE nie oznacza kopiowania zawartości[4], ale kopiowanie całego obiektu OLE. Skopiowany obiekt OLE można wkleić np do Worda i edytowac tak jak każdy osadzony obiektu OLE, identycznie jak w naszej aplikacji. Podobnie rzecz ma się z zapisywaniem do pliku metodą SaveToFile. Plik, który powstanie **nie jest arkuszem Excela**, a plikiem zawierającym obiekt OLE 2.0. Może on być następnie wczytany do aplikacji metodą ReadFromFile.

Plik, Zapisz OLE do pliku
```
Private Sub MenuZapiszOLE_Click()
    Dim NumerPliku As Integer
    NumerPliku = FreeFile()
    'Zapisany plik nie bedzie arkuszem Excela!
    Open "OLEFile.ole" For Random As #NumerPliku
    OLE1.SaveToFile NumerPliku
    Close #NumerPliku
End Sub
```

## 2. Automation - wykorzystanie obiektów Microsoft Excel 8.0/97

Jeżeli mamy zainstalowany Office 97 lub nowszy, możemy korzystać z obiektów Excela w taki sposób jakby były bibliotekami dołączonymi do Visual Basica. W rzeczywistości z Visual Basiciem rozprowadzane są jedynie deklaracje obiektów, które zaawiera Excel i zainstalowanie samego Excela jest niezbędne. Podobnie wygląda współpraca z innymi aplikacjami Offica.
Zbiór deklaracji dla Excela można dołączyć do projektu (i w ten sposób udostępnić odpowiednie obiekty) w menu **Project, References ...**, pozycja **Microsoft Excel 8.0 Object Library** (wersja 8.0 odpowiada Officowi 97). Poza możliwością współpracy z aplikacjami Excela uzyskujemy możliwość wykorzystania bogatego zbioru wbudowanych w niego funkcji.

Mechanizm umożliwiający korzystanie z odpowiednio zarejestrowanych w systemie obiektów innej aplikacji nazywa się „OLE Automation" (lub „ActiveX Automation" lub po prostu „Automation"). W tym kontekście używa się terminów serwer **serwer automatyzacji** dla określenia aplikacji udostępniającej obiekty (w naszym przypadku Excela) i **klient automatyzacji** – czyli aplikacji, która z tych obiektów korzysta.

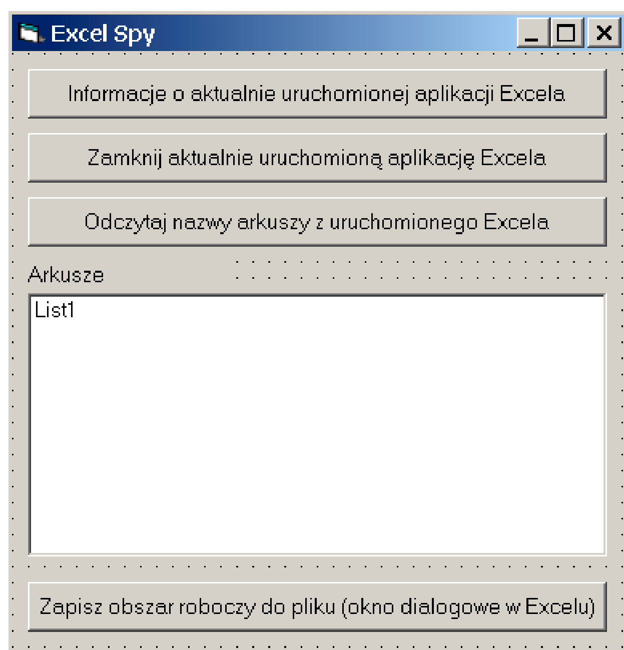
Zawartość dołączonej biblioteki można obejrzeć korzystając z Object Browser (F2). Wybierzmy w górnej rozwijalnej liście bibliotekę Excel. Wśród obiektów znajdujących się w tej bibliotece szczególną rolę pełni obiekt **Application** (w kodzie należy używać pełnej ścieżki `Excel.Application`) reprezentujący środowisko Excela. Jego podobiektami są **Workbooks** (zeszyty), który zawiera z kolei arkusze (**Worksheet**). Najniższym poziomem są elementy arkusza – komórki. W efekcie, jeżeli chcemy odczytać wartość komórki C2 w pierwszym arkuszu bieżącego skoroszytu musimy posłużyć się następującą konstrukcją:
`ObiektExcelApp.ActiveWorkbook.Sheets(1).Cells(2, 3).Value` lub
`ObiektExcelApp.ActiveWorkbook.Sheets(1).Range("C2").Value`, gdzie ObiektExcelApp został zadeklarowany jako aplikacja Excela: `Dim ObiektExcelApp As Excel.Application`[5].
Własności Worksheet.Range (*zakres komórek*)[6] i Worksheet.Cells zwracają obiekt typy **Range**. Jest to niezwykle istotny obiekt umożliwiający operacje na komórkach, od formatowania wyglądu po operacje na ich wartościach lub formułach.
Warto również wspomnieć o funkcjach Excela udostępnionych w obiekcie **Application.WorksheetFunction**.

---
[4] Aby skopiować zawartość osadzonego arkusza można wykorzystać pozycję menu dodanego przez osadzany obiekt.
[5] Sprawa znacznie się upraszcza, jeżeli interesuje nas aktywna komórka. Aby odczytać jej adres w arkuszu wystarczy użyć: `ObiektExcelApp.ActiveCell.Address`.
[6] Za pomocą metody Range można określić wiele komórek podając zakres w formacie Excela np. A1:C3.

**a) Współpraca aplikacji z uruchomioną aplikacją Excela**

Tworzymy aplikację składającą się z kilku niezależnych procedur – każda z nich będzie tworzyć, wykorzystywać i zwalniać obiekt Excel.Application i, w razie potrzeby, inne obiekty Excela.

Umieśćmy na formie cztery przyciski i listę (ListBox) podobnie jak na rysunku.




**Informacje o aktualnie uruchomionej aplikacji Excela** (przycisk Command1)
Do wyświetlenia informacji posłużymy się funkcją **MsgBox** *informacja*, , *tytuł*. Pominięty drugi argument określa znajdujące się na okienku z informacją przyciski (MsgBox jest odpowiednikiem Application.MessageBox z Delphi).

```
Private Sub Command1_Click()
    'deklaracja zmiennej typu obiekt Excel.Application
    'jezeli usuniemy deklaracje - zostanie zastapiona domyslna (Variant)
    Dim ObiektExcelApp As Excel.Application
    'Tu obiekt nie jest tworzony, a jedynie do zmiennej przypisywany jest
    'obiekt istniejacy (uruchomiona aplikacja Excela)
    Set ObiektExcelApp = GetObject(, "Excel.Application")
    'Informacja o aktywnym skoroszycie, arkuszu i komórce
    With ObiektExcelApp
        MsgBox "Aktywny zeszyt: " & .ActiveWorkbook.Name & Chr(10) & _
               "Aktywny arkusz: " & .ActiveSheet.Name & Chr(10) & _
               "Adres aktywnego pola: " & .ActiveCell.Address & Chr(10) & _
               "Wartość aktywnego pola: " & .ActiveCell.Value _
               , , "Informacje o aktualnie uruchomionej aplikacji Excela"
    End With
    Set ObiektExcelApp = Nothing
End Sub
```

W powyższej procedurze, żeby nie zaciemniać obrazu, pominięto obsługę błędu np. braku aktualnie uruchomionej aplikacji Excela.

**Uwaga!** Funkcja MsgBox przyjmuje jako pierwszy argument łańcuch utworzony z połączenia (znakiem &) łańcuchów komentarza i odczytanych z Excela nazw. Ponieważ całe to polecenie jest dość długie – podzielone zostało na kilka linii. Znakiem kontynuacji w Visual Basicu jest podkreślenie (znak _ ).

**Zamknij aktualnie uruchomioną aplikację Excela** (przycisk Command2)
W następnej metodzie powołujemy obiekt aplikacji tylko po to, żeby wykorzystać jego metodę Quit:

```
Private Sub Command2_Click()
    Dim ObiektExcelApp As Excel.Application 'Deklaracja obiektu
    Set ObiektExcelApp = GetObject(, „Excel.Application") 'Przypisanie
    ObiektExcelApp.Quit 'Wykorzystanie funkcji Excel.Application.Quit
    Set ObiektExcelApp = Nothing 'Zwolnienie zmiennej
End Sub
```

**Odczytaj nazwy z uruchomionego Excela** (przycisk Command3)
Zajrzymy do aktywnej aplikacji Exela i w ListBoxie umieścimy informacje o jego wszystkich zeszytach i arkuszach.

```
Private Sub Command3_Click()
    'Deklaracja zmiennej i pobranie obiektu
    Dim ObiektExcelApp As Excel.Application
    Set ObiektExcelApp = GetObject(, "Excel.Application")
    'Czyszczenie listboxu
    List1.Clear
    'Czytanie nazw zeszytów i arkuszy
    For n = 1 To ObiektExcelApp.Workbooks.Count 'Petla po skoroszytach
        For m = 1 To ObiektExcelApp.Workbooks.Item(n).Sheets.Count 'Arkusze
            With ObiektExcelApp.Workbooks.Item(n)
                List1.AddItem (.Name & ", " & .Sheets.Item(m).Name)
            End With
        Next m
    Next n
    'Zwolnienie zmiennej
    Set ObiektExcelApp = Nothing
End Sub
```

Procedura wykonuje dwie pętle. Zewnętrzna (ze zmienną n) przebiega po skoroszytach (wykorzystano własność zawierającą ilość skoroszytów w aplikacji – Application.Workbooks.Count). Pętla wewnętrzna (ze zmienną m przebiegającą od 1 do wartości własności Count w Sheets) odczytuje nazwy akruszy i razem z nazwami zeszytów umieszcza je w liście.

**Zapisz obszar roboczy do pliku (okno dialogowe w Excelu)** (przycisk Command4)

```
Private Sub Command4_Click()
    Dim ObiektExcelApp As Excel.Application 'Deklaracja zmiennej obiektowej
    Set ObiektExcelApp = GetObject(, "Excel.Application") 'Przypisanie
    Do
        NazwaPliku = ObiektExcelApp.GetSaveAsFilename 'Okno Save As Excela
    Loop Until NazwaPliku <> False 'Powtorzenie jezeli nacisniety Cancel
    ObiektExcelApp.SaveWorkbook FileName:=NazwaPliku 'Zapis do pliku
End Sub
```

Nowością jest użycie metody **GetSaveAsFilename** obiektu Excel.Application do otwarcia Excelowego okienka „Save As", co uwalnia nas od dodawania do projektu komponentu CommonDialog i jego konfiguracji.

**b) Edycja komórek Excela z poziomu aplikacji**

Podobnie jak poprzedni projekt, w tym również będziemy łączyć się z działającą aplikacją Excela. Oznacza to, że w przypadku braku uruchomionego Excela pojawi się błąd. Tym razem do procedury inicjującej automatyzację włączymy obsługę błędów instrukcją **On Error GoTo**.

Deklarujemy w module zmienną typu Excel.Application oraz zmienną arkusza (z obsługą zdarzeń):

```
'Deklaracja zmiennych obiektowych
Private ObiektExcelApp As Excel.Application
Private WithEvents ObiektExcelArkusz As Excel.Worksheet
```

Do komórek arkusza można by oczywiście dostać się wykorzystując ObiektExcelApp, ale jeżeli chcemy, aby aplikacja reagowała na zmiany zachodzące w arkuszu (edycja, zmiana aktywnej komórki) to niezbędna jest obsługa zdarzeń arkusza, a to wymaga zadeklarowania osobnego obiektu (inaczej nie ma możliwości takiego nazwania funkcji, żeby powiązać procedurę ze zdarzeniem obiektu).

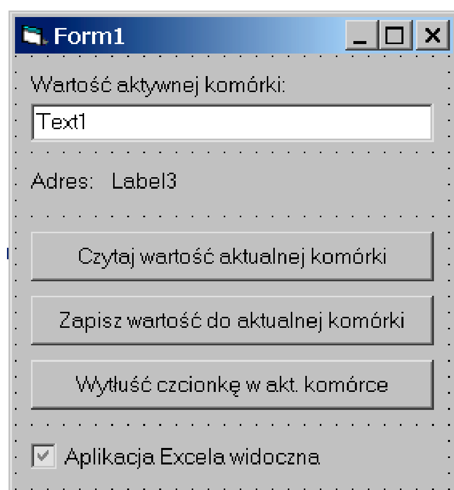
Po uruchomieniu aplikacji przypisujemy zadeklarowanym zmiennym obiekty z wcześniej działającego Excela:

```
Private Sub Form_Load()
    'Obsluga bledu np. gdy Excel nie jest uruchomiony
    On Error GoTo BladExcelApp

    'Laczenie z obiektem aktualnie uruchomionego Excela
    Set ObiektExcelApp = GetObject(, "Excel.Application")
    Set ObiektExcelArkusz = ObiektExcelApp.ActiveSheet
    Exit Sub 'Wyjscie z procedury

'Obsluga bledu (wyswietlenie komunikatu i zakonczenie dzialania programu)
BladExcelApp:
    MsgBox "Blad inicjacji obiektu Excela (" & Err.Description & ")"
    Unload Me
End Sub
```

Po tych przygotowaniach zadeklarowane obiekty Excela widoczne są w całym module – nie musimy ich deklarować w każdej procedurze z osobna. Ma to oczywistą zaletę w postaci przejrzystości programu, ale naraża aplikację na pojawienie się błędu gdy użytkownik zamknie Excela.




Na formie umieszczamy okienko edycyjne, label oraz trzy przyciski.

**Czytaj wartość aktualnej komórki**

```
Private Sub Command1_Click()
    With ObiektExcelApp
    Label3.Caption = .ActiveWorkbook.Name & " . " & _
                     .ActiveSheet.Name & " . " & _
                     .ActiveCell.Address
```

```
    Text1.Text = .ActiveCell.Value
    End With
End Sub
```

W tej procedurze wykonywane są dwie instrukcje. W pierwszej Label3 przypisywane są nazwa bieżącego skoroszytu, nazwa bieżącego arkusza i wreszcie adres aktywnej komórki. W drugiej do Text1 kopiowana jest zawartość tej komórki.
Ta procedura powinna być także wywoływana przy uruchomieniu aplikacji. Zapewni to właściwe wartości w polu tekstowym i wyświetlenie adresu aktywnej komórki bez podejmowania żadnych działań ze strony użytkownika. W tym celu należy dodać linię z poleceniem

```
Command1_Click
```

przed wywołaniem `Exit Sub` w procedurze `Form_Load`.

**Zapisz wartość do aktualnej komórki**

```
Private Sub Command2_Click()
    ObiektExcelApp.ActiveCell.Value = Text1.Text
End Sub
```

Tym razem działanie jest odwrotne – do aktywnej komórki Excela kopiowana jest zawartość okienka edycyjnego Text1.

**Wytłuść czcionkę w akt. komórce**
Jako przykład, bardzo prosty, formatowania arkusza napiszemy procedurę wytłuszczającą czcionkę w aktywnej komórce:

```
Private Sub Command3_Click()
    ObiektExcelApp.ActiveCell.Font.Bold = True
End Sub
```

**Aplikacja Excela widoczna**
Pozostaje ostatnia kontrolka (CheckBox), która ma umożliwić ukrywanie i pokazywanie aplikacji Excela. „Widzialność" aplikacji Excela związana jest z własnością **Excel.Application.Visible** i jej wartość w tej procedurze kontrolujemy.

```
Private Sub Check1_Click()
    ObiektExcelApp.Visible = Check1.Value
End Sub
```

**Uwaga!** Ukrycie aplikacji i zamknięcie naszej aplikacji spowoduje, ze w systemie będzie działać ukryty Excel, którego użytkownik nie może zamknąć w zwykły sposób. Na szczęście VB łączy się zawsze z Excelem uruchomionym jako pierwszym, więc możemy jeszcze raz uruchomić naszą aplikację i ujawnić Excela.

Efekt jaki uzyskaliśmy jest połowiczny, bowiem nasza aplikacja nie reaguje jeszcze na zmiany dokonane przez użytkownika w Excelu. Informację o Excelu można uzyskać jedynie naciskając Command1. Jeżeli chcemy, aby aplikacja reagowała na zmiany automatycznie, musimy „uczulić" nasz program na zdarzenia obiektu Excela. Zadeklarowaliśmy obiekt arkusza ObiektExcelArkusz, który posiada m.in. dwa interesujące nas zdarzenia: **SelectionChange** wywoływany przy zmianie aktywnej komórki oraz **Change** związany ze zmianą wartości w komórkach. Musimy więc napisać procedury, których nazwy odpowiadają schematowi *obiekt_zdarzenie*(), a VB zwiąże je z odpowiednimi zdarzeniami. Tym razem deklarację procedury musimy napisać sami (nie ma obiektu na formie do kliknięcia) lub wybrać obiekt i zdarzenie z list w edytorze kodu[7]. Zakończenie „End Sub" edytor doda sam.

```
'Tu trzeba samodzielnie napisac takze szkielet procedury
```

[7] Warunkiem pojawienia się obiektu w rozwijalnych listach na górze edytora jest jego poprawna deklaracja ze słowem WithEvents.

```
Private Sub ObiektExcelArkusz_SelectionChange(ByVal Target As Range)
    Form1.Caption = "Zmiana aktywnej komórki"
    Command1_Click
End Sub

'To samo dla zmiany wartosci aktywnej komorki
Private Sub ObiektExcelArkusz_Change(ByVal Target As Range)
    Form1.Caption = "Zmiana wartości"
    Command1_Click
End Sub
```

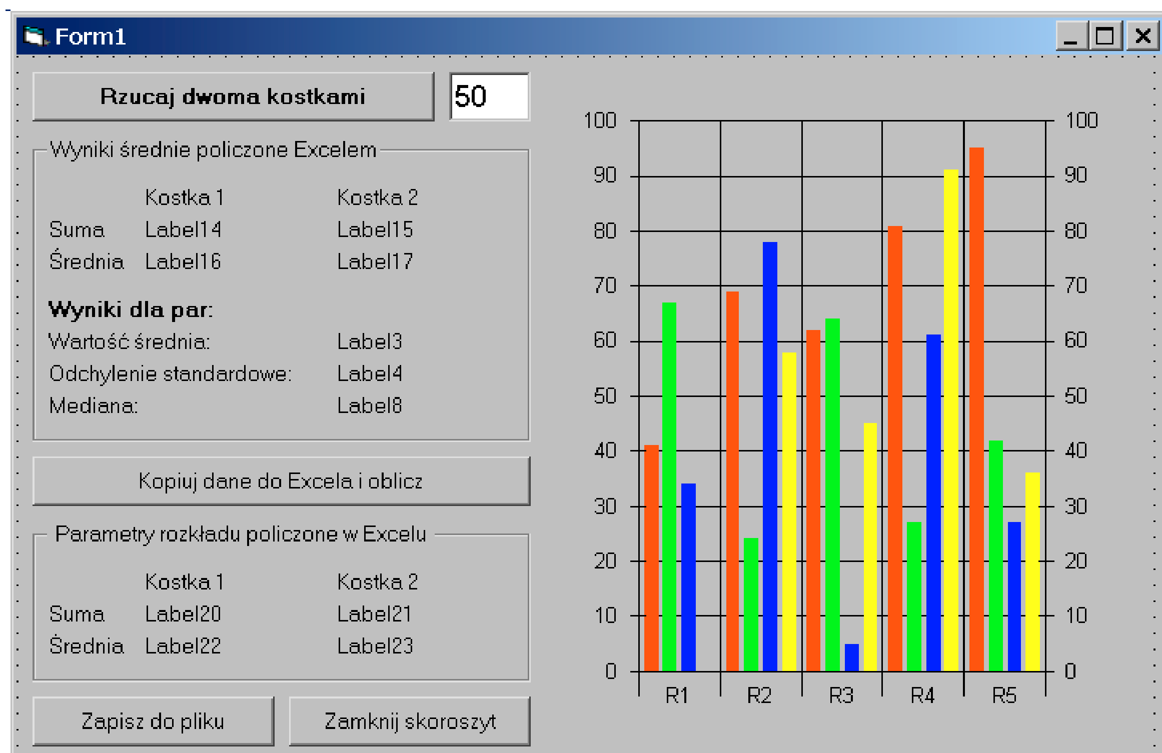
Jedynym istotnym zadaniem tych procedur jest wywołanie Command1_Click, która pobiera informacje o aktywnej komórce. Dodatkowo wyświetlana jest informacja na pasku tytułowym aplikacji identyfikująca zdarzenie.

**Uwaga!**
Aż się prosi, żeby do zdarzeń ObiektExcelArkusz.SelectionChange i ObiektExcelArkusz.Change przypisać funkcję Command1_Click bez tworzenie pośrednich procedur zdarzeniowych. Utrudnieniem jest fakt, że zdarzenia te wymagają procedury z argumentem, którego Command1_Click nie posiada.

**Zadanie 1**
Przypisanie zmiennej ObiektExcelArkusz bieżącego arkusza w Form_Load ma oczywistą wadę, a mianowicie po zmianie aktywnego arkusza przez użytkownika, aplikacja nadal będzie odnosiła się do arkusza pierwotnego. Należy obsłużyć zdarzenie zmiany zeszytu i arkusza tak, żeby cały czas adres podawany w aplikacji odnosił się do aktywnej komórki w bieżącym arkuszu.

**Zadanie 2**
Dodać obsługę błędów we wszystkich procedurach.

**c) Wykorzystanie funkcji Excela do wykonania obliczeń**
Z funkcji Excela można korzystać dwojako. Po pierwsze wykonywać je do obliczeń na wewnętrznych zmiennych i tablicach aplikacji. Po drugie można wykonywać obliczenia w stworzonym arkuszu i pobierać jedynie wynik. W drugim przypadku, kopiowanie dużej ilości danych do arkusza należy wykonać przekazując całą tablicę, a nie kolejne jej elementy w pętli. Oszczędzi to sporo czasu procesora.



Zadaniem aplikacji jest symulowanie serii rzutów kostką (ilość rzutów w serii będzie ustalana przez użytkownika w trakcie działania aplikacji), obliczenia sumy, wartości średniej, odchylenia standardowego i mediany oraz zapisanie wyników do labeli oraz wykresu (wykres będzie kumulował ilości kostek z poszczególną liczbą oczek, aby zilustrować rozkład normalny opisany krzywą Gaussa). Wszystko to umieścimy w procedurze zdarzeniowej Command1_Click. Drugi przycisk skopiuje dane do Excela i policzy wyniki wpisując do komórek arkusza odpowiednie funkcje. Wreszcie dodatkowymi przyciskami będzie można zapisać dane do pliku i zamknąć skoroszyt.

Poniżej znajduje się pełen listing projektu z komentarzami. Jednak zamiast go kopiować lepiej pobrać projekt znajdujący się w pliku http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad3.zip:

```
Option Explicit 'Wymuszenie deklaracji wszystkich zmiennych
Option Base 1 'Indeksowanie tablic zaczyna sie od 1

'Zmienne globalne modulu (Private - widoczne tylko w tym module)
Private Ile_Rzutow As Long
Private ObiektExcelApp As Excel.Application 'Aplikacja Excela
Private ObiektExcelSkoroszyt As Excel.Workbook 'Skoroszyt/Zeszyt Excela
Private Tablica_Wynikow_Kostka_1() As Long 'Uwaga! Tablice bez wymiaru
Private Tablica_Wynikow_Kostka_2() As Long
Private Tablica_Wynikow_Par() As Long
Private Tablica_Zliczen_Par(12) As Long

'Przycisk "Rzucaj dwoma kostkami"
Private Sub Command1_Click()
    Rzucaj_kostkami
End Sub

'Przycisk "Kopiuj dane do Excela i oblicz"
Private Sub Command4_Click()
    Oblicz_w_Excelu
End Sub

'Przycisk "Zapisz do pliku"
Private Sub Command5_Click()
    ObiektExcelSkoroszyt.SaveAs ("Kostki_aplikacja.xls")
End Sub

'Przycisk "Zamknij skoroszyt"
Private Sub Command6_Click()
    ObiektExcelSkoroszyt.Close
End Sub

'Inicjacja zmiennych, uruchamianie nowego Excela, zerowanie wykresu
Private Sub Form_Load()
    Ile_Rzutow = 50 'Wartosc poczatkowa
    ReDim Tablica_Wynikow_Kostka_1(Ile_Rzutow) 'Ustalenie wymiaru tablic
    ReDim Tablica_Wynikow_Kostka_2(Ile_Rzutow)
    ReDim Tablica_Wynikow_Par(Ile_Rzutow)

    Set ObiektExcelApp = New Excel.Application 'Uruchomienie Excela
    ObiektExcelApp.WindowState = xlMinimized 'Do paska zadan
    ObiektExcelApp.Visible = True 'Aplikacja Excela staje sie widoczna
    'Tworzenie nowego skoroszytu w biezacej aplikacji
    Set ObiektExcelSkoroszyt = ObiektExcelApp.Workbooks.Add
    Randomize 'Uruchamianie generatora liczb pseudolosowych

    'Ustalanie rozmiaru wykresu i jego zerowanie
    With MSChart1
        .Title = "Zliczenia kostek"
        .ChartData = Tablica_Wynikow_Par  'ustala macierz jako dane
```

```
    End With
End Sub


'Funkcja uzytkownika wywolywana w Command1_Click
Private Sub Rzucaj_kostkami()
    'Obliczanie losowych wynikow dla 50 rzutow 2 kostkami
    Dim n As Long 'Deklaracja wymuszona przez Option Explicit
    For n = 1 To Ile_Rzutow
        Tablica_Wynikow_Kostka_1(n) = Int(6 * Rnd()) + 1
        Tablica_Wynikow_Kostka_2(n) = Int(6 * Rnd()) + 1
        Tablica_Wynikow_Par(n) = Tablica_Wynikow_Kostka_1(n) _
                               + Tablica_Wynikow_Kostka_2(n)

        'Zliczanie wyrzucen danej liczby oczek
        'Tablica_Zliczen_Par nigdy nie jest zerowana, wiec kumulacja danych
        Tablica_Zliczen_Par(Tablica_Wynikow_Par(n)) = _
        Tablica_Zliczen_Par(Tablica_Wynikow_Par(n)) + 1
    Next n

    'Wykorzystanie funkcji Excela do lokalnych danych (tablic)
    'Argument funkcji w nastepnych liniach nie jest zakresem w arkuszu
    'tylko zwykłą lokalną tablica
    With ObiektExcelApp
    Label14.Caption = Str(.WorksheetFunction.Sum(Tablica_Wynikow_Kostka_1))
    Label15.Caption = Str(.WorksheetFunction.Sum(Tablica_Wynikow_Kostka_2))
    Label16.Caption = Str(.WorksheetFunction.Average(Tablica_Wynikow_Kostka_1))
    Label17.Caption = Str(.WorksheetFunction.Average(Tablica_Wynikow_Kostka_2))

    Label3.Caption = Str(.WorksheetFunction.Average(Tablica_Wynikow_Par))
    If Ile_Rzutow > 1 Then
        Label4.Caption = Str(.WorksheetFunction.Stdev(Tablica_Wynikow_Par))
    Else
        Label4.Caption = "- -"
    End If
    Label8.Caption = Str(.WorksheetFunction.Median(Tablica_Wynikow_Par))
    End With

    Command4.Font.Bold = True
    Command1.Font.Bold = False

    'Prezentacja tablicy zliczen na wykresie
    MSChart1.ChartData = Tablica_Zliczen_Par
End Sub


'Funkcja uzytkownika wywolywana w Command4_Click
'Teraz dla odmiany kopiowanie danych do Excela
'i wstawianie funkcji sumujacej w arkuszu
Private Sub Oblicz_w_Excelu()
    'Tworzenie obiektu arkusza nie jest niezbedne, ale wygodne
    Dim ObiektExcelArkusz As Excel.Worksheet
    Set ObiektExcelArkusz = ObiektExcelSkoroszyt.Sheets(1)

    With ObiektExcelArkusz

    'Kopiowanie danych i tworzenie funkcji w arkuszu
    Dim n As Long

    'Kopiowanie w petli (wolniejsze niz calej tablicy)
    'For n = 1 To Ile_Rzutow
    '    .Cells(n, 1).Value = Tablica_Wynikow_Kostka_1(n)
```

```
    '    .Cells(n, 2).Value = Tablica_Wynikow_Kostka_2(n)
    '    .Cells(n, 3).Value = "=A" & n & "+B" & n
    'Next n

    'Przygotowuje tablice i kopiuje ja w calosci
    Dim tablica_do_skopiowania() As Variant
    ReDim tablica_do_skopiowania(1 To Ile_Rzutow, 1 To 3)
    For n = 1 To Ile_Rzutow
        tablica_do_skopiowania(n, 1) = Tablica_Wynikow_Kostka_1(n)
        tablica_do_skopiowania(n, 2) = Tablica_Wynikow_Kostka_2(n)
        tablica_do_skopiowania(n, 3) = "=A" & n & "+B" & n
    Next n
    .Cells(1, 1).Resize(Ile_Rzutow, 3) = tablica_do_skopiowania

    'Wstawianie funkcji SUM/SUMA do arkusza
    .Cells(Ile_Rzutow + 1, 1).Value = "=SUM(A1:A" & Ile_Rzutow & ")"
    .Cells(Ile_Rzutow + 1, 2).Value = "=SUM(B1:B" & Ile_Rzutow & ")"
    .Cells(Ile_Rzutow + 1, 3).Value = "=SUM(C1:C" & Ile_Rzutow & ")"

    Label20.Caption = .Cells(Ile_Rzutow + 1, 1).Value
    Label21.Caption = .Cells(Ile_Rzutow + 1, 2).Value

    'Wstawianie funkcji AVERAGE/ŚREDNIA do arkusza
    .Cells(Ile_Rzutow + 2, 1).Value = "=AVERAGE(A1:A" & Ile_Rzutow & ")"
    .Cells(Ile_Rzutow + 2, 2).Value = "=AVERAGE(B1:B" & Ile_Rzutow & ")"
    .Cells(Ile_Rzutow + 2, 3).Value = "=AVERAGE(C1:C" & Ile_Rzutow & ")"

    Label22.Caption = .Cells(Ile_Rzutow + 2, 1).Value
    Label23.Caption = .Cells(Ile_Rzutow + 2, 2).Value

    'Formatowanie
    .Range("A" & Ile_Rzutow + 1 & ":C" & Ile_Rzutow + 2).Font.Bold = True
    .Range("C" & Ile_Rzutow + 1 & ":C" & Ile_Rzutow + 2).Font.Color = vbRed

    End With

    MsgBox ("Kopiowanie do Excela i obliczenia zakończone")

    Set ObiektExcelArkusz = Nothing
End Sub

'Przy zamknieciu aplikacji zamykany jest Excel
Private Sub Form_Unload(Cancel As Integer)
    ObiektExcelApp.DisplayAlerts = False 'Zapobiega pytaniu o zapisanie
    ObiektExcelApp.Quit
End Sub

'Procedura obslugujaca zmiane liczby w okinku edycyjnym - zmiana il. rzutow
Private Sub Text1_Change()
    Ile_Rzutow = Val(Text1.Text)
    If Ile_Rzutow > 0 Then
        ReDim Tablica_Wynikow_Kostka_1(Ile_Rzutow) 'Zmiana rozmiaru tablic
        ReDim Tablica_Wynikow_Kostka_2(Ile_Rzutow)
        ReDim Tablica_Wynikow_Par(Ile_Rzutow)
    End If
End Sub
```

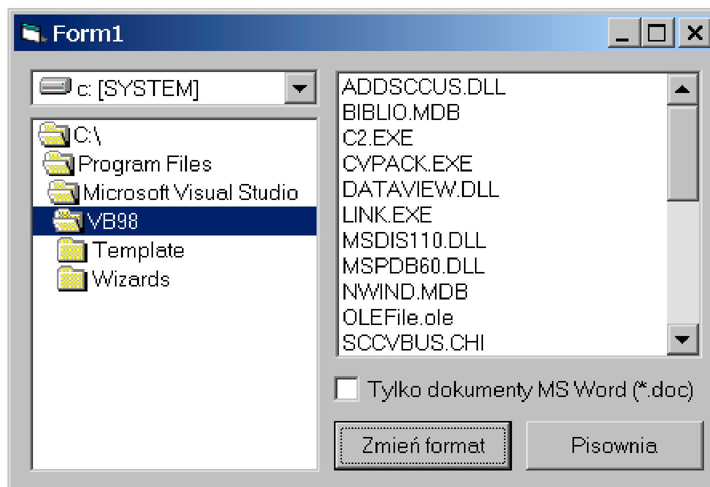
**Uwaga!** W drugiej linii kodu znajduje się wyrażenie `Option Base 1` ustalające, że tablice w bieżącym module indeksowane są nie od 0, jak jest domyślnie, a od 1. Konsekwentnie wszystkie pętle w programie powinny być indeksowane od 1. Jeżeli się ten szczegół zaniedba, to w obliczeniach pojawi się trudny do zlokalizowania błąd wiążący się z faktem, operacje na całych tablicach (np. obliczanie średniej za pomocą

funkcji Excela) uwzględniają wyraz o indeksie zero i wartości zero. Nie zmienia to oczywiście sumy, ale musi zmienić średnią i pozostałe wyniki. Sposobem na uniknięcie tych kłopotów jest zastąpienie deklaracji tablicy `Dim tablica(wymiar)` deklaracją `Dim tablica(1 To wymiar)`.

**Zadanie**
Dodać okno dialogowe SaveAs przy zapisie skoroszytu na dysk. Można wykorzystać albo CommonDialog.ShowSave albo funkcję Excela Application.GetSaveAsFilename.

## 3. Automation - wykorzystanie obiektów Microsoft Word 97

Ponieważ zasada współpracy z Wordem jest identyczna jak w przypadku Excela poprzestaniemy jedynie na ilustracji. Zmodyfikujemy dokumentu Worda zmieniając czcionkę w całym dokumencie oraz na uruchomimy funkcję Worda sprawdzania pisowni.




Na formie umieszczamy **DriveListBox** (Drive1), **DirListBox** (Dir1) i **FileListBox** (File1). Ich połączenie nie jest tak proste, jak w przypadku odpowiednich komponentów VCL. Należy napisać procedury zdarzeniowe reagujące na zmianę wybranej na listach pozycji:

```
'Zmiana wybranego dysku
Private Sub Drive1_Change()
    Dir1.Path = Drive1.Drive 'Zmiana dysku odswieza liste katalogow
End Sub

'Zmiana wybranego katalogu
Private Sub Dir1_Change()
    File1.Path = Dir1.Path 'Zmiana katalogu odsieza liste plikow
End Sub
```

Do formy dodajmy również CheckBox (Check1) ustalający filtr na listę plików dopuszczający tylko dokumenty Worda (*.doc). Zaznaczenie Check1 spowoduje włączenie filtru:

```
Private Sub Check1_Click()
    If Check1.Value = vbChecked Then
        File1.Pattern = "*.doc"
    Else
        File1.Pattern = "*.*"
    End If
End Sub
```

Przejdźmy do zasadniczego kodu programu. Są nim dwie niezależne procedury. Pierwsza uruchamia aplikację Worda i wczytuje wybrany w liście plików dokument, a następnie zmieniają format czcionki w całym

dokumencie na wybrany w okienku dialogowym. Druga procedura uruchamia w wybranym dokumencie sprawdzanie pisowni.

Procedura zmieniająca format czcionki:

```
Private Sub Command1_Click()
    If File1.FileName = "" Then Exit Sub 'Wyjscie jezeli nie wybrany plik

    'Deklarowanie obiektu i uruchamianie aplikacji Worda
    Dim WordObj As Word.Application
    Set WordObj = CreateObject("Word.Application")
    'WordObj.Visible = True 'Tu mozna ujawnic uruchomionego Worda

    'Deklaracja obiektu i otwieranie dokumentu
    Dim WordDoc As Word.Document
    Set WordDoc = WordObj.Documents.Open(File1.Path & File1.FileName)
    Form1.Caption = "Wybrany plik: " & File1.Path & File1.FileName

    'Pominieto obsluge bledow (przyklad obslugi w Excelu)

    'Wlasciwa akcja
    CommonDialog1.Flags = cdlCFBoth Or cdlCFEffects 'Konf. okna dialogowego
    CommonDialog1.ShowFont 'Pokazanie okna dialogowego
    With WordDoc.Range 'Przypisanie wybranej czcionki
        .Font.Bold = CommonDialog1.FontBold
        .Font.Name = CommonDialog1.FontName
        .Font.Size = CommonDialog1.FontSize
        .Font.Bold = CommonDialog1.FontBold
        .Font.Italic = CommonDialog1.FontItalic
        .Font.Underline = CommonDialog1.FontUnderline
        '.Font.Strikethru = CommonDialog1.FontStrikethru
        '.ForeColor = CommonDialog1.Color
    End With

    'Zapisanie i zamkniecie dokumentu, zamkniecie aplikacji
    WordDoc.Save
    WordDoc.Close
    WordObj.Quit


    'Zwolnienie zmiennej
    Set WordDoc = Nothing
    Set WordObj = Nothing

    MsgBox "Zmiany zostaly zachowane w pliku" 'Komunikat
End Sub
```

Procedura uruchamiająca sprawdzanie pisowni. Uruchomienie Worda i otworzenie dokumentu jest identyczne jak wyżej, więc pozostawię je bez komentarzy. Jedyna różnica, to fakt, że teraz Word jest widoczny. Moduł sprawdzania pisowni można uruchomić metodą dokumentu `WordDoc.CheckSpelling`.

```
Private Sub Command2_Click()
    If File1.FileName = "" Then Exit Sub

    Dim WordObj As Word.Application
    Set WordObj = CreateObject("Word.Application")
    WordObj.Visible = True

    Dim WordDoc As Word.Document
    Set WordDoc = WordObj.Documents.Open(File1.Path & File1.FileName)
```

```
    Form1.Caption = "Wybrany plik: " & File1.Path & File1.FileName

    'Wlasciwa akcja
    WordDoc.CheckSpelling

    WordDoc.Save
    WordDoc.Close
    WordObj.Quit

    Set WordDoc = Nothing
    Set WordObj = Nothing

    MsgBox "Zakonczono sprawdzanie pisowni"
End Sub
```

## 4. Visual Basic vs. Visual Basic for Applications

Nie wnikając w szczegóły można stwierdzić, że od wersji 5.0 Visual Basica oba języki są identyczne. Jest to niewątpliwa zaleta znajomości Visual Basic.
Aby zweryfikować tę zgodność na najprostszym przykładzie stworzymy makro w Excelu, do którego skopiujemy procedurę pobierającą informację o Excelu w przykładzie 2a).

Otwórzmy Excel. W menu Narzędzia, Marka, Makra ... . Należy w polu edycyjnym Nazwa makra wpisać dowolną nazwę i nacisnąć przycisk Utwórz, a następnie Edytuj. Otworzy się okno Visual Basica for Application (identyczne jak standardowe okno VB jednak zlokalizowane). W oknie edycyjnym znajdować się będzie procedura stanowiąca szkielet makra:

```
Sub VBvsVBA()
End Sub
```

Skopiujmy do niego zawartość naszej procedury tj.

```
    'deklaracja zmiennej typu obiekt Excel.Application
    'jezeli usuniemy deklaracje - zostanie zastapiona domyslna (Variant)
    Dim ObiektExcelApp As Excel.Application
    'Tu obiekt nie jest tworzony, a jedynie do zmiennej przypisywany jest
    'obiekt istniejacy (uruchomiona aplikacja Excela)
    Set ObiektExcelApp = GetObject(, "Excel.Application")
    'Informacja o aktywnym skoroszycie, arkuszu i komórce
    With ObiektExcelApp
        MsgBox "Aktywny zeszyt: " & .ActiveWorkbook.Name & Chr(10) & _
               "Aktywny arkusz: " & .ActiveSheet.Name & Chr(10) & _
               "Adres aktywnego pola: " & .ActiveCell.Address & Chr(10) & _
               "Wartość aktywnego pola: " & .ActiveCell.Value _
               , , "Informacje o aktualnie uruchomionej aplikacji Excela"
    End With
    Set ObiektExcelApp = Nothing
```

Po uruchomieniu makra pojawi się identyczne okienko z informacjami o Wordzie jak poprzednio.
Procedurę można skrócić ponieważ w uruchomionym Excelu znajduje się predefiniowany obiekt Application typu Excel.Application:

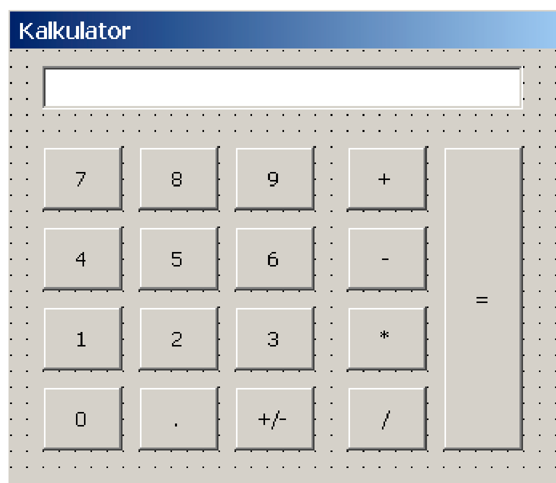
```
Sub VBvsVBA()
    With Application
            MsgBox "Aktywny zeszyt: " & .ActiveWorkbook.Name & Chr(10) & _
               "Aktywny arkusz: " & .ActiveSheet.Name & Chr(10) & _
               "Adres aktywnego pola: " & .ActiveCell.Address & Chr(10) & _
               "Wartość aktywnego pola: " & .ActiveCell.Value _
```

```
                , , "Informacje o aktualnie uruchomionej aplikacji Excela"
    End With
End Sub
```

## 5. Kalkulator – makro VBA osadzone w Excelu

Kalkulator będzie bardzo prosty – będzie wykonywał jedynie podstawowe operacje, ale dodanie innych, nawet bardzo złożonych operacji statystycznych będzie już tylko zwykłym powielaniem nakreślonego poniżej schematu.

Otwieramy Excel. Przechodzimy do menu Narzędzia, Makro, Edytor Visual Basic.
Aby utworzyć niezależne okienko kalkulatora należy z menu Wstaw edytora wybrać **UserForm**. Po kliknięciu formy pojawi się przybornik zawierający typowe kontrolki formularzy. Na formie należy umieścić przyciski odpowiadające cyfrom od 0 do 9, przycisk z przecinkiem, przycisk zmieniający znak liczby „+/−”, przyciski ze znakami operacji dodawania, odejmowania, mnożenia i dzielenia oraz przycisk wykonujący obliczenia „=” (podobnie jak na dołączonym wzorze formy). Za wyświetlacz ma posłużyć pole tekstowe (TextBox). Jeżeli go nie ma na przyborniku, można go dodać wywołując menu kontekstowe strony przybornika i wybrać pozycję Formanty dodatkowe ... . Na długiej liście dostępnych kontrolek ActiveX należy zaznaczyć **Microsoft Form 2.0 TextBox**.




Procedury zdarzeniowe związane z klawiszami 0 – 9 oraz przecinka są bardzo proste. Dołączają do łańcucha TextBox1.Text odpowiedni znak. Przykładowa procedura znajduje się poniżej:

```
Private Sub CommandButton1_Click()
TextBox1.Text = TextBox1.Text + "7"
End Sub
```

Przycisk zmieniający znak musi działać nieco inaczej. Dla ułatwienia nie operujemy na łańcuchu, a na przekonwertowanej liczbie. Do tego celu wykorzystujemy typowe dla BASICa funkcje `Val` konwertującą łańcuch na liczbę i `Str` działającą odwrotnie.

```
Private Sub CommandButton12_Click()
TextBox1.Text = Str(-Val(TextBox1.Text))
End Sub
```

Ponieważ okienko edycyjne jest tylko jedno, a podstawowe operacje algebry są dwuargumentowe musimy sobie zadeklarować zmienną przechowującą pierwszy argument podczas gdy wpisywany jest drugi:

```
Private pierwszy_argument As Double
```

Schemat korzystania z kalkulatora będzie typowy jak w większości kalkulatorów.

1) Po uruchomieniu kalkulatora edytowana będzie pierwsza liczba.
2) Po wciśnięciu przycisku operacji zapisujemy pierwszą liczbę do przygotowanej zmiennej i edytujemy drugą. Musimy przy tym zapamiętać jaką operację mamy wykonać.
3) Po naciśnięciu „=” wykonujemy zapamiętaną operację na dwóch argumentach – zapamiętanej zmiennej i zawartości pola edycyjnego. Pokazujemy wynik.
4) Naciśnięcie klawisza operacji pozwoli na dalsze działania z wynikiem jako pierwszym argumentem.

Z tego schematu wynika, że poza pierwszym argumentem musimy przechować także wykonywaną operację:
`Private dzialanie As Integer`
Ustalmy liczby naturalne odpowiadające działaniom:
1 – dodawanie
2 – odejmowanie
3 – mnożenie
4 – dzielenie
(można oczywiście zadeklarować wygodne w użyciu stałe). Wartość 0 zarezerwowana będzie na oznaczenie sytuacji, w której kalkulator oczekuje na wpisanie nowej liczby lub klawisza operacji (stan po naciśnięciu klawisza „=”)[8].

Naciśnięcie przycisku odpowiadającego np. operacji dodawania wiązać się powinno z procedurą:

```
Private Sub CommandButton13_Click()
dzialanie = 0 'ustalenie dzialania dodawania
pierwszy_argument = Val(TextBox1.Text) 'zapamietanie pierwszego argumentu
TextBox1.Text = "" 'wyczyszczenie wyswietlacza
End Sub
```

W pozostałych zmieniać się będzie jedynie linia odpowiedzialna za wybór działania.

Ostatecznie naciśnięcie klawisza „=” powoduje wykonanie odpowiedniego działania:

```
Private Sub CommandButton17_Click()
Dim drugi_argument As Double 'zmienna dla drugiego argumentu
Dim wynik As Double 'zmienna dla wyniku
drugi_argument = Val(TextBox1.Text)
Select Case dzialanie
    Case 1 'dodawanie
        wynik = pierwszy_argument + drugi_argument
    Case 2 'odejmowanie
        wynik = pierwszy_argument - drugi_argument
    Case 3 'mnozenie
        wynik = pierwszy_argument * drugi_argument
    Case 4 'dzielenie
        wynik = pierwszy_argument / drugi_argument
End Select
TextBox1.Text = Str(wynik)
dzialanie = 0
End Sub
```

Wykorzystano tutaj instrukcję wielokrotnego wyboru `Select Case`.

Aby ułatwić wywoływanie kalkulatora musimy stworzyć makro, które będzie pokazywało okno kalkulatora. Wywołujemy okno dialogowe makr (menu Narzędzia, Makro, Makra ...) i tworzymy makro o nazwie Kalkulator, które wywołuje metodę Show naszej formy:

```
Sub Kalkulator()
    UserForm1.Show 'UserForm1 to nazwa domyslna formy kalkulatora
End Sub
```

[8] zob. zadania do tego paragrafu

Można również do arkusza dodać przycisk (pasek narzędzi Formularze) i po pytaniu o skojarzenie z makrem wskazać na makro Kalkulator.

**Zadanie**
Uzupełnić program o obsługę błędów.

**Zadanie**
Po naciśnięciu klawisza „=” w polu tekstowym pokazywany jest wynik. Naciśnięcie klawiszy „0” – „9” powoduje dopisywanie do wyniku kolejnych cyfr. Zamiast tego powinna pojawić się nowa liczba będąca pierwszym argumentem kolejnej operacji. Wykorzystać wartość 0 zmiennej `działanie` ustalonej przy końcu procedury związanej z klawiszem „=”.

**Zadanie**
Dodaj przycisk „M” i zaimplementuj pamięć przechowującą jedną liczbę.
Dodaj przycisk „C” resetujący działanie kalkulatora.

**Zadanie**
Skopiować makro do Worda.