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

# Tworzenie aplikacji Windows
## Serwery automatyzacji OLE

# Ćwiczenia

Toruń, 14 grudnia 2002

Najnowsza wersja tego dokumentu znajduje się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad4_ole.pdf

Źródła opisanych w tym dokumencie programów znajdują się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad4_ole.zip

# I. Spis treści

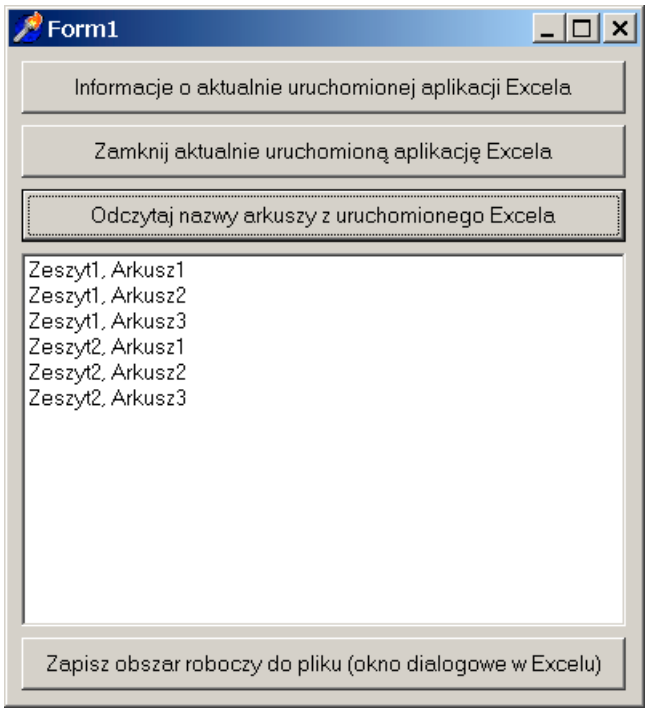
# II. Automatyzacja OLE

Automatyzacja (ang. *Automation*) jest elementem COM (ang. *Component Object Model*) standardu umożliwiającego aplikacjom udostępnianie klas (ActiveX) i obiektów (OLE) innym aplikacjom oraz w szczególności narzędziom programistycznym. Pierwszymi serwerami automatyzacji, a więc aplikacjami udostępniającymi obiekty były programy rozprowadzane razem z systemem Windows 3.1 (np. Paintbrush), a pierwszymi klientami – elementy pakietu Office (które np. do edycji obrazu korzystały z Paintbrusha). Standard OLE (ang. *Object Linking and Embedding* – łączenie i osadzanie obiektów) ewoluował razem z Windows do wersji 32-bitowej (OLE2) wzbogacając się między innymi o automatyzację. Mechanizm automatyzacji pozwala na udostępnianie obiektów innym aplikacjom, przez które mogą być w pełni kontrolowane.

Typowym przykładem serwera (dostarczyciela obiektów) automatyzacji jest Microsoft Office, a typowym zastosowaniem – gromadzenie danych i ich obróbka w arkuszach Excela czy wykorzystywanie szablonów Worda do tworzenia wydruków. Automatyzacja umożliwia niemal nieograniczony dostęp do komórek i funkcji Excela oraz każdego elementu dokumentu Worda. Jeżeli aplikacja jest serwerem automatyzacji, obiekty przez nią wystawione są dostępne i mogą być kontrolowane przez aplikacje klienckie. Naszym zadaniem jest właśnie tworzenie klientów automatyzacji[1].

**Uwaga!**
Spora część przedstawionych w pierwszym rozdziale przykładów jest wiernym tłumaczeniem kodów Visual Basica ze skryptu RAD3. Tam także można znaleźć bardziej wyczerpujące komentarze.

## 1. Współpraca aplikacji z uruchomioną aplikacją Excela




Klient automatyzacji może kontrolować uruchomioną już aplikację lub otworzyć nową jej instancję korzystając z funkcji VCL `GetActiveOleObject()` oraz `CreateOleObject()`. Obie mają bardzo prostą składnię, znacznie prostszą niż ich odpowiedniki z WinAPI. Ich jedynym argumentem jest nazwa klasy np. `GetActiveOleObject('Excel.Application')` żeby połączyć się z działającą aplikacją Excela lub

[1] O tworzeniu aplikacji serwerów można dowiedzieć się np. z *Delphi 4 Vademecum profesjonalisty* Teixeira i Pacheco, Wydawnictwo Helion Gliwice 1998.

`CreateOleObject('AcroExch.App')` żeby zainicjować aplikację Adobe Acrobat. Chyba najczęściej wykorzystywanymi obiektami automatyzacji są obiekty Microsoft Office – Delphi, na palecie Servers udostępnia nawet kontrolki ActiveX ułatwiające ich wykorzystanie – jednak pozbawione są wszelkiej dokumentacji. Jedynym sposobem jest analiza plików w katalogu Delphi `OCX\Servers`. Z kolei informacje o obiektach podłączanych za pomocą funkcji `GetActiveOleObject()` i `CreateOleObject()` można uzyskać na stronach MSDN, jeżeli udostępniane są przez serwery automatyzacji dostarczane przez Microsoft, lun ze stron innych producentów. Bardzo wygodnym narzędziem analizy obiektów automatyzacji OLE jest Object Browser w Visual Basicu, oczywiście o ile korzystamy także z konkurencji Borlanda.

Napiszmy prostą aplikację łączącą się z uruchomioną aplikacją Excel. W tym celu stwórzmy projekt Delphi zawierający cztery przyciski i ListBox podobnie jak na rysunku.

**„Informacje o aktualnie uruchomionej aplikacji Excela”** (przycisk `Button1`)

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ExlApp :Variant; //Deklaracja obiektu Excela jako wariant
  komunikat :String;
begin
//Tu obiekt nie jest tworzony, a jedynie do zmiennej przypisywany jest
//obiekt istniejacy (uruchomiona aplikacja Excela)
try
   ExlApp:=GetActiveOleObject('Excel.Application');
except
   on EOleSysError do
      begin
      ShowMessage('Błąd! Nie udało się podłączyć do Excela.');
      Exit;
      end;
end;

//Informacja o aktywnym skoroszycie, arkuszu i komórce
komunikat:='Aktywny zeszyt: '+ExlApp.ActiveWorkbook.Name +Chr(10)+
           'Aktywny arkusz: '+ExlApp.ActiveSheet.Name +Chr(10)+
           'Adres aktywnego pola: '+ExlApp.ActiveCell.Address +Chr(10)+
           'Wartość aktywnego pola: '+String(ExlApp.ActiveCell.Value);
MessageBox(Handle,PChar(komunikat),
            'Informacje o aktualnie uruchomionej aplikacji Excela',MB_OK);
ExlApp:=NULL;
end;
```

Najpierw podłączymy się do Excela funkcją `GetActiveOleObject()`. Ponieważ jej rezultat nie jest pewny (np. Excel może nie być uruchomiony) wykorzystaliśmy konstrukcję `try except`, aby wychwycić ewentualny wyjątek `EOleSysError`[2].

Jeżeli połączenie się powiodło pobieramy z obiektu informacje o nazwach aktywnego zeszytu i aktywnego arkusza oraz adresie i wartości aktywnej komórki. Następnie uzyskane informacje prezentujemy korzystając z metody `Application.MessageBox()`.

**Uwaga!**
W projektowaniu aplikacji korzystających z mechanizmu Automatyzacji OLE bardzo często przerywamy działanie tworzonej aplikacji bez zwalniania podłączonego obiektu. Często utrudnia to, lub nawet uniemożliwia, kolejne jej uruchomienia ze względu na działający w ukryciu egzemplarz serwera automatyzacji. W takiej sytuacji należy w Menadżerze zadań Windows na zakładce Procesy odnaleźć odpowiedni proces i go zakończyć.

[2] W dalszych przykładach pominiemy obsługę błędów w imię przejrzystości.

**„Zamknij aktualnie uruchomioną aplikację Excela”** (przycisk `Button2`)
Podobnie proste jak czytanie i zmienianie własności obiektu uzyskanego dzięki dobrodziejstwu automatyzacji jest wykonywanie jego metod. Można dla przykładu zamknąć uruchomioną aplikację Excela korzystając z jego metody `Quit`:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  ExlApp :Variant; //Deklaracja obiektu Excela jako wariant
begin
ExlApp:=GetActiveOleObject('Excel.Application'); //Przypisanie
if not VarIsEmpty(ExlApp) then
   ExlApp.Quit; //Wykorzystanie funkcji Excela
ExlApp:=NULL;
end;
```

Tym razem zamiast otaczać obsługą wyjątków moment podłączania do uruchomionej aplikacji Excela sprawdzamy czy połączenie się udało weryfikując przed wykonaniem jego metody, czy w reprezentujący go obiekt `ExlApp` rzeczywiście jest zainicjowany (powie nam to funkcja `VarIsEmpty()`).

**Odczytaj nazwy arkuszy z uruchomionego Excela** (`Button3`)
Zajrzymy do aktywnej aplikacji Exela i w ListBoxie umieścimy informacje o jego wszystkich zeszytach i arkuszach.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  ExlApp :Variant; //Deklaracja obiektu Excela jako wariant
  n,m :Integer;
begin
//Przypisanie
ExlApp:=GetActiveOleObject('Excel.Application');

//Czyszczenie listy
ListBox1.Clear;

//Czytanie nazw zeszytów i arkuszy
for n:=1 to ExlApp.Workbooks.Count do //Petla po skoroszytach
        for m:=1 to ExlApp.Workbooks.Item[n].Sheets.Count do //Arkusze
                ListBox1.Items.Add(ExlApp.Workbooks.Item[n].Name +', '+
                            ExlApp.Workbooks.Item[n].Sheets.Item[m].Name);

//Zwolnienie zmiennej
ExlApp:=NULL;
end;
```

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

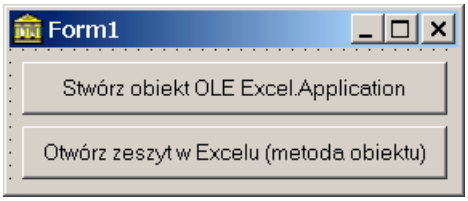
**„Zapisz obszar roboczy do pliku (okno dialogowe w Excelu)”** (`Button4`)
Korzystając z okienka „Zachowaj jako” Excela (metoda `GetSaveAsFilename()`), zachowamy aktywny zeszyt do pliku o wybranej przez nas nazwie. Alternatywnie można posłużyć się dialogiem z biblioteki VCL `TSaveDialog`:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  ExlApp :Variant; //Deklaracja obiektu Excela jako wariant
  NazwaPliku :String;
```

```
begin
ExlApp:=GetActiveOleObject('Excel.Application'); //Przypisanie
NazwaPliku:='';
while NazwaPliku='' do
      NazwaPliku:=ExlApp.GetSaveAsFilename(NazwaPliku);
ExlApp.ActiveWorkbook.SaveAs(FileName:=NazwaPliku);
//Trzeba pamietac o rozszerzeniu .xls
end;
```

## 2. Tworzenie obiektu Excel.Application

Równie proste jest jak podłączanie się do uruchomionej aplikacji jest tworzenie nowego egzemplarza tej aplikacji. Stwórzmy prostą aplikację z dwoma przyciskami:




Pierwszy otwiera pusty Excel, a drugi tworzy w nim nowy arkusz. Kompletnie pominięto obsługę błędów.

**„Stwórz obiekt OLE Excel.Application"**

```
procedure TForm1.Button1Click(Sender: TObject);
begin
ExlApp:=CreateOleObject('Excel.Application');
ExlApp.Visible:=True;
end;
```

**„Otwórz zeszyt w Excelu (metoda obiektu)"**

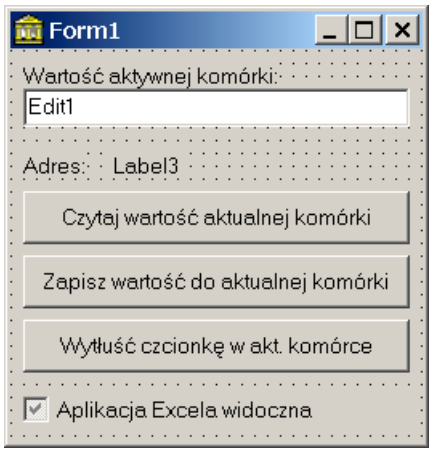
```
procedure TForm1.Button2Click(Sender: TObject);
begin
if not VarIsEmpty(ExlApp) then
   ExlApp.Workbooks.Add;
end;
```

## 3. Edycja komórek Excela

Ta aplikacja, w odróżnieniu od projektu z pierwszego paragrafu połączy się z Excelem przy uruchomieniu aplikacji i korzystać będzie z globalnego obiektu ExlApp, zamiast z lokalnych obiektów w każdej metodzie. Połączenie następuje w metodzie `FormCreate`:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
try
   ExlApp:=GetActiveOleObject('Excel.Application');
except
   on E: EOleSysError do
      begin
      ShowMessage('Blad inicjacji obiektu Excela ('+E.Message+')');
```

```
      Halt;
      end;
end;
end;
```




**„Czytaj wartość aktualnej komórki”**
Podobnie jak poprzednio metody mają cel jedynie demonstracyjny, więc są jak najprostsze:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Label3.Caption:=ExlApp.ActiveWorkbook.Name+'.'+
                ExlApp.ActiveSheet.Name+'.'+
                ExlApp.ActiveCell.Address;
Edit1.Text:=ExlApp.ActiveCell.Value;
end;
```

**„Zapisz wartość do aktualnej komórki”** i **„Wytłuść czcionkę w akt. komórce”**
Komunikacja z aplikacją może być dwustronna, tzn. z komórek można oczywiście czytać wartości, ale można je także modyfikować:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
ExlApp.ActiveCell.Value:=Edit1.Text;
end;
```

oraz

```
procedure TForm1.Button3Click(Sender: TObject);
begin
ExlApp.ActiveCell.Font.Bold:=True;
end;
```

**„Aplikacja Excela widoczna”**
Kolejną własnością obiektu `Excel.Application` jest `Visible`, która pozwala ukryć jego okno:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
ExlApp.Visible:=CheckBox1.Checked;
end;
```

# 4. Serwer automatyzacji OLE Internet Explorer

Zadaniem kolejnej mini aplikacji jest zilustrowanie łączenia z innym serwerem automatyzacji.

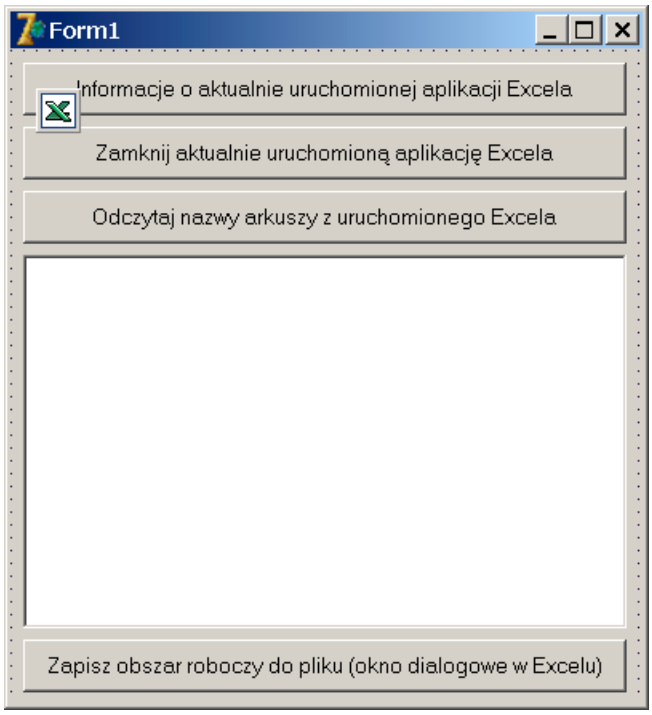

Jej działanie ani kod nie wymagają chyba komentarza.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
IE:=CreateOleObject('InternetExplorer.Application');
//IE:=GetActiveOleObject('InternetExplorer.Application');
IE.Visible:=true;
IE.Navigate(Edit1.Text);
end;
```

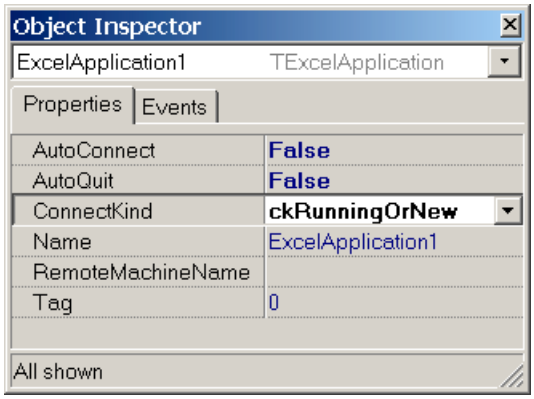
# III. Paleta serwerów automatyzacji

Na palecie Servers w Delphi Proffesional i Enterprise (od wersji 5) można znaleźć kontrolki ActiveX ułatwiające komunikację z serwerami automatyzacji. W szczególności udostępniają zdarzenia tych obiektów. Napiszmy ponownie dwa projekty z poprzedniego rozdziału korzystając z kontrolki `TExcelApplication`.

## 1. Współpraca aplikacji z uruchiomioną aplikacją Excela




Do identycznie jak poprzednio przygotowanej formy dodajemy komponent `TExcelApplication`. Metoda służąca do łaczenia z Excelem to `Connect` (można również ustawić własność `AutoConnect`). To czy ma być wykryta działająca aplikacja, czy stworzona nowa zależy od wartości własności `ConnectKind`. Co więcej komponent umożliwia wybranie zdalnego komputera, na którym chcemy wykryć lub uruchomić Excela.




Zmodyfikowana metoda projektu pobierająca informacje o uruchomionym Excelu może wyglądać następująco:

```
procedure TForm1.Button1Click(Sender: TObject);
var komunikat :String;
begin
ExcelApplication1.Connect;
```

```
with ExcelApplication1 do
begin
komunikat:='Aktywny zeszyt: '+ActiveWorkbook.Name +Chr(10)+
           'Aktywny arkusz: '+(ActiveSheet as _Worksheet).Name +Chr(10)+
           'Adres aktywnego pola: '+ActiveCell.Address[True,True,1,False,False] +Chr(10)+
           'Wartość aktywnego pola: '+String(ActiveCell.Value);
end;
MessageBox(Handle,PChar(komunikat),
              'Informacje o aktualnie uruchomionej aplikacji Excela',IDOK);
ExcelApplication1.Disconnect;
end;
```

W kodzie metody kontrolka `ExcelApplication1` zastąpiła zmienną `ExlApp` typu `Variant` z poprzedniej implementacji. Dostępny jest `ActiveWorkbook` typu `ExcelWorkbook` (równoważne z `_Workbook`) oraz `ActiveCell` typu `ExcelRange`. Natomiast własność `ActiveSheet` udostępniona jest za pomocą interface'u Delphi dla obiektów OLE `IDispatch`. Aby uzyskać dostęp do własności tego obiektu należy go wcześniej zrzutować na typ `ExcelWorksheet` (lub równoważne `_WorkSheet`): `(ActiveSheet as _Worksheet).Name`. Procedura powinna się zakończyć rozłączeniem z obiektem.

Pozostałe procedury należy zmodyfikować w analogiczny sposób.

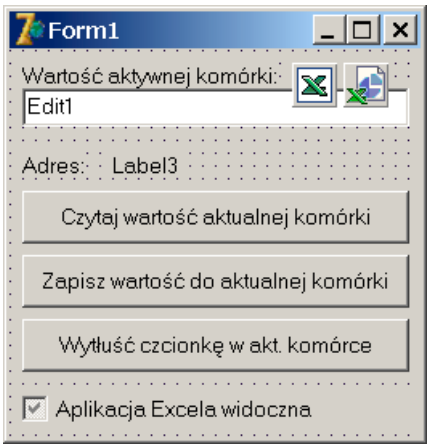
```
procedure TForm1.Button2Click(Sender: TObject);
begin
ExcelApplication1.Connect;
ExcelApplication1.Quit;
ExcelApplication1.Disconnect;
end;
```

oraz

```
procedure TForm1.Button3Click(Sender: TObject);
var n,m :Integer;
begin
ExcelApplication1.Connect;

//Czyszczenie listy
ListBox1.Clear;

//Czytanie nazw zeszytów i arkuszy
for n:=1 to ExcelApplication1.Workbooks.Count do //Petla po skoroszytach
       for m:=1 to ExcelApplication1.Workbooks.Item[n].Sheets.Count do //Arkusze
              ListBox1.Items.Add(ExcelApplication1.Workbooks.Item[n].Name +', '+
                   (ExcelApplication1.Workbooks.Item[n].Sheets.Item[m] as _Worksheet).Name);

ExcelApplication1.Disconnect;
end;
```

Analogicznie można zmodyfikować projekt z paragrafu 3 w poprzednim rozdziale. Tym razem do formy dodałem również kontrolkę `TExcelWorksheet`. Z jej zdarzeniami `OnSelectionChange` oraz `OnChange` wiążę metodę, która wywołuje z kolei metodę zdarzeniową związaną z naciśnięciem `Button1`. Ta metoda zdarzeniowa aktualizuje wyświetlany adres i wartość aktywnej komórki.

```
procedure TForm1.ExcelWorksheet1SelectionChange(ASender: TObject;
  const Target: ExcelRange);
begin
Button1Click(Self);
end;
```




## 2. Przykład współpracy aplikacji z Microsoft Word

Współpraca z edytorem tekstu dostępnym w Office za pomocą kontrolek Worda jest również prosta. Po połączeniu aplikacji z Wordem i otwarciu nowego dokumentu w reakcji na naciśnięcie pierwszego przycisku

```
procedure TForm1.Button1Click(Sender: TObject);
var NazwaPliku :OleVariant;
begin
NazwaPliku:=FileListBox1.FileName;

WordApplication1.Connect;
WordApplication1.Visible:=true;
WordApplication1.Documents.Open(NazwaPliku, EmptyParam, EmptyParam,
                                EmptyParam, EmptyParam, EmptyParam,
                                EmptyParam, EmptyParam, EmptyParam,
                                EmptyParam);
end;
```

możemy modyfikować dokument Worda. Tu na przykład modyfikowany jest zaznaczony fragment tekstu:

```
procedure TForm1.Button2Click(Sender: TObject);
var NazwaPliku :OleVariant;
begin
NazwaPliku:=FileListBox1.FileName;

if FontDialog1.Execute then
  begin
  WordApplication1.Connect;
  WordApplication1.Visible:=true;
  WordDocument1.ConnectTo(WordApplication1.Documents.Open(
              NazwaPliku, EmptyParam, EmptyParam, EmptyParam, EmptyParam,
              EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam));
  with WordDocument1.Range.Font do
    begin
    Bold:=Integer(fsBold in FontDialog1.Font.Style);
    Italic:=Integer(fsItalic in FontDialog1.Font.Style);
    Underline:=Integer(fsUnderline in FontDialog1.Font.Style);
    Strikethrough:=Integer(fsStrikeOut in FontDialog1.Font.Style);

    Name:=FontDialog1.Font.Name;
    Size:=FontDialog1.Font.Size;
    case FontDialog1.Font.Color of
      clBlack: ColorIndex:=wdBlack;
```

```
      clBlue:  ColorIndex:=wdBlue;
      clAqua: ColorIndex:=wdTurquoise;
      clLime : ColorIndex:=wdBrightGreen;
      clFuchsia: ColorIndex:=wdPink;
      clRed: ColorIndex:=wdRed;
      clYellow: ColorIndex:=wdYellow;
      clWhite: ColorIndex:=wdWhite;
      clNavy: ColorIndex:=wdDarkBlue;
      clTeal: ColorIndex:=wdTeal;
      clGreen: ColorIndex:=wdGreen;
      clPurple: ColorIndex:=wdViolet;
      clMaroon: ColorIndex:=wdDarkRed;
      clOlive: ColorIndex:=wdDarkYellow;
      clGray: ColorIndex:=wdGray50;
      clSilver: ColorIndex:=wdGray25;
      end;
    end;
  WordDocument1.Disconnect;
  end;
end;
```

Ze względu na różnice w implementacji klasy przechowującej wartość koloru (w tym przypadku koloru czcionki) konieczne jest tłumaczenie stałych określających kolory.

Wreszcie można wywoływać metody Worda, np. wywołanie sprawdzania pisowni:

```
procedure TForm1.Button3Click(Sender: TObject);
var NazwaPliku :OleVariant;
begin
  NazwaPliku:=FileListBox1.FileName;
  WordApplication1.Connect;
  WordApplication1.Visible:=true;
  WordDocument1.ConnectTo(WordApplication1.Documents.Open(NazwaPliku,
                          EmptyParam, EmptyParam, EmptyParam, EmptyParam,
                          EmptyParam, EmptyParam, EmptyParam, EmptyParam,
                          EmptyParam));
  WordDocument1.CheckSpelling;
  WordDocument1.Disconnect;
end;
```

**Uwaga!**
OLE Automation nie wyparł tradycyjnego osadzania i łączenia obiektów, czyli w środowisku 32-bitowym OLE2. Przykłady wykorzystania w tym celu komponentu `TOLEContainer` oraz użycie jego metod i własności można znaleźć w źródłach dołączonych do tego działu w katalogach rozpoczynających się od RAD4.ole.IV. W przeciwieństwie do serwerów są to dobrze udokumentowane komponenty VCL i dlatego pominąłem w tym skrypcie opis tych przykładów.