Drukuj Zamknij

XmlEdytor, czyli praktyczny opis Xml, DOM i kontrolki TreeView

2003-11-26 22:50:38
Paweł Matyjasek - Informatyka, Politechnika Szczecińska

Wprowadzenie

Jak zapewne zauważyliście skrót Xml (eXtensible markup language) znajduje się niemal w każdym artykule Microsoftu dotyczącym pakietu .NET. Pliki Xml są podstawową formą komunikacji z najnowszymi bazami danych, znajdują zastosowanie nawet przy zapisie plików projektu .NET (jak nie wierzycie, to sprawdźcie pliki *.csproj i *.resx). Co sprawia, że ten format zapisu robi coraz większą karierę i znajduje tyle zastosowań?

Oto niektóre powody:

- składnia Xml jest dobrze opisana i zaimplementowana w wielu rozwiązaniach
- Xml jest plikiem samoopisującym się tzn. użytkownik ma pełną swobodę w tworzeniu własnych struktur dokumentów
- Xml pozwala na szybką plików na np. strony internetowe za pomocą transformacji XSLT

Każdy dokument Xml ma strukturę drzewiastą tzn. mamy węzeł główny, który zawiera kolejne węzły itd. Każdy węzeł zapisany jest w taki sposób (zbliżony do HTML'a):

<węzeł atrybut1=”*wartość atrybutu*” atrybut2=”*wartość atrybutu2*” ...>

węzły potomne

</węzeł>

To, co odróżnia Xml od HTML to m.in. dbałość o prawidłowe zagnieżdżanie elementów. Utwórz sobie poniższy plik (nie jest prawidłowo uformowany) i spróbuj go otworzyć w IE:

<root>

<potomek>

</root>

<potomek>

SAX i DOM, czyli dwa podejścia do Xml

Przejdźmy do kwestii praktycznych. Chcemy otworzyć plik Xml i wyciągnąć z niego wszystkie dane. Mamy do wyboru dwa warianty: SAX i DOM.

SAX (Simple Api for Xml) jest najlepszym wyborem, jeśli odczytane dane chcesz upakować w swoje własne klasy i struktury. Reprezentuje go szybkość i niewielkie zużycie pamięci (aby poczytać więcej o użyciu tego interfejsu w .NET, sprawdź klasy XmlReader i XmlWriter). Krótko mówiąc otwierasz plik a parser SAX informuje Cię o odczytaniu elementu, atrybutu itd. aż do odczytania całego pliku.

DOM jest natomiast obiektową reprezentacją pliku Xml w pamięci. Nie musisz się martwić o tworzenie obiektów, DOM robi to za Ciebie. Głównymi wadami rozwiązania DOM są: mniejsza szybkość w stosunku do SAX i spore zapotrzebowanie na pamięć operacyjną (Microsoft twierdzi, że plik Xml w pamięci zajmuje 4 razy więcej miejsca niż na dysku, inne źródła twierdzą, że zapotrzebowanie na pamięć jest nawet 10 razy większe, ale o tym możecie się przekonać otwierając opisany w tym artykule projekt XmlEdytor). Ja wybrałem DOM, ponieważ do realizacji odczytu i zapisu nie potrzebujemy większych nakładów pracy.

Zanim przejdziemy do omawiania kodu zauważ, że aby zacząć zabawę z Xml'em musisz dodać referencję do System.Xml i dla ułatwienia pracy deklarację używania przestrzeni nazw System.Xml

Przeanalizujmy część procedury otwierania pliku (funkcja void menuItem3_Click(..) z Form1.cs):

```
.....
string plik = opendlg.FileName;
domdoc = new XmlDocument();     //implementacja DOM

//inicjujemy XmlTextReadera i przy jego pomocy
//napełnimy domdoc danymi
XmlTextReader XmlRdr = new XmlTextReader( plik );



//załaduj plik do obiektu XmlRdr
try
{
    domdoc.Load( XmlRdr );     //odczyt
    XmlRdr.Close();
    XmlRdr = null;     //teraz można się pozbyć XmlTextReader´a
    UstawPlik( plik );


    groupBox1.Enabled = true;


}


catch( Exception d)
{
    MessageBox.Show ("Błąd przy odczycie pliku " +d.Message ,          "XmlEdytor",
    MessageBoxButtons.OKCancel, MessageBoxIcon.Error );
    this.Close();//wyjdź:
}
.....
```

Okazuje się, że aby zapakować dane do obiektu XmlDocument (klasa reprezentująca obiekt DOM) potrzebujemy jednak TextReadera (zbliżony do SAX). Jakie wyjątki mogą nas tutaj zaskoczyć? Np. złe zagnieżdżenie węzłów w pliku, błąd odczytu pliku itd. Dalej rozpoczynamy procedurę:

```
void DOMnaDrzewo( XmlDocument domdoc, TreeView drzewo )
```

za pomocą której wyświetlimy zawartość domdoc na drzewo TreeView.

```
private void DOMnaDrzewo( XmlDocument domdoc, TreeView drzewo )
{
    drz.BeginUpdate();
    //zacznij od wstawienia głównego elementu drzewa
    //DomDocument dziedziczy po (System.Xml.XmlNode)
    //zwraca główny element drzewa
    //null, bo TreeView nie zawiera jeszcze węzłów
    WstawWezel( domdoc.DocumentElement, null );
    drz.EndUpdate();
}
```

Jak widać w procedurze tej nie ma nic szczególnie ciekawego oprócz wywołania funkcji drz.BeginUpdate() i WstawWezel(domdoc.DocumentElement, null). Pierwsza jest ściśle związana z szybkością wyświetlania zawartości pliku na kontrolce drz (TreeView), natomiast druga uruchamia rekurencyjne wstawianie węzłów do naszego drzewa (drz). Parametr domdoc.DocumentElement przekazuje funkcji główny węzeł pliku Xml.

Dalej mamy kolejne funkcje, które biorą udział we wstawianiu elementów do naszego drzewa:

```
private void WstawPotomkow( XmlNode xwezel, TreeNode twezel )
{
    foreach( XmlNode w in xwezel.ChildNodes )
        WstawWezel( w, twezel);
}

private void WstawAtrybuty( XmlNode xwezel, TreeNode twezel )
{
    foreach( XmlAttribute attrib in xwezel.Attributes )
        twezel.Nodes.Add( new xtnode( attrib ) );
}
private void WstawWezel( XmlNode xwezel, TreeNode twezel )
{
    if( drz.Nodes.Count == 0 )
    {//brak roota, wstaw roota, tu nie trzeba wstawiać atrybutów??
        xtnode dowstawienia = new xtnode(xwezel);
        drz.Nodes.Add( dowstawienia );

        //powstawiaj atrybuty:
        if( xwezel.Attributes != null )
            WstawAtrybuty( xwezel, dowstawienia );

        //jeśli ma potomków, to ich powstawiaj:
```

```
        if( xwezel.HasChildNodes )
        {

            WstawPotomkow( xwezel, dowstawienia );

        }
    }
    else    //normalny przypadek, root jest już dodany
    {
        //wstawić jako potomek węzła twezel
        xtnode dowstawienia = new xtnode( xwezel );
        twezel.Nodes.Add( dowstawienia );

        //powstawiaj atrybuty:
        if( xwezel.Attributes != null )
                WstawAtrybuty( xwezel, dowstawienia );

        if( xwezel.HasChildNodes )    //wstaw potomkow wezla
                WstawPotomkow( xwezel, dowstawienia );

    }

}
```

Tu chciałbym wyjaśnić pewien mechanizm, który może się wydać niektórym czytelnikom błędny: przekazywanie parametrów do powyższych funkcji przez wartość. Można zrealizować wstawianie przez referencję, ale należałoby się chwilkę pomęczyć nad pętlą foreach. Zapewne wielu czytelników uważa, że zrealizowana kopia elementu nie będzie działała poprawnie, twierdzenie takie jest błędne, co ilustruje dalej opisywana operacja usuwania węzła i zmiany jego wartości. W naszym przypadku obiekty typu XmlNode (podstawowy typ obiektu XmlDocument) są kopiowane na zasadzie *deep copy* (która kopiuje wszystkie referencje) w przeciwieństwie do *shallow copy* (nie kopiuje referencji). Więcej informacji znajdziecie w MSDN library szukając tych kluczy.

Podstawowe typy XmlDocument (DOM)

Przebrnęliśmy już przez operacje wstawiania węzłów do kontrolki drz (oczywiście typu TreeView), więc możemy przejść do nie mniej ważnego tematu: podstawowych typów implementacji DOM, czyli XmlDocument. Spróbuj otworzyć chociażby plik XmlEdytor.csproj.user, który znajdziesz w katalogu tego projektu (oczywiście za pomocą XMLEdytora). Po kliknięciu danego elementu w jednym z textboxów wyświetli się typ elementu, ewentualnie jego wartość (jeśli takową posiada) i przestrzeń nazw, w której się znajduje. Jak ta operacja została zrealizowana? Otóż do drzewa drz (TreeView) dodajemy obiekty klasy xtnode, które dziedziczą po obiekcie TreeNode(te zaś wstawiamy do kontrolki drz). Do składowych xtnode dodałem element typu XmlNode (xmldocel), który jest *głęboką kopią* odpowiedniego węzła w drzewie domdoc.

```
private void drz_AfterSelect(object sender,      System.Windows.Forms.TreeViewEventArgs e)
{
```

```
    //wyświetl parametry danego węzła:
    //typ:
    string temp;
    temp = ((xtnode)drz.SelectedNode).xmldocel.GetType().ToString();
    temp = temp.Substring (11, temp.Length -11);

    //wykasuj przedrostek: System.Xml.
    txtTyp.Text = temp;
    //i przestrzeń nazw:
    txtPrzestrzen.Text =        ((xtnode)drz.SelectedNode).xmldocel.NamespaceURI;

    //sprawdź, czy węzeł może mieć wartość:
    switch( ((xtnode)drz.SelectedNode).xmldocel.GetType().ToString() )
    {
        //te mogą:
        case "System.Xml.XmlAttribute":
        case "System.Xml.XmlProcessingInstruction":
        case "System.Xml.XmlComment":
        case "System.Xml.XmlText":
        case "System.Xml.XmlCDataSection":
            groupBox3.Enabled = true;
            break;

        //wszystkie inne nie mogą:
        default:
            groupBox3.Enabled = false;
            break;
    }


//umieść w najniższej grupie nazwę i wartość węzła (jeśli takowe posiada)
    if( ((xtnode)drz.SelectedNode).xmldocel.Value != null )
    {
        txtWartosc.Text = ((xtnode)drz.SelectedNode).xmldocel.Value;
    }
    else
    {
        txtWartosc.Text = null;
```

```
    }

    //dodaj parametr name (dla niektórych będzie to typ węzła):
    txtNazwa.Text = ((xtnode)drz.SelectedNode).xmldocel.Name;
}
```

Powyższy kod to procedura, którą wywołujemy wybierając jakiś węzeł wyświetlonego drzewa. Możemy zatem wywnioskować, że wszystkie elementy drzewa domdoc dziedziczą po klasie XmlNode (dzięki temu możemy dobierać się do ich nazw, wartości itd.). Ponadto niektóre z tych obiektów nie mogą mieć węzłów potomnych czy też list atrybutów (wszystkie znajdziecie w case'ach powyżej). Pozostałe właściwości innych typów na pewno rozpoznacie po lekturze kodu projektu.

Manipulacje węzłami

W procedurze dodawania nowych węzłów nie ma nic skomplikowanego: przed otwarciem okna DodajWezeldlg wywołujemy metodę public void Ustaw(string str, XmlDocument rdomdoc), która informuje nas o typie węzła, do którego chcemy dodać element (*str* - na jego podstawie wiemy, jakie elementy można wstawić do węzła a jakie nie można), drugi parametr jest zaś potrzebny przy wstawianiu węzła (nie możemy bezpośrednio używać konstruktora np. elementu XmlCDataSection, musimy najpierw użyć metody CreateCDataSection(...)). Uwaga: nazwa elementu czy też atrybutu nie może zawierać spacji i innych znaków typu '/', '\' itd. Jeśli chcesz wywołać wyjątek przy dodawaniu nowego węzła to zapraszam do lektury MSDN library, procedurę zabezpieczająca przed tym wyjątkiem pozostawiam do napisania Szanownemu Czytelnikowi.

```
....
else    //dodaj do zaznaczonego węzła:
{
xtnode nowyWezel = new xtnode( DodajWezeldlg.dodanyElement );
drz.SelectedNode.Nodes.Add( nowyWezel);
try
 {
    if( DodajWezeldlg.dodanyElement.GetType().ToString() ==          "System.Xml.XmlAttribute")
    {
                                    ((XmlElement)((xtnode)drz.SelectedNode).xmldocel).SetAttributeNode(
 (XmlAttribute)(DodajWezeldlg.dodanyElement) );
    }
    else
    {
                        ((xtnode)drz.SelectedNode).xmldocel.AppendChild( nowyWezel.xmldocel);
    }
}
catch(Exception d )
....
```

Powyższy kod to fragment procedury dodawania nowego węzła do drzewa domdoc (XmlDocument). Szczególną uwagę może zwrócić fakt, że atrybuty nie są traktowane jak potomkowie danego węzła, lecz jak jego własności. Poza tym Dopiero teraz (tzn. po wcześniejszym wywołaniu procedury Create... możemy dany węzeł wstawić na dane miejsce w drzewie dokumentu.

Pozostają nam jeszcze zmiana wartości węzła i jego usuwanie. Pierwsza jest banalna: po prostu wywołujemy odpowiednia metodę:

```
/czyli przycisk „Zmień”

private void button6_Click_1(object sender, System.EventArgs e)

{

    if( drz.SelectedNode == null )//nie wybrano węzła

    {

        MessageBox.Show ("Musisz wybrać element!" , "XmlEdytor",

                    MessageBoxButtons.OKCancel,                    MessageBoxIcon.Exclamation );

        return;

    }

    else

    {

        try

        {

        //zmień wartość węzła

            ((xtnode)drz.SelectedNode).xmldocel.Value =                    txtWartosc.Text;


        //i jeszcze tekst xtnode:

        drz.SelectedNode.Text =                                    ((xtnode)drz.SelectedNode).xmldocel.Name
+´=´+                    ((xtnode)drz.SelectedNode).xmldocel.Value;


        }

        catch(Exception d)

        {

            MessageBox.Show ("Błąd przy zmianie atrybutu węzła "                    +d.Message , "XmlEdytor",

            MessageBoxButtons.OKCancel, MessageBoxIcon.Error );

            //wyjdź:

            this.Close();

        }

    }

}
```

Natomiast usuwanie węzła musimy rozbić na trzy przypadki: usuwanie głównego węzła, usuwanie atrybutu i zwykłego węzła, co ilustruje poniższy kod:

```
public void Usun( XmlDocument domdoc) //z klasy xtnode

{


    //element nie może mieć rodzica
```

```
    if( xmldocel.ParentNode == null )
    {
        //kasujemy elementy roota oczywiście razem z rootem
        //this.Parent odnosi się do rodzica w drzewie drz
        if( this.Parent == null )        {
            domdoc.RemoveAll();
            domdoc.RemoveChild( xmldocel );
        }
        //a może węzeł jest atrybutem?
        else if( this.xmldocel.GetType().ToString() ==                    "System.Xml.XmlAttribute" ){
            System.Windows.Forms.TreeNode rodzic = this.Parent;
               ((xtnode)rodzic).xmldocel.Attributes.Remove(                    (System.Xml.XmlAttribute)
xmldocel );
            //skasuj atrybut z drzewa:
            Parent.Nodes.Remove( this );
        }
            //a może XmlDocument lub jakimś innym, który
            //nie posiada rodzica w drzewie domdoc (DOM)
        else if( (this.Parent != null) &&
            (this.xmldocel.ParentNode == null) )
        {
            System.Windows.Forms.TreeNode rodzic = this.Parent;
            ((xtnode)rodzic).xmldocel.RemoveChild( xmldocel );
            //skasuj atrybut z drzewa:
            Parent.Nodes.Remove( this );
        }
        /*else
        {    //wersja alternatywna, działająca dla wszystkich typów
            //z wyjątkiem atrybutów, ale może być czasochłonna
            //przy dokumencie o wielu węzłach
            System.Windows.Forms.TreeNode rodzic = this.Parent;

            //skasuj element z drzewa:
            domdoc.RemoveChild( xmldocel );
            Parent.Nodes.Remove( this );
        }*/
    }
    else
```

```
    {
    //wszystkie inne przypadki
    xmldocel.ParentNode.RemoveChild( xmldocel );
    }
}
```

Spowodowane jest to tym, że XmlNode nie ma metody np. Remove() dzięki której mógłby się sam usuwać, trzeba to zrobić za pomocą jego rodzica. Możemy napotkać na problem w przypadku tych typów: **XmlAttribute, XmlDocument, XmlDocumentFragment, XmlEntity, XmlNotation** nie posiadają właściwości Parent (zwracają jako null). Dodatkowo XmlAttribute nie jest uwzględniany jako węzeł (możemy do niego dotrzeć za pomocą własności xmldocel.Attributes, w której znajduja się atrybuty danego węzła typu XmlElement). Do jego usunięcia posługujemy się metodą:

XmlAttribute.Attributes.Remove(XmlAttribute *do_usunięcia*), którą stosujemy do węzła, do którego należy dany atrybut.

Zapis dokumentu Xml do pliku

```
//czyli menu Zamknij
private void menuItem4_Click(object sender, System.EventArgs e)
{
    //mamy pewność, że plik jest otwarty
    if( domdoc != null )
    {
        try
        {
            domdoc.Save( plikXml );
        }
        catch(Exception d)
        {
            MessageBox.Show ("Błąd przy zapisie pliku " +d.Message ,              "XmlEdytor",
            MessageBoxButtons.OKCancel, MessageBoxIcon.Error );
            //wyjdź:
            this.Close();
        }
    }

}
```

Jak widać zapis składa się z jednej funkcji – Save(string *plik*), pobierającej jeden parametr – nazwę pliku.

Uważny czytelnik zauważy, że za pomocą XmlEdytora nie jesteśmy w stanie dodać chociażby takich typów jak XmlEntity, XmlEntityReference itd. Projekt ten ma charakter szkoleniowy i dodanie tych możliwości skomplikowałoby niepotrzebnie kod. Czekam na wszelkie opinie, pomysły, udoskonalenia i pytania, zarówno co do kodu, jak i do tekstu artykułu.

Drukuj Zamknij