Drukuj Zamknij

Ułatwianie sobie życia z Visual Studio .NET

2003-11-24 12:25:21
Marcin Seredyński - Wydział Elektrotechniki i Elektroniki, Politechnika Łódzka

Wstęp

O czym jest ten artykuł?

Tak, jak widzicie po samym nieco zawiłym tytule, artykuł dotyczy Windows Forms i ich tworzenia w Visual Studio .NET. Przeznaczony jest dla początkujących, jak i dla zaawansowanych programistów. Zakładam, że Czytelnicy posiadają podstawową znajomość Visual Studio .NET, to znaczy wiedzą, w jaki sposób poruszać się po interfejsie programu oraz że znają podstawy C#. Myślę, że każdy z Was coś znajdzie dla siebie do przeczytania, postarałem się zawrzeć różne rady i ciekawostki w całym artykule.

Technika komputerowa idzie cały czas do przodu, ten, kto był ekspertem komputerowym pięć lat temu, dziś może okazać się laikiem, który nie wie, co za pomocą tego diabelstwa można zrobić. Zmienił się sprzęt, oprogramowanie, metody programowania, (co nas najbardziej interesuje), a także zmienił się stopień złożoności aplikacji. Aplikacje często tworzą wieloosobowe zespoły, gdzie jest ważne, by nie tworzyć od nowa gotowych już komponentów. Istotna jest dokumentacja, istotne jest projektowanie, lecz w tym artykule zajmiemy się tylko tworzeniem pewnego rodzaju „klocków", z których składa się aplikację.

Można łatwo wyobrazić sobie sytuację, w której jest programista (albo nawet cały zespół), szef i klient. Wiadomo, że programista to z reguły leń i chce oszczędzić sobie pracy, wiadomo, że szef to drań, który oczekuje od programisty na efekty jego pracy, a od klienta na pieniądze, zaś klient to dusigrosz, który chce zapłacić jak najmniej za oprogramowanie, które zamówił. W związku z tym, że wiedza ta jest powszechna – powstała idea szybkiego tworzenia oprogramowania – aby nie musieć wszystkiego pisać od początku i umożliwić łatwiejsze wykorzystywanie gotowych elementów. Szybkość pracy zwiększa jej efektywność, więc programista ma mniej na głowie dzięki automatyzacji tej pracy, szef nie musi poganiać programisty, a klient otrzymuje szybciej oprogramowanie, a więc i taniej (jeśli pomyśleć o wynagrodzeniu za godzinę pracy, które szef musi doliczyć do ceny produktu). Wszyscy dzięki automatyzacji, wspomaganiu i przyspieszeniu pracy są szczęśliwsi, a Wy, jeśli Wasz Wydział posiada licencję MSDNAA, macie dostęp do oprogramowania, które umożliwia wygodne tworzenie aplikacji – do Visual Studio .NET.

Jak posługiwać się tym artykułem?

Czyli co zrobić, żeby się nie narobić?

Czytajcie i pomóżcie mi pisać aplikację (to znaczy piszcie ją sami i patrzcie, czy nie ma błędów). Nie zamieściłem kompletnego kodu źródłowego, żeby Was zachęcić do dotknięcia klawiatury, a nie tylko ściągnięcia plików i przejrzenia ich zawartości. Z autopsji wiem, że to, co się samemu zrobi – zapamiętuje się najlepiej. Dodatkowo w cały artykuł wplecione są porady dla programistów, które mam nadzieję, że ułatwią Wam życie. Zaczynamy!

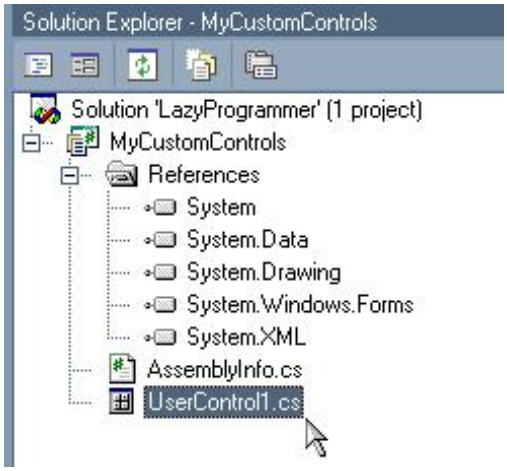
Tutorial

Czyli samouczek

Przede wszystkim, żeby zacząć pracę, należy uruchomić Visual Studio .NET. Wybieramy w menu **File > New > Blank Solution...** i wpisujemy nazwę nowego rozwiązania (solution). Ja wybrałem wiele mówiącą nazwę **LazyProgrammer**. Warto stworzyć sobie rozwiązanie, zamiast od razu pakować się w projekt, a to dlatego, że będzie można w przyszłości stworzyć sobie w ramach tego samego rozwiązania kolejne projekty. Taka organizacja pracy pozwala by aplikacje i biblioteki oraz zasoby były osobnymi projektami, ale jednocześnie umożliwia łatwiejsze użytkowanie każdego komponentu w innych projektach.

Po utworzeniu nowego rozwiązania należy stworzyć nowy projekt. W **Solution Explorerze** można kliknąć prawym przyciskiem na nazwę stworzonego rozwiązania i wybrać **Add > New Project**. Wybieramy jeden spośród dostępnych języków programowania (w tym przykładzie użyjemy C#) oraz **Windows Control Library** jako typ projektu. Jako nazwę nowego projektu wpisałem **MyCustomControls**. Tutaj komputer trochę pomieli twardym dyskiem (chyba że nie jest to Wasze pierwsze uruchomienie Visual Studio .NET) i wyświetli okienko o nazwie **UserControl1.cs [Design]**.

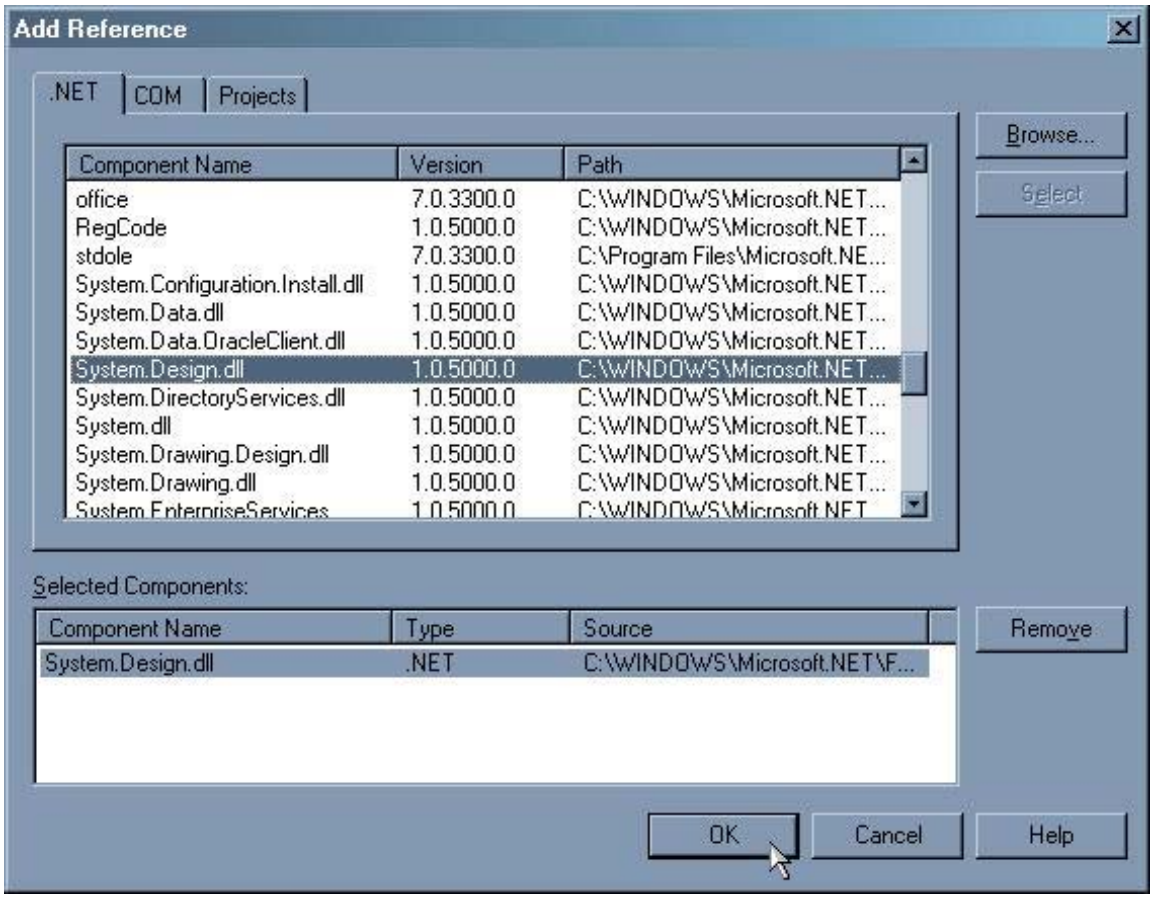
Ponieważ chcemy czegoś się nauczyć, a nie tylko wyklikać – skasujemy pliki, które powstały w rozwiązaniu. Aby tego dokonać, otwieramy **Solution Explorer**, gdzie będziemy mogli zaznaczyć pliki, a następnie wybrać, co chcemy z nimi zrobić (Kasować! Kasować!).



Rysunek 1 - Solution Explorer

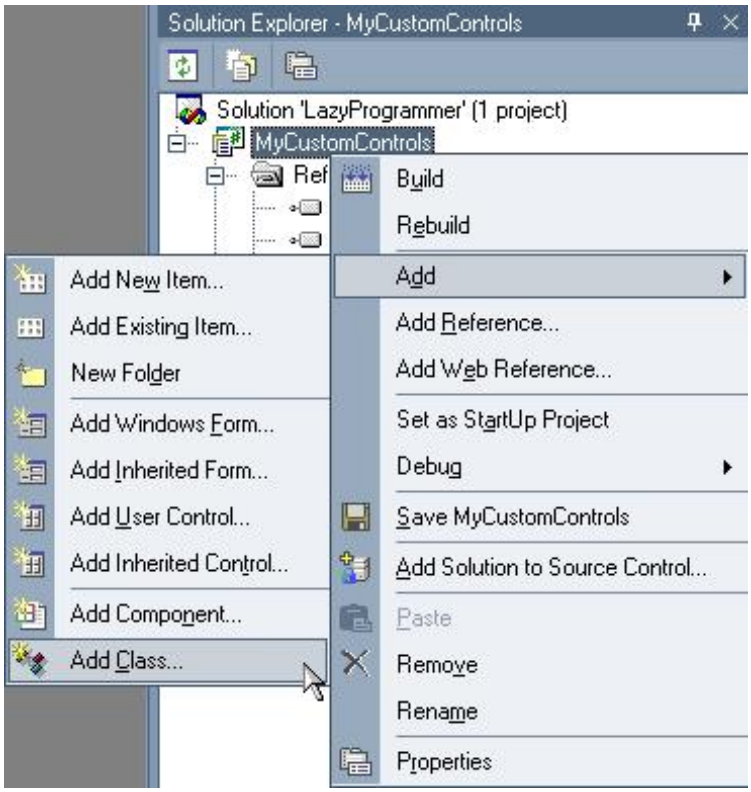
Warto w tym momencie wspomnieć o tym, co zostało wygenerowane przez Visual Studio .NET do tej pory. Otóż – stworzyliśmy rozwiązanie (solution), o nazwie **LazyProgrammer**, które zawiera jeden projekt (patrz rys. 1). Projekt nazywa się **MyCustomControls**. Projektów może być oczywiście kilka w obrębie jednego rozwiązania, ale póki co zajmiemy się tym jednym. W projekcie zawarte są dwa pliki, **AssemblyInfo.cs** i plik **UserControl1.cs**. W pliku automatycznie wygenerowanym przez Visual Studio .NET - **UserControl1.cs** – znajduje się definicja naszej własnej kontrolki. Jest ona nam zupełnie niepotrzebna, ponieważ chcemy pisać wszystko od podstaw. W pliku **AssemblyInfo.cs** znajduje się opis naszej aplikacji, lub biblioteki (w zależności od tego, co będziemy tworzyć). W detale dotyczące tego pliku nie będę się zagłębiał, to temat na inny artykuł. Możemy spokojnie usunąć wspomniane pliki, bo nie będą one nam potrzebne.

W sekcji **References** zapisane są odwołania do innych komponentów dostępnych w systemie. Z punktu widzenia programisty odwołania te są wyświetlane w **Solution Explorerze** jako przestrzenie nazw. Warto w tej chwili już dodać referencję do przestrzeni nazw **System.Design**. W **Solution Explorerze** klikamy na **References** prawym przyciskiem myszy i wybieramy **Add Reference...**, po czym znajdujemy odpowiedni plik z przestrzenią nazw, klikamy OK.



Rysunek 2 - Dodawanie referencji do bibliotek z poziomu Visual Studio .NET

Pora stworzyć własną klasę i trochę o niej opowiedzieć.



Rysunek 3 - Dodawanie klasy do projektu z poziomu Solution Explorera

Własną klasę dodajemy do projektu w sposób przedstawiony na rys. 3 – prawym przyciskiem na projekcie, wybieramy **Add > Add Class...** i w okienku dialogowym wpisujemy nazwę klasy. Nazwijmy klasę **TextPanel.** Po zaakceptowaniu nazwy klasy Visual Studio znów momencik pomieli dyskiem i wypluje na ekran kod źródłowy utworzonej klasy. Wszystkiego, co ten kod robi omawiać chyba nie musimy?

To, co nas interesuje w tej chwili, to dodanie jednej linijki. Zamieszczam screenshota, bo przypomniało mi się coś ciekawego.

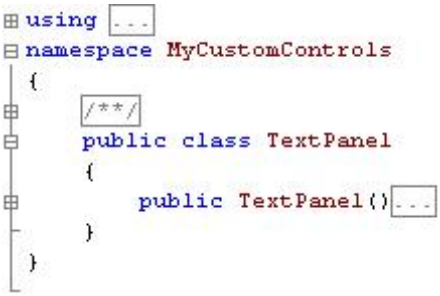
```
using System;
using System.Design;

namespace MyCustomControls
{
    /// <summary>
    /// Summary description for TextPanel.
    /// </summary>
    public class TextPanel
    {
        public TextPanel()
        {
            //
            // TODO: Add constructor logic here
            //
        }
    }
}
```

Rysunek 4 - Outlining (przed zastosowaniem)

Tak, jak widzicie – dopisałem jedną linijkę. Niezły ze mnie programista, co? Dopiszcie sobie jeszcze poniżej, tam gdzie kursor na obrazku:

```
using System.ComponentModel;

using System.Windows.Forms;

using System.Drawing;

using System.Drawing.Drawing2D;
```

Powinno się przydać. Chciałbym Wam coś pokazać - możliwość zwijania kodu (outlining). Po lewej stronie kodu są takie linie oraz kwadraty z minusami lub plusami. Po prostu kliknijcie na którymś kwadracie i sprawdźcie, co się dzieje.



Rysunek 5 - Outlining (po zastosowaniu)

Tak, jak ogólnie wiadomo, w C# nad klasą pracujemy tylko w jednym pliku. Klasa może zajmować tylko jeden plik, nie rozdzielamy pracy tak jak w C++ na plik nagłówkowy i plik z implementacją klasy. W C# podział na nagłówek i implementację nie istnieje. Outlining zwiększa przejrzystość kodu podczas jego edycji – jeśli pracujecie nad wielkimi klasami – możecie sobie ograniczyć widoczność elementów, które Was nie interesują. Bardzo przydatne. Można zamknąć część kodu pomiędzy znacznikami #region NazwaRegionu i #endregion, powstanie w ten sposób możliwość zwinięcia w ten sposób oznaczonego tekstu.

Pora zająć się dziedziczeniem. Jeśli piszemy klasę, która w zasadzie robi to samo, co potrafi inna klasa, „tylko że moja robi to lepiej", bardzo dobrze jest skorzystać z cudu techniki, który nazywa się dziedziczeniem. Pozwala ono na wykorzystanie istniejących już funkcji klasy, po której dziedziczymy. Ponieważ wymarzyłem sobie, że moja kontrolka będzie potrafiła robić różne cuda z tekstem (nikt Wam nie broni coś innego przetestować), a zostanie to zamknięte w panel, żeby ładnie wyglądało – dziedziczymy po klasie **System.Windows.Forms.Panel**, w sposób taki, jak poniżej podany (należy zmienić jedną linijkę w kodzie):

```
public class TextPanel : System.Windows.Forms.Panel
```

Dzięki temu nasza klasa **TextPanel** zawiera w sobie już funkcjonalność klasy **Panel** z przestrzeni nazw **System.Windows.Forms**. To jest pierwszy krok do lenistwa, które chcemy osiągnąć. OK, wszystko pięknie, tylko że w tej chwili ta klasa nie różni się od swojej klasy bazowej (czyli tej, po której dziedziczy). Trzeba pomyśleć, co klasa powinna robić. Powiedzmy, że pozwoli użytkownikowi na wybranie czcionki, jej wielkości, oraz kąta, pod jakim będzie pisany tekst zawarty w tworzonym przez nas panelu. Myślę, że to jako przykład wystarczy. Zamieszczę kawałek kodu źródłowego, który powinien powiedzieć wszystko za siebie:

```
public class TextPanel : System.Windows.Forms.Panel

{

    /// <summary>

    /// Text string to be displayed

    /// </summary>

    private System.String textString;
```

```
        /// <summary>
        /// Rotation of the text (in degrees)
        /// </summary>
        private System.Single textRotation;

        public TextPanel()
        {
            //
            // TODO: Add constructor logic here
            //
        }
}
```

Przyjrzyjmy się jeszcze konwencji nazewniczej. Jak wiadomo, klasy potrafią same zabezpieczać swoje dane i metody, uznać je za prywatne i nie dać nikomu do nich dostępu. Takie właściwości klasy (czyli po prostu dane) nazywane są w notacji **camelCasing**. Czyli pierwsza litera mała, a każda litera kolejnego słowa będącego częścią nazwy – duża. To jest konwencja notacyjna zalecana przez Microsoft do nazywania właściwości (zmiennych) prywatnych w klasie. Inaczej będą nazywane zmienne publiczne, w notacji **PascalCasing**. Warto chyba tutaj wspomnieć, że w całym .NET Frameworku są stosowane właśnie takie konwencje nazewnicze. Dobrze jest trzymać się tych zasad, by kod jednolicie i ładnie integrował się z klasami bazowymi Frameworka. Ponadto przy stosowaniu notacji camelCasing i PascalCasing dla pól prywatnych i publicznych na pierwszy rzut oka można już widzieć w kodzie, do jakiego rodzaju pola odwołujemy się. To chyba najlepszy argument na ich stosowanie.

W tym miejscu warto zatrzymać się na dłużej i wspomnieć więcej o nazewnictwie. Dobrą praktyką jest nazywanie zmiennych w sposób znaczący, to jest taki, żeby z samej nazwy zmiennej wiedzieć od razu, co zmienna przechowuje. Unikajmy więc nazw zmiennych takich, jak X2, Aa, abc, j23, ponieważ sama nazwa zmiennej nie mówi nic na temat danych, które przechowuje. Najlepiej stosować nazwy zmiennych takie jak przykładowo: sqlConnection (widać, że to prywatna zmienna - połączenie z serwerem SQL), albo SelectFilesDialog (publiczna zmienna - okienko dialogowego do wyboru plików).

Pora teraz na wyjaśnienie, co oznacza poniższy fragment kodu:

```
/// <summary>
/// Text string to be displayed
/// </summary>
private System.String textString;
```

Komentarz, który widzicie powyżej, różni się od standardowego tym, że zapisany jest za pomocą trzech slashy. W ten sposób zapisane komentarze będą później służyły do wygenerowania dokumentacji klas w formacie XML. Tag **<summary>** służy do zawarcia w nim, przed opisywanym elementem, krótkiego opisu, do czego ten element służy – na przykład opisania, że jest to łańcuch znaków, który będzie wyświetlany. Istnieje jeszcze kilka innych tagów, które najlepiej poznać przez wpisanie trzech slashy przed metodą, zmienną lub klasą. Na przykład przy wpisaniu takiego komentarza przed metodą w klasie – Visual Studio.NET automatycznie wygeneruje tagi dla opisu (<summary>), parametrów (<parameters>) i wartości zwracanej przez metodę (<returns>).

Myślę, że dobrze wszystko jest opisywać „dla przyszłych pokoleń" programistów, którzy będą korzystać z Waszej pracy w przyszłości, lub podczas pracy przy wieloosobowym projekcie. W tym też jest czysty egoizm – jeśli samemu komentuje się dobrze swój kod – łatwo połapać się w nim po przerwie w pracy nad nim. Zdarza się często tak, że programista przegląda swój kod, nad którym pracował wcześniej i chce dopracować go, coś dodać, coś zmienić i traci czas na odpowiedzenie sobie na pytanie: „O co mi tutaj właściwie chodziło?". Sam komentuję i nazywam zmienne po angielsku – ze względu na większą przejrzystość i jednolitość kodu podczas jego czytania. Ponadto większość dostępnego powszechnie kodu jest opisywana w języku angielskim i spójniej wygląda kod opisywany w jednym języku.

Ok, skoro mamy właściwości prywatne naszego komponentu, których nie można zmienić z zewnątrz... Dlaczego właściwie są prywatne? Jak w takim razie je można spoza klasy zmienić? A jest na to metoda – właściwości i akcesory, pozwalające ustawić metody zapisu i odczytu zmiennej.

Do każdej właściwości klasy w C# (i w pozostałych językach obsługiwanych przez Visual Studio .NET) można stworzyć tak zwane akcesory (**get** i **set)**, które pozwalają odpowiednio na odczyt i zapis do tej zmiennej. Można sobie wyobrazić nawet napisanie „interfejsu", który pozwala na podanie liczby arabskiej (na przykład 123), która zapisze do odpowiedniego stringa wartość rzymską „CXXIII", który będzie tę wartość przechowywał. Można też w drugą stronę – pobrać jako liczbę arabską wartość stringa określonego przez wartość rzymską. Zastosowanie raczej niepotrzebne, ale chyba obrazuje jedną z możliwości wykorzystania akcesorów. Innym przykładem może być użycie akcesora **set** w obiekcie koszyka zakupów w taki sposób, by od razu po zmianie ceny produktu zmieniał całkowitą sumę wartości produktów w koszyku. Warto zainteresować się właściwościami i akcesorami, szczególnie programistom, którzy po jednym dniu zmienili swój profil z C++ na C#, faktycznie nie zapoznając się z wieloma udogodnieniami, jakie daje C#. Poniżej fragment kodu:

```
public System.Single TextRotation
{
     get
     {
          return this.textRotation;
     }
     set
     {
          if(textRotation != value)
          {
               this.textRotation = value;
               this.Invalidate();
          }
     }
}
```

To, co dzieje się w tym fragmencie kodu, w akcesorze przy metodzie set, to sprawdzenie, czy wartość podana (**value**) różni się od wartości aktualnie przechowywanej w zmiennej **textString**. Jeśli wartości różnią się od siebie – zapisywana jest nowa wartość i kontrolka jest informowana o zmianie swojego stanu za pomocą metody **Invalidate()**. Istotne jest wywołanie tej metody, żeby kontrolka automatycznie „narysowała się" od nowa. Przy okazji kolejna złota myśl – warto wpisywać **this** przed odwołaniem się do zmiennej czy metody w klasie. A to dlaczego? Nie mamy z tego zysku w szybkości działania programu, ale mamy zysk podczas programowania. Po napisaniu **this** i kropki, IntelliSense w Visual Studio .NET będzie podpowiadał nam tylko to, co może zostać użyte w kontekście klasy – unikamy w ten sposób pomyłek i literówek.

Warto jeszcze zwrócić uwagę na linijkę:

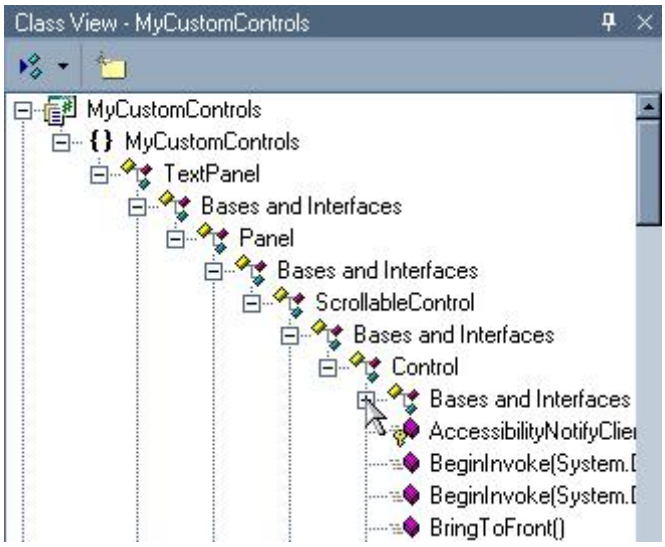
```
public System.Single TextRotation
```

Dlaczego napisałem ją w taki sposób? Dobrą szkołą pisania klas (które mają być później używane w większym projekcie) jest używanie pełnej „ścieżki" przestrzeni nazw, po której dostajemy się do klasy, aby zredukować szansę złego odwołania się. Można wyobrazić sobie sytuację, w której mamy zadeklarowane użycie dwóch przestrzeni nazw, które mają zdefiniowaną klasę **Single**, kompilator w przypadku, gdyby nie została podana pełna ścieżka dostępu do klasy, nie miał by jasności, do której z przestrzeni nazw się odwołać. Dlatego trzeba mu pomóc.

A teraz dam Wam chwilkę czasu na napisanie odpowiednich akcesorów dla pozostałych właściwości naszej

klasy, w sposób podobny, jak przedstawiłem...

Już? Super. Wszystko jest przygotowane, teraz postaramy się wyświetlić zawartość kontrolki. W tym celu posłużymy się **Class View**, czyli podglądem klas. **Class View** to zakładka w Visual Studio .NET, za pomocą którego możemy przeglądać hierarchię przestrzeni nazw i klas, w postaci drzewa. Przejdźmy przez to drzewo podobnie jak na ilustracji poniżej.



Rysunek 6 - Class View, hierarchia przestrzeni nazw i klas

I odnajdujemy metodę **OnPaint**, którą następnie przeciążamy – prawym przyciskiem na nazwie metody w liście, **Add > Override**. Visual Studio .NET powinien utworzyć kawałek kodu, który znajdziecie poniżej.

```
protected override void OnPaint(PaintEventArgs e)
{
     // TODO:  Add TextPanel.OnPaint implementation
     base.OnPaint (e);
}
```

Powyższy kod wygenerowany został przez Visual Studio.NET. Należy tylko teraz dodać samą implementację rysowania zawartości komponentu. Oczywiście przedtem należy wywołać metodę **OnPaint()** klasy, po której dziedziczymy, żeby odpowiednio narysował się cały „podkład" pod obiekt naszej klasy. To już załatwił za nas edytor. Tylko klepać...

```
protected override void OnPaint(PaintEventArgs e)
{
     base.OnPaint (e);

     e.Graphics.RotateTransform(this.textRotation);

     System.Drawing.SolidBrush brush = new System.Drawing.SolidBrush(this.ForeColor);

     e.Graphics.DrawString(this.textString, this.Font, brush,
          new System.Drawing.RectangleF(0.0f, 0.0f, (float)this.Width, (float)this.Height));
}
```

Nie będę zagłębiać się w detale powyższego kodu, bo nie jest to tematem tego artykułu. Kod rysuje

zawartość naszej kontrolki. Wspomnę tylko, że **this.ForeColor** i **this.Font** są dziedziczone z klasy **System.Windows.Forms.Panel** i nie trzeba było ich definiować wcześniej. To też ułatwi ustawianie w Designerze (edytorze formularzy) właściwości naszego panelu, skupimy je w jednym miejscu. W tym momencie już mamy własną kontrolkę, gotową do użycia w Designerze.

To znaczy prawie gotową... Trzeba jeszcze nowo dodane właściwości klasy opisać i przypisać je do odpowiedniej kategorii i oczywiście pozwolić Visual Studio .NET na wyświetlanie nowej kontrolki w Toolboxie. Wszystko, co trzeba zrobić, aby to umożliwić, to dopisanie paru linijek w kodzie.

```
[ToolboxItem(true)]

public class TextPanel : System.Windows.Forms.Panel
```

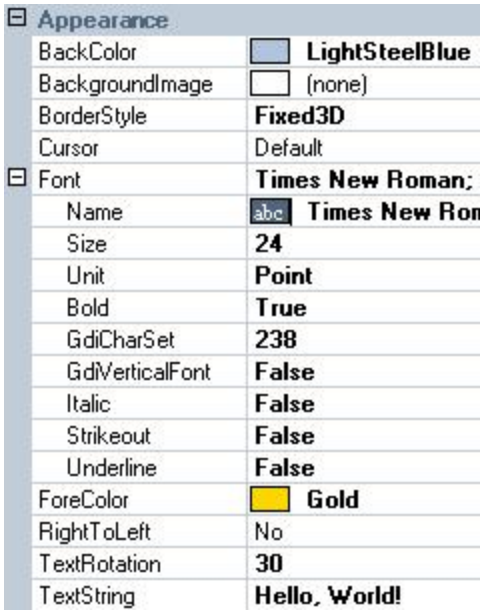
Nad definicją klasy należy umieścić tylko jedną linijkę, a kontrolka, którą opisuje dana klasa będzie dostępna do dodania w Toolboxie. W nawiasie przy **ToolboxItem** podaje się parametr **true**, pomimo że Visual Studio .NET automatycznie przyjmuje taki właśnie parametr dla każdej nowo stworzonej klasy. Jednak dobrze jest jawnie określić to dla każdej swojej kontrolki - lepiej co do pewnych spraw mieć pewność, niż liczyć na to, że już wcześniej została ustawiona odpowiednia właściwość. Jeszcze dwie małe rzeczy do zmiany:

```
[Category("Appearance"),

Description("Text to be displayed in a control")]

public System.String TextString
```

i...

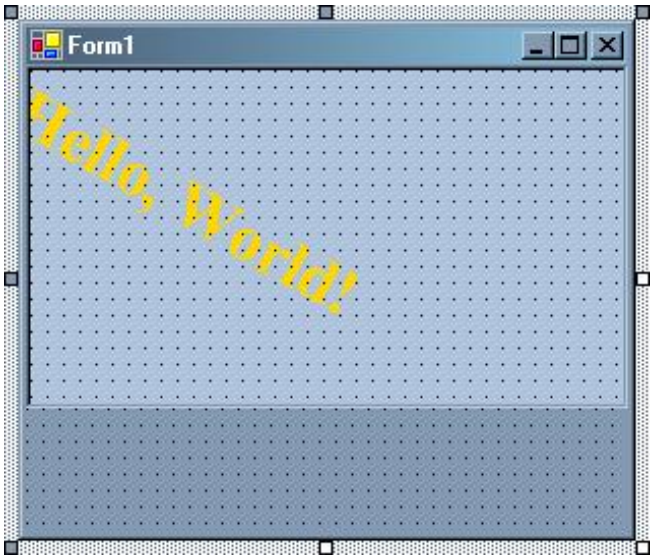
```
[Category("Appearance"),

Description("Rotation of text in degrees")]

public System.Single TextRotation
```

Te dwie ostatnie zmiany piszemy nad poprzednio zdefiniowanymi akcesorami. Ustawiają one odpowiednio kategorię (**Category**) oraz opis (**Description**) kontrolki, który będzie pojawiał się w Designerze. Dzięki temu do właściwości kontrolki będziemy mieli dostęp tak, jak poniżej na przykładzie.



Rysunek 7 - Designer Class w działaniu - nowe właściwości: TextRotation i TextString

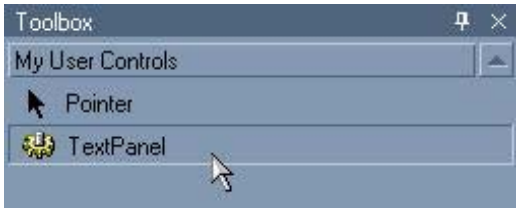
Tak, jak widać na załączonym obrazku – **TextRotation** i **TextString** pojawiły się w części **Appearance** we właściwościach naszej kontrolki. Wiele spośród innych właściwości zostało odziedziczone po kontrolce **System.Windows.Forms.Panel**, na przykład kolor tła, styl obramowania, czcionka użyta do pisania, kolor napisu, i inne. Dziedziczenie oszczędza bardzo wiele pracy. A sama kontrolka po wyświetleniu będzie wyglądała tak, jak poniżej.



Rysunek 8 - Stworzona przez nas kontrolka na formularzu

Nie omówię w tym artykule we wszystkich atrybutów oraz ich zastosowania, ponieważ jest to temat na całkiem odrębny artykuł. O wszystkim na ten (i nie tylko) temat można dowiedzieć się używając MSDN Library.

Jeszcze jedna rzecz. Jak używać tej kontrolki z poziomu innego projektu? Wystarczy dodać do niej referencję. Na przykład: należy stworzyć nowy projekt **C# Windows Application**, o nazwie na przykład **TesterApplication**. Później należy dodać w tym projekcie referencję do stworzonej biblioteki kontrolek. Jeśli znajduje się ona w zakresie tego samego rozwiązania znajdziemy łatwo odpowiednie biblioteki w zakładce **Projects** okienka dodawania referencji. Ostatnim krokiem jest dodanie kontrolki do **Toolboxa** w Designerze. Najlepiej chyba jest stworzyć sobie własną zakładkę – w projekcie **TesterApplication** wchodzimy w **Toolbox**, dodajemy nową zakładkę – prawym przyciskiem myszy na obszarze **Toolboxa**, wybieramy **Add Tab**, wpisujemy nazwę, na przykład **My User Controls**. Wybieramy nowo utworzoną zakładkę i wybieramy prawym przyciskiem myszy **Add/Remove Items...**, następnie w okienku **Customize Toolbox** wybieramy **Browse...** i nawigujemy do katalogu, w którym jest nasza biblioteka, już skompilowana, czyli po prostu odpowiedni plik DLL. Wybieramy ten plik i następnie możemy wybrać odpowiednią kontrolkę, którą ta biblioteka zawiera, czyli **TextPanel**. Odpowiedni przycisk pojawi się w **Toolboxie**, przyczyniając się do wzrostu naszego prywatnego współczynnika lenistwa. Od tej pory jest już gotowy do zastosowania tak jak inne przyciski, które Visual Studio .NET ma już skonfigurowane:



Rysunek 9 - Toolbox zawierający przycisk ze zdefiniowaną przez nas klasą

Podsumowanie

Czyli o przydatności artykułu

Tak, jak widzicie, życie programisty jednak lekkie nie jest. Niby jest leniwy, ale i tak musi się do tego lenistwa porządnie przygotować. Praca przy przygotowaniu sobie środowiska edycyjnego bardzo się opłaca. Raz stworzoną kontrolkę możemy wielokrotnie łatwo używać w innych projektach, zbudować własną imponującą bibliotekę komponentów i tak dalej. Przykład, tu zamieszczony celowo nie był zbyt przydatny, by skłonić Was do przemyślenia zastosowania i własnej pracy. Po co komu obracany tekst? Jest całkowicie nieprzydatny :). Inne pomysły na własną kontrolkę to na przykład kontrolka, w której podaje się równanie oraz zakresy współrzędnych X i Y, a kontrolka już będzie sama sobie rysowała wykres. Ciekawa by też była wizualizacja danych pobieranych przez kontrolkę z jakiegoś źródła. Możliwości jest sporo, jedynym ograniczeniem, jakie przed Wami stoi, jest Wasza wyobraźnia, czyli ograniczeń po prostu nie ma.

Mam nadzieję, że artykuł Wam się przydał. Do zobaczenia, a właściwie – do przeczytania!

-Marcin 'Vigrid' Seredyński

Drukuj Zamknij