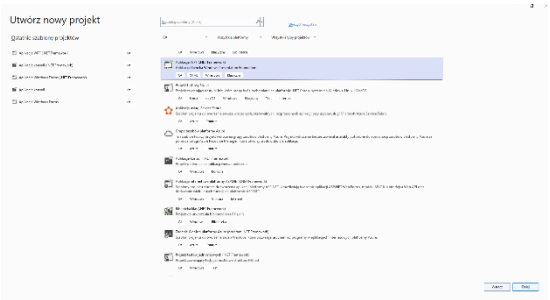
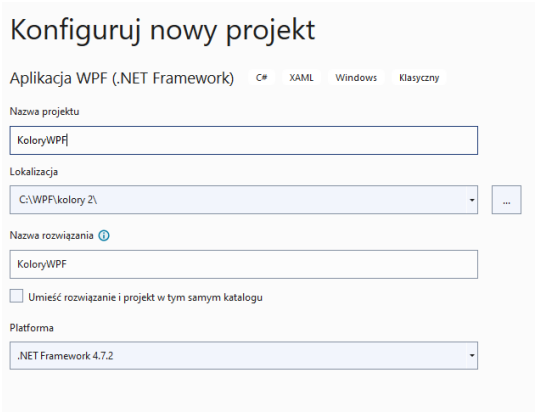
Tworzenie projektu

Stworzymy aplikację, w której za pomocą trzech suwaków będziemy kontrolować kolor widocznego w oknie prostokąta. To da nam pretekst do zapoznania się z narzędziami projektowania wizualnego przeznaczonymi dla aplikacji WPF.

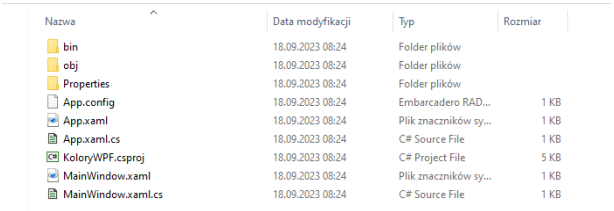
W menu *File* wybieramy podmenu *New*, a następnie polecenie *Project…*

- wybieramy projekt *WPF App (.NET Framework), podajemy nazwę aplikacji* KoloryWPF*;*

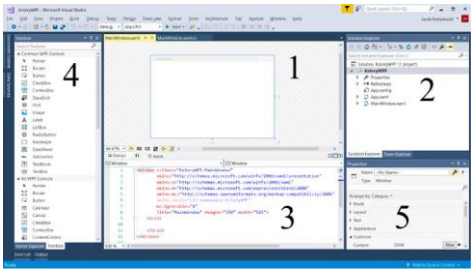




Utworzyliśmy projekt o nazwie *KoloryWPF*. W katalogu projektu znajdziemy dwie pary plików: *MainWindow.xaml*/*MainWindow.xaml.cs* oraz *App.xaml*/*App.xaml.cs*.

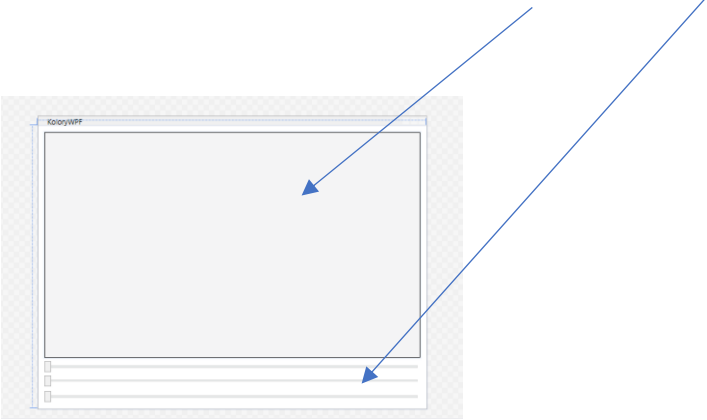


Pierwsza odpowiada za klasę głównego okna, druga — za klasę całej aplikacji. Najważniejsze są pliki z tej pierwszej pary. To ich zawartość widzimy w głównym oknie Visual Studio po utworzeniu projektu. Widoczne są tam dwie zakładki o nazwach *MainWindow.xaml* i *MainWindow.xaml.cs*. Na pierwszej z nich widzimy podgląd okna (na rysunku oznaczony numerem 1) i edytor kodu XAML (numer 3). Wielkość podglądu okna i wielkość czcionki w edytorze możemy dowolnie skalować, korzystając z rolki myszy (należy przytrzymać klawisz *Ctrl*). Z lewej strony znajduje się podokno *Przybornik* zawierające zbiór kontrolek WPF (numer 4). Z prawej widzimy podokno zawierające listę wszystkich plików rozwiązania i znajdującego się w nim jednego projektu. Pod nim widoczne jest podokno właściwości (numer 5).



Projektowanie interfejsu

Interfejs aplikacji będzie się składał z prostokąta (element Rectangle) i trzech suwaków (elementy Slider).

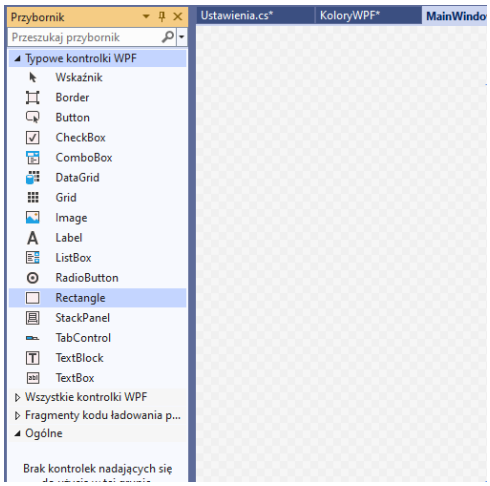


Możemy zbudować go, albo używając podglądu okna i umieszczając na nim elementy widoczne w panelu *Przybornik*, albo wprost pisząc kod XAML w edytorze.

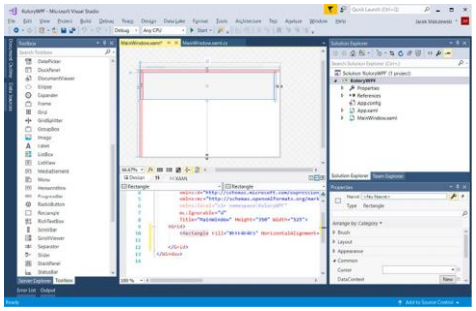
W domyślnym widoku Visual Studio widoczny jest zarówno podgląd okna, jak i edytor XAML. Wszelkie zmiany wprowadzone w jednym z tych edytorów są natychmiast widoczne w drugim, zatem jak najbardziej możliwe i wygodne jest używanie jednocześnie ich obu.

Zacznijmy od projektowania z użyciem podglądu okna, mimo że bezpośrednia edycja kodu XAML jest w większości przypadków wygodniejsza i wydajniejsza.

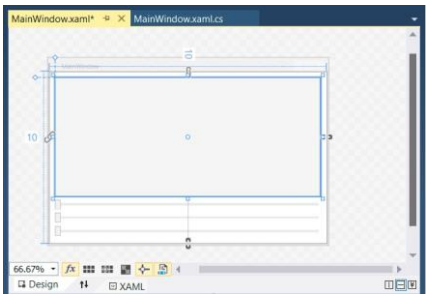
Przeciągnijmy komponent Rectangle z Przybornika na podgląd okna.



Dopasujmy jego rozmiar. Zwiększmy go tak, żeby między jego krawędzią a krawędzią okna pojawił się różowy pasek sygnalizujący zalecaną odległość kontrolki od krawędzi okna. Zróbmy tak z lewą, prawą i górną krawędzią prostokąta.



Umieśćmy w oknie trzy kontrolki Slider (suwaki) jedną pod drugą. Ich lewą i prawą krawędź ustawmy w odpowiedniej odległości od brzegów okna. Potem położenie najniższej dopasujmy do dolnej krawędzi okna. Wyższą umieśćmy nad najniższą.



Ustawmy wysokość prostokąta w taki sposób, żeby również jego odległość była optymalna.

Powyższe czynności doprowadziły do następującego kodu XAML:

```
<Window x:Class="KoloryWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:KoloryWPF"
        mc:Ignorable="d"
                Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Rectangle Fill="#FFF4F4F5" Stroke="Black"
                   Width="498" Height="224" Margin="10,10,0,0"
                   HorizontalAlignment="Left" VerticalAlignment="Top" />
        <Slider HorizontalAlignment="Left" Margin="10,239,0,0"
                VerticalAlignment="Top" Width="498" Height="18"/>
        <Slider HorizontalAlignment="Left" Margin="10,262,0,0"
                VerticalAlignment="Top" Height="21" Width="498"/>
        <Slider HorizontalAlignment="Left" Margin="10,288,0,0"
                VerticalAlignment="Top" Height="23" Width="498"/>
    </Grid>
</Window>
```

Kontrolkom odpowiadają elementy zawarte w elemencie Grid (siatka). Siatka określa sposób ułożenia kontrolek — kontrolki można w niej ustawiać w kolumnach i rzędach, czego w tej chwili nie wykorzystujemy. Określamy natomiast bezwzględnie ich odległości od lewej i górnej krawędzi jedynej komórki siatki, a zatem w praktyce położenie względem lewego górnego rogu całego okna (atrybuty Margin kontrolek). Każda kontrolka zawiera atrybuty VerticalAlignment i HorizontalAlignment. W tej chwili są ustawione odpowiednio na Left i Top, co oznacza, że położenie kontrolek jest wyznaczane przez marginesy ustalane względem lewej i górnej krawędzi. Ich szerokość i wysokość wyznaczają natomiast atrybuty Width i Height, a nie odległość od brzegów okna. W konsekwencji dwie ostatnie wartości w atrybucie Margin są ignorowane.

Taki sposób wyznaczania geometrii kontrolek ma zasadniczą wadę. Uruchommy aplikację. Każda zmiana rozmiaru okna powoduje, że rozmiary kontrolek przestają do niego pasować. Aby temu zaradzić, możemy użyć różnego typu pojemników dbających o układ kontrolek lub odpowiedniego ustawienia atrybutów samych kontrolek, które zamiast ustalać ich bezwzględny rozmiar, „zakotwiczą" je do brzegów okna.

W widoku projektowania zmieńmy ustawienie wszystkich kontrolek określające sposób ich „zaczepienia" do prawego brzegu okna i dolnej krawędzi. Prostokąt będzie w ten sposób zaczepiony do wszystkich czterech krawędzi i dlatego wraz ze zmianą rozmiaru okna zmieniać się będzie zarówno jego szerokość, jak i wysokość. Wyłączmy natomiast domyślnie ustawione zaczepienie suwaków do górnej krawędzi okna. To oznacza, że wraz

ze zwiększaniem wysokości okna suwaki będą bez zmiany wysokości przesuwać się przyczepione do jego dolnej krawędzi.

Te czynności spowodowały zmianę w kodzie XAML.

```
<Grid>
    <Rectangle Fill="#FFF4F4F5" Margin="10,10,10.4,83.8" Stroke="Black"/>
    <Slider Margin="10,0,10.4,60.8" Height="18" VerticalAlignment="Bottom"/>
    <Slider Margin="10,0,10.4,34.8" Height="21" VerticalAlignment="Bottom"/>
    <Slider Margin="10,0,10.4,9.8" Height="20" VerticalAlignment="Bottom"/>
</Grid>
```

Zniknęły atrybuty HorizontalAlignment, a atrybuty VerticalAlignment zmieniły wartość na Bottom. Dodatkowo w atrybucie Margin pojawiły się odległości od prawej i dolnej krawędzi. Po uruchomieniu aplikacji przekonamy się, że zmiana rozmiaru okna nie psuje już interfejsu — rozmiary kontrolek będą na bieżąco dopasowywane do rozmiaru okna.

Elementem nadrzędnym w z pliku *MainWindows.xaml* jest element Window reprezentujący okno aplikacji. W nim zagnieżdżony jest element Grid (siatka) odpowiadający za ułożenie kontrolek w oknie. W nim są pozostałe kontrolki: prostokąt i trzy suwaki. Zagnieżdżenie elementów oznacza, że „zewnętrzna" kontrolka jest pojemnikiem, w którym znajdują się kontrolki reprezentowane przez „wewnętrzne" elementy.

Warto zwrócić uwagę na atrybuty elementu Window. Atrybut x:Class tworzy pomost między elementem Window, określającym opisywane w pliku okno, a klasą C# o nazwie MainWindow w przestrzeni nazw KoloryWPF, której jeszcze nie edytowaliśmy, a która znajduje się w pliku *MainWindow.xaml.cs*. Atrybut xmlns określa domyślną przestrzeń nazw używaną w bieżącym elemencie XAML i w jego podelementach — w pewnym sensie odpowiada instrukcji using w kodzie C#. Z kodu wynika, że dostępnych jest pięć przestrzeni nazw.

Znaczenia pozostałych atrybutów elementu Window są bardziej oczywiste: Title określa tytuł okna widoczny na pasku tytułu, a Height i Width — jego rozmiary. Możemy je swobodnie zmieniać, przypisując im nowe wartości, np.:

```
Title="KoloryWPF" Height="480" Width="640">
```

Kolejnym etapem tworzenia aplikacji jest określenie jej „dynamiki”. Chcemy, aby suwaki umożliwiały ustalanie koloru prostokąta, a konkretnie, żeby możliwe było ustawianie za ich pomocą wartości trzech składowych RGB koloru.

Ustawmy nazwy suwaków tak, żeby odpowiadały składowym koloru, z którymi będą związane. Nazwijmy je sliderR, sliderG i sliderB. Nadajmy także nazwę prostokątowi. Bez tego nie będziemy mogli modyfikować jego własności z poziomu kodu C#. Aby nadać nazwę elementowi XAML, trzeba ustalić wartość jego atrybutu x:Name:

```
<Rectangle x:Name="rectangle" … />
<Slider x:Name="sliderR"… />
<Slider x:Name="sliderG"… />
<Slider x:Name="sliderB"… />
```

Kolejnym krokiem będzie związanie z suwakami metody zdarzeniowej reagującej na zmianę ich pozycji. Dwukrotne kliknięcie najwyższego suwaka na podglądzie okna tworzy domyślną metodę zdarzeniową i przenosi nas do edytora kodu C#, ustawiając kursor w nowo utworzonej metodzie klasy MainWindow. Metoda zostanie nazwana sliderR_ValueChanged przez połączenie nazwy kontrolki i nazwy zdarzenia. Jeżeli wrócimy do kodu XAML, zobaczymy, że jednocześnie do elementu Slider dodany został atrybut ValueChanged, którego wartością jest nazwa metody, tj. sliderR_ValueChanged:

```
<Slider x:Name="sliderR" Margin="10,0,10.4,60.8" Height="18"
VerticalAlignment="Bottom" ValueChanged="sliderR_ValueChanged"/>
```

Do nowej metody wstawmy polecenie zmieniające kolor prostokąta.

```
private void sliderR_ValueChanged(object sender,
                        RoutedPropertyChangedEventArgs<double> e)
{
     rectangle.Fill = Brushes.Crimson;

}
```

Wykorzystany przez nas prostokąt można pokolorować, zmieniając zarówno jego brzeg (Stroke), jak i wypełnienie (Fill). My zmieniamy wypełnienie.

Powiążmy stworzoną przed chwilą metodę zdarzeniową z dwoma pozostałymi suwakami. Możemy tego dokonać, używając zakładki *Events* podokna *Properties*, lecz w praktyce wygodniej to zrobić, edytując kod XAML. Dodajmy do drugiego i trzeciego elementu atrybuty zdarzenia ValueChanged. Za nimi umieśćmy operator przypisania = i cudzysłów „. Wówczas pojawi się okienko z listą metod, których sygnatury pasują do sygnatury zdarzenia. Wybieramy z tej listy istniejącą metodę sliderR_ValueChanged. Jeżeli lista zawiera tylko pozycję pozwalającą na utworzenie nowej metody zdarzeniowej, musimy nazwę istniejącej metody wpisać samodzielnie. Jeśli zrobimy to dla obu suwaków, to po uruchomieniu aplikacji zmiana pozycji każdego z nich spowoduje zmianę koloru prostokąta.

Aby aplikacja uzyskała zaplanowaną przez nas funkcjonalność, metoda musi zmieniać kolor prostokąta odpowiednio do bieżących pozycji suwaków. Musimy zatem odczytać ich wartości Value, ustalić na ich podstawie kolor i przypisać go do własności Fill prostokąta. Wartość Value jest typu double. Domyślnie przyjmuje wartości z zakresu od 0 do 10. Są to domyślne wartości własności Minimum i Maximum. Zmieńmy górną granicę zakresu na 255, tak aby umożliwiała swobodne dobieranie wartości liczb byte bez skalowania (za to z koniecznym rzutowaniem). Możemy to zrobić, albo zmieniając wartość własności Maximum przy wykorzystaniu okna *Properties*, albo edytując kod XAML.

```
<Grid>
  <Rectangle x:Name="rectangle" Fill="#FFF4F4F5" Margin="10,10,10.4,83.8"
Stroke="Black"/>
  <Slider x:Name="sliderR" Margin="10,0,10,60" Height="18"
              VerticalAlignment="Bottom"
              Maximum="255" ValueChanged="sliderR_ValueChanged"/>
  <Slider x:Name="sliderG" Margin="10,0,10,34" Height="21"
              VerticalAlignment="Bottom"
              Maximum="255" ValueChanged="sliderR_ValueChanged"/>
  <Slider x:Name="sliderB" Margin="10,0,10,9" Height="20"
              VerticalAlignment="Bottom"
              Maximum="255" ValueChanged="sliderR_ValueChanged"/>

</Grid>
```

```
private void sliderR_ValueChanged(object sender,
                                  RoutedPropertyChangedEventArgs<double> e)
{
   Color kolor = Color.FromRgb((byte)sliderR.Value, (byte)sliderG.Value,
                               (byte)sliderB.Value);
   rectangle.Fill = new SolidColorBrush(kolor);
}
```

Aby zsynchronizować początkowy kolor prostokąta z pozycją suwaków, po uruchomieniu programu wywołajmy metodę sliderR_ValueChanged z konstruktora klasy MainWindow

```
public MainWindow()
{
   InitializeComponent();
   sliderR_ValueChanged(null, null);
}
```

Zmodyfikujmy kod tak, aby tworzyć jedną trwałą instancję klasy SolidColorBrush i tylko zmieniać jej własność Color. To spowoduje, że kod stanie się nieco mniej przejrzysty, ale na pewno będzie bliższy optymalnemu. W tym celu do konstruktora klasy MainWindow przenieśmy polecenie tworzące obiekt pędzla odpowiedzialnego za kolorowanie prostokąta:

```
rectangle.Fill = new SolidColorBrush(Colors.Black);
```

Powinno ono zastąpić dodane przed chwilą wywołanie metody sliderR_ValueChanged. Natomiast w metodzie zdarzeniowej zmodyfikujmy własność Color obiektu pędzla dostępnego dzięki referencji rectangle.Fill.

```
private void sliderR_ValueChanged(object sender,
                                  RoutedPropertyChangedEventArgs<double> e)
{
    Color kolor = Color.FromRgb((byte)sliderR.Value, (byte)sliderG.Value,
                                (byte)sliderB.Value);
   //rectangle.Fill = new SolidColorBrush(kolor);
   (rectangle.Fill as SolidColorBrush).Color = kolor;
}
```

Stwórzmy metodę zamykającą okno, a tym samym całą aplikację, po naciśnięciu przez użytkownika klawisza *Escape*.

Przechodzimy do widoku projektowania.

Korzystając z mechanizmu uzupełniania kodu, do elementu Window (w kodzie XAML) dodajemy atrybut KeyDown, a następnie, korzystając z mechanizmu uzupełniania kodu, przypisujemy mu pozycję *<New Event Handler>* z okienka podpowiedzi. Wartość atrybutu zostanie ustalona na Window_KeyDown, a w pliku z kodem C# pojawi się metoda o tej nazwie. Zmodyfikujmy tę metodę.

```
private void Window_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Escape) Close();
}
```

Własności

Uprośćmy sposób dostępu do koloru prostokąta. Zdefiniujmy własność KolorProstokąta

```
private Color KolorProstokata
{
    get
    {
        return (rectangle.Fill as SolidColorBrush).Color;
    }
    set
    {
        (rectangle.Fill as SolidColorBrush).Color = value;
    }

}
```

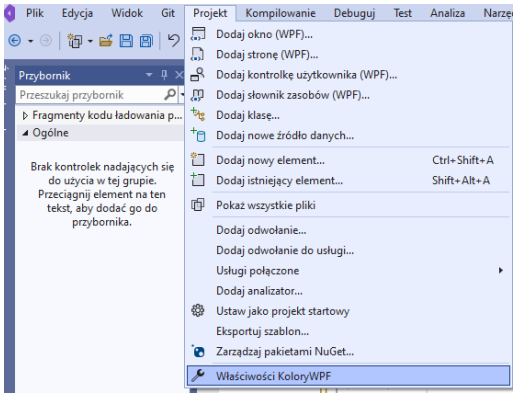
Dzięki tej własności ostatnie polecenie w metodzie sliderR_ValueChanged może być zamienione po prostu na KolorProstokąta = kolor;.

```
private void sliderR_ValueChanged(object sender,
                        RoutedPropertyChangedEventArgs<double> e)
{
    Color kolor = Color.FromRgb((byte)sliderR.Value, (byte)sliderG.Value,
                                (byte)sliderB.Value);
   //rectangle.Fill = new SolidColorBrush(kolor);
   //(rectangle.Fill as SolidColorBrush).Color = kolor;
   KolorProstokata = kolor;
}
```

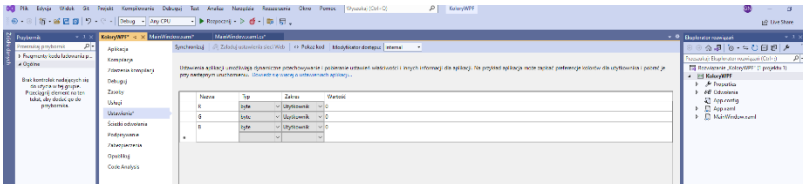
Zapisywanie i odtwarzanie stanu aplikacji

Zachowaniem, którego często oczekujemy od nowoczesnych aplikacji, jest odtwarzanie stanu aplikacji po jej zamknięciu i ponownym uruchomieniu. W przypadku tak prostej aplikacji jak nasza, w której stan aplikacji to w istocie trzy wartości typu byte, do zapisania jej stanu w zupełności wystarczy mechanizm ustawień aplikacji oferowany przez platformę .NET. Należy go wcześniej odpowiednio skonfigurować.

Z menu *Projekt* wybieramy „Właściwości *KoloryWPF”* i przechodzimy do zakładki *Ustawienia*



Następnie do tabeli widocznej w tej zakładce dodajemy trzy wiersze opisujące trzy wartości, które będą mogły być przechowywane w ustawieniach. Nazwijmy te wartości R, G i B. Ustalmy ich typ na *byte*.



Żeby uprościć odczyt i zapis ustawienia aplikacji, napiszemy dwie realizujące te zadania metody statyczne, umieszczone w osobnej klasie statycznej Ustawienia. Takie rozwiązanie ułatwi ewentualną zmianę sposobu przechowywania ustawień.

Dodajemy do projektu klasę o nazwie Ustawienia. Dodajemy przestrzeń nazw,

```
using System.Windows.Media;
```

w której zdefiniowana jest klasa Color używana w WPF.

Modyfikujemy jej kod (zamiast statycznej metody FromRgb, użyjemy konstruktora domyślnego wraz z inicjatorem obiektu).

```
using System.Windows.Media;

namespace KoloryWPF
{
    internal class Ustawienia
    {
        public static Color Czytaj()
        {
            Properties.Settings ustawienia = Properties.Settings.Default;
            Color kolor = new Color()
            {
                A = 255,
                R = ustawienia.R,
                G = ustawienia.G,
                B = ustawienia.B
            };
            return kolor;
        }
        public static void Zapisz(Color kolor)
        {
            Properties.Settings ustawienia = Properties.Settings.Default;
            ustawienia.R = kolor.R;
            ustawienia.G = kolor.G;
            ustawienia.B = kolor.B;
            ustawienia.Save();
        }
    }

}
```

Korzystając z metody Ustawienia.Zapisz, zapiszmy do ustawień aplikacji bieżący kolor prostokąta w momencie zamykania okna. Użyjemy do tego jego zdarzenia Closed. Postępując podobnie jak w przypadku zdarzenia Window.KeyDown, stwórzmy metodę związaną ze zdarzeniem Window.Closed

```
        private void Window_Closed(object sender, EventArgs e)
        {
            Ustawienia.Zapisz(KolorProstokata);

        }
```

Przy odczytywaniu ustawień po uruchomieniu aplikacji należy uwzględnić położenie suwaków.

```
        public MainWindow()
        {
            InitializeComponent();
//          rectangle.Fill = new SolidColorBrush(Colors.Black);
//          sliderR_ValueChanged(null, null);
            Color kolor = Ustawienia.Czytaj();
            rectangle.Fill = new SolidColorBrush(kolor);
            sliderR.Value = kolor.R;
            sliderG.Value = kolor.G;
            sliderB.Value = kolor.B;
        }
```

Ponieważ wszystkie ustawienia aplikacji, które zapisujemy w programie, należą do ustawień użytkownika, wykonanie metody Ustawienia.Zapisz spowoduje, że platforma .NET utworzy dla nich plik XML w katalogu domowym użytkownika (np. *C:\Users\gnowak*; korzystając z aliasu dla katalogu domowego *C:\Documents and Settings\gnowak* lub *C:\ Użytkownicy\gnowak*), a dokładniej w jego podkatalogu *AppData\Local* (względnie *Ustawienia lokalne\Dane aplikacji*). Powstanie tam katalog o nazwie aplikacji, z podkatalogiem oznaczającym konkretny plik wykonywalny i jeszcze jednym podkatalogiem zawierającym numer wersji aplikacji. Dopiero w tym miejscu powstanie plik XML o nazwie *user.config*. Plik *user.config* zawiera sekcję *userSettings*, czyli ustawienia aplikacji z zakresu użytkownika. Taki sam zbiór ustawień znajdziemy we wspomnianym już pliku *KoloryWPF.exe.config*, który jest umieszczony w katalogu aplikacji i powinien być z nią rozpowszechniany.

Zmodyfikuj projekt tak, żeby zapisywane były także rozmiar i położenie okna.