Jacek Matulewski

Visual Studio 2022

Wprowadzenie do .NET MAUI

Redaktor prowadzący: Szymon Szwajger

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/vs22ta
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-9026-3

Printed in Poland.

Spis treści

ROZDZIAŁ 1. Pierwsza aplikacja .NET MAUI

Tworzenie projektu

Aby przyjrzeć się .NET MAUI, proponuję zbudować prostą aplikację, od której zwykle zaczynam poznawanie nowych technologii pozwalających na tworzenie graficznych interfejsów użytkownika (czyli tzw. aplikacji z GUI, ang. *graphical user interface*). W tej aplikacji za pomocą trzech suwaków będziemy kontrolować kolor widocznego w oknie prostokąta. To rzeczywiście prosty projekt, ale umożliwia poznanie podstawowych mechanizmów i narzędzi służących do budowania graficznego interfejsu użytkownika. Zwykle oznacza to naukę wizualnych narzędzi projektowania interfejsów, które są obsługiwane myszką. Te jednak w przypadku .NET MAUI nie są (jeszcze?) dostępne — w chwili pisania tej książki nie ma nawet podglądu projektowanego interfejsu.

Uruchommy Visual Studio 2022 (VS; ja używam edycji Community o numerze wersji 17.3.6). Pojawi się okno początkowe (rysunek 1.1), w którym możemy wybrać istniejący lub utworzyć nowy projekt. Z listy z prawej strony tego okna wybieramy *Utwórz nowy projekt*. Na kolejnej stronie, aby wyszukać odpowiedni typ projektu, wprowadzamy „MAUI" w polu edycyjnym w prawym górnym rogu i z zaproponowanych szablonów wybieramy *Aplikacja platformy .NET MAUI*. Klikamy *Dalej*. W polu *Nazwa projektu* wpisujemy „KoloryMAUI" i klikamy *Dalej*. W kolejnym kroku kreatora wybieramy platformę .NET 6 (u mnie jedyna opcja) i wreszcie klikamy *Utwórz*.

Warto wspomnieć, że istnieje możliwość wyboru projektu z interfejsem tworzonym w technologii Blazor, która może być szczególnie atrakcyjna dla osób mających doświadczenie w tworzeniu aplikacji internetowych.

Utworzyliśmy projekt o nazwie *KoloryMAUI*, którego pliki domyślnie zostały umieszczone w podkatalogu *\source\repos\KoloryMAUI\KoloryMAUI* katalogu domowego (pierwszy katalog *KoloryMAUI* w ścieżce to katalog rozwiązania z plikiem *KoloryMAUI.sln*). W katalogu projektu znajdziemy m.in. dwie pary plików: *MainPage.xaml* i *MainPage.xaml.cs* oraz *App.xaml* i *App.xaml.cs*. Pierwsza para odpowiada za zawartość prezentowaną użytkownikowi w GUI, druga za działanie całej aplikacji. Ważnym uzupełnieniem tych plików są te umieszczone w podkatalogu *Platforms* — są to pliki odpowiedzialne m.in. za utworzenie okien na różnych platformach, w których umieszczona będzie zawartość zdefiniowana w pliku *MainPage.xaml* (w przypadku Androida właściwszym określeniem jest „aktywność", a nie „okno").

RYSUNEK 1.1. Okno początkowe w Visual Studio 2022

Skupię się przede wszystkim na plikach z pierwszej pary. To właśnie kod z tych plików zobaczymy w edytorze Visual Studio po utworzeniu projektu (zakładki o nazwach *MainPage.xaml* i *MainPage.xaml.cs* widoczne na rysunku 1.2). Na pierwszej z nich jest kod XAML opisujący wygląd zawartości okna (nie ma podglądu, przynajmniej na razie), a na drugiej kod C# związany z tym oknem. Warto zwrócić uwagę, że inaczej niż w WPF, a podobnie jak w UWP, pliki te opisują jedynie zawartość okna, a nie samo okno. Ważną konsekwencją tego faktu jest to, że nie mamy z tego miejsca dostępu do niektórych zdarzeń charakterystycznych dla okien.

RYSUNEK 1.2. Edytor kodu XAML w Visual Studio 2022

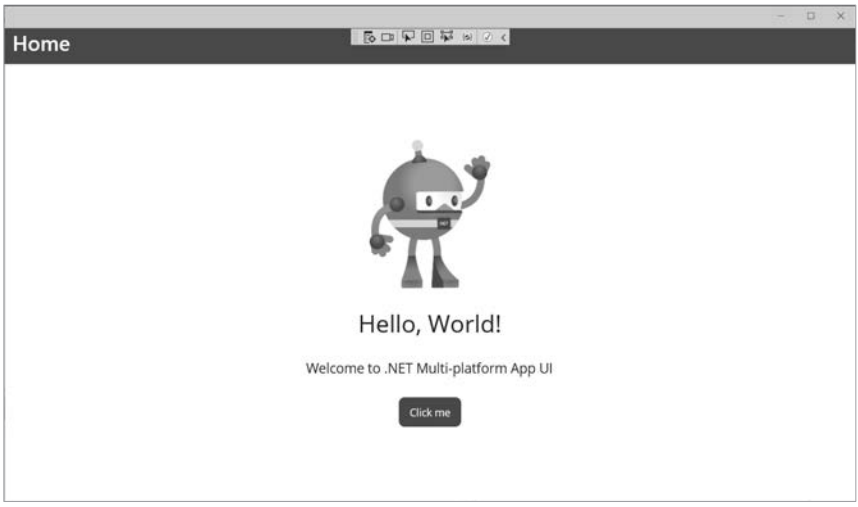
Zwróćmy uwagę, że w utworzonym rozwiązaniu jest tylko jeden projekt. Nie ma typowych dla platformy Xamarin wielu osobnych projektów opisujących warstwę widoku dla poszczególnych platform przy wspólnym projekcie modelu (tak było nawet w projektach Xamarin.Forms). Pliki specyficzne dla poszczególnych platform kryją się natomiast we wspomnianym już folderze *Platforms* z podfolderami

Android, *iOS*, *MacCatalyst*, *Tizen* i *Windows*. Dzięki nim można dostosować zasoby dla poszczególnych platform (np. używane pliki graficzne). Oczywiście nic nie stoi na przeszkodzie, abyśmy w miarę rozwoju aplikacji wyodrębnili do osobnego projektu np. bibliotekę klas, które będą tworzyły model aplikacji (jej logikę niezależną od GUI), natomiast nie ma konieczności tworzenia osobnych projektów opisujących GUI na każdej z platform. Należy jednak pamiętać, że mimo jednego projektu jest on kompilowany osobno dla każdej platformy.

Uruchamianie aplikacji w systemie Windows

Warto od razu uruchomić program, aby sprawdzić, czy wszystko jest w porządku. Proponuję zacząć od systemu Windows, żeby nie tworzyć teraz wirtualnego urządzenia z systemem Android uruchamianym w emulatorze lub nie podłączać rzeczywistego urządzenia przez kabel USB. Tym zajmiemy się w rozdziale 2.

Z rozwijanej listy przy ikonie uruchamiania aplikacji wybierzmy *Windows Machine* i kliknijmy przycisk uruchamiania. Po chwili zobaczymy aplikację .NET MAUI, która zawiera domyślnie utworzony projekt (rysunek 1.3).



RYSUNEK 1.3. Domyślna aplikacja z projektu .NET MAUI utworzonego na podstawie szablonu (w przypadku aplikacji z interfejsem tworzonym w technologii Blazor domyślny projekt jest inny)

Zaskoczyć może rozmiar folderu z projektem MAUI, który jest ogromny (kilkaset megabajtów). Aby go przenieść na inny komputer lub zarchiwizować, można usunąć największe podfoldery, a mianowicie *KoloryMAUI**bin* i *KoloryMAUI**obj**Debug* (także *KoloryMAUI**obj**Release*, jeżeli skompilujemy projekt w trybie *Release*). Rozmiar folderu maleje wówczas do kilku megabajtów.

Aplikacja sterowana zdarzeniami

Osoby, które dopiero zaczynają naukę XAML (w .NET MAUI, WPF lub UWP), a mają doświadczenie w programowaniu aplikacji „okienkowych" (np. z użyciem biblioteki Windows Forms lub narzędzi Borland i Embarcadero, jeżeli ktoś jeszcze je pamięta), najprawdopodobniej przeniosą swoje nawyki i w naturalny sposób zaczną korzystać ze zdarzeń. To zaprowadzi je do struktury projektu, w której dane tworzące stan aplikacji i dotycząca ich logika są przechowywane w klasach widoku, zwłaszcza w klasie `MainPage`, bez żadnej separacji modułów odpowiedzialnych za poszczególne funkcjonalności. W takim podejściu do określenia tego, jak aplikacja ma reagować na działania użytkownika, wykorzystywane są bardzo wygodne zdarzenia kontrolek. Brak separacji poszczególnych modułów utrudnia jednak testowanie kodu (w praktyce możliwe jest tylko testowanie funkcjonalne całej aplikacji) i zmniejsza elastyczność projektu (trudniejsze są zmiany). Ale to wcale nie musi być złe rozwiązanie. W tym wzorcu niewielkie aplikacje tworzy się szybko, szczególnie w początkowej fazie projektu, tzn. zanim okaże się, że zamawiający chce je jednak znacząco rozbudować lub zmienić. A zazwyczaj tak jest. Lecz nie tylko rozmiar projektu powinien decydować o wybieranym wzorcu architektonicznym. Nie zawsze warto dbać o rozdzielanie modułów i najlepsze praktyki. Czasem ważne jest, aby aplikacja powstała szybko i zadziałała w konkretnym przypadku. Jeżeli na tym kończy się życie projektu, to wysiłek włożony w jego „czystość" w żaden sposób nie zaprocentuje. Problem tylko w tym, że, jak pokazuje życie, naprawdę rzadko na tym się kończy — wbrew początkowym planom taki prowizoryczny kod trzeba zazwyczaj potem utrzymywać i rozwijać.

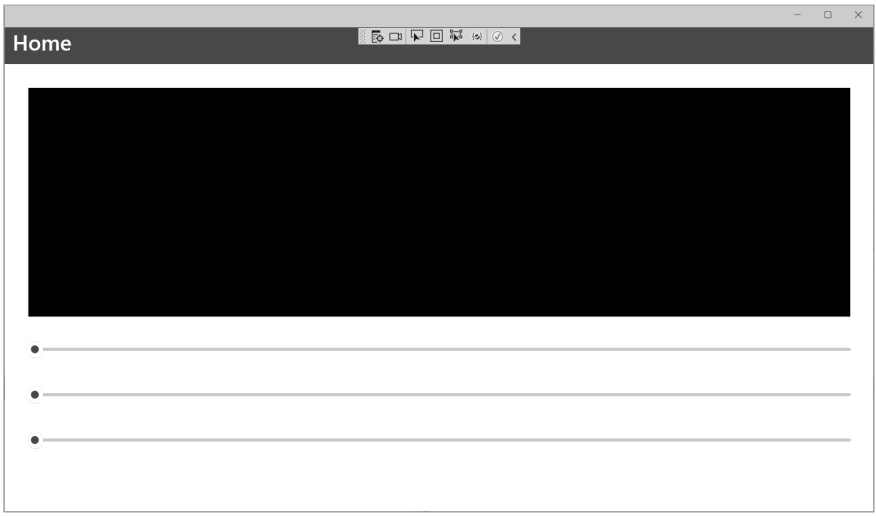
Projektowanie interfejsu z MAUI

Jak wspominałem, w obecnej wersji .NET MAUI nie ma jeszcze widoku projektowania, jaki znamy z projektów WPF lub UWP. Nie ma także możliwości tworzenia interfejsu za pomocą *Blend for Visual Studio 2022*. Brak wizualnego edytora XAML to może nie jest aż tak wielka strata, skoro i tak większość programistów bezpośrednio edytuje kod XAML, ale wygodnie byłoby mieć chociaż podgląd. Na razie musimy się jednak bez niego obyć.

Zatem nie mogąc układać kontrolek za pomocą myszy w widoku projektowania, jesteśmy zmuszeni do edycji kodu XAML. Usuńmy domyślny kod, ale nie cały, a jedynie zawartość elementu `ScrollView` (umożliwiającego przewijanie umieszczonej wewnątrz niego zawartości), i zastąpmy oryginalny element pojemnika `VerticalStackLayout` elementem pojemnika siatki `Grid` i kodem wyróżnionym na listingu 1.1. W pierwszym wierszu siatki umieszczamy prostokąt, a pod nim trzy suwaki — każdy w osobnym wierszu (rysunek 1.4). Suwaki będą kontrolować składowe RGB koloru prostokąta. Zwróćmy również uwagę na atrybut `RowDefinitions` w elemencie `Grid`.

LISTING 1.1. Kod XAML opisujący wygląd okna

```
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="KoloryMAUI.MainPage">

    <ScrollView>
        <Grid RowSpacing="25" RowDefinitions="300,Auto,Auto,Auto" Padding="30">
            <Rectangle Grid.Row="0" Fill="Black" />
            <Slider Grid.Row="1" />
            <Slider Grid.Row="2" />
            <Slider Grid.Row="3" />
        </Grid>
    </ScrollView>
</ContentPage>
```



RYSUNEK 1.4. Interfejs opisany kodem z listingu 1.1

Usunięcie domyślnego kodu XAML spowoduje, że projektu nie można skompilować, ponieważ w kodzie C# klasy `MainPage` z pliku *MainPage.xaml.cs* obecne są odwołania do nazw usuniętych kontrolek. Już niedługo przejdziemy do kodu C# i będziemy go zmieniać, ale jeżeli niemożność skompilowania projektu jest zbyt niepokojąca, wystarczy usunąć z klasy `MainPage` metodę `OnCounterClicked` i deklarację pola `count`.

Kilka uwag na temat kodu XAML

Cały kod z pliku *MainPage.xaml* jest widoczny na listingu 1.1. Elementem nadrzędnym jest element `ContentPage` (z ang. strona z zawartością), reprezentujący zawartość okna aplikacji. W nim zagnieżdżony jest element `ScrollView`, umożliwiający przewijanie zawartości, jeżeli jest większa od rozmiaru okna lub ekranu, a w nim

element `Grid` (z ang. siatka), odpowiadający za ułożenie kontrolek w oknie. To w tym elemencie są wszystkie kontrolki, które będzie widział użytkownik, czyli prostokąt i trzy suwaki. Zagnieżdżenie elementów oznacza, że „zewnętrzna" kontrolka jest pojemnikiem, w którym znajdują się kontrolki reprezentowane przez „wewnętrzne" elementy. Przy czym część ustawień pojemnika jest przekazywana elementom zagnieżdżonym.

Warto zwrócić uwagę na atrybuty elementu `ContentPage`. Atrybut `x:Class` tworzy pomost między elementem `ContentPage`, określającym opisywaną w pliku zawartość okna, a klasą C# o nazwie `MainPage` w przestrzeni nazw `KoloryMAUI`, która znajduje się w pliku *MainPage.xaml.cs*. Atrybut `xmlns` (od ang. *XML namespace*) 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 dostępne są dwie przestrzenie nazw. Pierwszą jest przestrzeń domyślna (bez nazwy), zadeklarowana jako `http://schemas.microsoft.com/dotnet/2021/maui`, która zawiera definicje większości elementów XAML, m.in. `Rectangle` i `Slider`. Drugą przestrzenią jest ta dostępna pod nazwą `x`. To w tej przestrzeni zdefiniowany jest atrybut `Name`. Właśnie dlatego w usuniętym kodzie znajdowały się atrybuty `x:Name`; za chwilę także my ich użyjemy.

Spójrzmy teraz na element `Grid` i jego atrybut `RowDefinitions`. Dzieli on dostępny obszar na cztery wiersze; pierwszy o wysokości 300 pikseli, a wysokość kolejnych będzie dostosowana do umieszczonej w nich zawartości, czyli w naszym przypadku do wysokości suwaków. Pierwsza rzecz, na którą chciałbym zwrócić uwagę, to to, że zamiast atrybutu można użyć także elementów zagnieżdżonych (listing 1.2). Wolę jednak skupić się na innej rzeczy — sztywne ustawienie wysokości pierwszego wiersza powoduje, że wysokość całej zawartości okna jest zafiksowana i w żaden sposób nie dostosowuje się do dostępnego miejsca. Pół biedy, jeżeli okno jest na tyle duże, że cała zawartość jest widoczna (rysunek 1.4). Nie mamy jednak pewności, że tak będzie na wszystkich platformach, szczególnie na urządzeniach mobilnych, w których użytkownik nie ma wpływu na rozmiar okna (poza sytuacją dzielenia ekranu przez dwie aplikacje). Tu jednak ograniczam się tylko do zapowiedzenia problemu, którym zajmiemy się w następnym rozdziale.

LISTING 1.2. Równoważny zapis podziału siatki na wiersze

```
<Grid RowSpacing="25" Padding="30">
    <Grid.RowDefinitions>
        <RowDefinition Height="300" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Rectangle Grid.Row="0" Fill="Black" />
    <Slider Grid.Row="1" />
    <Slider Grid.Row="2" />
    <Slider Grid.Row="3" />
</Grid>
```

Nazwy kontrolek i zdarzenia

Załóżmy, że etap projektowania graficznego interfejsu jest już zakończony. Kolejnym etapem tworzenia aplikacji jest określenie jej „dynamiki". Chcemy, aby suwaki umożliwiały ustalanie koloru prostokąta, a konkretnie, żeby możliwe było ustawianie z ich pomocą wartości trzech składowych RGB koloru.

Proponuję zacząć od nadania nazw kontrolkom, aby można było się do nich odwołać z kodu C# w klasie `MainPage` i w ten sposób odczytywać i modyfikować ich własności. Ustawmy nazwy suwaków tak, żeby odpowiadały składowym koloru, z którymi będą związane: `sliderR`, `sliderG` i `sliderB`. Musimy też nadać nazwę prostokątowi. Aby nadać nazwę elementowi XAML reprezentującemu kontrolkę, trzeba ustalić wartość jego atrybutu `x:Name` (listing 1.3).

LISTING 1.3. Nazwy w elementach XAML są widoczne w klasie MainPage jako pola tej klasy

```
<Grid RowSpacing="25" Padding="30" RowDefinitions="300,Auto,Auto,Auto">
    <Rectangle x:Name="rectangle" Grid.Row="0" Fill="Black" />
    <Slider x:Name="sliderR" Grid.Row="1" />
    <Slider x:Name="sliderG" Grid.Row="2" />
    <Slider x:Name="sliderB" Grid.Row="3" />
</Grid>
```

Kolejnym krokiem będzie związanie z suwakami metody zdarzeniowej reagującej na zmianę ich pozycji. Kod tej metody powinien się znaleźć w klasie `Kolory ↪MAUI.MainPage`, ponieważ jest to klasa wskazana w atrybucie `x:Class` elementu `ContentPage` kodu XAML. Klasa ta, razem z kodem XAML, stanowi zatem część warstwy widoku. Kod C# klasy `MainPage` jest określany jako *code-behind*, czyli „kod stojący za widokiem", „na zapleczu widoku". W kontekście wzorca MVVM, który poznamy w rozdziale 3., określenie to ma negatywny wydźwięk, bo zgodnie z najbardziej rygorystyczną egzegezą tego wzorca projekt aplikacji z graficznym interfejsem użytkownika (WPF, UWP lub MAUI) w ogóle nie powinien zawierać *code-behind*. To oznacza porzucenie mechanizmu zdarzeń na rzecz wiązań z niżej leżącą warstwą modelu widoku. O tym jednak w rozdziale 3.

Możemy utworzyć metodę zdarzeniową z edytora kodu XAML. Wystarczy, że wpiszemy atrybut odpowiadający zdarzeniu i znak równości. Wówczas edytor doda cudzysłowy i pojawi się menu z pozycją *<Nowa procedura obsługi zdarzeń>*. Jeżeli ją wybierzemy, do zdarzenia zostanie przypisana metoda, której nazwa będzie się składała z nazwy obiektu i nazwy zdarzenia. Dla suwaka `sliderR` i zdarzenia `ValueChanged` sygnalizującego zmianę pozycji suwaka będzie to `sliderR_Value ↪Changed`. W przypadku kolejnego suwaka, jeżeli znowu wpiszemy nazwę zdarzenia `ValueChanged`, w menu pojawi się również nazwa utworzonej przed chwilą metody. Dzięki temu możemy z drugą kontrolką związać istniejącą już metodę zdarzeniową. Zróbmy tak także w przypadku suwaków `sliderG` i `sliderB`. W efekcie powinien powstać kod widoczny na listingu 1.4, a jednocześnie w klasie `MainPage` powinna się znajdować definicja pustej metody.

LISTING 1.4. Związanie metody ze zdarzeniem kontrolki w kodzie XAML

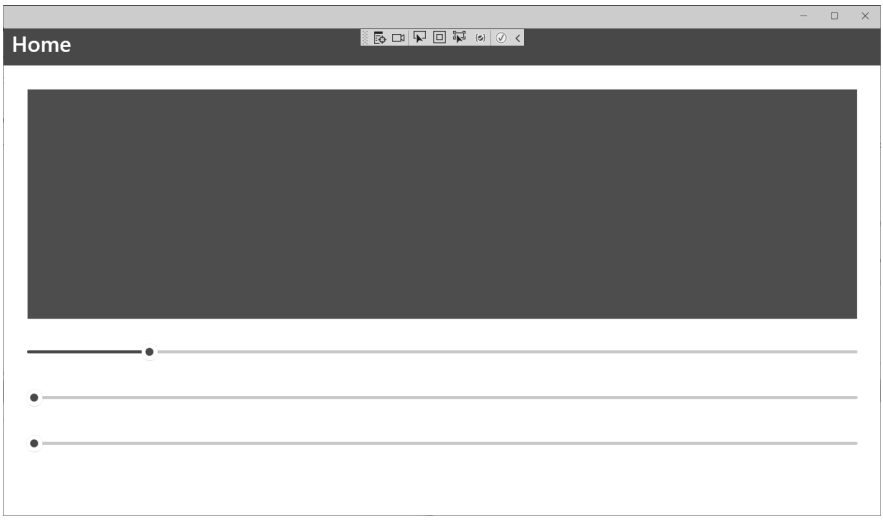
```
<Grid RowSpacing="25" Padding="30" RowDefinitions="300,Auto,Auto,Auto">
    <Rectangle x:Name="rectangle" Grid.Row="0" Fill="Black" />
    <Slider x:Name="sliderR" Grid.Row="1" ValueChanged="sliderR_ValueChanged" />
    <Slider x:Name="sliderG" Grid.Row="2" ValueChanged="sliderR_ValueChanged" />
    <Slider x:Name="sliderB" Grid.Row="3" ValueChanged="sliderR_ValueChanged" />
</Grid>
```

Do tej metody wstawmy polecenie zmieniające kolor prostokąta na ceglasty (listing 1.5). To oczywiście tylko tymczasowe rozwiązanie, ale dzięki niemu sprawdzimy, czy mechanizm zdarzeń w ogóle działa.

LISTING 1.5. Zmiana pozycji najwyższego suwaka spowoduje zmianę koloru prostokąta

```
private void sliderR_ValueChanged(object sender, ValueChangedEventArgs e)
{
    rectangle.Fill = Brush.Firebrick;
}
```

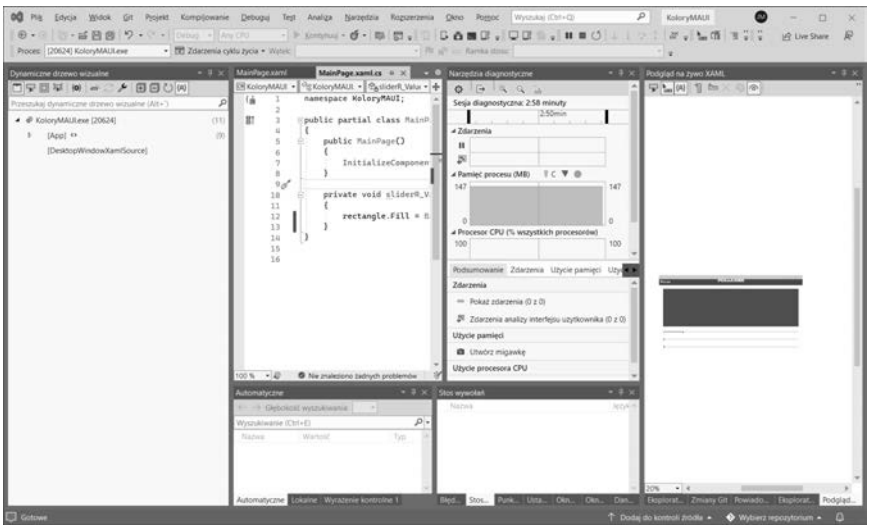
Wykorzystany przez nas prostokąt można pokolorować, zmieniając zarówno jego brzeg (atrybut/własność `Stroke`), jak i wypełnienie (`Fill`). W obu przypadkach należy wskazać obiekt pędzla, a nie tylko kolor. My zmieniamy jedynie wypełnienie, korzystając z predefiniowanego pędzla o jednolitym kolorze, zdefiniowanego w klasie `Brush`. Przekonajmy się, jak to będzie wyglądać — skompilujmy projekt, uruchamiając aplikację (klawisz *F5*) i zmieniając pozycję któregokolwiek suwaka (rysunek 1.5).



RYSUNEK 1.5. Zmiana koloru po przesunięciu suwaka

Pasek inspekcji

Przy okazji zwróćmy uwagę na pasek inspekcji widoczny pośrodku górnej krawędzi okna aplikacji uruchomionej w trybie debugowania (rysunek 1.5). Można go przesuwać i zwinąć, jeżeli przeszkadza. Zawiera siedem ikon kontrolujących inspekcję kodu XAML w trakcie działania aplikacji. Kliknięcie pierwszych dwóch ikon otwiera podokna Visual Studio. Pierwsza ikona otwiera podokno o nazwie *Dynamiczne drzewo wizualne* (ang. *Live Visual Tree*), zawierające aktualizowane na bieżąco drzewo kontrolek XAML działającej aplikacji. W tym drzewie widoczne są nie tylko kontrolki wstawione przez nas do kodu XAML projektu, ale również te, z których te kontrolki są zbudowane. Każdy element jest wymieniony z nazwy (jeżeli ją ma) i typu podanego w nawiasach kwadratowych (rysunek 1.6). Warto zwrócić uwagę, że to podokno ma pasek narzędzi z ikonami podobnymi do tych z paska inspekcji widocznego w oknie aplikacji (m.in. ikonę pozwalającą usunąć pasek z aplikacji oraz ikonę pozwalającą na ograniczenie pokazywanych elementów do tych zdefiniowanych w projekcie). Dla wygody podokno z drzewem można odpiąć od głównego okna Visual Studio i ustawić obok okna debugowanej aplikacji. Robi się to, „ciągnąc" za jego pasek tytułu. Druga ikona uruchamia podgląd interfejsu aplikacji na żywo (podokno *Podgląd na żywo XAML* także widoczne na rysunku 1.6), który może być szczególnie wygodny, gdy korzystamy z rzeczywistego urządzenia podłączonego do komputera kablem USB. Kolejne trzy ikony to przełączniki. Pierwszy włącza i wyłącza tryb selekcji w oknie debugowanej aplikacji. W trybie selekcji możemy zaznaczyć jakieś elementy XAML, z których jest zbudowany interfejs (kontrolkę lub pojemnik). Wskazanie jednego z nich w oknie aplikacji odzwierciedlane jest także w drzewie kontrolek. Po wybraniu elementu przełącznik wyłącza się.



RYSUNEK 1.6. Drzewo kontrolek XAML dla działającej aplikacji oraz aktualizowana na żywo lista własności

Drugi przełącznik pokazuje lub ukrywa strukturę interfejsu, uwidaczniając szczegóły elementów kontrolujących ułożenie kontrolek. W przypadku używanej przez nas siatki można dzięki temu zobaczyć jej podział na wiersze i kolumny. Natomiast użycie trzeciego przełącznika powoduje, że w podoknach z drzewem elementów oraz podglądu własności na żywo powinny być pokazywane kontrolki mające aktualnie „focus". Dwie ostatnie ikony pokazują informacje o stanie aplikacji. Szczególnie ważna jest pierwsza z nich (przedostatnia ze wszystkich ikon), pokazująca liczbę błędów wiązania. My jednak nie korzystamy z tego mechanizmu w tej aplikacji (tym zajmiemy się dopiero w rozdziale 3.). Ostatnia ikona pokazuje jedynie, czy możliwe jest skorzystanie z przeładowania aplikacji na gorąco — nowa umiejętność Visual Studio 2022.

Kontrola składowych koloru

Aby aplikacja uzyskała zaplanowaną funkcjonalność, metoda musi zmieniać kolor prostokąta odpowiednio do bieżących pozycji suwaków. Musimy zatem odczytać ich własności `Value`, ustalić na ich podstawie kolor i przypisać go do własności `Fill` prostokąta. Wartość `Value` jest typu `double`, podobnie jak wiele własności odpowiadających za pozycję i wygląd w WPF, UWP i MAUI (w odróżnieniu od starszej technologii Windows Forms). Domyślnie przyjmuje wartości z zakresu od `0` do `1` (domyślne wartości własności `Minimum` i `Maximum`). Taki zakres wartości nam odpowiada, więc nie będziemy go zmieniać. Alternatywą byłaby zmiana zakresu na 0 – 255 i ustalanie koloru z użyciem wartości typu `byte` (kolor i tak jest przechowywany w czterech bajtach: trzy składowe RGB i nieprzezroczystość w kanale alfa), ale zostańmy przy liczbach rzeczywistych.

Zmieńmy metodę zdarzeniową zgodnie ze wzorem z listingu 1.6. Użyta w niej klasa `Color` należy do przestrzeni nazw `Microsoft.Maui.Graphics`. Kiedyś tę przestrzeń należałoby dodać do sekcji poleceń `using` na początku pliku *MainPage.xaml.cs*. W nowszych wersjach VS 2022 w pliku *MainPage.cs.xaml* już jej nie ma (co nie oznacza, że takiego polecenia `using` nie możemy dodać sami, jeżeli zechcemy), a w zamian VS generuje dla każdej platformy plik *KoloryMAUI.GlobalUsings.g.cs*, w którym deklaruje użycie wszystkich potrzebnych przestrzeni nazw za pomocą poleceń typu `global using Microsoft.Maui.Graphics`. Modyfikator `global` na początku oznacza, że ta przestrzeń będzie widoczna we wszystkich plikach projektu.

Zaskakujące może być również to, że zamiast przestrzeni nazw otoczonej nawiasami klamrowymi na początku pliku znajduje się tylko formuła `namespace KoloryMAUI;`, która oznacza, że cała zawartość pliku należy do przestrzeni `KoloryMAUI`. Pamiętajmy także, aby zsynchronizować początkowy kolor prostokąta z pozycją suwaków. W tym celu po uruchomieniu programu wywołajmy metodę `sliderR_ValueChanged` z konstruktora klasy `MainPage` (listing 1.6).

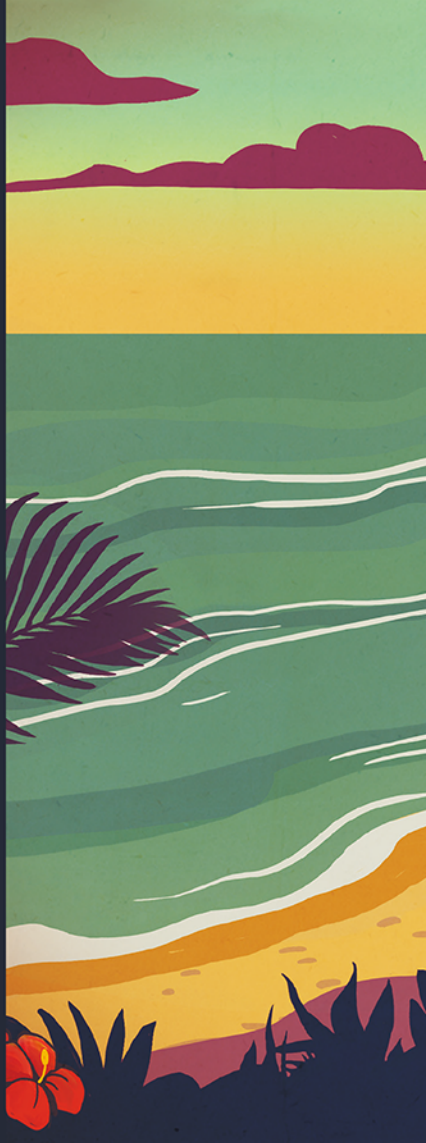
LISTING 1.6. Zmiana koloru prostokąta w zależności od pozycji suwaków

```
namespace KoloryMAUI;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        sliderR_ValueChanged(null, null);
    }

    private void sliderR_ValueChanged(object sender, ValueChangedEventArgs e)
    {
        Color color = Color.FromRgb(sliderR.Value, sliderG.Value, sliderB.Value);
        rectangle.Fill = new SolidColorBrush(color);
    }
}
```

Osoby, które dopiero zaczynają naukę C#, ale znają już C++, mogą mieć poważne wątpliwości co do metody z listingu 1.6, widząc w niej źródło wycieku pamięci. Wprawdzie na platformie .NET zarządzaniem pamięcią zajmuje się *garbage collector* (odśmiecacz) i cyklicznie usuwa z pamięci wszystkie obiekty, do których nie ma już referencji, jednak także w C# ciągłe tworzenie nowych obiektów może nie być najlepszym rozwiązaniem. Tworzenie nowego obiektu typu `SolidColor ↪Brush` (typ referencyjny) przy każdym poruszeniu suwakiem jest bowiem sporym wyzwaniem dla *garbage collectora*, który musi wówczas zwalniać z pamięci poprzednio używane obiekty. Lepiej byłoby modyfikować własność opisującą kolor w istniejącym obiekcie. Jednak w pierwszej aplikacji zostawmy ten kod w takiej postaci, w jakiej jest w tej chwili — chodzi teraz przede wszystkim o oswojenie kontrolek .NET MAUI.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Wejdź ze swoim kodem na wiele platform. Równocześnie!



Oferowane przez Microsoft oprogramowanie Visual Studio ma spore możliwości. W tym środowisku można samodzielnie projektować aplikacje, aplikacje sieciowe, usługi sieciowe i serwisy internetowe — czyli praktycznie wszystko. W ramach Visual Studio 2022 możliwa jest praca z .NET MAUI, międzyplatformową strukturą do tworzenia natywnych aplikacji mobilnych i klasycznych (z użyciem języków C# i XAML).

Z tego podręcznika dowiesz się, jak w ramach Visual Studio 2022 korzystać z .NET MAUI. Poznasz zasady tworzenia projektów i uruchamiania aplikacji w systemach Windows i Android, jak również przechowywania danych w plikach XML. Zaznajomisz się też z architekturą MVVM i z odpowiednimi poleceniami. Przyjrzysz się zachowaniom, własnościom zależności i własnościom doczepianym. Popracujesz z multimediami, zbadasz stan urządzenia i odczytasz czujniki. Wreszcie skupisz się na *Reversi* — na silniku tej gry, jej widoku w .NET MAUI, a także wykrywaniu szczególnych sytuacji. Wisienką na torcie będzie odpalenie gry na komputerze i dla systemu Android.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9026-3

Cena: 49,90 zł