Uniwersytet Mikołaja Kopernika
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Instytut Fizyki UMK w Toruniu

Mateusz Dudek
nr albumu: 244623

Praca magisterska
na kierunku Informatyka stosowana

Aplikacje Uniwersalne - Tutorial

Opiekun pracy dyplomowej
dr hab. Jacek Matulewski
Wydział Fizyki, Astronomii i Informatyki Stosowanej UMK

Toruń 2015

Pracę przyjmuję i akceptuję	Potwierdzam złożenie pracy dyplomowej
.....	.....
data i podpis opiekuna pracy	*data i podpis pracownika dziekanatu*

Dziękuję mojemu promotorowi,
dr hab. Jackowi Matulewskiemu
za poświęcony czas oraz cenne rady
związane z niniejszą pracą, a także za
wsparcie i wyrozumiałość.

Spis treści

1. Wstęp

Aplikacje Uniwersalne udostępniają programistom zestaw strategii współdzielenia kodu, dzięki którym dana aplikacja może zostać łatwo przeniesiona na różne platformy z minimalną liczbą modyfikacji. Dzięki nim proces tworzenia oraz rozwijania wieloplatformowego oprogramowania jest znacznie szybszy, a programistom jest łatwiej dotrzeć do szerszej grupy użytkowników. Niniejsza praca skupi się na platformie firmy Microsoft[1] wprowadzonej wraz z systemem operacyjnym Windows 8.1. Platforma ta umożliwia tworzenie Aplikacji Uniwersalnych na telefony, tablety oraz komputery stacjonarne z systemem operacyjnym Windows 8.1 lub Windows Phone 8.1.

Niniejsza praca jest skierowana przede wszystkim do programistów języka C#, którzy nie posiadają żadnego doświadczenia w tworzeniu Aplikacji Uniwersalnych. W drugim rozdziale tej pracy omówione zostaną podstawy języka XAML w stopniu wystarczającym do zrozumienia przykładów prezentowanych w dalszej części pracy. Rozdział ten ułatwi także naukę bardziej zaawansowanych mechanizmów tego języka. W trzecim rozdziale omówione zostaną ogólne idee aplikacji uniwersalnych oraz podstawowe mechanizmy wykorzystywane podczas ich tworzenia. Czwarty rozdział dotyczy kilku bardziej zaawansowanych zagadnień, które pomogą pełniej wykorzystać możliwości Aplikacji Uniwersalnych. W ostatnim rozdziale zaprezentowany zostanie proces tworzenia przykładowej aplikacji, co powinno ułatwić zrozumienie omówionych w tej pracy zagadnień.

Z uwagi na fakt, iż temat tej pracy jest bardzo obszerny, opisane zostaną jedynie wybrane zagadnienia dotyczące Aplikacji Uniwersalnych. Praca ta stanowi jedynie fragment całego tutorialu. Zagadnienia, które nie zostaną tutaj poruszone, mogą zostać odnalezione w pracach dyplomowych pisanych przez Pawła Stopińskiego oraz Rafała Wołowskiego.

[1] Istnieją także inne implementacje Aplikacji Uniwersalnych, jak na przykład rozwiązanie firmy Xamarin umożliwiające tworzenie Aplikacji Uniwersalnych korzystając z projektu Mono

2. Język XAML

Z uwagi na duże znaczenie języka XAML w aplikacjach uniwersalnych praca ta rozpocznie się od wprowadzenia do tego języka.

2.1. Wstęp do języka XAML

XAML, czyli Extensible Application Markup Language jest stworzonym przez firmę Microsoft językiem znacznikowym, będącym rozszerzeniem języka XML służącym do opisu interfejsu graficznego użytkownika (GUI) w aplikacjach Windows Store (w tym Aplikacjach Uniwersalnych) oraz Windows Presentation Foundation (WPF). Przy pomocy języka XAML programista może opisywać kontrolki, animacje, kształty lub jakiekolwiek inne elementy interfejsu, nie korzystając z kodu napisanego w języku C#. Dzięki temu interfejs graficzny jest zdefiniowany w bardziej czytelny sposób oraz jest oddzielony od logiki biznesowej programu, co znacznie ułatwia dalsze rozwijanie kodu w przyszłości. Należy jednak zwrócić uwagę na pewne ograniczenia języka XAML. Z uwagi na deklaratywny charakter tego języka programista może definiować interakcję z użytkownikiem jedynie w bardzo ograniczony sposób, na przykład przy pomocy prostych animacji wyświetlanych na podstawie określonych zdarzeń. Nie jest możliwe wyświetlenie wyniku działania określonego algorytmu lub dynamiczne tworzenie nowych kontrolek w zależności od działań użytkownika; w tym celu interfejs graficzny musi być modyfikowany z poziomu języka C#. [1]

2.2. Skalowanie interfejsu graficznego

W języku XAML rozmiar oraz pozycja elementów nie jest definiowana przy pomocy pikseli. Do tego celu wykorzystuje się jednostki zwane Device Independent Pixels, w skrócie DIP [2]. Faktyczny rozmiar lub pozycja poszczególnych elementów w pikselach jest zależna nie tylko od wartości DIP ale także od ustawień DPI (dots per inch) systemu operacyjnego. Dzięki takiemu rozwiązaniu wygląd interfejsu graficznego jest jednakowy na wszystkich urządzeniach, niezależnie od rozdzielczości oraz ustawień DPI systemu operacyjnego. Wartość w pikselach obliczana jest według następującego wzoru:

$$\boldsymbol{piksele} = \frac{\boldsymbol{DIP} * \boldsymbol{DPI}}{\mathbf{96.0}} \quad (1)$$

2.3. Pojemniki (Container Controls)

Podstawowym mechanizmem służącym do organizacji elementów interfejsu graficznego są tak zwane pojemniki (ang. Container Controls). Jest to specjalny typ kontrolek, których jedynym zadaniem jest przechowywanie i pozycjonowanie innych kontrolek zgodnie z regułami zdefiniowanymi przez programistę. Wprawdzie większość kontrolek także umożliwia dodawanie innych elementów do ich zawartości (jest to dokładniej opisane w rozdziale 2.4), to pojemniki wyróżniają się ze względu na mechanizmy, które umożliwiają łatwą organizację wielu kontrolek [3]. Omówione w tej pracy pojemniki nie są jedynymi kontrolkami umożliwiającymi przechowywanie wielu kontrolek jednocześnie. Każda kontrolka dziedzicząca po klasie `Panel` z przestrzeni nazw `Windows.UI.XAML.Controls` umożliwia przechowywanie wielu kontrolek [4]. Zestaw kontrolek, w tym Container Controls, oferowanych przez platformę WinRT jest nieco bardziej ograniczony niż w przypadku Windows Presentation Foundation. Jest on jednak w zupełności wystarczający, niezależnie od efektów jakie chcemy osiągnąć. W tej pracy opisane zostaną jedynie Container Controls, które dostępne są na platformie WinRT.

2.3.1. Siatka (Grid)

Pojemnik ten jest siatką składającą się z określonej liczby wierszy oraz kolumn. W komórkach siatki umieszczamy dowolne kontrolki, których pozycja oraz rozmiar będą przez nią automatycznie dobierane. Pierwszą rzeczą, która powinna się znaleźć wewnątrz siatki jest definicja wierszy oraz kolumn. Programista musi wedle uznania zdefiniować szerokość kolumn oraz wysokość wierszy, jednak nie jest to konieczne jeżeli siatka ma się składać tylko z jednej komórki. Przykładowa konfiguracja tej kontrolki w języku XAML wygląda następująco:

```
<Grid>
    <!--Definicja wierszy-->
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="50"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="3*"/>
    </Grid.RowDefinitions>

    <!--Definicja kolumn-->
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="50"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="3*"/>
```

```
    </Grid.ColumnDefinitions>

    <!--Tutaj można umieścić kontrolki-->
</Grid>
```

W powyższym przykładzie zdefiniowana została siatka zawierająca cztery wiersze oraz cztery kolumny. Wysokość pierwszego wiersza została ustawiona na `Auto`. Oznacza to, że wysokość tego wiersza zawsze przyjmie minimalną wartość, która pozwoli pomieścić wszystkie zawarte w tym wierszu kontrolki. Wysokość drugiego wiersza została ustawiona na stałą wartość 50 DIP i ta wartość pozostanie bez zmian niezależnie od wysokości kontrolek zawartych w tym wierszu. Cała pozostała wysokość siatki zostanie przydzielona dla wiersza trzeciego oraz czwartego. Wiersz czwarty otrzyma jednak trzykrotnie więcej miejsca niż wiersz trzeci. Identyczna sytuacja została zaprezentowana dla kolumn. Należy tylko zwrócić uwagę na to, że w przypadku kolumn zamiast parametru `Height` (wysokość) definiujemy parametr `Width` (szerokość).

Identyczną siatkę można zdefiniować także przy pomocy języka C#. W takim przypadku kod wyglądałby następująco:

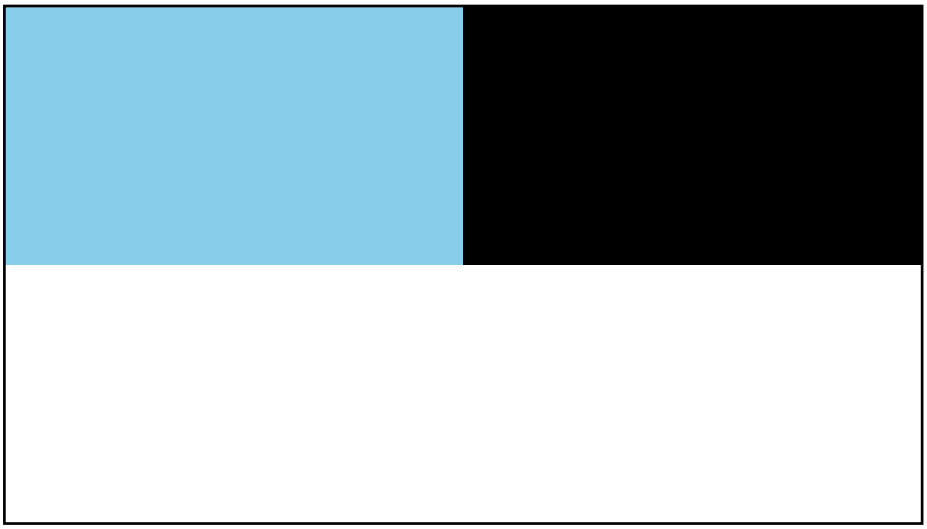
```
var grid = new Grid();

//definicja wierszy
var row = new RowDefinition { Height = new GridLength(1, GridUnitType.Auto) };
grid.RowDefinitions.Add(row);
row = new RowDefinition { Height = new GridLength(50) };
grid.RowDefinitions.Add(row);
row = new RowDefinition { Height = new GridLength(1, GridUnitType.Star) };
grid.RowDefinitions.Add(row);
row = new RowDefinition { Height = new GridLength(3, GridUnitType.Star) };
grid.RowDefinitions.Add(row);

//definicja kolumn - analogicznie
```

Powyższy zapis tworzy identyczną siatkę jak w przypadku wcześniej zaprezentowanego przykładu, jednak z uwagi na większą czytelność kodu interfejs graficzny powinien być tworzony przy pomocy języka XAML, kiedy tylko jest to możliwe. Należy też pamiętać, że modyfikacja interfejsu graficznego przy pomocy języka C# musi odbywać się po pełnym zainicjowaniu modyfikowanego elementu, czyli nie wcześniej niż po wykonaniu metody `InitializeComponent`.

Dodawanie elementów do siatki jest bardzo łatwe. Konieczne jest sprecyzowanie w którym wierszu oraz w której kolumnie ten element ma się znaleźć. Numerację rozpoczynamy od 0. Oto przykład:

```
<Grid Name="GridMain">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Rectangle Grid.Row="0" Grid.Column="0" Fill="SkyBlue"/>
    <Rectangle Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Fill="White"/>
</Grid>
```

Jak widać, dodane zostały dwa prostokąty (kontrolki `Rectangle`). Pierwszy prostokąt dodany został do pierwszego wiersza (o zerowym indeksie) i pierwszej kolumny, natomiast drugi prostokąt dodany został do drugiego wiersza i pierwszej kolumny. W przypadku drugiego prostokąta zdefiniowany został jednak dodatkowy parametr `Grid.ColumnSpan="2"`. Oznacza to, że drugi prostokąt będzie w rzeczywistości umieszczony w dwóch kolumnach: zerowej i pierwszej jednocześnie. Efekt jest widoczny na *Rysunku 1*.



Rysunek 1: Przykład zastosowania Grida

Możliwe jest także dodawanie nowych elementów do siatki dynamicznie, w dowolnym momencie po zainicjowaniu siatki. Jeżeli chcielibyśmy dodać nowy prostokąt do naszej siatki, możemy skorzystać z następującego kodu napisanego w języku C#:

```
var rectangle = new Rectangle();
rectangle.Fill = new SolidColorBrush(Colors.Green);

Grid.SetRow(rectangle, 0);
Grid.SetColumn(rectangle, 1);
```

```
GridMain.Children.Add(rectangle);
```

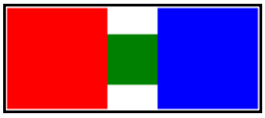
Powyższy kod może zostać umieszczony w pliku .cs powiązanym z plikiem .xaml, w którym zdefiniowany został element `GridMain`. Po wykonaniu, do zerowego wiersza oraz pierwszej kolumny, czyli do wolnej komórki oznaczonej na *Rysunku 1* kolorem czarnym dodany zostanie zielony prostokąt.

2.3.2. StackPanel

`StackPanel` jest kontrolką, która pozwala na łatwe rozmieszczanie elementów obok siebie w pionie lub w poziomie. Jego rozmiar dopasowuje się automatycznie do jego zawartości. Jeżeli orientacja `StackPanelu` została ustawiona na poziomą wszystkie zawarte w nim kontrolki muszą mieć zdefiniowaną szerokość, a wysokość `StackPanelu` będzie równa wysokości najwyższej kontrolki, która się w nim znajduje. Natomiast jeżeli orientacja `StackPanelu` została ustawiona na pionową, wszystkie zawarte w nim kontrolki muszą mieć zdefiniowaną wysokość, a szerokość `StackPanelu` będzie równa szerokości najszerszej zawartej w nim kontrolki. Przykład zastosowania:

```
<StackPanel Orientation="Horizontal">
    <Rectangle Fill="Red" Width="50" Height="50"/>
    <Rectangle Fill="Green" Width="25" Height="25"/>
    <Rectangle Fill="Blue" Width="50"/>
</StackPanel>
```

W powyższym przykładzie zdefiniowany został `StackPanel` o orientacji poziomej (horizontal), a następnie zostały umieszczone w nim trzy prostokąty. Wysokość niebieskiego prostokąta nie została zdefiniowana, zatem dopasuje się ona do wysokości StackPanelu. Efekt jest widoczny na *Rysunku 2*.



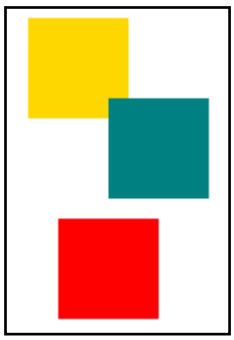
Rysunek 2: Przykład zastosowania StackPanelu

2.3.3. Canvas

Canvas (z ang. płótno) jest kontrolką, na której można umieszczać elementy, których pozycje wyznaczają współrzędne X i Y. Współrzędna X jest określana jako odległość od lewej krawędzi Canvasu, natomiast współrzędna Y jest określana względem jego górnej krawędzi. Przykład zastosowania:

```
<Canvas>
    <Rectangle Fill="Gold" Width="50" Height="50" Canvas.Top="10"
        Canvas.Left="20"/>
    <Rectangle Fill="Teal" Width="50" Height="50" Canvas.Top="50"
        Canvas.Left="60"/>
    <Rectangle Fill="Red" Width="50" Height="50" Canvas.Top="110"
        Canvas.Left="35"/>
</Canvas>
```

Na Canvasie umieszczone zostały trzy prostokąty o identycznych rozmiarach. Każdemu z tych prostokątów zostały przypisane odpowiednie współrzędne tak, aby dwa z tych prostokątów częściowo na siebie nachodziły. O tym, który z tych prostokątów znajdzie się na wierzchu decyduje kolejność w jakiej zostały dodane do pojemnika. Możemy na to jednak wpłynąć poprzez odpowiednie ustawienie parametru `Canvas.ZIndex`. Elementy z wyższą wartością tego parametru zostaną wyświetlone przed elementami z niższą wartością. W powyższym przykładnie ten parametr pozostał niezmieniony. Efekt jest widoczny na *Rysunku 3*.



Rysunek 3: Przykład zastosowania Canvasu

2.4. Zawartość kontrolek

Wiele kontrolek oferowanych przez platformę WinRT, między innymi `Button`, `Border` oraz `CheckBox` dziedziczy po klasie `ContentControl`. Kontrolki te posiadają pole `Content`, które określa ich zawartość [4]. Do tego pola możemy odwołać się jawnie, ustawiając wartość parametru `Content` lub niejawnie, wstawiając jego wartość pomiędzy znacznikami końca oraz początku elementu. Na przykład, poniższy zapis reprezentuje StackPanel zawierający dwa identyczne przyciski:

```
<StackPanel>
    <Button Content="Przycisk"/>
    <Button>Przycisk</Button>
</StackPanel>
```

W przypadku, gdy zawartość naszej kontrolki wstawiamy pomiędzy znacznikami początku oraz końca elementu, wartość ta nie zawsze zostanie przypisana do właściwości `Content`. Na przykład, w przypadku kontrolki `TextBlock` zawartość naszej kontrolki określa właściwość `Text`. W przypadku tworzenia własnej kontrolki, właściwość odpowiadającą zawartości kontrolki możemy wskazać oznaczając naszą klasę atrybutem `ContentProperty`, na przykład:

```
[ContentProperty(Name = "NazwaWlasciwosci")]
```

Polu `Content` możemy przypisać nie tylko wartość tekstową (string), ale również inną kontrolkę. Oznacza to, że możliwe jest łatwe tworzenie różnych użytecznych kombinacji istniejących już kontrolek. Poniżej zaprezentowany jest przykładowy kod XAML wraz z efektem jego wykonania na *Rysunku 4*:

```
<Button Background="CornflowerBlue">
    <StackPanel>
        <ProgressBar Value="50" Maximum="100" Foreground="Cornsilk"/>
        <TextBlock Text="Przycisk"/>
    </StackPanel>
</Button>
```

Rysunek 4: Przykładowa zawartość przycisku

2.5. Zasoby - resources

Język XAML umożliwia przechowywanie obiektów w zasobach. Możliwe jest przechowywanie różnego rodzaju obiektów, które mogą być później wykorzystane w różnych częściach aplikacji. W momencie, gdy w kodzie XAML przypisujemy atrybutowi danego elementu jakąś wartość, tworzony jest nowy obiekt. Na przykład, jeżeli nadajemy nową wartość atrybutu `Background` kontrolki `Button`, w rzeczywistości tworzony jest nowy obiekt klasy `Brush` i to ten obiekt staje się wartością atrybutu. Zatem jeżeli chcielibyśmy przypisać jednakowy kolor do atrybutu `Background` w kilku różnych elementach, dobrym pomysłem byłoby przechowanie tego koloru w lokalnym zasobie, a następnie odwołanie się do tego zasobu tam, gdzie jest to konieczne. Dzięki temu oszczędzimy pamięć operacyjną oraz ułatwimy dalsze rozwijanie kodu w przyszłości. Zasoby mogą być defi-

niowane globalnie w pliku *App.xaml* lub lokalnie dla danej strony lub kontrolki. Przykład zasobu zdefiniowanego lokalnie dla siatki:

```
<Grid Name="GridMain" Background="Black">
    <Grid.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="SteelBlue"/>
    </Grid.Resources>

    <StackPanel>
        <Button Background="{StaticResource MyBrush}" Content="Przycisk 1"/>
        <Button Background="{StaticResource MyBrush}" Content="Przycisk 2"/>
    </StackPanel>
</Grid>
```

Każdy zasób musi mieć przypisany klucz, po którym będzie można się do niego odwołać. W tym przypadku kluczem jest `MyBrush`. Jest to zasób lokalny, co oznacza, że poza obszarem siatki nie będzie się można do niego odwołać[2]. W celu stworzenia globalnego zasobu, dostępnego w całej aplikacji należy go umieścić w pliku *App.xaml*. Do zdefiniowanych zasobów można się także odwołać z poziomu kodu C#, na przykład:

```
var button = new Button();
button.Background = (SolidColorBrush)GridMain.Resources["MyBrush"];
```

2.6. Style

W rozdziale 2.5 omówiony został mechanizm zasobów, który pozwala na definiowanie różnego rodzaju danych współdzielonych przez zbiór obiektów. W przypadku, gdy konieczna jest modyfikacja większej liczby parametrów, zazwyczaj lepszym sposobem jest skorzystanie ze stylu. Style definiują wartości określonego zbioru parametrów dla kontrolek danej klasy. Taki styl można następnie przypisać do interesujących nas elementów, w wyniku czego wartości ich atrybutów ulegną zmianie. Oto przykład:

```
<Grid Background="Black">
    <Grid.Resources>
        <Style TargetType="Button">
            <Setter Property="Background" Value="SteelBlue"/>
        </Style>
    </Grid.Resources>

    <StackPanel>
        <Button Content="Przycisk 1"/>
        <Button Content="Przycisk 2"/>
    </StackPanel>
</Grid>
```

[2] Elementy interfejsu graficznego także nie mają możliwości odwołania się do zasobów zdefiniowanych wewnątrz nich samych. Dopiero elementy w nich zawarte mogą odwołać się do tych zasobów.

Wewnątrz siatki zdefiniowany został styl definiujący kolor tła dla wszystkich przycisków które się w nim znajdują. Elementy znajdujące się poza siatką nie ulegną modyfikacji. Podobnie jak w przypadku pozostałych zasobów styl może być zdefiniowany globalnie w pliku *App.xaml*, przez co będzie dostępny dla wszystkich elementów danego typu w aplikacji. Możliwe jest też przypisanie stylowi klucza. To spowoduje, że nie będzie on automatycznie stosowany do elementów z dostępnego zakresu, a jedynie do elementów w których został jawnie wskazany.

```
<Grid Name="GridMain" Background="Black">
    <Grid.Resources>
        <Style x:Key="MyStyle" TargetType="Button">
            <Setter Property="Background" Value="SteelBlue"/>
        </Style>
    </Grid.Resources>

    <StackPanel>
        <Button Content="Przycisk 1" Style="{StaticResource MyStyle}"/>
        <Button Content="Przycisk 2"/>
    </StackPanel>
</Grid>
```

W powyższym przykładzie stylowi został nadany klucz `MyStyle`. Pierwszy przycisk jako jedyny odnosi się do tego stylu korzystając z klucza, więc tylko w jego przypadku ten styl zostanie zastosowany. Wygląd drugiego przycisku pozostanie bez zmian.

2.7. Wiązanie (bindowanie)

Wiązanie (ang. binding) jest mechanizmem umożliwiającym współdzielenie oraz synchronizację danych pomiędzy elementami [5]. W przeciwieństwie do zasobów umożliwia on współdzielenie dynamicznych danych. Wartość własności danego obiektu może zostać połączona z dowolną własnością tego innego obiektu, jeżeli wartości te są tych samych typów. Mechanizmy umożliwiające wiązanie niekompatybilnych typów oraz pozwalające na przetwarzanie danych przekazywanych w procesie wiązania opisane zostały w rozdziale 2.7.1. W momencie, kiedy wartość będąca źródłem ulegnie zmianie, wywołane zostanie zdarzenie `PropertyChanged` zdefiniowane przez interfejs `INotifyPropertyChanged`. Dzięki temu wszystkie własności podpięte do tej wartości zostaną uaktualnione. Przykład:

```
<StackPanel>
    <TextBox Name="MyBox"/>
    <TextBlock Text="{Binding ElementName=MyBox, Path=Text}"/>
</StackPanel>
```

W powyższym przykładzie stworzone zostały dwa elementy: `TextBox` oraz `TextBlock`. Właściwość `Text` kontrolki `TextBlock` została podpięta do zawartości pola tekstowego. Dzięki temu, jeżeli zawartość pola tekstowego ulegnie zmianie, tekst wyświetlany przez `TextBlock` także zostanie automatycznie uaktualniony. Jest to oczywiście najprostszy przykład. Wiązanie posiada szereg dodatkowych parametrów, które pozwalają programiście na dostosowanie procesu synchronizacji do własnych potrzeb.

2.7.1. Konwertery

Konwertery są mechanizmem umożliwiającym modyfikację danych przekazywanych w procesie wiązania. Dzięki temu możliwe jest wiązanie danych o niekompatybilnych typach oraz dowolne przetwarzanie danych przed przekazaniem ich do odpowiedniego atrybutu. W celu stworzenia konwertera konieczne jest napisanie nowej klasy implementującej interfejs `IValueConverter` lub dostępny tylko w WPF interfejs `IMultiValueConverter`. Konwertery implementujące `IValueConverter` są wykorzystywane przy wiązaniu pojedynczej wartości, natomiast konwertery implementujące `IMultiValueConverter` są wykorzystywane przy wiązaniu wielu wartości jednocześnie. W obu przypadkach po dokonaniu konwersji zwracana jest pojedyncza wartość. Warto także zwrócić uwagę na to, że konwersja może odbywać się w dwie strony. Z tego powodu interfejsy konwerterów definiują dwie metody: `Convert` oraz `ConvertBack`. Przykładowy konwerter dzielący otrzymaną wartość liczbową przez 2:

```
namespace ExampleApp.Converters
{
    class HalfValueConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
                              string language)
        {
            return (double) value/2;
        }

        public object ConvertBack(object value, Type targetType, object
                                  parameter, string language)
        {
            return (double) value*2;
        }
    }
}
```

Aby skorzystać z tego konwertera w pliku XAML, konieczne jest sprecyzowanie przestrzeni nazw, w której ten konwerter się znajduje. Dla powyższego przypadku zapis jest następujący:

```
xmlns:conv="using:ExampleApp.Converters"
```

Po dodaniu referencji, konwerter może zostać dodany do zasobów oraz wykorzystany w kodzie:

```
<StackPanel>
    <StackPanel.Resources>
        <conv:HalfValueConverter x:Key="MyConverter"/>

        <Style TargetType="Rectangle">
            <Setter Property="Width" Value="50"/>
            <Setter Property="Height" Value="50"/>
            <Setter Property="Margin" Value="5"/>
            <Setter Property="Fill" Value="Red"/>
        </Style>
    </StackPanel.Resources>

    <Slider Name="MySlider" Maximum="1" StepFrequency="0.01"/>
    <Rectangle Opacity="{Binding ElementName=MySlider, Path=Value}"/>
    <Rectangle Opacity="{Binding ElementName=MySlider, Path=Value,
                          Converter={StaticResource MyConverter}}"/>
</StackPanel>
```

W powyższym przykładzie stworzone zostały dwa prostokąty. Przezroczystość obu prostokątów została związana z własnością `Value` znajdującego się nad nimi suwaka. Drugi prostokąt poddaje jednak wartość suwaka konwersji przy pomocy stworzonego wcześniej konwertera. Oznacza to, że wartość parametru `Opacity` drugiego prostokąta będzie w rzeczywistości o połowę mniejsza od wartości wskazywanej przez suwak.

2.7.2. Ustalanie kierunku wiązania

Parametr `Mode` konwertera umożliwia ustawienie kierunku bindowania. Możliwe wartości to:

- `OneWay` - zmiana wartości źródłowej spowoduje odpowiednią zmianę wartości docelowej. Zmiana wartości docelowej nie wpłynie w żaden sposób na wartość źródłową,
- `OneWayToSource` - wartość ta dostępna jest tylko w Windows Presentation Foundation. Zmiana wartości docelowej spowoduje odpowiednią zmianę wartości źródłowej. Zmiana wartości źródłowej nie wpłynie w żaden sposób na wartość docelową,
- `TwoWay` - zmiana wartości źródłowej lub docelowej zostanie zastosowana do obu wartości,

- `OneTime` - wartość źródłowa zostanie jednorazowo przypisana do elementu docelowego podczas jego inicjalizacji. Późniejsze zmiany wartości źródłowej nie będą wpływały w żaden sposób na wartość docelową.

Jeżeli wartość parametru `Mode` nie zostanie sprecyzowana przez programistę, w przypadku aplikacji WinRT standardowo przyjmie on wartość `OneWay`, natomiast w przypadku aplikacji korzystającej z WPF - `TwoWay`.

2.7.3. Ustalanie trybu uaktualniania

Przy pomocy parametru `UpdateSourceTrigger` możliwe jest ustalenie trybu uaktualniania wartości kontrolki. Możliwe wartości to [3]:

- `PropertyChanged` - nowa wartość zostaje przypisana natychmiastowo,
- `Explicit` - nowa wartość zostaje przypisana dopiero po wywołaniu metody `UpdateSource`,
- `Default` - standardowa wartość parametru. W przypadku większości właściwości jest ona jednoznaczna z `PropertyChanged`, jednak dla niektórych właściwości efekt może być różny. Na przykład, dla właściwości `Text` kontrolki `TextBox` wartość docelowa zostanie uaktualniona w momencie wywołania metody `LostFocus`. Zachowanie jest definiowane przez metadane właściwości.

3. Wprowadzenie do aplikacji uniwersalnych

3.1. Windows Runtime

Windows Runtime (WinRT) jest zestawem nowoczesnych API (Application Programming Interface) umożliwiających łatwe tworzenie aplikacji dla systemów operacyjnych z rodziny Windows. Zadaniem WinRT nie jest zastąpienie Win32 API, a jedynie stworzenie jednolitego, zamkniętego środowiska dla aplikacji, niezależnie od platformy na której są one uruchamiane [2]. Główne cechy architektury WinRT to [3]:

- wszystkie API zostały zaprojektowane z myślą o działaniu asynchronicznym. Korzystający z WinRT kod wykonujący się dłużej niż 50ms także powinien być wykonywany asynchronicznie [2],
- aplikacje działają w odizolowanym środowisku i są przystosowane do umieszczenia w Windows Store,
- WinRT wykorzystuje język znaczników XAML (zob. rozdział 2), który może zostać wykorzystany do stworzenia interfejsu graficznego użytkownika. Do tego celu można wykorzystać także HTML lub DirectX,
- znaczna część kodu jest łatwo przenaszalna pomiędzy różnymi platformami korzystającymi z architektury WinRT,
- funkcje API mogą być wywoływane z poziomu wielu różnych języków programowania, jednak oficjalnie wspieranymi przez Microsoft językami są C#, C++ oraz Javascript.

Windows 8.1 jest nadal całkowicie kompatybilny z klasycznymi aplikacjami, jednak takie aplikacje nie są aplikacjami uniwersalnymi i nie można ich umieścić w Windows Store. Należy też zwrócić uwagę na fakt, że aplikacje korzystające z architektury WinRT muszą być zwykłymi aplikacjami użytkowymi. Oznacza to, że korzystając z architektury WinRT nie można napisać sterowników lub usług systemowych, a z uwagi na bezpieczeństwo, oraz fakt, iż aplikacje te działają w zamkniętym środowisku, na programistę narzucane są pewne ograniczenia, na przykład brak bezpośredniego dostępu do dysku [4].

3.2. Języki programowania

W rozdziale 3.1 zostało wspomniane, że API WinRT mogą być wywoływane z poziomu wielu różnych języków programowania. Oficjalnie jednak wspierane przez firmę Microsoft są trzy języki programowania. Zależnie od wybranego języka na programistę zostanie także narzucone technologie, w których będzie musiała zostać wykonana warstwa prezentacji naszej aplikacji. Zależności są następujące:

Język logiki biznesowej	Język warstwy prezentacji
C#	XAML oraz C#
C++	XAML lub DirectX
Javascript	HTML oraz CSS

Tabela 1: Zależność pomiędzy językami logiki biznesowej a językami warstwy prezentacji

Decyzja odnoście wykorzystywanego języka jest oczywiście zależna od znajomości danej technologii przez programistę, chociaż niektóre języki mogą okazać się lepszym wyborem zależnie od wykonywanego projektu. W tej pracy zostanie wykorzystany język C# wraz z warstwą prezentacji wykonaną w języku XAML.

3.3. Cechy aplikacji uniwersalnych

Przed przystąpieniem do pracy z aplikacjami uniwersalnymi należałoby zwrócić uwagę na pewne cechy tych aplikacji:

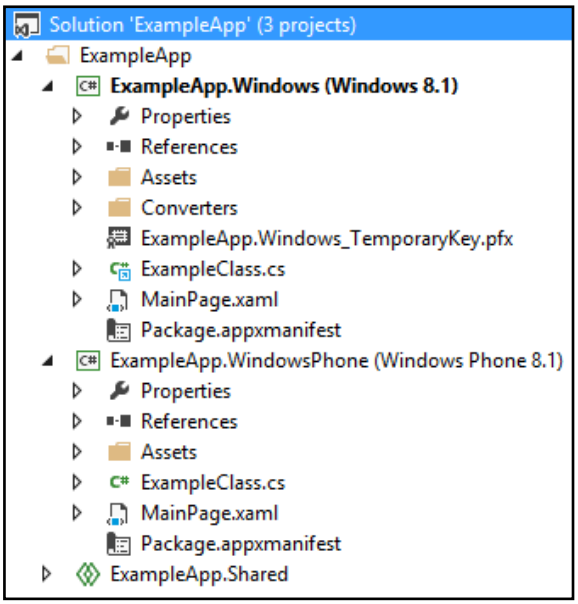
- Z uwagi na fakt, że aplikacja uruchamiana jest na zupełnie różnych platformach konieczne jest wygenerowanie osobnych plików binarnych odpowiednich dla danej platformy.
- W przypadku umieszczenia aplikacji w Windows Store, pojedynczy zakup aplikacji pozwala użytkownikowi na korzystanie z tej aplikacji zarówno na Windows 8.1 jak i Windows Phone 8.1. Zakupione mikrotransakcje także będą współdzielone pomiędzy platformami jeżeli tylko jest to możliwe.
- Dane synchronizowane z chmurą (roaming data) również będą synchronizowane pomiędzy wszystkimi urządzeniami niezależnie od platformy sprzętowej.
- Nazwa aplikacji jest jednakowa na wszystkich platformach.
- Do tworzenia aplikacji uniwersalnych wymagany jest Windows 8.1 oraz Visual Studio co najmniej w wersji 2013 Update 2.

3.4. Strategie współdzielenia kodu

Główną zaletą Aplikacji Uniwersalnych z punktu widzenia dewelopera jest przyśpieszenie procesu tworzenia aplikacji na wiele platform jednocześnie. W tym celu zostało zaprojektowanych kilka strategii współdzielenia kodu, które nie tylko zapobiegają duplikowaniu kodu, ale również umożliwiają wprowadzanie zmian w kodzie oraz poprawianie błędów w kilku projektach jednocześnie. W tym rozdziale zostaną omówione trzy strategie współdzielenia kodu: współdzielone pliki, współdzielony projekt oraz Portable Class Library (PCL).

3.4.1. Współdzielone pliki

Linked files, czyli współdzielone pliki są najprostszą strategią współdzielenia kodu. Obecnie nie są one zalecane przez Microsoft z uwagi na lepsze alternatywy, które zostaną omówione w następnych podrozdziałach. Nadal istnieje jednak możliwość ich wykorzystania. Linked files są pojedynczymi plikami, które dostępne są jednocześnie w różnych projektach. W celu dodania odwołania do istniejącego pliku, należy kliknąć prawym przyciskiem myszy na projekt a następnie z menu *Add* wybrać opcję *Existing Item*. W oknie dialogowym, które zostanie wyświetlone należy odszukać interesujący nas plik a następnie rozwinąć menu przycisku *Add* i wybrać opcję *Add as link*. Przykładowy efekt jest widoczny na *Rysunku 5*.



Rysunek 5: Przykład współdzielenia pliku ExampleClass.cs

W powyższym przykładzie do projektu przeznaczonego dla Windows Phone dodany został plik *ExampleClass.cs*, natomiast do projektu przeznaczonego dla Windows 8.1 dodany został odsyłacz do tego pliku. Zmiany dokonane w tym pliku zostaną zatem zastosowane jednocześnie do obu projektów. Skorzystanie z tej strategii współdzielenia kodu jest zalecane jedynie dla bardzo niewielkiej liczby plików. W sytuacji, gdy konieczne jest współdzielenie większej liczby plików najlepszym sposobem jest skorzystanie ze współdzielonego projektu, który omówiony został w podrozdziale 3.4.3

3.4.2. Portable Class Library

Portable Class Library, w skrócie PCL jest pojedynczą biblioteką, która może zostać wykorzystana na różnych platformach bez konieczności ponownej kompilacji. W celu stworzenia nowego projektu PCL przystosowanego do użycia w Aplikacjach Uniwersalnych, w oknie tworzenia projektu jako szablon należy wybrać *Class Library (Portable for Universal Apps)*. Taki projekt jest identyczny z projektem stworzonym za pomocą szablonu *Class Library (Portable)*. Jedyna różnica polega na tym, że automatycznie wybrane są platformy docelowe dla Aplikacji Uniwersalnych. Możliwa jest też zmiana platform docelowych we właściwościach projektu, w zakładce Library. Projekt PCL może zostać wykorzystany bezpośrednio w solucji, w której się on znajduje lub po kompilacji dodany jako referencja do zewnętrznego projektu kompatybilnego z jego docelowymi platformami.

Główną zaletą PCL jest prostota jego implementacji oraz bardzo łatwa przenaszalność pomiędzy różnymi platformami bez konieczności wprowadzania jakichkolwiek modyfikacji kodu. Istnieją jednak pewne ograniczenia, które w określonych sytuacjach mogą utrudnić albo nawet uniemożliwić skorzystanie z tego rozwiązania. W projekcie PCL może znajdować się jedynie kod, który jest kompatybilny ze wszystkimi docelowymi platformami. Oznacza to, że w przypadku Aplikacji Uniwersalnych kod musi być kompatybilny zarówno z systemem Windows 8.1 jak i Windows Phone 8.1. Nie ma możliwości stworzenia kodu współpracującego tylko z jedną z tych platform, jak na przykład kod korzystający z odbiornika GPS, który nie byłby kompatybilny z systemem Windows 8.1. Nie można także skorzystać z następujących funkcjonalności oferowanych przez platformę .NET [7]:

- biblioteka Data Annotations,
- słowo kluczowe dynamic,
- biblioteka Managed Extensibility Network,

- przestrzeń nazw System.Numerics,
- framework Windows Communication Foundation.

W przypadku, gdy nie jest możliwe skorzystanie z PCL, należy skorzystać ze współdzielonego projektu opisanego w podrozdziale 3.4.3.

3.4.3. Projekt współdzielony

Projekt współdzielony jest podstawową formą współdzielenia kodu w Aplikacjach Uniwersalnych. Jest on automatycznie dodawany do solucji w momencie stworzenia projektu z szablonu Aplikacji Uniwersalnej. Zasada działania współdzielonego projektu jest prosta: wszystkie pliki znajdujące się w tym projekcie będą dostępne dla pozostałych projektów w solucji. Należy jednak zadbać o to, aby nie występowały żadne konflikty nazw plików. Jeżeli zarówno w projekcie współdzielonym, jak i w jednym z pozostałych projektów znajduje się już plik o identycznej nazwie, kompilacja się nie powiedzie. Sam współdzielony projekt nie generuje w trakcie kompilacji plików binarnych, służy on jedynie do przechowywania wspólnych plików. W przeciwieństwie do PCL możliwe jest także umieszczenie kodu współpracującego tylko z jedną z platform docelowych. W tym celu wykorzystuje się dyrektywy kompilatora. Przykład:

```
#if WINDOWS_APP
    //kod przeznaczony dla Windows 8.1
#endif

#if WINDOWS_PHONE_APP
    //kod przeznaczony dla Windows Phone
#endif
```

Plik jest kopiowany do poszczególnych projektów w niezmienionej formie, jednak kod zawarty w dyrektywach kompilatora odpowiadających niekompatybilnym platformom zostanie pominięty w trakcie kompilacji. Należy jednak pamiętać, że nadużywanie tego rozwiązania może znacznie pogorszyć czytelność kodu, dlatego jeżeli konieczne jest umieszczenie wewnątrz pliku wielu tego typu dyrektyw, lepszym rozwiązaniem byłoby stworzenie zupełnie oddzielnych implementacji tego pliku na poszczególne platformy. Z uwagi na wpływ, jaki mają dyrektywy kompilatora na czytelność kodu, zalecane jest korzystanie z bibliotek PCL kiedy tylko jest to możliwe. Dodatkowym problemem, na jaki można się natknąć podczas pracy ze współdzielonym projektem, jest refaktoryzacja. Standardowy mechanizm refaktoryzacji oferowany przez Visual Studio 2013 nie modyfikuje kodu zawartego wewnątrz nieaktywnych dyrektyw kompilatora. Oznacza to, że kod zawar-

ty w dyrektywach kompilatora odpowiadających Windows Phone nie będzie podlegał refaktoryzacji, ponieważ dyrektywa warunkowa `#if WINDOWS_PHONE_APP` nie będzie spełniona.

3.5. Plik manifestu aplikacji uniwersalnej (Appmanifest)

Plik AppManifest zawiera szczegółowe informacje o aplikacji oraz określa funkcjonalności, z których ta aplikacja korzysta. Jest on automatycznie tworzony wraz z nowym projektem. Plik ten jest zapisywany w formacie XML, jednak z uwagi na jego ważną rolę nie powinien być edytowany "ręcznie" [8]. Środowisko Visual Studio zawiera wbudowany edytor, który umożliwia edycję pliku AppManifest bez obaw o jego niepoprawne formatowanie. W celu uruchomienia edytora wystarczy dwukrotnie kliknąć plik Package.appxmanifest w widoku projektu. Wprawdzie format tego pliku jest podobny w projektach Windows Phone 8.1 i Windows 8.1, jednak pomiędzy nimi występują pewne różnice. Z tego powodu plik AppManifest nie jest współdzielony pomiędzy projektami aplikacji uniwersalnej i należy go edytować osobno we wszystkich projektach. W tym rozdziale zostaną omówione poszczególne sekcje tego pliku, co powinno znacznie ułatwić konfigurację aplikacji oraz zmniejszyć szansę jej odrzucenia w Widnows Store.

3.5.1. Sekcja Application

W sekcji *Application* zdefiniowane są najbardziej podstawowe informacje oraz funkcjonalności aplikacji:

- **Display name** - wyświetlana nazwa aplikacji. W przypadku, gdy aplikacja ma zostać umieszczona w Windows Store, ta nazwa musi być unikatowa.
- **Entry point** - klasa (wraz z przestrzenią nazw) zawierająca punkt wejścia aplikacji. Musi ona dziedziczyć po klasie Application.
- **Default language** - standardowy język, który zostanie wykorzystany jeżeli aplikacja nie posiada tłumaczenia w języku użytkownika.
- **Description** - ogólny opis aplikacji.
- **Supported rotations** - wspierane orientacje ekranu.
- **SD Cards** - zezwolenie (lub jego brak) na instalację aplikacji na kartach SD. Opcja dostępna tylko w Windows Phone.

- **Minimum width** - minimalna szerokość ekranu wymagana przez aplikację, określona w DIP. Urządzenia o mniejszej szerokości nie będą kompatybilne z aplikacją.
- **Toast capable** - ta wartość musi zostać ustawiona na *Yes* jeżeli aplikacja będzie wyświetlała powiadomienia Toast. W przeciwnym przypadku wszelkie próby wyświetlenia powiadomień nie powiodą się, chociaż nie zostanie zwrócony żaden wyjątek. Powiadomienia Toast są szczegółowo opisane w rozdziale 4.2.
- **Lock screen notifications** - zezwolenie na wyświetlanie powiadomień na ekranie blokady.
- **Tile update** - umożliwia stworzenie dynamicznych ikon (kafelków) w menu start. Funkcjonalność ta została szczegółowo opisana w rozdziale 4.3.

3.5.2. Sekcja Visual Assets

Sekcja ta zawiera zasoby graficzne aplikacji: ikony na kafelki w menu Start, ikony na ekranie blokady, ikony wykorzystywane w Windows Store oraz splash screen wyświetlany przy starcie aplikacji. Wszystkie zasoby graficzne są zdefiniowane w różnych rozdzielczościach tak, aby były kompatybilne z wieloma urządzeniami. Jeżeli aplikacja ma zostać zaakceptowana do Windows Store, wszystkie te zasoby muszą zostać zdefiniowane.

3.5.3. Sekcja Requirements

Sekcja ta jest dostępna jedynie w pliku AppManifest projektu przeznaczonego dla Windows Phone. Znajdują się tutaj wymagania, które muszą zostać spełnione przez urządzenie w celu instalacji aplikacji. Jeżeli wymagania te nie są spełnione, aplikacja nie będzie mogła zostać pobrana ani zainstalowana. Aplikacja może mieć następujące wymagania:

- **Gyroscope** - wymagany telefon z wbudowanym żyroskopem,
- **Magentometer** - wymagany telefon z wbudowanym kompasem,
- **NFC** - wymagane wsparcie dla Near Field Communications (NFC),
- **Front Camera** - wymagana kamera znajdująca się z przodu telefonu,
- **Rear Camera** - wymagana kamera znajdująca się z tyłu telefonu.

3.5.4. Sekcja Capabilities

W tej sekcji zaznaczone muszą zostać wszystkie funkcjonalności, z których korzysta nasza aplikacja. Jeżeli odpowiednie funkcjonalności nie zostaną zaznaczone, aplikacja zostanie odrzucona przy próbie dodania jej do Windows Store. Jest to jeden z mechanizmów bezpieczeństwa, dzięki któremu użytkownik wie, z jakich funkcjonalności nasza aplikacja korzysta. Jeżeli zaznaczone zostaną funkcjonalności, które nie są wymaganie przez aplikację (na przykład wymagany dostęp do odbiornika GPS dla edytora tekstu) aplikacja także może zostać odrzucona ze względów bezpieczeństwa lub pominięta przez użytkownika w obawie, że może zawierać szkodliwe oprogramowanie. Funkcjonalności związane z dostępem do plików lub katalogów dotyczą jedynie dostępu z poziomu kodu. Jeżeli dostęp do plików na dysku będzie się odbywał jedynie przy pomocy file pickera, nie ma potrzeby zmiany pliku AppManifest. Udostępniane są następujące funkcjonalności [9]:

- **Internet (Client)** - możliwość dostępu do Internetu w trybie klienta (połączenia wychodzące).
- **Internet (Client & Server)** - możliwość dostępu do Internetu zarówno w trybie klienta jak i serwera (połączenia przychodzące oraz wychodzące). Jeżeli opcja ta została zaznaczona, nie jest konieczne zaznaczenie opcji **Internet (Client)**. Połączenia przychodzące na krytycznych portach (między innymi 21, 25, 80, 443, 8080) będą zawsze blokowane.
- **Location** - dostęp do mechanizmów umożliwiających określenie współrzędnych geograficznych urządzenia przy pomocy odbiornika GPS lub informacji o sieci.
- **Enterprise authentication** - umożliwia dostęp do zdalnych zasobów wymagających uwierzytelnienia przy pomocy nazwy użytkownika oraz hasła. Dotyczy to jedynie dostępu z poziomu kodu. Funkcjonalność ta nie jest wymagana w przypadku dostępu do tych zasobów z poziomu file pickera.
- **Microphone** - możliwość nagrywania dźwięku.
- **Music library** - umożliwia dostęp do biblioteki muzycznej użytkownika z poziomu kodu.
- **Pictures Library** - umożliwia dostęp do katalogu z zasobami graficznymi użytkownika z poziomu kodu.

- **Video Library** - umożliwia dostęp do katalogu z plikami wideo użytkownika z poziomu kodu.
- **Private networks** - dostęp do sieci oznaczonych przez użytkownika jako sieci domowe lub sieci w miejscu pracy.
- **Proximity** - umożliwia komunikację ze znajdującymi się w pobliżu urządzeniami. Automatycznie wybierany jest protokół oferujący w danej sytuacji najlepszą wydajność (między innymi Bluetooth, WiFi lub Internet).
- **Removable Storage** - dostęp do pamięci przenośnych, między innymi dysków zewnętrznych, kart pamięci, pendrive'ów.
- **Shared user certificates** - możliwość zarządzania certyfikatami zainstalowanymi na urządzeniu.
- **Webcam** - dostęp do strumienia wideo z kamery. W celu nagrywania dźwięku konieczne jest jednak zaznaczenie także funkcjonalności **Microphone**.

3.5.5. Sekcja Declarations

W tej sekcji znajdują się ustawienia mechanizmów, przy pomocy których aplikacja komunikuje się z systemem operacyjnym lub z innymi aplikacjami. Istnieją dwa typy deklaracji: rozszerzenia (extensions) oraz kontrakty (contracts). Kontrakt jest w pewnym sensie umową pomiędzy aplikacjami definiującą wymagania, które muszą zostać spełnione w celu zapewnienia poprawnej komunikacji pomiędzy tymi aplikacjami. Dzięki temu, aby mieć pewność, że proces wymiany danych przebiegnie poprawnie, aplikacja nie musi posiadać żadnych danych o pozostałych aplikacjach, wystarczy jedynie informacja, że spełnione zostały odpowiednie kontrakty. Rozszerzenie jest z kolei umową pomiędzy aplikacją a systemem operacyjnym. Pozwalają one na rozbudowę oraz modyfikowanie niektórych funkcjonalności systemu operacyjnego. Możliwe jest zdefiniowanie następujących kontraktów oraz rozszerzeń [10]:

- **Account picture provider** - rozszerzenie umożliwiające zmianę zdjęcia profilu obecnie zalogowanego użytkownika. Standardowo, użytkownik może zmienić zdjęcie samodzielnie, lub skorzystać z zewnętrznej aplikacji wylistowanej na stronie z ustawieniami profilu. Po zdefiniowaniu tego rozszerzenia aplikacja zostanie dodana do tej listy.

- **Autoplay Device** - rozszerzenie pozwalające na zarejestrowanie aplikacji do obsługi zdarzeń AutoPlay[3] dla urządzeń zidentyfikowanych jako Windows Portable Devices [11]
- **Autoplay Content** - rozszerzenie pozwalające na zarejestrowanie aplikacji do obsługi zdarzeń AutoPlay dla nośników danych (na przykład pendrive, karta pamięci, płyta DVD).
- **Background tasks** - rozszerzenie umożliwiające aplikacji na wykonywanie kodu w tle, nawet jeżeli aplikacja została wstrzymana (stan suspended). W tle wykonywany powinien być jedynie kod zużywający niewiele zasobów systemowych. W celu dodania nowego zadania wykonywanego w tle konieczne jest stworzenie klasy implementującej interfejs `IBackgroundTask` i umieszczenie wykonywanego kodu w metodzie `Run`.
- **Cached file updater** - kontrakt pozwalający na zadeklarowanie naszej aplikacji jako aplikacji zarządzającej danym plikiem. Możliwe jest zmuszenie systemu do wykonania konkretnych działań w określonych sytuacjach, np. wymuszenie otwarcia naszej aplikacji przed odczytem lub zapisem do pliku, wykorzystywanie ostatniej wersji pliku w sytuacji gdy nie ma dostępu do Internetu lub całkowity zakaz dostępu do pliku [12].
- **Camera Settings** - rozszerzenie umożliwiające zmianę panelu konfiguracyjnego kamery. Jeżeli użytkownik otworzy panel konfiguracyjny kamery gdy aplikacja jest uruchomiona, po wciśnięciu przycisku *More options* pojawi się nasz zmodyfikowany panel. Jeżeli rozszerzenie nie zostało zdefiniowane wyświetlony zostanie standardowy panel [13].
- **Certificates** - rozszerzenie umożliwiające aplikacji instalację certyfikatów SSL.
- **Contact Picker** - kontrakt umożliwiający udostępnienie listy kontaktów zawartej wewnątrz naszej aplikacji (jeżeli taka lista istnieje) innym aplikacjom.
- **File Open Picker** - kontrakt pozwalający innym aplikacjom na otwarcie plików przechowywanych przez naszą aplikację przy pomocy *File Open Pickera*. Na przy-

[3] Zdarzenie Autoplay jest wywoływane, gdy system Windows wykryje podłączenie nośnika danych. Na podstawie jego zawartości możliwe jest wykonanie odpowiednich działań, jak na przykład uruchomienie programu potrafiącego obsłużyć pliki znajdujące się na tym nośniku.

kład, aplikacja nagrywająca dźwięki może umożliwić otwarcie tych dźwięków odtwarzaczowi muzyki [14].

- **File Save Picker** - kontrakt działający analogicznie do kontraktu *File Open Picker*, z tą różnicą, że pozwala na dodawanie oraz modyfikację plików przechowywanych przez naszą aplikację [14].
- **File Type Associations** - rozszerzenie umożliwiające zarejestrowanie naszej aplikacji jako aplikacji potrafiącej obsłużyć dane typy plików. Jeżeli użytkownik będzie próbował otworzyć jeden ze zdefiniowanych w tym rozszerzeniu typów plików, nasza aplikacja pojawi się na liście aplikacji potrafiących obsłużyć ten plik.
- **Print task settings** - rozszerzenie umożliwiające zmianę panelu konfiguracyjnego drukowania. Jeżeli użytkownik otworzy panel konfiguracyjny drukowania gdy aplikacja jest uruchomiona, po wciśnięciu przycisku *More settings* pojawi się nasz zmodyfikowany panel. Jeżeli rozszerzenie nie zostało zdefiniowane wyświetlony zostanie standardowy panel [15].
- **Protocol** - kontrakt umożliwiający zarejestrowanie protokołu URI obsługiwanego przez naszą aplikację, np. *mailto* w przypadku klientów email
- **Search** - kontrakt umożliwiający indeksowanie zawartości aplikacji przez globalną wyszukiwarkę plików systemu Windows.
- **Share Target** - kontrakt umożliwiający współdzielenie zawartości naszej aplikacji z innymi aplikacjami, które także wspierają ten kontrakt.
- **Update task** - rozszerzenie umożliwiające wywołanie zadań (background tasks) zaraz po uaktualnieniu aplikacji. Dzięki temu możliwe jest wyświetlenie powiadomień oraz przeprowadzenie konfiguracji bez konieczności uruchamiania aplikacji przez użytkownika. Rozszerzenie to może zostać zdefiniowane jedynie w pliku *AppManifest* dla Widnows Phone.

4. Aplikacje Uniwersalne - wybrane zagadnienia

W poprzednich rozdziałach opisane zostały ogólne idee oraz bardzo podstawowe funkcjonalności Aplikacji Uniwersalnych, które pozwalają rozpocząć pracę nad własną aplikacją. W tym rozdziale zaprezentowane zostaną wybrane zagadnienia, które pozwolą pełniej wykorzystać możliwości oferowane nam przez architekturę WinRT[4].

4.1. Dostęp do plików

Zgodnie z informacjami zawartymi w rozdziale 3.1 aplikacje korzystające z architektury WinRT działają w odizolowanym środowisku i nie mają bezpośredniego dostępu do dysku. Dla każdej aplikacji wydzielone są osobne katalogi, w których aplikacja może przechowywać swoje pliki lokalne, pliki tymczasowe oraz pliki synchronizowane pomiędzy urządzeniami. Możliwy jest także dostęp do poszczególnych bibliotek użytkownika (muzyka, obrazy i filmy) o ile w pliku AppManifest zostały sprecyzowane określone uprawnienia oraz dostęp do plików dodanych do projektu (np. plików graficznych).

4.1.1. Dostęp do plików lokalnych

Każda aplikacja ma przydzielony katalog *Local* służący do przechowywania plików, które związane są jedynie z danym urządzeniem lub nie mogą zostać umieszczone w chmurze. Pozostałe aplikacje nie mają dostępu do tych plików, zatem katalog ten nie może służyć do udostępniania plików innym aplikacjom. Na katalog *Local* nie są narzucane żadne ograniczenia odnośnie rozmiaru lub liczby przechowywanych w nim plików. Dostęp do tego katalogu można uzyskać przy pomocy adresu "*ms-appdata:///local/*", np.:

```
<Image Source="ms-appdata:///local/image.png"/>
```

4.1.2. Synchronizacja plików

W katalogu *Roaming* umieszczane są pliki współdzielone pomiędzy wszystkimi urządzeniami, na których zainstalowana została nasza aplikacja. Cały proces wykonywany jest przez system operacyjny, dzięki czemu programista nie musi wykonywać żadnej dodatkowej pracy w celu zapewnienia synchronizacji plików. Dzięki takiemu rozwiązaniu aplikacja może pozwolić użytkownikowi na kontynuowanie wcześniej rozpoczętej pracy na po-

[4] Dodatkowe zagadnienia pokryte zostaną w pracach dyplomowych pisanych przez Pawła Stopińskiego oraz Rafała Wołowskiego

zostałych urządzeniach, a wszelkie zmiany w konfiguracji mogą zostać łatwo zastosowane na wszystkich urządzeniach jednocześnie [16].

W momencie umieszczenia lub modyfikacji plików w katalogu *Roaming*, system operacyjny przesyła dokonane zmiany do chmury. Te zmiany są następnie przekazywane do pozostałych urządzeń, na których nasza aplikacja została zainstalowana. System operacyjny decyduje o tym, kiedy zawartość tego katalogu jest synchronizowana. To oznacza, że zmiany niekoniecznie zostaną zastosowane natychmiast. W przypadku obciążonego łącza lub braku dostępu do Internetu czas zastosowania zmian będzie odpowiednio dłuższy. Jeżeli w momencie aktualizacji zawartości katalogu *Roaming* uruchomione są osobne instancje naszej aplikacji na pozostałych urządzeniach, wywołane zostanie zdarzenie `DataChanged`, które musi zostać obsłużone w celu zastosowania otrzymanych zmian [17].

Na dane przechowywane w katalogu *Roaming* nakładane są jednak pewne ograniczenia. Przede wszystkim rozmiar tego katalogu nie może przekraczać 100kB. Oznacza to, że katalog ten może służyć jedynie do przechowywania niewielkich plików, takich jak pliki konfiguracyjne. W celu przechowywania większych plików należy skorzystać z katalogu *Local*. Dodatkowo, jeżeli użytkownik nie będzie korzystał z aplikacji przez dany okres czasu (obecnie okres ten wynosi 30 dni), zawartość katalogu *Roaming* zostanie usunięta. Dlatego też w katalogu *Roaming* nie należy umieszczać danych, których utrata nie jest dopuszczalna.

Dostęp do katalogu *Roaming* można uzyskać przy pomocy protokołu "*ms-appdata:///roaming/*" w sposób analogiczny, jak w przypadku plików lokalnych opisanych w rozdziale 4.1.1.

4.1.3. Pliki tymczasowe

Kolejnym katalogiem wydzielonym dla każdej aplikacji jest katalog *Temp*. W tym katalogu można umieszczać pliki tymczasowe. Pliki te przechowywane są tylko lokalnie, zatem nie są one synchronizowane z chmurą. System operacyjny okresowo usuwa pliki tymczasowe, przez co nie ma żadnej gwarancji na to, że pliki te zostaną zachowane pomiędzy sesjami naszej aplikacji. Użytkownik może także samodzielnie usunąć zawartość katalogu *Temp* przy pomocy narzędzia oczyszczania dysku dołączonego do systemu Windows. Dostęp do katalogu *Temp* można uzyskać przy pomocy protokołu "*ms-appdata:///temp/*", podobnie jak przypadku zwykłych plików lokalnych.

4.1.4. Katalog aplikacji

Każda aplikacja ma dodatkowo nieograniczony dostęp do wszystkich plików zlokalizowanych w katalogu, w którym została zainstalowana. W tym katalogu znajdują się także pliki dodane do zasobów projektu. Do tych plików można odwołać się przy pomocy protokołu "*ms-appx://*", zatem jeżeli w projekcie naszej aplikacji w katalogu *Assets* utworzony został przykładowy plik *image.png*, możemy się do niego odwołać przy pomocy adresu URI "*ms-appx:///Assets/image.png*"

4.1.5. Dostęp do plików z poziomu kodu

Oczywiście operacje na plikach możliwe są także z poziomu kodu C#. Poniżej znajduje się przykład pokazujący w jaki sposób można utworzyć nowy plik w katalogu *Local* oraz jak zapisać lub odczytać zawartość tego pliku:

```
using Windows.Storage;

[...]

//wybieramy katalog - Local
var directory = ApplicationData.Current.LocalFolder;

//tworzymy plik, jednak jeżeli już istnieje zostanie on otwarty
var file = await directory.CreateFileAsync("test.txt",
CreationCollisionOption.OpenIfExists);

//zapisujemy tekst do pliku
await FileIO.WriteTextAsync(file, "przykładowa zawartość pliku");

//odczytujemy zawartość pliku
file = await directory.GetFileAsync("test.txt");
var fileContent = await FileIO.ReadTextAsync(file);
```

Możliwy jest także dostęp do katalogów *Roaming* oraz *Temp*, jak i poszczególnych bibliotek, jeżeli w pliku AppManifest zostały określone odpowiednie uprawnienia. Przykładowe wartości zmiennej `directory` dla powyższego przykładu mogą być następujące:

- `ApplicationData.Current.RoamingFolder` - katalog *Roaming*,
- `ApplicationData.Current.TemporaryFolder` - katalog *Temp*,
- `KnownFolders.VideosLibrary` - biblioteka wideo,
- `KnownFolders.PicturesLibrary` - biblioteka obrazów.

Należy zwrócić uwagę na fakt, że na platformie WinRT dostęp do plików następuje w sposób asynchroniczny; tak więc do wykonywanych operacji na plikach często konieczne jest dodanie słowa kluczowego `await`.

4.2. Powiadomienia Toast

Powiadomienia Toast są wiadomościami wyświetlanymi przez aplikację, podobnymi do okienek pop-up. W przypadku Windows Phone wiadomości te wyświetlane są na górze ekranu, natomiast w przypadku systemu Windows 8.1 wyświetlane są one w prawym górnym rogu ekranu. Główną rolą powiadomień Toast jest komunikacja z użytkownikiem, gdy nasza aplikacja działa w tle. Mogą one na przykład powiadamiać użytkownika o uaktualnieniu aplikacji lub otrzymaniu nowej wiadomości e-mail. Należy jednak pamiętać, że użytkownik ma możliwość wyłączenia powiadomień Toast dla naszej aplikacji, zatem nie powinny one służyć do przekazywania ważnych informacji.

4.2.1. Wyświetlanie powiadomień Toast

W celu wyświetlenia powiadomienia Toast należy wczytać jego szablon. Szablony są zwykłymi dokumentami XML definiującymi wartości poszczególnych elementów powiadomienia, takich jak tekst czy ikona. Dostępnych jest osiem szablonów, które są oznaczone poprzez poszczególne wartości enumeracji `ToastTemplateType` dostępnej w przestrzeni nazw `Windows.UI.Notifications` [18]. Możliwe jest wczytanie jednego z tych szablonów, a następnie dynamiczne ustawienie odpowiednich jego wartości lub stworzenie własnego szablonu bazującego na jednym ze standardowych szablonów z interesującymi nas parametrami już ustawionymi na odpowiednie wartości. Oto przykład prostego powiadomienia zawierającego jedynie tekst:

```
//wybieramy typ powiadomienia
const ToastTemplateType ToastTemplate = ToastTemplateType.ToastText01;

//wczytujemy plik XML odpowiadający temu typowi powiadomienia
var toastXml = ToastNotificationManager.GetTemplateContent(ToastTemplate);

//ustalamy tekst powiadomienia
XmlNodeList toastTextElements = toastXml.GetElementsByTagName("text");
toastTextElements[0].AppendChild(toastXml.CreateTextNode("Example text"));

//tworzymy nowe powiadomienie Toast a następnie je wyświetlamy
var toast = new ToastNotification(toastXml);
ToastNotificationManager.CreateToastNotifier().Show(toast);
```

Rysunek 6: Powiadomienie Toast na Windows 8.1 (z lewej) oraz na Windows Phone 8.1 (z prawej)

Jeżeli powiadomienie nie jest wyświetlane, należy upewnić się, że w pliku *AppManifest* opcja *Toast Capable* została ustawiona na *Yes*. W powyższym przykładzie wczytany został jeden ze standardowych szablonów XML, a następnie zmieniony został jego tekst. Możliwe jest jednak przedefiniowanie tego standardowego szablonu w osobnym pliku XML, a następnie wczytanie tego pliku. Dla powiadomienia identycznego jak powiadomienie zaprezentowanego na *Rysunku 6*, plik XML wyglądałby następująco:

```
<toast>
  <visual>
    <binding template="ToastText01">
      <text id="1">Example text</text>
    </binding>
  </visual>
</toast>
```

Taki plik można następnie wczytać i na jego podstawie utworzyć powiadomienie Toast bez żadnych dodatkowych modyfikacji:

```
var file = await StorageFile.GetFileFromApplicationUriAsync(
                  new Uri("ms-appx:///ExampleToast.xml"));
var toastXml = await XmlDocument.LoadFromFileAsync(file);
var toast = new ToastNotification(toastXml);
ToastNotificationManager.CreateToastNotifier().Show(toast);
```

4.3. Live tiles

Live tiles, czyli dynamiczne kafelki pozwalają na wyświetlanie interaktywnych ikon w menu start. Dzięki temu możliwe jest wyświetlanie różnego rodzaju zmiennych informacji na podstawie danych pobieranych z Internetu lub generowanych na bieżąco przez naszą aplikację. System Windows standardowo oferuje kilka przykładowych dynamicznych kafelków prezentujących między innymi aktualną pogodę, wiadomości oraz kursy walut. Programista może dodać taką ikonę przy pomocy pliku AppManifest lub wysłać uaktualnienie z poziomu kodu. Niezależnie jednak od sposobu tworzenia kafelków, podobnie jak w przypadku powiadomień Toast, konieczne jest stworzenie pliku XML definiującego wygląd naszego kafelka.

4.3.1. Tworzenie dynamicznych kafelków

Platforma WinRT oferuje mnóstwo stylów dla dynamicznych kafelków. Listę wszystkich szablonów wraz z odpowiadającymi im plikami XML można znaleźć w dokumentacji na stronie Microsoftu [19]. Należy jednak pamiętać, że niektóre szablony nie są kompatybilne ze wszystkimi systemami operacyjnymi. Istnieją trzy wersje szablonów oznaczone numerami od 1 do 3. Ich kompatybilność z poszczególnymi systemami operacyjnymi pokazuje *Tabela 2*:

	Windows 8	**Windows 8.1**	**Windows Phone 8.1**
Wersja 1	Kompatybilne	Kompatybilne	Zależnie od szablonu
Wersja 2	Niekompatybilne	Kompatybilne	Zależnie od szablonu
Wersja 3	Niekompatybilne	Niekompatybilne	Kompatybilne

Tabela 2: Kompatybilność szablonów kafelków z poszczególnymi systemami operacyjnymi

Gdy metoda `GetTemplateContent` zostanie wykonana, w systemie Windows 8 zwrócony zostanie szablon w wersji 1, a w systemie Windows 8.1 lub Windows Phone 8.1 zwrócony zostanie szablon w wersji 2 lub 3 (zależnie od szablonu). Jeżeli w pliku *AppManifest* została ustawiona kompatybilność z systemem Windows 8, metoda ta zwróci szablon w wersji 1 niezależnie od systemu na jakim działa nasza aplikacja [20].

Każdy szablon może zostać zastosowany jedynie dla odpowiadającego mu rozmiaru kafelka, co oznacza, że jeżeli nasza aplikacja ma wspierać dynamiczne kafelki w wielu rozmiarach, konieczne jest zastosowanie osobnego szablonu dla każdego rozmiaru. Nie jest jednak możliwe zastosowanie dwóch szablonów przeznaczonych dla tego samego rozmiaru.

4.3.2. Wysyłanie powiadomień o uaktualnieniu kafelka

W celu uaktualnienia kafelka konieczne jest wysłanie do systemu operacyjnego powiadomienia zawierającego nowy szablon kafelka, podobnie jak to było w przypadku wiadomości Toast. To powiadomienie usuwa wszystkie dotychczas zastosowane szablony, dlatego wszystkie nowe szablony muszą zostać zawarte w pojedynczym powiadomieniu. Zdefiniujmy dla przykładu pojedynczy dynamiczny kafelek wspierający rozmiar "szeroki" oraz "duży". Zacznijmy od zdefiniowania szerokiego kafelka:

```
var tile1Content TileUpdateManager.GetTemplateContent(
                 TileTemplateType.TileWide310x150Text01);
var tile1Lines = tile1Content.SelectNodes("tile/visual/binding/text");
tile1Lines[0].InnerText = "Header";
```

```
tile1Lines[1].InnerText = "Line 1";
tile1Lines[2].InnerText = "Line 2";
```

Powyższy kafelek jest prostym kafelkiem tekstowym, definiowanym bardzo podobnie do powiadomienia Toast. Zdefiniujmy drugi kafelek:

```
var tile2Content = TileUpdateManager.GetTemplateContent(
                   TileTemplateType.TileSquare310x310ImageAndTextOverlay01);
var tile2Text = tile2Content.SelectSingleNode("tile/visual/binding/text");
tile2Text.InnerText = "Large tile text";
var tile2Image = (XmlAttribute)tile2Content.SelectSingleNode(
                 "tile/visual/binding/image/@src");
tile2Image.Value = "ms-appx:///Assets/image310x310.png";
```

Powyższy kafelek zawiera już nie tylko tekst, ale także i obraz, który zostanie wyświetlony w tle. Oznacza to, że musieliśmy nie tylko zmienić zawartość węzła `text`, ale również zmienić wartość atrybutu `src` węzła `image`. Przed wysłaniem uaktualnienia należy jeszcze złączyć definicje obu kafelków w jeden dokument XML. W przeciwnym przypadku konieczne byłoby wysłanie dwóch powiadomień, co skutkowałoby nadpisaniem pierwszego z tych powiadomień. Zatem dopiszmy zawartość węzła `visual` drugiego kafelka do zawartości węzła `visual` pierwszego kafelka:

```
var node = tile1Content.ImportNode(
           tile2Content.SelectNodes("tile/visual/binding")[0],
           true);
tile1Content.SelectNodes("tile/visual")[0].AppendChild(node);
```

W efekcie otrzymamy pojedynczy dokument XML definiujący wygląd obu naszych kafelków. Zmienna `tile1Content` będzie reprezentować następujący dokument XML:

```
<tile>
  <visual version="2">
    <binding template="TileWide310x150Text01" fallback="TileWideText01">
      <text id="1">Header</text>
      <text id="2">Line 1</text>
      <text id="3">Line 2</text>
    </binding>

    <binding template="TileSquare310x310ImageAndTextOverlay01">
      <image id="1" src="ms-appx:///Assets/image310x310.png" alt="alt text"/>
      <text id="1">Large tile text</text>
    </binding>
  </visual>
</tile>
```

Na koniec musimy jeszcze wysłać powyższy dokument XML w powiadomieniu o uaktualnieniu kafelka. Metoda jest bardzo podobna do wysyłania powiadomienia Toast,

jednak dobrą praktyką jest zdefiniowanie daty ważności naszego kafelka w celu uniknięcia wyświetlania przestarzałych informacji:

```
var notification = new TileNotification(tile1Content);
notification.ExpirationTime = new DateTimeOffset(DateTime.Now.AddDays(1));
var updater = TileUpdateManager.CreateTileUpdaterForApplication();
updater.Update(notification);
```

4.3.3. Uaktualnienie przy pomocy pliku AppManifest

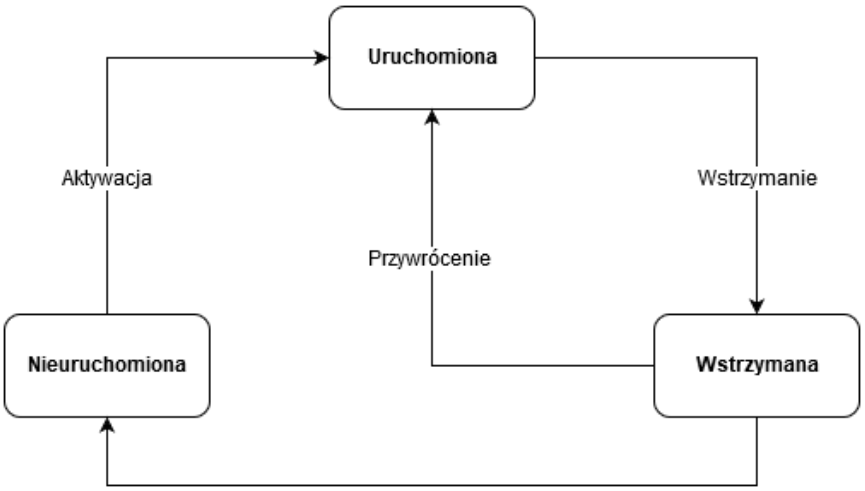
W rozdziale 4.3.2 omówiony został sposób uaktualniania kafelków z poziomu kodu. Metodę tą można zastosować także w przypadku, gdy konieczne jest uaktualnianie kafelka na podstawie danych z Internetu, jednak w takiej sytuacji plik *AppManifest* oferuje nam znacznie prostszą metodę, która nie wymaga pisania kodu uaktualniającego nasz kafelek. W zakładce *Application* pliku *AppManifest* dostępne mamy dwa parametry: *Recurrence* oraz *URI Template*. Parametr *URI Template* jest adresem, pod którym znajduje się plik XML definiujący nasz kafelek, na przykład "*http://example.com/template.xml*". Zawartość tego pliku może być identyczna jak w przypadku zaprezentowanym w rozdziale 4.3.2. Przy pomocy parametru *Recurrence* musimy także sprecyzować częstotliwość, z jaką adres ten będzie odświeżany w celu ponownej aktualizacji kafelka. Mamy pięć możliwości: co pół godziny, co godzinę, co sześć godzin, co dwanaście godzin lub raz dziennie.

Główną zaletą takiego rozwiązania jest jego prostota. Programista nie musi pisać kodu ani tworzyć zadań działających w tle w celu okresowej aktualizacji kafelka. To skutkuje znaczną oszczędnością czasu. Rozwiązanie to nie jest jednak pozbawione wad. Programista ma znacznie mniejszą kontrolę nad procesem uaktualniania kafelka. Na przykład ograniczone są możliwości ustawienia częstotliwości aktualizacji. Dodatkowo, wymagany jest serwer na którym przechowywany będzie aktualny szablon XML kafelka.

4.4. Cykl życia aplikacji

W trakcie normalnego korzystania z systemu operacyjnego należy spodziewać się, że użytkownik będzie często przełączał się pomiędzy wieloma różnymi aplikacjami. W momencie, gdy użytkownik uruchomi nową aplikację, poprzednio uruchomiona aplikacja zostanie wstrzymana. Wstrzymane aplikacje nie wykonują żadnego kodu, poza ewentualnymi drobnymi zadaniami działającymi w tle. Dzięki temu aplikacja w tym stanie nie spowalnia systemu ani nie zwiększa zużycia baterii w przypadku urządzeń mobilnych lub laptopów. Wstrzymana aplikacja może następnie zostać przywrócona przez użytkownika

wstrzymując tym samym obecnie uruchomioną aplikację lub całkowicie zakończona przez system operacyjny. Cykl życia aplikacji WinRT prezentuje graf z *Rysunku 7*.



Rysunek 7: Cykl życia aplikacji uniwersalnej

Wstrzymana aplikacja nie jest od razu usuwana z pamięci. Dzięki temu można zapewnić użytkownikowi szybkie przełączanie pomiędzy aplikacjami bez pisania dodatkowego kodu. System operacyjny może jednak w dowolnym momencie zakończyć wstrzymaną aplikację jeżeli konieczne jest zwolnienie wykorzystywanych przez nią zasobów. Jeżeli użytkownik zamknie aplikację korzystając ze skrótu Alt+F4 zostanie ona wstrzymana, a następnie po kilku sekundach całkowicie zakończona. Dobrze zaprojektowana aplikacja zapisuje swój stan w chwili wstrzymania, dzięki czemu nawet w sytuacji zakończenia jej przez system operacyjny sprawia wrażenie, że nigdy nie przestała działać. Jeżeli aplikacja została wstrzymana lub zakończona przez system operacyjny, przy następnym uruchomieniu zostanie przywrócona do ostatniego zapamiętanego stanu. Jeżeli została całkowicie zakończona przez użytkownika lub zawiesiła się z powodu nieprzewidzianego błędu, przy następnym starcie zostanie uruchomiona nowa sesja a wszystkie niezapisane dane sesji zostaną utracone [24].

Nasza aplikacja musi zarządzać dwoma typami danych: danymi aplikacji oraz danymi sesji. Dane sesji są tracone po całkowitym wyłączeniu aplikacji przez użytkownika (zakończeniu sesji). Dane aplikacji są natomiast zapamiętywane pomiędzy sesjami i dostępne przy każdym uruchomieniu aplikacji. Przed rozpoczęciem pracy nad aplikacją należy jeszcze zwrócić uwagę na fakt, że aplikacje uruchomione za pośrednictwem środowiska Visual Studio i działające pod kontrolą debuggera nigdy nie zostaną wstrzymane. Jeżeli chcemy przetestować kod wykonujący się w trakcie wstrzymywania lub zatrzymywania aplikacji należy skorzystać z dodatkowego panelu dostępnego w środowisku Visual Studio w trak-

cie debuggowania, który pozwala na samodzielne wstrzymanie lub zakończenie naszej aplikacji w dowolnym momencie.

4.4.1. Suspension Manager

Gdy z poziomu menu kontekstowego dodamy do naszego projektu nową stronę bazującą na szablonie `BasicPage`, utworzony zostanie dodatkowo nowy katalog o nazwie *Common*. Katalog ten zawiera definicje czterech nowych klas: `NavigationHelper`, `ObservableDictionary`, `RelayCommand` oraz `SuspensionManager`. Jako że ten rozdział dotyczy zarządzania stanem sesji, interesują nas jedynie klasy `SuspensionManager` oraz `NavigationHelper`. Rolą klasy `SuspenionManager` jest zapamiętywanie stanu sesji w sytuacji, gdy aplikacja została wstrzymana lub zakończona przez system operacyjny [25]. Podstawową informacją do zapamiętania jest strona, na której znajdował się użytkownik w momencie wstrzymania aplikacji. Strony umieszczane są wewnątrz ramek (kontrolka `Frame`), dlatego aby umożliwić Suspension Managerowi zapamiętanie wyświetlanej w danym momencie strony konieczne jest zarejestrowanie ramki dla której zapamiętywany ma być stan sesji. Standardowo wykorzystywana jest tylko jedna ramka którą rejestrujemy w pliku *App.xaml.cs*. W metodzie `OnLaunched` odszukujemy linijkę, w której ta ramka jest tworzona i poniżej dopisujemy kod rejestrujący ramkę:

```
rootFrame = new Frame(); //tworzenie ramki
SuspensionManager.RegisterFrame(rootFrame, "ExampleSessionStateKey");
```

Należy jeszcze zadbać o to, aby Suspension Manager zapisywał oraz przywracał sesję kiedy jest to konieczne. Stan sesji zapisywany powinien być w chwili wstrzymania naszej aplikacji. W pliku *App.xaml.cs* standardowo znajduje się metoda `OnSuspending` obsługująca zdarzenie `Suspending`. Jako że stan sesji zapisywany jest w pliku, wykonywane będą operacje asynchroniczne, zatem konieczne jest dodanie do tej metody słowa kluczowego `async`. Następnie można skorzystać z Suspension Managera w celu zapisania stanu sesji. W najprostszym przypadku metoda `OnSuspending` powinna wyglądać następująco:

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    await SuspensionManager.SaveAsync();
    deferral.Complete();
}
```

Konieczne jest zapewnienie, aby Suspension Manager przywrócił sesję, gdy aplikacja jest uruchamiana po wcześniejszym wyłączeniu jej przez system operacyjny. Przywracanie stanu sesji po wstrzymaniu aplikacji (do stanu suspended) nie jest konieczne, ponieważ wstrzymanie nie usuwa aplikacji z pamięci. Ponownie należy zmodyfikować metodę `OnLaunched`. Podobnie jak w przypadku zapisania stanu sesji wykonywane będą operacje asynchroniczne, zatem do metody `OnLaunched` trzeba dopisać słowo kluczowe `async`. Jako że sesję aplikacji przywracamy tylko po przywróceniu zamkniętej przez system aplikacji, musimy sprawdzić w jakim stanie znajdowała się aplikacja przed uruchomieniem. Tę informację dostarcza nam jedyny argument metody `OnLaunched` typu `LaunchActivatedEventArgs`. Standardowo metoda ta zawiera miejsce przygotowane do przywrócenia stanu aplikacji, więc powinna wyglądać mniej więcej tak:

```
protected override async void OnLaunched(LaunchActivatedEventArgs e)
{
    [...]
    if (rootFrame == null)
    {
        [...]
        if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            await SuspensionManager.RestoreAsync();
        }
        [...]
    }
   [...]
}
```

To wystarczy, aby po ponownej aktywacji naszej aplikacji powróciła ona do ostatnio przeglądanej strony. Dane związane z tą stroną (zawartość lub stan kontrolek) nie zostaną jednak przywrócone. Przywracanie danych strony zostało opisane w rozdziale 4.4.2

4.4.2. Navigation Helper

`NavigationHelper` jest klasą umożliwiającą zapisywanie danych strony w momencie wstrzymania naszej aplikacji, lub w chwili, gdy użytkownik przejdzie do innej strony. W rzeczywistości `NavigationHelper` korzysta z Suspension Managera w celu zapisywania stanu sesji. Dane zapisywane są w słowniku `SuspensionManager.SessionState` pod kluczami Page-X, gdzie X oznacza numer strony poczynając od 0. Navigation Helper w przeciwieństwie do Suspension Managera nie korzysta ze statycznych pól do przechowywania stanu sesji, zatem dla każdej strony konieczne jest stworzenie osobnego obiektu klasy `NavigationHelper`. Strona stworzona przy pomocy

szablonu `BasicPage` automatycznie implementuje Navigation Helpera w celu zapamiętywania danych strony. W pozostałych przypadkach należy zaimplementować go ręcznie. Jeżeli stworzymy naszą stronę korzystając z szablonu `BasicPage`, odnajdziemy w naszym kodzie dwie metody: `navigationHelper_SaveState` oraz `navigationHelper_LoadState`. Metody te obsługują odpowiednio zdarzenia `LoadState` oraz `SaveSate` Navigation Helpera i udostępniają parametr, przy pomocy którego możliwe jest odczytanie oraz zapisanie danych strony. Przykładowe zastosowanie:

```
private void navigationHelper_LoadState(object sender, LoadStateEventArgs e)
{
    if (e.PageState != null)
        TextBox.Text = e.PageState["text"] as string ?? string.Empty;
}

private void navigationHelper_SaveState(object sender, SaveStateEventArgs e)
{
    e.PageState["text"] = TextBox.Text;
}
```

Dane zapisywane w ten sposób są oczywiście tracone po zakończeniu sesji. Metoda `SaveState` jest wykonywana przed nawigacją na inną stronę oraz po wstrzymaniu aplikacji.

4.4.3. Dane przechowywane pomiędzy sesjami

Możliwe oczywiście jest także przechowywanie danych tak, aby nie były one tracone w momencie całkowitego zakończenia sesji naszej aplikacji. W tym przypadku nie jest konieczne korzystanie z Navigation Helpera ani z Suspenion Managera. Dostępne są dwa słowniki: `ApplicationData.Current.LocalSettings` służacy do przechowywania danych lokalnych oraz `ApplicationData.Current.RoamingSettings` służący do przechowywania danych synchronizowanych pomiędzy urządzeniami. Rozwiązanie jest zatem analogiczne jak w przypadku dostępu do plików:

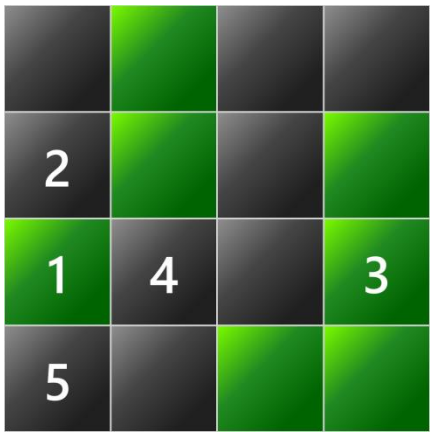
```
//zapisanie wartości
ApplicationData.Current.LocalSettings.Values["test"] = "Przykładowy tekst";

//odczytanie wcześniej zapisanej wartości
var tekst = ApplicationData.Current.LocalSettings.Values["test"] as string;
```

5. Przykładowy projekt

Aby podsumować i powtórzyć zawarte w tej pracy informacje, w niniejszym rozdziale krok po kroku omówiony zostanie proces tworzenia przykładowej Aplikacji Uniwersalnej. Aplikacja ta będzie wykorzystywać mechanizmy omówione w poprzednich rozdziałach, co powinno ułatwić ich zrozumienie oraz zaprezentować praktyczny przykład ich zastosowania. Do pracy dołączona zostanie płyta CD zawierająca pracę w wersji elektronicznej oraz kod źródłowy tworzonego w tym rozdziale projektu.

5.1. Założenia projektu

Celem projektu jest stworzenie prostej gry będącej modyfikacją gry Lights Out. Gra odbywa się na siatce o rozmiarach 4x4. Każde pole tej siatki może przyjąć dwa stany: włączony lub wyłączony. Gdy gracz zmieni stan jednej komórki, stan komórek sąsiadujących z tą komórką w pionie oraz w poziomie także ulegnie zmianie. Dodatkowo, gra implementuje periodyczne warunki brzegowe. Oznacza to, że w przypadku gdy modyfikowane pola znajdują się na krawędzi siatki, zmiany zostaną zastosowane także po drugiej stronie tej siatki. Przykład zaprezentowany jest na *Rysunku 8*.



Rysunek 8: Przykładowa plansza do gry

Pola oznaczone kolorem zielonym są włączone, natomiast pola oznaczone kolorem szarym są wyłączone. Jeżeli gracz kliknie na pole z numerem 1, to pola oznaczone numerami 1 oraz 3 zmienią stan z włączonego na wyłączony, a pola oznaczone numerami 2, 4 oraz 5 zmienią stan z wyłączonego na włączony. Celem gry jest wyłączenie wszystkich pól.

Gra powinna także umożliwiać zapisywanie oraz podgląd najlepszych wyników osiągniętych przez gracza. Możemy zatem podzielić naszą grę na trzy ekrany: ekran głównego menu umożliwiający ustawienie imienia gracza, ekran z listą najlepszych wyników, oraz ekran z planszą gry.

5.2. Implementacja logiki gry

W celu rozdzielenia warstwy logiki biznesowej i warstwy prezentacji w naszej aplikacji, implementacja logiki gry powinna znajdować się w osobnym projekcie, najlepiej w bibliotece PCL. Z uwagi na fakt, że tworząc logikę biznesową nie korzystamy z funkcjonalności typowych dla platformy WinRT, proces tworzenia logiki nie zostanie opisany w tej pracy. Przykładową implementację logiki gry można jednak zobaczyć w pliku *LightsOutGame.cs* znajdującym się w kodzie źródłowym dołączonym do tej pracy.

5.2.1. Zapisywanie wyników

Najlepsze wyniki powinny być sortowane zależnie od ilości wykonanych ruchów, a następnie zależnie od czasu, jaki został wykorzystany na zakończenie gry. W naszym przypadku klasa służąca do reprezentacji pojedynczego wyniku wygląda następująco:

```
public class PlayerScore
{
    public string PlayerName { get; set; }
    public int NumberOfMoves { get; set; }
    public long TimeSpanTicks { get; set; }
}
```

Dodatkowo, dobrym pomysłem byłaby implementacja interfejsu `IComparable<PlayerScore>` w celu łatwego sortowania listy wyników. Pełna definicja klasy `PlayerScore` znajduje się w pliku *PlayerScore.cs*.

Wyniki zapisywane powinny być do pliku XML. Serializacja listy graczy odbywa się przy pomocy statycznej klasy `Serializer` udostępniającej dwie publiczne metody:

- `string SerializeScoreList(List<PlayerScore> scoreList)` - metoda służąca do serializacji listy graczy do ciągu tekstowego reprezentującego plik XML.
- `List<PlayerScore> DeserializeScoreList(string xmlDoc)` - metoda, która umożliwia deserializację zawartości pliku XML do listy graczy.

Należy jeszcze stworzyć prostą klasę, która umożliwi zapis oraz odczyt listy wyników z pliku. Jako że tworzona przez nas gra będzie uruchamiana na różnych urządzeniach, plik ten będzie zapisywany w katalogu *Roaming*, dzięki czemu najlepsze wyniki będą synchronizowane pomiędzy wszystkimi urządzeniami z których korzysta użytkownik naszej aplikacji. Klasa ta powinna implementować następujący interfejs:

```
public interface IScoreManager
{
    /// <summary>
    /// Posortowana lista zawierająca najlepsze wyniki
    /// </summary>
    IReadOnlyList<PlayerScore> TopScores { get; }

    /// <summary>
    /// Dodaj nowy wynik do posortowanej listy i zapisz zmiany do pliku
    /// </summary>
    /// <param name="playerScore">Wynik do dodania</param>
    Task AddScoreAsync(PlayerScore playerScore);

    /// <summary>
    /// Zapisz obecną listę do pliku
    /// </summary>
    /// <param name="fileName">Nazwa pliku</param>
    Task WriteToFileAsync(string fileName = "scores.xml");

    /// <summary>
    /// Odczytaj zawartość pliku do listy TopScores
    /// </summary>
    /// <param name="fileName">Nazwa pliku</param>
    Task ReadFromFileAsync(string fileName = "scores.xml");
}
```

Pełna definicja klasy implementującej ten interfejs znajduje się w pliku *ScoreManager.cs*. Należy też zwrócić uwagę na to, że z uwagi na dostęp do plików, wszystkie metody definiowane przez ten interfejs powinny wykonywać się w sposób asynchroniczny.

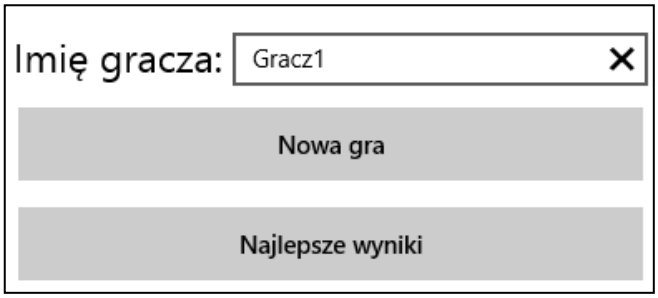
5.3. Interfejs graficzny

Do stworzenia interfejsu graficznego wykorzystane zostaną mechanizmy omówione w rozdziale 2. Zaprezentowane zostaną także sposoby wykorzystania tych mechanizmów do współdzielenia kodu XAML. Warto też wspomnieć, że dobrą praktyką w przypadku tworzenia interfejsu graficznego w języku XAML jest skorzystanie ze wzorca architektonicznego Model-View-ViewModel (MVVM), jednak z uwagi na to, że nie został on w tej pracy omówiony, gra zostanie stworzona bez jego wykorzystania. Przed rozpoczęciem pracy należy także upewnić się, że w projekcie współdzielonym znajdują się klasy definiujące Suspenion Managera oraz Navigation Helpera. Zostaną one automatycznie dodane po do-

daniu nowej strony korzystającej z szablonu `BasicPage` do projektu Windows 8.1 lub Windows Phone 8.1.

5.3.1. Menu główne

Na stronie z głównym menu powinny znajdować się jedynie trzy elementy: pole tekstowe umożliwiające ustawienie imienia gracza (o długości od 3 do 10 znaków), przycisk umożliwiający przejście do listy z najlepszymi wynikami oraz przycisk umożliwiający rozpoczęcie nowej gry. Z uwagi na prostotę tej strony, zaprojektowanie jej w taki sposób, aby była ona kompatybilna zarówno z systemem Windows 8.1 jak i Windows Phone 8.1 nie powinno stanowić problemu. W projekcie współdzielonym stwórzmy nową stronę o nazwie PageMainMenu. Przykładowy projekt strony z głównym menu zaprezentowany jest na *Rysunku 9*. Poniżej rysunku umieszczony został także kod XAML tej strony.



Rysunek 9: Przykładowy projekt strony z głównym menu

```
<Grid VerticalAlignment="Center" HorizontalAlignment="Center">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>

    <TextBlock Text="Imię gracza:" Grid.Row="0" Grid.Column="0"
               VerticalAlignment="Center" FontSize="24" Margin="0,0,5,0"/>

    <TextBox Name="TextBoxName" Grid.Row="0" Grid.Column="1" Width="250"/>

    <Button Name="ButtonStartGame" Content="Nowa gra"
            Grid.Row="1"  Grid.Column="0" Grid.ColumnSpan="2"
            HorizontalAlignment="Stretch"
            Height="50" Margin="0,10,0,0"
            Click="ButtonStartGame_Clicked"/>

    <Button Name="ButtonHighScores" Content="Najlepsze wyniki"
            Grid.Row="2"  Grid.Column="0" Grid.ColumnSpan="2"
            HorizontalAlignment="Stretch"
```

```
            Height="50" Margin="0,10,0,0"
            Click="ButtonHighScores_Clicked"/>
</Grid>
```

W pliku *PageMainMenu.cs* należy jeszcze stworzyć dwie metody: `ButtonStartGame_Clicked` oraz `ButtonHighScores_Clicked` obsługujące zdarzenia wciśnięcia przycisku. Po wciśnięciu przycisku `ButtonHighScores` aplikacja powinna przejść do strony z listą najlepszych wyników. Natomiast po wciśnięciu przycisku `ButtonStartGame` aplikacja powinna sprawdzić poprawność imienia wprowadzonego przez gracza, nadpisać ostatnio wykorzystane imię w ustawieniach aplikacji, a następnie przejść do strony z planszą do gry. Metody te powinny wyglądać zatem następująco:

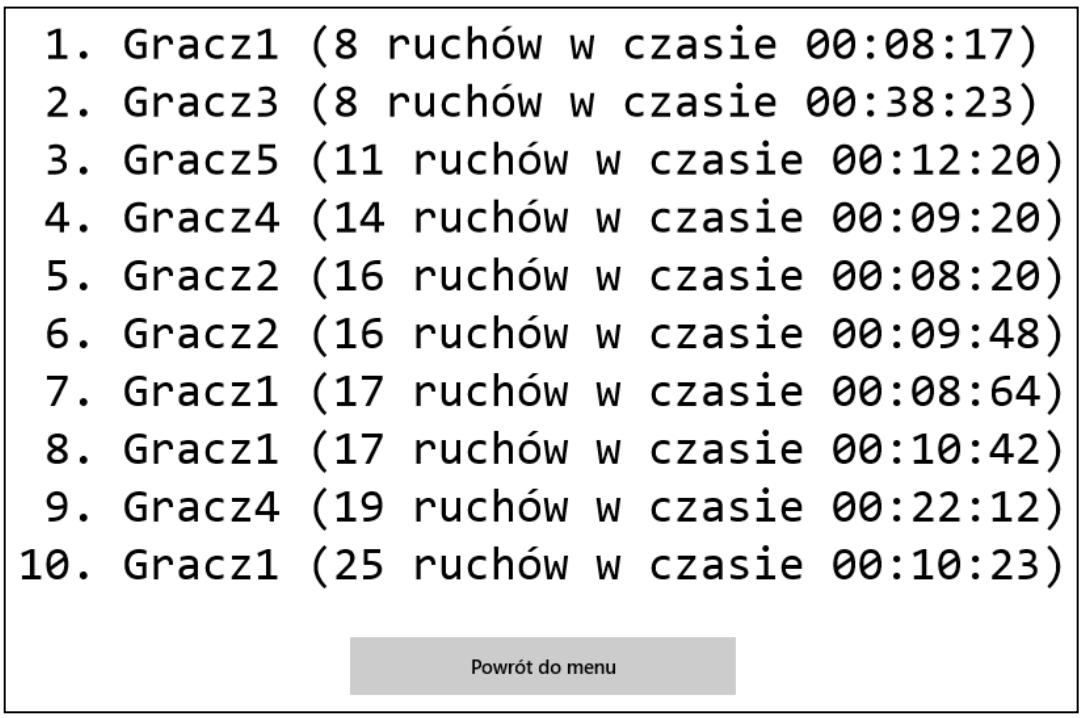
```
private void ButtonStartGame_Clicked(object sender, RoutedEventArgs e)
{
    if (TextBoxName.Text.Length < 3 || TextBoxName.Text.Length > 10)
    {
        const string Message = "Imię musi mieć długość od 3 do 10 znaków!";
        new MessageDialog(Message).ShowAsync();
        return;
    }
    var roamingSettings = ApplicationData.Current.RoamingSettings;
    roamingSettings.Values["lastPlayerName"] = TextBoxName.Text;
    this.Frame.Navigate(typeof(PageGame), TextBoxName.Text);
}

private void ButtonHighScores_Clicked(object sender, RoutedEventArgs e)
{
    this.Frame.Navigate(typeof(PageHighScores));
}
```

Oczywiście strony `PageGame` oraz `PageHighScores` jeszcze nie istnieją, jednak zostaną one utworzone w podrozdziałach 5.3.2 oraz 5.3.4. W pliku *PageMainMenu.cs* można także usunąć metody obsługujące zdarzenia `SaveState` oraz `LoadState` Navigation Helpera, ponieważ strona z głównym menu nie zawiera danych, które powinny zostać zapamiętane w momencie wstrzymania aplikacji. Metody `OnNavigatedTo` oraz `OnNavigatedFrom` Navigation Helpera powinny jednak nadal być wykonywane, w celu zapewnienia poprawnego funkcjonowania metody `GoBack`.

5.3.2. Strona z najlepszymi wynikami

Strona z głównymi wynikami zawierać powinna dwa elementy: posortowaną listę najlepszych wyników oraz przycisk umożliwiający powrót do głównego menu. Stwórzmy zatem w projekcie współdzielonym nową stronę o nazwie `PageHighScores`. Przykła-

dowy projekt strony z najlepszymi wynikami zaprezentowany jest na *Rysunku 10*. Poniżej rysunku umieszczony został także kod XAML tej strony.



Rysunek 10: Przykładowy projekt strony z najlepszymi wynikami

```
<Grid VerticalAlignment="Center">
    <Grid.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="/Styles/ViewBoxScoreStyle.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Grid.Resources>

    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Viewbox Name="ViewboxMain" Stretch="Uniform" Grid.Row="0"
             Style="{StaticResource ViewboxScoreStyle}" MaxWidth="800">
        <StackPanel Name="StackPanelMain" Orientation="Vertical"
                    VerticalAlignment="Top" HorizontalAlignment="Center"/>
    </Viewbox>
    <Button Grid.Row="1"
            Command="{Binding NavigationHelper.GoBackCommand,
                              ElementName=PageRoot}"
            Width="{Binding ElementName=ViewboxMain, Path=ActualWidth}"
            HorizontalAlignment="Center" Height="50" MinWidth="300"
            Content="Powrót do menu"/>
</Grid>
```

Główny element strony, czyli lista najlepszych wyników znajduje się w StackPanelu o nazwie `StackPanelMain`. Ten StackPanel jest z kolei umieszczony wewnątrz elementu

`ViewBox`. `ViewBox` jest kontrolką, która automatycznie skaluje swoją zawartość w sposób zależny od wartości parametru `Stretch`.

Taka konfiguracja strony z wynikami nie jest jednak automatycznie kompatybilna z systemem Windows 8.1 i Windows Phone 8.1. Zauważmy, że elementowi `ViewBox` przypisywany jest styl `ViewBoxScoreStyle`, który zdefiniowany jest w słowniku (ResourceDictionary) znajdującym się w pliku */Styles/ViewBoxScoreStyle.xaml*. Ten plik znajduje się w projektach przeznaczonych dla Windows 8.1 oraz Windows Phone 8.1. Oznacza to, że zależnie od platformy, na której nasza aplikacja jest uruchamiana, kontrolce `ViewBox` przypisany zostanie inny styl. Dzięki takiemu rozwiązaniu możemy uniknąć dwukrotnego tworzenia strony z najlepszymi wynikami, zapewniając kompatybilność z obiema platformami przy pomocy odrębnych stylów.

Należy jeszcze zapełnić listę wyników lub w przypadku, gdy lista wyników jest pusta wyświetlić odpowiedni komunikat. Przykładowy kod uzupełniający listę wyników znajduje sie w pliku *PageHighScores.xaml.cs* w metodzie `PopulateScoreBoard`.

5.3.3. Plansza do gry

Przed przystąpieniem do projektowania ostatniej strony, należałoby zaprojektować własną kontrolkę implementująca planszę do gry. Z uwagi na to, że plansza do gry jest zwykłą siatką, której komórki mogą przyjąć dwa stany, skorzystamy z kontrolki `Grid` o rozmiarze 4x4 wypełnionej kontrolkami `ToggleButton`. Dobrym pomysłem byłoby także stworzenie stylu dla przycisku `ToggleButton`, aby lepiej prezentować stan komórki, w której został on umieszczony. Przykładowy styl znajduje się w projekcie współdzielonym, w pliku *ButtonLightStyle.xaml*. Styl ten implementuje proste animacje, zależnie od stanu wizualnego (ang. *Visual State*) przycisku. Listę wszystkich stanów wizualnych tej kontrolki można znaleźć na stronie Microsoftu [21].

Plik XAML naszej kontrolki jest bardzo prosty z uwagi na to, że definicje kolumn i wierszy oraz zawartość siatki będą generowane z poziomu kodu:

```
<Grid Name="GridMain">
    <Grid.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="/Styles/ButtonLightStyle.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Grid.Resources>
</Grid>
```

Zdefiniujmy właściwości, pola prywatne oraz konstruktor kontrolki. Brakujące metody stworzymy w dalszej części pracy:

```
public string PlayerName { get; set; }

private readonly ILightsOutGame _lightsOutGame;
private readonly IScoreManager _scoreManager;

public GameBoard()
{
    this.InitializeComponent();
    this.IsEnabled = false;

    _lightsOutGame = new LightsOutGame();
    _scoreManager = new ScoreManager();
    _scoreManager.ReadFromFileAsync();

    InitializeGame();
    _lightsOutGame.GameEnded += LightsOutGame_OnGameEnded;
    _lightsOutGame.LightSwitched += LightsOutGame_OnLightSwitched;
    ApplicationData.Current.DataChanged += ApplicationData_OnDataChanged;
}
```

Po zainicjowaniu kontrolki ustawiamy wartość parametru `IsEnabled` na `false`. Wartość ta zostanie ustawiona na `true` po rozpoczęciu gry. Dzięki temu kontrolka nie będzie reagowała na akcje gracza jeżeli gra nie została rozpoczęta.

Standardowo gra korzysta jedynie z siatki o rozmiarze 4x4, jednak w celu przećwiczenia dynamicznego modyfikowania kontrolki `Grid` oraz umożliwienia łatwiejszej modyfikacji naszego programu w przyszłości, stwórzmy metody potrafiące zainicjować siatkę dowolnych rozmiarów. Metoda `InitializeGame` wraz z metodami pomocniczymi wygląda zatem następująco:

```
// Metoda tworzące definicje kolumn
private void SetColumnDefinitions()
{
    GridMain.ColumnDefinitions.Clear();
    for (int i = 0; i < _lightsOutGame.ColumnCount; i++)
    {
        var columnDefinition = new ColumnDefinition
        {
            Width = new GridLength(1, GridUnitType.Star)
        };

        GridMain.ColumnDefinitions.Add(columnDefinition);
    }
}

// Metoda tworzące definicje wierszy - analogicznie do SetColumnDefinitions()
[...]

//Metoda wypełniająca siatkę przyciskami
private void InitializeGame()
{
    // CZyscimy siatke i ustawiamy definicje wierszy i kolumn
```

```
        GridMain.Children.Clear();
        SetRowDefinitions();
        SetColumnDefinitions();

        // Dodajemy kontrolki ToggleButton do siatki
        for (int i = 0; i < _lightsOutGame.RowCount; i++)
        {
            for (int j = 0; j < _lightsOutGame.ColumnCount; j++)
            {
                var newButton = new ToggleButton();
                Grid.SetRow(newButton, i);
                Grid.SetColumn(newButton, j);
                newButton.HorizontalAlignment = HorizontalAlignment.Stretch;
                newButton.VerticalAlignment = VerticalAlignment.Stretch;
                newButton.Click += ToggleButton_OnClick;
                newButton.Style = GridMain.Resources["ButtonLightStyle"] as Style;
                GridMain.Children.Add(newButton);
            }
        }
    }
```

W każdym z przycisków siatki przechwytujemy zdarzenie `Click`. Wciśnięcie przycisku przez gracza jest jednoznaczne z wykonaniem ruchu. Zatem za każdym razem, po wciśnięciu przycisku wykonana powinna zostać metoda `PlayerAction` obiektu `_lightsOutGame`:

```
    private void ToggleButton_OnClick(object sender, RoutedEventArgs e)
    {
        var toggleButton = (ToggleButton)sender;
        var row = Grid.GetRow(toggleButton);
        var column = Grid.GetColumn(toggleButton);
        _lightsOutGame.PlayerAction(row, column);
    }
```

Przechwytywane są także zdarzenia `GameEnded` oraz `LightSwitched` obiektu `_lightsOutGame`. Zdarzenia te są wywoływane odpowiednio w chwili zakończenia gry oraz w chwili zmiany stanu jednej z komórek. W momencie zakończenia gry aplikacja powinna wyświetlić komunikat z wynikiem (ilość ruchów oraz czas). Wynik ten powinien następnie zostać dodany do listy wyników przez obiekt `_scoreManager`. Natomiast w chwili zmiany stanu danej komórki odpowiedni przycisk na naszej siatce także powinien zmienić swój stan:

```
    void LightsOutGame_OnLightSwitched(object sender, LightSwitchedEventArgs e)
    {
        var row = e.Row;
        var column = e.Column;
        var gridIndex = row * _lightsOutGame.ColumnCount + column;
        var button = (ToggleButton)GridMain.Children[gridIndex];
        button.IsChecked = _lightsOutGame[row, column];
    }

    void LightsOutGame_OnGameEnded(object sender, GameEndedEventArgs e)
    {
```

```
            this.IsEnabled = false;

            var score = new PlayerScore(PlayerName, _lightsOutGame.MoveCount,
                                        e.TimeSpan.Ticks);
            _scoreManager.AddScoreAsync(score);

            var formattedString = "Gra zakonczona!\n";
            formattedString += string.Format("Ilość ruchów: {0}\n",
                                             _lightsOutGame.MoveCount);
            formattedString += string.Format("Czas: {0:D2}:{1:D2}:{2:D2}",
                                             e.TimeSpan.Minutes,
                                             e.TimeSpan.Seconds,
                                             e.TimeSpan.Milliseconds);

            var dialog = new MessageDialog(formattedString);
            dialog.ShowAsync();
        }
```

Z uwagi na to, że plik z wynikami zapisywany jest w katalogu *Roaming*, istnieje niebezpieczenstwo, że zostanie on nadpisany przez inne urządzenie w trakcie gry. Gdy to nastąpi, wywołane zostanie zdarzenie `DataChanged`. W takiej sytuacji użytkownik powinien być poinformowany o wystąpieniu konfliktu, oraz mieć możliwość wyboru wersji pliku z której chce dalej korzystać. W naszym przypadku metoda obsługująca zdarzenie `DataChanged` wygląda następująco:

```
async void ApplicationData_OnDataChanged(ApplicationData sender, object args)
{
    var message = "Inne urządzenie nadpisało plik z najlepszymi wynikami!
                   Czy chcesz wczytać ten plik i nadpisać obecne wyniki?";
    var clickedYes = await MessageDialogHelper.DisplayYesNoDialogAsync(message);
    if (clickedYes)
    {
        await _scoreManager.ReadFromFileAsync();
    }
    else
    {
        await _scoreManager.WriteToFileAsync();
    }
}
```

5.3.4. Strona z planszą do gry

Strona z planszą do gry powinna zawierać dwa dodatkowe elementy: przycisk umożliwiający powrót do głównego menu oraz przycisk pozwalający na wygenerowanie nowego układu i rozpoczęcie nowej gry. Podobnie jak w przypadku głównego menu, stronę tą można zaprojektować tak, aby była kompatybilna z obiema platformami. Najpierw jednak konieczne jest dodanie do strony przestrzeni nazw, która zawiera stworzoną przez nas wcześniej kontrolkę:

```
    xmlns:ctrls="using:MiniLightsOut.Controls"
```

Zdefiniujmy siatkę, która zawierać będzie elementy naszej strony. W pierwszym rzędzie znajdować się będzie plansza do gry, natomiast w drugim przyciski pozwalające na rozpoczęcie nowej gry oraz powrót do głównego menu:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    </Grid>
</Grid>
```

Po zdefiniowaniu siatki możemy umieścić w niej opisaną w podrozdziale 5.3.3 kontrolkę. W celu zapewnienia jej poprawnych rozmiarów na obu platformach niezależnie od rozdzielczości, rozmiar tej kontrolki wiążemy z wysokością strony, a następnie korzystamy z kontrolki `ViewBox` do jej przeskalowania:

```
<Viewbox Grid.Row="0">
    <ctrls:GameBoard x:Name="GameBoard"
      Height="{Binding ElementName=PageRoot, Path=ActualHeight, Mode=OneTime}"
      Width="{Binding ElementName=PageRoot, Path=ActualHeight, Mode=OneTime}"
    />
</Viewbox>
```

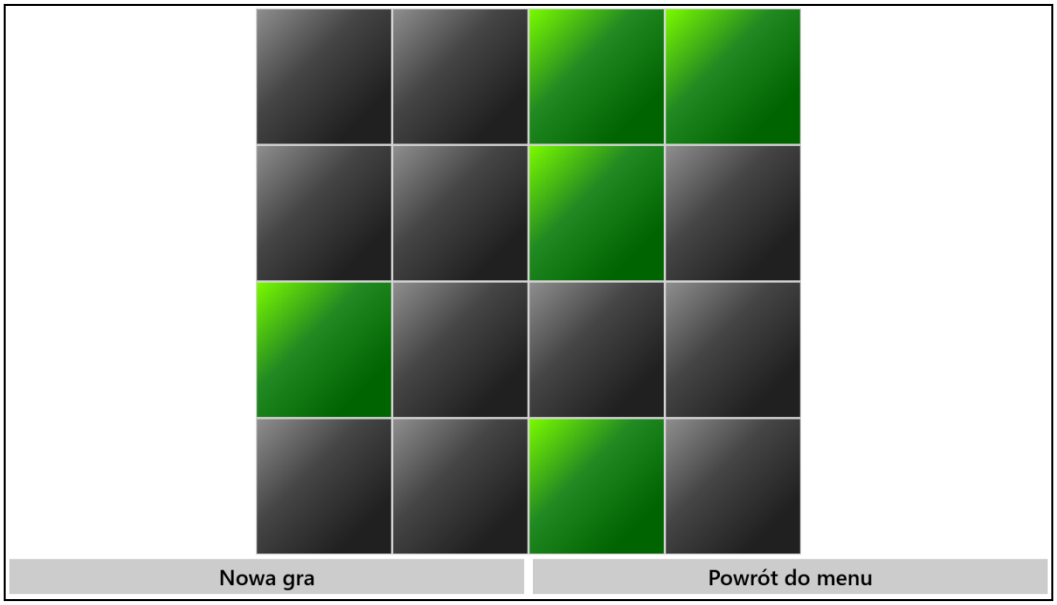
Pozostaje jeszcze dodanie do strony brakujących przycisków. Ponieważ przyciski te będą do siebie podobne, skorzystamy ze stylu, co wymusi uzgodnienie wartości niektórych atrybutów:

```
<Grid Grid.Row="1" HorizontalAlignment="Stretch">
    <Grid.Resources>
        <Style TargetType="Button">
            <Setter Property="Height" Value="70"/>
            <Setter Property="Margin" Value="5"/>
            <Setter Property="HorizontalAlignment" Value="Stretch"/>
        </Style>
    </Grid.Resources>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Button Content="Nowa gra"
            Grid.Column="0"
            Click="ButtonNewGame_Clicked"/>
    <Button Content="Powrót do menu"
            Grid.Column="1"
            Command="{Binding NavigationHelper.GoBackCommand,
                      ElementName=PageRoot}"/>
</Grid>
```

Wciśnięcie przycisku "*Nowa gra*" powinno wywoływać metodę `NewGame` stworzonej przez nas wcześniej kontrolki z planszą do gry:

```
private void ButtonNewGame_Clicked(object sender, RoutedEventArgs e)
{
    GameBoard.NewGame();
}
```

Wygląd tak zaprojektowanej strony z planszą do gry pokazany jest na *Rysunku 11*.



Rysunek 11: Przykładowy projekt strony z planszą do gry

W projekcie dołączonym do pracy użyty został mechanizm zapamiętywania stanu sesji z wykorzystaniem Suspension Managera. Nie jest to oczywiście konieczne w przypadku tak małej aplikacji, stanowi jednak ilustracje dla informacji zawartych w rozdziale 4.4.1.

6. Podsumowanie

Efektem pracy jest tutorial będący wprowadzeniem do tworzenia Aplikacji Uniwersalnych z wykorzystaniem języka C#. Tutorial ten przeznaczony jest dla programistów znających język C#, jednak nie mających doświadczenia w tworzeniu aplikacji na platformę WinRT, w tym Aplikacji Uniwersalnych. W drugim rozdziale został omówiony język XAML, który służy do tworzenia interfejsu graficznego użytkownika. Omówione zostały podstawowe kontrolki służące do organizacji interfejsu oraz podstawowe mechanizmy, które znacznie ułatwią tworzenie warstwy prezentacji aplikacji. Dodatkowo, większość omówionych w tym rozdziale mechanizmów współpracuje także z Windows Presentation Foundation, co sprawia, że rozdział ten jest bardziej ogólny i przydatny dla szerszej grupy odbiorców. W trzecim rozdziale omówione zostały podstawowe idee oraz mechanizmy wykorzystywane przez Aplikacje Uniwersalne. Przede wszystkim omówiony został plik AppManifest, będący sercem konfiguracji aplikacji oraz strategie współdzielenia kodu, które przyśpieszają proces tworzenia oprogramowania na wiele platform jednocześnie oraz ułatwiają dalsze rozwijanie kodu w przyszłości. Czwarty rozdział zawiera wybrane dodatkowe zagadnienia, których znajomość pozwoli na pełniejsze wykorzystanie możliwości platformy WinRT. Omówione zostały powiadomienia Toast, służące do komunikacji z użytkownikiem, dynamiczne kafelki umożliwiające wyświetlanie różnego rodzaju informacji w menu start oraz techniki służące do zarządzania cyklem życia aplikacji, co jest niezbędne w przypadku każdej zaawansowanej aplikacji. W ostatnim rozdziale zaprezentowany został proces tworzenia przykładowej Aplikacji Uniwersalnej, co daje czytelnikowi praktyczny przykład zastosowania omówionych w tej pracy mechanizmów.

Należy jednak zwrócić uwagę na fakt, że Aplikacje Uniwersalne są bardzo obszernym tematem, dlatego poruszenie wszystkich zagadnień w jednej pracy nie jest możliwe. Z tego powodu praca ta jest jedynie częścią całego tutorialu wraz z innymi pracami dyplomowymi pisanymi przez studentów Wydziału Fizyki Astronomii i Informatyki Stosowanej UMK. Pozostałe prace poruszają inne zagadnienia, które nie były tutaj omówione, zatem w celu uzupełnienia swojej wiedzy zalecane jest przeczytanie także pozostałych prac. Samodzielnie praca ta jest jedynie wprowadzeniem do Aplikacji Uniwersalnych, wystarczającym do tego, aby korzystając z informacji w niej zawartych, programista był w stanie rozpocząć pracę nad własną aplikacją.

Literatura

[1] Windows Dev Center. [Online]. https://msdn.microsoft.com/en-us/library/cc295302.aspx

[2] DPI and Device-Independent Pixels (Windows). [Online]. https://msdn.microsoft.com/en-us/library/windows/desktop/ff684173%28v=vs.85%29.aspx

[3] Michael Mayberry, *WinRT Revealed.*, 2012.

[4] John Garland Jeremy Likness, *Programming the Windows Runtime by Example: A Comprehensive Guide to WinRT with Examples in C# and XAML*.

[5] Charles Petzold, *Programming Windows, 6th Edition*.

[6] Joel Ivory Johnson. CodeProject - WPF Data Binding. [Online]. http://www.codeproject.com/Articles/29054/WPF-Data-Binding-Part

[7] Abel Avram. InfoQ: Software Development News, Videos & Books. [Online]. http://www.infoq.com/news/2011/09/Design-Details-Windows-Runtime

[8] Technopedia. [Online]. http://www.techopedia.com/definition/27608/windows-runtime-library-winrt-windows-8

[9] Abel Avram. InfoQ. [Online]. http://www.infoq.com/news/2011/09/Design-Details-Windows-Runtime

[10] Microsoft. Cross-Platform Development with the Portable Class Library. [Online]. https://msdn.microsoft.com/en-us/library/gg597391%28v=vs.110%29.aspx

[11] App manifest file for Windows Phone 8. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/ff769509%28v=vs.105%29.aspx

[12] Microsoft. App capability declarations. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx#domaincredentials

[13] App contracts and extensions (Windows Runtime apps). [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/hh464906.aspx

[14] Auto-launching with AutoPlay. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/hh452731.aspx

[15] The cached file updater contract. [Online]. http://www.jonathanantoine.com/2013/03/25/win8-the-cached-file-updater-contract-or-how-to-make-more-useful-the-file-save-picker-contract/

[16] How to customize camera options with a Windows Store device app. [Online]. http://go.microsoft.com/fwlink/p/?LinkId=320605

[17] Integrating with file picker contracts. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/hh465174.aspx

[18] How to customize print settings. [Online]. https://msdn.microsoft.com/en-us/library/windows/hardware/dn391713%28v=vs.85%29.aspx

[19] App data storage. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/hh464917.aspx

[20] ApplicationData.DataChanged | datachanged event. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.applicationdata.datachanged.aspx

[21] ToastTemplateType enumeration. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/xaml/windows.ui.notifications.toasttemplatetype.aspx

[22] The tile template catalog. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/hh761491.aspx

[23] Sending a tile update. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/hh465439.aspx

[24] ToggleButton styles and templates. [Online]. https://msdn.microsoft.com/en-us/library/windows/apps/xaml/jj709930.aspx