POLITECHNIKA LUBELSKA

PODRĘCZNIKI

Marcin Badurowicz
Beata Pańczyk

Programowanie aplikacji
dla systemu Windows Phone

Lublin 2014

Programowanie aplikacji
dla systemu Windows Phone

Podręczniki – Politechnika Lubelska

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Publikacja współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Beata Pańczyk
Marcin Badurowicz

Programowanie aplikacji
dla systemu Windows Phone

Politechnika Lubelska
Lublin 2014

Recenzent:
dr inż. Jakub Smołka

Publikacja dystrybuowana bezpłatnie.

Publikacja finansowana z projektu „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW"

Projekt „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW" współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego. Umowy UDA-POKL.04.03.00-00-129/12

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-7947-065-5

Wydawca: Politechnika Lubelska
ul. Nadbystrzycka 38D, 20-618 Lublin
Realizacja: Biblioteka Politechniki Lubelskiej
Ośrodek ds. Wydawnictw i Biblioteki Cyfrowej
ul. Nadbystrzycka 36A, 20-618 Lublin
tel. (81) 538-46-59, email: wydawca@pollub.pl
www.biblioteka.pollub.pl
Druk: TOP Agencja Reklamowa Agnieszka Łuczak
www.agencjatop.pl

Elektroniczna wersja książki dostępna w Bibliotece Cyfrowej PL www.bc.pollub.pl
Nakład: 100 egz.

Spis treści

Wstęp

Platforma Windows Phone jest świeżym podejściem do budowy coraz powszechniejszego trendu w aplikacjach użytkowych – aplikacji mobilnych.

Treść niniejszego podręcznika stanowią wybrane zagadnienia dotyczące programowania aplikacji w języku C# dla platformy Windows Phone. Teoretyczne podstawy bazują na pozycjach [1, 2, 3], które obejmują znacznie obszerniejszy materiał. Niniejszy podręcznik zawiera tylko wyselekcjonowane informacje, które są zwykle omawiane na wykładach. Autorzy ograniczyli się do niezbędnych elementów teorii, bardziej koncentrując się na przykładach dobranych w taki sposób, aby jak najprościej zobrazować omawiany temat. Każdy rozdział zawiera odpowiednio opracowane zbiory zadań programistycznych. Samodzielna realizacja tych zadań pomoże Czytelnikowi w utrwaleniu i zrozumieniu prezentowanego materiału.

Poszczególne rozdziały omawiają kolejne elementy programowania aplikacji i koncentrują się przede wszystkim na uniwersalnych przykładach opracowanych w trakcie zajęć prowadzonych przez autorów w ostatnich latach ze studentami kierunku Informatyka.

Wszelkie uwagi i propozycje prosimy kierować do autorów na adres *b.panczyk@pollub.pl* lub *m.badurowicz@pollub.pl*.

Autorzy

1. Platforma Windows Phone

1.1. Wstęp

Platforma Windows Phone w wersji 7 została udostępniona użytkownikom i programistom jesienią 2010 roku i przede wszystkim reprezentowała nowe, świeże podejście do kwestii mobilnych systemów operacyjnych oraz zrywała kompatybilność ze wszystkimi wcześniejszymi systemami mobilnymi firmy Microsoft. Przez dwa lata rozwoju platformy udostępniono trzy znaczące aktualizacje, dodające nowe funkcje zarówno dla użytkowników, jak i nowe, oczekiwane przez programistów, możliwości.

Windows Phone 8 pojawił się dwa lata później, oferując na bazie swojego poprzednika dwie kolejne innowacje – nowe jądro systemu, oparte na swoim odpowiedniku ze stacjonarnych komputerów – Windows 8; oraz elementy nowego interfejsu programistycznego oraz dziesiątki nowych możliwości, niedostępnych w poprzedniej wersji.

Z punktu widzenia użytkownika, Windows Phone wyróżnia się spośród pozostałych systemów operacyjnych na urządzeniach mobilnych dzięki swojemu interfejsowi graficznemu, opartemu na zasadach tzw. Microsoft Design Language (poprzednio znanemu jako Metro lub Metro UI), gdzie interfejs użytkownika polega ściśle na prostocie, „czystości" oraz płynności, z naciskiem na informacje przekazywane użytkownikowi, bez ozdobników graficznych.

Rysunek 1.1. przedstawia ekran *Start* telefonu z systemem Windows Phone 8 z układem tzw. żywych kafelków (ang. live tiles).

Windows Phone 7 oraz Windows Phone 8 są niekompatybilne z wcześniejszym systemem mobilnym firmy Microsoft, Windows Mobile, inny jest też model programowania na tę platformę, o którym więcej informacji przedstawiono w rozdziale 1.3.

Rys. 1.1. Ekran Start telefonu z systemem Windows Phone 8 (emulator)

1.1.1. Wersje platformy Windows Phone

System Windows Phone jest również nazywany Windows Phone OS i taka nazwa pojawia się m.in. w Visual Studio oraz w dokumentacji. Jego wersja 7.0, która pojawiła się na samym początku, nie jest już wspierana, ani jej wersja zaktualizowana, tzw. „NoDo". Została ona zastąpiona wersją oznaczaną numerem wewnętrznym 7.1.

Windows Phone OS 7.1 był znany pod nazwą kodową „Mango" i został udostępniony oficjalnie jako Windows Phone 7.5, ale pomimo tego numer wewnętrzny 7.1 nadal jest stosowany w dokumentacji. Udostępniony pod koniec 2012 roku Windows Phone 7.8 również posiada numer wewnętrzny 7.1, jednak w środowiskach programistycznych oznaczana jest najczęściej swoim numerem głównym (np. przy wyborze emulatora). Pomiędzy wydaniami 7.5 oraz 7.8 wy-

dano kilka pomniejszych aktualizacji systemu, m.in. *Tango* wprowadzające modyfikację wymagań sprzętowych.

W momencie pisania tego wydania książki, najświeższa udostępniona wersja platformy ma numer 8.0.10328, jest to tzw. General Distribution Release 2, a kolejne aktualizacje (GDR3 oraz Windows Phone 8.1) są zaplanowane na koniec 2013 i początek 2014 roku.

1.1.2. Wymogi sprzętowe platformy

Telefon działający pod kontrolą systemu operacyjnego Windows Phone musi, niezależnie od producenta, spełniać pewną minimalną konfigurację sprzętową. Takie rozwiązanie ogranicza producentów sprzętu, ale zapewnia kompatybilność oprogramowania oraz wymusza jednakową jakość działania systemu i aplikacji na wszystkich urządzeniach.

W przypadku platformy w wersji 7.8 procesor główny urządzenia powinien być taktowany co najmniej częstotliwością 0,8 GHz, telefon powinien mieć co najmniej 256 MB RAM, ekran mieć rozdzielczość 800x480 pikseli i musi zapewniać co najmniej cztery punkty dotyku za pomocą technologii pojemnościowej. Wymagany jest także odbiornik sieci bezprzewodowej Wi-Fi, aparat cyfrowy o rozdzielczości co najmniej 5 megapikseli, akcelerometr, odbiornik systemu nawigacyjnego GPS i radio FM. Niezbędne są także przyciski sprzętowe „Wstecz", „Start", „Szukaj", przycisk blokady telefonu oraz przyciski zmiany głośności.

W wersji 7.5 wymagany był również fizyczny przycisk spustu migawki, to ograniczenie zostało usunięte później. Limit 256 MB RAM początkowo wynosił 512 MB i został ograniczony aby wprowadzić na rynek tańsze modele telefonów.

Urządzenia mogą również mieć dodatkowe możliwości, takie jak żyroskop, kompas cyfrowy (magnetometr), przednią kamerkę wideo czy też sprzętową klawiaturę QWERTY i niektóre modele korzystają z tych możliwości.

W przypadku wersji 8.0 zmodyfikowane zostały wymagania sprzętowe aby sprostać nowym trendom na rynku. Od tej wersji wymagany jest procesor dwurdzeniowy Qualcomm Snapdragon S4 o częstotliwości taktowania minimum 1,5 GHz, 512 MB RAM dla telefonów o rozdzielczości WVGA lub 1 GB dla telefonów o rozdzielczości WXGA lub 720p, a co za tym idzie pojawiła się możliwość obsługi ekranów o rozdzielczościach 1280x720 oraz 1280x768.

1.2. Środowisko programistyczne

Do tworzenia aplikacji dla platformy Windows Phone można wykorzystywać środowisko programistyczne Microsoft Visual Studio w wersji 2012 lub nowszej. W momencie pisania tej książki najnowszą wersją jest wersja 2012, a wersja 2013 pozostaje w fazie wczesnej wersji testowej.

Oprócz tego należy doinstalować darmowy pakiet SDK ze strony Windows Phone Dev Center:
https://dev.windowsphone.com/en-us/downloadsdk.

Pakiet SDK można również zainstalować na komputerze bez zainstalowanej jakiejkolwiek wersji pakietu Visual Studio, spowoduje to automatyczne doinstalowanie darmowego środowiska Microsoft Visual Studio Express for Windows Phone, które może być wykorzystane do budowy aplikacji, nie udostępniając niektórych funkcji z wydań komercyjnych, ale pozwalając także na wytwarzanie aplikacji komercyjnych bez ograniczeń. W momencie pisania tej książki udostępniony został również pakiet SDK, aktualizujący emulatory urządzeń do wersji 8.0.10322, tzw. GDR2.

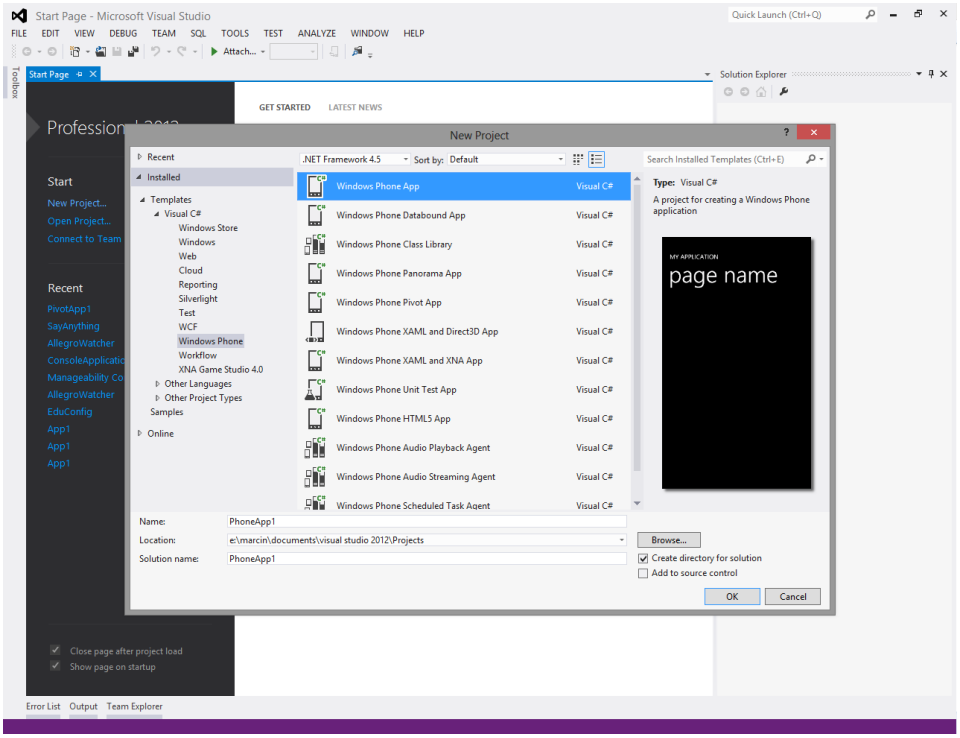
Instalując pakiet Visual Studio należy jednak zwrócić uwagę na wymagania sprzętowe – aby móc wykorzystywać emulator platformy Windows Phone należy używać komputera z systemem operacyjnym Windows 8 lub nowszym wyposażonym w procesor z obsługą tzw. SLAT (Second Level Address Translation). Dostępność technologii SLAT można sprawdzić np. narzędziem coreinfo (dostępnym na stronie:
http://technet.microsoft.com/en-us/sysinternals/cc835722.aspx).

Bez obsługi technologii SLAT możliwe jest uruchamianie aplikacji tylko poprzez podłączenie fizycznego urządzenia.

W tej książce przykłady będą bazowały na wersji Visual Studio 2012 Professional, ale będzie się je dało również uruchomić na wersji Express.

Rysunek 1.2 przedstawia listę możliwych projektów aplikacji Windows Phone dostępnych w środowisku Visual Studio po zainstalowaniu pakietu SDK.

Językiem programowania, który będzie wykorzystywany w przykładach w tej książce jest język C# oparty na platformie .NET Framework. Windows Phone i Visual Studio pozwalają również na wykorzystanie języka Visual Basic.NET oraz elementów języka C++ wykorzystywanych do tworzenia aplikacji natywnych, o czym będzie szerzej powiedziane w rozdziale 1.3.



Rys 1.2. Lista projektów platformy Windows Phone oraz interfejs Visual Studio 2012

1.3. Typy aplikacji

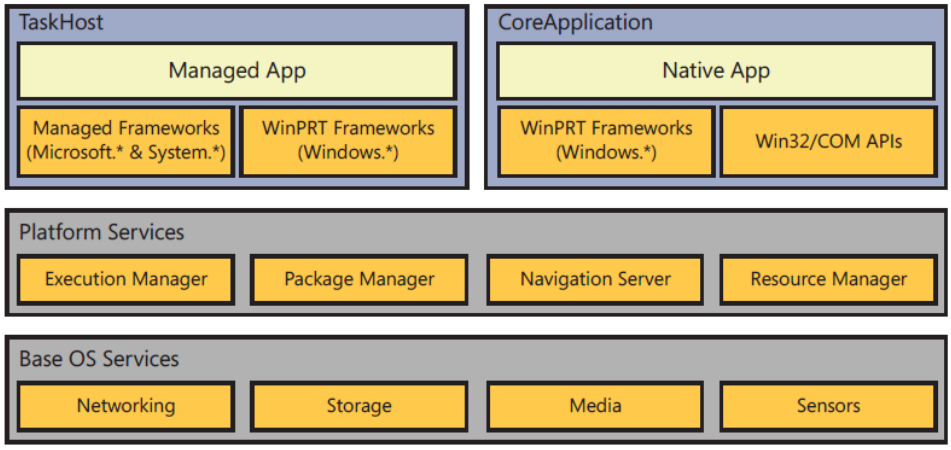
Rysunek 1.3 przedstawia konceptualny diagram architektury platformy Windows Phone 8.

Na szczycie stosu aplikacyjnego znajdują się dwa główne modele aplikacji. TaskHost reprezentuje aplikacje zarządzane, napisane w kodzie zarządzanym platformy Microsoft .NET. Model CoreApplication, niedostępny w wersji 7.x, reprezentuje nowy model aplikacyjny, podzbiór modelu aplikacji Windows 8, który w Windows Phone jest ograniczony tylko do natywnych aplikacji Direct3D.

Aplikacje zarządzane mogą być tworzone z użyciem jednego z dwóch głównych frameworków – Silverlight oraz XNA:

- Silverlight jest podzbiorem Windows Presentation Foundation, pierwotnie przeznaczonym do aplikacji internetowych i oferuje tradycyjne kontrolki interfejsu użytkownika, grafikę wektorową, media, animacje i wiązanie danych – językiem opisu warstwy prezentacji jest tutaj XAML (wymawiaj: zaml);

- XNA to framework do budowy gier obsługująca grafikę rastrową 2D oraz 3D w tradycyjnej pętli gry. XNA jednak jest powoli wycofywany i gry napisane w XNA a nakierowane na wersję 7.0 będą działać, jednak budując aplikację ukierunkowaną na wersję 8.0 nie jest już możliwe wybranie frameworka XNA.



Rys 1.3. Architektura platformy Windows Phone 8 [5]

Windows Phone 8 rozszerza te dwie możliwości o elementy aplikacji natywnych pisanych w języku C++. W obecnej wersji dostępne są tylko aplikacje wykorzystujące interfejs Direct3D lub biblioteki. Aplikacje Direct3D są przeznaczone do tworzenia gier z przenośnością pomiędzy Windows, Windows Phone oraz konsolą Xbox.

Aplikacje napisane w kodzie zarządzanym korzystają z osadzonej (Embedded) wersji platformy .NET Framework, która jest ograniczona w stosunku do pełnej wersji dla urządzeń desktopowych. Mogą również wykorzystywać natywne API opakowane w tzw. komponenty WinPRT (Windows Phone Runtime) oraz pewne API wspólne z modelem Win32/COM, o ile te zostaną opakowane w komponenty WinPRT. Możliwe jest również mieszanie aplikacji natywnych i zarządzanych.

Te dwa główne modele aplikacji osadzone są na współdzielonym zbiorze głównych usług platformy oraz wspólnym zbiorze bazowych usług dostarczanych przez system operacyjny.

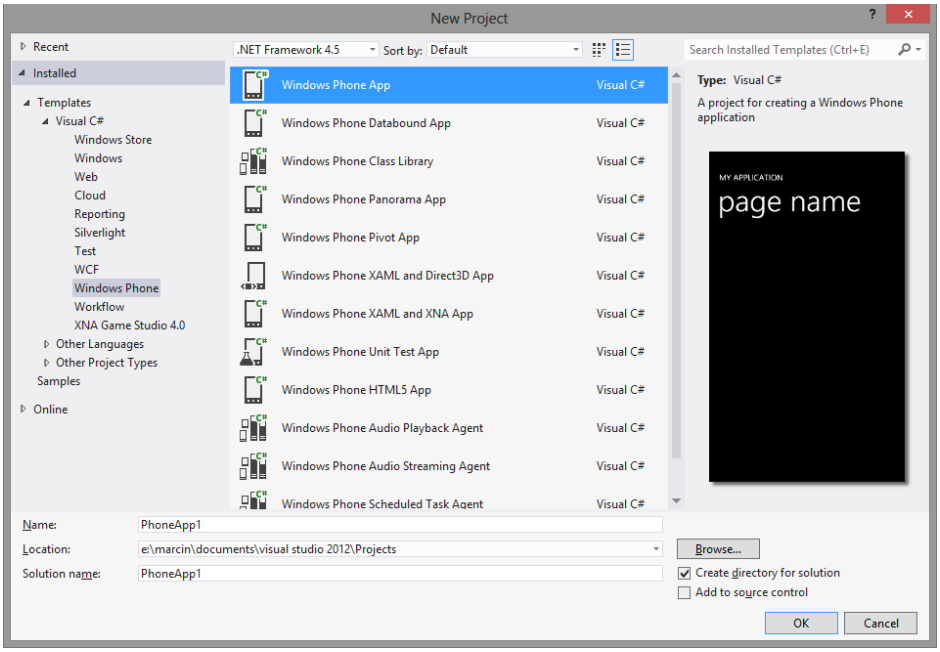
Podsumowując, są trzy możliwe typy aplikacji dla Windows Phone 8:

- aplikacja zarządzana napisana w języku C# lub Visual Basic.NET, której warstwą prezentacji jest Silverlight/XAML;

- aplikacja mieszana napisana w C#, Visual Basic.NET wraz z elementami napisanymi w C++, której warstwą prezentacji będzie Silverlight/XAML oraz Direct3D;
- aplikacja Direct3D napisana w języku C++, która prezentuje treści korzystając tylko z Direct3D.

W tej książce będą omówione głównie elementy aplikacji zarządzanych wraz z opisem tworzenia interfejsu użytkownika we frameworku Silverlight i języku XAML.

Lista możliwych do utworzenia projektów dla typu projektu Windows Phone udostępnia jeszcze kilka szablonów projektów z interfejsem użytkownika oraz bez niego, co dokładniej widać na rysunku 1.4.



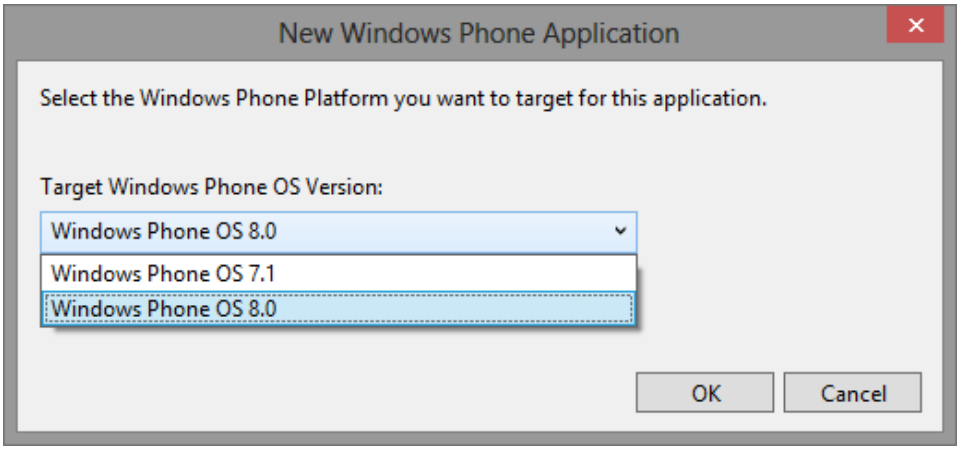
Rys 1.4. Lista możliwych szablonów projektów dla platformy Windows Phone

Z głównych szablonów można wymienić:

- biblioteki klas (Windows Phone Class Library);
- aplikacje wykorzystujące kontrolki Panorama oraz Pivot;
- aplikacje HTML5;
- agenty – odtwarzania audio, streamingu audio oraz zaplanowanych zadań.

1.4. Budowa pierwszej aplikacji

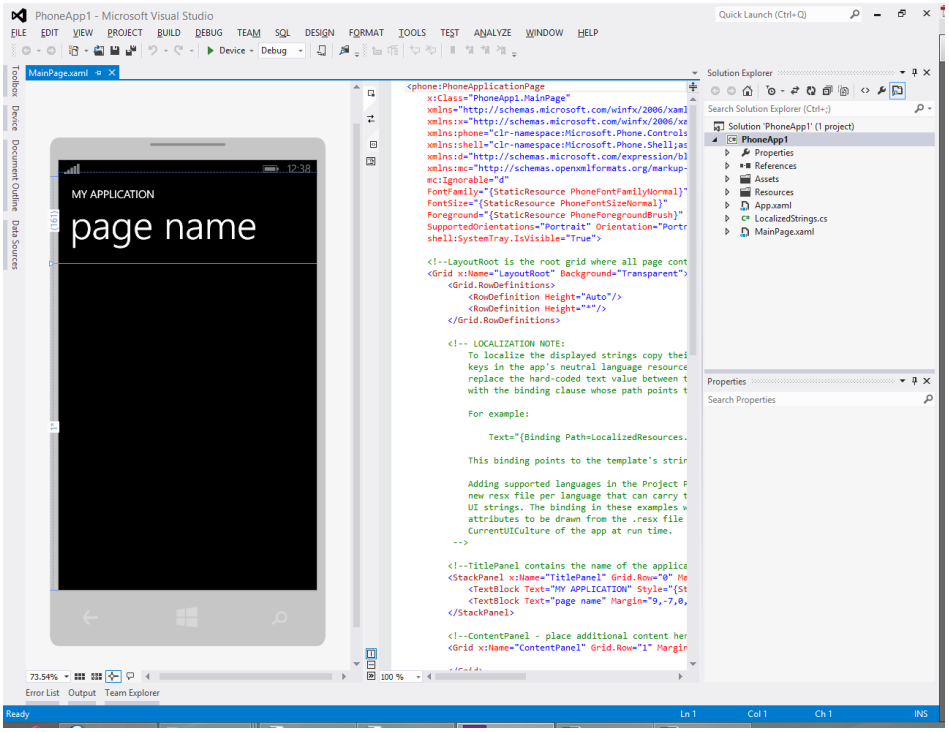
Po utworzeniu szkieletu prostej aplikacji Windows Phone App w języku C# mamy możliwość wyboru wersji platformy, na którą będzie on budowany, jak pokazuje rysunek 1.5.



Rys 1.5. Wybór wersji platformy

Wybór wersji platformy definiuje możliwe do wykorzystania API. Po wyborze platformy w wersji 7.1 możliwe jest uruchomienie takich aplikacji na systemie Windows Phone 8, jednak niemożliwe będzie skorzystanie z niektórych nowych mechanizmów dostępnych w nowszej wersji. Warto zwrócić tutaj też uwagę, że niemożliwe jest już skorzystanie z platformy w wersji 7.0, minimalną wersją jest wersja 7.1.

Rysunek 1.6 demonstruje okno środowiska Visual Studio po utworzeniu nowej aplikacji Windows Phone App.



Rys 1.6. Okno Visual Studio po stworzeniu projektu

Standardowo tworzona jest pierwsza formatka aplikacji, znajdująca się w pliku *MainPage.xaml* oraz *MainPage.xaml.cs*. Aplikacje Windows Phone korzystające z frameworka Silverlight posiadają rozdzielenie warstwy prezentacji oraz warstwy zachowania. Warstwa prezentacji opisana jest za pomocą języka XAML (czyt. zaml), będącego pochodną XML, a wszystkie kontrolki graficzne będą reprezentowane jako znaczniki XAML. Środkowe okno z rysunku 1.6 przedstawia edytor kodu XAML, natomiast lewe okno jest dynamicznym podglądem tworzonej formatki.

Warstwa zachowania opisana jest za pomocą języka C# i znajduje się w plikach o rozszerzeniu *.cs*. Dla każdej formatki (ekranu) tworzona jest oddzielna klasa języka C#.

1.4.1. Charakterystyczne pliki projektu

Po utworzeniu projektu tworzony jest również plik *App.xaml* (Przykład 1.1) oraz *App.xaml.cs*. *App.xaml* definiuje m.in. zasoby całej aplikacji, natomiast w *App.xaml.cs* stworzone są, możliwe do modyfikacji przez programistę, szkielety inicjalizacji aplikacji i zdarzeń takich jak jej uruchamianie oraz zamykanie.

Przykład 1.1. Plik App.xaml

```
<Application
x:Class="PhoneApp2.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:phone=
   "clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
xmlns:shell=
    "clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone">

    <!--Application Resources-->
    <Application.Resources>
    </Application.Resources>

    <Application.ApplicationLifetimeObjects>
        <!--Required object that handles lifetime events for
            the application-->
        <shell:PhoneApplicationService
            Launching="Application_Launching"
            Closing="Application_Closing"
            Activated="Application_Activated"
            Deactivated="Application_Deactivated"/>
    </Application.ApplicationLifetimeObjects>

</Application>
```

Jak zaprezentowano na przykładzie 1.1, plik *App.xaml* jest plikiem opisanym w języku XAML, w którym główny znacznik *Application* zawiera w sobie podznaczniki takie jak *Application.Resources*, definiujące zasoby dla całej aplikacji, oraz *Application.ApplicationLifeTimeObjects*, w którym znajduje się odwołanie do obiektu *PhoneApplicationService* z przestrzeni nazw *Microsoft.Phone.Shell*, który z kolei definiuje cztery zdarzenia będące podstawowymi zdarzeniami całej aplikacji. W przykładzie 1.2 znajduje się automatycznie generowany kod w pliku *App.xaml.cs*.

Przykład 1.2. Kod w pliku App.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
```

```
using System.Windows.Media.Animation;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Shell;

namespace PhoneApp2
{
    public partial class App : Application
    {
        /// <summary>
        /// Provides easy access to the root frame of the Phone
        /// Application.
        /// </summary>
        /// <returns>The root frame of the Phone Application.
        /// </returns>
        public PhoneApplicationFrame RootFrame { get; private set; }

        /// <summary>
        /// Constructor for the Application object.
        /// </summary>
        public App()
        {
            // Global handler for uncaught exceptions.
            UnhandledException += Application_UnhandledException;

            // Standard Silverlight initialization
            InitializeComponent();

            // Phone-specific initialization
            InitializePhoneApplication();

            // Show graphics profiling information while debugging.
            if (System.Diagnostics.Debugger.IsAttached)
            {
    // Display the current frame rate counters.
    Application.Current.Host.Settings.EnableFrameRateCounter = true;
    // Show the areas of the app that are being redrawn in each
    // frame.

    //Application.Current.Host.Settings.EnableRedrawRegions = true;

    // Enable non-production analysis visualization mode,
    // which shows areas of a page that are handed off to GPU
    // with a colored overlay.
   //Application.Current.Host.Settings.EnableCacheVisualization
   //                                                  =true;
```

```
        // Disable the application idle detection by setting the
        // UserIdleDetectionMode property of the
        // application's PhoneApplicationService object to Disabled.
        // Caution:- Use this under debug mode only. Application that
        // disables user idle detection will continue to run
        // and consume battery power when the user is not using
        // the phone.

        PhoneApplicationService.Current.UserIdleDetectionMode =
                                    IdleDetectionMode.Disabled;
        }

    }

        // Code to execute when the application is launching
        // (eg, from Start) This code will not execute when the
        // application is reactivated
        private void Application_Launching(object sender,
                                    LaunchingEventArgs e)
            {
            }

        // Code to execute when the application is activated
        // (brought to foreground)
        // This code will not execute when the application is first
        // launched
        private void Application_Activated(object sender,
                                           ActivatedEventArgs e)
            {
            }

        // Code to execute when the application is deactivated
        // (sent to background)
        // This code will not execute when the application is
        // closing
        private void Application_Deactivated(object sender,
                                             DeactivatedEventArgs e)
            {
            }

        // Code to execute when the application is closing
        // (eg, user hit Back)
        // This code will not execute when the application is
        // deactivated
        private void Application_Closing(object sender,
                                         ClosingEventArgs e)
            {
            }
```

```
// Code to execute if a navigation fails
private void RootFrame_NavigationFailed(object sender,
                                    NavigationFailedEventArgs e)
    {
        if (System.Diagnostics.Debugger.IsAttached)
        {
            // A navigation has failed; break into the debugger
            System.Diagnostics.Debugger.Break();
        }
    }

// Code to execute on Unhandled Exceptions
private void Application_UnhandledException(object sender,
                      ApplicationUnhandledExceptionEventArgs e)
    {
        if (System.Diagnostics.Debugger.IsAttached)
        {
            // An unhandled exception has occurred;
            // break into the debugger
            System.Diagnostics.Debugger.Break();
        }
    }

    #region Phone application initialization

    // Avoid double-initialization
    private bool phoneApplicationInitialized = false;

    // Do not add any additional code to this method
    private void InitializePhoneApplication()
    {
        if (phoneApplicationInitialized)
            return;

        // Create the frame but don't set it as RootVisual yet;
        // this allows the splash
        // screen to remain active until the application is
        // ready to render.
        RootFrame = new PhoneApplicationFrame();
        RootFrame.Navigated +=
                    CompleteInitializePhoneApplication;

        // Handle navigation failures
        RootFrame.NavigationFailed +=
                            RootFrame_NavigationFailed;

        // Ensure we don't initialize again
```

```
            phoneApplicationInitialized = true;
        }

        // Do not add any additional code to this method
        private void CompleteInitializePhoneApplication(object
                                        sender, NavigationEventArgs e)
        {
         // Set the root visual to allow the application to render
            if (RootVisual != RootFrame)
                RootVisual = RootFrame;

            // Remove this handler since it is no longer needed
            RootFrame.Navigated -=
                          CompleteInitializePhoneApplication;
        }

        #endregion
    }
}
```

Z najważniejszych definiowanych tam rzeczy należy wymienić odwołanie do *RootFrame*, głównej „ramki" aplikacji, wewnątrz której będą wyświetlane wszystkie strony (formatki) aplikacji. W konstruktorze *App()* widać dodawanie obsługi zdarzenia *UnhandledException* oraz wyświetlanie informacji wydajnościowych, gdy system operacyjny wykryje, że aplikacja uruchamiana jest pod debuggerem, które można odkomentować lub usunąć wedle uznania programisty.

Możliwe jest również dodanie własnego kodu do metod obsługi zdarzeń takich jak *Launching*, *Activated*, *Deactivated* i *Closing*.

Klasa *MainPage*, pierwsza strona (formatka) aplikacji, tak jak wszystkie inne strony, dziedziczy po klasie *PhoneApplicationPage*.

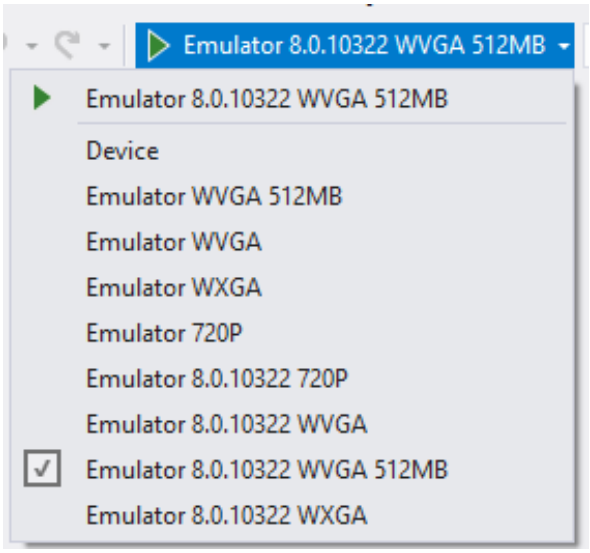
W oknie **Solution Explorer** możliwe jest również podejrzenie pozostałych plików projektu – standardowo tworzone są dwa pliki *AppManifest.xml* oraz *WMAppManifest.xml*, które są tzw. manifestami zawierającymi meta dane opisujące aplikacje. Plik *AssemblyInfo.cs* definiuje meta dane dotyczące budowanego w ramach projektu „assembly". Domyślny szablon projektu generuje również katalog *Assets* zawierający zasoby takie jak domyślna ikona aplikacji oraz plik zasobów *AppResources.resx*. Tworzona jest również klasa *LocalizedStrings.cs* oraz dodawane do niej odwołania. Klasa ta wykorzystywana jest do tworzenia aplikacji wielojęzycznych.

1.5. Uruchamianie i testowanie

Rysunek 1.7 przedstawia listę rozwijaną obok przycisku pozwalającego na uruchomienie aplikacji w trybie debugowania. Lista ta pozwala na wybranie domyślnego celu, na którym aplikacja zostanie uruchomiona.

SDK instaluje tutaj zestaw emulatorów:

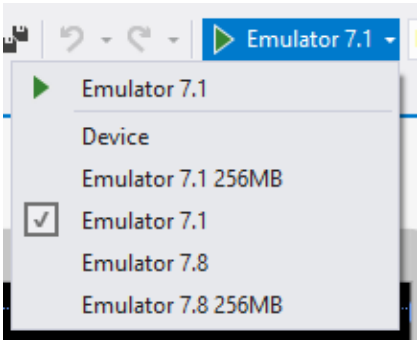
- emulator o rozdzielczości WVGA (800x480) i 512 MB pamięci RAM;
- emulator o rozdzielczości WVGA (800x480) i 1 GB pamięci RAM;
- emulator o rozdzielczości WXGA (1280x768) i 1 GB pamięci RAM;
- emulator o rozdzielczości 720P (1280x720) i 1 GB pamięci RAM;
- możliwość uruchomienia i testowania aplikacji na rzeczywistym urządzeniu podłączonym do komputera (Device);
- te same emulatory, ale w wersji systemu 8.0.10322 (GDR2) (po zainstalowaniu aktualizacji pakietu SDK).



Rys 1.7. Lista emulatorów dla projektu w wersji 8.0

W przypadku, gdy projekt jest budowany dla platformy Windows Phone 7.1, dostępne są opcje uruchomienia jak na rysunku 1.8:

- urządzenie fizyczne;
- emulator z systemem w wersji 7.1 (właściwie 7.5) i 256 MB RAM;
- emulator z systemem w wersji 7.5 i 512 MB RAM;
- emulator z systemem w wersji 7.8 i 256 MB RAM;
- emulator z systemem w wersji 7.8 i 512 MB RAM.



Rys 1.8. Lista emulatorów dla projektu w wersji 7.1.

Emulatory dla platformy w wersji 7.0 i 8.0 różnią się jedynie nieznacznie wyglądem i technicznymi sposobami działania oraz oczywiście zainstalowaną na nich wersją systemu (odpowiednio 7.5 lub 7.8 i 8.0). Ważną jednak różnicą jest fakt, że w przypadku emulatora systemu w wersji 8.0 zainstalowany tam system nie jest w żaden sposób „okrojony" i udostępnia wszystkie funkcje dostępne w zwykłym telefonie, podczas gdy emulowany system 7.x zawiera znacznie ograniczony zbiór aplikacji i ustawień.

Na rysunkach 1.9 oraz 1.10 zaprezentowano startowy ekran telefonu na emulatorach w wersji 7.8 oraz 8.0, wraz z otoczką standardowego emulatora, tj. dostępnymi emulacjami fizycznych klawiszy.

Rys 1.9. Emulator platformy Windows Phone 7.8

Rys. 1.10. Emulator platformy Windows Phone 8

Aby można było w efektywny sposób korzystać z emulatora i dostępnych w nim narzędzi warto poznać zestaw skrótów klawiaturowych, które opisane są w tabeli 1.1.

Tabela 1.1. Skróty klawiaturowe dostępne w emulatorze

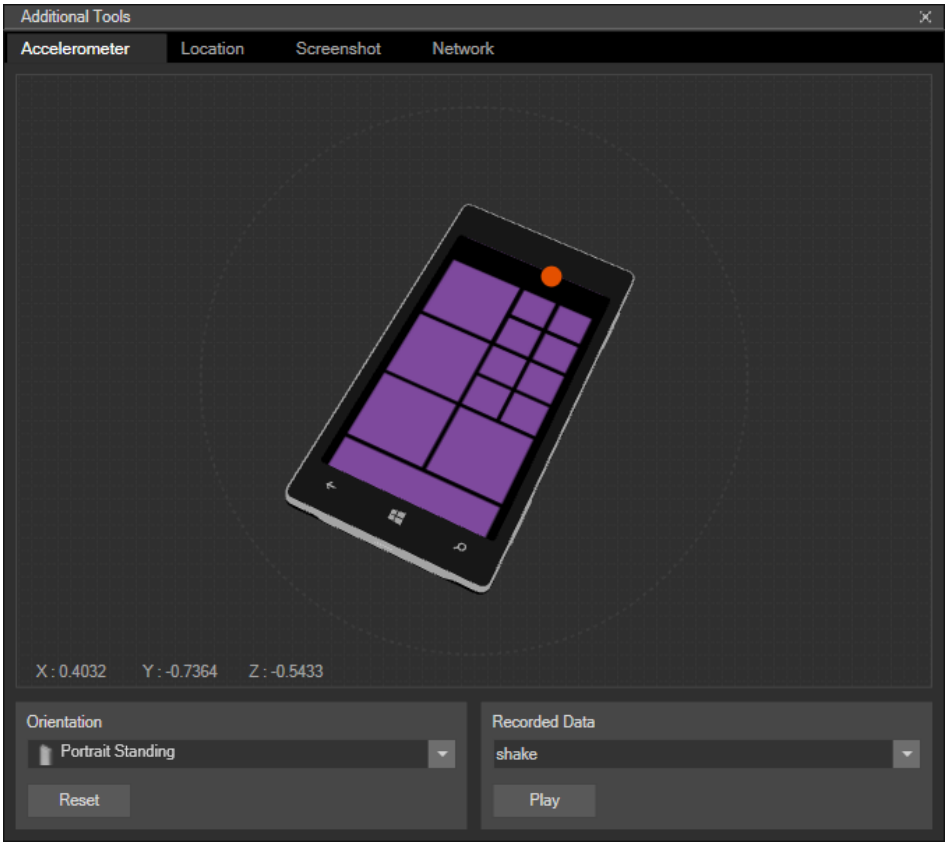
Skrót	Opis
F1	Sprzętowy klawisz „Wstecz”
F2	Sprzętowy klawisz „Start”
F3	Sprzętowy klawisz „Szukaj”
F7	Sprzętowy klawisz spustu migawki
F9	Sprzętowy klawisz rozgłaśniania
F10	Sprzętowy klawisz ściszania
F12	Sprzętowy klawisz blokady
Page Down	Wysunięcie sprzętowej klawiatury, co pozwala na użycie klawiatury komputera do wprowadzania tekstu
Page Up	Schowanie sprzętowej klawiatury

Oprócz tego warte poznania są również dodatkowe narzędzia dostępne w emulatorze. Na rysunku 1.11 przedstawiono pasek menu emulatora. Dostępne tam opcje to kolejno: zamknięcie emulatora, minimalizacja okna, obrócenie ekranu w lewo, obrócenie ekranu w prawo, dopasowanie wielkości okna emulatora do ekranu, możliwość wybrania dowolnego skalowania okna oraz uruchomienie okna narzędzi dodatkowych.

Rys 1.11. Pasek menu emulatora

Na rysunku 1.12 przedstawiono okno dodatkowych narzędzi emulatora. Karta *Accelerometer* udostępnia użytkownikowi możliwość modyfikacji położenia wirtualnego urządzenia w przestrzeni, zmieniając odczyt emulowanego przyśpieszeniomierza (więcej o obsłudze czujników zawarto w rozdziale 3). Karta *Location* pozwala na emulowanie pozycji lub ruchu w emulowanym odbiorniku GPS. Karta *Screenshot* pozwala na wykonywanie zrzutów ekranu emulatora. Karta *Network* podaje informacje dotyczące sieci podłączonych do emulowanego systemu.

Emulator umożliwia także testowanie m.in. ekranów dotykowych z obsługą wielu punktów dotyku, ale tutaj wymagane jest posiadanie monitora z obsługą wielu punktów dotyku, które są przekazywane do wirtualnego urządzenia. Emulator w wersji 8.0 udostępnia również narzędzie *Simulation Dashboard* pozwalające na testowanie różnych rodzajów połączenia bezprzewodowego w różnej jakości i szybkości połączenia.

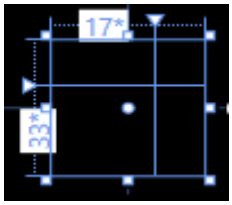
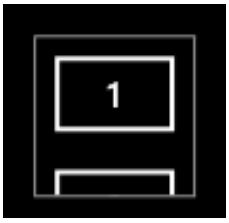


Rys 1.12. Dodatkowe narzędzia emulatora

1.6. Kontrolki i język XAML

Podczas tworzenia aplikacji z wykorzystaniem języka XAML, kontrolki na formatce można dodawać za pomocą „przeciągnij i upuść”, przeciągając je z paska widocznego z boku ekranu Visual Studio (*Toolbox*). Wśród najpopularniejszych kontrolek można wyróżnić takie jak *Button*, *TextBox*, *TextBlock* oraz inne, opisane w tabeli 1.2.

Tabela 1.2. Podstawowe kontrolki interfejsu użytkownika

Nazwa	Wygląd	Opis
Button		Kontrolka *Button* to przycisk posiadający zdarzenie kliknięcia na niego
Grid		Kontrolka *Grid* to kontener udostępniający niewidoczne dla użytkownika, jedynie dla programisty, wiersze i kolumny wewnątrz których można rozmieszczać inne kontrolki
StackPanel		Kontrolka *StackPanel* pozwala umieszczać wewnątrz niej inne kontrolki, które zostaną automatycznie dopasowane jedna pod drugą.
TextBox		Kontrolka *TextBox* reprezentuje pole, do którego użytkownik może wpisać tekst. Automatycznie po uzyskaniu skupienia pojawia się klawiatura ekranowa
TextBlock		*TextBlock* jest statyczną kontrolką wyświetlającą tekst zdefiniowany jako jego właściwość *Text*.

Przykładowy kod XAML strony (formatki) aplikacji prezentuje przykład 1.3. Standardowo tworzona formatka zawiera już pewne elementy, takie jak kontrolkę prezentującą nazwę aplikacji i duży napis informacyjny z tytułem strony oraz podstawowy kontener, kontrolkę *Grid*, według której pozycjonowane są pozostałe kontrolki na formatce.

Przykład 1.3. Domyślnie tworzona strona aplikacji

```
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
xmlns:shell="clr-
namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
mc:Ignorable="d"
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
SupportedOrientations="Portrait" Orientation="Portrait"
shell:SystemTray.IsVisible="True">

<!--LayoutRoot is the root grid where all page content is placed-->
<Grid x:Name="LayoutRoot" Background="Transparent">
       <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <!-- LOCALIZATION NOTE:
         To localize the displayed strings copy their values to
         appropriately named
         keys in the app's neutral language resource file
         (AppResources.resx) then replace the hard-coded
         text value between the attributes' quotation marks
         with the binding clause whose path points to that string
         name.
         For example:
         Text="{Binding Path=LocalizedResources.ApplicationTitle,
                Source={StaticResource LocalizedStrings}}"
         This binding points to the template's string resource named
         "ApplicationTitle".
```

```
        Adding supported languages in the Project Properties tab
        will create a new resx file per language that can carry
        the translated values of your
        UI strings. The binding in these examples will cause the
        value of the attributes to be drawn from the .resx file
        that matches the CurrentUICulture of the app at run time.
       -->

      <!--TitlePanel contains the name of the application and page
      title-->
      <StackPanel x:Name="TitlePanel" Grid.Row="0"
              Margin="12,17,0,28">
          <TextBlock Text="MY APPLICATION" Style="{StaticResource
              PhoneTextNormalStyle}" Margin="12,0"/>
          <TextBlock Text="page name" Margin="9,-7,0,0"
              Style="{StaticResource PhoneTextTitle1Style}"/>
      </StackPanel>

      <!--ContentPanel - place additional content here-->
      <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
      </Grid>

      <!--Uncomment to see an alignment grid to help ensure your
          controls are aligned on common boundaries.  The image
          has a top margin of -32px to account for the System
          Tray. Set this to 0 (or remove the margin altogether)
          if the System Tray is hidden.
          Before shipping remove this XAML and the image itself.
       -->
      <!--<Image Source="/Assets/AlignmentGrid.png"
       VerticalAlignment="Top" Height="800" Width="480"
           Margin="0,-32,0,0" Grid.Row="0" Grid.RowSpan="2"
           IsHitTestVisible="False" />-->
  </Grid>

</phone:PhoneApplicationPage>
```

Jak pokazano na przykładzie 1.3 domyślnie wszystkie kontrolki dodawane przez programistę zawierają się wewnątrz automatycznie tworzonego elementu *Grid* o nazwie *ContentPanel*. Kontrolki mogą zawierać się jedna w drugiej – nie tylko w przypadku kontenerów takich jak *Grid* albo *StackPanel*, ale możliwe jest to również w przypadku pozostałych elementów. Daje to rozbudowane możliwości prezentacji treści. Dodatkowo, język XAML jest oparty na XML, z pewnymi dodatkami w postaci specjalnych wyrażeń opisanych w klamrach, o których będzie mowa w następnych rozdziałach. Automatycznie dodawane są również komentarze zawierające przykładowe instrukcje implementacji mecha-

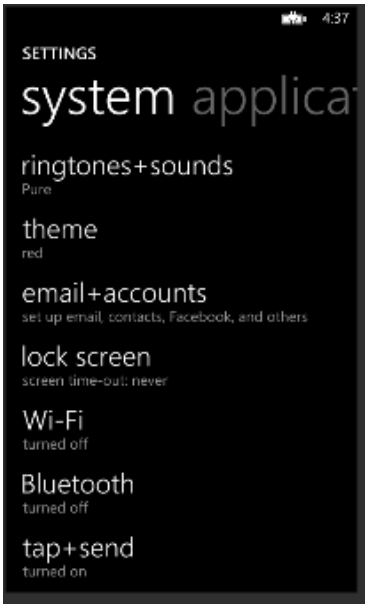
nizmów lokalizacji aplikacji wielojęzycznych oraz pokazania tzw. *aligment grid*, czyli siatki pozwalającej na lepsze wyrównanie kontrolek względem siebie.

Kontrolki mogą mieć również określone style – napis będący tytułem formatki/strony jest duży, podczas gdy ten określający tytuł aplikacji – mały, ale dalej jest to pojedynczy rodzaj kontrolki. Oprócz możliwości ustawiania rozmiarów i parametrów danej kontrolki istnieje możliwość grupowania odpowiednich wartości. Jest to realizowane za pomocą określonych stylów. Wykorzystanie stylów daje możliwość uniezależnienia aplikacji od rozdzielczości ekranu oraz wybranych przez użytkownika kolorów tła oraz tzw. koloru akcentu i będzie powodować, że aplikacja zacznie się dopasowywać do schematu kolorów użytkownika. W przykładzie 1.3 jest to zrealizowane poprzez odwołanie się do statycznych zasobów. Właściwość *Foreground* dla całej strony aplikacji ma ustaloną wartość na *{StaticResource PhoneForgroundBrush}* co powoduje to, że w zależności od ustawionego stylu w telefonie użytkownika zostanie nadany automatycznie odpowiedni kolor tła strony. Podobnie *PhoneTextNormalStyle* oraz *PhoneTextTitle1Style* dla odpowiednich kontrolek *TextBlock* definiują nie tylko wielkość fontu, którym prezentowane są te napisy, ale także odpowiedni kolor, najczęściej będący przeciwieństwem koloru tła. Więcej szczegółów związanych z doborem kolorów znajduje się w rozdziale 6.

Dodatkowymi, bardzo ważnymi kontrolkami, o których nie wspomniano wcześniej, są kontrolki *Panorama* oraz *Pivot*. Pozwalają one stworzyć aplikację i ustalić jej wygląd. Aplikacje w stylu *Panoramy* są kilkuekranowymi zbiorami kontrolek i informacji, a użytkownik ma dostęp tylko do konkretnego wycinka ekranu w danym momencie, co pokazuje rysunek 1.13. Aplikacje te reprezentują jedną z podstawowych cech interfejsu systemu Windows Phone, kiedy to użytkownik widzi jedynie skrawek informacji z ekranu obok i może przeciągnąć ekran w lewo, aby ujrzeć całość.

Kontrolka *Pivot* to rozbudowany zestaw kolumn, z których aktualnie widoczna jest tylko jedna, a do przenoszenia się pomiędzy pozostałymi służy gest przeciągnięcia w lewo lub dotknięcia odpowiedniego nagłówka kolumny. Przykładowo może to służyć do filtrowania danych, jak demonstruje aplikacja przedstawiona na rysunku 1.14.

Rys 1.13. Kontrolka Panorama



Rys 1.14. Aplikacja ustawienia zbudowana w konwencji Pivot

1.7. Zarządzanie orientacją strony

Domyślnie programy napisane w Silverlight mają pionową orientację ekranu (*portrait*), natomiast programy XNA – poziomą (*landscape*). Zmiana orientacji ekranu obsługiwanej przez stronę polega na zmianie wartości właściwości *SupportedOrientation* elementu *PhoneApplicationPage*. Właściwość ta przyjmuje trzy możliwe wartości: *Portrait*, *Landscape* lub *PortraitOrLandscape*. Ta ostatnia powoduje, że telefon będzie wyświetlał ciągle tę samą stronę, odpowiednio ją skalując. W przeciwnym wypadku telefon będzie statycznie wyświetlał tylko stronę w odpowiedniej orientacji, nie uzależniając jej od fizycznego położenia urządzenia. Przykład 1.4 pokazuje dokładniej fragment kodu gdzie można ustawić obsługiwane przez stronę orientacje ekranu.

Przykład 1.4. Kod odpowiedzialny za obsługę orientacji ekranu

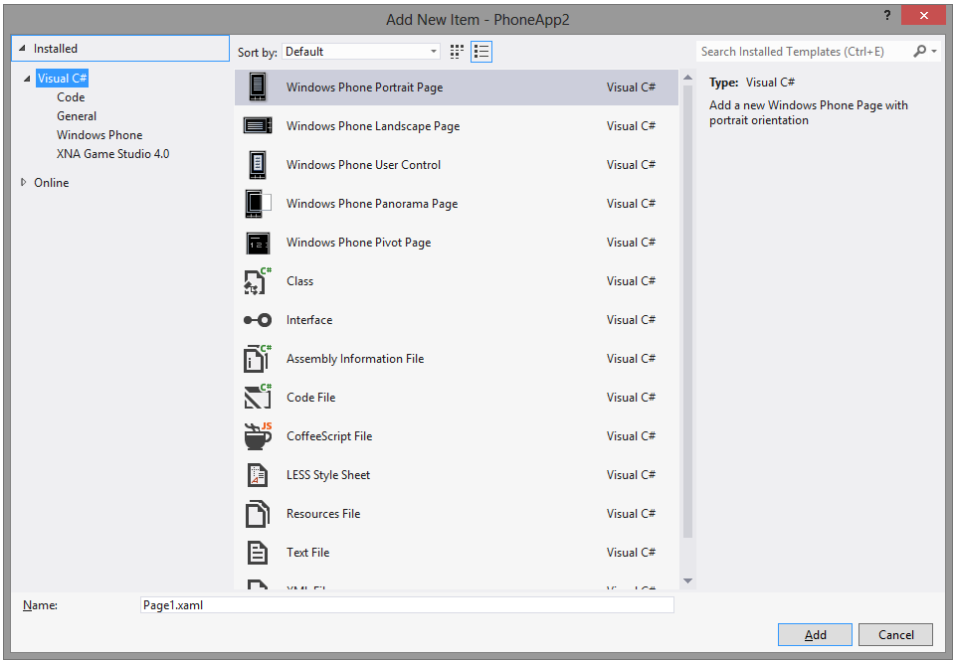
```
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.MainPage"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    (…)
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">
```

Na przykładzie 1.3 widać również dwie kolejne istotne właściwości. Właściwość *Orientation*, przyjmująca wartości *Portrait* oraz *Landscape* pozwala na ustawienie orientacji strony widocznej w wizualnym edytorze XAML. Właściwość *SystemTray.IsVisible* z przestrzeni nazw *Shell* definiuje, czy widoczny jest pasek systemowy z wyświetlaniem godziny, poziomu baterii itp. Warto tutaj zauważyć, że ikonki na tym pasku standardowo są schowane, dopóki użytkownik nie dotknie palcem miejsca paska, a widoczne są tylko informacje o aktualnej godzinie, poziomie baterii podczas ładowania lub przy oszczędzaniu baterii oraz przy braku zasięgu sieci GSM.

1.8. Przechodzenie pomiędzy stronami

Aby dodać nową stronę do projektu należy prawym klawiszem myszy kliknąć na projekt w oknie *Solution Explorer*, wybrać *Add -> New item* (skrót klawiaturowy *Ctrl+Shift+A*) i w kreatorze wybrać dodanie strony w orientacji *Portrait* albo *Landscape* oraz stron z automatycznie dodanymi kontrolkami *Panorama* albo *Pivot*. Kreator przedstawiony jest na rysunku 1.15.

Do przenoszenia się pomiędzy stronami należy wykorzystać statyczną metodę *Navigate()* statycznej klasy *NavigationService*, przyjmującej jako parametr adres strony, do której ma zostać przeniesiona kontrola (faktycznie chodzi o adres strony, która zostanie wyświetlona w *RootFrame*). Przykład 1.5. przedstawia przeniesienie kontroli do strony *Page2.xaml*.



Rys. 1.15. Kreator dodawania nowego elementu projektu

Przykład 1.5. Użycie Navigate()

```
NavigationService.Navigate(new Uri("/Page2.xaml",
                                   UriKind.Relative));
```

Tworzenie obiektu klasy *Uri*, którego oczekuje metoda *Navigate* wymaga podania adresu strony oraz rodzaju identyfikatora *Uri*, który w przypadku stron będących częścią tej samej aplikacji jest względny (dlatego też jest symbol '/' na początku jego adresu).

Aby powrócić do poprzedniej strony wystarczy użyć sprzętowego klawisza *Wstecz*. Możliwe jest zmodyfikowanie tego zachowania poprzez obsłużenie zdarzenia *BackKeyPress* dla całej strony. W przykładzie 1.6. zaprezentowano dodatkowe pytanie użytkownika, czy powrót jest zamierzony – jeśli wynik funkcji *MessageBox.Show()* nie jest równy naciśnięciu klawisza *OK*, to zdarzenie jest anulowane poprzez ustawienie wartości *Cancel* na *true*.

Przykład 1.6. Obsługa zdarzenia BackKeyPress

```
pivate void PhoneApplicationPage_BackKeyPress(object sender,
                        System.ComponentModel.CancelEventArgs e)
{
  if (MessageBox.Show("Czy jesteś pewny, że chcesz wrócić?",
     "Aplikacja", MessageBoxButton.OKCancel) != MessageBoxResult.OK)
    e.Cancel = true;
}
```

Warto w tym miejscu wspomnieć o zachowaniu dotyczącym przechodzenia pomiędzy stronami. W sytuacji, kiedy użytkownik przechodzi na drugą stronę, jest ona od nowa tworzona, a kiedy z niej wraca – niszczona. Główna (pierwsza) strona aplikacji nie jest niszczona aż do czasu zakończenia aplikacji.

1.9. Przekazywanie danych pomiędzy stronami

Każda strona programu jest odrębna i inna od każdej pozostałej strony. Dane (pola, metody, zmienne) zadeklarowane na jednej stronie nie są dostępne na innej. Czasami jednak zachodzi konieczność przekazania danych z jednej strony do drugiej. Jeśli są to proste dane, np. liczby czy teksty, to można taki parametr przekazać do URI przy przechodzeniu do drugiej strony. Przykład 1.7 przekazuje parametr do innej strony, korzystając z metody *Navigate()* klasy *NavigationService*.

Przykład 1.7. Przekazywanie danych przez adres URI

```
private void GoWithParam(string param)
{
   NavigationService.Navigate(new Uri("/Page2.xaml?arg=" +
                              param, UriKind.Relative));
}
```

Metoda z przykładu 1.7 realizuje przejście do strony *Page2.xaml* przekazującej jej argument o wartości swojego parametru (*param*). Strona odbierająca może z niego skorzystać odczytując *QueryString* z tzw. kontekstu nawigacji (klasy *NavigationContext*). Przechodząc do nowej strony wywoływane są dwie metody – metoda *OnNavigatedFrom* strony, z której odbywa się nawigacja, oraz metoda *OnNavigatedTo* strony, na którą odbywa się przejście. Metody te pochodzą z klasy bazowej *PhoneApplicationPage* i w przypadku stron stworzonych przez użytkownika należy je nadpisać wedle własnego uznania. Przykład 1.8 demonstruje nadpisanie metody *OnNavigatedTo* i odczytanie danych z przekazanego argumentu.

Przykład 1.8. Odczyt danych podczas nawigacji z adresu

```
   protected override void OnNavigatedTo(
                  System.Windows.Navigation.NavigationEventArgs e)
{
   string arg1;
   if (NavigationContext.QueryString.TryGetValue("arg", out arg1))
         MessageBox.Show("Parametr przekazany: " + arg1);
   base.OnNavigatedTo(e);
}
```

Przykład 1.8 pokazuje, że próbujemy odczytać z *QueryString* (identycznego jak w przypadku aplikacji internetowych) argument o nazwie *arg*. Jeśli operacja się powiedzie, pobrana wartość argumentu zapisywana jest do zmiennej *arg1* oraz wyświetlany jest komunikat z wartością pobranego parametru.

1.9.1. Współdzielenie obiektów pomiędzy stronami – obiekty globalne

W celu współdzielenia obiektu pomiędzy różnymi stronami możliwe jest zdefiniowanie obiektu globalnego dla całej aplikacji. W zasadzie jest to nie tyle obiekt globalny, co obiekt będący polem lub właściwością klasy *App*, która jest wspólna dla całej aplikacji i wszystkich jej stron. Przykład 1.9 demonstruje dodanie do klasy *App* nowej zmiennej, co odbywa się w pliku *App.xaml.cs*.

Przykład 1.9. Dodanie pola do klasy App

```
public partial class App : Application
{
   // zmienna publiczna, która będzie widziana przez wszystkie
   // strony
   public string SharedData;
   //…
}
```

Po zadeklarowaniu pola *SharedData*, aby dostać się do niego z dowolnej strony aplikacji, wystarczy odwołać się do statycznego pola *Current* klasy *App* i potraktować je jako obiekt klasy *App*, jak pokazuje przykład 1.10. Można również pole z przykładu 1.9 oznaczyć jako statyczne, co spowoduje, że będzie możliwy dostęp do niego bezpośrednio ze statycznego odwołania do klasy *App*, z pominięciem rzutowania.

Przykład 1.10. Dostęp do danych z klasy App - wyświetlenie

```
MessageBox.Show((App.Current as App).SharedData);
```

1.10. Cykl życia aplikacji i zachowywanie danych

Jak wspomniano w rozdziale 1.4, cała aplikacja posiada takie zdarzenia jak *Launching*, *Activated*, *Deactivated* i *Closing*. Związane są one bezpośrednio z cyklem życia aplikacji.

Aplikacje na platformie Windows Phone, w odróżnieniu od klasycznych aplikacji desktopowych, a podobnie do aplikacji *Windows 8-style*, nie mają możliwości (poza specjalnymi przypadkami) działania w tle w Windows 8.

Aplikacje działające są w stanie aktywnym (*active*). W sytuacji, kiedy użytkownik zamyka aplikację za pomocą naciśnięcia klawisza *Wstecz*, aplikacja jest usuwana z pamięci. Kiedy jednak użytkownik przechodzi do innej aplikacji lub do ekranu startowego, aplikacja jest *zamrażana* w pamięci i nie uzyskuje żadnego cyklu procesora. Stan ten określany jest jako uśpienie (*dormant*) i w zasadzie jest kombinacją dwóch kolejnych stanów aplikacji: zapauzowania (*paused*) albo stanu *tombstoned* (którym to terminem będziemy się posługiwać w oryginale, z uwagi na brak dobrego odpowiednika w języku polskim).

Zasadniczo, tylko jedna aplikacja naraz, może być całkowicie aktywna, bo w przeciwnym razie musi być uśpiona (zapauzowana lub w stanie *tombstone)*. Możliwe jest, że aplikacja nie jest aktywna, ale nie jest uśpiona, jest to jednak przypadek graniczny, występujący w jednej, szczególnej sytuacji, przy uruchamianiu tzw. zadań lub *launcherów*, które szerzej są opisane w kolejnych rozdziałach.

Kiedy aplikacja jest w stanie zapauzowanym, a użytkownik do niej powróci, natychmiast jest wznawiana. Możliwa jednak jest sytuacja, kiedy uruchomione są w tle inne aplikacje i system w pewnym momencie stwierdzi, że tę najdawniej uruchomioną należy zamknąć. W tym momencie następuje tzw. *tombstoning*. Aplikacja jest zamykana, a jej proces zakończony, ale system udostępnia jej część mocy procesora, która może być wykorzystana na przykład na zapisanie swojego stanu pracy tak, aby można było do niej powrócić bez przerwy widocznej dla użytkownika.

Taki sposób zarządzania zasobami pozwala na najbardziej efektywne wykorzystanie energii oraz na zachowanie odpowiednio działającego i efektywnego interfejsu użytkownika. Zapauzowanie aplikacji i natychmiastowe przekazanie jej mocy procesora do innych zadań, na przykład rozmowy przychodzącej, jest niezbędne na urządzeniu będącym telefonem. Prowadzi to jednak do problemów ze strony programisty, który musi opanować przechodzenie pomiędzy stanami aplikacji.

1.10.1. Zdarzenie zamykania i deaktywacji

Zdarzenie *Closing* zachodzi w sytuacji, kiedy aplikacja jest zamykana. Nie ma tutaj dwuznaczności – jest to sytuacja, kiedy użytkownik naciśnie przycisk *Wstecz* i aplikacja jest zamykana.

Zdarzenie *Deactivated* zachodzi w sytuacji, kiedy aplikacja jest deaktywowana. Może to się wiązać z zakończeniem procesu, ale nie musi, jeśli system ma odpowiednią ilość wolnej pamięci (zależy to od wersji platformy, ilości pamięci, trybu zasilania i kilku innych opcji). Programista nie jest w stanie rozróżnić, czy jest to stan pauzy czy *tombstoning*, przez co musi zapewnić zapisanie stanu aplikacji.

Możliwy jest jeszcze trzeci scenariusz – aplikacja jest zamykana z powodu nieobsłużonego wyjątku, ale tutaj programista nie ma już możliwości wznowienia czy przerwania tego procesu.

1.10.2. Tombstoning

Tombstoning zachodzi w sytuacji, kiedy użytkownik nawiguje z aplikacji (przechodząc na przykład do innej), proces aplikacji jest zabijany, ale zapisywany jest fragment instancji aplikacji.

Tutaj należy też zwrócić uwagę, że aplikacje firm trzecich nie mogą mieć więcej niż jedną instancję. W przypadku próby załadowania drugiej instancji konkretnej aplikacji, pierwsza z nich jest zamykana. To ograniczenie nie dotyczy niektórych aplikacji wbudowanych w system i może zostać zniesione w następnej wersji platformy, analogicznie do Windows 8.1.

1.10.3. Aktywacja i uruchamianie

Zdarzenie *Launching* zachodzi w sytuacji, kiedy aplikacja jest uruchamiana po raz pierwszy. Zdarzenie *Activated* zachodzi w sytuacji, kiedy aplikacja jest wznawiana ze stanu *tombstoning*.

1.10.4. Zachowywanie stanu aplikacji

Aby zachować stan aplikacji w sytuacji, kiedy aplikacja jest deaktywowana i zachodzi *tombstoning,* najwygodniej jest skorzystać z kolekcji *State* klasy *PhoneApplicationService*. Klasa ta udostępnia swoją instancję przez statyczną właściwość *Current* i zawiera w sobie kolekcję słownikową o nazwie *State* przechowującą dowolne obiekty. Mechanizm ten pozwala zapisać tam dowolne dane pod dowolną nazwą wybraną przez programistę.

Odczytując jednak dane z tej kolekcji należy pamiętać o ich zrzutowaniu na oryginalny typ. Błąd w rzutowaniu zaowocuje wyjątkiem. Przykład zapisu i odczytu danych w zdarzenia dezaktywacji i aktywacji aplikacji jest zaprezentowany na przykładzie 1.11.

Przykład 1.11. Wykorzystanie PhoneApplicationService.State

```
// Code to execute when the application is activated (brought to
// foreground)
// This code will not execute when the application is first launched
```

```
private void Application_Activated(object sender,
                                   ActivatedEventArgs e)
{
  // wyświetlanie i konwersja pewnych danych stanu aplikacji
  MessageBox.Show( PhoneApplicationService.Current.State["tst"] as
                   string);
}

// Code to execute when the application is deactivated (sent to
// background)
// This code will not execute when the application is closing
private void Application_Deactivated(object sender,
                                     DeactivatedEventArgs e)
{
    // zapisujemy pewne dane dotyczące stanu aplikacji
    PhoneApplicationService.Current.State["tst"] = "hello";
}
```

W przykładzie 1.11 znajdują się też standardowe komentarze objaśniające cel metod odpowiedzialnych za obsługę zdarzeń i sytuację, kiedy są uruchamiane, o czym szerzej wspomniano wcześniej.

PhoneApplicationService może być również wykorzystane do przekazywania danych pomiędzy stronami aplikacji, ponieważ jest to klasa wspólna dla aplikacji, niezależna od poszczególnych stron.

Istnieje również *PhoneApplicationPage.State* działający w identyczny sposób, ale dotyczy tylko konkretnej strony. Stan z usługi *PhoneApplicationService* jest zachowywany przez system. System również narzuca limit – pojedyncza strona nie może przechowywać w *PhoneApplicationPage* więcej niż 2 MB danych, a cała aplikacja w *PhoneApplicationServic*e nie może być większa niż 4 MB danych. W przypadku, kiedy trzeba zapisać więcej informacji, należy skorzystać z klasy *Isolated Storage* opisanej szerzej w rozdziale 5.

1.11. Zadania do samodzielnego wykonania

Zadanie 1.1.

Korzystając z graficznego edytora, umieść na formatce kontrolkę *TextBox* oraz *Button*. Następnie oprogramuj zdarzenie *Click* przycisku w taki sposób, aby wyświetlił komunikat „Witaj, <tekst z kontolki TextBox>”. Użyj metody *Show* klasy *MessageBox* do wyświetlenia okna z komunikatem. Następnie użyj kontrolki *TextBlock*, aby dodać opis do pola *TextBox*.

Zadanie 1.2.

Stwórz nową formatkę w orientacji poziomej oraz przycisk na poprzedniej formatce. Wykorzystaj metodę *Navigate* klasy *NavigationService* aby przejść do nowostworzonej formatki. Zastosowanie *Navigate* pokazuje Przykład 1.4.

Zadanie 1.3.

Na nowej formatce umieść element *StackPanel*, a następnie dodaj do niego dynamicznie kilka przycisków z wykorzystaniem pętli *for*.

Przykładowe dynamiczne tworzenie przycisków realizuje kod:

```
for (int i = 0; i < 5; i++)
{
    var b = new Button();
    b.Content = "Przycisk " + i.ToString();

    stackPanel1.Children.Add(b);

}
```

Następnie oprogramuj zdarzenie *BackKeyPress* drugiej strony, tak aby powrót wymagał potwierdzenia użytkownika.

2. Modele danych i wiązanie danych

2.1. Wstęp

W tym rozdziale poruszony zostanie temat dotyczący modeli danych oraz powiązania pomiędzy danymi i kontrolkami. Daje to możliwość rozdzielenia warstwy prezentacji (czyli wyglądu aplikacji) od warstwy danych (jakimi danymi dysponuje). Dzięki oddzieleniu warstwy prezentacji możliwe będzie również przenoszenie aplikacji Windows Phone na pozostałe platformy, takie jak Windows 8.

2.2. Modele danych

Model danych jest strukturą przechowującą dane. Najczęściej są to różnej postaci listy lub inne kontenery przechowujące proste obiekty wyposażone tylko w pola, tzw. POCO (*Plain Old CLR Object*), nie zawierające metod. Listing prostej klasy POCO prezentuje przykład 2.1. Przykład 2.2 przedstawia tworzenie listy obiektów modelu danych z dodaniem danych poprzez utworzenie obiektu wraz z nadaniem wartości jego polom.

Przykład 2.1. Obiekt modelu danych

```
public class Owoc
{
  public string Nazwa { get; set; }
  public bool IsSlodki { get; set; }
}
```

Przykład 2.2. Lista obiektów modelu danych

```
public void StworzListeObiektow()
{
  List<Owoc> owoce = new List<Owoc>();
  owoce.Add(new Owoc { Nazwa = "Jabłko", IsSlodki = true });
  owoce.Add(new Owoc { Nazwa = "Gruszka", IsSlodki = false });
}
```

Pola w modelu danych powinny być przedstawione w postaci właściwości. Najprostsze właściwości tworzone są poprzez stworzenie automatycznych metod *get* i *set*, tzw. *akcesorów*. Często są one nazywane również *getterami* i *setterami*, z użyciem odpowiednich słów kluczowych, jak to jest zaprezentowane na listin-

gu z przykładu 2.1. Pomimo, że możliwe jest wykorzystanie zwykłych pól klasy, nie jest to metoda zalecana.

Możliwe jest również tworzenie bardziej rozbudowanych *getterów* i s*etterów* oraz prywatnych *setterów*, co pozwala na bardziej dokładne ustawianie wartości wymaganych przez model danych. Przykład 2.3 demonstruje prywatny setter i getter pobierający automatycznie przekonwertowane dane.

Przykład 2.3. Rozbudowana metoda get

```
class Temperatura
{
    public int StopnieCelsjusza { get; private set; }
    public int StopnieFahrenheita
    {
        get
        {
            return this.StopnieCelsjusza * 9 / 5 + 32;
        }
    }
}
```

W przykładzie 2.3 pole *StopnieCelsjusza* może zostać zmodyfikowane tylko z wewnątrz klasy *Temperatura* (z uwagi na prywatny *setter*). Dzięki rozbudowanej formie *gettera* istnieje możliwość, że za każdorazową próbą odczytu wartości z pola *StopnieFahrenheita* wartość tego pola za każdym razem jest zwracana przez metodę *get*. Najogólniej można powiedzieć, że akcesory pozwalają na zwiększoną kontrolę przekazywanych do obiektu danych.

2.3. Mechanizm wiązania danych

Każda kontrolka dysponuje tzw. kontekstem danych *DataContext*, co pozwala na przypisanie jej pewnego modelu danych, z których będą one pobierane i przypisane do właściwości tej kontrolki. Do ustalenia wartości właściwości (jej źródła) niezbędne będą tzw. dowiązania (ang. binding) opisane wewnątrz języka XAML. Przykłady 2.4 i 2.5 demonstrują kod wiążący dane z modelu do treści kontrolek. W przykładzie 2.4 kontrolka *TextBlock* ma swoją właściwość *Text* ustawioną na dowiązanie do pola o nazwie *Nazwa* swojego obiektu *DataContext*. Dowiązania realizowane są poprzez napisany w nawiasach klamrowych napis *Binding*, a następnie nazwę pola, do którego będą dowiązane. Niestety, w obecnej wersji środowiska Visual Studio nie jest dostępny mechanizm podpowiadania nazw dla wiązania danych.

Przykład 2.4. Wiązanie danych z modelu do treści Text kontrolki

```
<TextBlock x:Name="textBlock1" HorizontalAlignment="Left"
   Margin="97,168,0,0" TextWrapping="Wrap"
   Text="{Binding Nazwa}"
   VerticalAlignment="Top"
/>
```

Przykład 2.5. Stworzenie kilku obiektów Owoc i ustalenie DataContext

```
List<Owoc> owoce = new List<Owoc>();
owoce.Add(new Owoc { Nazwa = "Jabłko", IsSlodki = true });
owoce.Add(new Owoc { Nazwa = "Gruszka", IsSlodki = false });

            textBlock1.DataContext = owoce[0];
```

Wynikowy program będzie wyglądał tak, jak na rysunku 2.1.

Rys 2.1. Uruchomiona aplikacja z przykładów 2.4 i 2.5

Ważnym elementem jest fakt, że *DataContext* jest dziedziczony, więc po ustawieniu *DataContext* dla całej strony na obiekt klasy *Owoc*, dowolna kontrolka zawarta na stronie, która wiąże się z polem pochodzącym z klasy *Owoc* uzyska odpowiednią wartość.

Składnia pola wiązania danych, *{Binding}* może zawierać przecinki, a po przecinkach zawarte są dodatkowe opcje, na przykład typ dowiązania. Możliwe są mechanizmy wiązania danych w drugą stronę (lub obie strony), tak że wartość kontrolki przypisywana jest do wartości w modelu danych dynamicznie, tuż po wykonaniu zmiany w kontrolce, bez konieczności obsługi zdarzeń. Kod takiego mechanizmu demonstruje przykład 2.6.

Przykład 2.6. Dwukierunkowe dowiązanie

```
<CheckBox
   IsChecked="{Binding IsSlodki, Mode=TwoWay}"
   Content="Słodki?"
/>
```

Możliwe jest również bezpośrednie wiązanie danych pomiędzy kontrolkami. Kod ilustrujący takie zachowanie przedstawia przykład 2.7. Demonstruje on również bardzo ważną cechę wiązania danych – ustalenie *DataContext* nie musi odbywać się wewnątrz kodu C#, ale możliwe jest także wewnątrz kodu XAML. W tym przykładzie wartość *Value* kontrolki *ProgressBar* dowiązana jest do wartości *Value* kontrolki *Slider* (o nazwie *slider)*. Działanie takiego programu przedstawia rysunek 2.2.

Przykład 2.7. Dowiązanie do innej kontrolki

```
<ProgressBar HorizontalAlignment="Left" Height="10"
       Margin="117,132,0,0" VerticalAlignment="Top" Width="100"
       DataContext="{Binding ElementName=slider}"
       Value="{Binding Value}"
/>
<Slider x:Name="slider" HorizontalAlignment="Left"
   Margin="117,202,0,0" VerticalAlignment="Top" Width="239"
   Maximum="100"
/>
```

Rys 2.2. Dowiązanie danych pomiędzy kontrolkami

2.4. Konwersje przy wiązaniu danych

Czasami przy wiązaniu danych zachodzi potrzeba wykonania automatycznej konwersji danych. Konwersja ta może mieć różną postać, od prostego przekształcenia typów (aby liczbę wyświetlać jako napis), przez bardziej zaawansowane konstrukcje. Przykład 2.8 demonstruje model danych, w którym istnieją 3 pola: pole *Col* jest obiektem typu *Color*, pole *FemaleName* oznacza pełną nazwę koloru, a pole *MaleName* określa nazwę jednego z jego bliskich odpowiedników. Przykład 2.9 wyświetla ten model danych. *SolidColorBrush* został wykorzystany do zdefiniowania wypełnienia prostokąta reprezentowanego przez kontrolkę *Rectangle*. Kontrolka *TextBlock* wyświetla nazwę *MaleName*.

Przykład 2.8. Model danych

```
public class FemaleColor
{   public Color Code { get; set; }
    public string FemaleName { get; set; }
    public string MaleName { get; set; }
}
```

Przykład 2.9. Wyświetlanie danych z modelu

```
<StackPanel
      Grid.Row="1" HorizontalAlignment="Stretch" Margin="0,0,0,0"
      Name="spResult" VerticalAlignment="Stretch" >

  <TextBlock
       Text="{Binding MaleName}"
       Style="{StaticResource PhoneTextLargeStyle}">
  </TextBlock>
  <Rectangle Height="70" Margin="12,6">
    <Rectangle.Fill>
      <SolidColorBrush Color="{Binding Code}"></SolidColorBrush>
    </Rectangle.Fill>
  </Rectangle>
</StackPanel>
```

Aby dodać do programu opcję wyświetlania kodu szesnastkowego koloru, stosowanego na przykład w tworzeniu stron WWW, można posłużyć się własnym konwerterem, który przekształci obiekt klasy *Color* na łańcuch tekstowy opisujący kolor w postaci szesnastkowej. Klasa *Color* posiada metodę *ToString()* opisującą zawarty w niej kolor nie dokładnie w takiej postaci jak w języku HTML (zawiera dodatkowe znaki, nie ma znaku # z przodu).

Własny konwerter pozwoli dowiązać dane z wykonaniem automatycznych przekształceń. *TextBlock* prezentujący dowiązanie wraz z użyciem konwertera pokazany jest na przykładzie 2.10.

Przykład 2.10. Wiązanie danych z wykorzystaniem konwertera

```
<TextBlock Text="{Binding Code,
   Converter={StaticResource MyColorConverter}}"
   Margin="6">
</TextBlock>
```

Jak przedstawia przykład 2.10, aby odnieść się do konwertera wykorzystane jest odwołanie do statycznego zasobu o nazwie *MyColorConverter*. Jest to klasa, która automatycznie dokona konwersji. Zasób ten musiał być zdefiniowany wcześniej, jak prezentuje przykład 2.11.

Przykład 2.11. Definiowanie statycznego zasobu na stronie

```
<UserControl.Resources>
    <converter:MyColorConverter
        x:Key="MyColorConverter">
    </converter:MyColorConverter>
</UserControl.Resources>
```

Znaczniki te zawarte są bezpośrednio wewnątrz strony aplikacji. Do zasobów strony dodawane jest odwołanie o nazwie *MyColorConverter* do elementu o nazwie *MyColorConverter* z przestrzeni nazw *XML converter*. Ta przestrzeń nazw została wcześniej dopisana do całej strony, jak w przykładzie 2.12.

Przykład 2.12. Zdefiniowanie przestrzeni nazw wewnątrz strony

```
<phone:PhoneApplicationPage
    x:Class="Ktos.Eudore.MainPage"
    (…)
    xmlns:converter="clr-namespace:Ktos.Eudore.Models"
    (…)
```

Sama klasa *MyColorConverter* może już teraz zostać użyta, o ile znajduje się w przestrzeni nazw języka C# *Ktos.Eudore.Models*. Dodatkowo, aby można ją było wykorzystać, musi implementować interfejs *IValueConverter*, który wymaga od programisty implementacji metody *Convert* oraz *ConvertBack*. W tym przykładzie zakładamy, że konwersja będzie zachodziła tylko w jedną stronę i z odpowiednich typów. W praktyce pisanie konwertera powinno być obwarowane sprawdzaniem różnych warunków. Przykład 2.13 pokazuje przykładową klasę konwertera, której metoda *Convert* przetwarza typ *Color* do łańcucha znaków, co automatycznie tworzy wartość szesnastkową, jednak dodatkowo z przo-

du napisu dodawany jest znak „#” oraz usuwane są pierwsze dwa znaki. Wynika to z faktu, że metoda *Color.ToString()* zwraca ciąg w postaci czterech wartości, gdzie pierwsza para znaków (jednobajtowa liczba) oznacza obecność kanału alfa, którego nie stosuje się w zapisie dla stron internetowych.

Metoda *ConvertBack* jest natomiast niezaimplementowana i rzuca wyjątek.

Przykład 2.13. Klasa konwertująca

```
public class MyColorConverter : IValueConverter
{ public object Convert(object value, Type targetType,
                  object parameter,
                  System.Globalization.CultureInfo culture)
  {
     return "#" + value.ToString().Substring(3);
  }

public object ConvertBack(object value, Type targetType,
                  object parameter,
                  System.Globalization.CultureInfo culture)
      {
          throw new NotImplementedException();
      }
}
```

Podsumowując, przestrzeń nazw zawierająca model danych oraz klasę konwertera pokazano w kodzie przykładu 2.14.

Przykład 2.14. Przestrzeń nazw dla modelu i konwertera

```
namespace Ktos.Eudore.Models
{
    public class FemaleColor
    {
        public Color Code { get; set; }
        public string FemaleName { get; set; }
        public string MaleName { get; set; }
    }

    public class MyColorConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                        object parameter,
                        System.Globalization.CultureInfo culture)
        {
            return "#" + value.ToString().Substring(3);
        }

    public object ConvertBack(object value, Type targetType,
                         object parameter,
```

```
                        System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

Przykład 2.15. Strona aplikacji

```
<phone:PhoneApplicationPage
  x:Class="Ktos.Eudore.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:phone = "clr-namespace:Microsoft.Phone.Controls;
                 assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;
                 assembly=Microsoft.Phone"
    xmlns:converter="clr-namespace:Ktos.Eudore.Models"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
                        markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="696"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True"
    Loaded="PhoneApplicationPage_Loaded"
       xmlns:toolkit="clr-namespace:Microsoft.Phone.Controls;
                      assembly=Microsoft.Phone.Controls.Toolkit">
  <UserControl.Resources>
    <converter:MyColorConverter
        x:Key="MyColorConverter"></converter:MyColorConverter>
  </UserControl.Resources>

<!--LayoutRoot is the root grid where all page content is placed-->
  <Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
  </Grid.RowDefinitions>
(…)

  <!--ContentPanel - place additional content here-->
  <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
     <Grid.RowDefinitions>
     <RowDefinition Height="110*" />
     <RowDefinition Height="203*" />
     <RowDefinition Height="314*" />
```

```
  </Grid.RowDefinitions>
  <StackPanel Grid.Row="1" HorizontalAlignment="Stretch"
       Margin="0,0,0,0" Name="spResult" VerticalAlignment="Stretch">

   <TextBlock
     Text="{Binding MaleName}"
     Style="{StaticResource PhoneTextLargeStyle}">
   </TextBlock>
   <TextBlock Margin="6"
     Text="{Binding Code,
            Converter={StaticResource MyColorConverter}}" >
   </TextBlock>
   <Rectangle Height="70" Margin="12,6">
     <Rectangle.Fill>
        <SolidColorBrush Color="{Binding Code}"></SolidColorBrush>
     </Rectangle.Fill>
   </Rectangle>
  </StackPanel>
  </Grid>
  </Grid>

</phone:PhoneApplicationPage>
```

Dla takiej aplikacji, po ustaleniu *DataContext* kontenera *spResult* na obiekt klasy *Ktos.Eudore.Models.FemaleColor*, zawarte w niej kontrolki automatycznie ustawia odpowiednie wartości, także korzystając z automatycznie zdefiniowanej konwersji.

2.5. Interfejs INotifyPropertyChanged

W prezentowanych przykładach dowiązania danych działają poprawnie, jednak istnieje jedna sytuacja, kiedy to działanie będzie błędne. Gdy obiekt posiada właściwość, natomiast kontrolka dowiązana jest do tej właściwości, to pomimo, że wyświetlanie działa po przypisaniu *DataContext*, dane nie są aktualizowane w warstwie prezentacji, gdy zostały zmodyfikowane w modelu. Aby temu zapobiec, interfejs *INotifyPropertyChanged* musi być zaimplementowany w modelach danych.

Wymaga to dodania do klasy zdarzenia typu *PropertyChangedEventHandler*, do którego będzie wysyłana informacja o zmianie właściwości. Dodatkowo, nie jest wtedy możliwe użycie automatycznie tworzonych właściwości i obiektów POCO, ale możliwe będzie ich emulacja z wykorzystaniem akcesorów.

Dla naszej przykładowej klasy *Owoc*, interfejs *INotifyPropertyChanged* może być następująco zaimplementowany: pola *Nazwa* i *IsSlodki* uzyskają swoje prywatne odpowiedniki, natomiast one same będą posiadały akcesory pobierają-

ce wartość prywatnego pola lub je ustawiające, wraz z wysłaniem informacji (czyli wywołaniem zdarzenia typu *PropertyChangedEventHandler*).

W tym momencie pola klasy, dzięki napisaniu własnych akcesorów, są tylko interfejsem metod dostępowych do prywatnych odpowiedników pól. Dzięki rozróżnianiu wielkości liter w języku C# można je nazwać tak samo, a rozróżniać prywatne od publicznych za pomocą wielkości liter, zgodnie z konwencją nazewniczą języka.

Przykład 2.16. Implementacja INotifyPropertyChanged

```
using System;
using System.Collections.Generic;
using System.ComponentModel; // w tej przestrzeni nazw znajduje się
INotifyPropertyChanged
using System.Linq;
using System.Text;

namespace PhoneApp2
{
    public class Owoc : INotifyPropertyChanged
    {
        private string nazwa; // prywatne pole klasy
        public string Nazwa // publiczna właściwość
        {
            get
            {
                return nazwa; // zwróć wartość pola prywatnego
            }

            set
            {
                nazwa = value; // ustal wartość pola prywatnego
                if (PropertyChanged != null) // jeśli zdarzeniu
                                  // przypisano obsługę, wywołaj
                    PropertyChanged(this,
                          new PropertyChangedEventArgs("Nazwa"));
            }
        }

        private bool isSlodki;
        public bool IsSlodki {
            get
            {
                return isSlodki;
            }
            set
            {
                isSlodki = value;
```

```
                if (PropertyChanged != null)
                    PropertyChanged(this,
                        new PropertyChangedEventArgs("IsSlodki"));
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
    }
}
```

2.7. Formatowanie danych w wiązaniu danych

W poprzednich przykładach zastosowano własny konwerter do wykonania zaawansowanej konwersji danych z jednego formatu do drugiego. Aby skorzystać z prostszych metod formatowania danych, przy wiązaniu danych można wykorzystać atrybut *StringFormat* wiązania. Przykładowo, jeśli w modelu znajduje się pole o nazwie *CurrentTime* typu *DateTime*, a chcemy w wiązaniu danych wypisać dane w swoim własnym formacie, można to zrobić tak jak w przykładzie 2.17.

Przykład 2.17. Atrybut StringFormat przy wiązaniu danych

```
<TextBlock
   Style="{StaticResource PhoneTextLargeStyle}"
   Text="{Binding CurrentTime,
                  StringFormat='yyyy-MM-dd HH:mm:ss.fff'}"
   HorizontalAlignment="Center">
</TextBlock>
```

W przykładzie 2.17 tekst przypisany do kontrolki *TextBlock* jest pobierany z pola o nazwie *CurrentTime* pewnego modelu danych. To pole jest wyświetlane przy użyciu formatu opisanego jako łańcuch znaków, powodując wyświetlenie danych z wykorzystaniem pełnych numerów miesięcy i dni oraz godziny wraz z milisekundami.

W podobny sposób można formatować dowolne inne liczby – atrybut *StringFormat* wiązania przyjmuje identyczne parametry formatujące jak funkcja *String.Format()*. Aby można było skorzystać ze znaków ‘*{*’ i ‘*}*’ określając elementy formatowania, niezbędne jest poprzedzenie ich znakiem ‘\’ . Bez tego znaku nawiasy klamrowe normalnie będą potraktowane jako instrukcja wiązania danych.

Wiązanie danych z formatowaniem w innej postaci przedstawiono na przykładzie 2.18. Kod z przykładu formatuje dane typu *double* przedstawiając je w postaci liczby z dwoma miejscami po przecinku oraz dodając za nimi znak stopnia.

Przykład 2.18. Formatowanie liczby

```
<TextBlock
    Text="{Binding Azimuth, StringFormat='\{0:F2\}°'}"
    HorizontalAlignment="Center">
</TextBlock>
```

2.8. Zadania do samodzielnego rozwiązania

Zadanie 2.1.

Stwórz model reprezentujący *Studenta*, zawierający pola *imię*, *nazwisko* i *numer indeksu*.

Zadanie 2.2.

Przedstaw dane *Studenta* na formatce aplikacji używając kilku kontrolek *TextBox*. Umieść je we wspólnym kontenerze, i kontenerowi nadaj *DataContext*, a kontrolkom – odpowiednie wartości *{Binding}*.

Zadanie 2.3.

Stwórz dwustronne wiązanie, aby zmieniając dane w kontrolkach zmieniały się dane w modelu.

Zadanie 2.4.

W stworzonym modelu *Student* wykorzystaj interfejs *INotifyPropertyChanged*. Przetestuj jego działanie, na przykład zmieniając model po naciśnięciu przycisku.

Zadanie 2.5.

Użyj konwertera modelu danych w taki sposób, aby reprezentować całego *Studenta* w postaci tekstu powiązanego z kontrolką *TextBlock*.

3. Lokalizacja i czujniki

3.1. Wstęp

Każde urządzenie z systemem Windows Phone musi spełniać określone przez firmę Microsoft wymogi. Jednym z nich jest konieczność posiadania przez telefon odbiornika systemu nawigacji satelitarnej GPS oraz czujnika przyspieszenia ziemskiego (przyśpieszeniomierza, zwanego również akcelerometrem), który odpowiada między innymi za możliwość zmiany orientacji ekranu po obróceniu urządzenia.

Oprócz tego, telefony mogą być wyposażone w dodatkowe czujniki, takie jak żyroskop oraz cyfrowy kompas realizowany na bazie magnetometru, jak również czujnik zbliżeniowy pozwalający na wykrycie, czy użytkownik przyłożył telefon do ucha.

API platformy pozwala na dostęp do danych pochodzących z tych czujników i ich wykorzystanie we własnych aplikacjach oraz na sprawdzanie, czy dany typ czujnika jest dostępny w konkretnym urządzeniu. W tym rozdziale opisane zostanie użycie odbiornika systemu GPS oraz akcelerometru i kompasu. Czujnik zbliżeniowy nie jest dostępny ze strony oficjalnego API i jest uaktywniany automatycznie w aplikacji systemowej „telefon".

3.2. Mechanizm geolokalizacji

Mechanizm geolokalizacji opiera się na odbiorniku systemu nawigacji satelitarnej GPS oraz na lokalizacji opartej o sieć komórkową, gdzie używana jest triangulacja położenia na podstawie znanego położenia stacji bazowych sieci (BTS lub NodeB) i ich mocy. System nawigacji GPS, a w zasadzie A-GPS (Assisted GPS), opiera się na siatce satelitów nadających czas z wewnętrznych zegarów atomowych oraz na różnicach czasów. Mechanizm A-GPS, który jest zastosowany w telefonach, polega na przesyłaniu przez sieć dodatkowych danych na temat efemeryd satelitów, co przyspiesza ustalenie pozycji urządzenia.

GPS/A-GPS są o wiele dokładniejsze od lokalizacji za pomocą sieci komórkowej, jednak czas ich uruchomienia może być o wiele dłuższy. System okresowo zapamiętuje też swoją pozycję dla szybszej lokalizacji po uruchomieniu odpowiedniej aplikacji.

Aby można było skorzystać z mechanizmów geolokalizacji telefon musi mieć włączoną odpowiednią opcję, aplikacja musi też wymagać od użytkownika potwierdzenia, że ma ona pozwolenie na wykorzystanie danych lokalizacyjnych; niezbędna jest także możliwość wyłączenia lokalizacji (patrz wymagania certy-

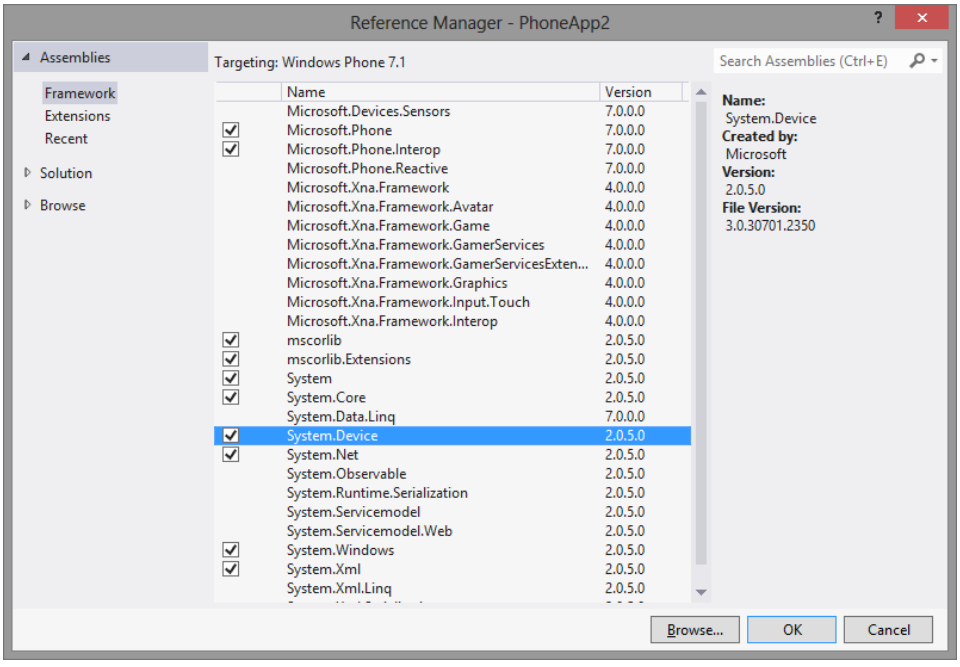
fikacji w rozdziale 6). Włączenie globalnej opcji usługi lokalizacji dostępne jest w menu *Ustawienia -> lokalizacja*.

Przykład 3.1 przedstawia fragment kodu, który pozwala na wykorzystanie mechanizmów geolokalizacji. W procedurze obsługi zdarzenia przycisku *Button* dodawany jest obiekt klasy *GeoCoordinateWatcher*.

Przykład 3.1. Użycie GeoCoordinateWatcher

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    GeoCoordinateWatcher watcher = new GeoCoordinateWatcher();
}
```

Klasa *GeoCoordinateWatcher* pochodzi z przestrzeni nazw *System.Device*. W przypadku projektów Windows Phone 8 jest ona automatycznie dodawana, ale w przypadku projektów Windows Phone 7 żeby ją wykorzystać należy najpierw dodać do projektu referencję do tej biblioteki. W tym celu należy kliknąć prawym przyciskiem myszy na sekcji *References* w oknie Solution Explorer, wybrać z menu *Add Reference* oraz zaznaczyć przestrzeń nazw *System.Device*, jak zaprezentowano na rysunku 3.1.



Rys 3.1. Dodawanie referencji do biblioteki

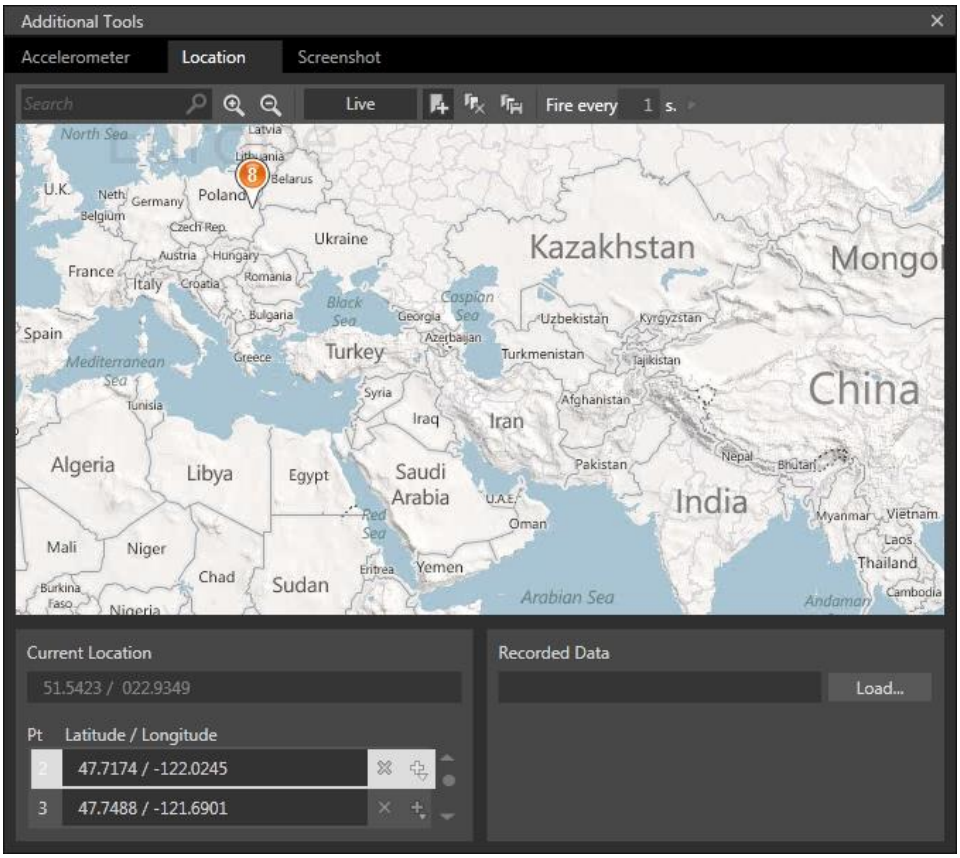
Następnie, należy dodać własną funkcję obsługi zdarzenia *PositionChanged* obiektu klasy *GeoCoordinateWatcher*. W przykładzie 3.2 zostało to zrealizowane poprzez stworzenie dynamicznie tzw. metody anonimowej, poprzez wykorzystanie wyrażenia lambda. Taka skrócona składnia pozwala na szybsze pisanie obsługi niektórych zdarzeń, kosztem czytelności kodu. W tym przykładzie do pól *textBlock1* i *textBlock2* przypisywane są wartości odczytanej szerokości i długości geograficznej.

Przykład 3.2. Obsługa PositionChanged

```
GeoCoordinateWatcher watcher = new GeoCoordinateWatch-
er(GeoPositionAccuracy.High);
watcher.PositionChanged += (s, a) =>
    {
      textBlock1.Text = a.Position.Location.Latitude.ToString();
      textBlock2.Text = a.Position.Location.Longitude.ToString();
    };
watcher.Start();
```

Warto zwrócić też uwagę na opcjonalny parametr konstruktora klasy *GeoCoordinateWatcher*, który określa dokładność – w przykładzie 3.2 została mu przekazana wartość *GeoPositionAccuracy.High*, która oznacza konieczność wykorzystania mechanizmów GPS, a nie lokalizacji za pomocą sieci telefonii komórkowej.

Ostatnia linijka przykładu uruchamia mechanizm geolokalizacyjny i zaczyna nasłuchiwanie zmian pozycji. Można to przetestować z wykorzystaniem narzędzi emulatora. Na rysunku 3.2 pokazane jest narzędzie emulatora odbiornika GPS. Klikając wybraną lokalizację na mapie automatycznie wysyłamy do aplikacji informację, że nowa pozycja użytkownika jest w danym miejscu. Zdarzenie dostaje dwa obiekty w parametrach – obiekt *sender*, czyli obiekt nadający zdarzenie, oraz obiekt o nazwie *a*, który jest typu *GeoCoordinate* i zawiera pozycję geograficzną aktualizowaną, kiedy położenie urządzenia ulegnie zmianie. Pozycja składa się z dwóch elementów – faktycznych współrzędnych geograficznych w postaci liczb typu *double* określających długość geograficzną i szerokość (jak w przykładzie) oraz m.in. wysokość i kurs. Oprócz tego w danych pozycji zawarty jest stempel czasu oznaczający datę, z której pobrana została pozycja.



Rys 3.2. Emulator odbiornika GPS

3.3. Mapy

Platforma Windows Phone 7 korzystała domyślnie z dostawcy map w postaci map firmy Microsoft pochodzących z usługi Bing. Platforma Windows Phone 8 wprowadziła mapy firmy Nokia, które zostały włączone w usłudze Bing oraz udostępniła opcję zapisu map na telefonie do nawigacji w trybie off-line. Do skorzystania z map można użyć kontrolkę *Map* pochodzącą z przestrzeni nazw *Microsoft.Phone.Maps.Controls*. Kontrolka ta umieszczona na formatce wygląda jak na rysunku 3.3.

Dopiero po faktycznym uruchomieniu kontrolka pobiera mapy z Internetu (Rys. 3.4), lub z pamięci urządzenia i pokazuje swoją zawartość, zachowując się podobnie do kontrolki *WebBrowser* opisanej w następnym rozdziale.

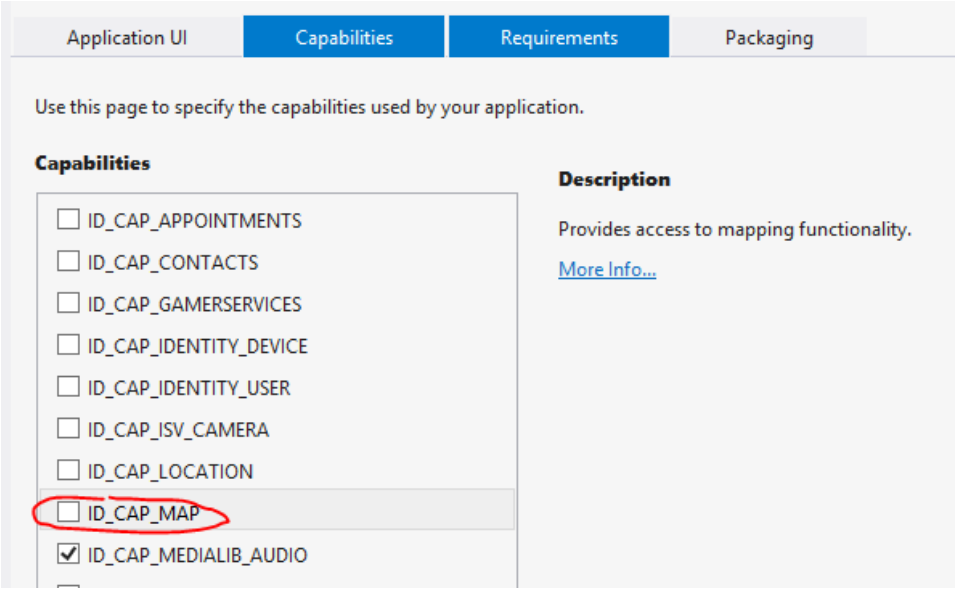
Rys 3.3. Kontrolka Map w oknie projektowania interfejsu

Rys 3.4. Kontrolka Map z załadowanymi mapami

Może zdarzyć się sytuacja, że aplikacja z kontrolką *Map* nie zostanie poprawnie uruchomiona, a zamiast tego debugger pokaże wystąpienie wyjątku *XamlParseException* razem z wcześniejszymi wystąpieniami wyjątków typu *UnathorizedAccessException*. Pojawienie się tego błędu, wynika z braku uprawnień aplikacji do obsługi map. Aby ustawić w aplikacji możliwość obsługi map, należy otworzyć plik *WMAppManifest.xml* z katalogu *Properties*, który udostępnia graficzny edytor właściwości aplikacji, a następnie w zakładce *Capabilities* zaznaczyć *ID_CAP_MAP* (Rys. 3.5). Więcej informacji o uprawnieniach aplikacji można znaleźć w rozdziale 6.

Kontrolka *Map* dysponuje szeregiem właściwości, przede wszystkim:

- właściwością *Center* oznaczającą miejsce (punkt geograficzny), na którym mapa zostanie wycentrowana;
- zdarzeniem *CenterChanged*, które pozwala wykryć sytuację, kiedy wycentrowanie mapy ulegnie zmianie przez użytkownika;
- *CartographicMode*, który określa typ mapy (zdjęcia satelitarne, drogi itp.).



Rys 3.5. Wybór ID_CAP_MAP w oknie Capabilities

3.4. Czujniki urządzenia

Poza odbiornikiem systemu nawigacji satelitarnej, telefon najczęściej wyposażony w akcelerometr. Akcelerometr, czyli przyspieszeniomierz, mierzy własny ruch (przyspieszenie w stosunku do ziemskiego) i pozwala wykryć przestrzenne położenie urządzenia, w którym jest zamontowany. Odczytem danych z akcelerometru zajmuje się klasa *Accelerometer*. *Klasa ta jest zdefiniowana w* przestrzeni nazw *Microsoft.Devices.Sensors*. W przypadku projektu dla Windows Phone 7 ponownie wymagane jest dodanie referencji do biblioteki zawierającej tę przestrzeń nazw.

Aby skorzystać z obsługi akcelerometru warto sprawdzić, czy akcelerometr jest obsługiwany na konkretnym urządzeniu. Klasa *Accelerometer* posiada statyczną metodę *IsSupported*, która zwraca *true* w sytuacji, kiedy akcelerometr jest obsługiwany na danym urządzeniu. Akcelerometr w praktyce powinien być obecny na wszystkich urządzeniach z systemem Windows Phone. Nie dotyczy to jednak innych czujników, a metoda sprawdzania jest identyczna dla pozostałych typów czujników.

Obiekt klasy *Accelerometer* posiada możliwość dodania obsługi zdarzenia *CurrentValueChanged* następującego w chwili, kiedy zaszła zmiana danych z akcelerometru. Właściwość *CurrentValue* typu *AccelerometerReading* posiada

obiekt *Acceleration*, reprezentowany jako trójwymiarowy wektor wartości odpowiadających przyspieszeniu w wymiarach *X*, *Y* oraz *Z*.

Aby stworzyć aplikację korzystającą z akcelerometru skorzystamy z podobnej metody jak w przypadku odbiornika GPS. Na formatce zostały dodane trzy pola tekstowe w kontenerze *StackPanel* wewnątrz domyślnego *ContentPanel*, co spowodowało, że ustawiły się jedno pod drugim (Przykład 3.3).

Przykład 3.3. Dodanie kontrolek TextBox

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="14,10,10,-10">
  <StackPanel>
    <TextBlock x:Name="textBox1" />
    <TextBlock x:Name="textBox2" />
    <TextBlock x:Name="textBox3" />
  </StackPanel>
</Grid>
```

W konstruktorze *MainPage*, po inicjalizacji komponentów, dodana zostanie obsługa zdarzenia (w podobny sposób, jak wcześniej przy obsłudze odbiornika GPS) przedstawiona na przykładzie 3.4.

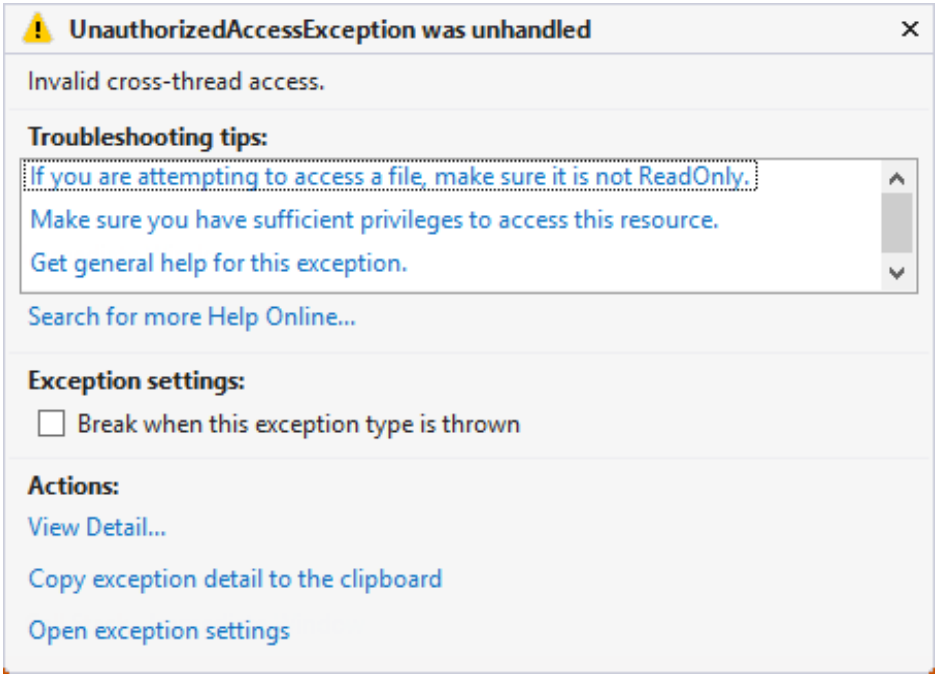
Przykład 3.4. Obsługa akcelerometru

```
public MainPage()
{
  InitializeComponent();

  if (Accelerometer.IsSupported)
  {
     Accelerometer accel = new Accelerometer();
     accel.CurrentValueChanged += (s, e) =>
       {
        textBox1.Text = e.SensorReading.Acceleration.X.ToString();
        textBox2.Text = e.SensorReading.Acceleration.Y.ToString();
        textBox3.Text = e.SensorReading.Acceleration.Z.ToString();
       };

      accel.Start();
  }
}
```

Aby ta konstrukcja mogła działać, niezbędne jest dodanie do projektu referencji do biblioteki *Microsoft.Xna.Framework* (tylko dla Windows Phone 7) oraz dodanie *Microsoft.Xna.Framework* do sekcji *using*, ponieważ *Acceleration* jest typu *Microsoft.Xna.Framework.Vector3*. Po próbie uruchomienia aplikacji pojawi się jednak wyjątek *UnauthorizedAccessException* z komunikatem *Invalid cross-thread access* jak przedstawiono na rysunku 3.8.



Rys 3.8. Błąd dostępu pomiędzy wątkami

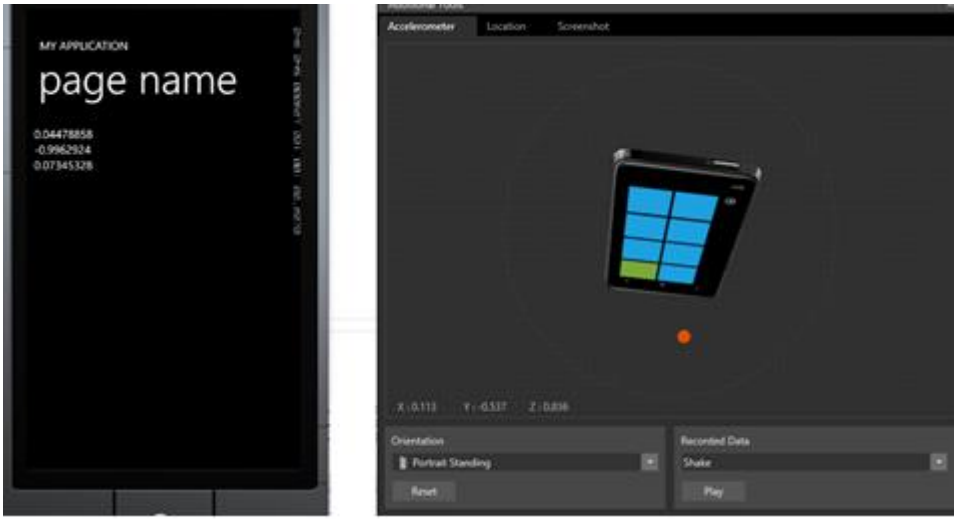
Błąd dostępu pomiędzy wątkami wynika z faktu, że obsługa akcelerometru przeprowadzana jest w innym wątku systemu niż obsługa interfejsu użytkownika, przez co niemożliwy jest bezpośredni dostęp. Z drugiej strony zapewnia to płynność działania interfejsu użytkownika podczas wykonywania złożonych operacji. Aby zapobiec pojawianiu się wyjątku można skorzystać z mechanizmu *Dispatcher*. *Dispatcher* jest klasą przypisaną do strony aplikacji, która pozwala wykonywać na niej działania poprzez „zlecenie" obiektowi klasy *Dispatcher* wykonanie aktualizacji interfejsu użytkownika. Aktualizacja ta będzie wykonana w momencie, kiedy system operacyjny zapewni mechanizmowi *Dispatcher* odpowiednią ilość czasu procesora. Aby to wykonać wystarczy wywołać z obiektu *Dispatcher* metodę *BeginInvoke()*, która oczekuje w parametrze delegata dla metody nie przyjmującej żadnych parametrów i nie zwracającej wartości, a jej wykonanie zostanie dodane do kolejki operacji dla obiektu *Dispatcher*.

Podobnie jak w przypadku korzystania z metod anonimowych do obsługi zdarzeń, możemy je wykorzystać również w tym przypadku. Nie trzeba wówczas tworzyć delegata ani żadnej dodatkowej metody. Wystarczy, że zastosujemy konstrukcję, jak na przykładzie 3.5.

Przykład 3.5. Dispatcher i metoda anonimowej do aktualizacji UI

```
public MainPage()
{
  InitializeComponent();
  if (Accelerometer.IsSupported)
  {
    Accelerometer accel = new Accelerometer();
    accel.CurrentValueChanged += (s, e) =>
      {
         this.Dispatcher.BeginInvoke(() =>
           {
              textBox1.Text =
                   e.SensorReading.Acceleration.X.ToString();
              textBox2.Text =
                   e.SensorReading.Acceleration.Y.ToString();
              textBox3.Text =
                   e.SensorReading.Acceleration.Z.ToString();
            });
       };
    accel.Start();
  }
}
```

Teraz po uruchomieniu aplikacji można wykorzystać wbudowany emulator akcelerometru aby zmieniać dane z niego pochodzące, jak przedstawiono na rysunku 3.9.



Rys 3.9. Gotowa aplikacja wraz z emulatorem akcelerometru

Jak wynika z podanych przykładów i konieczności zastosowania obiektu *Dispatcher*, system Windows Phone dysponuje mechanizmami wielowątkowości i programowania asynchronicznego. Więcej o podobnych mechanizmach czytelnik znajdzie w kolejnych rozdziałach.

3.4.1. Cyfrowy kompas

W podobny sposób jak akcelerometr, możliwe jest wykorzystanie także magnetometru, wykorzystywanego jako cyfrowy kompas w niektórych urządzeniach. Ten czujnik nie jest jednak dostępny w emulatorze, a tylko na niektórych urządzeniach fizycznych. Przykład 3.6 poniżej demonstruje sposób wykorzystania kompasu cyfrowego.

Przykład 3.6. Wykorzystanie kompasu cyfrowego

```
if (Compass.IsSupported)
{
  Compass comp = new Compass();
  comp.CurrentValueChanged += (s, e) =>
  {
     this.Dispatcher.BeginInvoke(() =>
       {
         textBox1.Text = e.SensorReading.TrueHeading.ToString();
        });
  };
  comp.Start();
}
```

Użycie cyfrowego kompasu jest prawie identyczne do wykorzystania akcelerometru. Pole *TrueHeading* wyniku w przykładzie 3.6 zapisuje do kontrolki *TextBox* wartość prawdziwego skierowania góry urządzenia (w stopniach) w stosunku do północy geograficznej. Oprócz tego możliwe jest również odczytanie *MagneticHeading*, czyli kierunku w stosunku do bieguna magnetycznego, a nie bieguna geograficznego.

Możliwy jest też odczyt trójwymiarowego wektora *MagnetometerReading*, który zawiera dane z czujnika w postaci odczytów w mikro teslach ze wszystkich trzech kierunków przestrzennych.

3.5. Aparat fotograficzny

Z aparatu fotograficznego, w który wyposażony jest każdy telefon z Windows Phone, można korzystać dwojako. Jedną metodą jest uruchomienie aparatu i dostęp do zrobionego zdjęcia przez użytkownika, do czego wykorzystuje się mechanizm zadań, opisany w rozdziale 5.

W tym rozdziale zajmiemy się bezpośrednim dostępem do danych z matrycy aparatu fotograficznego w czasie rzeczywistym. Najprostszym mechanizmem będzie wykorzystanie danych z kamery jako tła. Jeśli utworzony zostanie prostokąt (*Rectangle*) o nazwie *recCamera* to możliwe będzie skorzystanie z klasy *VideoBrush* i ustawienie tła na pewien strumień wideo (Przykład 3.7).

Przykład 3.7. Ustawienie tła prostokąta na obraz z kamery

```
/// <summary>
/// Inicjalizacja kamery - ustawienie VideoBrusha na tło pewnego
/// prostokąta
// </summary>
private void InitializeCamera()
{
  cam = new PhotoCamera();
  cam.Initialized += (s, e) =>
     {
        cam.FlashMode = FlashMode.Off;
     };

  videoBrush = new VideoBrush();
  videoBrush.Stretch = Stretch.UniformToFill;
  videoBrush.SetSource(cam);
  recCamera.Fill = videoBrush;
}
```

Metoda inicjalizującą aparat *InitializeCamera* tworzy nowy obiekt o nazwie *cam* klasy *Microsoft.Devices.PhotoCamera.* Po zainicjowaniu obiektu *cam* wyłączana jest lampa doświetlająca kamery. Następnie tworzony jest nowy pędzel wideo (*VideoBrush)*, którego źródło jest ustawiane na kamerę. Ostatnim krokiem jest ustawienie wypełnienia prostokąta *recCamera*.

Taka technika pozwala stworzyć w aplikacji możliwość widzenia świata. Poprzez nałożenie na ten wygląd dodatkowych danych możliwe jest stworzenie aplikacji typu rzeczywistości rozszerzonej.

Klasa *PhotoCamera* pozwala jednak na bezpośredni dostęp do pikseli obrazu, co może być użyteczne w tworzonych w czasie rzeczywistym filtrach obrazu. Rodzina metod *GetPreviewBuffer**:

- *GetPreviewBufferArgb(),*
- *GetPreviewBufferY(),*
- *GetPreviewBufferYCbCr()*

pozwala zapisać do pewnej tablicy bajtów dane pochodzące z aktualnego bufora obrazu. Nazwa metody określa rodzaj zapisu – *Argb* zapisuje dane jakoARGB, *YCbCr* przedstawia wszystkie informacje w postaci danych luminancji i dwóch składowych chrominancji, podczas gdy *GetPreviewBufferY()* zapisze tylko informacje dotyczące luminancji poszczególnych pikseli.

Klasa *PhotoCamera* udostępnia także możliwość pobrania pełnej klatki zdjęcia, ustawienia źródła na drugi aparat fotograficzny (jeśli urządzenie takim dysponuje) i tak dalej. Warto tutaj zwrócić uwagę, że pobieranie klatki zdjęcia, ustawianie ostrości i inne funkcje realizowane są asynchronicznie.

3.5.1. Wyliczenie średniej jasności

Do naszej aplikacji dodana zostanie teraz druga funkcja, która po stuknięciu w ekran aplikacja policzy średnią jasność pikseli widocznych na ekranie podglądu aparatu. Takie zadanie można zrealizować w dość prosty sposób.

Dla prostokąta *recCamera* niezbędne będzie dodanie obsługi zdarzenia *Tap*. Zdarzenie to reprezentuje stuknięcie palcem w kontrolkę. W procedurze obsługi tego zdarzenia będziemy tworzyć bufor, przechowujący informacje z podglądu tego, co widzi aparat. Jeśli prostokąt *recCamera* zajmuje cały ekran, którego rozdzielczość to WVGA to wystarczy nam bufor o rozmiarach 800*480*1 bajt na opisanie jasności. Przykład 3.8 prezentuje wyliczanie średniej jasności za pomocą wyrażenia lambda oraz wyświetlenie jej w kontrolce *TextBlock*.

Przykład 3.8. Pobranie klatki podglądu i operacje na niej

```
private void recCamera_Tap(object sender, GestureEventArgs e)
{
   byte[] buf = new byte[800 * 480];
   cam.GetPreviewBufferY(buf);
   textBlock1.Text = buf.Average<byte>(x => x).ToString();
}
```

3.6. Zadania do samodzielnego rozwiązania

Zadanie 3.1.

Napisz aplikację, która pobierze od użytkownika jego lokalizację i sformatuje wynik w postaci stopni, minut oraz sekund a następnie:

- dodaj do aplikacji możliwość przedstawianie wyniku w innym formacie (np. xx.yyyy stopnia);
- zmodyfikuj aplikację tak, aby korzystała z kontrolki *Pivot*. Na pierwszym elemencie kontrolki *Pivot* powinny znaleźć się informację o lokalizacji;
- dodaj do aplikacji kolejny element do kontrolki *Pivot*, informujący o kącie wychylenia telefonu w stosunku do północy magnetycznej z uwzględnieniem braku kompasu;
- przetestuj aplikację na rzeczywistym urządzeniu posiadającym kompas;

Stwórz aplikację (albo dodaj do poprzedniej) kolejny element *Pivot* przedstawiający informacje o fizycznym wychyleniu telefonu. Wykorzystaj do tego celu akcelerometr.

Zadanie 3.2.

Na podstawie przykładów aparatu fotograficznego stwórz nową aplikację. Zmodyfikuj aplikację tak, aby wyliczała maksymalną jasność pikseli z podglądu obrazu w aparacie fotograficznym.

4. Komunikacja z usługami sieciowymi

4.1. Wstęp

W obecnym świecie urządzenia mobilne komunikują się poprzez połączenie z siecią Internet z innymi usługami sieciowymi. Jest to podstawa, aby można było pobrać dodatkowe informacje do aplikacji, sprawdzić pogodę, pobrać nowe wiadomości czy synchronizować dane pomiędzy urządzeniami, aby mogły być dostępne na wielu telefonach lub na przykład na telefonie i komputerze.

W tym rozdziale opisane zostaną mechanizmy łączenia się z usługami sieciowymi Web Service, jak również z innymi usługami z wykorzystaniem klasy *WebClient*. Opisana zostanie również kontrolka *WebBrowser*, którą będzie można wykorzystać do stworzenia własnej przeglądarki internetowej.

4.2. Web Services

Koncepcja architektury zorientowanej na usługi (SOA – Service Oriented Architecture) zakłada, że logika biznesowa aplikacji nie jest monolitycznym tworem, ale może być rozbita pomiędzy wiele rozproszonych komponentów. To w szczególności dobrze pasuje do koncepcji systemów mobilnych, gdzie klient mobilny w postaci telefonu komórkowego będzie w stanie skorzystać z wielu źródeł danych.

Aby ustandaryzować sposób wymiany danych pomiędzy usługami sieciowymi pojawił się szereg technologii – CORBA, DCOM, oraz Web Services, którego składowe stanowią takie technologie jak WSDL oraz UDDI, a która opiera się w głównej mierze na języku opisu danych XML oraz na komunikacji z wykorzystaniem standardowego protokołu HTTP (lub jego szyfrowanej odmiany). To daje możliwość wykorzystania i mieszania wielu technologii do celów wymiany danych pomiędzy elementami projektu. W tym rozdziale opisane zostanie tworzenie klienta usługi Web Services korzystającej z języka C# na platformie Windows Phone. Nic nie stanowi jednak przeszkody, aby serwer usługi był napisany w języku Java i uruchomiony na systemie Linux. Uniwersalne technologie zapewniają interoperacyjność.

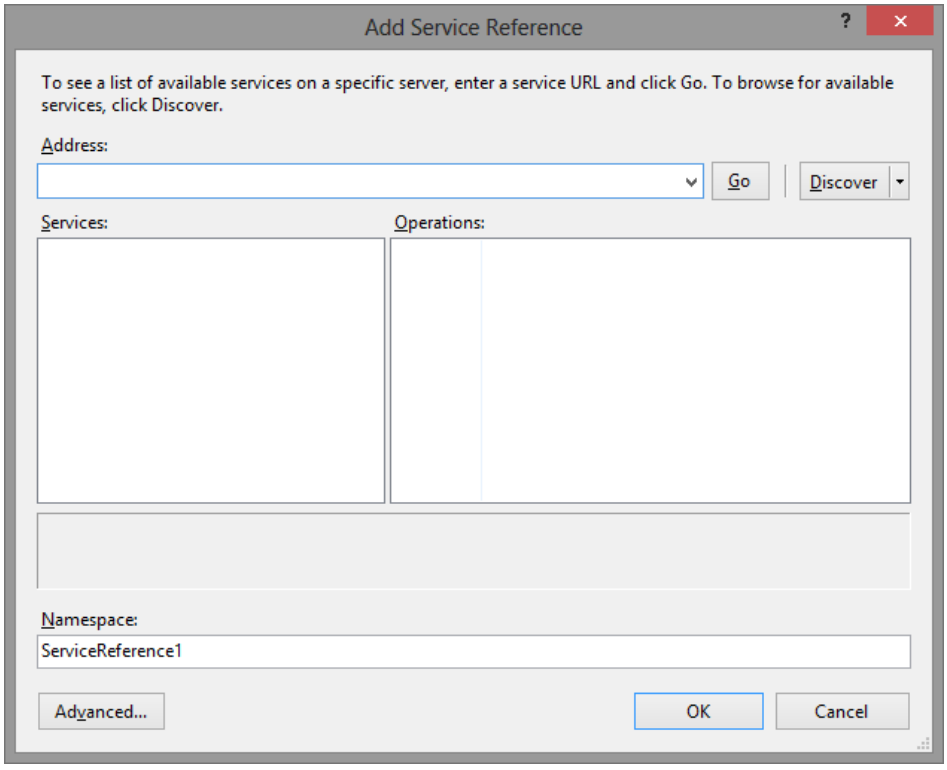
Klient usługi Web Services wysyła żądanie do serwera usług, zapisując je w postaci danych języka XML z użyciem serializacji w postaci tzw. komunikatu SOAP. Serwer usług odbiera komunikat, który najczęściej jest przesyłany za pomocą zwykłego protokołu HTTP, jak w przypadku większości obecnych usług internetowych, a następnie deserializuje go. Dalej serwer usług przekazuje informacje do odpowiedniego swojego komponentu, który odbiera żądanie i opracowuje odpowiedź, która znów jest serializowana do postaci komunikatu

SOAP i znów przesyłana do klienta. Komunikacja może być bezstanowa, jak w przypadku protokołu HTTP, lub stanowa, może istnieć możliwość wcześniejszego uwierzytelniania klienta i tak dalej. Język WSDL opisuje, w postaci znaczników, dane dotyczące usługi sieciowej. Pozwala to na automatyczną konfigurację usługi w komputerze programisty.

Dla celów tego przykładu będziemy korzystać z przykładowej usługi Web Service służącej do konwersji temperatury – TempConvert – udostępnianej przez serwis W3Schools. Dostępna ona jest pod adresem: http://www.w3schools.com/webservices/tempconvert.asmx.

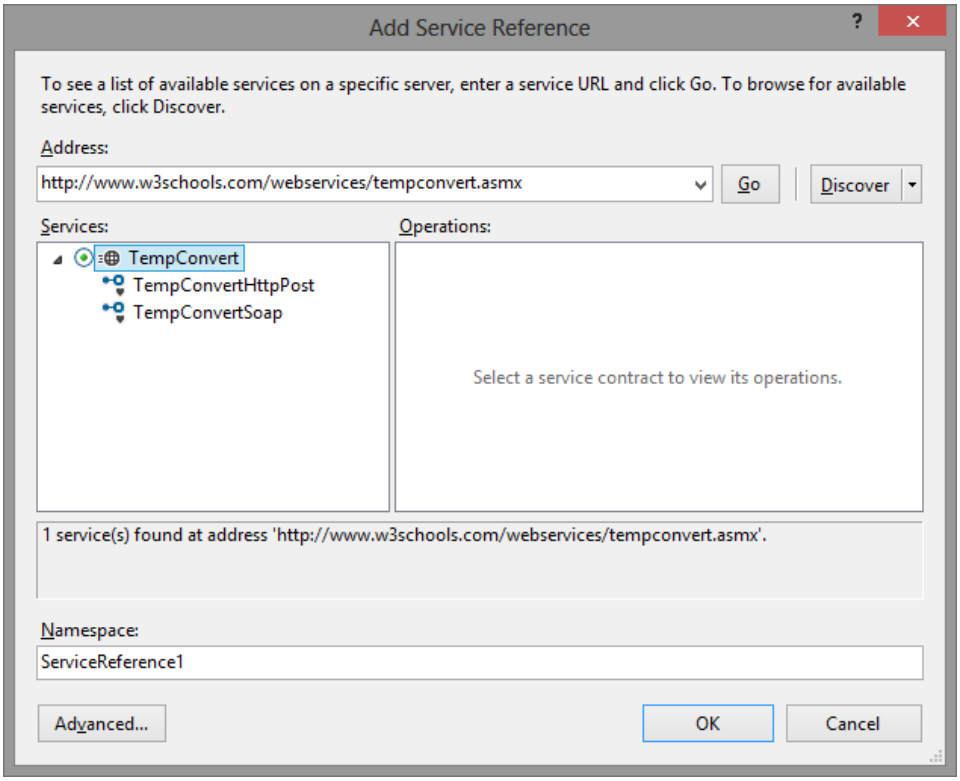
4.2.1. Dodawanie referencji do usługi

Aby dodać odniesienie do usługi sieciowej, należy w Visual Studio kliknąć prawym przyciskiem myszy okno *References* i z menu kontekstowego wybrać opcję *Add Service Reference*. Otworzy to okno dodawania usługi sieciowej, co jest przedstawione na rysunku 4.1.



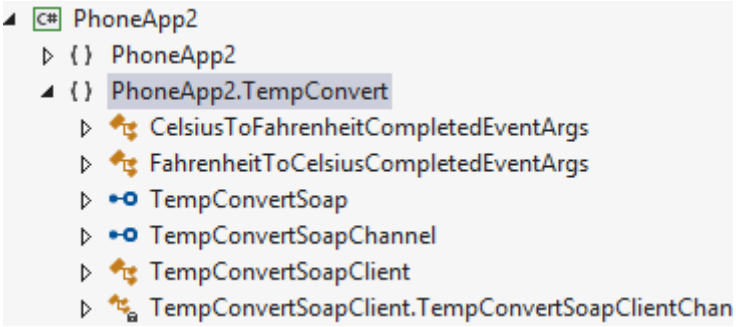
Rys 4.1. Okno dodawania usługi sieciowej

W tym oknie należy w adresie podać adres usługi sieciowej a po jego podaniu wybrać przycisk *Go*. Pod podanym wcześniej adresem znajduje się jedna usługa – *TempConvert*, jak zademonstrowano na rysunku 4.2.



Rys 4.2. Znalezione usługi sieciowe

Warto również podać swoją własną nazwę przestrzeni nazw, pod którą zostaną utworzone automatycznie klasy służące do obsługi usługi. W tym przykładzie będzie ona nazwana *TempConvert* i będzie to przestrzeń nazw zawarta wewnątrz przestrzeni nazw całej aplikacji (np. *PhoneApp2*). Po dodaniu odwołania do usługi zostaną automatycznie stworzone pewne klasy, zdarzenia oraz interfejsy, które można obejrzeć w oknie *Object Browser* (Rys. 4.3).



Rys 4.3. Automatycznie stworzone klasy, zdarzenia i interfejsy

4.2.2. Obsługa usługi sieciowej

Do tworzonej aplikacji dodany zostanie przycisk, kontrolka *TextBox* oraz kontrolka *TextBlock*. Tekst z kontrolki *TextBox* będzie oznaczał temperaturę w stopniach Celsjusza, w kontrolce *TextBlock* będzie prezentowana jej reprezentacja w stopniach Fahrenheita pobrana z usługi sieciowej, natomiast przycisk *Button* będzie realizował wysyłanie żądania. Zawartość standardowego pojemnika *ContentPanel* przedstawiono na przykładzie 4.1.

Przykład 4.1. Ustawienie kontrolek

```
<Button x:Name="bConvert" Content="Przelicz"
          HorizontalAlignment="Left" Margin="10,104,0,0"
          VerticalAlignment="Top"
/>
<TextBox x:Name="tbCelsius" HorizontalAlignment="Left" Height="72"
           TextWrapping="Wrap" VerticalAlignment="Top" Width="456"
/>
<TextBlock x:Name="tFahrenheit" HorizontalAlignment="Left"
             Margin="10,72,0,0" TextWrapping="Wrap"
             VerticalAlignment="Top"
/>
```

Następnie (Przykład 4.2), dodajemy metodę obsługującą zdarzenie kliknięcia do przycisku *Button*, którego nazwa *Name=bConvert.* Wewnątrz metody tworzymy nowy obiekt klasy *TempConvertSoapClient()* z przestrzeni *TempConvert* (względnie w stosunku do aktualnej przestrzeni nazw). Użyto tutaj operatora *var*, który automatycznie tworzy zmienną o odpowiednim typie, przez co programista może zaoszczędzić pisania długich nazw typów.

Zauważmy, że wywołania metod *FahrenheitToCelsius* i *CelsiusToFahrenheit* uzyskały przyrostek *Async* oraz dodatkowo pojawiły się zdarzenia takie jak *FahrenheitToCelsiusCompleted*. Jest to związane z praktyką programowania

asynchronicznego. Aby zapewnić m.in. płynność interfejsu użytkownika podczas wykonywania długotrwałych operacji, odwołania do sieci są realizowane w sposób asynchroniczny – wysyłane jest żądanie lub zapytanie, a system w momencie kiedy przyjdzie odpowiedź, sam uruchomi procedurę obsługi zdarzenia, czyli metodę do niego przypisaną. Z tego powodu przed wywołaniem metody należy dodać funkcję obsługi zdarzenia. Można do tego wykorzystać automatyczne mechanizmy Visual Studio (dwukrotnie nacisnąć *Tab*), co utworzy automatycznie metodę *t_CelsiusToFahrenheitCompleted*. W metodzie tej odwołujemy się do parametru *e* typu *CelsiusToFahrenheitCompletedEventArgs* i zawiera m.in. właściwość *Result* z wartością zwrotną usługi sieciowej. Przypiszemy ją do właściwości *Text* kontrolki *TextBlock*. Wracając do procedury obsługi zdarzenia kliknięcia, zostaje już tylko wywołanie metody usługi sieciowej z przekazaną jej wartością *Text* pola tekstowego. Przykład 4.2 demonstruje obydwie metody obsługujące odpowiednie zdarzenia.

Przykład 4.2. Konwersja temperatury przez usługę sieciową

```
private void bConvert_Click(object sender, RoutedEventArgs e)
{
  var t = new TempConvert.TempConvertSoapClient();
  t.CelsiusToFahrenheitCompleted += t_CelsiusToFahrenheitCompleted;
  t.CelsiusToFahrenheitAsync(tbCelsius.Text);
}

void t_CelsiusToFahrenheitCompleted(object sender,
             TempConvert.CelsiusToFahrenheitCompletedEventArgs e)
{
  tFahrenheit.Text = e.Result;
}
```

W rezultacie aplikacja działa jak zademonstrowano na rysunku 4.3.

Rys 4.3. Działanie aplikacji konwertującej temperaturę

4.2.3. Obsługa błędów

Co się jednak stanie, jeśli użytkownik poda np. litery w polu tekstowym, zamiast liczbowej wartości temperatury? Wartość zwrócona wtedy w polu *Result* to łańcuch *Error*, więc wystarczy proste porównanie łańcuchów. Niektóre usługi sieciowe będą jednak korzystać z pola *Error* w obiekcie zwróconych argumentów i wówczas niezbędne będzie ujęcie bloku w klauzulę *try..catch*.

Możliwe jest też, że operacja zostanie anulowana – przykładowo przez wywołanie metody *Abort*. Wówczas należy sprawdzić, czy wartość właściwości *Cancelled* nie jest prawdą.

4.3. Klasa WebClient

Klasa *WebClient* zawarta w przestrzeni nazw *System.Net* pozwala na wykonywanie operacji wysyłania i odbierania danych poprzez protokół HTTP. Jest ona bardziej rozbudowana i zapewnia większą wygodę dla programisty niż ręczne korzystanie z klas *HttpWebRequest* i *HttpWebResponse*. Na przykład udostępnia automatyczną metodę zapisującą zawartość zdalnego zasobu do zmiennej typu *string*.

W tym podrozdziale wykorzystamy klasę *WebClient* oraz technologię RSS do stworzenia prostego czytnika aktualności. Technologia RSS (Really Simple Syndication) oraz jej nowsza wersja, język Atom 1.0, to w rzeczywistości dane dotyczące aktualizacji na stronach internetowych zapisane w postaci dokumentów XML. Wiele popularnych serwisów internetowych udostępnia swoje kanały RSS/Atom. Jednym z nich jest serwis Gazeta.pl, udostępniający kanał RSS z aktualnymi informacjami. W chwili pisania tego tekstu kanał ten dostępny był pod adresem:
http://gazeta.pl.feedsportal.com/c/32739/f/592282/index.rss.

4.3.1. Klasa SyndicationFeed

Aby mieć wygodny dostęp do danych w postaci RSS można skorzystać z klasy *System.ServiceModel.Syndication.SyndicationFeed*. Niezbędne jest zatem dodanie referencji do biblioteki DLL – *System.ServiceModel.Syndication.dll*, zawartej w katalogu:
C:\Program Files (x86)\Microsoft SDKs\Silverlight\v4.0\Libraries\Client.

W tym celu należy w oknie dodawania referencji wybrać opcję *Browse* i wskazać plik z odpowiedniego katalogu. Ostrzeżenie dotyczące nieoczekiwanego zachowania może zostać zignorowane.

Klasa *SyndicationFeed* pozwoli na korzystanie z danych kanału RSS w postaci zbioru obiektów.

4.3.2. Wiązanie danych z kanału RSS

Aby pobrać dane z kanału RSS wykorzystamy mechanizm wiązania danych oraz szablony danych. Szablony danych pozwolą, aby dane przypisane z pewnego obiektu zostały automatycznie przedstawione jako zbiór kontrolek. W tym celu do naszego projektu dodany zostanie przycisk służący do pobierania danych oraz kontrolka *ListBox* zawierająca w sobie szablon. Szablon określa, że każdy element listy zawierać w sobie będzie kontener *StackPanel* z trzema blokami tekstu. Ustawienie kontrolek i całą zawartość *ContentPanel* zademonstrowano na przykładzie 4.3. Oprócz dowiązania danych wykorzystano też style takie jak *PhoneAccentBrush*, które wyróżniają niektóre informacje.

Przykład 4.3. Ustawienie kontrolek i szablon danych

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="0,0,12,0">
  <Button Content="Pobierz" Height="72" HorizontalAlignment="Left"
          Margin="9,6,0,0" Name="loadFeedButton"
          VerticalAlignment="Top" Width="273"
          Click="loadFeedButton_Click"
  />
  <ListBox Name="feedListBox" Height="468"
          HorizontalAlignment="Left" Margin="20,100,0,0"
          VerticalAlignment="Top" Width="444"
           ScrollViewer.VerticalScrollBarVisibility="Auto">
           <ListBox.ItemTemplate>
             <DataTemplate>
                <StackPanel VerticalAlignment="Top">
                  <TextBlock TextDecorations="Underline"
                             FontSize="24" Name="feedTitle"
                             TextWrapping="Wrap" Margin="12,0,0,0"
                             HorizontalAlignment="Left"
                             Foreground="{StaticResource
                                          PhoneAccentBrush}"
                             Text="{Binding Title.Text}"
                  />
                  <TextBlock Name="feedSummary" TextWrapping="Wrap"
                             Margin="12,0,0,0"
                             Text="{Binding Summary.Text}"
                  />
                  <TextBlock Name="feedPubDate"
                       Foreground="{StaticResource
                                    PhoneSubtleBrush}"
                       Margin="12,0,0,10"
                       Text="{Binding PublishDate.DateTime}"
                  />
                </StackPanel>
             </DataTemplate>
           </ListBox.ItemTemplate>
```

```
  </ListBox>
</Grid>
```

Przykład 4.4 przedstawia kod metody do obsługi przycisku pobierania nowych informacji. W metodzie tworzony jest obiekt klasy *WebClient* oraz dodawana obsługa zdarzenia *DownloadStringCompleted* zachodzącego w momencie, kiedy pobieranie łańcucha znaków się zakończy. W ostatniej linijce, uruchamiana jest asynchroniczna metoda pobierania danych ze wskazanego adresu URI.

Przykład 4.4. Obsługa kliknięcia przycisku „pobierz”

```
private void loadFeedButton_Click(object sender, RoutedEventArgs e)
{
  WebClient webClient = new WebClient();

  // obsługa zdarzenia zakończenia pobierania
  webClient.DownloadStringCompleted += new
              DownloadStringCompletedEventHandler(
                          webClient_DownloadStringCompleted);
  // pobieranie (asynchroniczne) zawartości pewnego adresu URI
  webClient.DownloadStringAsync(new Uri(
   "http://gazeta.pl.feedsportal.com/c/32739/f/592282/index.rss"));
}
```

Kolejny element przykładowej aplikacji to obsługa zdarzenia pobrania danych. W systemie Windows Phone 7.5 lub nowszym zdarzenia *WebClient* są uruchamiane w tym samym wątku, z którego zostały wywołane. Ponieważ w naszej aplikacji posługujemy się zdarzeniami asynchronicznymi, więc są one uruchamiane w wątku tła. Niezbędne jest znowu użycie mechanizmu *Dispatcher* omówionego wcześniej. Bez niego niemożliwa będzie współpraca z elementami interfejsu użytkownika.

Zdarzenie obsługi posiada dostęp do odebranych danych w postaci obiektu *DownloadStringCompletedEventArgs*. Klasa ta zawiera m.in. właściwość *Error*, która jeśli jest pusta (równa *null*) oznacza, że nie wystąpiły problemy z pobieraniem danych.

Jeśli nie pojawiły się błędy, tworzony jest obiekt *SyndicationFeed* korzystający z klasy *XmlReader* opartej na strumieniu pochodzącym z łańcucha zawartego we właściwości *Result* klasy wyniku operacji. Przykład 4.5 demonstruje procedurę obsługi zdarzenia.

Przykład 4.5. Obsługa zdarzenia pobrania danych z serwera

```
private void webClient_DownloadStringCompleted(object sender,
                DownloadStringCompletedEventArgs e)
{
  // jeśli nie ma błędu
```

```
  if (e.Error != null)
  {
     Dispatcher.BeginInvoke(() =>
       {
         // pokazywanie informacji o błędzie
         MessageBox.Show(e.Error.Message);
       });
  }
  else
  {
     StringReader stringReader = new StringReader(e.Result);
     XmlReader xmlReader = XmlReader.Create(stringReader);
     SyndicationFeed feed = SyndicationFeed.Load(xmlReader);
     Dispatcher.BeginInvoke(() =>
       {
         // dowiązanie listy elementów kanału RSS do naszego ListBox
         feedListBox.ItemsSource = feed.Items;
          });
  }
}
```

Ustawienie *ItemsSource* dla listy elementów powoduje odświeżenie listy i gotowa aplikacja wygląda podobnie jak na rysunku 4.4.

Rys 4.4. Gotowa aplikacja – czytnik kanału RSS

Niestety, dane z kanału RSS zawierają także dodatkowe informacje w postaci znaczników języka HTML, widoczne również na rysunku 4.4. Usunięcie ich będzie stanowiło jedno z zadań do rozwiązania samodzielnego.

4.3.3. Pozostałe funkcje klasy WebClient

Klasa *WebClient* pozwala również na wykorzystanie metod *CancelAsync()*, która anuluje aktualnie oczekującą operację oraz *UploadStringAsync()*, która jest bezpośrednim odpowiednikiem pobierania danych w postaci łańcucha znaków, ale odpowiada za wysyłanie łańcucha do pewnej usługi za pomocą metody POST protokołu HTTP.

Klasa *WebClient* udostępnia również strumienie – metody *OpenReadAsync()* i *OpenWriteAsync()* oferują możliwość zapisywania i odczytywania danych z usług HTTP poprzez wykorzystanie klas strumieniowych, co pozwala między innymi na stworzenie własnej implementacji wysyłania i odbierania plików.

Klasa *WebClient* pozwala też na odczytanie nagłówków odpowiedzi serwera poprzez właściwość *ResponseHeaders* oraz na uwierzytelnianie HTTP-Basic za pomocą przekazania odpowiednich danych do właściwości *Credentials*. Poprzez właściwość *Headers*, możliwe jest ustawienie własnych nagłówków przekazywanych do zdalnego serwera, takich jak *User-Agent* czy dodatkowe nagłówki wymagane przez niektóre usługi (np. wersja używanego API).

4.4. Kontrolka WebBrowser

Kontrolka *WebBrowser* służy do umieszczenia w aplikacji okna, którego zawartość jest renderowana w identyczny sposób jak zawartość okna systemowej przeglądarki Internet Explorer. Pozwala to na wykorzystanie silnika tej przeglądarki internetowej do zbudowania własnej przeglądarki stron WWW lub do wzbogacenia aplikacji o odczyt dodatkowych informacji ze stron bez konieczności przełączania się przez użytkownika do innej aplikacji.

Zawartość kontrolki *WebBrowser*, podobnie jak *Map*, nie jest widoczna w podglądzie formatki i zawiera tylko ikonę przeglądarki. Jej podstawową metodą jest metoda *Navigate()* pozwalająca na przejście do wskazanego adresu URL. Przykład 4.6 prezentuje załadowanie do kontrolki o nazwie *webBrowser1* strony internetowej Politechniki Lubelskiej.

Przykład 4.6. Wykorzystanie metody Navigate()

```
webBrowser1.Navigate(new Uri("http://pollub.pl"));
```

Kontrolka *WebBrowser* dysponuje kilkoma zdarzeniami, które mogą posłużyć do obsługi sytuacji, kiedy nawigacja się powiodła lub nie.

Są to zdarzenia:

- *Navigating* (strona w trakcie ładowania),
- *Navigated* (nastąpiło przejście do strony),
- *NavigationFailed* (przejście się nie powiodło – na przykład wystąpił błąd w adresie serwera).

Kiedy strona się już w pełni załaduje następuje zdarzenie *LoadCompleted*. Wszystkie te zdarzenia można wykorzystać przykładowo do stworzenia pasków postępu ładowania strony internetowej w przeglądarce.

WebBrowser dysponuje też metodą *NavigateToString()* pozwalającą na załadowanie pewnego kodu HTML pochodzącego z dowolnego obiektu typu *string*, co pozwala na wyświetlenie w niej 'bogato sformatowanych' danych. W prezentowanym przykładzie czytnika RSS można wykorzystać kontrolkę *WebBrowser*, aby wyświetlać dane zachowując oryginalne formatowanie źródła.

Dodatkowo, kontrolka *WebBrowser* obsługuje język JavaScript osadzony na stronach internetowych. Wymagane jest do tego włączenie obsługi skryptów przez ustawienie wartości *true* dla pola *IsScriptEnabled*.

Związana jest z tym bardzo ważna własność kontrolki *WebBrowser* – dostępność zdarzenia *ScriptNotify*, które zachodzi gdy JavaScript na stronie wywoła metodę *window.external.notify()* i służy do stworzenia komunikacji między kontrolką w naszej aplikacji mobilnej, a zewnętrzną aplikacją internetową. Przykładowo, można przygotować aplikację płatniczą w taki sposób, że będzie ona uruchamiała stronę internetową z formularzem do dokonania płatności, a ta z kolei powiadomi aplikację na telefonie z użyciem *ScriptNotify*, że opłata została wykonana. Zachowa to spójność pomiędzy systemami operacyjnymi ponieważ część internetową można wykonać na wielu różnych platformach.

4.5. Zadania do samodzielnego rozwiązania

Zadanie 4.1.

Napisz program korzystający z usługi sieciowej i realizujący konwersję temperatury, wyposażony w kontrolkę *TextBox*, *TextBlock* oraz *Radio* do celów wyboru kierunku konwersji.

Zadanie 4.2.

Wykorzystaj klasę *WebClient* do stworzenia prostego czytnika RSS zgodnie z przykładem. Następnie wykorzystaj kontrolkę *WebBrowser* umieszczoną w szablonie wyświetlania danych z czytnika RSS, aby dane HTML zapisane w kanale informacyjnym były poprawnie wyświetlanie.

5. Przechowywanie danych

5.1. Wstęp

Dane dotyczące stanu aplikacji można zachować z wykorzystaniem kolekcji *State*. *State* jednak wygasa i dane z niej są usuwane kiedy aplikacja jest zamykana. Dodatkowo ilość danych zachowywanych w ten sposób jest ograniczona.

Platforma Windows Phone dysponuje dwoma mechanizmami przechowywania dużych ilości danych w aplikacjach – za pomocą *Isolated Storage*, mechanizmu podobnego do systemu plików w systemach desktopowych oraz wbudowanej bazy danych SQL Server Compact Edition (SQL CE).

Dodatkowo, możliwa jest współpraca pomiędzy zewnętrznymi aplikacjami z wykorzystaniem kilku mechanizmów oraz pobieranie danych z samego systemu operacyjnego. Mechanizmy te opisane będą w kolejnych podrozdziałach.

5.2. Isolated Storage

Windows Phone uruchamia aplikacje w ramach tzw. piaskownicy (*sandbox*). Każda aplikacja jest izolowana od innych i dane każdej aplikacji są niezależne od danych innych aplikacji. Pomimo istnienia w systemie rzeczywistego systemu plików identycznego z systemem plików na komputerze osobistym, niemożliwe jest dla aplikacji uzyskanie dostępu do plików poza tzw. *Isolated Storage* czyli izolowanym obszarem przechowywania danych.

Przechowywanie danych za pomocą *Isolated Storage* obsługiwane jest przez przestrzeń nazw *System.IO.IsolatedStorage* i jest bardzo podobne do klasycznych metod wejścia/wyjścia w plikach opartych o strumienie. Szeroko używana jest tu konstrukcja *using {}* pozwalająca automatycznie zwalniać nie używane już obiekty dynamiczne.

Statyczna metoda *IsolatedStorageFile.GetUserStoreForApplication()* pobiera dla aplikacji obiekt klasy *IsolatedStorageFile* pozwalający na podstawowe operacje takie jak tworzenie plików, usuwanie plików, tworzenie katalogów, sprawdzanie istnienia plików itp.

Przykład 5.1 przedstawia wykorzystanie klasy *IsolatedStorageFile* oraz klasy *IsolatedStorageFileStream* w celu zapisania dowolnego tekstu do pliku tekstowego o podanej nazwie. Metoda z przykładu 5.1. może być wykorzystana również w kolejnych aplikacjach.

Przykład 5.1. Zapis tekstu do nowego pliku w IsolatedStorage

```
// zapis tekstu z parametru text do pliku o nazwie fileName
public void SaveText(string fileName, string text)
```

```
{
  // pobieramy obiekt IS
  using (IsolatedStorageFile isf =
        IsolatedStorageFile.GetUserStoreForApplication())
        {
          // tworzony jest nowy plik i automatycznie zwalniany,
          // kiedy przestajemy korzystać
          using (IsolatedStorageFileStream isfStream =
                    isf.CreateFile(fileName))
                {
                    // tworzony jest zapisywacz do strumienia pliku
                    StreamWriter writer = new
                              StreamWriter(isfStream);
                    writer.Write(text);
                    writer.Close(); // zamykanie strumienia!
                }
        }
}
```

Z kolei w przykładzie 5.2 przedstawiono metodę komplementarną, odczytującą dane z pliku tekstowego do zmiennej typu *string*. Ponieważ możliwe jest, że na przykład plik nie istnieje, więc wywołanie tej metody powinno być ujęte w blok *try..catch*.

Przykład 5.2. Odczyt tekstu z pliku do łańcucha znaków

```
// odczyt pliku o nazwie fileName i zwrócenie jego zawartości
public string LoadText(string fileName)
{
   string result;
   using (IsolatedStorageFile isf =
       IsolatedStorageFile.GetUserStoreForApplication())
       {
         // tutaj możliwe jest, że plik nie istnieje
         using (IsolatedStorageFileStream readerStream =
                    isf.OpenFile(fileName, FileMode.Open))
                {
                  // tutaj tworzony jest odczytywacz - ale możliwe
                  //jest, że pliku nie da się odczytać
                  // i nastąpi wyjątek
                  StreamReader reader = new
                                      StreamReader(readerStream);

                  // odczyt od początku do końca do łańcucha znaków
                  result = reader.ReadToEnd();
                  reader.Close();
         }
       }
```

```
    return result;
}
```

5.2.1. IsolatedStorageSettings

Kiedy tworzone są i zapisywane wyłącznie pewne dane konfiguracyjne, własna obsługa plików może być zbyt skomplikowana. Choć oczywiście pozwala ona na zdecydowanie więcej możliwości. Do zapisu prostych ustawień można jednak wykorzystać klasę *IsolatedStorageSettings*. Klasa ta zapisuje pewne proste dane w automatycznie tworzonych strukturach w koncepcji słownika klucz-wartość. Metoda *Remove()* usuwa dany klucz. Metoda *Add()* dodaje klucz o danej wartości lub nadpisuje już istniejący. Metoda *Save()* zapisuje informacje w *Isolated Storage*. Do odczytania danych można zastosować proste wykorzystanie słownika, jak zademonstrowano na przykładzie 5.3. Pewną wadą jest tu jednak fakt, że *IsolatedStorageSettings* nie dysponuje metodą *ContainsKey()* lub podobną, przez co nie da się sprawdzić, czy klucz o danej nazwie istnieje. Dodatkowo, *IsolatedStorageSettings* także zapisuje wszystkie przekazane do niej wartości jako *object*, więc wymagane jest rzutowanie przy odczycie danych.

Przykład 5.3. Wykorzystanie IsolatedStorageSettings

```
// pobranie obiektu właściwości (lub utworzenie nowego
// jeśli nie istniał)
IsolatedStorageSettings isolatedStore =
                    IsolatedStorageSettings.ApplicationSettings;
isolatedStore.Remove("klucz"); // usunięcie klucza
isolatedStore.Add("klucz2", "wartość2"); //dodanie klucza o wartości
isolatedStore.Save(); // zapis danych
MessageBox.Show((string)isolatedStore["klucz"]); // odczyt danych
                    // spod klucza "klucz" (z rzutowaniem)
```

5.3. SQL CE

Począwszy od Windows Phone 7.5 możliwe jest przechowywanie przez programistę danych relacyjnych w lokalnej bazie danych zachowywanej w lokalnym folderze aplikacji.

Aplikacje Windows Phone mogą wykorzystywać mechanizm LINQ to SQL do wszystkich operacji bazodanowych, co zapewnia podejście zorientowane obiektowo. W przypadku platformy Windows Phone, LINQ to SQL nie obsługuje bezpośredniego wywoływania języka T-SQL dla silnika bazy danych, niemożliwe jest także uzyskanie bezpośredniego dostępu do obiektów ADO.NET.

Lokalna baza danych działa w ramach procesu aplikacji. Nie działa w tle, jak zwykły silnik bazodanowy. Nie jest również dostępna z poziomu żadnej innej aplikacji.

5.3.1. Definiowanie kontekstu danych

Aby stworzyć lokalną bazę danych należy zdefiniować tzw. kontekst danych, będący klasą dziedziczącą po *DataContext* i zawierającą odwołania do tabel.

Tabele bazodanowe realizowane są jako znane już proste obiekty podobne do modelu danych, mogą jednak zawierać dodatkowe atrybuty oznaczające pewne informacje dotyczące bazy danych.

Przykładowy kontekst danych dla aplikacji przechowującej listę rzeczy do zrobienia mógłby wyglądać tak jak w przykładzie 5.4.

Przykład 5.4. DataContext aplikacji

```
public class ToDoDataContext : DataContext
{
  // ConnectionString – definicja pliku w którym
  // przechowywana jest baza
  public static string DBConnectionString =
                       "Data Source=isostore:/ToDo.sdf";

  // wywołanie bazowego konstruktowa
  public ToDoDataContext(string connectionString) :
                              base(connectionString) { }

  // definicja tabel
  public Table<ToDoItem> ToDoItems;
}
```

Aplikacja z przykładu 5.4 posiada tylko jedną tabelę – za jej definicję odpowiada klasa *ToDoItem*. Typową konwencją nazewniczą jest, aby nazwa pola w kontekście danych była liczbą mnogą nazwy klasy reprezentującej tabelę. Klasa *DataContext* pochodzi z przestrzeni nazw *System.Data.Linq*.

W kolejnym przykładzie 5.5, w *Table* prezentowane jest tylko jedno pole identyfikatora, a pozostałe można zdefiniować analogicznie.

Interfejsy *INotifyPropertyChanged* i *INotifyPropertyChanged* wymagają samodzielnej implementacji.

Przykład 5.5. Definicja klasy encji (tabeli)

```
[Table]
public class ToDoItem : INotifyPropertyChanged,
                        INotifyPropertyChanging
{
  // definicja id – prywatne pole, publiczna właściwość
  private int _toDoItemId;

  // definicja kolumny – jest kluczem prywatnym, nie moze byc nullem
  [Column(IsPrimaryKey = true, IsDbGenerated = true,
```

```
          DbType = "INT NOT NULL Identity", CanBeNull = false,
          AutoSync = AutoSync.OnInsert)]
  public int ToDoItemId
  {
     get { return _toDoItemId; }
     set { if (_toDoItemId != value)
           {
              NotifyPropertyChanging("ToDoItemId");
              _toDoItemId = value;
              NotifyPropertyChanged("ToDoItemId");
           }
         }
  }

  (…) // pozostałe atrybuty encji

  public event PropertyChangedEventHandler PropertyChanged;

  public event PropertyChangingEventHandler PropertyChanging;
}
```

Atrybuty *Table* i *Column* pochodzą z przestrzeni *System.Data.Linq.Mapping*. Możliwe jest także zdefiniowanie atrybutów *Index*, który włącza indeksowanie dla danego pola oraz *Association*, które zapewnia możliwość asocjacji, powiązania pomiędzy kluczem obcym i kluczem pierwotnym.

5.3.2. Tworzenie i użycie bazy danych

Kiedy zdefiniowane są typy danych i opisane tabele za pomocą odpowiednich klas encji możliwe jest utworzenie bazy danych, jeśli nie istniała wcześniej, lub jej otwarcie, jeśli istniała. W przykładzie 5.6 ponownie zastosowano konstrukcję *using*.

Przykład 5.6. Tworzenie bazy danych

```
// otwórz bazę danych
using (ToDoDataContext db = new
       ToDoDataContext(ToDoDataContext.DBConnectionString))
       {
          if (db.DatabaseExists() == false)
          {
             // jeśli nie istnieje - stwórz ją
             db.CreateDatabase();
          }
       }
```

Kiedy baza została stworzona, możliwe jest wykorzystanie LINQ to SQL do wybierania danych, w podobny sposób jak zademonstrowano na przykładzie 5.7.

Przykład 5.7. Wybieranie danych z bazy

```
// otwórz bazę danych
using (ToDoDataContext db = new
       ToDoDataContext(ToDoDataContext.DBConnectionString))
       {
          // zapytanie wybierające wszystkie elementy o kluczu
          // większym od 2
          var toDoItemsInDB = from ToDoItem todo in db.ToDoItems
                              where todo.ToDoItemId > 2 select todo;

          // faktycznie wywołanie zapytania i wstawienie wyniku
          // do kolekcji
          var ToDoItems = new
               ObservableCollection<ToDoItem>(toDoItemsInDB);
       }
```

Wstawianie danych do bazy (Przykład 5.8) wymaga z kolei operacji dwustopniowej – najpierw obiekt dodawany jest do kontekstu danych, a potem metoda *SubmitChanges()* faktycznie zapisuje go do bazy danych.

Przykład 5.8. Wstawianie danych do bazy

```
// tworzenie nowego obiektu z polem Id, ItemName i innymi
ToDoItem newToDo = new ToDoItem { ToDoItemId = 3,
                                  ItemName = "test" (…) };

// dodawanie obiektu do bazy
db.ToDoItems.InsertOnSubmit(newToDo);
db.SubmitChanges();
```

Aktualizacja bazy danych wymaga zmiany w odpowiednim obiekcie reprezentującym encję, a potem wywołania metody *SubmitChanges()*. Ostatnia z operacji bazodanowych – usuwanie, jest realizowana w podobny sposób, co demonstruje przykład 5.9.

Przykład 5.9. Usuwanie danych z bazy danych

```
// wybranie wpisu o id = 3
var doUsuniecia = from ToDoItem todo in db.ToDoItems
                  where todo.ToDoItemId == 3 select todo;
db.ToDoItems.DeleteOnSubmit(doUsuniecia.FirstOrDefault());
// zatwierdzenie usuwania
db.SubmitChanges();
```

5.4. Wymiana danych pomiędzy aplikacjami

Czasem niezbędna jest interakcja pomiędzy jedną aplikacją, a innymi aplikacjami na tym samym urządzeniu. W przypadku oprogramowania działającego na telefonie może też zajść konieczność wymiany informacji pomiędzy samym oprogramowaniem telefonu oraz zewnętrznymi aplikacjami. W tym celu będzie można wykorzystywać mechanizmy *zadań* (*launchers* oraz *choosers*), wyboru dowiązań do plików oraz własnych protokołów. Te dwa ostatnie mechanizmy pozwalają też na współpracę z zewnętrznymi plikami lub protokołami.

5.4.1. Launchers

Standardowe aplikacje z systemu Windows Phone udostępniają zbiór metod, które pozwalają na uruchomienie tych standardowych aplikacji z pewnymi parametrami. *Launchers* nie zwracają wyników do aplikacji. Zdefiniowane są w przestrzeni nazw *Microsoft.Phone.Tasks*.

Przykładowymi zadaniami *Launchers* są:

- *PhoneCallTask* – uruchomienie programu telefonu, można przekazać numer telefonu, ale użytkownik musi potwierdzić rozpoczęcie rozmowy; nie można przekazać kodów USSD;
- *EmailComposeTask* – z poziomu programu można zdefiniować własności wiadomości e-mail (temat, treść, odbiorca), a użytkownik zobaczy okno redagowania wiadomości z wpisanymi informacjami, ale musi potwierdzić wysyłanie (a konto e-mail musi być skonfigurowane na telefonie);
- *SmsComposeTask* – uruchomienie programu redagowania wiadomości SMS, można zdefiniować odbiorcę i treść, ale to użytkownik musi potwierdzić wysyłanie;
- *WebBrowserTask* – uruchomienie przeglądarki internetowej z załadowanym z poziomu programu użytkownika adresem URL;
- *SearchTask* – uruchomienie aplikacji Bing z hasłem do wyszukania;
- *MediaPlayerLauncher* – uruchomienie programu *Muzyka+Wideo* z załadowanym plikiem do odtwarzania;
- *MarketplaceDetailsTask* – uruchomienie sklepu i pokazanie szczegółów konkretnej aplikacji;
- *MarketplaceHubTask* – uruchomienie programu Sklep;
- *MarketplaceSearchTask* – uruchomienie wyszukiwania w Sklepie i wyświetlenie wyników wyszukiwania;
- *MarketplaceReviewTask* – uruchomienie Sklepu w celu wyświetlenia strony oceny i recenzji dla określonego produktu

- *SaveEmailAddressTask* – uruchomienie programu Kontakty w celu zapisania lub dodania do istniejącego kontaktu adresu e-mail z programu użytkownika
- *SavePhoneNumberTask* – uruchomienie programu Kontakty w celu zapisania lub dodania do istniejącego kontaktu numeru telefonu z programu użytkownika;
- *SaveAppointmentTask* – uruchomienie programu Kalendarz i zapis nowego spotkania.

Po uruchomieniu (pokazaniu) zadania na ekranie pojawia się okno odpowiedniego programu, a po zakończeniu zadania program użytkownika zostanie wznowiony. Przykładowo, aby uruchomić opcję dzwonienia pod dany numer telefonu można posłużyć się konstrukcją jak w przykładzie 5.10.

Przykład 5.10. Wykorzystanie telefonu

```
// automatyczne tworzenie anonimowego obiektu, ustawienie jego
// właściwości PhoneNumber na numer telefonu
// oraz wywołanie na nim metody Show()
(new PhoneCallTask() { PhoneNumber = "+48815384368" }).Show();
```

Z kolei aby wywołać okno redagowania wiadomości e-mail można wykorzystać taką konstrukcję, jak w przykładzie 5.11. Podczas gdy w przykładzie 5.10 zastosowano skróconą wersję jednolinijkową, w 5.11 podana jest wersja bardziej rozbudowana.

Przykład 5.11. Wykorzystanie komponowania wiadomości e-mail

```
var n = new EmailComposeTask();
n.To = "billg@contoso.com"; // adres odbiorcy
n.Subject = "Ważne!"; // temat
n.Show(); // pokazanie okna
```

5.4.2. Choosers

Choosers („wybieracze”) są konstrukcją analogiczną do *Launchers*, ale potrafią one zwracać do programu użytkownika wyniki swojego działania. Aby umożliwić zwracanie wyników, program użytkownika musi obsłużyć odpowiednie zdarzenia zakończenia działania *Choosers*.

Dostępne są następujące *Choosers*:

- *CameraCaptureTask* – uruchomienie programu Aparat w celu zrobienia zdjęcia;
- *EmailAddressChooserTask* – uruchomienie programu Kontakty i umożliwienie użytkownikowi zaznaczenia określonego adresu e-mail danego kontaktu;

- *PhoneNumberChooserTask* – uruchomienie programu Kontakty i umożliwienie użytkownikowi zaznaczenia określonego adresu e-mail danego kontaktu;
- *PhotoChooserTask* – uruchomienie programu Zdjęcia, i umożliwienie użytkownikowi wyboru pliku z danym zdjęciem (obrazem);
- *Contacts* – uruchomienie programu Kontakty w celu wyszukania kontaktu na podstawie adresu e-mail czy numeru telefonu, przekazywane są wszystkie dane na temat kontaktu;
- *Appointments* – uruchomienie programu Kalendarz i wybór pewnego spotkania (terminu).

Kontakty (*Contacts*) oraz Terminy (*Appointments*) zawarte są w przestrzeni nazw *Microsoft.Phone.UserData*, podczas gdy pozostałe – w przestrzeni nazw *Tasks*, jak wszystkie *Launchers*.

W celu wykorzystania tego mechanizmu w przykładzie 5.12, aby pobrać zrobione przez użytkownika zdjęcie – zastosowano *CameraCaptureTask*. W konstruktorze strony inicjalizowany jest obiekt klasy *CameraCaptureTask* oraz dodawana jest procedura obsługi zdarzenia *Completed* tej klasy. *Chooser* uruchamiany jest automatycznie. W obsłudze zdarzenia następuje sprawdzenie, czy zadanie się powiodło i jeśli tak, to do kontrolki *Image* ładowany jest zrobiony przez użytkownika obrazek.

Przykład 5.12. Wykorzystanie CameraCaptureTask

```
public partial class MainPage : PhoneApplicationPage
{
   CameraCaptureTask cct;
   // Constructor
   public MainPage()
   {
      InitializeComponent();

      cct = new CameraCaptureTask();
      cct.Completed += cct_Completed;

      cct.Show();
   }

   void cct_Completed(object sender, PhotoResult e)
   {
     if (e.TaskResult == TaskResult.OK)
         image1.Source =
                new BitmapImage(new Uri(e.OriginalFileName));
   }
(…)
```

5.4.3. Własne protokoły

W Windows Phone 8 aplikacja może wystawić własny punkt wejścia procedury, który inne aplikacje mogą wykorzystać, aby ją uruchomić. To realizowane jest poprzez rejestrację własnego protokołu, podobnego w działaniu do znanych HTTP czy HTTPS. Własnym protokołem może być każdy, który nie jest zarezerwowany w systemie. Inne aplikacje mogą przekazać pewne dodatkowe dane uruchamiając aplikację z danego protokołu.

W pliku *WMAppManifest.xml* w sekcji *<Extensions>* (należy ją dodać po sekcji *Tokens*) należy dodać nowy element *<Protocol>*. Przykład 5.13 prezentuje dodawanie protokołu o nazwie *foo*, czyli będzie on realizowany przez adresy URI w postaci *foo://cokolwiek*. Aby otworzyć plik w postaci edytora XML, a nie edytora graficznego, należy kliknąć prawym przyciskiem myszy i wybrać *Open with… -> XML Editor*.

Przykład 5.13. Rejestracja protokołu foo

```
<Extensions>
  <Protocol Name="foo"
            NavUriFragment="encodedLaunchUri=%s"
            TaskID="_default"
  />
</Extensions>
```

Niezbędne jest też sprawdzanie danych protokołu przy uruchomieniu aplikacji (Przykład 5.14). Wykorzystamy do tego klasę pochodną od klasy *UriMapper*, a aplikacja będzie przekazywać wynik do strony *MainPage.xaml* z wykorzystaniem parametrów, które podano w pierwszym rozdziale.

Przykład 5.14. Własna klasa UriMapper

```
public class MyAppUriMapper : UriMapperBase
{
   public override Uri MapUri(Uri uri)
   {  string tempUri = uri.ToString();
      if (tempUri.StartsWith("/Protocol?encodedLaunchUri="))
      {
         string deeplink =
                tempUri.Substring(tempUri.IndexOf("foo"));
         return new Uri("/MainPage.xaml?deepLink=" + deeplink,
                        UriKind.Relative);
      }
      else
      {  return uri;
      }
   }
}
```

Kolejny krok to dodanie do ramki *RootFrame* w pliku *App.xaml.cs* instrukcji uruchamiania *MyAppUriMapper* (Przykład 5.15). Można to wykonać na przykład po metodzie *InitializePhoneApplication.*

Przykład 5.15. Ustawienie Mappera

```
RootFrame.UriMapper = new MyAppUriMapper();
```

W metodzie *OnNavigatedTo* strony *MainPage* można teraz odczytać dane, które zostały przekazane do strony z protokołu w sposób zaprezentowany na przykładzie 5.16.

Przykład 5.16. Odczyt danych z adresu URI strony

```
protected async override void OnNavi-
gatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
   base.OnNavigatedTo(e);
   if (NavigationContext.QueryString.Any())
   {  StringBuilder sb = new StringBuilder();
      foreach (var key in NavigationContext.QueryString.Keys)
      {
         sb.AppendLine(key + ": " +
                       NavigationContext.QueryString[key]);
      }
      MessageBox.Show(sb.ToString(), "Querystring",
                                      MessageBoxButton.OK);
  }
}
```

W tym momencie aplikacja zostanie uruchomiona automatycznie, kiedy taki adres zostanie gdziekolwiek wykorzystany. Na przykład na stronie internetowej po kliknięciu na link postaci:

```
<a href="foo://someText">
   Open Custom Protocol foo://someText </a>
```

aplikacja się uruchomi i pokaże przekazane informacje.

5.4.4. Pozostałe mechanizmy

Windows Phone 8 wprowadził też możliwość rejestracji dowiązań do plików o określonym typie w podobny sposób jak jest to robione przy własnych protokołach (także sekcja *Extensions* w *WMAppManifest.xml*).

Dostępny jest także mechanizm do odczytu plików (powiązanych z aplikacją) z karty SD oraz bezpośredni zapis do kontaktów bez zgody użytkownika, jednak te kontakty znikną kiedy aplikacja zostanie odinstalowana.

5.5. Zadania do samodzielnego wykonania

Zadanie 5.1.

Napisz aplikację, która będzie notatnikiem zapisującym dane do pliku i odczytującym dane z pliku, aby były wciąż widoczne po zamknięciu i ponownym otwarciu aplikacji.

Zadanie 5.2.

Napisz aplikację, która przechowa dane dotyczące studentów z rozdziału 2 w bazie SQL CE. Następnie dodaj do aplikacji możliwość zapisu danych studenta do danych kontaktowych telefonu poprzez odpowiednie mechanizmy systemu.

Zadanie 5.3.

Napisz aplikację, która wczyta zrobione przez użytkownika zdjęcie i je wyświetli w pełnej wielkości, pozwalając na jego dowolne przesuwanie. Wykorzystaj w tym celu odpowiednie kontrolki związane z przewijaniem.

6. Projektowanie i publikowanie aplikacji

6.1. Wstęp

Aplikacje dla platformy Windows Phone mogą być na telefonach zwykłych użytkowników instalowane tylko z poziomu Sklepu. Tylko telefony odblokowane narzędziem Phone Registration (opisanym w podrozdziale 6.5) mają możliwość pracy z aplikacjami zbudowanymi w trybie debugowania i przesłanymi przez kabel USB.

Aby opublikować aplikację w sklepie z aplikacjami, niezbędne jest posiadanie opłaconego konta *Windows Phone Dev Center*, a jego koszt w momencie pisania tej książki został obniżony do 19 dolarów rocznie. Umożliwia to odblokowanie do 3 telefonów oraz publikowanie aplikacji i zarabianie na nich.

Możliwe jest też odblokowanie za darmo jednego telefonu i testowanie na nim do 2 aplikacji instalowanych spoza sklepu.

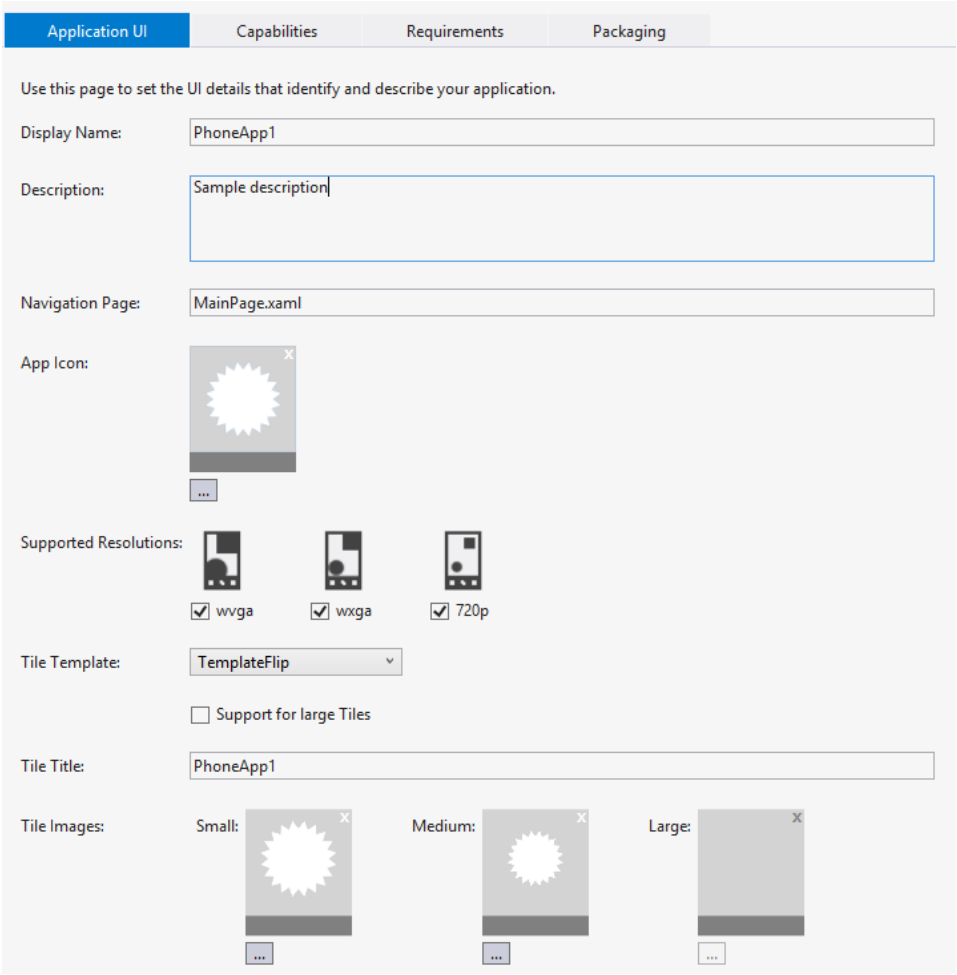
6.2. Uprawnienia aplikacji i manifest

W poprzednich rozdziałach kilkukrotnie było wspomniane o uprawnieniach aplikacji oraz manifeście, pliku *WMAppManifest.xml*. Plik ten grupuje pewne informacje publikowane wraz z aplikacją w sklepie, określając jej właściwości. Przede wszystkim są to uprawnienia aplikacji, obsługiwane rozdzielczości oraz ikona aplikacji. Po otwarciu pliku manifestu w Visual Studio pojawia się graficzny edytor, który przedstawiono na rysunku 6.1.

Korzystając z graficznego edytora manifestu możliwe jest ustalenie widocznej w systemie nazwy aplikacji, jej opisu, głównej strony, ikony aplikacji widocznej na liście na ekranie startowym.

Kolejną ważną rzeczą są obsługiwane rozdzielczości – domyślnie aplikacja się automatycznie skaluje, więc obsługa zmodyfikowanych rozdzielczości nie wymaga ingerencji od strony programisty. Problem może się jednak pojawić przy specyficznych układach ekranu lub przy grach, gdzie wymagane najpierw będą odpowiednie testy.

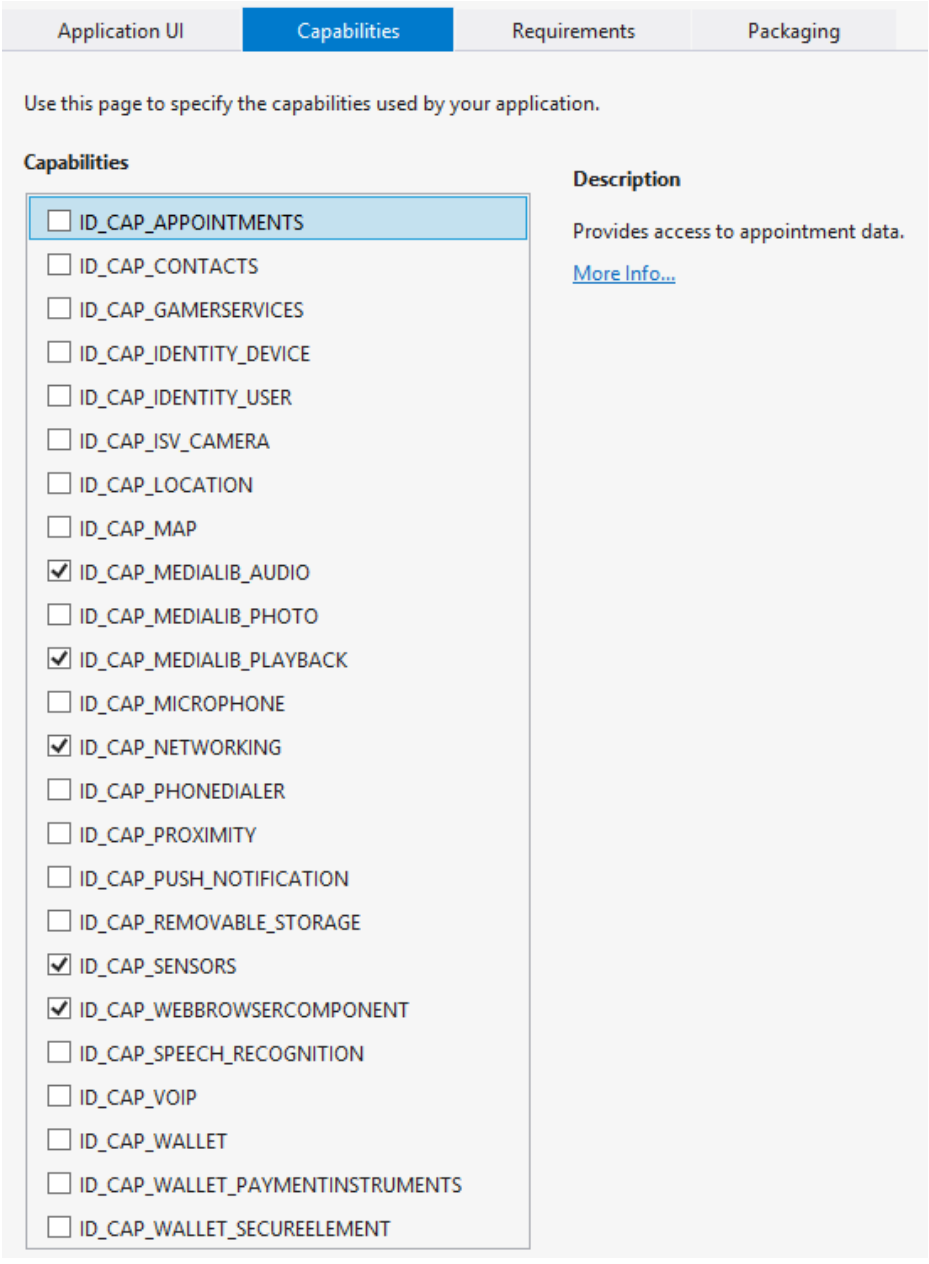
Kolejne opcje dotyczą ikon i nazw kafelków na ekranie głównym oraz ich szablonu. Zalecane jest, aby ikony dla poszczególnych wersji kafelków (mały, średni, ewentualnie duży) były przygotowywane oddzielnie, aby cały czas zachowywały wysoką jakość obrazu. Możliwe jest również ustalenie podpisu pod kafelkiem na ekranie głównym, który będzie inny niż nazwa aplikacji na liście aplikacji.



Rys 6.1. Graficzny edytor manifestu

6.2.1. Uprawnienia

Kolejna sekcja, *Capabilities*, określa uprawnienia dla konkretnej aplikacji (Rysunek 6.2). Jak wspomniano w przykładzie Map, aplikacja bez określonych uprawnień zostanie zatrzymana z wyjątkiem w sytuacji, kiedy spróbuje dostać się do elementów objętych uprawnieniami. W przypadku kiedy aplikacja publikowana jest w Sklepie, automatyczne systemy statycznej analizy kodu analizują i wybierają najmniejszy zbiór uprawnień wymaganych przez aplikację.



Rys 6.2. Sekcja ustawień uprawnień

Uprawnienia widoczne są przez użytkownika w chwili przeglądania aplikacji w Sklepie, przez co może on zwrócić uwagę na aplikacje potrzebujące nadmiernych uprawnień i ich nie zainstalować.

Najpopularniejsze uprawnienia opisane są w tabeli 6.1.

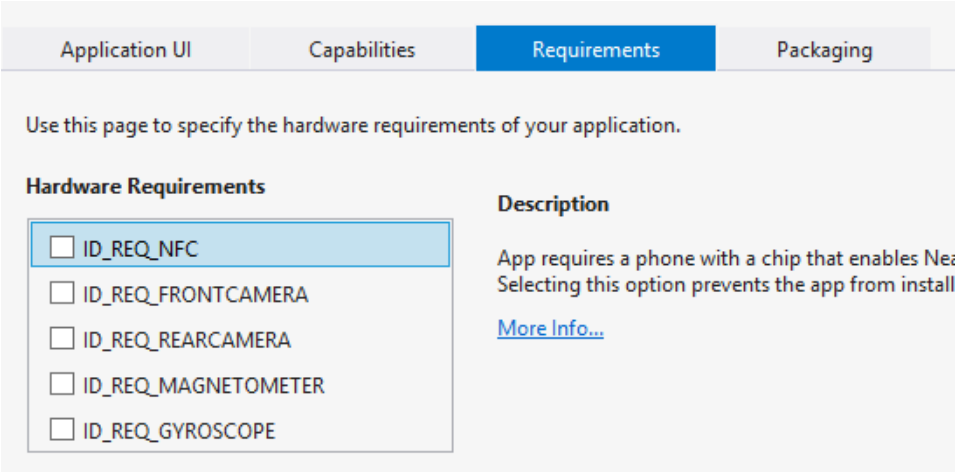
Tabela 6.1. Najczęściej stosowane uprawnienia aplikacji

ID_CAP_APPOINTMENTS	Dostęp do danych kalendarza
ID_CAP_CONTACTS	Dostęp do danych kontaktów
ID_CAP_IDENTITYDEVICE	Dostęp do identyfikatora urządzenia, jego nazwy lub modelu
ID_CAP_ISV_CAMERA	Bezpośredni dostęp do aparatu fotograficznego
ID_CAP_LOCATION	Dostęp do systemu GPS
ID_CAP_MAP	Dostęp do kontrolki Map
ID_CAP_NETWORKING	Dostęp do sieci
ID_CAP_SENSORS	Dostęp do danych z czujników
ID_CAP_WEBBROWSERCOMPONENT	Dostęp do kontrolki WebBrowser

Trzecia sekcja manifestu, *Requirements*, określa obowiązkowe wymogi dla aplikacji (rysunek 6.3). Możliwe jest, że aplikacja wymaga pewnych sprzętowych funkcji. W takiej sytuacji aplikacja nie pojawi się na liście aplikacji przeznaczonych dla danego telefonu w Sklepie, a po jej znalezieniu w inny sposób użytkownik zostanie poinformowany, że aplikacja jest niekompatybilna z jego telefonem.

Takich opcji nie należy zaznaczać, jeśli aplikacja korzysta z pewnych elementów, ale ich nie wymaga – np. potrafi działać bez cyfrowego kompasu lub żyroskopu, ale w inny sposób.

Ostatnia sekcja manifestu, *Packaging*, określa pewne dodatkowe informacje dotyczące samej aplikacji (takie jak obsługiwane języki, domyślny język, wersja pliku), które są pokazywane w Sklepie.



Rys 6.3. Wymagania aplikacji

6.3. Certyfikacja aplikacji

Zanim aplikacja zostanie opublikowana w Sklepie, musi przejść testy przeprowadzane przez pracowników firmy Microsoft. Testowane są, oprócz poprawności działania aplikacji, specyficzne dla Windows Phone wymagania certyfikacyjne. Wymagania są często uaktualniane i zmieniane, aby dostosować je do zmieniających się wymogów rynku. Aktualne wymagania certyfikacyjne dostępne są na stronie:
http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh184843%28v=vs.105%29.aspx

6.3.1. Wymogi co do aplikacji

Podstawową, najważniejszą zasadą w sekcji ogólnych wymogów co do aplikacji jest, że aplikacja nie może w żaden sposób naruszać bezpieczeństwa albo funkcjonalności aplikacji na telefonie, Sklepu, ani wpływać w nielegalny sposób na użytkowników.

Wymagane są również zgodne z prawem amerykańskim restrykcje co do weryfikacji wieku przy możliwości tworzenia kont do systemów wymiany informacji. Niezbędne jest w aplikacjach podawanie polityki prywatności przy wykorzystaniu danych geolokalizacyjnych, dostępu do kontaktów lub fotografii, zwłaszcza, kiedy możliwe jest wysyłanie ich na zdalne serwisy internetowe.

6.3.2. Wymogi co do zawartości

Aplikacje dla platformy Windows Phone nie mogą, z oczywistych powodów, naruszać żadnych praw do nazw firm i aplikacji, czy też logo.

Zabronione jest również przedstawianie lub zachęcanie do wrogości i okrucieństwa wobec ludzi i zwierząt w realnym świecie. Z podobnych wymogów aplikacja nie może promować nielegalnych środków odurzających, alkoholu, narkotyków oraz używania broni.

Zabronione w aplikacjach jest przedstawianie nagości i szeroko pojętej pornografii. Reguły te są wspólne dla wszystkich rynków na jakich może zostać opublikowana aplikacja, ale mogą w niektórych krajach obowiązywać dodatkowe wymogi – na przykład aplikacja na rynku chińskim nie może odnosić się do terytoriów o spornym statusie politycznym.

6.3.3. Wymogi co do publikowania aplikacji

Aplikacja musi korzystać tylko z udokumentowanych API – w aplikacjach Windows Phone 7 zabronione jest korzystanie z mechanizmów COM lub P/Invoke. To zalecenie zostało zmodyfikowane dla potrzeb Windows Phone 8, gdzie można korzystać z odpowiedniej listy dozwolonych API.

Aplikacja musi być zbudowana w konfiguracji *Release*, bez danych debugujących (nie dotyczy to aplikacji przeznaczonych do dystrybucji w wersji testowej), musi zostać także opublikowana przynajmniej w jednym języku wyświetlania Windows Phone.

Podczas umieszczania aplikacji w sklepie należy umieścić również zrzuty ekranu aplikacji, bez umieszczania informacji dla programistów czy składowych emulatora. Wśród wymogów podane są również informacje dotyczące dozwolonej wielkości ikon w aplikacji i widocznych w Sklepie.

6.3.4. Wymogi techniczne

Sekcja wymogów technicznych jest najszerszą w Sklepie. Wśród nich najważniejsze jest, aby:

- aplikacja obsługiwała wiele urządzeń spełniających ewentualne warunki techniczne;
- aplikacja obsługiwała wszystkie wyjątki, które może napotkać. Jeśli aplikacja „wywróci się" podczas testowania, najprawdopodobniej nie przejdzie certyfikacji;
- aplikacja nie może powodować braku odpowiedzi urządzenia na więcej niż 3 sekundy bez wyświetlania paska postępu lub wskaźnika zajętości;
- aplikacja musi pokazać pierwszy ekran po 5 sekundach od startu, a po 20 sekundach musi obsługiwać już interfejs użytkownika;
- przycisk Wstecz musi zapewnić tylko i wyłącznie nawigację wstecz;

- wciśnięcie przycisku wstecz na pierwszym ekranie aplikacji musi ją zamykać;
- wciśnięcie przycisku wstecz podczas pokazywania menu albo okna dialogowego musi je anulować;
- w przypadku gier przycisk Wstecz powinien wstrzymywać rozgrywkę;
- aplikacja nie może przekraczać 90 MB zajętej pamięci w Windows Phone 7.1;
- aplikacja nie może przeszkadzać w wykonywaniu połączeń telefonicznych i odbieraniu/wysyłaniu wiadomości SMS. Nie może również przestać działać w momencie nadejścia połączenia telefonicznego lub SMS;
- aplikacja nie może zawierać złośliwego oprogramowania.

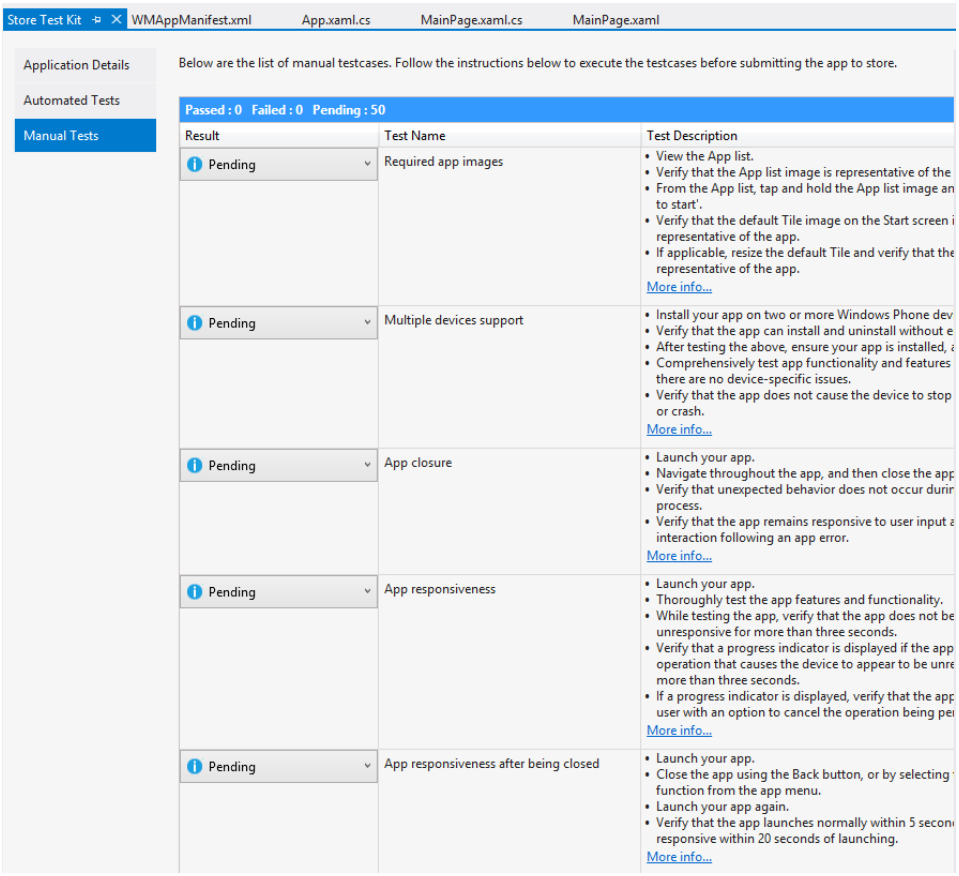
W przypadku specjalnych typów aplikacji istnieją dodatkowe wymagania. Dotyczą one aplikacji wykorzystujących geolokalizację, działających *pod* ekranem blokady, odtwarzaczy multimedialnych lub aplikacji typu VoIP.

6.3.5. Store Test Kit

Wymogi certyfikacyjne można przetestować otwierając *Store Test Kit* zawarty w menu kontekstowym projektu Windows Phone. Dostępna tam lista testów jest aktualizowana na bieżąco ze zmienianymi wymogami certyfikacyjnymi.

Dostępne są testy automatyczne, sprawdzające wielkość i format ikon oraz zrzutów ekranu, jak również wielkość plików wynikowych czy wymogów pamięci i czasu uruchomienia.

Jest również lista ręcznych testów (Rys. 6.4) zgodnych ze wszystkimi wymaganiami certyfikacyjnymi, wraz z informacjami jak przeprowadzić dany test. Przejście jednak testów w zbiorze testowym nie gwarantuje przejścia certyfikacji w Sklepie.

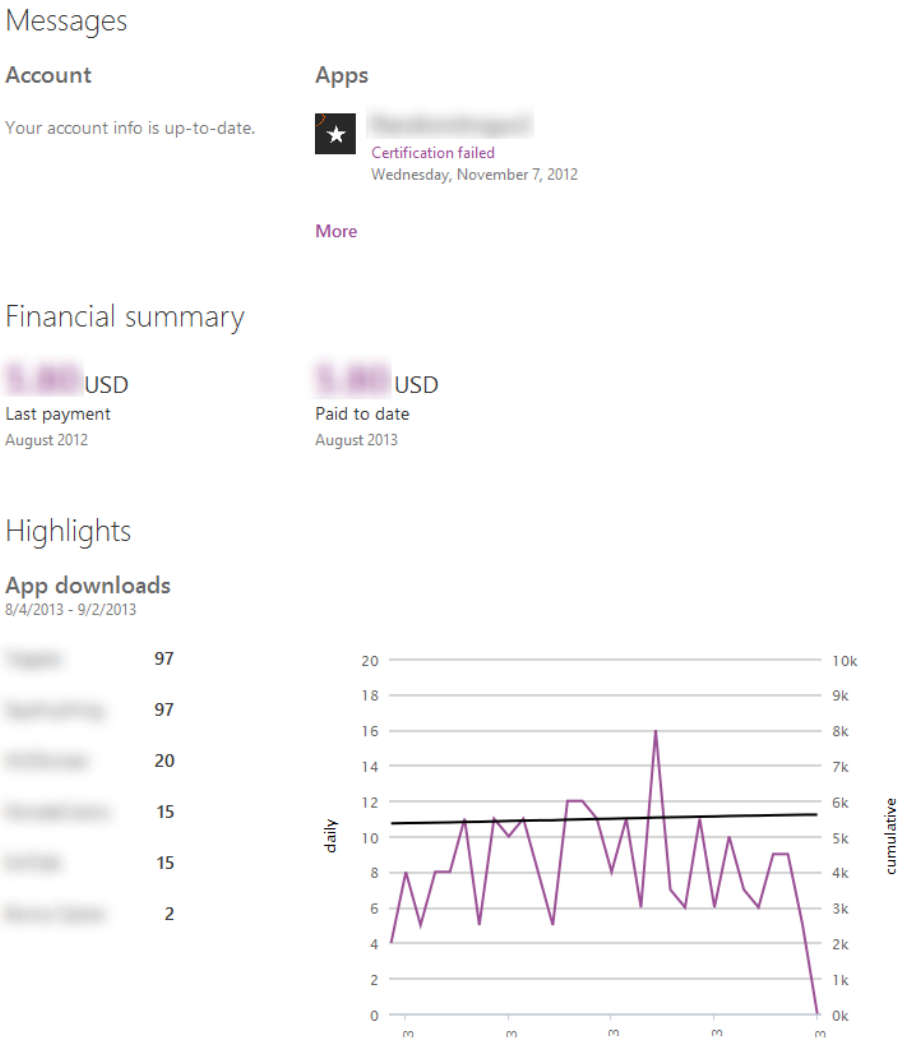


Rys 6.4. Store Test Kit z otwartą listą ręcznych testów

6.4. Publikowanie aplikacji w Sklepie

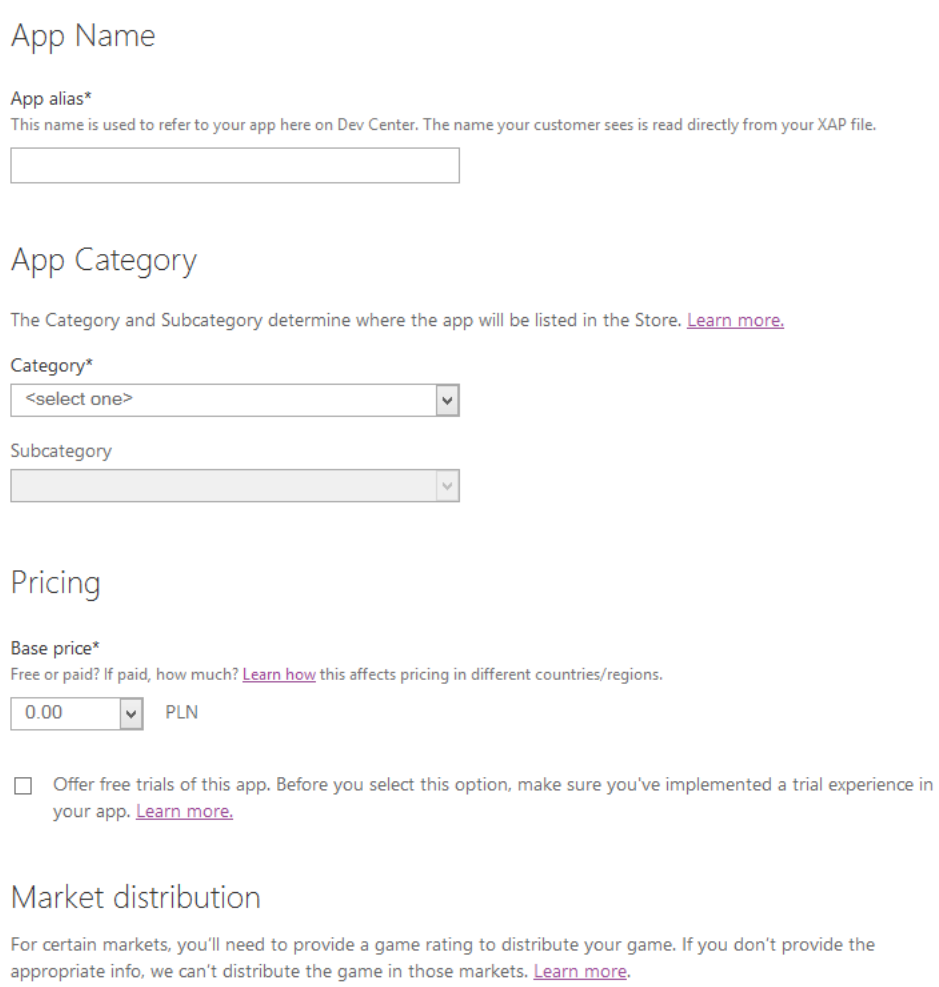
Kiedy aplikacja została zbudowana, a programista posiada opłacone konto w Sklepie, możliwe jest jej opublikowanie, a także śledzenie popularności, zarobionych pieniędzy na sprzedaży aplikacji płatnych, jak również analizy ewentualnych błędów, które są wysyłane z telefonów z włączoną opcją informowania o problemach.

Przedstawiony na rysunku 6.5 fragment okna *Dashboard* w Sklepie prezentuje listę aplikacji ostatnio dodanych, które nie przeszły certyfikacji, a także wykresy popularności pobrań wszystkich aplikacji właściciela konta.



Rys 6.5. Okno Dashboard konta programisty w Sklepie

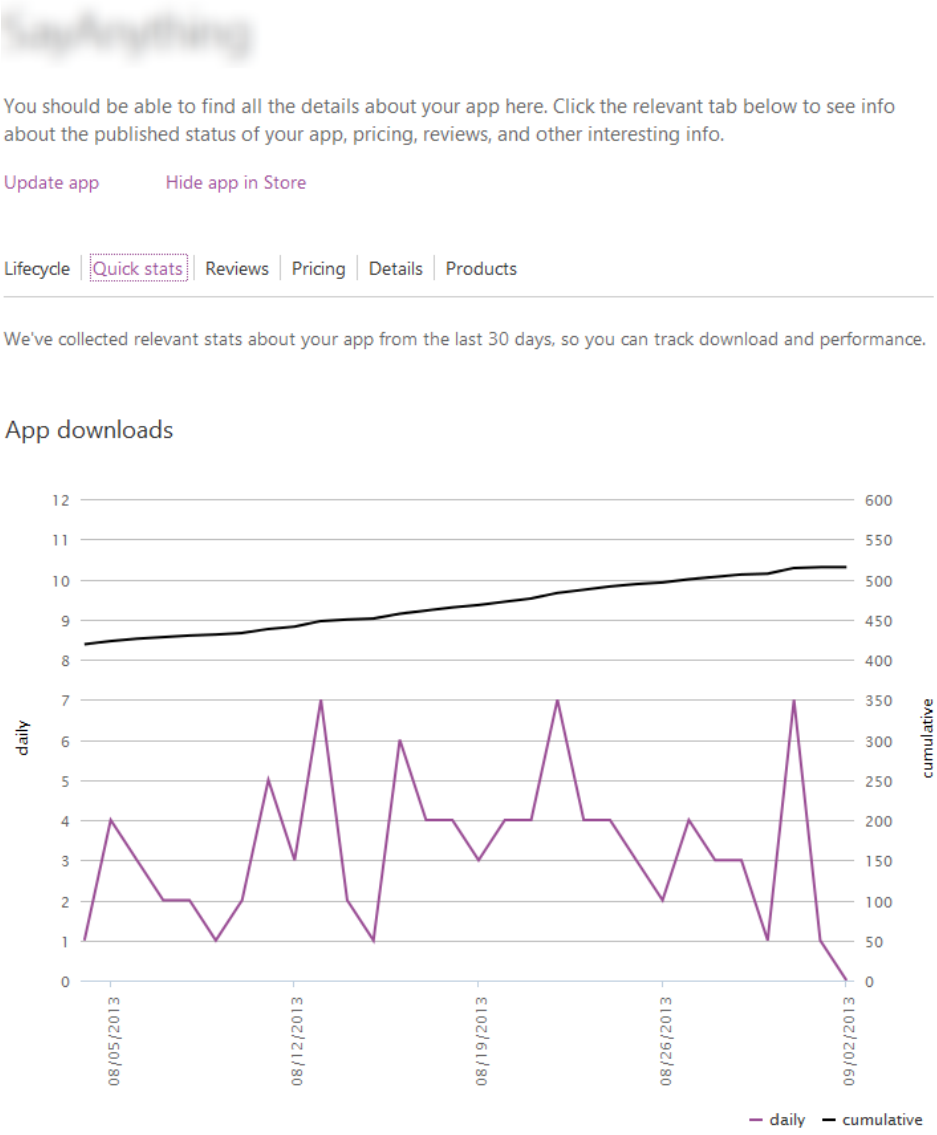
W przypadku dodawania nowej aplikacji do sklepu wymagane jest między innymi podanie jej nazwy (widoczne w panelu programisty), kategorii, ceny oraz rynków na jakich będzie dostępna. Fragment tego okna przedstawiono na rysunku 6.6.



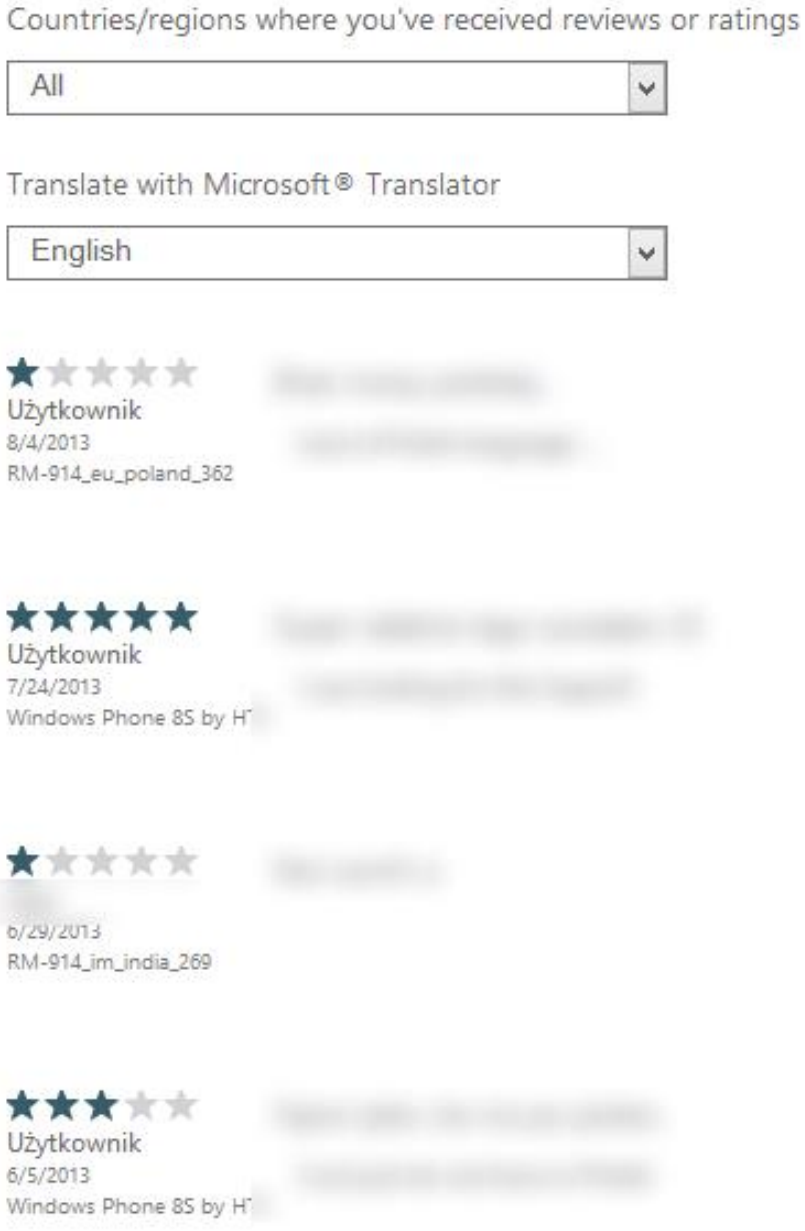
Rys. 6.6. Okno dodawania nowej aplikacji

Po spełnieniu tych wymogów można na serwer załadować plik XAP, paczkę aplikacji, wygenerowaną w Visual Studio, gdzie statyczna analiza paczki automatycznie wygeneruje między innymi nazwę oraz ikonografię aplikacji w dostępnych językach. Wysłana do celów certyfikacji aplikacja może zostać opublikowana od razu w Sklepie lub po ręcznym kliknięciu akceptacji. O wyniku certyfikacji autor zostanie powiadomiony wiadomością e-mail.

Kiedy aplikacja jest opublikowana w Sklepie, a użytkownicy mogą ją pobierać, programista ma możliwość analizy statystyk dotyczących aplikacji, jak przedstawiono na rysunkach 6.7 i 6.8 oraz zmiany jej opisu w Sklepie lub cen, za jaką jest dostępna.



Rys. 6.7. Popularność aplikacji w jej szczegółach

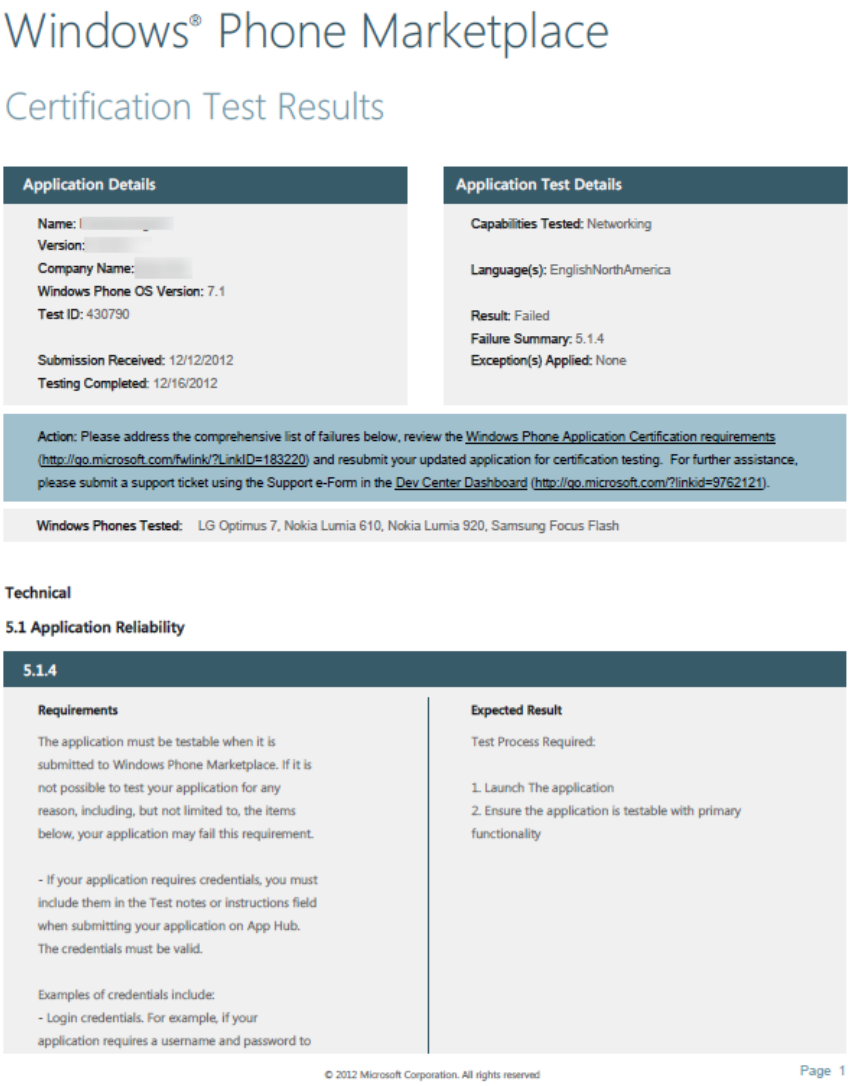


Rys 6.8. Oceny aplikacji widoczne dla programisty

6.4.1. Odrzucenie aplikacji podczas certyfikacji

Jeśli aplikacja nie spełnia wymogów certyfikacyjnych może zostać odrzucona. Programista dostaje wtedy wiadomość e-mail z informacjami o wersji aplikacji testowanej, testowanych telefonach, wymogach, które nie zostały spełnione oraz ewentualne dodatkowe komentarze testerów co dokładnie nie zostało spełnione.

Na rysunkach 6.9 oraz 6.10 zademonstrowano przykładowy błędny wynik testu certyfikacyjnego.



Windows® Phone Marketplace

Certification Test Results

Application Details

Name: [redacted]
Version:
Company Name:
Windows Phone OS Version: 7.1
Test ID: 430790

Submission Received: 12/12/2012
Testing Completed: 12/16/2012

Application Test Details

Capabilities Tested: Networking

Language(s): EnglishNorthAmerica

Result: Failed
Failure Summary: 5.1.4
Exception(s) Applied: None

Action: Please address the comprehensive list of failures below, review the Windows Phone Application Certification requirements (http://go.microsoft.com/fwlink/?LinkID=183220) and resubmit your updated application for certification testing. For further assistance, please submit a support ticket using the Support e-Form in the Dev Center Dashboard (http://go.microsoft.com/?linkid=9762121).

Windows Phones Tested: LG Optimus 7, Nokia Lumia 610, Nokia Lumia 920, Samsung Focus Flash

Technical

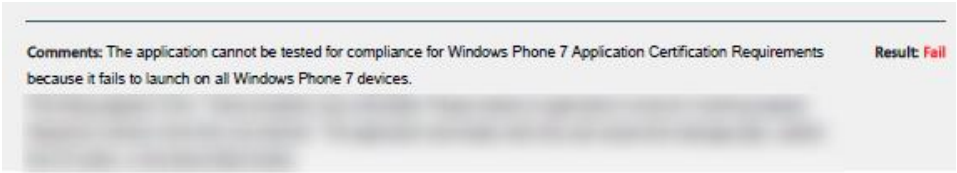
5.1 Application Reliability

5.1.4

Requirements	Expected Result
The application must be testable when it is submitted to Windows Phone Marketplace. If it is not possible to test your application for any reason, including, but not limited to, the items below, your application may fail this requirement. - If your application requires credentials, you must include them in the Test notes or instructions field when submitting your application on App Hub. The credentials must be valid. Examples of credentials include: - Login credentials. For example, if your application requires a username and password to	Test Process Required: 1. Launch The application 2. Ensure the application is testable with primary functionality

Rys 6.9. Pierwsza strona z informacjami o niepowodzeniu testu.

W przykładowych zrzutach ekranu przedstawionych na rysunkach 6.9 i 6.10 aplikacja nie mogła zostać przetestowana, ponieważ nie uruchamiała się poprawnie na żadnym z testowanych urządzeń.



Rys 6.10. Fragment drugiej strony z komentarzem testera

Testerzy potrafią też znaleźć wiele innych błędów w działaniu aplikacji, które mogły zostać pominięte przez testy programisty. Jeden z autorów niniejszej książki wysłał aplikację, która została odrzucona, ponieważ następował nieobsługiwany wyjątek po 37 (sic!) naciśnięciach klawisza *Dalej*.

6.5. Rejestracja telefonu

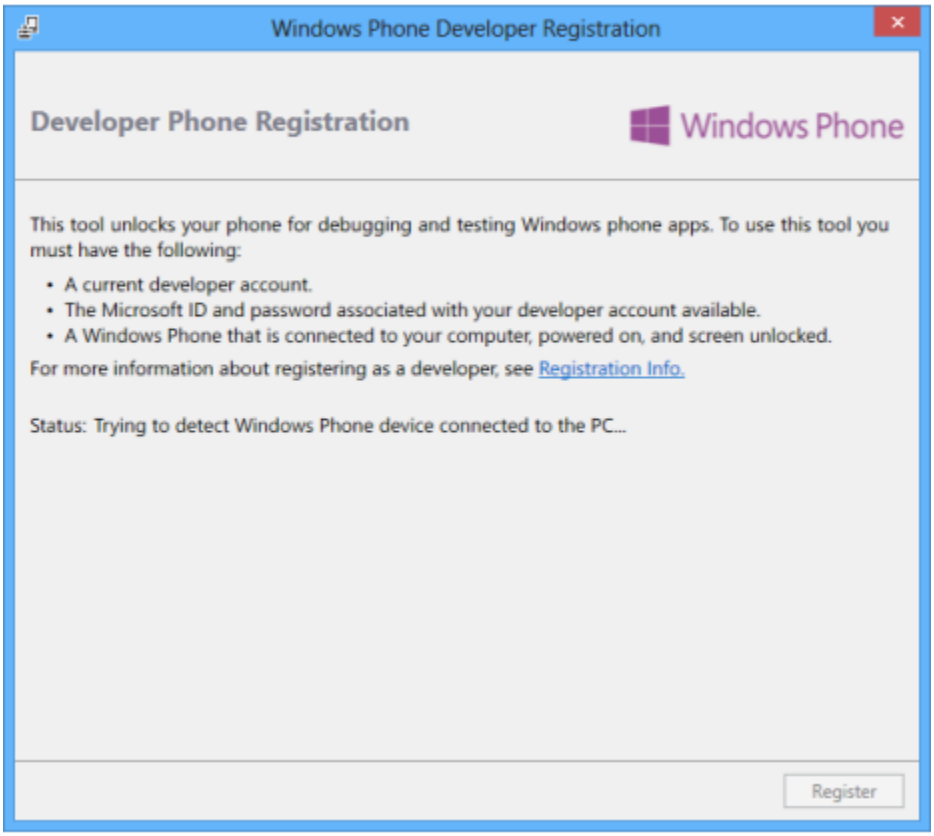
Aby można było wykorzystać własny telefon z systemem Windows Phone do testowania swoich aplikacji niezbędne jest jego odblokowanie. Do sierpnia 2013 roku tylko programiści dysponujący opłaconymi kontami mogli wykonać tę operację. Obecnie ograniczenie to zostało zniesione i wymagane jest tylko posiadanie konta Microsoft Account.

Po zainstalowaniu pakietu SDK należy uruchomić narzędzie *Windows Phone Developer Registration*. Przedstawiono je na rysunku 6.11.

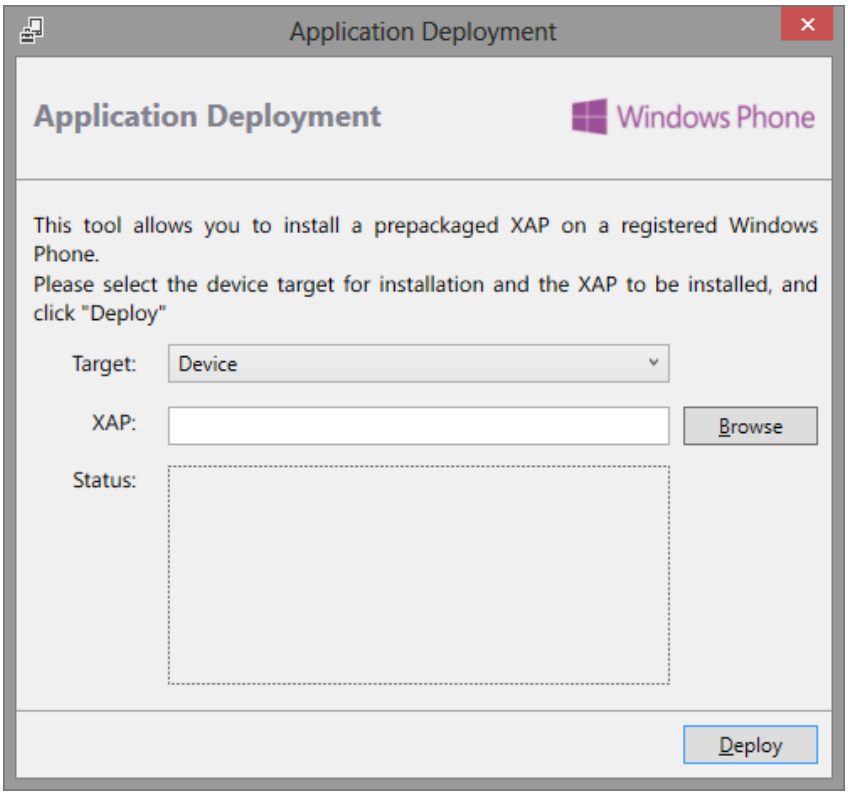
Kiedy narzędzie nawiąże kontakt z urządzeniem możliwe jest, że przycisk *Register* zmieni się na *Unregister* jeśli urządzenie jest już zarejestrowane. Jeśli nie jest, to wybranie tej opcji spowoduje wyświetlenie okna logowania do konta Microsoft, do którego zostanie przypisany odblokowany telefon.

Pojedyncze konto Microsoft może odblokować jeden telefon, na który będą mogły zostać załadowane maksymalnie 2 aplikacje spoza Sklepu. Konto Microsoft powiązane z kontem Sklepu może jednak zarejestrować do 3 telefonów, na które będzie można załadować do 10 własnych aplikacji.

Ładowanie aplikacji na telefon możliwe jest z poziomu Visual Studio lub z poziomu narzędzia *Application Deployment* (Rys. 6.12), kiedy dysponujemy plikiem XAP.



Rys 6.11. Narzędzie Windows Phone Developer Registration



Rys 6.12. Narzędzie Application Deployment

6.6. Zadania do samodzielnego wykonania

Zadanie 6.1.

Użyj mechanizmów emulatora do stworzenia zrzutów ekranu aplikacji.

Zadanie 6.2.

Wykorzystaj *Store Test Kit* i przetestuj jedną ze stworzonych przez siebie aplikacji na zgodność z wymogami certyfikacji. Przetestuj działanie aplikacji korzystając z *Simulation Dashboard*.

Zadanie 6.3.

Uruchom narzędzie do rejestracji telefonu i sprawdź stan rejestracji fizycznego urządzenia dostępnego na laboratorium.

Bibliografia

[1] Andrew Troelsen, Język C# 2010 i platforma .NET 4, PWN, 2011
[2] Marcin Lis, C#. Praktyczny kurs, Helion, 2012
[3] Charles Petzold, Programming Windows Phone 7, Microsoft Press, 2010
[4] Andrew Whitechapel, Windows Phone 7 Development Internals, Microsoft Press 2012
[5] Andrew Whitechapel, Windows Phone 8 Development Internals Preview 1, Microsoft Press 2012
[6] https://dev.windowsphone.com/
[7] http://developer.nokia.com/Community/Wiki/What%27s_new_in_Windows_Phone_8

Indeks