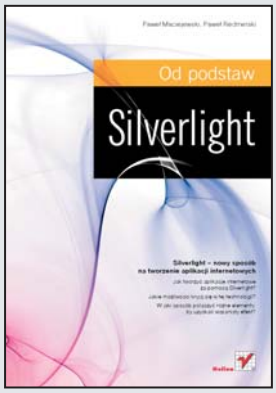
Silverlight. Od podstaw



Autor: Paweł Maciejewski, Paweł Redmerski
ISBN: 978-83-246-1984-9
Tytuł oryginału: COM+ Developer's Guide
Format: 158x235, stron: 208

Silverlight – nowy sposób na tworzenie aplikacji internetowych

- Jak tworzyć aplikacje internetowe za pomocą Silverlight?
- Jakie możliwości kryją się w tej technologii?
- W jaki sposób połączyć różne elementy, by uzyskać wspaniały efekt?

Technologia Silverlight, opracowana przez Microsoft, jest propozycją dla tych, którzy przymierzają się do projektowania aplikacji internetowych lub zajmują się tym już od jakiegoś czasu, a chcieliby poszerzyć spektrum swoich umiejętności.
Do podstawowych zalet tej technologii należą prostota jej używania oraz ogromna elastyczność – oprócz języka XAML, służącego do opisania interfejsu użytkownika, twórca aplikacji może posługiwać się dowolnym językiem programowania, dostępnym dla platformy .NET. Gotowe programy przygotowane w Silverlight są kompatybilne z większością przeglądarek, zainstalowanych na różnych platformach.

Książka „Silverlight. Od podstaw" zawiera wszelkie informacje na temat najnowszej wersji tej technologii. Znajdziesz tu opis tego środowiska programistycznego, jego funkcjonalności oraz narzędzi, które musisz zainstalować, by móc sprawnie korzystać z Silverlight. Poznasz także zasady tworzenia nowych projektów i całą masę przykładów konkretnych rozwiązań. Wszystko to uzupełnione zostało licznymi zrzutami ekranu, ilustrującymi niemal każdy krok oraz efekty działania podanego kodu.
Jeśli chcesz nauczyć się projektować aplikacje internetowe w Silverlight, nie znajdziesz lepszej książki!

- Zasady działania technologii Silverlight
- Silverlight a platforma .NET
- Programowanie w języku XAML
- Używanie kontrolek interfejsu
- Pozycjonowanie za pomocą kontenerów
- Wykorzystywanie grafiki
- Integracja z multimediami
- Obsługa zdarzeń
- Tworzenie animacji
- Stosowanie stylów i szablonów

Otwórz się na nowe możliwości. Wypróbuj Silverlight!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.

Kontenery i pozycjonowanie

W poprzednim rozdziale zostały omówione szczegółowo kontrolki dostępne w Silverlighcie. Znajomość szerokiego wachlarza kontrolek, które oferuje technologia, jest niezmierne ważna, ale nie wystarczy do napisania aplikacji silverlightowej z poprawnym (przyjaznym dla użytkownika) interfejsem. Aby było to możliwe, należy rozumieć, jak działa *pozycjonowanie* elementów wizualnych w technologii Silverlight, które jest tematem tego rozdziału.

Pozycjonowanie kontrolek odbywa się przy pomocy tzw. *kontenerów* (ang. *containers*). Są one niewidocznymi obiektami, których jedynym zadaniem jest pozycjonowanie kontrolek (lub innych kontenerów). Tylko elementy będące wewnątrz kontenera mogą być przezeń *pozycjonowane* (patrz: rozdział 2., „Drzewo zależności"). Pozycjonowanie elementów odbywa się zatem m.in. przez wkładanie ich do kontenerów. To jednak nie wszystko. Kontener musi mieć określone pewne własności, które determinują sposób, w jaki pozycjonuje on elementy w jego wnętrzu. W końcu pozycjonowany element może mieć pewne dołączone atrybuty, które wpływają na jego pozycję. Podsumowując, pozycję obiektu określają:

- zależność *rodzic-dziecko* (element pozycjonowany musi być wewnątrz kontenera),
- właściwości kontenera (np. `Orientation` panelu stosu),
- *atrybuty dołączone* do elementu pozycjonowanego (np. `Canvas.Left`).


Uwaga

W polskiej literaturze można się spotkać z tłumaczeniem słowa „containers" jako „pojemniki". W naszej książce pozostaniemy przy określeniu „kontenery".

W pierwszej kolejności omówimy metody pozycjonowania elementów graficznych w Silverlighcie. Następnie omówimy zasady pozycjonowania *absolutnego* i *relatywnego*. Obydwie zasady pozycjonowania elementów sprowadzają się w większości przypadków do wybrania właściwego kontenera oraz nadania odpowiednich wartości jego własnościom (a także własnościom dołączonym do jego dzieci). Z racji tego, że w Silverlighcie programista ma dostęp do trzech różnych kontenerów:

- *panelu stosu* `StackPanel`,
- *siatki* `Grid`,
- *płótna* `Canvas`,

omówimy po kolei każdy z nich oraz pokażemy najbardziej typowe zastosowania.


Wskazówka

Silverlight posiada tylko trzy kontenery, podczas gdy WPF ma ich aż siedem (`StackPanel`, `Grid`, `Canvas`, `UniformGrid`, `WrapPanel`, `DockPanel`, `TabControl`). Wraz z rozwojem technologii Silverlight powinny się w niej pojawiać kontenery, których brakuje w stosunku do WPF.

Pozycjonowanie relatywne

Marginesy

Każdy obiekt klasy `FrameworkElement` posiada własność `Margin`, czyli margines. Przypisując margines obiektowi, programista określa, jaki odstęp dzieli go od innych elementów interfejsu.


Uwaga

Obiekty typu `FrameworkElement` posiadają własność `Margin`, ale nie mają własności `Padding`. Istnieje natomiast kontrolka (opisana w rozdziale 3.), która posiada zarówno tę, jak i inne własności mające wpływ na wyświetlanie elementów graficznych na ekranie użytkownika. Tą kontrolką jest `Border`, która na liście członków klasy posiada własności: `Margin`, `Padding`, `Background`, `BorderBrush`, `BorderThickness` i inne, dzięki którym programista ma większą kontrolę nad tym, w jaki sposób są wyświetlane elementy wizualne Silverlighta.

Podczas korzystania z marginesów należy pamiętać o trzech rzeczach:

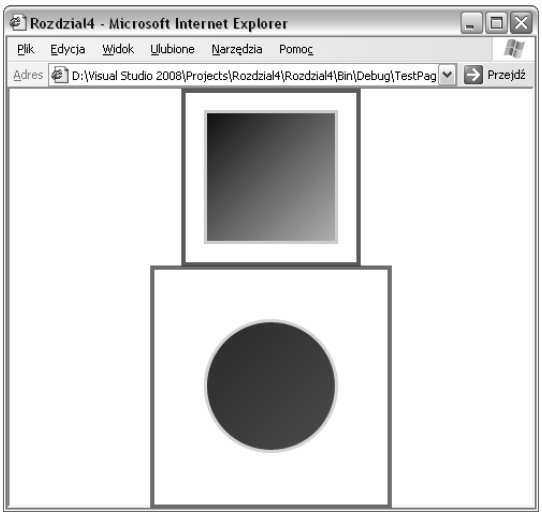
1. addtywności,
2. sposobach zapisu,
3. ujemnych marginesach.

Addytywność

Na rysunku 4.1 mamy pokazany zrzut ekranu przykładowej aplikacji (źródło można znaleźć w solucji */Rozdzial4.sln* w pliku *Margin.xaml*).

Na rysunku 4.1 kwadrat posiada margines równy `20` pikselom, natomiast koło ma margines równy `50` pikselom. **Addytywność** marginesów przejawia się w tym, że są one do siebie dodawane podczas obliczania pozycji elementu graficznego na ekranie użytkownika. Dlatego odstęp pomiędzy kwadratem i kołem na rysunku 4.1 jest równy w sumie `70` pikselom.

Rysunek 4.1. *Addywne działanie marginesów*



Różne sposoby zapisu

Margines jest strukturą typu `Thickness`, którą już pokrótce omawialiśmy (patrz rozdział 2.). Zgodnie z tym, co wtedy wytłumaczyliśmy, marginesy można definiować w XAML-u na trzy różne sposoby. W ramach przypomnienia wszystkie zostały pokazane na listingu 4.1 (linijki od 1. do 3.).

Listing 4.1. *Przykłady definiowania marginesu*

```
1 <StackPanel x:Name="LayoutRoot" Background="White" Margin="30">
2    <Rectangle Width="100" Height="100" Fill="Red" Margin="20 40" />
3    <Ellipse Width="100" Height="100" Fill="Blue" Margin="50 10 5 15" />
4 </StackPanel>
```

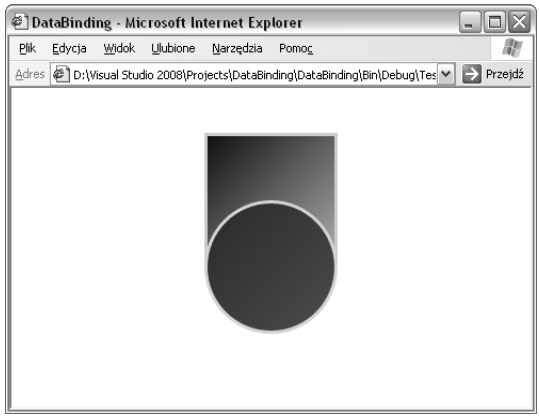
Ujemne marginesy

Czymś, o czym do tej pory jeszcze nie wspominaliśmy, są ujemne marginesy. Otóż w Silverlighcie istnieje możliwość definiowania marginesów o ujemnych wartościach. Tak jak dodatnie marginesy zwiększają odstęp pomiędzy elementami interfejsu, tak marginesy ujemne go zmniejszają. Jeśli ujemny margines jest większy niż odległość dzieląca elementy, wtedy zaczynają one na siebie zachodzić (jeden przykrywa drugi), tak jak na rysunku 4.2.

Wyrównanie elementów

Każdy obiekt typu `FrameworkElement` posiada dwie własności:

- `HorizontalAlignment`,
- `VerticalAlignment`,

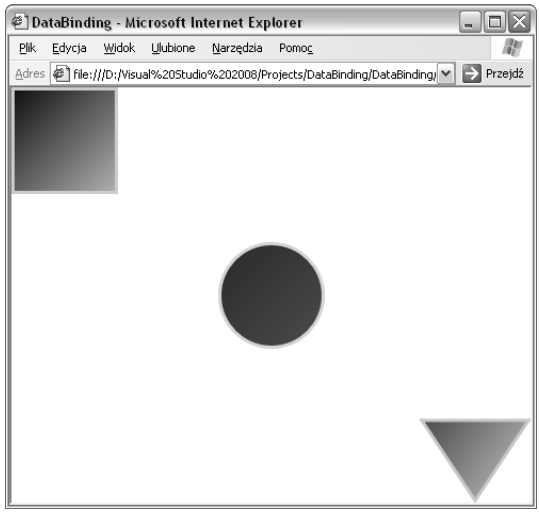


Rysunek 4.2. *Nachodzenie na siebie elementów — wynik zastosowania ujemnych marginesów*

dzięki którym programista może określić jego wyrównanie. Własność `HorizontalAlignment` określa wyrównanie w poziomie, natomiast `VerticalAlignment` — wyrównanie w pionie. Na listingu 4.2 można zobaczyć użycie tych własności w XAML-u, a rysunek 4.3 przedstawia efekt ich działania.

Listing 4.2. *Użycie podstawowych wartości własności HorizontalAlignment i VerticalAlignment*

```
<Grid x:Name="LayoutRoot" Background="White">
   <Rectangle Width="100" Height="100"
      HorizontalAlignment="Left" VerticalAlignment="Top"
      Stroke="LightBlue" StrokeThickness="3">
         <Rectangle.Fill>
            <LinearGradientBrush>
               <GradientStop Offset="0" Color="DarkBlue" />
               <GradientStop Offset="1" Color="Cyan" />
            </LinearGradientBrush>
         </Rectangle.Fill>
   </Rectangle>
   <Ellipse Width="100" Height="100"
      HorizontalAlignment="Center" VerticalAlignment="Center"
      Stroke="Pink" StrokeThickness="3">
      <Ellipse.Fill>
         <LinearGradientBrush>
            <GradientStop Offset="0" Color="DarkRed" />
            <GradientStop Offset="1" Color="Red" />
         </LinearGradientBrush>
      </Ellipse.Fill>
   </Ellipse>
   <Polygon Points="50,75 0,0 100,0"
      HorizontalAlignment="Right" VerticalAlignment="Bottom"
      Stroke="LightGreen" StrokeThickness="3">
      <Polygon.Fill>
         <LinearGradientBrush>
            <GradientStop Offset="0" Color="Green" />
            <GradientStop Offset="1" Color="LightGreen" />
         </LinearGradientBrush>
      </Polygon.Fill>
   </Polygon>
</Grid>
```



Rysunek 4.3. *Demonstracja działania podstawowych wartości własności pozycjonujących*

Własność `HorizontalAlignment` posiada wartości:

- `Left` — wyrównanie do lewej strony,
- `Center` — wycentrowanie w poziomie,
- `Right` — wyrównanie do prawej strony,
- `Stretch` — wyrównanie w pionie.

Analogicznie, `VerticalAlignment` przyjmuje wartości:

- `Top` — wyrównanie do góry,
- `Middle` — wyśrodkowanie w pionie,
- `Bottom` — wyrównanie do dołu,
- `Stretch` — wyrównanie w poziomie.

W kodzie programu z listingu 4.2 nie użyliśmy jednej z wartości, która jest wspólna dla obu własności (zarówno `HorizontalAlignment`, jak i `VerticalAlignment`). Mowa tutaj o wartości `Stretch`.

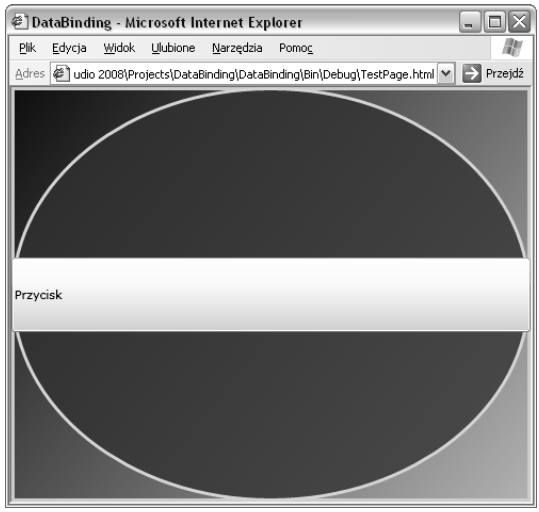
`Stretch` jest wartością, która mówi, aby obiekt został rozciągnięty w danym kierunku. Innymi słowy, przypisanie obiektowi własności `HorizontalAlignment="Stretch"` sprawi, że zabierze on całe dostępne miejsce w poziomie. Listing 4.3 pokazuje wykorzystanie wartości `Stretch` (rysunek 4.4 przedstawia zrzut ekranu aplikacji).

Listing 4.3. *Wykorzystanie własności Stretch*

```
1 <Grid x:Name="LayoutRoot" Background="White">
2    <Rectangle HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
3       Stroke="LightBlue" StrokeThickness="3">
4       <Rectangle.Fill>
5          <LinearGradientBrush>
```

Listing 4.3. *Wykorzystanie własności Stretch — ciąg dalszy*

```
            <GradientStop Offset="0" Color="DarkBlue" />
            <GradientStop Offset="1" Color="Cyan" />
         </LinearGradientBrush>
      </Rectangle.Fill>
   </Rectangle>
   <Ellipse Stroke="Pink" StrokeThickness="3">
      <Ellipse.Fill>
         <LinearGradientBrush>
            <GradientStop Offset="0" Color="DarkRed" />
            <GradientStop Offset="1" Color="Red" />
         </LinearGradientBrush>
      </Ellipse.Fill>
   </Ellipse>
   <Button Content="Przycisk" Height="70"
      HorizontalContentAlignment="Stretch"
      VerticalAlignment="Stretch" />
</Grid>
```



Rysunek 4.4. *Działanie wartości Stretch*

Proszę zwrócić uwagę na to, że w linijce 11. listingu 4.3 elipsa nie ma nadanej ani własności `HorizontalAlignment`, ani `VerticalAlignment`, a pomimo tego zostanie ona rozciągnięta (jak widać na rysunku 4.4). Dzieje się tak dlatego, że wartość `Stretch` jest domyślną wartością zarówno dla `HorizontalAlignment`, jak i `VerticalAlignment`.

Własności `Width` i `Height` obiektów typu `FrameworkElement` mają wyższy priorytet niż wartość `Stretch` własności pozycjonujących. W przypadku gdy określimy wielkość obiektu w pionie lub poziomie, wartość `Stretch` nie ma znaczenia. Tak też jest w przypadku przycisku z listingu 4.3 (linijka 19. i 20.). Ma on określoną wysokość, dlatego zajmuje całe miejsce w poziomie.

StackPanel

`StackPanel`, czyli inaczej panel stosu, to najprostszy z kontenerów dostępnych w Silverlighcie. Elementy znajdujące się w jego wnętrzu są rozmieszczane wzdłuż jednej z prostych (albo pionowej, albo poziomej), którą nazywa się *orientacją*.

Elementy znajdujące się wewnątrz panelu stosu są wyświetlane w określonej kolejności. W przypadku orientacji poziomej elementy są wyświetlane od strony lewej do prawej. Gdy `StackPanel` ma orientację pionową, wtedy wyświetlane są od góry do dołu.

Określenie orientacji panelu

Własność `Orientation` określa orientację panelu stosu i może przyjmować dwie wartości:

- `Vertical` (domyślnie) — orientacja pionowa („jeden element pod drugim”),
- `Horizontal` — orientacja pozioma („jeden element obok drugiego”).

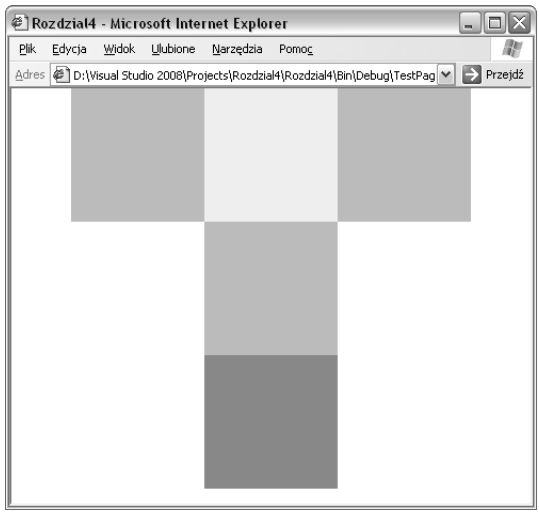
W solucji */Rozdzial4.sln* w pliku */StackPanelOrientation.xaml* można znaleźć kod przykładowej aplikacji. Listing 4.4 pokazuje fragment kodu tego programu, w którym widać działanie panelu stosu z wykorzystaniem własności `Orientation`. Rysunek 4.5 przedstawia zrzut ekranu aplikacji.

Listing 4.4. *Użycie własności Orientation panelu stosu*

```
<StackPanel x:Name="LayoutRoot" Background="White">
   <StackPanel Orientation="Horizontal">
      <Rectangle Width="125" Height="125" Fill="#BBBBBB" />
      <Rectangle Width="125" Height="125" Fill="#EEEEEE" />
      <Rectangle Width="125" Height="125" Fill="#BBBBBB" />
   </StackPanel>
   <StackPanel >
      <Rectangle Width="125" Height="125" Fill="#BBBBBB" />
      <Rectangle Width="125" Height="125" Fill="#888888" />
   </StackPanel>
</StackPanel>
```

Jak można zobaczyć na listingu 4.4, w przykładowej aplikacji mamy trzy panele stosu:

- Panel nadrzędny o nazwie `LayoutRoot` — przechowuje inne dwa panele stosu. Pozycjonuje je wzdłuż pionowej linii („jeden pod drugim”). Nie ma potrzeby definiowania własności `Orientation` (jej domyślna wartość to `Vertical`).
- Panel stosu z *trzema* poziomymi kwadratami — ma określoną przez programistę wartość `Orientation=„Horizontal”`.
- Panel stosu z *dwoma* kwadratami. W tym panelu również nie określa się parametru `Orientation` ze względu na wartość domyślną.



Rysunek 4.5. *StackPanel — użycie własności Orientation*

W gruncie rzeczy panele stosu są bardzo prostym narzędziem. Proszę jednak zwrócić uwagę na to, że w powyższym przykładzie użyliśmy trzech paneli stosu, zagnieżdżając dwa z nich w jednym nadrzędnym. Jak na tak prosty zabieg zaowocowało to ciekawym efektem. W dalszej części książki Czytelnik będzie miał okazję bardzo często spotykać się z podobnym użyciem paneli stosu, ich siła bowiem tkwi w tym, że można je łatwo (wręcz intuicyjnie) zagnieżdżać oraz łączyć z innymi kontenerami.

Grid

Siatka `Grid` jest najwszechstronniejszym kontenerem dostępnym w technologii Silverlight. Bez niej projektowanie interfejsów byłoby o wiele bardziej uciążliwe. Aby z niej skorzystać, programista najpierw definiuje siatkę, określając liczbę wierszy i kolumn (a także ich rozmiary). Efekt końcowy przypomina trochę to, co można zobaczyć po otwarciu pustego arkusza kalkulacyjnego. Pozycjonowanie za pomocą siatki polega po prostu na wkładaniu elementów (kontrolek lub innych kontenerów) do jej komórek. W tym podrozdziale omówimy zarówno etap projektowania siatki i wkładania do niej elementów, jak i dodatkowe możliwości, takie jak choćby scalanie ze sobą komórek.

Definiowanie wierszy i kolumn

Definiowanie wierszy i kolumn siatki odbywa się poprzez nadanie jej odpowiednich właściwości. Definicja tych własności (mowa tutaj konkretnie o właściwościach `Column` ↪`Definitions` i `RowDefinitions`) nie może być jednak realizowana poprzez *składnię atrybutową*. Dzieje się tak ze względu na to, że teoretycznie programista może określić dowolną liczbę wierszy i kolumn. Ponadto, jak się za moment okaże, kolumna (czy też wiersz) może być określona nie tylko jedną liczbą (która charakteryzuje jej rozmiar), ale także paroma innymi wielkościami (choćby takimi jak minimalny i maksymalny rozmiar).

Dlatego właśnie definicja zarówno kolumn, jak i wierszy odbywa się przy pomocy *składni elementów właściwości*.

Jak już zostało powiedziane, chodzi tutaj o dwie własności (listing 4.5 przedstawia ich użycie):

- `ColumnDefinitions` — definicje kolumn,
- `RowDefinitions` — definicje wierszy.

Listing 4.5. *Własności ColumnDefinitions i RowDefinitions*

```
 1 <Grid x:Name="LayoutRoot" ShowGridLines="True">
 2    <Grid.ColumnDefinitions>
 3       <ColumnDefinition Width="100" />
 4       <ColumnDefinition Width="200" />
 5       <ColumnDefinition Width="100" />
 6 </Grid.ColumnDefinitions>
 7    <Grid.RowDefinitions>
 8       <RowDefinition Height="50" />
 9       <RowDefinition Height="150" />
10       <RowDefinition Height="50" />
11    </Grid.RowDefinitions>
12 </Grid>
```

Wewnątrz własności `ColumnDefinitions` muszą się znaleźć elementy typu `Column` ↪`Definition`. Jak wskazuje nazwa, są to obiekty reprezentujące w Silverlighcie kolumnę. `ColumnDefinition` posiada m.in. własności:

- `Width` — szerokość początkowa kolumny,
- `MinWidth` — minimalna szerokość kolumny,
- `MaxWidth` — maksymalna szerokość kolumny.

Własności obiektu `RowDefinition`, który opisuje wiersz, są analogiczne.

Wielkości kolumn i wierszy

W Silverlighcie mamy dwie możliwości określania wysokości wierszy czy szerokości kolumny:

- *statyczne*,
- *procentowe*.

W listingu 4.5, w trzeciej linijce, mamy najprostszą z możliwych definicję własności `Width`:

```
Width="100"
```

która mówi, że szerokość kolumny wynosi `100` pikseli. Jest to wielkość *statyczna*, tzn. podana jawnie w pikselach, w praktyce najczęściej stosowana.

Podczas pracy z siatkami bardzo szybko się okaże, że statyczne określanie wielkości jest niewystarczające i trzeba sięgnąć po wielkości *procentowe*. Wielkość wyrażona procentowo określa, ile procent dostępnego dla elementu graficznego miejsca może on zabrać. W Silverlighcie 100% symbolizuje gwiazdka *. Aby kolumna zajmowała całe dostępne jej miejsce, należy napisać:

```
<ColumnDefinition Width="*" />
```

Jeśli natomiast użyjemy zapisu:

```
<ColumnDefinition Width="0.8*" />
```

wtedy kolumna zajmie 80% wolnego miejsca. Co ciekawe, równoznaczne są zapisy:

```
<ColumnDefinition Width="8*" />
<ColumnDefinition Width="80*"
```

To, czy programista napisze w kodzie programu `0.8*`, czy `80*`, jest tylko kwestią jego wyboru.

Wkładanie elementów do komórek

Gdy programista ma już określoną siatkę za pomocą definicji wierszy i kolumn, wtedy może wkładać elementy do poszczególnych komórek. Podobnie jak to miało miejsce w przypadku panelu stosu, pozycjonowany element musi być dzieckiem siatki w drzewie zależności. Aby określić, w której komórce ma się znajdować pozycjonowany obiekt, programista musi dołączyć do niego atrybuty (patrz rozdział 2., „Atrybuty dołączone"):

- `Grid.Column` — indeks kolumny,
- `Grid.Row` — indeks wiersza.


Wskazówka

Indeksy kolumn i wierszy są numerowane w Silverlighcie od zera. Na przykład, mając zdefiniowane dwie kolumny w *siatce*, indeksem pierwszej z nich będzie `0`, natomiast drugiej `1`.

Włożenie przycisku do drugiej kolumny i drugiego wiersza siatki zdefiniowanej kodem z listingu 7.2 wygląda następująco:

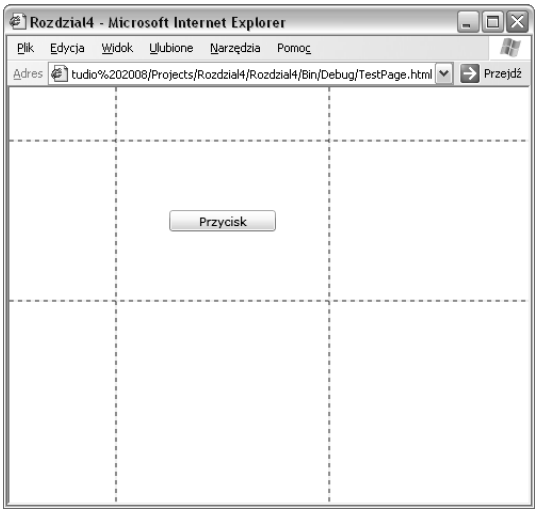
```
<Button Grid.Column="1" Grid.Row="1" Content="Przycisk" Width="100" Height="20" />
```

W solucji */Rozdzial4.sln* w pliku *ColumnsAndRowsDefinitions.xaml* można znaleźć przykładową aplikację. Definiuje ona siatkę z listingu 4.5 oraz umieszcza w niej przycisk. Zrzut ekranu aplikacji przedstawia rysunek 4.6.


Wskazówka

Jeśli piszemy w programie pierwszy obiekt siatki, to często nie ma nadanych atrybutów `Grid.Row` i `Grid.Column`. To dlatego, że w przypadku gdy nie dołączymy atrybutów `Grid.Row` i `Grid.Column` do obiektu, który znajduje się wewnątrz siatki, wtedy zostanie on umieszczony w pierwszej kolumnie i pierwszym wierszu (czyli tak, jakby miał dołączone atrybuty `Grid.Row="0"` i `Grid.Column="0"`).

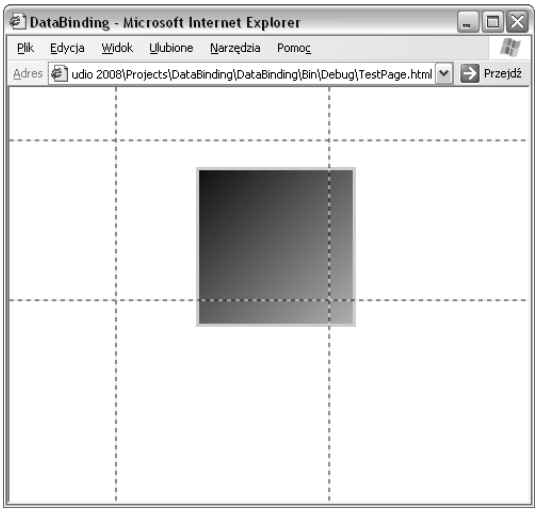
Rysunek 4.6.
Pozycjonowanie za pomocą siatki Grid



Scalanie komórek

Posługując się siatką, w Silverlighcie istnieje możliwość scalania ze sobą komórek. Dzięki temu możliwe jest osiągnięcie efektów, takich jak na rysunku 4.7.

Rysunek 4.7.
Możliwości, jakie daje scalanie komórek



Scalanie realizowane jest przez dwa atrybuty dołączone:

- `Grid.ColumnSpan`,
- `Grid.RowSpan`,

które przyjmują jako wartość liczbę całkowitą określającą ilość scalanych wierszy/kolumn. Dla przykładu, kwadrat z powyższego rysunku scala dwie komórki w pionie i w poziomie.

Informacje o scaleniu są dołączane do obiektu (podobnie jak atrybuty `Grid.Row` i `Grid.Column`), a nie definiowane wraz z kolumnami i wierszami. Proszę zwrócić uwagę na to, że dzięki temu istnieje możliwość nadania obiektom znajdującym się w tej samej komórce różnego scalania, uzyskując przy tym ciekawe efekty.

Inne właściwości siatki

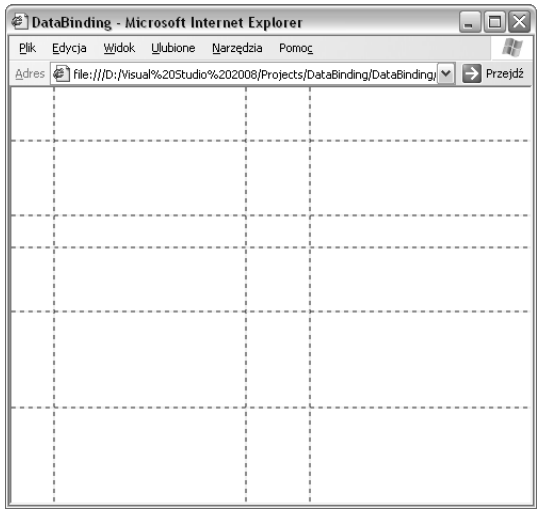
Pisaliśmy już o definiowaniu kolumn i wierszy siatki, co w gruncie rzeczy jest nadawaniem wartości odpowiednim własnościom klasy `Grid`. Rzecz jasna nie są to jedyne własności siatek (chociaż prawdopodobnie najważniejsze). Na koniec wypada nam pokrótce omówić pozostałe wyróżniające się właściwości siatki.

ShowGridLines

`ShowGridLines` to własność, która znacznie ułatwia pracę programisty z siatkami. Przyjmuje ona wartość logiczną:

- `true` — na siatce są pokazywane linie pomocnicze w miejscach, w których przebiegają granice kolumn i wierszy,
- `false` (domyślnie) — ukrywa linie pomocnicze.

Rysunek 4.8 przedstawia pustą siatkę z właściwością `ShowGridLine` ustawioną na `true`.



Rysunek 4.8. *Efekt włączenia własności ShowGridLines*

Width i Height

`Width` i `Height` to z pozoru trywialne właściwości, które określają rozmiary elementu graficznego. W przypadku wszystkich kontenerów (nie tylko siatki) te dwie niepozorne własności mają jednak duże znaczenie. Przyjrzyjmy się listingowi 4.6 z kodem aplikacji, którą mieliśmy okazję oglądać na rysunku 4.8.

Listing 4.6. *Siatka z ustalonym rozmiarem*

```
 1 <Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True"
 2    Width="450" Height="350" >
 3    <Grid.ColumnDefinitions>
 4       <ColumnDefinition Width="40" />
 5       <ColumnDefinition Width="180" />
 6       <ColumnDefinition Width="60" />
 7       <ColumnDefinition />
 8    </Grid.ColumnDefinitions>
 9    <Grid.RowDefinitions>
10       <RowDefinition Height="50" />
11       <RowDefinition Height="70" />
12       <RowDefinition Height="30" />
13       <RowDefinition Height="60" />
14       <RowDefinition Height="90" />
15       <RowDefinition />
16    </Grid.RowDefinitions>
17 </Grid>
```

Pierwszą ważną cechą siatki z listingu 4.6 jest to, że ma ona określony rozmiar: `450` pikseli szerokości i `350` wysokości. Drugą ciekawą cechą jest to, że zarówno definicja ostatniej kolumny (linijka 7.), jak i definicja ostatniego wiersza (linijka 15.) nie ma określonego rozmiaru.

Jest tak dlatego, że w przypadku siatek ich rozmiar określony właściwościami `Width` i `Height` ma większe znaczenie niż definicje ich wierszy i kolumn. Oznacza to mniej więcej to, że szerokość siatki nie jest równa szerokości jej kolumn, a wysokość siatki nie jest sumą wysokości wierszy. Z tego faktu wynika kilka konsekwencji, o których trzeba pamiętać, używając siatek.

Pierwszą z nich jest to, że definicje *ostatnich* kolumn/wierszy mogą nie mieć określonej szerokości/wysokości. Niezależnie od tego, jaki nadamy rozmiar *ostatniej* kolumnie i wierszowi, ich rozmiar zostanie tak dobrany, aby siatka miała rozmiar zdefiniowany przez `Width` i `Height`. Dlatego rozmiar *ostatniej* kolumny czy wiersza nie ma znaczenia. Drugą konsekwencją jest natomiast to, że jeśli zdefiniujemy *za mały* rozmiar siatki, to kolumny i wiersze, które wykraczają poza jej granice, nie będą widoczne.


Uwaga

Niezależnie od tego, co zrobimy z ostatnią kolumną/wierszem, siatka i tak będzie tak szeroka i tak wysoka, jak to określiliśmy właściwościami `Width` i `Height`. W przypadku gdy nie określimy własności `Width` i `Height`, siatka najpewniej zabierze całe dostępne dla niej miejsce i będzie się zachowywała dalej tak, jakby jej `Width` i `Height` były określone.

Takie zachowanie siatki może się wydawać początkowo nieintuicyjne. W praktyce jednak jest ono przydatne, programista bowiem, ustalając rozmiar siatki, ma gwarancję, że elementy przez nią pozycjonowane nie zmienią jej rozmiaru. Taka gwarancja ma dużą wartość, w przypadku gdy programista nie ma pewności, co się może znaleźć w kontenerze (np. gdy w siatce są dynamicznie wkładane informacje pochodzące z bazy danych).

Canvas

Jak już zostało kilka razy wspomniane, w Silverlighcie programista ma do dyspozycji trzy kontenery. Dwa z nich, które służą do *pozycjonowania relatywnego*, zostały już omówione. Ostatni, kontener `Canvas` (ang. *płótno*), jest jedynym narzędziem w technologii Silverlight służącym do *pozycjonowania absolutnego*.

Można powiedzieć, że pozycjonowanie absolutne pozwala programiście na określenie współrzędnych obiektu pozycjonowanego. Przy użyciu narzędzi pozycjonowania relatywnego (panelu stosu i siatki) nie ma takiej możliwości. W tym podrozdziale skupimy się na omówieniu sposobu korzystania z *płótna*.

Z racji tego, że w tym przypadku nie ma potrzeby definiowania ani kolumn, ani wierszy (określamy tylko współrzędne elementów pozycjonowanych), kontener `Canvas` nie posiada tylu ciekawych właściwości, co choćby siatka `Grid`. Posługiwanie się płótnem sprowadza się do dołączania odpowiednich atrybutów do jego dzieci (w celu określenia m.in. współrzędnych). Mowa tutaj o trzech atrybutach dołączonych:

- `Canvas.Left`,
- `Canvas.Right`,
- `Canvas.ZIndex`.

Przejdziemy teraz do pokazania, jak korzystać z tych atrybutów.

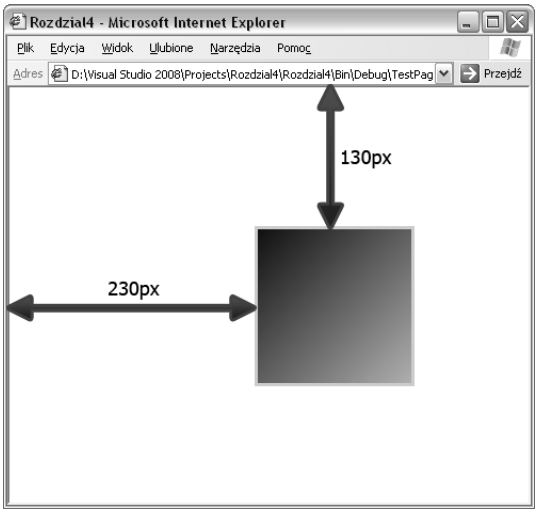
Canvas.Left i Canvas.Right

Te dwie własności służą do określenia odległości pozycjonowanego obiektu od lewego górnego boku kontenera. Listing 4.7 pokazuje użycie tych dwóch atrybutów dołączonych. W linijce 2. są dołączone do kwadratu tak, aby jego pozycja na płótnie wynosiła `130` pikseli od góry i `230` pikseli od lewej strony.

Listing 4.7. *Użycie własności* `Canvas.Left` *i* `Canvas.Right` *do pozycjonowania kwadratu*

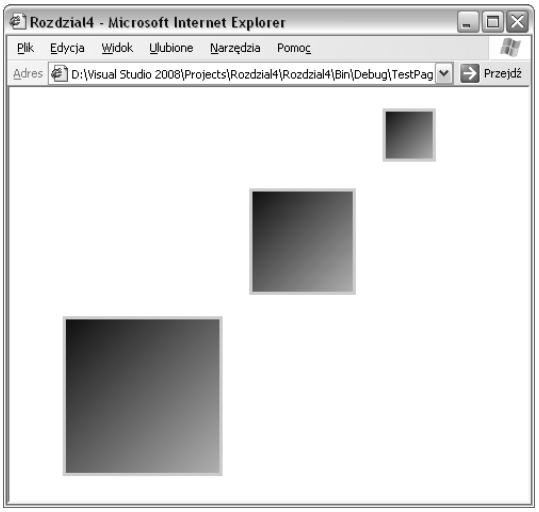
```
 1 <Canvas>
 2    <Rectangle Canvas.Left="230" Canvas.Top="130"
 3       Width="150" Height="150"
 4       Stroke="LightBlue" StrokeThickness="3">
 5       <Rectangle.Fill>
 6          <LinearGradientBrush>
 7             <GradientStop Offset="0" Color="DarkBlue" />
 8             <GradientStop Offset="1" Color="Cyan" />
 9          </LinearGradientBrush>
10       </Rectangle.Fill>
11    </Rectangle>
12 </Canvas>
```

W solucji */Rozdzial4.sln* w pliku */CanvasLeftRight.xaml* można znaleźć kod aplikacji, której fragment kodu został przedstawiony na listingu 4.7. Zrzut ekranu aplikacji (wraz z naniesionymi strzałkami pomocniczymi) przedstawia rysunek 4.9.



Rysunek 4.9. *Pozycjonowanie kwadratu za pomocą płótna*

Płótno `Canvas`, dzięki możliwości dokładnego określenia współrzędnych pozycjonowanego obiektu (za pomocą atrybutów dołączonych `Canvas.Left` i `Canvas.Right`), daje programiście największą swobodę pozycjonowania. Każdy obiekt wewnątrz płótna, niezależnie od tego, którym w kolejności jest dzieckiem, jest zawsze pozycjonowany względem lewego, górnego boku płótna. Dzięki temu można w mgnieniu oka napisać aplikację, która wygląda np. tak, jak ta przedstawiona na rysunku 4.10 (kod programu można znaleźć w solucji */Rozdzial4.sln* w pliku *CanvasLeftRight2.xaml*).



Rysunek 4.10. *Możliwości pozycjonowania przy pomocy płótna Canvas*

Wykonanie takiego programu, jaki został przedstawiony na rysunku 4.10, przy pomocy np. siatki jest również możliwe. W tym celu należałoby zdefiniować najpierw odpowiednie kolumny i wiersze, a następnie powkładać w nie kwadraty. Lista czynności jest tutaj o wiele dłuższa, dlatego płótno `Canvas` nadaje się do tego zadania o wiele lepiej.

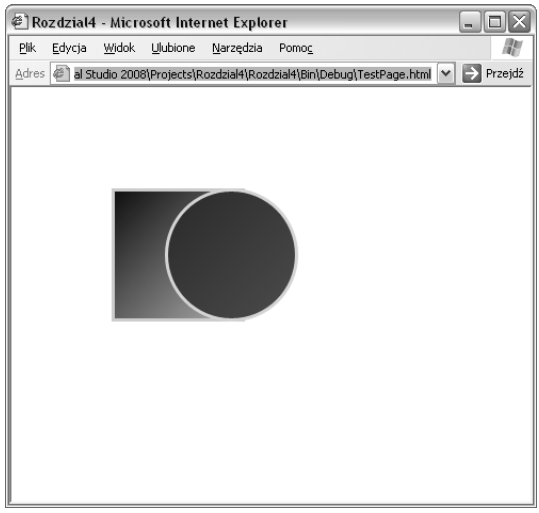
Płótno jest narzędziem stosunkowo prostym, ale potężnym. W przypadku gdybyśmy chcieli dynamicznie zmieniać pozycje kwadratów z rysunku 4.10 (np. w zależności od tego, gdzie na ekranie znajduje się myszka), wystarczy tylko odpowiednio zmieniać własności atrybutów do nich dołączonych.

Pomimo tych wszystkich zalet samo płótno najczęściej nie przyda nam się do pisania interfejsów — siatka jest w takich sytuacjach niezastąpiona.

Canvas.ZIndex

Podczas pozycjonowania absolutnego elementów często jest tak, że jedne zachodzą na drugie, zakrywając je w mniejszym bądź większym stopniu — taka sytuacja pokazana jest na rysunku 4.11.

Rysunek 4.11. *Nakładanie się obiektów wizualnych*

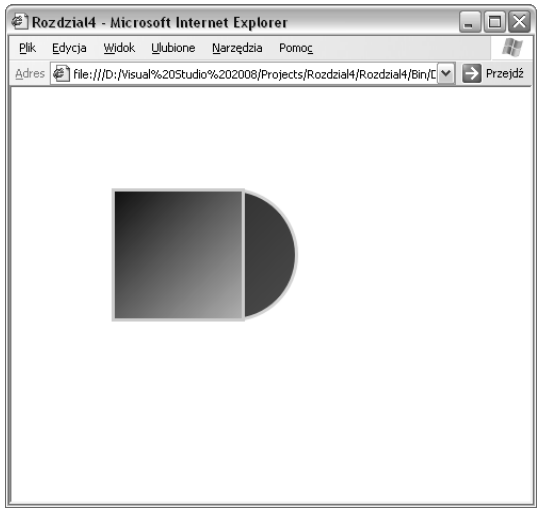


Często jest tak, że zachodzenie na siebie elementów nie jest wypadkiem przy pracy, a efektem zamierzonym przez programistę. Dlatego w Silverlighcie istnieje możliwość określenia kolejności wyświetlania elementów, tak aby programista miał nad nim pełną kontrolę. Koncepcja nie jest może i nowa, ale prosta i skuteczna. Polega na tym, że elementom graficznym znajdującym się na kanwie można przypisać liczbę naturalną, która determinuje to, jako który zostanie on wyświetlony. Obiekty o numerach wyższych są wyświetlane „ponad" elementami o numerach niższych. Jeśli zatem chcemy, aby kwadrat z rysunku zawsze znajdował się nad kołem (niezależnie od jego miejsca w kodzie programu), wtedy napiszemy kod programu z listingu 4.8.

Listing 4.8. *Uporządkowanie elementów za pomocą własności* `Canvas.ZIndex`

```
<Canvas>
   <Rectangle Canvas.Left="50" Canvas.Top="50"
      Canvas.ZIndex="1"
      Width="125" Height="125"
      Stroke="LightBlue" StrokeThickness="3">
      <Rectangle.Fill>
         <LinearGradientBrush>
            <GradientStop Offset="0" Color="DarkBlue" />
            <GradientStop Offset="1" Color="Cyan" />
         </LinearGradientBrush>
      </Rectangle.Fill>
   </Rectangle>
   <Ellipse Canvas.Left="100" Canvas.Top="50"
      Width="125" Height="125"
      Stroke="Pink" StrokeThickness="3">
      <Ellipse.Fill>
         <LinearGradientBrush>
            <GradientStop Offset="0" Color="DarkRed" />
            <GradientStop Offset="1" Color="Red" />
         </LinearGradientBrush>
      </Ellipse.Fill>
   </Ellipse>
</Canvas>
```

W trzeciej linijce kodu używamy właściwości `Canvas.ZIndex`. Jest to atrybut dołączony, który służy w Silverlighcie do określenia kolejności wyświetlania elementów graficznych. Dzięki ścisłemu określeniu kolejności wyświetlania udało nam się wyświetlić kwadrat ponad kołem, jak to widać na rysunku 4.12. Pełen kod programu można znaleźć w solucji */Rozdzial4.sln* w pliku *CanvasZIndex.xaml*.



Rysunek 4.12. *Użycie atrybutu Canvas.ZIndex*

Kontenery z poziomu kodu

Mamy już wiedzę na temat tego, jak pisać w XAML-u kontenery i jak osadzać na nich kontrolki. Wiemy także, jak pozycjonować elementy znajdujące się wewnątrz kontenerów. Zdarza się jednak czasem tak, że istnieje potrzeba wygenerowania kontrolek z poziomu kodu, a następnie wyświetlenia ich użytkownikowi. Z taką sytuacją mamy do czynienia w przypadku pobierania informacji z bazy danych, a potem prezentacji ich w postaci listy.

W tym rozdziale pokażemy, jak rozwiązać trudności, które pojawiają się przy generowaniu treści z poziomu kodu aplikacji.

Generowanie kontrolek

Zanim przejdziemy do omawiania kontenerów tworzonych z poziomu kodu, pokażemy, jaka jest ogólna metoda generowania większości kontrolek interfejsu, takich jak `Button` czy `TextBox`.

Jak już zostało napisane, w języku XAML piszemy znaczniki, których własnościom możemy przypisać wartości (za pomocą składni atrybutowej bądź elementów własności). Powiedzieliśmy też, że znaczniki z pliku *.xaml* są mapowane na obiekty języka CLR, tak samo jak kod napisany w innym, dowolnym języku platformy .NET. Dlatego też pisanie kontrolek z poziomu pliku *code-behind* sprowadza się do utworzenia obiektu odpowiedniej klasy (jeśli ma to być przycisk, to tworzymy obiekt klasy `Button`) oraz nadania własnościom nowo utworzonego obiektu odpowiednich wartości. Listing 4.9 pokazuje, jak tworzy się blok tekstu `TextBlock` z pliku *code-behind*.

Listing 4.9. *Stworzenie bloku tekstu z poziomu kodu*

```
1 TextBlock t = new TextBlock();
2 t.Width = 200;
3 t.Height = 50;
4 t.TextWrapping = TextWrapping.Wrap;
5 t.TextAlignment = TextAlignment.Center;
6 t.Text = "To jest wygenerowana kontrolka interfejsu";
```

Jak widać, przypisujemy wartości własnościom, które nazywają się tak samo jak własności w XAML-u (i nie powinno być w tym nic dziwnego). Pozostaje jednak jeden problem. Tak napisanej kontrolki nie zobaczymy po uruchomieniu aplikacji Silverlight. Aby kontrolka była wyświetlana na ekranie użytkownika, musi być osadzona na kontenerze. W poniższym podrozdziale dowiemy się, jak to zrobić.

Generowanie kontenerów

Kolekcja Children

Wszystkie kontenery dziedziczą po jednej wspólnej klasie o nazwie `Panel`. Ta klasa definiuje kolekcję `Children`. Jest to kolekcja typu `UIElementCollection`, co w praktyce oznacza, że możemy do niej włożyć dowolny element wizualny Silverlighta. Może być to zarówno kontrolka interfejsu, kształt, jak i inny kontener.

Wkładając elementy wygenerowane w kodzie do kolekcji `Children` kontenera, sprawimy, że będzie on wyświetlony i pozycjonowany przez ten kontener. Przy czym należy pamiętać, że kontener wcale nie musi być utworzony dynamicznie. Listing 4.10 pokazuje kod, w którym osadzamy kontrolkę z poprzedniego listingu na siatce `LayoutRoot`.

Listing 4.10. *Włożenie kontrolki do kontenera*

```
1 TextBlock t = new TextBlock();
2 t.Width = 200;
3 t.Height = 50;
4 t.TextWrapping = TextWrapping.Wrap;
5 t.TextAlignment = TextAlignment.Center;
6 t.FontSize = 16;
7 t.Text = "To jest wygenerowana kontrolka interfejsu";
8
9 LayoutRoot.Children.Add(t);
```

Taki kod wyświetli nam kontrolkę bloku tekstu — proszę spojrzeć na rysunek 4.13, który pokazuje zrzut ekranu działającej aplikacji. Pełny kod programu można znaleźć w solucji */Rozdzial4.sln* w pliku *TextBlockFromCodeBehind.cs*.

Rysunek 4.13. *Pierwsza kontrolka umieszczona w kontenerze z poziomu kodu*

Pozycjonowanie wygenerowanych kontrolek

Pisaliśmy powyżej o tym, że aby wyświetlić kontrolki, musimy umieścić je w kolekcji `Children` kontenera. Jest to zupełnie wystarczające w przypadku panelu stosu `StackPanel` — kolejne elementy kolekcji `Children` będą prawidłowo przez niego pozycjonowane tylko dzięki temu, że `StackPanel` to najprostszy z dostępnych kontenerów. Inne kontenery są już bardziej wymagające. Aby pozycjonować element osadzony na kanwie, musimy określić dla niego własności `Left` i `Top`. Podobnie jest z siatką `Grid`, gdzie możemy podać, w której kolumnie i wierszu ma się znajdować pozycjonowany obiekt. W tym podrozdziale pokażemy, jak pozycjonować elementy graficzne Silverlighta w tych dwóch kontenerach.

Pozycjonowanie na siatce Grid

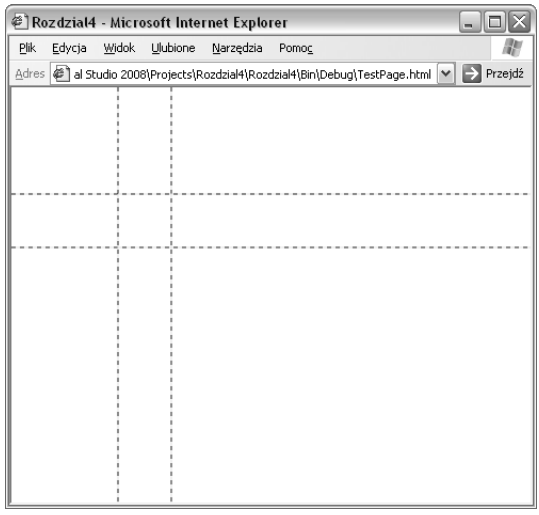
Napiszmy przykładową siatkę, taką jak pokazana na listingu 4.11.

Listing 4.11. *Siatka, na której osadzimy z poziomu kodu kontrolkę*

```
<Grid x:Name="LayoutRoot" Background="White"
  ShowGridLines="True" Loaded="LayoutRoot_Loaded" >
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="100" />
      <ColumnDefinition Width="50" />
      <ColumnDefinition Width="250" />
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
     <RowDefinition Height="100" />
     <RowDefinition Height="50" />
     <RowDefinition Height="150" />
  </Grid.RowDefinitions>
</Grid>
```

Rysunek 4.14 pokazuje, jaka jest struktura tej siatki.

Rysunek 4.14. *Siatka z listingu 4.11*



Jak widać, mamy w niej trzy kolumny i tyle samo wierszy. Będziemy teraz chcieli umieścić w drugiej kolumnie i wierszu przycisk. Jeśli mielibyśmy to zrobić w XAML-u, to musielibyśmy wewnątrz siatki umieścić znacznik:

```
<Button Width="100" Height="20" Content="Przycisk" Grid.Row ="1" Grid.Column="1" />
```

Jeśli jesteśmy natomiast po drugiej stronie, czyli po stronie pliku *code-behind*, to musimy najpierw utworzyć przycisk i dodać go do listy członków siatki, tak jak to robiliśmy w poprzednich przykładach. Następnie musimy określić kolumnę i wiersz przycisku, co nie jest prostym zadaniem z racji tego, że w XAML-u te własności są atrybutami dołączonymi. Z pomocą przychodzi tutaj klasa `Grid`. Na jej liście członków można znaleźć cztery metody statyczne:

- `void SetColumn(FrameworkElement element, int value)` — wkłada `element` do kolumny o numerze `value`,
- `void SetRow (FrameworkElement element, int value)` — wkłada `element` do wiersza o numerze `value`,
- `void SetColumnSpan (FrameworkElement element, int value)` — ustawia scalanie kolumn o wartości `value` dla obiektu `element`,
- `void SetRowSpan (FrameworkElement element, int value)` — ustawia scalanie wierszy o wartości `value` dla obiektu `element`.

Za pomocą tych metod możemy z poziomu kodu zrealizować nasze cele. Listing 4.12 pokazuje kod, za pomocą którego umieszczamy przycisk w drugiej kolumnie i drugim wierszu siatki, a także nadajemy mu scalanie dwóch kolumn.

Listing 4.12. *Umieszczenie przycisku w drugiej kolumnie i drugim wierszu siatki*

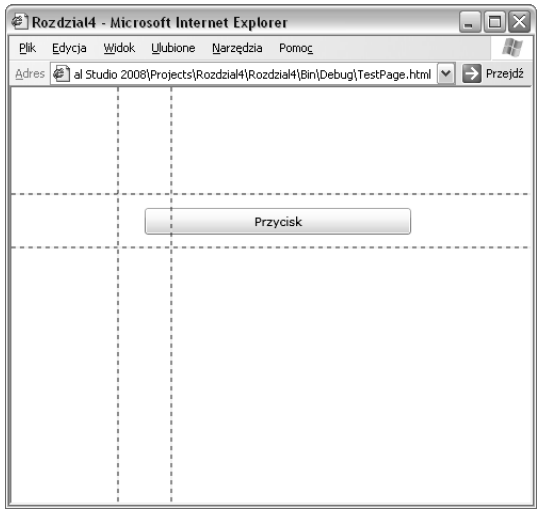
```
1 Button b = new Button();
2 b.Content = "Przycisk";
3 b.Width = 250;
4 b.Height = 25;
5 LayoutRoot.Children.Add(b);
6 Grid.SetColumn(b, 1);
7 Grid.SetRow(b, 1);
8 Grid.SetColumnSpan(b, 2);
```

Rysunek 4.15 pokazuje, że rzeczywiście udało nam się osiągnąć zamierzony efekt. Pełen kod programu można znaleźć w solucji */Rozdzial4.sln* w plikach *ButtonOnGrid FromCodeBehind.xaml* i *ButtonOnGridFromCodeBehind.xaml.cs*.

Pozycjonowanie na kanwie Canvas

W przypadku kanwy `Canvas` mamy problem podobny do tego z poprzedniego podrozdziału, gdzie własności pozycjonujące element były atrybutami dołączonymi. I tak jak w przypadku siatek, tak i tutaj rozwiązaniem są metody statyczne udostępniane poprzez klasę `Canvas`. Są to:

- `void SetLeft(UIElement element, double length)` — ustala dla obiektu `element` odsunięcie od lewej krawędzi o długość `length`,



Rysunek 4.15. *Zrzut ekranu aplikacji, która osadza przycisk na siatce z poziomu kodu*

- `void SetTop(UIElement element, double length)` — ustala dla obiektu `element` odsunięcie od górnej krawędzi o długość `length`,
- `void SetZIndex(UIElement element, int value)` — dla obiektu `element` kolejność wyświetlania jako `value`.

Napiszemy przykładowy program, w którym stworzymy dwa kwadraty, ustalimy ich pozycje na ekranie i kolejność wyświetlania. Listing 4.13 pokazuje kod pliku *.cs* tej aplikacji.

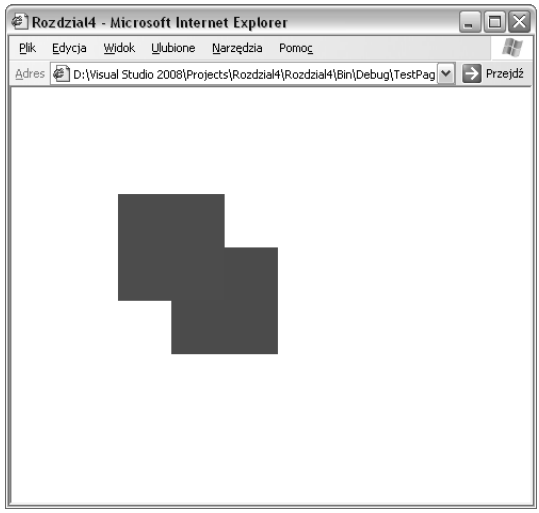
Listing 4.13. *Użycie metod pozycjonujących klasy Canvas*

```
// Zdefiniowanie kwadratów
Rectangle r1 = new Rectangle();
Rectangle r2 = new Rectangle();
r1.Width = r2.Width = 100;
r1.Height = r2.Height = 100;
r1.Fill = new SolidColorBrush(Colors.Red);
r2.Fill = new SolidColorBrush(Colors.Green);

// Osadzenie kwadratów na kanwie
LayoutRoot.Children.Add(r1);
LayoutRoot.Children.Add(r2);

// Użycie metod klasy Canvas
Canvas.SetLeft(r1, 50);
Canvas.SetTop(r1, 50);
Canvas.SetZIndex(r1, 2);
Canvas.SetLeft(r2, 100);
Canvas.SetTop(r2, 100);
Canvas.SetZIndex(r2, 1);
```

W kodzie aplikacji utworzyliśmy dynamicznie dwa kwadraty: jeden czerwony, drugi zielony. Następnie dodaliśmy je do istniejącej kanwy o nazwie `LayoutRoot`. Najpierw dodaliśmy kwadrat czerwony, a potem zielony — ma to znaczenie, ponieważ kwadrat

zielony będzie wyświetlany ponad czerwonym. Na koniec ustawiliśmy kwadraty tak, aby nachodziły na siebie. Zdefiniowaliśmy im także *z-indeks*, dzięki czemu ostatecznie kwadrat zielony będzie wyświetlany ponad czerwonym.

Kod aplikacji można znaleźć w solucji */Rozdzial4.sln* w pliku *ButtonOnCanvasFromCodeBehind.xaml.cs*. Rysunek 4.16 pokazuje zrzut ekranu działającego programu.



Rysunek 4.16. *Aplikacja demonstrująca działanie metod statycznych klasy Canvas*