

ANNA KEMPA

# WPROWADZENIE *do*



Tworzenie aplikacji w WPF przy użyciu

**XAML i C#**

Autorka: Anna Kempa – pracownik naukowo-dydaktyczny Wydziału Informatyki i Komunikacji  
Uniwersytetu Ekonomicznego w Katowicach

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik  
Projekt okładki: Jan Paluch

Fotografia na okładce została wykorzystana za zgodą Shutterstock.com

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!  
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
[http://helion.pl/user/opinie/jchata\\_ebook](http://helion.pl/user/opinie/jchata_ebook)  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:  
<ftp://ftp.helion.pl/przyklady/jchata.zip>

ISBN: 978-83-283-3699-5

Copyright © Helion 2017

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

*Tylko nieznane przeraża człowieka.  
Ale dla tego, kto staje mu naprzeciw, ono już nie jest nieznane*

(Antoine de Saint-Exupéry)



# Spis treści

<b>Wstęp .....</b>	<b>9</b>
Dla kogo jest ta książka? .....	9
Jak czytać tę książkę? .....	10
Zakres książki .....	11
<b>Rozdział 1. Przed przystąpieniem do zadań .....</b>	<b>13</b>
1.1 Instalacja środowiska i uruchomienie aplikacji WPF .....	13
1.2 Wymagany zakres znajomości języka C# .....	15
1.3 Podstawy WPF .....	21
1.4 Podstawy XAML .....	22
1.5 Sterowanie rozmiarem i pozycją elementów .....	29
<b>Rozdział 2. Pierwsza aplikacja — Przywitanie .....</b>	<b>33</b>
2.1 Warstwa prezentacji, czyli jak ma wyglądać .....	33
2.2 Code-behind, czyli jak ma działać .....	37
2.3 Zadania .....	40
2.4 Wskazówki do zadań .....	41
<b>Rozdział 3. Podstawowe kontrolki .....</b>	<b>45</b>
3.1 Kontrolki Label, TextBox, Button — aplikacja Kwadrat .....	45
3.2 Kontrolki ComboBox i CheckBox — aplikacja Rysowanie kwadratu .....	48
3.3 Zadania .....	50
3.4 Wskazówki do zadań .....	51
<b>Rozdział 4. Panele .....</b>	<b>53</b>
4.1 Canvas .....	53
4.2 StackPanel .....	54
4.3 WrapPanel .....	56
4.4 DockPanel .....	56
4.5 Grid .....	57
<b>Rozdział 5. Wiązanie danych — aplikacja Produkt .....</b>	<b>61</b>
5.1 Testowanie wiązania danych .....	61
5.2 Kod XAML .....	63
5.3 Definicja klasy Produkt i code-behind .....	65
5.4 Zadania .....	67
5.5 Wskazówki do zadań .....	68

<b>Rozdział 6. Wiązanie kolekcji danych — aplikacja Lista produktów .....</b>	<b>71</b>
6.1 Kod XAML .....	71
6.2 Definicja klasy Produkt i code-behind .....	72
6.3 Sortowanie wykazu .....	73
6.4 Formatowanie danych w wykazie .....	74
6.5 Wyrównanie tekstu w kolumnie .....	75
6.6 Filtrowanie danych .....	76
6.7 Edycja danych w nowym oknie .....	78
6.8 Zadania .....	80
6.9 Wskazówki do zadań .....	81
<b>Rozdział 7. Kontrolka DataGrid — aplikacja Edycja produktów .....</b>	<b>85</b>
7.1 Kontrolka DataGrid z autogenerowaniem kolumn .....	85
7.2 Definiowanie kolumn dla DataGrid .....	88
7.3 Kolumna DataGridComboBoxColumn .....	89
7.4 Wiązanie kontrolki DataGrid z dokumentem XML .....	90
7.5 Zadania .....	93
7.6 Wskazówki do zadań .....	94
<b>Rozdział 8. Menu — aplikacja Przeglądarka www .....</b>	<b>101</b>
8.1 Kod XAML .....	101
8.2 Code-behind .....	104
8.3 Zadania .....	107
8.4 Wskazówki do zadań .....	108
<b>Rozdział 9. Zakładki (TabControl) — aplikacja Odtwarzacz audio .....</b>	<b>111</b>
9.1 Kod XAML .....	111
9.2 Code-behind .....	113
9.3 Zadania .....	116
9.4 Wskazówki do zadań .....	116
<b>Rozdział 10. Zasoby, style i wyzwalacze .....</b>	<b>119</b>
10.1 Zasoby binarne .....	119
10.2 Zasoby logiczne .....	120
10.3 Style .....	126
10.4 Wyzwalacze .....	131
Wyzwalacze właściwości .....	132
Wyzwalacze danych .....	132
Warunki logiczne w wyzwalaczach .....	135
<b>Rozdział 11. Szablony danych, konwertery i szablony kontroltek .....</b>	<b>137</b>
11.1 Drzewo logiczne i drzewo prezentacji .....	137
11.2 Szablony danych — aplikacja Lista zadań .....	141
11.3 Konwertery wartości .....	145
11.4 Szablony kontroltek .....	147
11.5 Zadania .....	150
11.6 Wskazówki do zadań .....	151
<b>Rozdział 12. Walidacja danych .....</b>	<b>153</b>
12.1 Wbudowane mechanizmy walidacji .....	153
12.2 Definiowanie własnych reguł walidacji .....	158
12.3 Wyrażenia regularne .....	160
12.4 Zadania .....	167
12.5 Wskazówki do zadań .....	168

**Rozdział 13. Wprowadzenie do wzorca projektowego MVVM ..... 175**

13.1 Model-View-ViewModel ..... 176

13.2 Budujemy widok dla przykładowej aplikacji ..... 177

13.3 Implementacja modelu ..... 178

13.4 Implementacja modelu widoku ..... 181

13.5 Przed dalszą nauką MVVM ..... 183

**Rozdział 14. Trochę teorii na temat WPF ..... 187**

14.1 Hierarchia klas WPF ..... 187

14.2 Kontrolki ..... 189

    Kontrolki z zawartością wpisywaną do właściwości Content ..... 190

    Kontrolki z zawartością Items ..... 194

    Kontrolki tekstowe ..... 197

    Kontrolki zakresu ..... 198

    Pozostałe kontrolki ..... 199

14.3 Kierunki dalszej nauki WPF ..... 200

**Literatura ..... 203**

**Skorowidz ..... 205**





# Wstęp

**Windows Presentation Foundation** (WPF) firmy Microsoft jest jedną z wiodących technologii do tworzenia desktopowych aplikacji dla systemu Windows. Integruje interfejs użytkownika, grafikę 2D i 3D, multimedia oraz dokumenty. Umożliwia definiowanie interfejsu użytkownika w deklaratywnym języku XAML, a także pozwala na łatwą implementację wzorców projektowych, które oddzielają warstwę logiczną od warstwy prezentacji. Ponieważ jest zbudowany na bazie Direct3D, aplikacje WPF korzystają z przyspieszenia sprzętowego. Znaczącym walorem WPF jest możliwość kompozycji i adaptacji poszczególnych elementów, z których budowany jest interfejs.

Czy nauka tak zaawansowanego narzędzia jest trudna? Trud wkładany w naukę programowania (i wielu innych dziedzin) jest zazwyczaj stanem przejściowym, postrzeganym jako doraźne, tymczasowe problemy, które mogą wystąpić na każdym etapie nauki. Wraz z nabywaniem doświadczenia każdy programista uświadamia sobie, że ewentualne trudności tkwią nie tyle w stopniu skomplikowania danego narzędzia, ile w dostępności odpowiednich materiałów, w tym dokumentacji i podręczników. Starsze pokolenie programistów — które uczyło się programować jeszcze przed upowszechnieniem internetu i przy ograniczonej ofercie podręczników w księgarniach — szczególnie docenia ten aspekt.

Obecnie nie brakuje pomocy naukowych, a zróżnicowanie oferty pozwala na bardziej swobodny dobór źródeł dostosowanych do oczekiwań Czytelnika. Niniejszy podręcznik dedykowany jest osobom początkującym. Nie przedstawia wszystkich bazowych funkcjonalności dostarczanych przez WPF. Główny akcent położony jest na podstawach, umożliwiających zbudowanie biznesowej aplikacji, bez animacji i grafiki 3D. Kolejną cechą książki jest jej zadaniowy charakter — większość rozdziałów opisuje wykonanie konkretnej aplikacji.

## Dla kogo jest ta książka?

Podręcznik jest przeznaczony dla osób początkujących, które niedawno rozpoczęły naukę programowania i znają podstawy C#. Nie wymaga znajomości Windows Forms ani innych rozwiązań służących do tworzenia interfejsu graficznego. Celem książki

jest ułatwienie pierwszych kroków w zakresie technologii WPF i języka XAML. Takie przygotowanie umożliwi Czytelnikowi samodzielne wykonanie prostych aplikacji biznesowych, a ponadto utoruje drogę do dalszej nauki WPF w oparciu o bardziej zaawansowane źródła, w tym dokumentację techniczną.

Wprowadzający charakter podręcznika pozwala polecać jego lekturę studentom i uczniom szkół średnich oraz wszystkim innym osobom, które mają powody i ochotę nauczyć się WPF i nie mają dużego doświadczenia informatycznego.

## Jak czytać tę książkę?

Podręcznik został przygotowany raczej do *pracy z WPF* niż do *czytania o WPF*. Rozdział 1. wprowadza w podstawowe zagadnienia dotyczące WPF i XAML, tak aby możliwie szybko można było przejść do praktycznego etapu nauki. Większość rozdziałów opisuje wykonanie kompletnych i niezależnych aplikacji.

Tytułowe pytanie powinno zatem brzmieć: jak pracować z tą książką? Zalecane jest wykonywanie omawianych programów według podanych objaśnień. Kody programów można pobrać ze strony <http://helion.pl/pobierz-przyklady/jchata/>. Dostępne są w dwóch wersjach: do nauki oraz w postaci gotowych projektów<sup>1</sup>. Polecam korzystanie przede wszystkim z tej pierwszej wersji, przeznaczonej do nauki, która zawiera pliki tekstowe z fragmentami kodu opatrzone krótkimi komentarzami. Na podstawie wyjaśnień ujętych w treści podręcznika można z owych fragmentów, niczym z klocków, budować program. Jest to sposób znany z wielu poradników i podręczników software'owych, który dobrze sprawdza się w przypadku nauki takich technologii jak WPF. Budowanie aplikacji z przygotowanych „klocków” nie polega jednak na bezmyślnym używaniu opcji *Kopiuj* i *Wklej*. Praca taka wymaga analizy poszczególnych fragmentów kodu w oparciu o wyjaśnienia zawarte w podręczniku i w innych źródłach (zwłaszcza jeśli ktoś uczy się jednocześnie C#). Przesadne dążenie do zrozumienia od razu wszystkiego w każdym fragmencie programu też nie jest wskazane. Wiele rzeczy będzie się powtarzać w kolejnych programach, w nowych odsłonach. Ponadto będą zadania do samodzielnego wykonania, różne modyfikacje, będzie wiele okazji do tego, aby przekonać się, czy dany mechanizm bądź konstrukcja są zrozumiałe. Jak wspomniałam kody programów dostępne są także w postaci gotowych projektów, do których można zajrzeć w przypadku większych trudności z uruchomieniem swojej wersji programu.

Na końcu większości rozdziałów znajdują się podrozdziały z zadaniami i wskazówkami. Czytelnik może samodzielnie pracować nad danym programem i w razie konieczności korzystać ze szczegółowych wskazówek i propozycji rozwiązań. Zachęcam Czytelnika do przeglądania dokumentacji technicznej w trakcie tych prac w celu bliższego poznania danej klasy, jej właściwości czy metod.

---

<sup>1</sup> Wśród gotowych projektów nie ma programów dla zadań do samodzielnego wykonania. Dla takich zadań umieszczono w treści podręcznika jedynie wskazówki i szczegółowe wyjaśnienia.

Zadaniowy charakter książki daje Czytelnikowi w trakcie pracy dużo okazji do satysfakcji. Tworzone programy prezentują wizualne, od razu widoczne efekty. Zachęcam to kreatywności i doskonalenia danego programu według własnych upodobań z użyciem poznanych na danym etapie konstrukcji i mechanizmów. Opisy objaśniające wykonanie danego programu, gdyby patrzeć na nie „z boku”, to znaczy nie angażując się w proces tworzenia, mogą się wydać zbyt techniczne. Wynika to w dużej mierze ze specyfiki WPF (i innych zaawansowanych rozwiązań wspomagających tworzenie GUI), dla której trafne jest polskie przysłowie „diabeł tkwi w szczegółach”. Tworzenie interfejsu wymaga ustalenia wielu drobnych detali, których opisanie nie brzmi jak lektura do poczytania. Ponadto WPF ma trochę swoich „osobliwości”, do których należą m.in. właściwości dołączone, nazywane przez niektórych „magią WPF”. Proszę się nie obawiać — to wszystko ma swoje solidne uzasadnienie i ani się Czytelnik zorientuje, jak zacznie swobodnie używać tych i innych poznanych tu rozwiązań WPF. Podsumowując: powody do zadowolenia znajdzie Czytelnik przede wszystkim w wyniku naszej współpracy — w postaci działających programów uruchamianych na swoim komputerze, a następnie samodzielnie modyfikowanych.

## Zakres książki

W WPF wiele rzeczy można zrobić na kilka różnych sposobów. W tym podręczniku przedstawiam tylko wybrane sposoby. Wśród zastosowanych kryteriów wyboru można wymienić popularność i jakość danego rozwiązania. Niemniej w niektórych przypadkach, zwłaszcza na początku książki, uwzględniam stopień trudności, prezentując prostszy sposób. To ostatnie kryterium traci na znaczeniu wraz z zaawansowaniem nauki w kolejnych rozdziałach. Zaraz po zapoznaniu się z podstawami WPF opisanymi w rozdziale 1. można przystąpić w rozdziale 2. do wykonania pierwszej prostej aplikacji, zawierającej dwa przyciski. Pierwsze programy będą tworzone w tak zwanym widoku autonomicznym (zbliżonym nieco do Windows Forms). Takie programowanie ma swoje wady, ujawniające się w bardziej złożonych projektach, do których zalicza się brak dostatecznej elastyczności czy utrudnienia dotyczące testowania aplikacji. Zaletą tego podejścia jest natomiast większa przystępność dla osób uczących się. Można wykorzystać tę zaletę i jeszcze przed wprowadzeniem bardziej zaawansowanych zagadnień omówić podstawowe elementy WPF, które są niezależne od stylu programowania. Program realizowany w rozdziale 3. będzie wymagał zdefiniowania większej liczby kontroltek, co zrodzi potrzebę bardziej uporządkowanego podejścia do planowania interfejsu. Odpowiedź na to zapotrzebowanie przynosi rozdział 4., opisujący standardowe panele, pozwalające zarządzać układem graficznym elementów. Zaraz później, w rozdziale 5., Czytelnik dokona kolejnego ważnego kroku, mianowicie pozna wiązanie danych. Wówczas okaże się, że część mrówczej pracy wykonanej w rozdziale 3. może być zastąpiona przez ten nieoceniony mechanizm WPF. W dwóch kolejnych rozdziałach zostały przedstawione aplikacje wykorzystujące kontrolki przewidziane dla kolekcji danych — `ListView` i `DataGrid`. Rozdziały 8. i 9. prezentują możliwości kontroltek służących do tworzenia menu oraz zakładek (`Menu` i `TabControl`) na przykładzie popularnych aplikacji — przeglądarki www i odtwarzacza audio. Rozdział 10. traktuje o ważnych zagadnieniach, takich jak zasoby, style i wyzwalacze, dzięki którym definiowanie interfejsu jest bardziej wygodne i profesjonalne. Bogate możliwości WPF w zakresie

kompozycji elementów interfejsu będzie można szerzej poznać w rozdziale 11., poświęconym przede wszystkim tematyce szablonów. Gdyby nie wiązanie danych, czyli specyficzny mechanizm WPF umożliwiający automatyczne aktualizacje danych, podręcznik mógłby się w tym miejscu już zakończyć. Ale korzystanie w pełni z komfortu, jaki zapewnia to rozwiązanie, wymaga poznania sposobów wykonywania konwersji danych oraz ich walidacji. Konwertery zostały omówione w jednym z podrozdziałów rozdziału 11., natomiast walidacja danych — w rozdziale 12. Rozdział kolejny zawiera wprowadzenie do wzorca projektowego MVVM, który opiera się na mechanizmie wiązania danych i tak zwanych powiadomieniach. Zasady tego wzorca odbiegają od podejścia z zastosowaniem widoku autonomicznego. Wzorec MVVM pozwala na bardziej konsekwentną separację logiki aplikacji i sposobu wyświetlenia danych, co wpływa na zwiększenie elastyczności programu i ułatwia jego testowanie. Rozdział 14., ostatni, zawiera uporządkowanie pewnych aspektów teoretycznych, głównie poprzez omówienie hierarchii klas WPF. Spoglądanie na diagramy dziedziczenia klas niczym na mapy pozwoli Czytelnikowi wyłonić to, co poznał w tym podręczniku podczas wykonywania zadań, oraz to, co pozostało mu do poznania.

Podsumowując, zakres tej książki obejmuje: podstawy języka XAML (wykorzystywanego w WPF do deklaratywnego opisanie interfejsu), większość kontrolek (elementów wywodzących się z klasy `Control`) oraz wybrane elementy wywodzące się z klasy `FrameworkElement`. Czytelnik tej książki łatwo przyswoi mechanizmy i rozwiązania dostarczane przez WPF, takie jak wiązanie danych, wyzwalacze, konwertery, zasoby, style i szablony. Pozna także wybrane sposoby walidacji danych i podstawy wzorca projektowego MVVM. To wszystko, po dodaniu umiejętności w zakresie baz danych (będących poza tematem tego podręcznika), powinno przygotować Czytelnika do napisania aplikacji biznesowej.

Gdyby porównać naukę WPF do zwiedzania jakiegoś większego miasta, to moglibyśmy powiedzieć, że ten podręcznik (jako przewodnik) proponuje uproszczoną trasę wycieczki po ciekawszych zakątkach miasta z pomięciem kilku ważnych miejsc, które poleca podróżnikowi w przyszłości (w tym grafika 3D i animacje), oraz z pominięciem wielu mniejszych zakątków. Podręcznik ten należy zatem traktować jako pierwszą podróż i przetarcie szlaków. Mam nadzieję, że z jego pomocą będzie Czytelnikowi łatwiej zgłębiać zaawansowane możliwości WPF podczas dalszej nauki.

## Rozdział 1.

# Przed przystąpieniem do zadań

Przed przystąpieniem do wykonywania zadań wskazane jest krótkie przygotowanie. W tym rozdziale omawiam instalację środowiska, wymagany zakres znajomości języka C# oraz podstawy WPF i języka XAML. Na koniec przedstawiam kilka najbardziej powszechnych właściwości wykorzystywanych do ustalania położenia i rozmiaru elementów.

## 1.1 Instalacja środowiska i uruchomienie aplikacji WPF

Programy przedstawiane w niniejszym podręczniku zostały wykonane w darmowym środowisku Visual Studio Community 2017. Z uwagi na wprowadzający charakter tego podręcznika zachowuje on względną niezależność od aktualnych wersji środowiska uruchomieniowego i WPF. Na stronie <http://helion.pl/pobierz-przyklady/jchata>, gdzie udostępnione są kody programów, będą także dołączane informacje dotyczące nowszych wersji środowiska, jeśli wprowadzone zmiany dotyczyłyby w istotny sposób zakresu książki.

Instalacja środowiska nie jest skomplikowana. Wpisujemy adres [www.visualstudio.com/pl/downloads](http://www.visualstudio.com/pl/downloads) i wybieramy Visual Studio 2017 Community (lub nowszą wersję, gdy będzie dostępna). Zalecam najpierw sprawdzić, czy komputer spełnia wymagania sprzętowo-systemowe<sup>1</sup>. Jeśli tak, można pobrać instalator dla Visual Studio Community 2017. W trakcie instalacji pojawi się wykaz zestawów nazwanych „obciążeniami” (*Workloads*). Do pracy z niniejszym podręcznikiem wystarczy wybrać tylko jeden – *Programowanie aplikacji klasycznych dla platformy .NET* (w angielskiej wersji instalacji).

---

<sup>1</sup> Wymagania systemowe (ang. *system requirements*): <https://www.visualstudio.com/pl/productinfo/vs2017-system-requirements-vs>.

tora ten zestaw nazywa się *.NET desktop development*). Po zainstalowaniu środowiska można w dowolnym momencie uruchomić ponownie instalator i zmodyfikować instalację. Instalator zawiera także opcję *Pakiety językowe (Language packs)*, w której można zaznaczyć języki interfejsu. Zalecam zaznaczenie dwóch języków: polskiego i angielskiego. W niniejszym podręczniku będziemy używać interfejsu angielskiego<sup>2</sup>. Po wybraniu zestawu i języków należy kliknąć przycisk *Instaluj*. Wówczas powinno wyświetlić się okno z prezentacją przebiegu instalacji. Ten etap może trochę potrwać. Jeśli wszystko zainstaluje się prawidłowo, pojawi się okno z informacją o zakończeniu instalacji. Program można uruchomić od razu lub później. Przy pierwszym uruchomieniu środowiska zostaną wyświetlone dwa dodatkowe okna. Pierwsze z napisem „Połącz się z wszystkimi Twoimi usługami dla deweloperów” zaprasza użytkownika do zarejestrowania się i zapisania swoich ustawień osobistych. Można kliknąć link *Nie teraz, może później* i zrobić to później. W drugim oknie można wskazać wstępne ustawienia środowiska (możliwa jest ich późniejsza zmiana). Zalecam rozwinąć listę i wybrać z niej Visual C#. Poniżej można wybrać motyw koloru.

Jeżeli zostało zainstalowane środowisko z dwoma językami (polskim i angielskim), można zmieniać język interfejsu w opcji *Narzędzia/Opcje/Środowisko/Ustawienia międzynarodowe* (z polskiego na angielski) lub w opcji *Tools/Options/Environment/International Settings* (z angielskiego na polski). Po wyborze języka pojawi się komunikat z informacją, że zmiana języka będzie widoczna dopiero po ponownym uruchomieniu programu. Dalsze wskazówki dotyczące interfejsu będą podawane tylko w wersji angielskiej i takie ustawienie języka jest zalecane. Instalacja obu pakietów językowych umożliwi Czytelnikowi doraźną zmianę języka na polski, co może być przydatne w trakcie zaznajamiania się ze środowiskiem.

Aplikacje napisane w WPF w środowisku Visual Studio uruchamia się podobnie jak aplikacje konsolowe, to znaczy z debuggerem, naciskając klawisz *F5* (lub *Debug/Start Debugging*), oraz bez debuggera, używając klawiszy *Ctrl+F5* (lub *Debug/Start Without Debugging*). Program można uruchomić od razu po utworzeniu projektu WPF w opcji *File/New/Project*<sup>3</sup>, a następnie (dla Visual C#) po wybraniu *WPF App (.NET Framework)*. Uruchomiony program (bez wykonywania żadnych zmian w automatycznie wygenerowanym kodzie) wyświetli puste okno *MainWindow*. Dawniej to okno było całkiem puste, ale w wersji Visual Studio 2017 (a także Visual Studio 2015) domyślnie po uruchomieniu programu z debuggerem w oknie aplikacji pokazuje się dodatkowe okienko (pasek) z narzędziami diagnostycznymi (rysunek 1.1).

### Rysunek 1.1.

Pasek narzędziowy  
wspierający tworzenie  
widoków w kodzie XAML



<sup>2</sup> W szkołach i uczelniach jest zazwyczaj dostępny właśnie taki interfejs.

<sup>3</sup> Nowy projekt można także utworzyć, klikając na stronie początkowej (*Start Page*) link *Więcej szablonów projektów (More projects templates)* lub wybierając jeden z ostatnio używanych szablonów (*Recent project templates*).

Pierwszy przycisk tego paska (*Go To Live Visual Tree*) umożliwia przeglądanie drzewa wizualnego w trakcie działania aplikacji. O przydatności tej funkcjonalności będzie mowa w dalszej części podręcznika (w podrozdziale 11.1). Można wyłączyć pokazywanie tego paska w aplikacji (ewentualnie zwinąć go do mniejszego paska, klikając w jego dolną część). Na czas wykonywania pierwszych programów zalecam zmianę ustawienia w opcji *Tools/Options/Debugging/General* według wzoru umieszczonego na rysunku 1.2.

**Rysunek 1.2.**

*Proponowane  
ustawienia dla  
parametru Enable  
UI Debugging  
Tools for XAML*

- ☒ **Enable UI Debugging Tools for XAML**
- ☒ **Preview selected elements in Live Visual Tree**
- ☐ **Show runtime tools in application**
- ☒ **Enable XAML Edit and Continue**

Wówczas pasek narzędziowy nie będzie się wyświetlał w uruchomionej aplikacji, ale jego przyciski będą dostępne w zewnętrznym oknie *Live Visual Tree*. Ponadto będzie można w dowolnej chwili pasek przywrócić do okna aplikacji (przyciskiem *Show runtime tools in application* w oknie *Live Visual Tree*). Ostatni parametr, *Enable XAML Edit and Continue*, pozwala uwidocznąć efekty zmiany kodu XAML w uruchomionej aplikacji (bez konieczności ponownego jej kompilowania).

## 1.2 Wymagany zakres znajomości języka C#

Niniejszy podręcznik przeznaczony jest dla osób początkujących, które niedawno zaczęły się uczyć języka C#. Wymagana jest jednak pewna podstawowa znajomość tego języka. Do podstawowego zakresu można zaliczyć:

- ♦ zmienne, stałe i ich typy oraz operatory;
- ♦ instrukcje warunkowe i pętle;
- ♦ tablice, listy oraz inne kolekcje;
- ♦ definiowanie klas (w tym składowe klas: pola, właściwości i metody);
- ♦ interfejsy;
- ♦ podstawy dotyczące dziedziczenia.

Nieznajomość niektórych spośród wymienionych tematów nie wyklucza możliwości korzystania z tej książki — pod warunkiem równoległej nauki C#. Powyższe zagadnienia można znaleźć w wielu podręcznikach do języka C# oraz w kursie wideo *Język C#. Kurs video. Poziom pierwszy. Programowanie dla początkujących*, dostępnym na platformie [videopoint.pl](http://videopoint.pl). Tu dla przypomnienia omówione zostaną krótko tylko dwa z nich — te, które mają szczególne znaczenie dla programów w WPF, a mianowicie właściwości i interfejsy. Jeżeli znasz dobrze oba zagadnienia, możesz ominąć tę część podrozdziału.

## Właściwości

Właściwości służą do kontrolowania dostępu (odczytu i zapisu) do danego pola klasy<sup>4</sup>. Spójrzmy na konsolowy prosty przykład klasy zawierającej właściwość:

```
public class Pracownik
{
    private string nazwisko;
    private double zarobki;
    public double Zarobki           // Właściwość dla pola zarobki
    {
        get { return zarobki; }
        set { zarobki = value; }
    }
    public Pracownik(string naz, double zar)  // Konstruktor
    {
        nazwisko = naz;
        zarobki = zar;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Pracownik p1 = new Pracownik("Kowalski", 2000);
        p1.Zarobki = 2250.0;           // Użycie akcesora set
        Console.WriteLine(p1.Zarobki); // Użycie akcesora get
        Console.ReadKey();
    }
}
```

W klasie są dwa prywatne pola: `nazwisko` i `zarobki`. Ponadto jest właściwość publiczna `Zarobki` dla pola `zarobki`. W pewnym uproszczeniu moglibyśmy powiedzieć, że element klasy o nazwie `Zarobki` (tzn. właściwość) to specjalna „metoda”, która składa się z dwóch „metod wykonawczych” — tak zwanych akcesorów `get` i `set`. Akcesor `get` jest wykonywany, gdy w programie użyto danej właściwości „do odczytu”, natomiast akcesor `set` jest wykonywany, gdy program napotka instrukcję przypisania dla danej właściwości (czyli gdy użyto właściwości „do zapisu”). Właściwość ma dostęp publiczny i typ taki sam jak pole, którego dotyczy, a w jej ciele znajdują się definicje obu akcesorów. W omawianym przykładzie oba akcesory zawierają tylko jedną instrukcję (może być ich więcej). Akcesor `get` zwraca wartość pola (`return zarobki;`). Akcesor `set` przypisuje do pola wartość `value` — to nie jest zmienna, lecz niejawnny parametr, który symbolizuje wartość przypisywaną do właściwości (ma taki sam typ jak dana właściwość). W metodzie `Main` są dwie instrukcje, które „wywołują” odpowiednie akcesory (oznaczone w komentarzach kodu).

Właściwość powinna mieć dostęp publiczny, ale niekoniecznie oba akcesory muszą być publiczne. Poniżej umieszczono definicję właściwości z modyfikatorem `private` dla akcesora `set`. Dla tak zdefiniowanej właściwości nie można przypisać wartości na zewnątrz klasy, możliwy będzie jedynie jej odczyt.

<sup>4</sup> Kempa A., Staś T., *Wstęp do programowania w C#. Łatwy podręcznik dla początkujących*, Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice 2014.



```
public double Zarobki
{
    get { return zarobki; }
    private set { zarobki = value; }
}
```

W przypadku gdy właściwości nie mają żadnej dodatkowej pracy do wykonania<sup>5</sup>, co ma miejsce w omawianych przykładach (tzn. akcesor `get` udostępnia daną do odczytu, a `set` do zapisu — i nie robią nic więcej), począwszy od wersji C# 3.0, jest możliwe użycie tak zwanych **automatycznych właściwości**<sup>6</sup>. Mają one uproszczoną (zwięzłą) składnię i z tego powodu są chętnie używane.

Definicja automatycznej właściwości w prezentowanym przykładzie wymagałaby usunięcia kodu:

```
private double zarobki;
public double Zarobki // Właściwość dla pola zarobki
{
    get { return zarobki; }
    set { zarobki = value; }
}
```

I wpisania w jego miejsce linii:

```
public double Zarobki { get; set; } // Właściwość automatyczna
```

Przyjrzyjmy się definicji automatycznej właściwości. Mamy tu tylko jedną składową, która jest publiczna. W tym przypadku kompilator automatycznie utworzy prywatną składową (jako anonimowe pole, do którego dostęp możliwy będzie jedynie przez właściwość). Modyfikator `public` dotyczy dostępu do właściwości, a nie pola (które wygeneruje się automatycznie). Nie definiujemy tu kodu dla akcesorów, umieszczamy jedynie ich nazwy (które mogą być poprzedzone modyfikatorem dostępu). Podobnie jak w przypadku pełnej definicji właściwości i tutaj można wybiórczo ustalić dostęp do akcesorów. Przykładowo jest możliwa deklaracja, która udostępnia akcesor `set` jako prywatny:

```
public double Zarobki {get; private set; }
```

Cechą specyficzną WPF jest używanie tak zwanych właściwości zależnych, o których będzie mowa w dalszej części książki.

<sup>5</sup> Właściwości można wykorzystać także do dodatkowych zadań. Na przykład akcesor `get` może zwrócić wartość po przeliczeniu na odpowiednią jednostkę, a `set` może sprawdzać, czy parametr niejawni `value` ma dozwoloną wartość dla danego pola. W przypadku WPF akcesory `set` wykorzystuje się m.in. do przekazywania powiadomień o zmianie właściwości.

<sup>6</sup> Nazwa „automatyczna właściwość” nie oddaje w pełni „automatycznego” charakteru tej właściwości. Gdyby tę nazwę uzupełnić, musiałaby brzmieć mniej więcej tak: „automatycznie implementowana właściwość z automatycznym definiowaniem prywatnego pola anonimowego”. Zostaniemy jednak przy krótszej nazwie (w języku angielskim używa się nazwy *auto-implemented properties*).

## Interfejsy

W tej książce słowo **interfejs** pojawia się w dwóch kontekstach:

- ♦ jako interfejs graficzny (GUI), umożliwiający interakcje użytkownika z programem, oraz
- ♦ jako konstrukcja programistyczna, pod niektórymi względami przypominająca klasy (**interface**).

W tym miejscu interesuje nas tylko ten drugi kontekst. Krótko przypomnimy te aspekty dotyczące interfejsów, które mają znaczenie dla prezentowanych w tym podręczniku programów w WPF.

Projekty programistyczne powstają zazwyczaj w zespołach. Już na etapie tworzenia bibliotek wyodrębnia się pewne uniwersalne funkcjonalności i je definiuje. Niektóre komponenty tworzy się jedynie „częściowo”, bez pewnych szczegółów, bo te mogą być konkretyzowane dopiero w chwili ich zastosowania podczas tworzenia konkretnego projektu. Interfejsy pozwalają właśnie na ową konkretyzację szczegółów, a tym samym umożliwiają łączenie poszczególnych części programu, na przykład własnej klasy z kodem klasy bibliotecznej.

Interfejs definiuje klasę i jej elementy bez ich implementacji. Stanowi coś w rodzaju kontraktu (umowy, zobowiązania) nakazującego klasie określone działania. Klasa implementująca interfejs tworzy implementację wszystkich elementów wymienionych w tym interfejsie. Interfejs może zawierać jedynie metody (samą ich deklarację) i właściwości oraz zdarzenia i indeksatory. Ponadto przyjęto konwencję co do nazwy interfejsów — powinny zaczynać się od litery I (np. **IWektor**, **IPunkt**).

Definicja interfejsu przypomina definicję klasy. Zamiast słowa **class** pisze się **interface**. Domyślny dostęp dla interfejsu jest publiczny. Przykładowy interfejs może mieć następującą postać:

```
interface IPunkt
{
    // Składowymi interfejsu mogą być jedynie właściwości i metody (oraz zdarzenia i indeksatory)
    // Nie podajemy dla nich modyfikatora dostępu
    int X { get; set; }      // Właściwość
    int Y { get; set; }      // Właściwość
    string DajOpis();        // Deklaracja metody
}
```

Interfejs **IPunkt** wymaga dwóch właściwości typu **int** **X** i **Y** oraz bezargumentowej metody **DajOpis** zwracającej wartość typu **string**.

Klasa, która implementuje dany interfejs<sup>7</sup>, musi zawierać definicje wymaganych przez ten interfejs składowych. Po nazwie takiej klasy i dwukropku umieszczę nazwę interfejsu (niekoniecznie jedną nazwę, klasa może implementować więcej interfejsów). Przykładowa klasa implementująca interfejs **IPunkt** może wyglądać tak:

---

<sup>7</sup> W kontekście interfejsów częściej się mówi „implementuje” niż „dziedziczy”.

```
class Punkt : IPunkt    // Klasa Punkt implementuje interfejs IPunkt
{
    public int X { get; set; }    // Implementacja interfejsu - musi być public
    public int Y { get; set; }    // Implementacja interfejsu - musi być public

    public string DajOpis()    // Implementacja interfejsu - musi być public
    {
        return String.Format("Współrzędne {0},{1}", X, Y);
    }
}
```

Klasa `Punkt` mogłaby mieć jeszcze inne składowe, pola, właściwości czy metody. Ważne, że jeśli „zobowiązała się” do implementacji danego interfejsu, to musi zawierać definicje tego, czego on wymaga. Prezentowany program można uruchomić jako aplikację konsolową:

```
class Program
{
    static void Main(string[] args)
    {
        Punkt p1 = new Punkt{ X = 4, Y = 5 };
        Console.Write(p1.DajOpis());
        Console.ReadKey();
    }
}
```

Użyteczność interfejsów można wyraźniej zaobserwować, tworząc program z użyciem jakiegoś rozwiązania, które wymaga implementacji interfejsów. Przykładem może być sortowanie kolekcji zawierającej elementy typu naszej klasy (czyli niewbudowanego). Metoda sortująca w klasie `List<T>` dla danych typu `int` nie potrzebuje od nas żadnych dodatkowych informacji, bo „wie”, jak posortować wartości typu `int`. Jeżeli jednak chcemy posortować elementy jakiejś swojej klasy (np. powyższej klasy `Punkt`), to musimy dostarczyć informacje, jak należy sortować te elementy, to znaczy według jakiego kryterium je porównywać. W tym celu implementuje się interfejs `IComparable`, który wymaga zdefiniowania metody `CompareTo`. W naszych programach będziemy używać interfejsów w rozwiązaniach udostępniających walidację danych oraz mechanizm przesyłania powiadomień.

Nie przedstawiam tu wszystkich istotnych zagadnień związanych z interfejsami. Oprócz tego, że odgrywają istotną rolę w mechanizmie scalania poszczególnych części programu, interfejsy pełnią także ważną funkcję we wspieraniu polimorfizmu. Ten drugi aspekt jednak nie będzie miał większego znaczenia w prezentowanych programach.

Na koniec poruszę jeszcze trzy kwestie przydatne podczas dalszej pracy z podręcznikiem: nazewnictwo elementów kodu, przestrzenie nazw oraz dokumentacja MSDN.

## Nazewnictwo elementów kodu

Wielu programistów preferuje angielskie nazwy dla tworzonych obiektów, klas i innych elementów kodu. W programach używanych w celach dydaktycznych sprawdza się jednak dwujęzyczność w nazewnictwie. Zarówno cała platforma .NET, jak i WPF mają

bogate zasoby gotowych klas. Używanie polskich nazw pozwala wyróżnić kod własny. Na początku nauki może to być pomocne i dlatego przyjąłem tę konwencję w niniejszej książce (z małymi wyjątkami).

## Przestrzenie nazw w kodzie C#

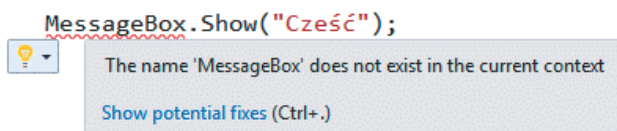
Podczas tworzenia nowych plików klas C# środowisko Visual Studio wstępnie umieszcza kilka dyrektyw `using`, np.:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

W omawianych programach (w dalszej części podręcznika) umieszczane są informacje o konieczności dodania wymaganych deklaracji przestrzeni nazw. Jeżeli jednak ktoś pracuje z inną wersją środowiska lub wykasował sobie dyrektywy `using`, może je dodać w przypadku, gdy jakieś nazwy metod lub klas nie zostaną rozpoznane. Taką sytuację zasygnalizuje nam edytor (kompilator oczywiście także). Po najechnięciu myszą na nieznaną nazwę pojawi się komunikat prezentowany na rysunku 1.3. Można wówczas skorzystać z podpowiedzi, jaką sugeruje środowisko, i przy użyciu opcji *Show potential fixes* wskazać przestrzeń nazw.

### Rysunek 1.3.

*Komunikat informujący, że w bieżącym kontekście dana nazwa jest nieznaną*



## Dokumentacja MSDN

W wielu różnych miejscach (dotyczących języka C# i kodu XAML) tej książki znajdują się odniesienia zarówno do bardziej obszernych podręczników, jak i do dokumentacji MSDN (*Microsoft Developer Network*). Biblioteka MSDN, dostępna na stronie <http://msdn.microsoft.com/library>, to scentralizowana baza oficjalnych dokumentów zawierających opisy techniczne dla programistów i deweloperów. Linki do konkretnych tematów w tej dokumentacji nie są wygodne do ręcznego przepisywania, dlatego przed każdym linkiem w okrągłych nawiasach umieszczone zostały tytuły artykułów. Na ich podstawie i skrótu „MSDN” można wyszukać daną stronę w wyszukiwarce. Dokumentacja dla klas platformy .NET jest sporządzona w języku angielskim. Dostępne polskie tłumaczenie nie jest dobrej jakości, dlatego podaję linki do wersji angielskiej.

## 1.3 Podstawy WPF

Zgodnie z przyjętą dla tego podręcznika konwencją wprowadzamy jedynie niezbędne minimum teoretycznych zagadnień. Pozostałe tematy zostaną omówione przy okazji przedstawiania konkretnych funkcjonalności oraz w ostatnim, teoretycznym rozdziale.

Omawiając podstawy WPF, zazwyczaj rozpoczyna się od przedstawienia **diagramu dziedziczenia**. Klasy WPF mają bardzo głęboką hierarchię dziedziczenia i omawianie jej na tym etapie nauki jest przedwczesne. W prezentowanych programach będziemy mieli okazję dostrzec pewne konsekwencje dziedziczenia klas. Zdobyta wiedza praktyczna na ten temat zostanie uporządkowana w końcowym rozdziale, gdzie omówię uproszczoną hierarchię klas.

Drugim aspektem, jaki pojawia się nierzadko we wstępnych informacjach na temat WPF, jest drzewo logiczne i drzewo wizualne (prezentacji). W tym podręczniku zagadnienia te wyjaśnię dopiero tam, gdzie będą potrzebne, a mianowicie podczas omawiania szablonów. Wówczas Czytelnik będzie już dość dobrze znał podstawy XAML i znacznie łatwiej będzie mu zrozumieć różnice między jednym a drugim drzewem. Teraz jedynie nadmienię, że interfejs w WPF ma budowę hierarchiczną, czyli drzewiastą. Uproszczona wersja tego drzewa (pozbawiona pewnych szczegółów) nazywana jest **drzewem logicznym**. Natomiast pełna wersja to **drzewo wizualne** (lub drzewo prezentacji).

Jednym z ważniejszych mechanizmów WPF są **właściwości zależne** (ang. *dependency properties*). Właściwości zależne stanowią podstawę dla wiązania danych, stylów oraz animacji. Wartości takich właściwości mogą zależeć od innych obiektów. Wśród bardziej istotnych funkcjonalności, jakie zapewniają właściwości zależne, należy wymienić możliwość powiadamiania o zmianie wartości. Temat ten pozna Czytelnik od strony praktycznej w podrozdziale 10.4, gdzie omówię wyzwalacze właściwości (ang. *property triggers*). Kolejną ważną funkcjonalnością właściwości zależnych jest dziedziczenie wartości właściwości. Dziedziczenie wartości właściwości nie dotyczy tradycyjnego obiektowego dziedziczenia po klasie bazowej. W tym przypadku chodzi o przekazywanie wartości właściwości w dół drzewa elementów. Z tym zagadnieniem Czytelnik zetknie się w podrozdziale 10.3, w którym omówię style. Szczególnym rodzajem właściwości zależnych są **właściwości dołączane** (ang. *attached properties*), które są dołączane do obiektów „obcych” klas w celach dziedziczenia w strukturze kodu XAML.

W programach prezentowanych w tym podręczniku korzystać będziemy z możliwości deklaratywnego opisu interfejsu użytkownika w języku XAML wraz z zapleczem w języku w C# zwanym *code-behind* („kod pod spodem”)<sup>8</sup>.

---

<sup>8</sup> Aplikacje w WPF można tworzyć na kilka sposobów, my wykorzystamy najczęściej stosowany sposób — z użyciem skompilowanego XAML (BAML).

## 1.4 Podstawy XAML

**XAML** (ang. *Extensible Application Markup Language*) jest to oparty na standardzie XML język stosowany do definicji interfejsu użytkownika w WPF. XAML może współpracować także z innymi technologiami, takimi jak Windows Workflow Foundation (WWF) czy Windows Communication Foundation (WCF).

XAML pozwala w sposób deklaratywny zdefiniować interfejs użytkownika, ułatwiając oddzielenie logiki programu od warstwy wizualnej. Wiąże się to z inną cechą XAML, którą jest usprawnienie współpracy programistów i grafików w zespole pracującym nad danym projektem.

### Dokument XML

Przed omówieniem XAML krótko przedstawię język **XML** (ang. *Extensible Markup Language*). XML to uniwersalny język znaczników, za pomocą którego można opisać dane w ustrukturalizowany sposób. Przykładowe dane prezentuje rysunek 1.4.

**Rysunek 1.4.**

*Dane o produktach*

Symbol	Nazwa	Liczba sztuk	Magazyn
O1-11	ołówek	8	Katowice 1
PW-20	pióro wieczne	75	Katowice 2

Na rysunku znajdują się dwa wiersze danych o produktach. Zapiszemy te dane w postaci dokumentu XML na dwa różne sposoby. Najpierw przeanalizujemy pierwszy sposób:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<ListaProduktow>
  <Produkt>
    <Symbol>O1-11</Symbol>
    <Nazwa>ołówek</Nazwa>
    <LiczbaSztuk>8</LiczbaSztuk>
    <Magazyn>Katowice 1</Magazyn>
  </Produkt>
  <Produkt>
    <Symbol>PW-20</Symbol>
    <Nazwa>pióro wieczne</Nazwa>
    <LiczbaSztuk>75</LiczbaSztuk>
    <Magazyn>Katowice 2</Magazyn>
  </Produkt>
</ListaProduktow>
```

Każdy dokument zaczyna się od prologu, w którym jest deklaracja XML (i ewentualnie strony kodowej) — i ten zapis widzimy w pierwszej linii. Popatrzmy na resztę kodu. Wyrazy umieszczone w nawiasach ostrokątnych (< >) to tak zwane **znaczniki**. Znaczniki mogą być otwierające (np. <Produkt>) i zamykające (np. </Produkt>). Poprzez znaczniki opisujemy **elementy** dokumentu XML. Każdy dokument XML ma **element główny**, tak zwany **korzeń**. W tym dokumencie jest to element <ListaProduktow>.

Zasadniczą cechą dokumentów XML jest hierarchiczna budowa, czyli możliwość zagnieżdżania elementów. Elementy niższego szczebla są elementami potomnymi, natomiast element wyższego szczebla w stosunku do danego elementu jest jego rodzicem. I tak element główny `<ListaProduktow>` ma dwa elementy potomne `<Produkt>`. Elementy te także mają swoje elementy potomne, opisujące poszczególne cechy produktu. A cechy te stanowią wartość danego elementu i już nie zawierają elementów potomnych.

Popatrzmy teraz na drugi przykład dokumentu XML, który opisuje te same dane:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<ListaProduktow>
  <Produkt Symbol="01-11" Nazwa="ołówek" LiczbaSztuk="8" Magazyn="Katowice 1"/>
  <Produkt Symbol="PW-20" Nazwa="pióro wieczne" LiczbaSztuk="75"
    Magazyn="Katowice 2"/>
</ListaProduktow>
```

W elemencie głównym `<ListaProduktow>` także są dwa elementy ze znacznikami `<Produkt>`, ale tym razem poszczególne cechy produktów zostały opisane za pomocą **atrybutów**. `Symbol`, `Nazwa`, `LiczbaSztuk` oraz `Magazyn` to atrybuty elementu `<Produkt>`. Ponieważ wszystko, co opisuje produkt, zostało zdefiniowane w postaci atrybutów, nie było konieczności, aby wstawiać osobno znacznik kończący ten element. W takich przypadkach można zastosować zapis skrócony, w którym znacznik zarówno się otwiera, jak i zamyka. Ta jego podwójna rola jest oznaczona poprzez ukośnik umieszczony przed zamykającym nawiasem ostrokątnym. Inaczej mówiąc zapis:

```
<Produkt Symbol="01-11" Nazwa="ołówek" LiczbaSztuk="8" Magazyn="Katowice 1"/>
```

stanowi skrócenie zapisu:

```
<Produkt Symbol="01-11" Nazwa="ołówek" LiczbaSztuk="8" Magazyn="Katowice 1">
</Produkt>
```

Oczywiście można łączyć oba sposoby opisu danych w XML, to znaczy można w poszczególnych elementach używać zarówno atrybutów, jak i elementów potomnych.

## Kod XAML

Omówione zasady obowiązują także w języku XAML. Popatrzmy na przykładowy kod XAML:

```
<Button Width="120" Height="30" Content="Zapisz"/>
```

Uwzględniając terminologię dla składni XML, powiedzielibyśmy, że znacznik `Button` (przycisk) opisuje element mający trzy atrybuty: `Width` (szerokość), `Height` (wysokość) oraz `Content` (zawartość). Ale rzadko będziemy używać terminów składniowych XML (jedynie w początkowych programach). Bardziej użyteczne będą dla nas pojęcia wynikające z kontekstu opisanych informacji. I tak przykładowo `Button` jest kontrolką, która ma swoje właściwości. Jedną z tych właściwości jest `Width` i raczej będziemy o niej mówić „właściwość”, a nie „atrybut”.

Warto także dodać, że XAML (podobnie jak XML) rozróżnia wielkie i małe litery, zatem znacznik `<Produkt>` nie jest tym samym co `<produkt>`.

Popatrzmy na jeszcze jeden przykład kodu XAML, w którym zastosowany jest zarówno zapis z użyciem atrybutów, jak i elementów podrzędnych:

```
<ComboBox SelectedIndex="0" Height="30" Width="145">
    <ComboBoxItem Content="Biały"/>
    <ComboBoxItem Content="Czerwony"/>
    <ComboBoxItem Content="Zielony"/>
    <ComboBoxItem Content="Niebieski"/>
</ComboBox>
```

`ComboBox` jest kontrolką umożliwiającą wyświetlanie listy rozwijanej i wybór elementu z tej listy. Właściwości takie jak `SelectedIndex` (wstępnie wybrana pozycja) czy `Height` lub `Width` dotyczą całej kontrolki, można było więc zapisać je w postaci atrybutów dla listy. Ale lista ma swoje pozycje, które definiujemy w postaci elementów zagnieżdżonych. Tak zdefiniowana lista rozwijana będzie miała cztery pozycje: *Biały*, *Czerwony*, *Zielony* i *Niebieski*.

W kodzie XAML, podobnie jak w XML, można umieszczać komentarze, czyli oznaczać fragmenty, które mają być ignorowane. Można w nich podawać dodatkowe opisy lub „wyłączyć” za ich pomocą fragmenty kodu na czas testów. Komentarze umieszcza się w znacznikach `<!--` oraz `-->`. Oto przykład kodu XAML z listą rozwijaną, w którym zakomentowano dwie linie:

```
<ComboBox SelectedIndex="0" Height="30" Width="145">
    <!--ComboBoxItem Content="Biały"/>
    <ComboBoxItem Content="Czerwony"/-->
    <ComboBoxItem Content="Zielony"/>
    <ComboBoxItem Content="Niebieski"/>
</ComboBox>
```

Tak zdefiniowana kontrolka `ComboBox` pokazałaby nam jedynie dwa kolory do wyboru (zielony i niebieski).

Podobnie jak w dokumentach XML także w kodzie XAML obowiązują zasady dotyczące hierarchicznej struktury, przy czym dla XAML rodzi to specyficzne konsekwencje. O niektórych z nich wspomniałam w poprzednim podrozdziale, nawiązując do drzewa logicznego i dziedziczenia właściwości zależnych.

## Automatycznie generowany kod XAML dla nowego projektu

Podczas tworzenia nowego projektu kod XAML zostaje wstępnie wpisany przez środowisko. Zanim przystąpimy do pisania pierwszych programów, byłoby dobrze omówić najważniejsze elementy tego kodu. Uruchom środowisko Visual Studio i utwórz nowy projekt (*File/New/Project*), a następnie (dla Visual C#) wybierz *WPF App (.NET Frame*



work). Nazwij projekt **TwojaNazwaAplikacji**. Po utworzeniu projektu w dolnym oknie pojawi się wstępnie zdefiniowany kod XAML<sup>9</sup>:

```
<Window x:Class="TwojaNazwaAplikacji.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:TwojaNazwaAplikacji"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Najpierw przyjrzyjmy się strukturze tego dokumentu. Elementem głównym (korzeniem) jest **Window**. Dla tego elementu określone są atrybuty. Wewnątrz elementu głównego jest jeden element potomny **Grid**. Element **Grid** to panel pozwalający porządkować inne elementy (np. kontrolki).

## Atrybuty elementu Window

Przeanalizujemy teraz atrybuty użyte w elemencie głównym **Window**. Jest tam atrybut **x:Class**, kilka kolejnych zawiera w nazwie **xmlns**, następnie mamy atrybut **mc:Ignorable**, a trzy ostatnie to **Title**, **Height** i **Width**. Wymienione atrybuty zostaną pokrótce omówione.

```
x:Class="TwojaNazwaAplikacji.MainWindow"
```

Atrybut **x:Class** może być umieszczany tylko wewnątrz elementu głównego. Atrybut ten definiuje klasę głównego elementu, która dziedziczy po typie elementu (tu **Window**). Moglibyśmy powiedzieć, że atrybut **x:Class** stanowi rodzaj pomostu między elementem **Window** a *code-behind*, czyli klasą C# o nazwie **MainWindow** w przestrzeni nazw **TwojaNazwaAplikacji**<sup>10</sup>.

Atrybut **xmlns** (nazwa pochodzi od *XML Namespace*) określa przestrzeń nazw wykorzystywaną w danym elemencie XAML (jest odpowiednikiem dyrektywy **using** w języku C#). Popatrzmy na pierwszą przestrzeń nazw:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Ta przestrzeń nazw zawiera klasy WPF i jest domyślną przestrzenią nazw.

W XAML można deklarować wiele przestrzeni nazw, ale każda następna musi mieć jakiś postfixs oddzielony dwukropkiem. W dalszej części definiowana jest przestrzeń nazw z użyciem postfixu **x**:

<sup>9</sup> Prezentujemy definicję kodu XAML wygenerowaną automatycznie w wersji Visual Studio 2017.

<sup>10</sup> Matulewski J., *MVVM i XAML w Visual Studio 2015*, Helion, Gliwice 2016 (str. 15). Porównaj także: Petzold C., *Windows 8. Programowanie aplikacji z wykorzystaniem C# i XAML*, Helion, Gliwice 2013 (str. 23 – 24).

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Przestrzeń ta zawiera opis słów kluczowych XAML i rozszerzeń znaczników.

Obie wymienione przestrzenie nazw nie są adresami internetowymi, są to jedynie wartości instancji obiektu klasy `XmlnsDefinitionAttribute`, która jest wykorzystywana do mapowania przestrzeni nazw.

Kolejne dwie przestrzenie nazw z postfixami `d` i `mc` oraz atrybut `mc:Ignorable` dotyczą współpracy WPF ze środowiskiem Microsoft Blend, które daje bardziej zaawansowane możliwości w zakresie tworzenia komponentów graficznych interfejsu. Nie będziemy wykorzystywać tego narzędzia w programach tworzonych w tej książce.

Została nam do przedstawienia jeszcze jedna przestrzeń nazw, mianowicie `xmlns:local`. Dzięki tej deklaracji możemy się odwoływać do przestrzeni nazw dla bieżącego projektu, używając nazwy `local`.

Pozostałe atrybuty znajdujące się w elemencie `Window` nie wymagają specjalnego wyjaśniania. `Title` określa tytuł okna, `Height` wysokość, a `Width` szerokość okna. Możemy je dowolnie zmieniać.

W naszych pierwszych programach będziemy zazwyczaj zmieniać jedynie ustawienia dla tytułu i rozmiaru okna, np. `Title="Przywitanie" Height="300" Width="200"`. Ponadto będziemy coś wpisywać wewnątrz znaczników `<Grid>` i `</Grid>`. Możesz przykładowo wpisać tam definicję przycisku:

```
<Button Width="120" Height="30" Content="Zapisz"/>
```

Wówczas pojawi się przycisk *Zapisz* (na razie bez żadnej funkcjonalności).

## Słowa kluczowe XAML

Wspomniana wcześniej przestrzeń nazw języka XAML definiowana w linii:

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

zawiera definicję słów kluczowych, które powinny być specjalnie traktowane przez kompilator i parser. Przedstawimy tylko kilka wybranych słów kluczowych, które zostały użyte w tym podręczniku<sup>11</sup>. Postfix `x` w nazwie atrybutu `xmlns:x` jest jednocześnie prefiksem dla elementów dostępnych w tej przestrzeni nazw:

- ◆ `x:Class` — to atrybut korzenia. Jak już wcześniej wspomniałam, definiuje on klasę głównego elementu, która dziedziczy po typie elementu.
- ◆ `x:Key` — pozwala określić klucz dla danego elementu. Jednoznacznie identyfikuje elementy, które są definiowane jako zasoby<sup>12</sup>.

<sup>11</sup> Pełna lista słów kluczowych jest dostępna m.in. w: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 63 – 65).

<sup>12</sup> Zasoby XAML przechowywane są w obiekcie klasy `ResourceDictionary` — słowniku mającym klucze i wartości typu `object` (klucze są zazwyczaj łańcuchami znaków).

- ♦ `x:Shared` — określa, czy zasób ma być dzielony przez tę samą instancję (`True` — tak jest domyślnie), czy też powinna być tworzona nowa instancja zasobu (`False`).
- ♦ `x:Name` — definiuje nazwę danego elementu, za pomocą której można się odwoływać do niego w *code-behind*. Można używać także `Name` bez prefiksu `x`, ale tylko dla elementów wywodzących się z klas, dla których dostępne są właściwości `Name`, mogące pełnić rolę nazwy elementu (`FrameworkElement` i `FrameworkContentElement`)<sup>13</sup>.

## Code-behind

Po utworzeniu nowego projektu w Visual Studio zostaje automatycznie wygenerowany kod XAML w pliku *MainWindow.xaml* oraz kod zaplecza dla tego kodu (*code-behind*) w pliku *MainWindow.xaml.cs*.

Najpierw przypomnijmy definicję atrybutu `x:Class` znajdującego się w elemencie `Window`:

```
x:Class="TwojaNazwaAplikacji.MainWindow"
```

Na podstawie tej definicji zostanie utworzona klasa `MainWindow`, dziedzicząca po klasie `Window`, w przestrzeni nazw `TwojaNazwaAplikacji`. Zerknijmy zatem do *code-behind* w pliku *MainWindow.xaml.cs*:

```
using System.Windows;
namespace TwojaNazwaAplikacji
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Klasa `MainWindow` jest klasą częściową (`partial class`). W projekcie Visual Studio widzimy tylko jedną część definicji tej klasy, to znaczy plik *MainWindow.xaml.cs*. Pozostałą jej część wygenerowaną automatycznie można odszukać w folderach projektu w eksploratorze Windows<sup>14</sup>, ale nie ma potrzeby, aby to robić, ponieważ nie będziemy tego kodu zmieniać. W obecnej postaci programu w klasie tej jest tylko metoda `InitializeComponent`, która tworzy egzemplarze wszystkich obiektów XAML i umieszcza je w drzewie<sup>15</sup>. Nie możemy usuwać wywołania tej metody.

<sup>13</sup> Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 52). Porównaj także: Petzold C., *Windows 8. Programowanie aplikacji z wykorzystaniem C# i XAML*, Helion, Gliwice 2013 (str. 39 – 40).

<sup>14</sup> Pliki *MainWindow.g.cs* i *MainWindow.g.i.cs* w folderze *obj/Debug* (i ewentualnie *obj/Release*).

<sup>15</sup> Petzold C., *Windows 8. Programowanie aplikacji z wykorzystaniem C# i XAML*, Helion, Gliwice 2013 (str. 41 – 42).

Drugą ważną klasą obok `Window` jest klasa `Application`, która odpowiada za tworzenie wspólnej infrastruktury aplikacji i zarządzanie nią<sup>16</sup>. Po stworzeniu nowego projektu WPF, obok omówionych plików dla okna, tworzą się także pliki dla aplikacji, w tym widoczne w projekcie `App.xaml` i `App.xaml.cs`. W programach tworzonych w tym podręczniku nie będziemy zmieniać automatycznie wygenerowanej zawartości tych plików. Jeśli ktoś się zastanawia, gdzie w aplikacjach WPF podziela się metoda `Main`, może zajrzeć do folderu projektu i odnaleźć plik `App.g.cs`, zawierający wygenerowany przez środowisko kod dla pliku `App.xaml`<sup>17</sup>.

## Rozszerzenia znaczników

Rozszerzenia znaczników pozwalają rozszerzyć możliwości XAML (obok konwerterów, które omówię później). Ogólny schemat rozszerzenia ma postać:

```
{NazwaKlasy [Opcjonalnie argumenty]}
```

Definicja rozszerzenia jest umieszczana w nawiasach klamrowych i składa się z nazwy klasy dla danego rozszerzenia i opcjonalnie z argumentów. Oto krótkie przedstawienie kilku wbudowanych rozszerzeń znaczników, które pojawią się w programach tworzonych w tej książce:

- ◆ `Binding` — to rozszerzenie wykorzystywane jest do definiowania wiązania danych.
- ◆ `RelativeSource` — pozwala przypisać wartość właściwości `RelativeSource` dla rozszerzenia `Binding`.
- ◆ `x:Null` — reprezentuje referencję `null`.
- ◆ `x:Static` — jest to referencja do statycznego składnika klasy.
- ◆ `x:Type` — pełni podobną funkcję jak operator `typeof` w kodzie napisanym w C#.

Rozszerzenia poprzedzone prefiksem `x` definiowane są w przestrzeni nazw XAML, czyli `xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`.

Opis przedstawionych w tym podrozdziale słów kluczowych i rozszerzeń znaczników XAML stanie się bardziej zrozumiały podczas wykonywania programów w dalszych rozdziałach podręcznika.

<sup>16</sup> Cisek J., *Tworzenie nowoczesnych aplikacji graficznych w WPF*, Helion, Gliwice 2012 (str. 217 – 221).

<sup>17</sup> Plik `App.g.cs` znajduje się w folderze `obj/Debug` (i ewentualnie `obj/Release`).

## 1.5 Sterowanie rozmiarem i pozycją elementów

Sterowanie układem graficznym elementów zależy od interakcji między elementami potomnymi a elementami nadrzędnymi (rodzicami). W WPF możemy ustalać ułożenie elementów względem siebie w tak zwanych panelach. Panele zostaną omówione w rozdziale 4, a teraz zajmiemy się podstawowymi właściwościami wykorzystywanymi do ustalania położenia i rozmiaru elementów wewnątrz elementu nadrzędnego.

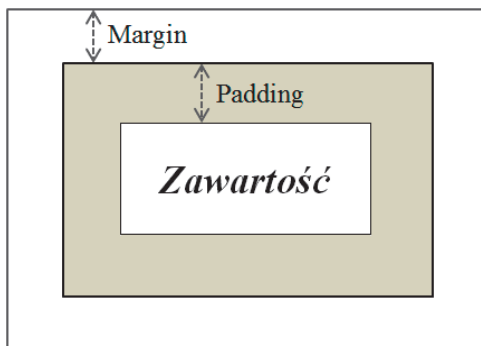
### Height i Width

WPF udostępnia więcej właściwości wpływających na rozmiar elementu niż wspomniane już **Height** (wysokość) i **Width** (szerokość). Są to między innymi także: **MinHeight** (minimalna wysokość), **MaxHeight** (maksymalna wysokość), **MinWidth** (minimalna szerokość), **MaxWidth** (maksymalna szerokość). Można łączyć poszczególne ustawienia. Przykładowo możemy określić wysokość za pomocą właściwości **Height** oraz właściwości **MinHeight** lub/i **MaxHeight** — wówczas ustawienie **Height** będzie miało pierwszeństwo (pod warunkiem że mieści się w przedziale od wartości minimalnej do wartości maksymalnej).

### Margin i Padding

Właściwość **Margin** określa margines zewnętrzny dla danego elementu. Natomiast właściwość **Padding** określa „margines” wewnętrzny. Rysunek 1.5 przedstawia hipotetyczny element (np. przycisk) w kolorze szarym. **Margin** wyznacza przestrzeń wokół tego elementu. Niektóre elementy mogą mieć zawartość, na przykład przycisk może mieć napis lub rysunek. **Padding** wyznacza wolną przestrzeń między zawartością a krawędzią elementu. Zawartość prezentowana jest na rysunku w postaci białego prostokąta wewnątrz przycisku. Należy pamiętać, że jest to rysunek poglądowy i niekoniecznie obszar wyznaczony przez właściwość **Padding** musimy widzieć tak wyraźnie jak na tym rysunku.

**Rysunek 1.5.**  
Właściwości  
*Margin i Padding*



Zarówno właściwość `Margin`, jak i `Padding` można ustawić w kodzie XAML na trzy sposoby: podając jedną wartość, dwie wartości lub cztery wartości. Zostanie to wyjaśnione na przykładach dla właściwości `Margin`:

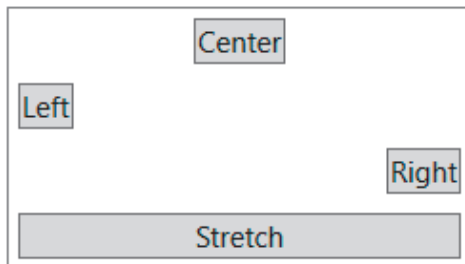
- ◆ `Margin="10"` — oznacza, że cały margines wokół elementu ma wynosić 10 (kwestia jednostek zostanie omówiona w kolejnym rozdziale).
- ◆ `Margin="10,5"` — oznacza, że lewy i prawy margines mają mieć wartość 10, natomiast górny i dolny wartość 5.
- ◆ `Margin="10,5,20,30"` — oznacza, że lewy margines ma mieć wartość 10, górny 5, prawy 20, a dolny 30.

## Wyrównywanie w pionie i poziomie

Właściwość `HorizontalAlignment` pozwala ustalić sposób wyrównania w poziomie, natomiast właściwość `VerticalAlignment` stosuje się do określenia wyrównania w pionie. Obie właściwości mają typ wyliczeniowy, udostępniający następujące wartości:

- ◆ `HorizontalAlignment` — `Left`, `Center`, `Right` oraz `Stretch` (do lewej, wyśrodkowany, do prawej, rozciągnięty). Poszczególne warianty dla tej właściwości zostały przedstawione na rysunku 1.6.
- ◆ `VerticalAlignment` — `Top`, `Center`, `Bottom` oraz `Stretch` (do góry, wyśrodkowany, do dołu, rozciągnięty).

**Rysunek 1.6.**  
*Prezentacja  
wartości właściwości  
`HorizontalAlignment`*



Domyślny sposób wyrównania dla obu właściwości to `Stretch`.

Ustawienia dotyczące marginesu można oczywiście łączyć z ustawieniami dotyczącymi wyrównania. Uważny Czytelnik może zauważyć, że taka sytuacja ma miejsce na rysunku 1.6, ponieważ poszczególne elementy nie stykają się z krawędzią obszaru nadrzędnego, w którym są umieszczone. Ponadto dwa ostatnie nie stykają się ze sobą. Dla każdego z tych elementów przypisano mały margines (`Margin`). Nie zostały tu wykorzystane ustawienia dotyczące rozmiaru elementów (w szczególności `Width`). W takiej sytuacji szerokość elementu dla właściwości `Stretch` zostaje rozciągnięta na całej wolnej szerokości w panelu (z uwzględnieniem marginesu). Natomiast szerokość pozostałych elementów jest dopasowana do zawartości. Domyślny sposób wyrównania dla obu właściwości to `Stretch`.

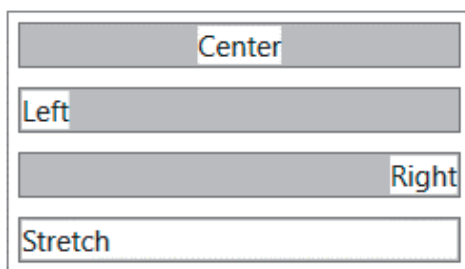
## Wyrównywanie zawartości w pionie i poziomie

W przypadku elementów mających zawartość możemy sterować także wyrównaniem ich zawartości w poziomie i pionie. Służą do tego właściwości `HorizontalContentAlignment` i `VerticalContentAlignment`. Obie właściwości mają te same typy wyliczeniowe co odpowiadające im właściwości wyrównania, to znaczy:

- ♦ `HorizontalContentAlignment` — `Left`, `Center`, `Right` oraz `Stretch` (do lewej, wyśrodkowany, do prawej, rozciągnięty). Poszczególne warianty dla tej właściwości zostały przedstawione na rysunku 1.7.
- ♦ `VerticalContentAlignment` — `Top`, `Center`, `Bottom` oraz `Stretch` (do góry, wyśrodkowany, do dołu, rozciągnięty).

### Rysunek 1.7.

Prezentacja wartości  
właściwości  
`HorizontalContentAlignment`



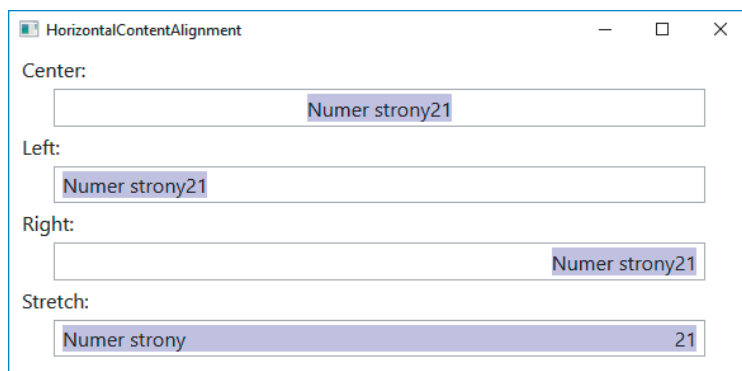
Domyślną wartością dla `HorizontalContentAlignment` jest pozycja `Left`, a dla `VerticalContentAlignment` pozycja `Top`. Inaczej jest w przypadku kontrolki `Button`, dla której domyślne ustawienie zawartości w pionie i poziomie to `Center`.

Elementy znajdujące się na rysunku 1.7 to przyciski. Dla wszystkich czterech przycisków pozostawiono domyślne wyrównanie w poziomie (czyli właściwość `HorizontalContentAlignment` ma wartość `Stretch`). Dzięki temu wszystkie cztery przyciski mają ustawioną właściwość dla tła (`Background`) na kolor szary. Każdy przycisk ma swoją zawartość. Są to teksty z celowo tu ustawionym białym kolorem tła (dla lepszego uwidocznienia obszaru zajmowanego przez zawartość). Dla wyrównania zawartości `Center` widzimy, że cała zawartość (mająca taką szerokość jak sam tekst) jest na środku. Dla dwóch pozostałych ustawień mamy odpowiednio zawartość z lewej i prawej strony. W przypadku ostatniego przycisku z ustawieniem `Stretch` zawartość (mająca białe tło) została rozciągnięta na całą szerokość przycisku. Nie dotyczy to jednak poszczególnych liter tekstu, a zatem sam tekst jest z lewej strony.

Omówię jeszcze jeden przykład, który wyraźniej demonstruje przydatność ustawienia `Stretch` dla zawartości tekstowej. Wymaga to jednak użycia bardziej złożonych komponentów. Na rysunku 1.8 znajdują się cztery listy (`ListBox`) z pojedynczymi wierszami. Ustawiono szablon dla tych list tak, aby zawartość każdego wiersza składała się z dwóch wartości (tekstu i liczby). Obie te wartości to tak naprawdę osobne elementy (bloki tekstu), mające swoje ustawienia wyrównania w poziomie (`HorizontalAlignment`) — jeden do lewej, a drugi do prawej strony. Oba jednak stanowią pewną całość w danym wierszu listy i jeśli właściwość `HorizontalContentAlignment` dla listy jest ustawiona

**Rysunek 1.8.**

Testowanie  
właściwości  
*Horizontal*  
↪ *ContentAlignment*



na *Center*, to obie części są blisko siebie (tekst i liczba) oraz umiejscawiają się na środku, natomiast dla ustawienia *Left* sytuują się z lewej strony, a dla *Right* — z prawej. Jedynie dla ustawienia *Stretch* obie części oddalają się od siebie na szerokość udostępnianą przez element nadrzędny.

Dla wszystkich czterech list (mających po jednym wierszu) wyrównanie w poziomie *HorizontalAlignment* ustawione jest na *Stretch* — wiersze tych list są rozciągnięte na szerokość panelu (z uwzględnieniem marginesu), ale nie ich zawartość, która rozciągnięta jest dopiero w ostatniej liście, z ustawieniem *Stretch* także dla właściwości *HorizontalContentAlignment*.

Do sterowania pozycją elementów można używać również właściwości *FlowDirection* (stosowana do odwrócenia przepływu wewnętrznej zawartości elementu) oraz właściwości służących do transformacji<sup>18</sup>.

Na tym zakończymy zasadniczą część teoretycznego wprowadzenia do WPF i XAML. Kolejne ważne pojęcia i zagadnienia pojawiają się w trakcie prezentacji ich praktycznego zastosowania w programach.

<sup>18</sup> Właściwość *FlowDirection* i transformacje nie będą wykorzystywane w tym podręczniku. Oba tematy omówiono m.in. w: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 97 – 107).



## Rozdział 2.

# Pierwsza aplikacja — Przywitanie

Podtrzymując tradycję w nauce programowania, zaczniemy od napisania programu, który wyświetli tekst „Witaj, świecie!”. Ale żeby nie było tak łatwo — wszak sam napis dla WPF to przecież żadne wyzwanie — tekst wyświetli się dopiero po kliknięciu przycisku. Wzbogacimy to przywitanie jeszcze jednym przyciskiem, który pokaże aktualny czas. W trakcie tych eksperymentów wstępnie poznamy środowisko Visual Studio dla aplikacji WPF, co stanowi główny cel niniejszego rozdziału. Ponadto podczas tworzenia przycisków przetestujemy różne sposoby definiowania kontroltek.

## 2.1 Warstwa prezentacji, czyli jak ma wyglądać

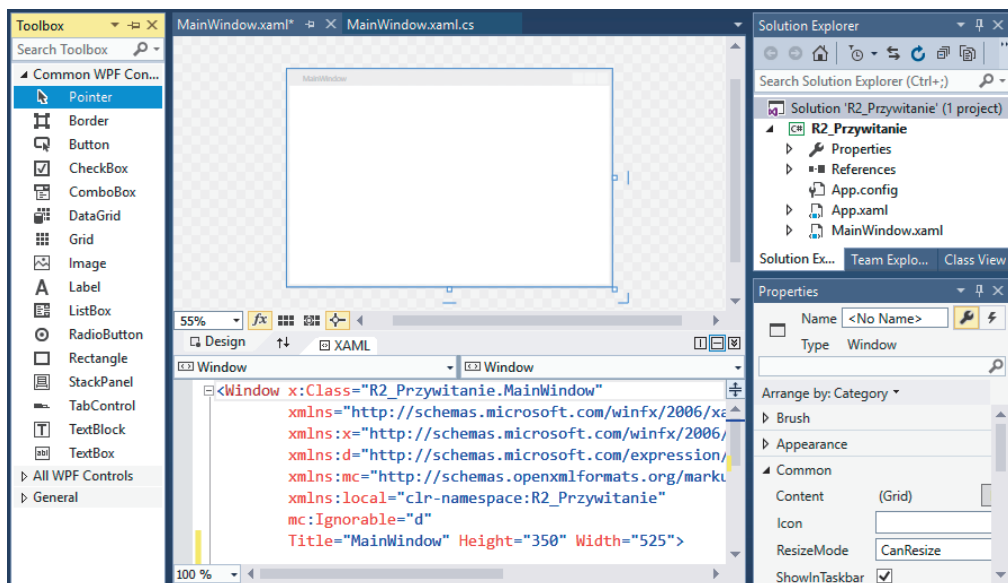
W Visual Studio po utworzeniu nowego projektu (*File/New/Project*) i wybraniu aplikacji WPF (*WPF App*) pojawia się okno, którego domyślny układ przedstawia rysunek 2.1.

Okno *Toolbox* to panel narzędziowy z wykazem kontroltek. To okno może nie być widoczne po uruchomieniu środowiska — można je wyświetlić, klikając na lewym pasku napis *Toolbox* lub poprzez opcję z menu środowiska *View/Toolbox*.

Środkowe okno zawiera kod. Jeśli aktywna jest zakładka *MainWindow.xaml*, okno podzielone jest na dwa okna: domyślnie u góry jest widok *Design*, a poniżej okno *XAML*.

Okno *Design* (okno z widokiem *Design*) pozwala nanosić kontrolki z okna *Toolbox* oraz prezentuje wygląd kontroltek po zmianach dokonanych w oknie *Properties* lub bezpośrednio w kodzie *XAML*.

Okno *XAML* przedstawia kod XAML zawierający deklaracyjny opis warstwy prezentacji programu.



**Rysunek 2.1.** Domyślny układ okien aplikacji WPF

Okno *Solution Explorer* zawiera wykaz plików aplikacji (podobnie jak w aplikacjach konsolowych).

Okno *Properties* umożliwia definiowanie właściwości (przycisk z kluczem) i zdarzeń (przycisk z błyskawicą). W tym oknie pokazywane są właściwości i zdarzenia dla aktywnej kontrolki w widoku *Design* (na rysunku 2.1 aktywne jest całe okno budowanej aplikacji).

Na rysunku widoczna jest jeszcze jedna nieaktywna zakładka w środkowym górnym oknie dla pliku *MainWindow.xaml.cs* — jest to plik z *code-behind*, czyli zapleczem kodu XAML, gdzie zajrzemy w kolejnym podrozdziale.

## Definiujemy pierwszy przycisk

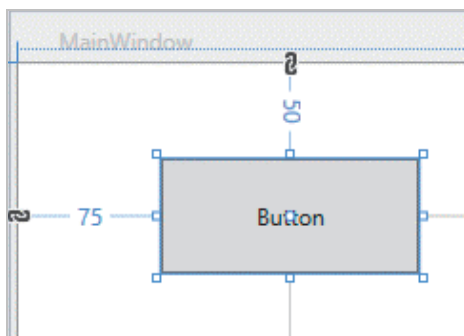
Pierwszą kontrolkę dodamy do naszego okna w nowym projekcie z użyciem okien *Toolbox* i *Properties*. Umieścimy przycisk z podpisem *Start*. Wybierz z okna *Toolbox* kontrolkę o nazwie *Button* i nanieś ją w obszar okna *Design*. Po umieszczeniu kontrolki w widoku *Design* pojawia się przycisk podobny do tego z rysunku 2.2.

W widoku *Design* możemy z użyciem myszy zmieniać położenie przycisku oraz jego wysokość i szerokość. Wykonaj kilka takich eksperymentów, obserwując jednocześnie konsekwencje tych zmian w kodzie XAML. Dla przycisku prezentowanego na rysunku 2.2 kod wygląda następująco:

```
<Button Content="Button" HorizontalAlignment="Left"
        Height="60" Margin="75,50,0,0" VerticalAlignment="Top" Width="135"/>
```

**Rysunek 2.2.**

Przycisk  
w widoku Design



Przeanalizujemy ten kod:

- ♦ `Button` — na samym początku jest nazwa znacznika, tu jest to `Button`, czyli przycisk. W dalszej części definiowane są atrybuty tego znacznika.
- ♦ `Content="Button"` — tekst wyświetlany na przycisku to `Button`.
- ♦ `HorizontalAlignment="Left"` — ten atrybut decyduje o wyrównaniu kontrolki w poziomie, tu do lewej.
- ♦ `Height="60"` — wysokość przycisku 60.
- ♦ `Margin="75,50,0,0"` — marginesy względem okna (a dokładniej względem komórki panelu `Grid`, wypełniającego okno).
- ♦ `VerticalAlignment="Top"` — ten atrybut decyduje o wyrównaniu kontrolki w pionie, tu do góry.
- ♦ `Width="135"` — szerokość przycisku 135.



Uwaga

Dla podanych wyżej wartości liczbowych domyślną jednostką jest piksel (`px`). Nie są to fizyczne piksele, lecz tak zwane **piksele niezależne od urządzenia**. Jeden taki piksel to 1/96 cala niezależnie od ustawienia DPI ekranu. Dane te są wartościami typu `double` (mogą być ułamkowe). Piksele niezależne od urządzenia stanowią podstawową jednostkę miary WPF.

Prezentowany kod XAML nie opisuje wszystkich informacji o tym przycisku, nie ma tu przykładowo atrybutu dla koloru tła przycisku albo dla rodzaju czcionki. Przycisk dla tych nieopisanych atrybutów przyjmuje wartości domyślne. Jeśli zmienimy te ustawienia, to pojawią się nowe zapisy w definicji znacznika. I tym się teraz zajmiemy.

Sprawdź w oknie *Design*, czy jest aktywna kontrolka z przyciskiem. Jeśli nie jest aktywna, kliknij ją jeden raz (tylko jeden raz), tak aby była oznaczona podobnie jak na rysunku 2.2. Następnie w oknie *Properties* na samej górze w polu *Name* wpisz nazwę tej kontrolki. Jedną z konwencji podczas ustalania nazwy kontrolki polega na dodawaniu przedrostka skrótu kontrolki, np. dla przycisku `btn`, dla etykiety `lbl`, a dla pola tekstowego `txt`. Dla przycisku nadamy nazwę `btnStart` (rysunek 2.3).

**Rysunek 2.3.**

Wpisanie  
nazwy przycisku  
w oknie Properties

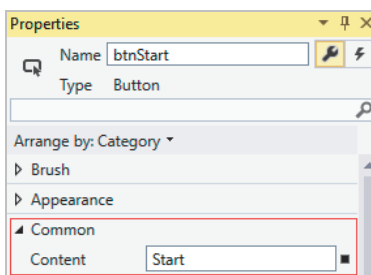


Po wpisaniu nazwy przycisku i potwierdzeniu (np. klawiszem *Enter*) widzimy efekt tej zmiany także w kodzie XAML. Obecnie definicja atrybutu dla nazwy przycisku wygląda tak: `x:Name="btnStart"`. Nazwę kontrolki wykorzystamy podczas pisania *code-behind* w kolejnym podrozdziale.

Teraz zmienimy zawartość przycisku, czyli wpiszęmy nowy tekst, jaki ma być wyświetlany na przycisku zamiast słowa *Button*. Poszukaj w oknie *Properties* właściwości *Content* (jest w grupie *Common*) i zmień tekst na *Start* (rysunek 2.4).

**Rysunek 2.4.**

Zmiana tekstu  
na przycisku  
w oknie Properties

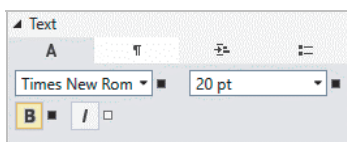


Po dokonaniu zmiany tekstu na przycisku i potwierdzeniu widzimy efekt tej zmiany także w kodzie XAML. Teraz definicja atrybutu dla zawartości przycisku ma postać: `Content="Start"`. Zmiana jest widoczna również w oknie *Design*.

Zmienimy jeszcze czcionkę i rozmiar. W tym celu poszukaj w oknie *Properties* właściwości *Text* i zmień czcionkę na *Times New Roman*, wielkość 20 punktów, pogrubiona („B” jak *Bold*), zgodnie z rysunkiem 2.5.

**Rysunek 2.5.**

Zmiana czcionki  
i rozmiaru tekstu  
w oknie Properties



Zmianę tę można zobaczyć w oknie *Design* i w oknie *XAML* (na końcu pojawiają się trzy nowe atrybuty):

```
<Button x:Name="btnStart" Content="Start" HorizontalAlignment="Left"
        Height="60" Margin="75,50,0,0" VerticalAlignment="Top" Width="135"
        FontFamily="Times New Roman" FontSize="26.667" FontWeight="Bold"/>
```

Rozmiar czcionki w oknie *Properties* jest podany w punktach (*pt*), a w kodzie XAML (jeśli nie podano jednostki) domyślnie zapisuje się go w pikselach (*px*).



Jak już wspominałam, 1 piksel niezależny od urządzenia (*px*) to 1/96 cala. Natomiast 1 punkt (*pt*) to 1/72 cala. Łatwo policzyć, że 1 piksel = 96/72 × liczba punktów.

Możesz w celach testowych zmienić kod XAML dla rozmiaru czcionki: `FontSize="20 pt"`. Wówczas czcionka nie ulegnie zmianie (będzie zmieniona jedynie jednostka, a nie rozmiar czcionki).

## Definiujemy drugi przycisk

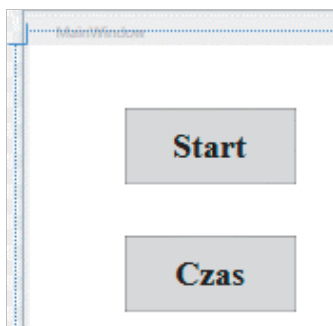
Utworzymy nowy przycisk, ale tym razem nie będziemy używać okien *Toolbox* ani *Properties*. Założmy, że chcemy mieć pod przyciskiem *Start* nowy przycisk, z tymi samymi atrybutami oprócz nazwy (to jest konieczne), tekstu na przycisku i oczywiście położenia. Będziemy pracować wyłącznie w oknie *XAML*. W tym celu skopiuj w oknie *XAML* całą definicję przycisku *Start*, wklej ją poniżej i w tej przekopiowanej wersji zmień atrybuty `x:Name`, `Content` oraz `Margin` (wartość dla atrybutu `Margin` musisz dopasować do swojego położenia przycisku *Start*), tak aby po zmianach definicja nowego przycisku miała postać zbliżoną do poniższej:

```
<Button x:Name="btnTime" Content="Czas" HorizontalAlignment="Left" Height="60"
        Margin="75,150,0,0" VerticalAlignment="Top" Width="135"
        FontFamily="Times New Roman" FontSize="26.667" FontWeight="Bold"/>
```

Po wprowadzeniu zmian powinniśmy zobaczyć w oknie *Design* oba przyciski, podobnie jak na rysunku 2.6.

### Rysunek 2.6.

Wygląd okna  
*Design* po dodaniu  
dwóch przycisków



Operację kopiowania przycisku (lub innego elementu) można wykonać także w oknie *Design* z użyciem myszy (poprzez opcje *Copy* i *Paste* w menu kontekstowym lub *Ctrl+C* i *Ctrl+V*). Wówczas nowy element (po wklejeniu) może zasłonić źródłowy. W takim przypadku należy go myszą przesunąć i przenieść w odpowiednie miejsce.

Nasz program już jakoś wygląda. Zakładaliśmy, że będą dwa przyciski i są. Tylko że poza wyglądem nic się nie dzieje. Nie mamy jeszcze „logiki” dla naszego programu. Zdefiniujemy dla przycisków bardzo proste działanie w kolejnym podrozdziale.

## 2.2 Code-behind, czyli jak ma działać

W *code-behind* zdefiniujemy działanie obu umieszczonych przycisków. Pierwszy z nich (*Start*) po kliknięciu ma spowodować pojawienie się w osobnym okienku komunikatu „Witaj, świecie!”. Natomiast po najechnaniu myszą w obszar drugiego przycisku napis

na nim ma się zmienić na aktualną godzinę, a po cofnięciu myszy ma zostać przywrócony poprzedni tekst („Czas”).

## Definiujemy działanie dla pierwszego przycisku

W oknie *Design* kliknij dwa razy lewym klawiszem myszy kontrolkę z przyciskiem *Start*. Wówczas znajdziesz się w oknie *MainWindow.xaml.cs*, w którym będzie (wpisany automatycznie) następujący kod:

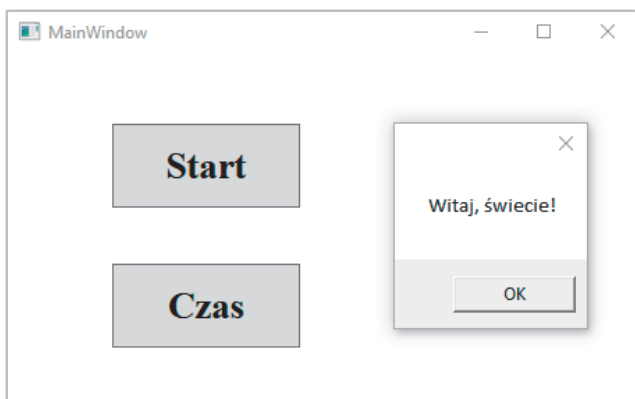
```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void btnStart_Click(object sender, RoutedEventArgs e)
    {
    }
}
```

Wewnątrz metody `btnStart_Click`, obsługującej zdarzenie kliknięcia przycisku (w pustej linii między klamrami), wpisz kod: `MessageBox.Show("Witaj, świecie!");`.

Program można już uruchomić (*F5*) i przetestować w obecnej wersji, z działającym na razie tylko przyciskiem *Start* (rysunek 2.7).

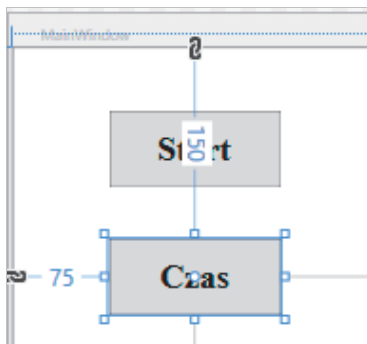
**Rysunek 2.7.**  
Testowanie działania  
przycisku *Start*



## Definiujemy działanie dla drugiego przycisku

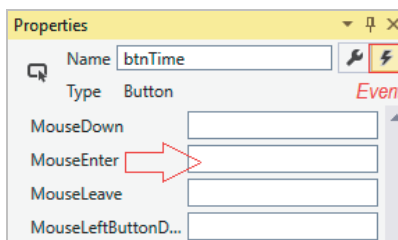
Dla drugiego przycisku zrobimy obsługę dwóch innych zdarzeń, mianowicie najechania myszą i „odjechania” myszą (czyli opuszczenia obszaru danego elementu). Po najechaniu myszą ma się pojawić aktualna godzina, a po odsunięciu myszy ma wrócić napis „Czas”. Tym razem skorzystamy z okna *Properties*. W pierwszej kolejności w widoku *Design* kliknij raz lewym klawiszem myszy kontrolkę przycisku *Czas*, tak aby kontrolka ta była oznaczona jako wybrana (aktywna), jak na rysunku 2.8.

**Rysunek 2.8.**  
Wybrana kontrolka  
— przycisk Czas



Następnie w oknie *Properties* kliknij w przycisk z błyskawicą (Event), odszukaj zdarzenie *MouseEnter* i kliknij puste pole przy nazwie tego zdarzenia (wskazane strzałką na rysunku 2.9) dwa razy lewym klawiszem myszy.

**Rysunek 2.9.**  
Definiowanie  
zdarzenia  
*MouseEnter*



Wówczas w oknie *MainWindow.xaml.cs* pojawi się pusta definicja nowej metody *btnTime\_MouseEnter*, obsługującej zdarzenie najechnania myszą. Musimy napisać kod tej metody — dopiszemy tam dwie linie i wtedy cała definicja metody będzie wyglądać następująco:

```
private void btnTime_MouseEnter(object sender, MouseEventArgs e)
{
    DateTime data = DateTime.Now;           // Odczytanie bieżącej daty
    btnTime.Content = data.ToString("T");    // Przypisanie łańcucha znakowego
                                           // prezentującego godzinę
}
```

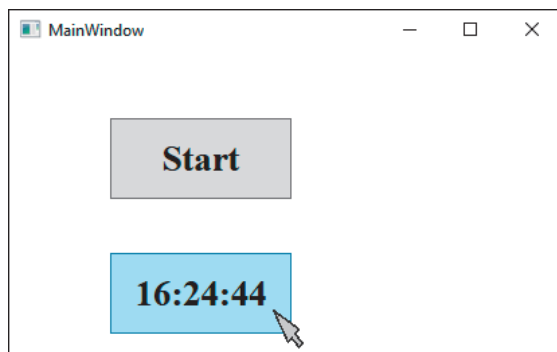
Podobnie wykonamy obsługę zdarzenia odsunięcia myszy. Poszukaj w oknie *Properties* na liście zdarzeń zdarzenia *MouseLeave* i kliknij dwa razy puste pole przy jego nazwie, a następnie uzupełnij definicję metody obsługującej to zdarzenie tak, aby miała taką oto postać:

```
private void btnTime_MouseLeave(object sender, MouseEventArgs e)
{
    btnTime.Content = "Czas";
}
```

Teraz możemy testować działanie programu. Po najechnaniu myszą widzimy aktualną godzinę, a po opuszczeniu obszaru przycisku pojawia się ponownie napis „Czas” (rysunek 2.10).

**Rysunek 2.10.**

Testowanie  
działania  
przycisku Czas



Na koniec zobaczymy, jak obecnie wygląda kod XAML dla obu przycisków:

```
<Button x:Name="btnStart" Content="Start" HorizontalAlignment="Left"
        Height="60" Margin="75,50,0,0" VerticalAlignment="Top" Width="135"
        FontFamily="Times New Roman" FontSize="26.667" FontWeight="Bold"
        Click="btnStart_Click"/>
<Button x:Name="btnTime" Content="Czas" HorizontalAlignment="Left" Height="60"
        Margin="75,150,0,0" VerticalAlignment="Top" Width="135"
        FontFamily="Times New Roman" FontSize="26.667" FontWeight="Bold"
        MouseEnter="btnTime_MouseEnter" MouseLeave="btnTime_MouseLeave"/>
```

Jak widać, zostały dopisane atrybuty dla zdarzeń `Click` (dla pierwszego przycisku) oraz `MouseEnter` i `MouseLeave` (dla drugiego przycisku).



Uwaga

Co zrobić, jeśli klikając w oknie *Design* lub *Properties* (w zakładce dla uchwytów zdarzeń), spowodujemy, że do kodu (XAML i *code-behind*) automatycznie dopiszą się elementy, jakich nie planowaliśmy wpisywać? Można się z tego tradycyjnie wycofać, klikając dwukrotnie przycisk *Undo* (cofnij). Można też samodzielnie usunąć ślady po automatycznym wpisie w obu miejscach (tzn. pustą definicję metody w *code-behind* oraz dodany fragment w kodzie XAML). Jeżeli przykładowo został wygenerowany kod metody obsługującej zdarzenie `Click` dla przycisku `btnTime` (w zadaniu miało takiej nie być), to należy usunąć z pliku `MainWindow.xaml.cs` definicję metody `btnTime_Click` oraz w kodzie XAML fragment definicji tego przycisku: `Click="btnTime_Click"`.

## 2.3 Zadania

W prostym zadaniu z dwoma przyciskami wykorzystaliśmy zaledwie kilka właściwości opisujących przyciski. Na przykładzie przycisków można poeksperymentować z definiowaniem innych właściwości. Kolejny podrozdział zawiera dość szczegółowe wskazówki do zadań, zalecam jednak, aby najpierw próbować samodzielnie rozwiązać zadania, szukając pomocy jedynie w środowisku (np. przeglądając dostępne właściwości dla danej kontrolki) i w dokumentacji MSDN.



## Zadanie 2.1

Używając okna *Properties*, zmodyfikuj przycisk *Start* w wykonanym programie: zmień kolor tła, kolor obramowania oraz kolor tekstu, ustawiając odpowiednio właściwości w kategorii *Brush* (pędzel) — *Background*, *BorderBrush* oraz *Foreground* — na dowolne kolory.

## Zadanie 2.2

Zmień działanie programu tak, aby kliknięcie przycisku *Start* spowodowało (oprócz wyświetlenia okienka z napisem „Witaj, świecie!”) pokazanie tego przycisku w postaci półprzezroczystej (o przezroczystości decyduje właściwość *Opacity*). Natomiast po zamknięciu okienka z tekstem przywitania przycisk ma być wyświetlony jak poprzednio.

## Zadanie 2.3

Kolejne zadanie będzie podobne do poprzedniego, ale tym razem przycisk *Start* po kliknięciu go ma zostać ukryty (ang. *hidden*), a po zamknięciu okna z przywitanem ma być ponownie widoczny (ang. *visible*). W tym zadaniu użyj właściwości *Visibility*.

## Zadanie 2.4

Za pomocą okna *Properties* zmodyfikuj przycisk *Start* w wykonanym programie, zmieniając właściwość *IsEnabled* (w grupie *Common*) na *false* (tzn. odznaczając pole). Przycisk stanie się wówczas nieaktywny. Ponadto dodaj do programu trzeci przycisk, którego kliknięcie spowoduje uaktywnienie przycisku *Start*.

## Zadanie 2.5

Jakiś czas temu popularny był programik-żart rozsyłany między innymi przez szefów do pracowników. Zawierał on pytanie: „Czy chcesz podwyżkę?”. Poniżej zaś były dwa przyciski: *Tak* i *Nie*. Żart polegał na tym, że gdy próbowaliśmy najechać myszą na przycisk *Tak*, on oddalał się (uciekał od wskaźnika myszy) albo, w innej wersji programu, zamieniał się miejscami z przyciskiem *Nie*. Twoim zadaniem będzie wykonanie takiego programu w wersji ze zmianą położenia przycisków, to znaczy w chwili najechania myszą na przycisk *Tak* oba przyciski mają zmienić swoje położenie.

## 2.4 Wskazówki do zadań

Ponieważ dopiero zaczynamy naukę, z myślą o Czytelnikach początkujących nie tylko w zakresie WPF, ale i programowania w ogóle umieściłam tutaj dość szczegółowe wskazówki dla wybranych zadań (z fragmentami rozwiązania).

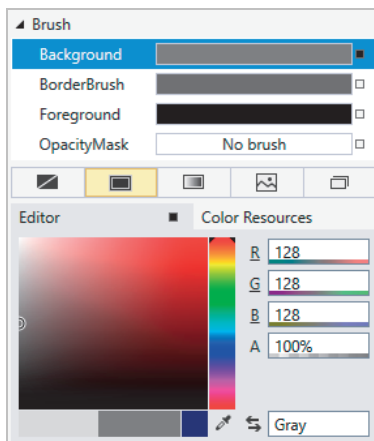
## Wskazówki do zadania 2.1

Zadanie 2.1 właściwie nie wymaga wskazówek. Wykorzystam jednak to miejsce do krótkiej wzmianki na temat kolorów. W WPF kolory są definiowane z użyciem struktury `Color`. Struktura ta opisuje kolor na podstawie czterech kanałów: A (*alfa*), R (ang. *red*, czyli czerwony), G (ang. *green*, czyli zielony), B (ang. *blue*, czyli niebieski). Kanał alfa domyślnie przyjmuje maksymalną wartość, która oznacza brak przezroczystości barwy. WPF umożliwia stosowanie dwóch przestrzeni barw<sup>1</sup>: sRGB i scRGB. W przestrzeni sRGB dla każdego koloru składowego (R, G i B) można ustalić wartość z zakresu od 0 do 255. Natomiast w przestrzeni scRGB poszczególne barwy składowe (R, G i B) określa się za pomocą liczb rzeczywistych z przedziału od 0 do 1. W podręczniku będziemy wykorzystywać tylko przestrzeń sRGB.

W oknie *Properties* kolor dla tła i pozostałych elementów można wpisać na kilka sposobów. Można w tym celu użyć edytora, wybierając barwę lub wpisując jej dziesiętny kod dla każdego kanału (R, G i B). Dodatkowo można ustalić wartość kanału alfa prezentowanego w oknie *Properties* poprzez wartość z zakresu od 0% do 100% (0% oznacza pełną przezroczystość, a 100% brak przezroczystości). Przykładowo wpisanie dla każdego koloru składowego (R, G i B) wartości 0 (i 100% dla kanału alfa) spowoduje uzyskanie barwy czarnej. Można także w osobnym polu wpisać kod szesnastkowy danej barwy według formatu `#aarrggbb` (np. kolor czarny ma kod `#FF000000`) lub `#rrggbb` (kolor czarny wówczas zapisuje się jako `#000000`, a kanał alfa przyjmuje wartość domyślną). W tym samym polu jest również możliwość wpisania angielskiej nazwy koloru (np. `Gray`) dla barw predefiniowanych, co można zobaczyć na rysunku 2.11. Na stronie dokumentacji MSDN, opisującej klasę `Brushes` (w dolnej części), znajduje się tabela prezentująca predefiniowane kolory wraz z ich kodami szesnastkowymi<sup>2</sup>. W tym podręczniku kolory zazwyczaj definiowane są w kodzie XAML z użyciem predefiniowanych barw, np. `Background="Gray"`.

### Rysunek 2.11.

Definiowanie koloru w oknie *Properties* (wpisana nazwa „Gray” dla predefiniowanego koloru zostanie podmieniona na szesnastkowy kod tego koloru: `#FF808080`)



<sup>1</sup> Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 513 – 514).

<sup>2</sup> Opis klasy `Brushes` („Brushes Class”): [https://msdn.microsoft.com/en-us/library/system.windows.media.brushes\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.media.brushes(v=vs.110).aspx).

Choć w kodzie XAML nie zawsze jest to widoczne (dzięki konwerterom, które zostaną przedstawione później), kolory w WPF są ustalane za pośrednictwem tak zwanych pędzli, czyli obiektów klas wywodzących się z klasy `Brush`. W tym podręczniku najczęściej jest wykorzystywany (malujący jednolitym kolorem) pędzel `SolidColorBrush`. W jednym z dalszych rozdziałów zostanie użyty także pędzel `LinearGradientBrush`.

## Wskazówki do zadania 2.2

Zmiany dokonaj w *code-behind*, w metodzie obsługującej zdarzenie kliknięcia przycisku `Start`. Przed wywołaniem metody wyświetlającej komunikat „Witaj, świecie!” ustaw właściwość `Opacity` na 50% (`btnStart.Opacity = 0.5;`), a po jej wywołaniu z powrotem na 100% (`btnStart.Opacity = 1.0;`).

## Wskazówki do zadania 2.3

Wykonaj zmianę, podobnie jak w poprzednim zadaniu, w metodzie obsługującej zdarzenie kliknięcia przycisku `Start`. Przed wywołaniem metody wyświetlającej komunikat „Witaj, świecie!” ustaw właściwość `Visibility` na jedną z wartości typu wyliczeniowego `Visibility`, mianowicie `Hidden` (ukryty) (`btnStart.Visibility = Visibility.Hidden;`), a po jej wywołaniu przywróć poprzednie ustawienie (`btnStart.Visibility = Visibility.Visible;`)<sup>3</sup>.

## Wskazówki do zadania 2.5

Pytanie „Czy chcesz podwyżkę?” (lub inne, jak wolisz) wpisz za pomocą okna *Toolbox*, używając kontrolki `Label` (etykieta). Poniżej umieść oba przyciski z podpisami *Tak* i *Nie* (najlepiej o różnych kolorach tła). W tym zadaniu należy obsługiwać zdarzenie najechania myszą `MouseEnter` w obszar przycisku *Tak* (i tylko to jedno zdarzenie). W metodzie obsługującej to zdarzenie należy dokonać podmiany położenia przycisków. Można w tym celu wykorzystać właściwość `Margin`, na przykład:

```
var tmpMargin = btnYes.Margin; // Zapamiętanie początkowego położenia przycisku Tak
btnYes.Margin = btnNo.Margin;  // Przycisk Tak przyjmuje położenie przycisku Nie
btnNo.Margin = tmpMargin;      // Przycisk Nie przyjmuje początkowe położenie przycisku Tak
```

Słowo kluczowe `var` użyte w pierwszej linii prezentowanego kodu pozwala nam ominąć nazwę typu danego obiektu w jego definicji — kompilator „domyśli się”, jaki jest typ tego obiektu, na podstawie jego zawartości. I tak przykładowo w przypadku marginesu jest to struktura `Thickness`.

---

<sup>3</sup> Typ wyliczeniowy `Visibility` ma jeszcze jedną wartość: `Collapsed`. Element, któremu przypisano wartość `Hidden`, jest niewidoczny, ale ma zarezerwowane miejsce w elemencie nadrzędnym. Element, któremu przypisano wartość `Collapsed`, jest niewidoczny i żadnego miejsca nie rezerwuje. Różnica między wartościami `Hidden` a `Collapsed` stanie się wyraźniejsza, gdy Czytelnik zapozna się z panelami (w rozdziale 4.).



## Rozdział 3.

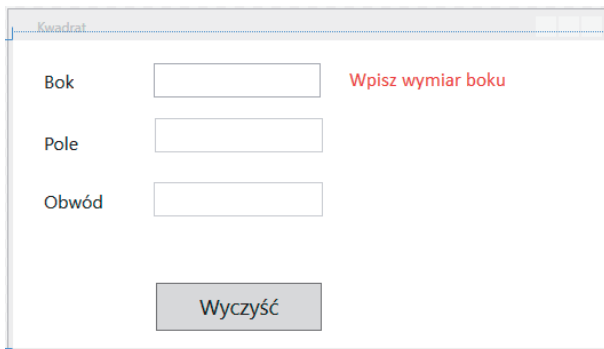
# Podstawowe kontrolki

W tym rozdziale zostaną zaprezentowane proste kontrolki, takie jak `Label` (etykieta), `TextBox` (pole tekstowe) czy `ComboBox` (lista rozwijana). Celem jest nabycie sprawności w definiowaniu kontroltek. Dopiero w kolejnym rozdziale wykorzystamy panele, które pozwalają organizować układ kontroltek w oknie aplikacji. A jeszcze później poznamy wiązanie danych i inne mechanizmy pozwalające omawiane tu funkcjonalności usprawnić.

## 3.1 Kontrolki Label, TextBox, Button — aplikacja Kwadrat

Wykonamy prosty program, który oblicza obwód i powierzchnię kwadratu z użyciem kontroltek `Label`, `TextBox` oraz `Button`. W tym celu otwórz nowy projekt WPF. W oknie `XAML` podmień tytuł okna na „Kwadrat”, to znaczy ustaw atrybut `Title="Kwadrat"`. Następnie, korzystając z okna `Toolbox`, dodaj kontrolki według wzoru umieszczonego na rysunku 3.1.

**Rysunek 3.1.**  
Okno aplikacji  
Kwadrat  
w widoku Design



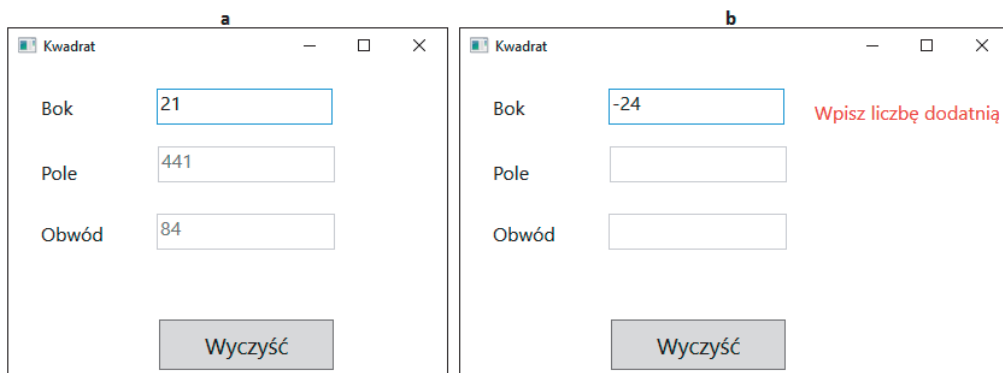
Gdzie:

- ◆ `Bok` to `Label` (etykieta) z atrybutem `x:Name="lblBok"` i atrybutem/właściwością `Content="Bok"`.

- ◆ *Pole* to `Label` z atrybutem `x:Name="lblPole"` i atrybutem/właściwością `Content="Pole"`.
- ◆ *Obwód* to `Label` z atrybutem `x:Name="lblObwod"` i atrybutem/właściwością `Content="Obwód"`.
- ◆ Kontrolka (pusta ramka) po prawej stronie etykiety *Bok* to `TextBox` (pole tekstowe) z atrybutem `x:Name="txtBok"`.
- ◆ Kontrolka po prawej stronie etykiety *Pole* to `TextBox` (`x:Name="txtPole"`) — należy dla tej kontrolki ustawić właściwość `IsEnabled` na `false` (odznaczyć w oknie *Properties*).
- ◆ Kontrolka po prawej stronie etykiety *Obwód* to `TextBox` (`x:Name="txtObwod"`) — tu także należy ustawić właściwość `IsEnabled` na `false`.
- ◆ Kontrolka w prawym górnym rogu okna z tekstem *Wpisz wymiar boku* to `Label` z atrybutem `x:Name="lblKomunikat"` i atrybutem/właściwością `Content="Wpisz wymiar boku"`. W tym miejscu będziemy wyświetlać komunikat o błędnej danej. Ustaw kolor czcionki (właściwość `Foreground`) na czerwony.
- ◆ Przycisk z napisem *Wyczyść* — to kontrolka `Button` (`x:Name="btnWyczyśc"`) z atrybutem/właściwością `Content="Wyczyść"`.

W dalszej części napiszemy *code-behind* dla dwóch kontroltek, tak aby wpisanie liczby dodatniej w polu tekstowym *Bok* (to często stosowany skrót myślowy, chodzi o pole tekstowe opisane etykietą *Bok*, czyli `txtBok`) spowodowało umieszczenie obliczonej powierzchni i obwodu kwadratu w kolejnych polach. Jeśli użytkownik wprowadzi liczbę ujemną lub tekst niedający się konwertować do liczby `double`, ma zostać wyświetlony po prawej stronie tego pola komunikat o treści: „Wpisz liczbę dodatnią” w kolorze czerwonym. Użycie przycisku *Wyczyść* ma wyczyścić wszystkie pola tekstowe oraz przypisać do etykiety z komunikatem tekst „Wpisz wymiar boku”.

Jeśli w polu oznaczonym etykietą *Bok* wpisano prawidłową wartość, program ma działać jak na przykładowym zrzucie ekranu (rysunek 3.2a). W przypadku błędnie wprowadzonej danej program ma wyświetlić stosowny komunikat (rysunek 3.2b).



Rysunek 3.2. Działanie aplikacji Kwadrat

Aby wykonać zadanie, musimy obsłużyć dwa zdarzenia: zmiany w polu tekstowym, w którym ma zostać wprowadzony wymiar boku, oraz kliknięcia przycisku *Wyczyść*.

## Obsługa zdarzenia zmiany w polu tekstowym

W widoku *Design* kliknij dwa razy lewym klawiszem myszy pole tekstowe do wprowadzania wymiaru boku. Wówczas w pliku *MainWindow.xaml.cs* pojawi się kod z definicją metody `txtBok_TextChanged`. Jest to definicja metody obsługującej zdarzenie zmiany tekstu w polu tekstowym. We wnętrzu (w klamrach) należy wpisać kod, w którym zostanie wykonane sprawdzenie, czy wpisany tekst daje się przekonwertować do liczby typu `double`, a ponadto czy ta liczba jest dodatnia. W przypadku gdy zostanie wprowadzona prawidłowa wartość liczbowa, mają zostać wykonane stosowne obliczenia. Zdefiniowana metoda `txtBok_TextChanged` powinna mieć następujący kod:

```
private void txtBok_TextChanged(object sender, TextChangedEventArgs e)
{
    double bok; // Wartość tej zmiennej będzie ustalona w metodzie TryParse (argument out)
    if (double.TryParse(txtBok.Text, out bok) && bok >= 0)
    {
        txtPole.Text = Math.Pow(bok, 2.0).ToString();
        txtObwod.Text = (bok * 4).ToString();
        lblKomunikat.Content = String.Empty;
    }
    else
    {
        lblKomunikat.Content = "Wpisz liczbę dodatnią";
    }
}
```

Użyto metody `TryParse` z dwoma argumentami: pierwszy to tekst, a drugi to liczba `double` (jako `out`)<sup>1</sup>. Jeśli podany tekst można przekonwertować do wartości typu `double`, wówczas metoda zwraca `true`, a argument z modyfikatorem `out` przyjmie wartość tej liczby. W przeciwnym wypadku metoda zwraca `false`. Po wpisaniu powyższego kodu można sprawdzić działanie programu na tym etapie.

## Obsługa zdarzenia kliknięcia przycisku Wyczyść

Teraz obsłużymy zdarzenie kliknięcia przycisku *Wyczyść*. Moglibyśmy zrobić podobnie (klikając przycisk *Wyczyść* dwukrotnie lewym klawiszem myszy), ale tym razem zrobimy inaczej. Klikamy tylko raz przycisk *Wyczyść*, tak aby zaznaczyć tę kontrolkę jako aktywną. Następnie w oknie *Properties* klikamy przycisk z błyskawicą (`Event`) i szukamy zdarzenia `Click`. Klikamy dwa razy puste pole obok nazwy `Click`, a wówczas zostanie wygenerowany kod dla metody `btnWyczysc_Click`, obsługującej zdarzenie `Click`. Wpisz między klamrami ciało tej metody. Kompletna definicja metody powinna mieć następującą postać:

---

<sup>1</sup> Modyfikator `out` umożliwia wykorzystanie danego argumentu jako jednego z wyjść (rezultatów) metody. Opis metody `TryParse` w dokumentacji MSDN („Double.TryParse Method”): [https://msdn.microsoft.com/en-us/library/994c0zb1\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/994c0zb1(v=vs.110).aspx).

```
private void btnWyczysc_Click(object sender, RoutedEventArgs e)
{
    txtBok.Text = String.Empty;
    txtPole.Text = String.Empty;
    txtObwod.Text = String.Empty;
    lblKomunikat.Content = "Wpisz wymiar boku";
}
```

Program można już uruchomić i przetestować.

## 3.2 Kontrolki ComboBox i CheckBox — aplikacja Rysowanie kwadratu

Rozwińmy poprzedni projekt o możliwość rysowania kwadratu. Za pomocą okna *Toolbox* wykonaj następujące zmiany:

- ◆ Zmień w kodzie XAML tytuł okna na „Rysowanie kwadratu” i zwiększ rozmiar okna głównego, np.: `Height="500" Width="700"`.
- ◆ Dodaj kontrolkę `Label` z właściwością `Content = "Kolor"`.
- ◆ Dodaj kontrolkę `ComboBox` (lista rozwijana) z właściwością `SelectedIndex="0"` (wstępnie wybrana pozycja). Kontrolka ta ma mieć następujące pozycje: `Black`, `Red`, `Yellow`, `Green`, `Blue`.
- ◆ Dodaj kontrolkę `CheckBox` z właściwością `Content="Półprzezroczysty"`.
- ◆ Dodaj element `Rectangle` — najpierw nanieś z okna *Toolbox*, a następnie zmień w kodzie XAML właściwości: `Height="0" Width="0"` (po uruchomieniu programu kwadrat będzie niewidoczny).
- ◆ Dodaj przycisk *Rysuj*. Kliknięcie tego przycisku ma spowodować wyświetlenie kwadratu według zadanych parametrów, czyli dla elementu `Rectangle` mają zostać ustawione odpowiednio (tzn. według tego, co wskazał użytkownik) takie właściwości jak: `Height`, `Width`, `Stroke` (kolor konturu), `Fill` (kolor wypełnienia), `Opacity` (nieprzezroczystość, liczba z przedziału 0 – 1).

Przykładowy wygląd okna w widoku *Design* przedstawia rysunek 3.3.

### Rysunek 3.3.

Okno aplikacji  
Rysowanie  
kwadratu  
w widoku *Design*



Spośród wymienionych nowych elementów wyjaśnień wymaga kontrolka `ComboBox` oraz obsługa zdarzenia kliknięcia przycisku *Rysuj*.

## Tworzenie listy rozwijanej (`ComboBox`) dla kolorów

Dodaj kontrolkę `ComboBox`, używając okna *Toolbox*. Automatycznie zostanie wygenerowany kod XAML dla tej kontrolki, podobny do poniższego:

```
<ComboBox HorizontalAlignment="Left" Height="30"
    Margin="123,180,0,0" VerticalAlignment="Top" Width="145"/>
```

Wpiszemy nazwę kontrolki `"cmbKolory"` oraz dodamy zgodnie z poleceniem atrybut wskazujący wstępnie wybraną pozycję `SelectedIndex="0"`. Ponadto dopiszemy pozycje listy rozwijanej dla poszczególnych kolorów. W tym celu musimy osobno wyodrębnić znacznik końca `</ComboBox>`, tak aby wewnątrz można było zdefiniować listę. Po tych zmianach definicja listy rozwijanej w kodzie XAML będzie wyglądać tak:

```
<ComboBox x:Name="cmbKolory" SelectedIndex="0" HorizontalAlignment="Left"
    Height="30" Margin="123,180,0,0" VerticalAlignment="Top" Width="145">
    <ComboBoxItem Content="Black"></ComboBoxItem>
    <ComboBoxItem Content="Red"></ComboBoxItem>
    <ComboBoxItem Content="Yellow"></ComboBoxItem>
    <ComboBoxItem Content="Green"></ComboBoxItem>
    <ComboBoxItem Content="Blue"></ComboBoxItem>
</ComboBox>
```

Elementy listy rozwijanej można zdefiniować na podstawie danych (umieszczonych np. w kolekcji) — taki sposób wykorzystamy w jednym z dalszych rozdziałów.

## Obsługa zdarzenia kliknięcia przycisku *Rysuj*

W widoku *Design* kliknij dwa razy lewym klawiszem myszy przycisk *Rysuj* (lub odśzukaj zdarzenie `Click` na liście zdarzeń w oknie *Properties*) i uzupełnij zawartość metody obsługującej zdarzenie kliknięcia tego przycisku:

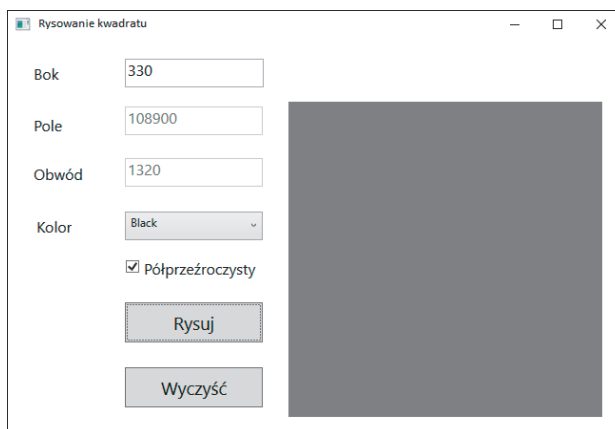
```
private void btnRysuj_Click(object sender, RoutedEventArgs e)
{
    double bok; // Maksymalny bok 380 (większy się nie zmieści w zadanym oknie)
    if (double.TryParse(txtBok.Text, out bok) && bok <= 380)
    {
        rectangle1.Height = bok;
        rectangle1.Width = bok;
        SolidColorBrush color = (SolidColorBrush)new BrushConverter().
            ↪ConvertFromString(cmbKolory.Text); // Konwersja koloru z typu string
        rectangle1.Stroke = color; // Przypisanie wybranego koloru dla konturu
        rectangle1.Fill = color; // Przypisanie wybranego koloru dla wypełnienia
        // IsChecked.Value jest typu bool, do sprawdzenia użyjemy operatora
        // warunkowego (?)
        // (jeśli ma wartość true, to ustawiamy Opacity na 50%,
        // w przeciwnym razie 100%)
        rectangle1.Opacity = (cbPrzezroczysty.IsChecked.Value) ? 0.5 : 1;
    }
    else
```

```
{  
    lblKomunikat.Content = "Brak danych lub zbyt duży bok";  
}  
}
```

Korzystając z powyższego kodu, zwrócić uwagę na nazwy kontroltek (np. `cmbKolory`, `cbPrzezroczysty`, `rectangle1`), bo muszą być zgodne z tymi, jakie zostały zdefiniowane w kodzie XAML Twojej aplikacji. Po wykonaniu wskazanych zmian można przetestować program (rysunek 3.4).

### Rysunek 3.4.

*Okno aplikacji  
Rysowanie  
kwadratu*



W wykonanych w tym rozdziale przykładach skupialiśmy się na prostych kontrolkach. W ostatnim przykładzie wystąpiło ich na tyle dużo, że należałoby już pomyśleć o organizacji układu kontroltek w oknie aplikacji. Do tego celu służą specjalne elementy grupujące zwane panelami. Panele zostaną przedstawione w następnym rozdziale. Natomiast rozdział 5., w którym omówię wiązanie danych, ukaże Czytelnikowi w nowym świetle wykonaną tu pracę, prezentując możliwości jej usprawnienia. Poznane w kolejnych rozdziałach elementy, takie jak konwertery czy mechanizm walidacji danych, wprowadzą dalsze udoskonalenia. W pewnym uproszczeniu i zarazem w przenośni można by powiedzieć, że większa część tego podręcznika jest o tym, jak poprawić przykład z rozdziału 3.

## 3.3 Zadania

W poniższych zadaniach należy wykorzystać kontrolki, które nie były używane w poprzednich programach — `RadioButton` i `Image`.

### Zadanie 3.1

Dodaj do projektu dwie kontrolki `RadioButton`: jedną opisaną jako „Ukryj”, a drugą jako „Pokaż”. Kontrolki te mają umożliwić użytkownikowi ukrywanie lub pokazanie narysowanego wcześniej kwadratu.

## Zadanie 3.2

Dodaj do projektu mały rysunek (np. logo).

## Zadanie 3.3

W wykonywanym projekcie zmień kolor tła głównego okna na jasnoszary.

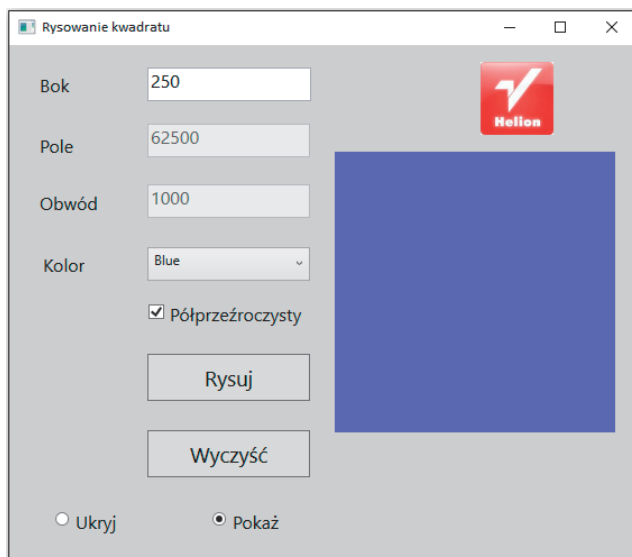
# 3.4 Wskazówki do zadań

Podobnie jak w poprzednim rozdziale umieszczone tu wskazówki są dość szczegółowe i powinny być wystarczające dla osób początkujących.

## Wskazówki do zadania 3.1

Kontrolki `RadioButton` wymagają zazwyczaj grupowania, tak aby można było wybrać tylko jedną opcję z danej grupy. Można zgrupować je za pomocą panelu, ale na razie proponuję inny sposób, mianowicie wykorzystanie w tym celu właściwości `GroupName`. Po naniesieniu z okna *Toolbox* obu kontrolerek dodaj w kodzie XAML dla obu znaczników atrybut `GroupName` (np. z zawartością `GroupName="Ukrywanie"`). Następnie należy w *code-behind* napisać kod dla metod obsługujących zdarzenie `Checked` (dla obu kontrolerek). Możesz w tym celu odszukać zdarzenie na liście zdarzeń lub kliknąć dwa razy lewym klawiszem myszy kontrolkę w oknie w widoku *Design*. O ukrywaniu i pokazywaniu elementów decyduje właściwość `Visibility`, o której była mowa we wskazówkach do zadania 2.3. Rysunek 3.5 przedstawia okno programu z kontrolkami `RadioButton`.

**Rysunek 3.5.**  
*Okno aplikacji  
Rysowanie  
kwadratu po  
modyfikacjach*



## Wskazówki do zadania 3.2

Rysunek możemy dodać za pomocą elementu `Image` (używając okna *Toolbox*). Wcześniej należy wgrać do projektu plik z grafiką. Możesz w tym celu utworzyć nowy folder w projekcie (w oknie *Solution Explorer*), klikając prawym klawiszem myszy na nazwie projektu, a potem wybierając z kontekstowego menu opcję *Add/New Folder*. Nazwij folder (np. `Rysunki`). Następnie, klikając na nazwie nowego folderu prawym klawiszem myszy, wybierz opcję *Add/Existing Item* i wskaż plik z obrazkiem. Po dodaniu pliku z obrazkiem nanieś element `Image` do okna aplikacji, a następnie w oknie *Properties* dla właściwości `Source` (w grupie `Common`) wskaż plik z obrazkiem. Rysunek 3.5 przedstawia okno programu z umieszczonym rysunkiem.

## Wskazówki do zadania 3.3

Kolor tła okna można zmienić za pomocą okna *Properties* lub bezpośrednio w oknie *XAML*, dodając (w tej linii, w której jest tytuł aplikacji) atrybut: `Background="LightGray"`. Rysunek 3.5 przedstawia okno programu z umieszczonym rysunkiem i zmienionym tłem.

## Rozdział 4.

# Panele

Panele pozwalają grupować inne elementy i definiować układ graficzny elementów. W tym rozdziale omawiam dostępne w WPF standardowe panele, których można użyć bezpośrednio w interfejsie użytkownika: `Canvas`, `StackPanel`, `WrapPanel`, `DockPanel` oraz `Grid`.

### 4.1 Canvas

`Canvas` jest prostym panelem, który nie zapewnia zaawansowanych możliwości pozycjonowania elementów. Element może być pozycjonowany względem kierunków:

- ◆ `Top` — należy wskazać odległość elementu od górnej krawędzi.
- ◆ `Bottom` — należy wskazać odległość elementu od dolnej krawędzi.
- ◆ `Left` — należy wskazać odległość elementu od lewej krawędzi.
- ◆ `Right` — należy wskazać odległość elementu od prawej krawędzi.

Podane właściwości wzajemnie wykluczają się parami, to znaczy jeśli zastosujemy odniesienie względem `Top` w jednym elemencie, to zostanie zignorowane odniesienie `Bottom`. Analogicznie w przypadku pary `Left` i `Right`. Jeśli nie wskażemy żadnego odniesienia względem podanych kierunków, to dany element zostanie umieszczony w lewym górnym rogu. Jeśli wskażemy tylko odniesienie względem prawej (lub lewej) krawędzi, to element będzie umieszczony u góry. Jeśli wskażemy tylko odniesienie względem dolnej (lub górnej) krawędzi, to element będzie umieszczony z lewej strony.

Przetestujemy podane wyżej możliwości. Utwórz nowy projekt. Zmień tytuł okna i jego rozmiar na podane: `Title="Panel Canvas" Height="200" Width="300"`. Następnie usuń znaczniki `<Grid>` i `</Grid>` i wpisz w tym miejscu kod:

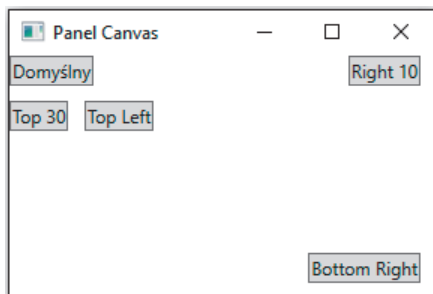
```
<Canvas>
  <Button Content="Domyślny"/>
  <Button Content="Top 30" Canvas.Top="30"/>
  <Button Content="Right 10" Canvas.Right="10"/>
</Canvas>
```

```
<Button Content="Top Left" Canvas.Top="30" Canvas.Left="50"/>
<Button Content="Bottom Right" Canvas.Bottom="10" Canvas.Right="10"/>
</Canvas>
```

Tak zdefiniowany panel ułoży przyciski zgodnie podanymi wyżej założeniami (rysunek 4.1).

#### Rysunek 4.1.

*Okno aplikacji  
testującej  
panel Canvas*



Zwróćmy jeszcze uwagę na zapis:

```
<Button Content="Top 30" Canvas.Top="30"/>
```

Jak wynika z dokumentacji MSDN<sup>1</sup>, przycisk `Button` udostępnia bardzo dużo właściwości, ale żadna z nich nie nazywa się `Top` ani `Canvas.Top`. Użyty tu atrybut `Canvas.Top` to tak zwana **właściwość dołączona**, o której była wzmianka w podrozdziale 1.3. Właściwość dołączona została tu zastosowana w celu umiejscowienia elementu podrzędnego (przycisku) w panelu. Nazwę właściwości dołączonej (tu `Top`) należy poprzedzić nazwą klasy, która jest dostawcą danej właściwości (tu `Canvas`). Na użytek zewnętrzny taka właściwość jest definiowana za pomocą metod statycznych (przykładowo dla właściwości dołączonej `Canvas.Top` w klasie `Canvas` zdefiniowane są metody statyczne `GetTop` i `SetTop`).

Panel `Canvas` mimo swej prostoty może być użyteczny w nieskomplikowanych układach. Elementy w tym układzie mogą na siebie nachodzić — domyślnie element zdefiniowany wcześniej zostaje przykryty przez element opisany w dalszej kolejności. Można wymusić inną kolejność poprzez właściwość `ZIndex`. Wykonaj testy dla takich przypadków na omówionym przykładzie.

## 4.2 StackPanel

`StackPanel` pozwala ułożyć elementy obok siebie w poziomie lub pionie (domyślnie). Panel ten nie ma właściwości pozwalających na precyzyjne pozycjonowanie elementów w układzie. Pamiętać należy jednak o właściwościach elementów takich jak `Margin`, dzięki którym można wpływać na ułożenie elementów w panelu.

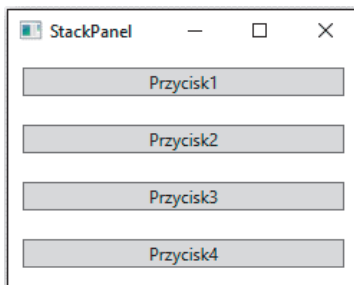
<sup>1</sup> Opis klasy `Button` („Button Class”): [https://msdn.microsoft.com/en-us/library/system.windows.controls.button\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.controls.button(v=vs.110).aspx).

Przetestujemy `StackPanel` najpierw na przyciskach. Utwórz nowy projekt. Zmień tytuł okna i jego rozmiar na podane: `Title="StackPanel" Height="200" Width="200"`. Następnie usuń znaczniki `<Grid>` i `</Grid>` i wpisz w tym miejscu kod:

```
<StackPanel>
  <Button Content="Przycisk1" Margin="10"/>
  <Button Content="Przycisk2" Margin="10"/>
  <Button Content="Przycisk3" Margin="10"/>
  <Button Content="Przycisk4" Margin="10"/>
</StackPanel>
```

Rysunek 4.2 prezentuje wygląd okna dla utworzonego panelu.

**Rysunek 4.2.**  
*Okno aplikacji  
testującej panel  
StackPanel*

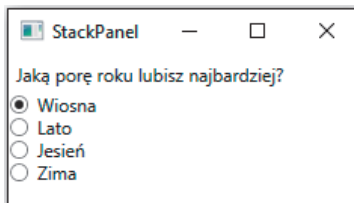


Wykonamy jeszcze jeden przykład z tym panelem, pokazujący jego rolę nie tylko w prezentacji kontrolki, ale i w grupowaniu elementów takich jak `RadioButton`. Utwórz nowy projekt. Zmień tytuł okna i jego rozmiar na następujące: `Title="StackPanel" Height="150" Width="260"`. Usuń znaczniki `<Grid>` i `</Grid>` i wpisz w tym miejscu kod:

```
<StackPanel>
  <Label x:Name="lblPytanie" Content="Jaką porę roku lubisz najbardziej?"/>
  <RadioButton x:Name="radioBtn1" Content="Wiosna"/>
  <RadioButton x:Name="radioBtn2" Content="Lato"/>
  <RadioButton x:Name="radioBtn3" Content="Jesień"/>
  <RadioButton x:Name="radioBtn4" Content="Zima"/>
</StackPanel>
```

Rysunek 4.3 przedstawia wygląd okna z zdefiniowanym panelem. Po uruchomieniu programu widzimy, że wszystkie kontrolki `RadioButton` stanowią jedną grupę, to znaczy można wybrać tylko jedną z opcji. Jest tak mimo braku identyfikatora grupy, o którym była mowa we wskazówkach do zadania 3.1. Panel daje alternatywną możliwość grupowania przycisków `RadioButton`.

**Rysunek 4.3.**  
*Okno aplikacji  
z kontrolkami  
RadioButton  
wewnątrz  
StackPanel*



`StackPanel` może prezentować elementy także poziomo. W tym celu należy dodać właściwość `Orientation` do definicji panelu: `<StackPanel Orientation="Horizontal">`.

## 4.3 WrapPanel

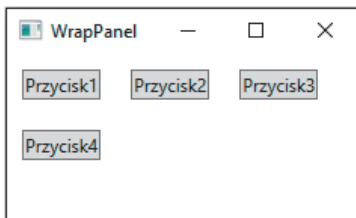
Bardzo podobny do omawianego wcześniej panelu `StackPanel` jest `WrapPanel`, który pozwala tworzyć dodatkowe wiersze (lub kolumny w orientacji pionowej) dla elementów niemieszczących się w zadanym obszarze. Dla panelu `WrapPanel` domyślną orientacją jest pozioma (można to zmienić za pomocą właściwości `Orientation`).

Przetestujemy prosty przykład dla tego panelu. Ustaw tytuł i rozmiary okna nowej aplikacji: `Title="WrapPanel" Height="150" Width="250"`, a następnie w miejsce znaczników `<Grid>` i `</Grid>` wpisz kod:

```
<WrapPanel>
  <Button Content="Przycisk1" Margin="10"/>
  <Button Content="Przycisk2" Margin="10"/>
  <Button Content="Przycisk3" Margin="10"/>
  <Button Content="Przycisk4" Margin="10"/>
</WrapPanel>
```

W tym przykładzie są cztery przyciski, podobnie jak w pierwszym przykładzie dla `StackPanel`, ale tym razem okno aplikacji ma mniejsze wymiary, stąd mamy możliwość zobaczyć efekt zawijania wiersza z przyciskami (rysunek 4.4).

**Rysunek 4.4.**  
Okno aplikacji  
testującej panel  
`WrapPanel`



Dla panelu `WrapPanel` można ustawić dodatkowe właściwości wpływające na położenie elementów: `ItemHeight` (wysokość wszystkich elementów w panelu) i `ItemWidth` (szerokość elementów). Przetestuj użycie tych właściwości, zmieniając pierwszą linię definicji panelu na przykład na taką: `<WrapPanel ItemHeight="50" ItemWidth="90">`.

## 4.4 DockPanel

`DockPanel` pozwala umieszczać elementy przy jednej z krawędzi okna: `Top` (górna), `Bottom` (dolna), `Left` (lewa — domyślna) oraz `Right` (prawa). Umieszczanie elementów umożliwia właściwość dołączona `Dock`. Panel `DockPanel` wykorzystuje całą dostępną powierzchnię dla definiowanych elementów; można takie zachowanie panelu zmienić poprzez właściwość `LastChildFill`.

Przetestujemy omawiany panel. Ustaw tytuł i rozmiary okna nowej aplikacji: `Title="DockPanel" Height="150" Width="250"`, a następnie w miejsce znaczników `<Grid>` i `</Grid>` wpisz kod:

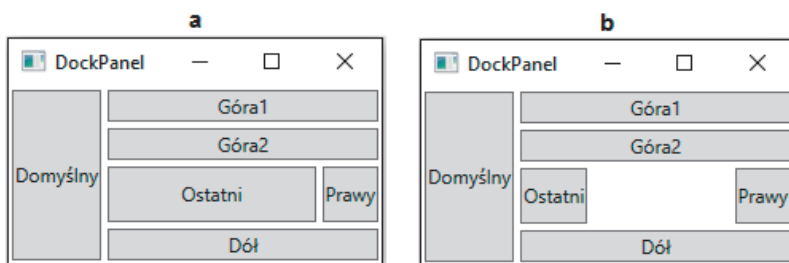


```

<DockPanel LastChildFill="True">
  <Button Content="Domyślny" Margin="2"/>
  <Button Content="Góra1" Margin="2" DockPanel.Dock="Top"/>
  <Button Content="Góra2" Margin="2" DockPanel.Dock="Top"/>
  <Button Content="Dół" Margin="2" DockPanel.Dock="Bottom"/>
  <Button Content="Prawy" Margin="2" DockPanel.Dock="Right"/>
  <Button Content="Ostatni" Margin="2"/>
</DockPanel>

```

Prezentowany kod spowoduje ułożenie elementów poprzez wypełnienie całej przestrzeni panelu (rysunek 4.5a). Zmiana właściwości `LastChildFill` na `False` sprawi, że po umieszczeniu ostatniego elementu zostanie wolne miejsce (4.5b).



**Rysunek 4.5.** Okno aplikacji testującej panel `DockPanel` z ustawieniem właściwości `LastChildFill="True"` (a) i z ustawieniem właściwości `LastChildFill="False"` (b)

Ze względu na specyfikę tego panelu jest on wykorzystywany zazwyczaj do umieszczania w nim innych paneli.

## 4.5 Grid

Panelowi `Grid` poświęcimy nieco więcej miejsca. Panel ten ma więcej możliwości w zakresie pozycjonowania elementów i z tego powodu jest bardziej uniwersalny — można za jego pomocą zaplanować różne układy. Panel `Grid` (siatka) pozwala zorganizować widok w formie tabelarycznej. W układzie tym każdy element umieszczany jest w siatce na podstawie numeru wiersza (ang. *row*) i kolumny (ang. *column*). Panel ten definiujemy przy użyciu elementu `<Grid>`. W jego wnętrzu należy zdefiniować wiersze i kolumny wraz z ich wymiarami (wysokość dla wierszy, szerokość dla kolumn). Wartości dla wysokości wierszy i szerokości kolumn można definiować na trzy sposoby:

1. Przy użyciu **wartości bezwzględnych** (wartości liczbowych). Przy takim ustawieniu wielkość zdefiniowanego komponentu w panelu nie zmieni się mimo zmiany okna.
2. Stosując opcję `Auto`, która umożliwi automatyczne dopasowanie rozmiaru kolumny/wiersza na podstawie najszerszego/najwyższego elementu, jaki ma być w niej/nim umieszczony. Przy takim ustawieniu wielkość elementu także się nie zmieni podczas zmiany rozmiaru okna.

3. Stosując rozmiar proporcjonalny. Za pomocą znaku **gwiazdki** (\*) możemy zdefiniować rozmiar wolnej przestrzeni, jaka pozostała po przydzieleniu miejsca dla innych elementów. W przypadku gdy więcej wierszy lub kolumn jest ustawionych w ten sposób, to pozostała do rozdzielenia przestrzeń dzielona jest proporcjonalnie (chyba że zdecydujemy inaczej, wpisując liczbę przed gwiazdką, np. 2\*).

Można łączyć wymienione rodzaje ustawień w jednym panelu. Rozmiary kolumn i wierszy zdefiniowanych w sposób proporcjonalny (z użyciem gwiazdki) będą się zmieniać podczas zmiany rozmiaru okna (co widać na rysunku 4.6).

Po zdefiniowaniu wierszy i kolumn należy przypisać numer wiersza i kolumny wszystkim elementom, które mają być umieszczone w panelu **Grid**.

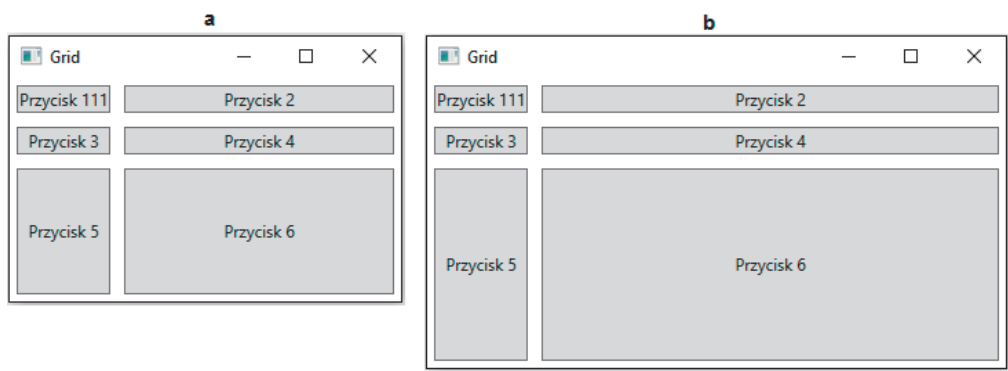
Przetestujemy omawiany panel. Ustaw tytuł i rozmiary okna nowej aplikacji: **Title="Grid" Height="200" Width="300"**, a następnie w miejsce znaczników **<Grid>** i **</Grid>** wpisz kod:

```
<Grid>
  <!-- Definicja wierszy i kolumn-->
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="*/>
  </Grid.ColumnDefinitions>

  <!-- Definicja elementów panelu-->
  <Button Content="Przycisk 111" Grid.Row="0" Grid.Column="0" Margin="5"/>
  <Button Content="Przycisk 2" Grid.Row="0" Grid.Column="1" Margin="5"/>
  <Button Content="Przycisk 3" Grid.Row="1" Grid.Column="0" Margin="5"/>
  <Button Content="Przycisk 4" Grid.Row="1" Grid.Column="1" Margin="5"/>
  <Button Content="Przycisk 5" Grid.Row="2" Grid.Column="0" Margin="5"/>
  <Button Content="Przycisk 6" Grid.Row="2" Grid.Column="1" Margin="5"/>
</Grid>
```

Popatrzmy na wykonanie tak zdefiniowanego panelu w dwóch wariantach bez zmiany rozmiaru okna (rysunek 4.6a) i po powiększeniu przez użytkownika rozmiaru okna aplikacji (rysunek 4.6b).

Spójrzmy ponownie na kod XAML, na sekcję oznaczoną komentarzem **<!-- Definicja wierszy i kolumn-->**. W miejscu tym definiowane są wiersze i kolumny panelu. Mamy tu trzy wiersze: dwa pierwsze mają wysokość **Auto**, natomiast ostatni jest zdefiniowany z użyciem gwiazdki (\*). Po zmianie wysokości okna rozmiar elementów w dwóch pierwszych wierszach pozostaje bez zmian, ale zmienia się wysokość ostatniego wiersza (rysunek 4.6). Dalej jest definicja dwóch kolumn: pierwsza jest **Auto**, a druga proporcjonalna (z gwiazdką). Ponieważ pierwsza kolumna jest zdefiniowana jako **Auto**, jej szerokość została dopasowana do najszerszego elementu (jest nim *Przycisk 111*).



**Rysunek 4.6.** Okno aplikacji testującej panel `Grid` przed zmianą rozmiaru okna (a) i po zmianie rozmiaru okna (b)

Natomiast proporcjonalne określenie rozmiaru dla drugiej kolumny skutkuje wypełnieniem pozostałej wolnej przestrzeni — obszar ten zmienia się wraz ze zmianą całego okna (rysunek 4.6).

Teraz przeanalizujemy kod w sekcji oznaczonej komentarzem `<!-- Definicja elementów panelu-->`. Poprzednia sekcja określiła nam wygląd samej siatki (tabelki), a teraz do poszczególnych komórek tej siatki należy przypisać komponenty. W tym celu używa się właściwości dołączonych `Grid.Row` (numer wiersza) i `Grid.Column` (numer kolumny). Obie wielkości numeruje się od zera. Na rysunku 4.7 została przedstawiona siatka dla naszego panelu. Rysunek ten jednoznacznie wskazuje „współrzędne” dla poszczególnych elementów panelu. Wielkości te zostały wpisane w omawianym kodzie XAML. Przykładowo `Przycisk 111` ma współrzędne `Grid.Row="0"` `Grid.Column="0"`, co oznacza, że zostanie umieszczony w komórce siatki w zerowym wierszu i zerowej kolumnie.

**Rysunek 4.7.**  
Siatka dla  
przykładowego  
panelu `Grid`

	0	1
0	Row=0; Col=0	Row=0; Col=1
1	Row=1; Col=0	Row=1; Col=1
2	Row=2; Col=0	Row=2; Col=1

Komórki panelu `Grid` można scalać w poziomie i pionie poprzez ustawienie właściwości dołączonych `Grid.ColumnSpan` i `Grid.RowSpan`. Wykorzystamy możliwość scalania komórek w kolejnym przykładzie. Ustaw tytuł i rozmiary okna nowej aplikacji: `Title="Grid" Height="200" Width="300"`, a następnie w miejsce pustych znaczników `<Grid>` i `</Grid>` wpisz kod:

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

```

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

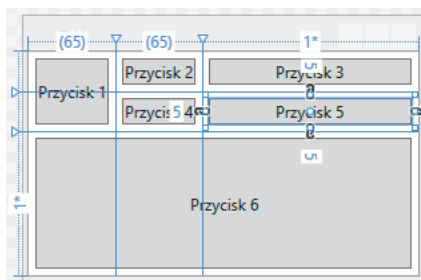
<!-- Definicja elementów panelu-->
<Button Content="Przycisk 1" Grid.Row="0" Grid.Column="0" Grid.RowSpan="2"
        Margin="5"/>
<Button Content="Przycisk 2" Grid.Row="0" Grid.Column="1" Margin="5"/>
<Button Content="Przycisk 3" Grid.Row="0" Grid.Column="2" Margin="5"/>
<Button Content="Przycisk 4" Grid.Row="1" Grid.Column="1" Margin="5"/>
<Button Content="Przycisk 5" Grid.Row="1" Grid.Column="2" Margin="5"/>
<Button Content="Przycisk 6" Grid.Row="2" Grid.Column="0"
        Grid.ColumnSpan="3" Margin="5"/>
</Grid>

```

Spójrzmy na rysunek 4.8, na którym jest zdefiniowany `Grid` w widoku *Design*. Zwróć uwagę na linie siatki i elementy, które są umieszczone w obszarze kilku komórek siatki.

#### Rysunek 4.8.

Panel `Grid`  
w widoku *Design*



*Przycisk 1* umieszczony jest w obszarze dwóch komórek: pierwsza z nich (o współrzędnych `Grid.Row="0"` `Grid.Column="0"`) została wskazana w opisie elementu, druga (w poniższym wierszu) wynika z zapisu `Grid.RowSpan="2"`. Właściwość `Grid.RowSpan` pozwala wpisać liczbę wierszy, na jakie należy rozciągnąć element w pionie. *Przycisk 1* ma zająć dwa wiersze. Analogicznie jest z przyciskiem *Przycisk 6*: zaczyna się w komórce o współrzędnych `Grid.Row="2"` `Grid.Column="0"` i został rozciągnięty na trzy kolumny poprzez użycie właściwości dołączonej `Grid.ColumnSpan="3"`, zatem zajmuje wszystkie trzy komórki w ostatnim wierszu.

Panel `Grid` oferuje więcej możliwości, ale te, które omówiłam, wystarczą w pierwszym etapie nauki. W bieżącym rozdziale nie wykonywaliśmy kompletnych aplikacji, lecz jedynie same przykłady paneli. Nie będzie tu zadań do samodzielnego rozwiązania, niemniej wskazane jest, aby na podstawie prezentowanych przykładów poeksperymentować z różnymi wariantami układów.

Oprócz przedstawionych w tym podrozdziale paneli są dostępne jeszcze inne, które kontrolują układ elementów wewnątrz innych kontrolki (np. `TabPanel` wewnątrz kontrolki `TabControl`). Ponadto istnieje możliwość definiowania własnych paneli<sup>2</sup>.

<sup>2</sup> Tworzenie własnych paneli opisano m.in. w: Cisek J., *Tworzenie nowoczesnych aplikacji graficznych w WPF*, Helion, Gliwice 2012 (str. 118 – 131).

## Rozdział 5.

# Wiązanie danych — aplikacja Produkt

Wiązanie danych w WPF jest mechanizmem, który ustanawia połączenie między dwiema właściwościami różnych obiektów. Jednym z ważniejszych zastosowań tego mechanizmu jest łączenie danych i wizualnych elementów interfejsu. Definiując wiązanie, należy określić źródło danych i cel wiązania. Źródłem danych może być obiekt, baza danych, zasób, XML — właściwie „wszystko”. Natomiast celem powinna być właściwość zależna elementu WPF. Aby móc w pełni korzystać z automatyzmu, jaki dostarcza ten mechanizm, należy jeszcze uwzględnić kilka innych rozwiązań WPF, takich jak konwertery, walidacja danych, powiadomienia o zmianach czy wiązanie poleceń. Zagadnienia te zostaną przedstawione w dalszej części książki. Niemniej już teraz będzie można docenić wygodę wiązania danych. W niniejszym rozdziale w pierwszej kolejności przetestujemy wiązanie danych na prostym przykładzie z suwakiem, a następnie przystąpimy do wykonania aplikacji, w której zostanie ustanowione połączenie między polami tekstowymi a właściwościami obiektu klasy `Produkt`. Wykorzystamy umiejętności nabyte w poprzednim rozdziale i umieścimy kontrolki w oknie przy użyciu panelu.

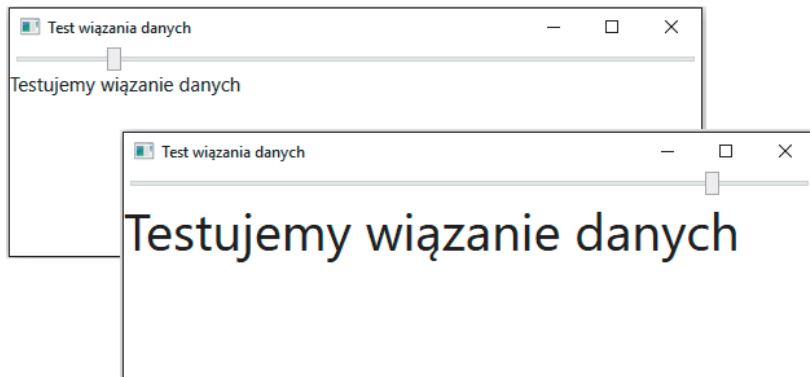
## 5.1 Testowanie wiązania danych

Wiązanie danych można wykonać w *code-behind* lub kodzie XAML. W tym przykładzie wykorzystamy ten drugi sposób. Otwórz nowy projekt WPF. W kodzie XAML ustaw atrybut `Title="Test wiązania danych"`. Zmień także rozmiar okna aplikacji: `Height="200" Width="550"`. W oknie *XAML* podmień kod ze znacznikami `<Grid>` na następujący:

```
<StackPanel>
  <Slider x:Name="rozmiarTekstu" Minimum="10" Value="15" Maximum="45"/>
  <TextBlock FontSize="{Binding Path=Value, ElementName=rozmiarTekstu}">
    Testujemy wiązanie danych
  </TextBlock>
</StackPanel>
```

Programik ten można już uruchomić. Dzięki wiązaniu danych nie potrzebujemy obsługiwać zdarzenia dla zmiany wartości suwaka. Po uruchomieniu programu możemy przesunąć suwak i w ten sposób zwiększać lub zmniejszać rozmiar czcionki tekstu, jaki się wyświetla poniżej suwaka (rysunek 5.1).

**Rysunek 5.1.**  
*Testowanie  
wiązania danych  
z użyciem suwaka*



Suwak stanowi źródło dla omawianego wiązania danych. Natomiast blok tekstu jest celem tego wiązania. Zatem po obu stronach wiązania w tym przypadku są jakieś elementy WPF.

Omówmy dokładnie przedstawiony kod XAML. W znaczniku `Slider` zdefiniowany jest suwak. Kontrolka ta ma więcej właściwości, tu zostały wykorzystane następujące: wartość minimalna (`Minimum`), wartość aktualna (`Value`) oraz wartość maksymalna (`Maximum`).

Znacznik `TextBlock` definiuje blok tekstu. Bez wiązania danych definicja tego elementu mogłaby wyglądać przykładowo tak:

```
<TextBlock FontSize="15">  
    Testujemy wiązanie danych  
</TextBlock>
```

Albo w zapisie równoważnym tak:

```
<TextBlock FontSize="15" Text="Testujemy wiązanie danych"/>
```

Skupmy się zatem na tym, co różni ten prosty zapis, podobny do wielu innych prezentowanych wcześniej w tej książce, od zapisu uwzględniającego wiązanie. Spójrzmy na kod:

```
<TextBlock FontSize="{Binding Path=Value, ElementName=rozmiarTekstu}">
```

Zamiast konkretnej wartości dla atrybutu `FontSize` przypisane jest wyrażenie w klamrach. Wiązanie dwóch elementów w kodzie XAML realizowane jest za pomocą specjalnego rozszerzenia znaczników `Binding`. Definiując wiązanie, po lewej stronie znaku przypisania określamy cel wiązania, tu jest to właściwość `FontSize` (rozmiar tekstu). Natomiast po prawej stronie znaku przypisania umieszcza się źródło wiązania. W klamrach oprócz słowa `Binding` widzimy dwie właściwości: `ElementName`, która wskazuje źródło wiązania, oraz `Path`, która wskazuje właściwość obiektu źródłowego.

Warto nadmienić, że można użyć alternatywnego rozszerzenia znaczników poprzez przekazanie obiektu `Path` do konstruktora. Dzięki temu znacznik otwierający `TextBlock` możemy zapisać także w następujący sposób:

```
<TextBlock FontSize="{Binding Value, ElementName=rozmiarTekstu}">
```

Jak widać, nie ma tu jawnego przypisania wartości do właściwości `Path`.

Można także inaczej wykonać samo wiązanie w kodzie XAML — używając `Binding` jako elementu XAML, jak w przykładzie:

```
<TextBlock>
  <TextBlock.FontSize>
    <Binding Path="Value" ElementName="rozmiarTekstu"/>
  </TextBlock.FontSize>
  Testujemy wiązanie danych
</TextBlock>
```

Po „rozgrzewce” z suwakiem, która pozwoliła Czytelnikowi poznać istotę mechanizmu wiązania danych, przejdziemy w dalszej części rozdziału do pracy nad tworzeniem aplikacji *Produkt*. Zaczniemy od kodu XAML.

## 5.2 Kod XAML

Nierzadko mamy do czynienia z przypadkiem, w którym kilka różnych kontrolki jest wiązanych z tym samym obiektem źródłowym (ale z różnymi jego właściwościami). W takich sytuacjach można skorzystać z **kontekstu danych**, który umożliwia podanie źródła danych w jednym miejscu, w elemencie nadrzędnym. Definicja kontekstu wykonywana za pomocą właściwości `DataContext` może być umieszczona w kodzie XAML lub *code-behind*. W tym i w dwóch kolejnych rozdziałach użyjemy tego drugiego sposobu.

Otwórz nowy projekt WPF. W kodzie XAML ustaw atrybut `Title="Produkt"`. Zmień także rozmiar okna aplikacji: `Height="220" Width="350"`. W oknie *XAML* podmień kod ze znacznikami `<Grid>` i `</Grid>` na następujący:

```
<Grid x:Name="gridProdukt">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="*/>
  </Grid.ColumnDefinitions>
```

```

<Label Content="Symbol:" Grid.Row="0" Grid.Column="0" Margin="5"/>
<TextBox Grid.Row="0" Grid.Column="1" Margin="5" Text="{Binding Symbol}"/>
<Label Content="Nazwa:" Grid.Row="1" Grid.Column="0" Margin="5"/>
<TextBox Grid.Row="1" Grid.Column="1" Margin="5" Text="{Binding Nazwa}"/>
<Label Content="Liczba sztuk:" Grid.Row="2" Grid.Column="0" Margin="5"/>
<TextBox Grid.Row="2" Grid.Column="1" Margin="5"
    Text="{Binding LiczbaSztuk}"/>
<Label Content="Magazyn:" Grid.Row="3" Grid.Column="0" Margin="5"/>
<TextBox Grid.Row="3" Grid.Column="1" Margin="5" Text="{Binding Magazyn}"/>
<Button x:Name="btnPotwierdz" Grid.Row="5" Grid.Column="0"
    Grid.ColumnSpan="2" Margin="4" MinWidth="120"
    HorizontalAlignment="Center" Content="Potwierdź"
    Click="btnPotwierdz_Click"/>
</Grid>

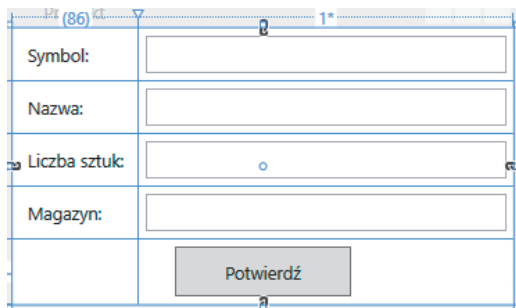
```

Panel składa się z 5 wierszy i 2 kolumn, przy czym w ostatnim wierszu, w którym definiowany jest przycisk *Potwierdź*, dwie kolumny zostały połączone (poprzez ustawienie właściwości `Grid.ColumnSpan="2"`). Dla pól tekstowych atrybut `Text` ma przypisaną wartość, umożliwiającą wiązanie z poszczególnymi właściwościami klasy (klasę *Produkt* wykonamy w kolejnym podrozdziale). I tak przykładowo pole tekstowe, w którym będzie wpisywany symbol produktu, ma dla tego atrybutu wartość: `Text="{Binding Symbol}"`, co oznacza, że zostanie powiązane z właściwością `Symbol`. Wykorzystano tu zapis pozwalający pominąć jawne przypisanie właściwości `Path`. Można by użyć także alternatywnego zapisu: `Text="{Binding Path=Symbol}"`. Należy zwrócić uwagę na to, że panel `Grid` ma tu swoją nazwę — `x:Name="gridProdukt"`. Ta nazwa zostanie wykorzystana w *code-behind* do przypisania kontekstu dla wiązania danych. Wewnątrz panelu tylko jeden element ma nadaną nazwę — przycisk *Potwierdź* (`btnPotwierdz`). Dzięki wiązaniu danych nie ma potrzeby używać nazw pozostałych elementów w *code-behind*.

Dla przedstawionego kodu XAML otrzymamy okno w widoku *Design*, jak na rysunku 5.2.

### Rysunek 5.2.

Panel `Grid`  
aplikacji *Produkt*





## 5.3 Definicja klasy Produkt i code-behind

W dalszym kroku stworzymy nową klasę w osobnym pliku<sup>1</sup>. Plik nazwijmy *Produkt.cs*. Należy tam umieścić kod klasy:

```
class Produkt
{
    public string Symbol { get; set; }
    public string Nazwa { get; set; }
    public int LiczbaSztuk { get; set; }
    public string Magazyn { get; set; }

    public Produkt(string sym, string naz, int lszt, string mag)
    {
        Symbol = sym;
        Nazwa = naz;
        LiczbaSztuk = lszt;
        Magazyn = mag;
    }

    public override string ToString()
    {
        return String.Format("{0} {1} {2} {3}", Symbol, Nazwa, LiczbaSztuk,
            Magazyn);
    }
}
```

Kod klasy jest prosty, zawiera cztery publiczne właściwości opisujące produkt, konstruktor oraz metodę *ToString*, zwracającą informacje o produkcie.

W pliku *MainWindow.xaml.cs* należy podmienić kod klasy *MainWindow* na następujący:

```
public partial class MainWindow : Window
{
    private Produkt p1 = null;

    public MainWindow()
    {
        InitializeComponent();
        PrzygotujWiazanie();
    }

    private void PrzygotujWiazanie()
    {
        p1 = new Produkt("DZ-10", "długopis żelowy", 132, "Katowice 1");
        gridProdukt.DataContext = p1;
    }
}
```

---

<sup>1</sup> Aby dodać nowy plik w projekcie, należy kliknąć nazwę projektu w oknie *Solution Explorer*, a następnie po naciśnięciu prawego klawisza myszy wybrać opcję *Add/Class*.

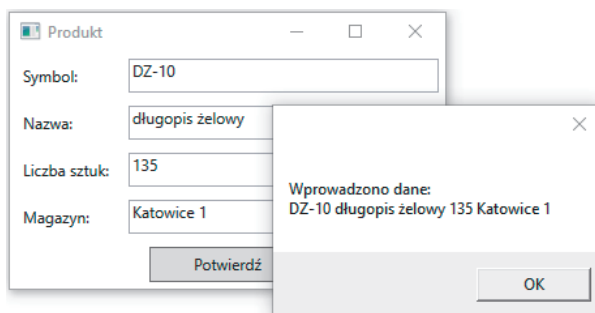
W metodzie `PrzygotujWiazanie` tworzony jest obiekt klasy `Produkt` i ustawiany jest kontekst dla wiązania danych. Wiązanie będzie realizowane z kontrolkami tego panelu, którym przypisano (w kodzie XAML) rozszerzenie znaczników `Binding`, np. `Text="{Binding Symbol}"`.

Została nam już tylko ostatnia rzecz do wykonania w tym programie, mianowicie definicja metody obsługującej zdarzenie kliknięcia przycisku *Potwierdź*. Kliknij w tym celu dwa razy lewym klawiszem myszy ten przycisk w widoku *Design* i wpisz kod metody:

```
private void btnPotwierdz_Click(object sender, RoutedEventArgs e)
{
    string tekst = String.Format("{0}{1}{2}", "Wprowadzono dane:",
        Environment.NewLine, p1.ToString());
    MessageBox.Show(tekst);
}
```

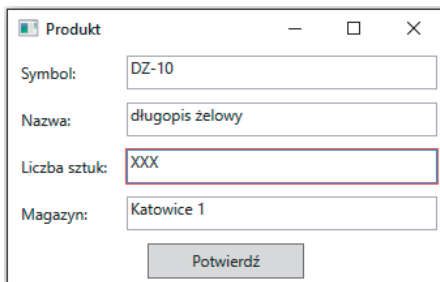
Ponieważ nigdzie nie zapisujemy danych, metodę tę wykorzystamy jedynie do tego, aby wyświetlić informacje o produkcie. Program można już uruchomić. Wstępnie pojawiają się dane dla produktu, można je dowolnie zmieniać, a następnie kliknąć przycisk *Potwierdź* (rysunek 5.3).

**Rysunek 5.3.**  
*Testowanie  
aplikacji Produkt*



W programie nie wykonaliśmy walidacji, czyli sprawdzania poprawności danych, jakie wprowadza użytkownik. Mimo to pewne możliwości program w tym zakresie posiada. Popatrzmy na rysunek 5.4. Po wpisaniu tekstu **XXX** w polu *Liczba sztuk* i kliknięciu dowolnego innego pola (lub przycisku) obramowanie tego pola zmienia kolor na czerwony.

**Rysunek 5.4.**  
*Walidacja pola  
tekstowego*



Nie jest możliwa konwersja tekstu `XXX` do danej typu `int`, czyli typu właściwości `LiczbaSztuk` (z klasy `Produkt`) połączonej z tą kontrolką. Wyświetlenie czerwonej ramki pola tekstowego zapewnia domyślny szablon<sup>2</sup>, nie zapewnia on jednak stosownego komunikatu. Ponadto nie mamy obecnie możliwości sprawdzania poprawności formatu danych. Przykładowo wiedząc, że symbol produktu składa się z sekwencji dużych liter, łącznika oraz kilku cyfr, moglibyśmy sprawdzić, czy użytkownik wpisał poprawny symbol. Istnieje kilka sposobów przeprowadzenia takiej walidacji danych, wybrane z nich zostaną opisane w rozdziale 12.

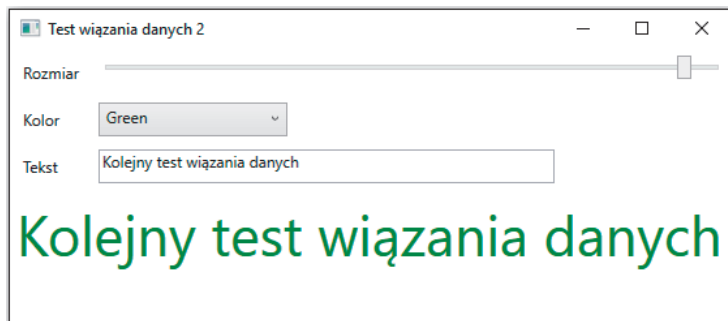
## 5.4 Zadania

W pierwszym zadaniu należy dowiązać kilka kontrolki do jednej. Drugi program będzie wymagał zmiany sposobu aktualizacji kontrolki. Natomiast ostatnie zadanie wymaga użycia właściwości służącej do formatowania danej użytej w wiązaniu.

### Zadanie 5.1

Program testujący wiązanie danych z rozdziału 5.1 rozwiń o dwie nowe kontrolki, które mają podlegać wiązaniu: listę rozwijaną z listą kilku kolorów oraz pole tekstowe do wpisywania tekstu. Program ma wyświetlać zadany tekst w wybranym kolorze dla określonego rozmiaru (rysunek 5.5).

**Rysunek 5.5.**  
*Przykładowe działanie programu*



### Zadanie 5.2

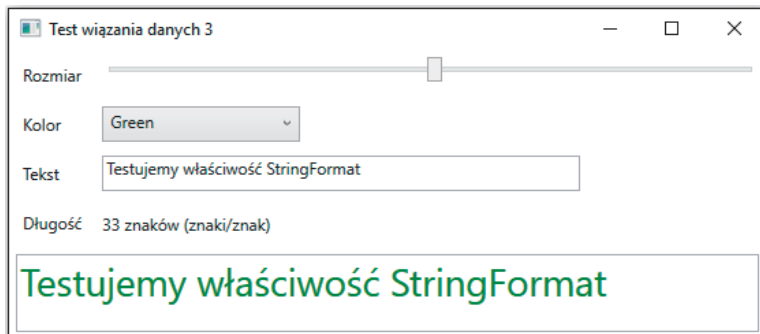
Wykonaj nową wersję programu z poprzedniego zadania, zmieniając element docelowy z `TextBlock` na `TextBox`. Po zmianie i uruchomieniu program będzie wyglądał bardzo podobnie, ale będzie widoczna zmiana w działaniu, a mianowicie będzie można zmienić element docelowy wiązania. Efekt tej zmiany jest jednak widoczny dopiero po zmianie fokusu (aktywacji innej kontrolki). Wykonaj taką wersję programu, aby zmiana w kierunku od celu do źródła wiązania była natychmiastowa (podobnie jak to ma miejsce w drugą stronę).

<sup>2</sup> Szablony zostaną przedstawione w rozdziale 11.

## Zadanie 5.3

Uzupełnij program z poprzedniego zadania o nowy element — `TextBlock`, w którym będzie się wyświetlać aktualna liczba znaków tekstu umieszczonego w polu *Tekst*. W polu tym zaraz za liczbą znaków ma się wyświetlić tekst „znaków (znaki/znak)”. Wykorzystaj w tym celu właściwość `StringFormat` klasy `Binding`. Przykładowy efekt działania programu przedstawia rysunek 5.6.

**Rysunek 5.6.**  
Przykładowe  
działanie  
programu



## 5.5 Wskazówki do zadań

Wskazówki zawierają dość szczegółowe wyjaśnienia i przykładowe propozycje rozwiązania.

### Wskazówki do zadania 5.1

W programie mają być trzy kontrolki źródłowe: suwak dla rozmiaru czcionki, lista rozwijana dla koloru czcionki i pole tekstowe do wpisywania tekstu, a także jedna kontrolka dla celu wiązania — blok tekstu. Należałoby jeszcze dodać trzy kontrolki z etykietą opisującą trzy źródłowe komponenty. W szczegółach zmiany względem bazowego programu wyglądałyby następująco: należy powiększyć nieco rozmiar okna, dodać nowe kontrolki, które zalecam ułożyć w panelu `Grid` mającym 4 wiersze i 2 kolumny, a ostatni element (`TextBlock`) powinien zajmować obie kolumny w ostatnim wierszu. Przykład użycia listy rozwijanej (`ComboBox`) został opisany w podrozdziale 3.2. Wpisz do listy angielskie nazwy kolorów (np. `Black`, `Red`, `Green`)<sup>3</sup>. Przykładowa realizacja wiązania danych w definicji znacznika `TextBlock` mogłaby wyglądać tak:

```
<TextBlock Grid.Row ="3" Grid.Column ="0" Grid.ColumnSpan="2" Margin="5"
    FontSize="{Binding Path=Value, ElementName=rozmiarTekstu}"
    Text="{Binding Path=Text, ElementName=txtTekst}"
    Foreground="{Binding Path=SelectedItem.Content, ElementName=cmbKolor}"/>
```

<sup>3</sup> W jednym z dalszych rozdziałów dowiesz się, jak wykonać wiązanie danych z użyciem konwerterów wartości (wówczas nazwy kolorów będą mogły być polskie).

## Wskazówki do zadania 5.2

Możemy wykorzystać kod XAML znacznika `TextBlock` podany we wskazówkach do poprzedniego zadania. W pierwszej kolejności należy podmienić znacznik `TextBlock` na `TextBox`, zostawiając pozostałe ustawienia. To już wystarczy do testowania działania w domyślnym trybie aktualizacji. Zmiana elementu docelowego jest aktualizowana w elemencie źródłowym dopiero po **utracie fokusu** w kontrolce docelowej (czyli sprawieniu, że kontrolka ta przestaje być aktywna). Po przeprowadzeniu testu można przejść do wykonania drugiej części zadania, mianowicie zmiany domyślnego sposobu aktualizacji elementu źródłowego.

W wykonanych dotąd w tym rozdziale programach aktualizacja danych następowała od obiektu źródłowego do obiektu docelowego, ale może przebiegać także w drugim kierunku. W klasie `Binding` jest właściwość `Mode`, która przyjmuje jedną z wartości typu wyliczeniowego `BindingMode`. Typ ten zawiera pięć elementów<sup>4</sup>, wśród których są `OneWay` i `TwoWay`. Tryb `OneWay` oznacza, że element docelowy jest aktualizowany przy każdej zmianie źródła, natomiast tryb `TwoWay` oznacza aktualizację obustronną (od źródła do celu i od celu do źródła). Większość kontroltek ma domyślnie przypisany tryb `OneWay`, ale takie kontrolki jak `TextBox`, umożliwiające edycję danych, mają ustawiony domyślnie tryb `TwoWay`.

W przypadku gdy aktualizacja następuje w kierunku od celu do źródła, można ją wykonać na trzy sposoby: natychmiast, po utracie fokusu lub po jawnym wywołaniu metody aktualizującej źródło. Decyduje o tym właściwość `UpdateSourceTrigger`, której można przypisać wartość typu wyliczeniowego `UpdateSourceTrigger`. Wśród elementów tego typu są wartości<sup>5</sup>: `PropertyChanged` (natychmiastowa aktualizacja celu na podstawie źródła) i `LostFocus` (źródło jest aktualizowane po utracie fokusu). Domyślnym ustawieniem sposobu aktualizacji celu dla kontrolki `TextBox` jest `LostFocus`. Na podstawie przytoczonych informacji widzimy, że zmiana w naszym programie będzie niewielka. Wystarczy dodać do wyrażenia `Binding` przypisanie odpowiedniej wartości dla właściwości `UpdateSourceTrigger`. Definicja wiązania dla właściwości `Text` będzie wyglądać następująco:

```
Text="{Binding Path=Text, ElementName=txtTekst,  
UpdateSourceTrigger= PropertyChanged}"
```

## Wskazówki do zadania 5.3

Dodaj w panelu `Grid` nowy wiersz, a następnie dotychczasową lokalizację dla ostatniej kontrolki (`TextBox`) przenieś do nowego ostatniego wiersza (`Grid.Row ="4"`). W wierszu powyższym dodaj w lewej kolumnie etykietę z opisem *Długość*, a w prawej kolumnie zdefiniuj element `TextBlock` według wzoru:

<sup>4</sup> Wartości typu wyliczeniowego `BindingMode` („BindingMode Enumeration”):

[https://msdn.microsoft.com/en-us/library/system.windows.data.bindingmode\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.data.bindingmode(v=vs.110).aspx).

<sup>5</sup> Wartości typu wyliczeniowego `UpdateSourceTrigger` („UpdateSourceTrigger Enumeration”):

[https://msdn.microsoft.com/en-us/library/system.windows.data.updatesourcetrigger\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.data.updatesourcetrigger(v=vs.110).aspx).

```
<TextBlock Grid.Row ="3" Grid.Column ="1" HorizontalAlignment="Left"
    Margin="5,11,0,5"
    Text="{Binding StringFormat={}{0} znaków (znaki/znak),
    Path=Text.Length, ElementName=txtTekst}"/>
```

Jeżeli wartość, do której określono wiązanie, ma być wyświetlona w innej postaci niż domyślna, można użyć właściwości `StringFormat` z klasy `Binding`. Jak wspomniałam w podrozdziale 1.4 na temat podstaw XAML, ujęcie wartości atrybutu w nawiasy klamrowe `{}` oznacza, że stanowi ona rozszerzenie znaczników. Jeśli na początku tekstu określającego wartość właściwości chcemy użyć nawiasu klamrowego jako zwykłego literału znakowego, nie traktowanego jako rozszerzenie znaczników, musimy przed nim (po znaku przypisania) użyć pustej pary nawiasów klamrowych. I tak jest właśnie w prezentowanym przykładzie: `StringFormat={}{0} znaków (znaki/znak)`. Pusta para nawiasów klamrowych po znaku przypisania byłaby zbędna, gdyby wartość dla właściwości `StringFormat` zaczynała się od innego znaku niż klamra. Na przykład przetłumacz taki wariant: `StringFormat=liczba znków: {0}`. Puste klamry byłyby zbędne także w przypadku, gdybyśmy użyli `Binding` jako elementu XAML. Wówczas kod znacznika dla bloku tekstu mógłby wyglądać tak:

```
<TextBlock Grid.Row ="3" Grid.Column ="1" HorizontalAlignment="Left"
    Margin="5,11,0,5">
    <TextBlock.Text>
        <Binding Path="Text.Length" ElementName="txtTekst">
            <Binding.StringFormat>
                {0} znaków (znaki/znak)
            </Binding.StringFormat>
        </Binding>
    </TextBlock.Text>
</TextBlock>
```

Kod taki byłby jednak dłuższy i z tego powodu zazwyczaj wybierany jest ten pierwszy wariant.

Zwróć uwagę, że choć definiowana tu kontrolka jest powiązana bezpośrednio tylko z jednym polem tekstowym, to efekt zmian w postaci aktualnej liczby znaków widzimy także wówczas, gdy użytkownik zmienia tekst w drugim polu tekstowym, co jest konsekwencją tego, że oba pola tekstowe są ze sobą powiązane.

Właściwość `StringFormat`, którą wykorzystaliśmy w tym zadaniu, stanowi jedną z możliwości WPF w zakresie definiowania formatu wyświetlanej wartości dla wiązanych danych. W dalszej części książki (w rozdziale 11.) omówię inne możliwości, a mianowicie szablony danych i konwertery.

## Rozdział 6.

# Wiązanie kolekcji danych — aplikacja Lista produktów

W poprzednim rozdziale przedstawiłam program, w którym zostało zastosowane wiązanie właściwości obiektu klasy z kontrolkami umieszczonymi w panelu `Grid`. W wielu aplikacjach potrzebujemy jednak wyświetlić na ekranie całą kolekcję obiektów, a nie tylko pojedynczy rekord. W tym rozdziale wykorzystamy kontrolkę `ListView`, która pozwala prezentować kolekcję danych. Do wiązania danych zostanie użyta właściwość `ItemsSource`.

## 6.1 Kod XAML

Wykonamy program wyświetlający listę produktów w postaci tabeli z użyciem kontrolki `ListView`. W tym celu otwórz nowy projekt WPF. W oknie *XAML* podmień tytuł okna na **Lista produktów**. Domyślny rozmiar okna można pozostawić bez zmian lub nieznacznie zmniejszyć.

Następnie w oknie *XAML* podmień kod ze znacznikami `<Grid>` na następujący:

```
<Grid>
  <ListView x:Name="lstProdukty">
    <ListView.View>
      <GridView>
        <GridView.Columns>
          <GridViewColumn Header="Symbol"
            DisplayMemberBinding="{Binding Symbol}"/>
          <GridViewColumn Header="Nazwa"
            DisplayMemberBinding="{Binding Nazwa}"/>
          <GridViewColumn Header="Liczba sztuk"
            DisplayMemberBinding="{Binding LiczbaSztuk}"/>
        </GridView.Columns>
      </GridView>
    </ListView.View>
  </ListView>
</Grid>
```

```

        <GridViewColumn Header="Magazyn"
            DisplayMemberBinding="{Binding Magazyn}" />
    </GridView.Columns>
</GridView>
</ListView.View>
</ListView>
</Grid>

```

Kontrolka `ListView` dostarcza widok `GridView`, prezentujący dane w postaci tabeli. Widok ten ustawiany jest za pomocą właściwości `View`. Właściwość `GridView.Columns` służy do przechowywania kolekcji obiektów typu `GridViewColumn`, opisujących poszczególne kolumny. Widok `GridView` (udostępniany przez klasę o tej samej nazwie) ma przydatne funkcjonalności:

- ◆ Pozwala na zmianę kolejności kolumn przez przeciągnięcie myszą.
- ◆ Umożliwia zmianę szerokości kolumn przez przeciągnięcie separatora kolumn.
- ◆ Daje możliwość zmiany szerokości kolumny do domyślnej (dopasowanej do zawartości kolumny) poprzez dwukrotne kliknięcie separatora kolumny.

W opisie każdej kolumny istotne są dwie właściwości: `Header`, której przypisuje się tytuł kolumny, oraz `DisplayMemberBinding`, która umożliwia wiązanie z odpowiednią właściwością klasy.

## 6.2 Definicja klasy Produkt i code-behind

W kolejnym kroku stworzymy nową klasę w osobnym pliku nazwanym *Produkt.cs*. Należy tam wpisać kod klasy `Produkt` umieszczony na początku podrozdziału 5.3.

Kolekcję produktów można umieścić w zwykłej liście `List<T>`, ale w dalszej części pracy nad programem będziemy dodawać nowe produkty i kasować istniejące. Zamiast odświeżać widok po każdej takiej zmianie możemy skorzystać z kolekcji `ObservableCollection<T>`, która pozwala na automatyczną aktualizację kontrolki powiązanej z kolekcją. Klasa `ObservableCollection` jest wyposażona w mechanizm powiadamiania o zmianach (wynikłych z dodania nowego obiektu, wykasowania lub odświeżania kolekcji). Użycie tej klasy wymaga deklaracji przestrzeni nazw, którą należy umieścić w pliku *MainWindow.xaml.cs*:

```
using System.Collections.ObjectModel;
```

Następnie w pliku *MainWindow.xaml.cs* podmień kod klasy `MainWindow` na następujący:

```

public partial class MainWindow : Window
{
    private ObservableCollection<Produkt> ListaProduktow = null;
    public MainWindow()
    {

```



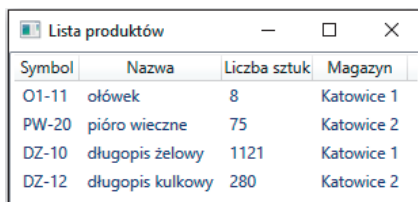
```
        InitializeComponent();
        PrzygotujWiazanie();
    }

    private void PrzygotujWiazanie()
    {
        ListaProduktow = new ObservableCollection<Produkt>();
        ListaProduktow.Add(new Produkt("O1-11", "ołówek", 8, "Katowice 1"));
        ListaProduktow.Add(new Produkt("PW-20", "pióro wieczne", 75, "Katowice 2"));
        ListaProduktow.Add(new Produkt("DZ-10", "długopis żelowy", 1121,
            "Katowice 1"));
        ListaProduktow.Add(new Produkt("DZ-12", "długopis kulkowy", 280,
            "Katowice 2"));
        1stProdukty.ItemsSource = ListaProduktow;
    }
}
```

W programie tym, podobnie jak w programie z rozdziału 5., połączenie z danymi wykonane zostało w *code-behind*, tym razem jednak została użyta właściwość `ItemsSource`, która umożliwia wiązanie kolekcji danych. Program można już uruchomić i przetestować (rysunek 6.1).

#### Rysunek 6.1.

*Działanie programu  
Lista produktów*



Symbol	Nazwa	Liczba sztuk	Magazyn
O1-11	ołówek	8	Katowice 1
PW-20	pióro wieczne	75	Katowice 2
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2

W dalszej części rozdziału będziemy dodawać do wykonanego programu przydatne funkcjonalności, takie jak sortowanie, formatowanie danych w wykazie i filtrowanie. Kolejnym krokiem będzie dołączenie do bieżącego programu funkcjonalności wykonanej w rozdziale 5. (edytowanie pojedynczego produktu).

## 6.3 Sortowanie wykazu

Możemy wykaz posortować według pojedynczego lub złożonego kryterium. Najpierw posortujemy według pojedynczego kryterium, mianowicie według magazynu. W tym celu dodaj do przestrzeni nazw dyrektywę `using System.ComponentModel;`, a następnie na końcu metody `PrzygotujWiazanie` dopisz kod:

```
CollectionView widok =
    (CollectionView)CollectionViewSource.DefaultView(1stProdukty.ItemsSource);
widok.SortDescriptions.Add(new SortDescription("Magazyn",
    ListSortDirection.Ascending));
```

Każde wiązanie z kolekcją ma przypisany niejawnie widok domyślny. W pierwszej linii prezentowanego kodu widok ten został odczytany i przypisany do obiektu *widok*. W kolejnej linii odczytany widok zostaje rozszerzony o możliwość sortowania według wybranej komuny. Sortowanie widoku umożliwia właściwość *SortDescriptions*. Można do niej wpisać obiekty typu *SortDescription*, podając dla konstruktora dwa argumenty: właściwość, według której ma być przeprowadzone sortowanie (w naszym programie będzie to *Magazyn*), oraz wartość typu wyliczeniowego, określająca porządek sortowania — rosnący (*Ascending*) lub malejący (*Descending*).

Obecnie wykaz produktów został posortowany według magazynu (rysunek 6.2).

### Rysunek 6.2.

*Wykaz produktów  
posortowany  
według magazynu*

Symbol	Nazwa	Liczba sztuk	Magazyn
O1-11	ołówek	8	Katowice 1
DZ-10	długopis żelowy	1121	Katowice 1
PW-20	pióro wieczne	75	Katowice 2
DZ-12	długopis kulkowy	280	Katowice 2

Możemy ustalić **kryterium złożone** sortowania: przeprowadzimy sortowanie według magazynu, a w ramach magazynu — według nazwy produktu. W tym celu na końcu metody *PrzygotujWiazanie* dopisz linię:

```
widok.SortDescriptions.Add(new SortDescription("Nazwa",
    ListSortDirection.Ascending));
```

Po uruchomieniu programu wykaz zostanie posortowany według magazynu i nazwy produktu (rysunek 6.3).

### Rysunek 6.3.

*Wykaz produktów  
posortowany  
według magazynu  
i nazwy produktu*

Symbol	Nazwa	Liczba sztuk	Magazyn
DZ-10	długopis żelowy	1121	Katowice 1
O1-11	ołówek	8	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2
PW-20	pióro wieczne	75	Katowice 2

## 6.4 Formatowanie danych w wykazie

Dokonamy drobnych zmian dotyczących formatowania tekstu w kolumnie *Liczba sztuk*. Wyróżnimy wartości pokazywane w tej kolumnie, zmieniając ich kolor, rozmiar czcionki i typ czcionki na pogrubioną. W tym celu w oknie *XAML* podmień fragment kodu. Zamiast dotychczasowych linii:

```
<GridViewColumn Header="Liczba sztuk"
    DisplayMemberBinding="{Binding LiczbaSztuk}"/>
```

wpisz kod:

```

<GridViewColumn Header="Liczba sztuk">
  <GridViewColumn.CellTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding LiczbaSztuk}"
        FontSize="12" Foreground="Green" FontWeight="Bold"/>
    </DataTemplate>
  </GridViewColumn.CellTemplate>
</GridViewColumn>

```

Nowy kod jest wyraźnie dłuższy — zamiast właściwości `DisplayMemberBinding` została tu użyta właściwość `GridViewColumn.CellTemplate`. W ten sposób „zasłaniamy” domyślny szablon i definiujemy swoje preferencje dotyczące wskazanej kolumny. Rysunek 6.4 przedstawia efekt działania programu po modyfikacjach.

#### Rysunek 6.4.

*Lista produktów po zmianie formatowania kolumny Liczba sztuk*



Symbol	Nazwa	Liczba sztuk	Magazyn
DZ-10	długopis żelowy	1121	Katowice 1
O1-11	ołówek	8	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2
PW-20	pióro wieczne	75	Katowice 2

## 6.5 Wyrównanie tekstu w kolumnie

Wszystkie kolumny prezentują nam dane domyślnie z wyrównaniem do lewej (nagłówki są centrowane). W przypadku kolumny *Liczba sztuk*, która zawiera wartości liczbowe, warto byłoby ustawić wyrównanie do prawej. Wydawałoby się, że to powinna być kwestia dodania jednej właściwości do definicji `TextBlock`, mianowicie `TextAlignment="Right"`. To jednak nie wystarczy, dodanie tego zapisu nie zmieni wyrównania tekstu. W przypadku kontrolki `ListView` konieczna jest uprzednia zmiana domyślnego ustawienia wyrównania zawartości w poziomie z wartości `Left` na `Stretch`. Aby było nam łatwiej zauważyć modyfikację tego ustawienia, najpierw zmienimy kolor tła (`Background`) na jasnoszary. Podmień znacznik `TextBlock` na następujący:

```


<TextBlock Text="{Binding LiczbaSztuk}"
  FontSize="12" Foreground="Green" FontWeight="Bold"
  Background="LightGray"/>

```

Do poprzedniego zapisu zostało dodane ustawienie właściwości dla tła: `Background="LightGray"`. Zmiana koloru tła pozwoli nam uwidocznnić obszar, jaki zajmują wartości danych w kolumnie (rysunek 6.5).

#### Rysunek 6.5.

*Lista produktów po dodaniu tła w kolumnie Liczba sztuk*



Symbol	Nazwa	Liczba sztuk	Magazyn
DZ-10	długopis żelowy	1121	Katowice 1
O1-11	ołówek	8	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2
PW-20	pióro wieczne	75	Katowice 2

Jak możemy zaobserwować, każda „komórka” w kolumnie, to znaczy każdy pojedynczy `TextBlock`, ma rozmiar dopasowany do zawartości danej, a nie do szerokości kolumny. Moglibyśmy ustalić na sztywno szerokość kontrolki poprzez właściwość `Width`, ale to nie byłby dobry pomysł, ponieważ użytkownik może zmieniać szerokość kolumny i zdecydowanie lepsze będzie takie rozwiązanie, które pozwoli rozciągnąć rozmiar komórki do aktualnej szerokości kolumny. Zmienimy sposób wyrównania zawartości w poziomie z domyślnego `Left` na `Stretch`. Przed znacznikiem otwierającym panel `Grid` (czyli przed panelem) wpisz kod:

```
<Window.Resources>
    <Style TargetType="ListViewItem">
        <Setter Property="HorizontalContentAlignment" Value="Stretch" />
    </Style>
</Window.Resources>
```

Zasoby i style zostaną omówione w rozdziale 10. Na razie wystarczy wiedzieć, że prezentowany kod dokonuje zmiany domyślnych ustawień dla klasy `ListViewItem` odpowiedzialnej za wygląd elementu wyświetlanego w `ListView`.

Po uruchomieniu programu uzyskamy oczekiwany efekt dopasowania rozmiaru elementów w kolumnie do jej aktualnej szerokości (rysunek 6.6a). Komórka została rozciągnięta na całą szerokość kolumny i teraz możemy „manewrować” pozycją tekstu w tym obszarze. Dodamy zatem ustawienie wyrównania dla bloku tekstu do prawej: `TextAlignment="Right"`. Obecnie kod definiujący `TextBlock` będzie miał postać:

```
<TextBlock Text="{Binding LiczbaSztuk}" FontSize="12" Foreground="Green"
    FontWeight="Bold" Background="LightGray" TextAlignment="Right"/>
```

Po tej zmianie program wyświetli wartości w kolumnie *Liczba sztuk* wyrównane do prawej (rysunek 6.6b).

Symbol	Nazwa	Liczba sztuk	Magazyn
DZ-10	długopis żelowy	1121	Katowice 1
O1-11	ołówek	8	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2
PW-20	pióro wieczne	75	Katowice 2

**Rysunek 6.6.** Lista produktów po ustawieniu wyrównania elementu w `ListView` na `Stretch` (a) i dodatkowo po ustawieniu wyrównania tekstu do prawej (b)

## 6.6 Filtrowanie danych

Dodamy możliwość filtrowania wykazu według nazwy produktu. Najpierw rozbudujemy panel `Grid`, ponieważ dotychczas zawierał tylko jedną domyślną komórkę, w której była lista produktów. W tym celu w oknie `XAML` między znacznikiem `<Grid>` a znacznikiem `<ListView x:Name="lstProdukty">` wpisz fragment kodu:

```

<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="Auto" />
  <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>
<Label Content="Filtruj wg nazwy produktu:" Grid.Row="0" Grid.Column="0"
  Margin="5" />
<TextBox Name="txtFilter" Grid.Row="0" Grid.Column="1" Margin="5"
  TextChanged="txtFilter_TextChanged" />

```

Należy także zmienić znacznik otwierający dla `ListView` — zamiast `<ListView x:Name="lstProdukty">` ma być:

```

<ListView x:Name="lstProdukty" Grid.Row="1" Grid.Column="0"
  Grid.ColumnSpan="2" Margin="5">

```

Następnie przechodzimy do pliku `MainWindow.xaml.cs` i tam na końcu metody `Przygotuj` ↪ `Wiazanie` dodamy instrukcję:

```

widok.Filter = FiltrUzytkownika; // Przypisanie delegata do właściwości filtra

```

W kolejnym kroku, w tym samym pliku, do klasy `MainWindow` dopiszemy kod dwóch metod:

```

private bool FiltrUzytkownika(object item)
{
    if (String.IsNullOrEmpty(txtFilter.Text))
        return true;
    else
        return ((item as Produkt).Nazwa.IndexOf(txtFilter.Text,
            StringComparison.OrdinalIgnoreCase) >= 0);
}
private void txtFilter_TextChanged(object sender, System.Windows.Controls.
    ↪ TextChangedEventArgs e)
{
    CollectionViewSource.GetDefaultView(lstProdukty.ItemsSource).Refresh();
}

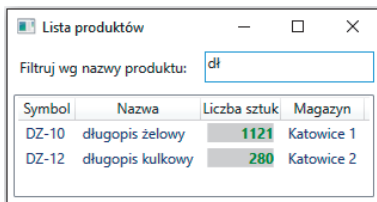
```

Właściwość `Filter` korzysta z delegata przyjmującego jeden argument typu `object` i zwracającego wartość typu `bool`. Takie rozwiązanie pozwala w wygodny sposób zdefiniować kryterium filtrowania danych, co zostało zrealizowane w metodzie delegata `FiltrUzytkownika`. Jeżeli w polu tekstowym są jakieś znaki (czyli nie jest puste), zwracana jest wartość logiczna `true`, gdy efekt metody `IndexOf` sprawdzającej umiejscowienie podłańcucha `txtFilter.Text` w nazwie produktu jest liczbą równą lub większą od zera<sup>1</sup>; w przeciwnym razie zwracana jest wartość `false`. Ponieważ argument metody jest typu `object`, musi być wykonana konwersja typu na naszą klasę `Produkt`, co zostało zrobione przy użyciu operatora `as` (`item as Produkt`). Użycie elementu typu wyliczeniowego `StringComparison.OrdinalIgnoreCase` jako drugiego argumentu metody

<sup>1</sup> Metoda `IndexOf` zwraca wartość ujemną (`-1`), jeśli nie znajduje szukanego podłańcucha w danym ciągu.

`IndexOf` pozwala ignorować wielkość liter. Po dokonaniu wymienionych zmian można uruchomić program i filtrować dane według nazwy produktu (rysunek 6.7).

**Rysunek 6.7.**  
Filtrowanie listy  
produktów  
(wyświetlone  
zostały produkty  
mające w nazwie  
łańcuch „dl”)



Obsługa zdarzenia zmiany tekstu w polu tekstowym `txtFilter_TextChanged` wymusza odświeżanie zawartości `ListView` po każdym wpisanym znaku.

## 6.7 Edycja danych w nowym oknie

Połączymy teraz dwa programy w jedną całość, to znaczy bieżący program z wykonaną w poprzednim rozdziale aplikacją *Produkt*. W programie z listą produktów utworzymy nowe okno, które zostanie wyświetlone po wybraniu produktu z listy (przez podwójne kliknięcie myszą).

W tym celu w oknie *Solution Explorer* kliknij nazwę projektu, a następnie prawym klawiszem myszy wybierz z menu kontekstowego pozycję *Add New Item* i dalej *Window (WPF)*. Dla nowego okna możemy zostawić domyślną nazwę *Window1.xaml*.

W widoku *Design* przejdź do okna głównego (*MainWindow*) i zaznacz myszą jako aktywną kontrolkę `ListView`, a następnie odszukaj w oknie *Properties* zdarzenie `Mouse` → `DoubleClick`. Zdefiniuj następującą obsługę tego zdarzenia:

```
private void 1stProdukty_MouseDoubleClick(object sender, MouseButtonEventArgs e)
{
    Window1 okno1 = new Window1(this);
    okno1.Show();
}
```

W nowym oknie *Window1.xaml* ustaw atrybut `Title="Produkt"`. Zmień rozmiar okna aplikacji: `Height="220" Width="350"`, a następnie przekopiuj kod panelu `Grid` z aplikacji *Produkt* (umieszczonego w podrozdziale 5.2).

W *code-behind* dla nowego okna (w pliku *Window1.xaml.cs*) dodaj przestrzeń nazw:

```
using System.ComponentModel;
```

Następnie podmień kod klasy `Window1` na następujący:

```
public partial class Window1 : Window
{
    private MainWindow mainWindow = null;    // Obiekt dla okna głównego

    public Window1()
    {
```

```

        InitializeComponent();
    }
    // Przeladowana (przeciążona) wersja konstruktora z obiektem dla głównego okna jako
    // argumentem
    public Window1(MainWindow mainWin)
    {
        InitializeComponent();
        mainWindow = mainWin;
        PrzygotujWiazanie();
    }
    private void PrzygotujWiazanie()
    {
        Produkt produktZListy = mainWindow.lstProdukty.SelectedItem as Produkt;
        if (produktZListy != null)
        {
            gridProdukt.DataContext = produktZListy; // Wybrany produkt z listy
        }
    }
    private void btnPotwierdz_Click(object sender, RoutedEventArgs e)
    {
        this.Close(); // Zamknięcie bieżącego okna
    }
}

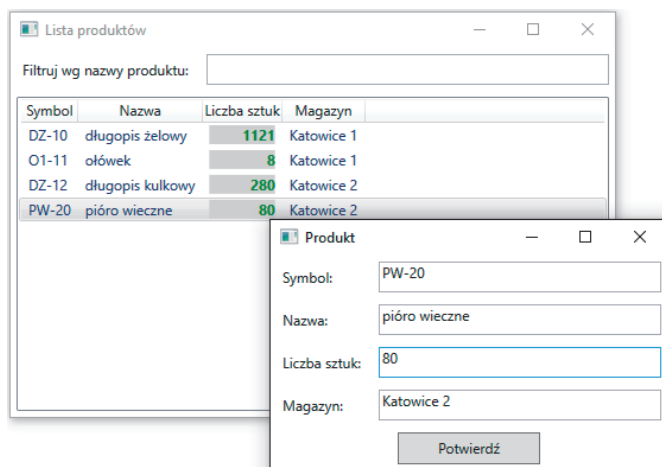
```

Dodano przeladowaną wersję konstruktora, której argumentem jest obiekt okna głównego (*MainWindow*). W metodzie *PrzygotujWiazanie* do obiektu *produktZListy* przypisana została wartość właściwości *SelectedItem*, która zwraca wybrany obiekt z listy w oknie głównym. Konieczna jest konwersja typu tego obiektu z *object* do typu klasy *Produkt*. Następnie jest ustalany kontekst dla wiązania danych. W podrozdziale 6.9 zaprezentuję inny sposób połączenia wybranej pozycji w kolekcji z danymi umieszczanymi w osobnym panelu (w kodzie XAML, a nie w *code-behind*).

W metodzie obsługi zdarzenia kliknięcia przycisku *Potwierdz* zamykane jest bieżące okno z danymi wybranego produktu.

Program można uruchomić i przetestować. Przykładowe działanie aplikacji przedstawia rysunek 6.8.

**Rysunek 6.8.**  
Aplikacja Lista produktów z możliwością edycji wybranego produktu w osobnym oknie



Program w obecnej postaci pozwala otworzyć więcej okien dodatkowych niż jedno. W niektórych aplikacjach takie działanie jest pożądane, w innych nie. W tym przypadku lepszym rozwiązaniem będzie blokowanie pracy w oknie głównym, dopóki nie zostanie zamknięte okno dodatkowe. Można ten efekt łatwo uzyskać, zmieniając wywołanie metody prezentującej okno z `Show` na `ShowDialog`:

```
okno1.ShowDialog();
```

W WPF obok wbudowanych okien, takich jak `MessageBox`, `OpenFileDialog` czy `SaveFileDialog`<sup>2</sup>, można tworzyć własne okna dialogowe. Definiuje się je podobnie jak zwykłe okna WPF, a różni je przede wszystkim sposób wywołania. Zastosowanie metody `ShowDialog` (zamiast `Show`) powoduje zablokowanie głównego okna aplikacji do czasu zamknięcia okna dialogowego. Ponadto okno dialogowe może zwrócić jakiś rezultat. W celu zamknięcia okna dialogowego można zamiast wywołania metody `Close` użyć właściwości `DialogResult`, przypisując jej wartość `false` lub `true`. W przypadku omawianego okna (w którym nie ma przycisku *Anuluj*) możemy wpisać wartość `true`:

```
private void btnPotwierdz_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
}
```

W prezentowanym programie formularz z danymi wybranego produktu niekoniecznie musiał być tworzony w nowym oknie. Można w tym celu wyodrębnić fragment głównego okna (odpowiednio powiększonego). Taką wersję programu należy wykonać w ramach jednego z zadań w kolejnym podrozdziale.

## 6.8 Zadania

Zadania dotyczą pisanego w tym rozdziale programu, który należy rozbudować o możliwość usuwania i dodawania produktów. Trzecie zadanie polega na napisaniu nowej wersji programu, w której wszystkie operacje będą wykonywane w jednym oknie.

### Zadanie 6.1

Dodaj do wykonanego w tym rozdziale programu możliwość kasowania wybranej pozycji.

### Zadanie 6.2

Uzupełnij pisany w tym rozdziale program o możliwość dodawania nowych produktów.

---

<sup>2</sup> Okna dialogowe `OpenFileDialog` i `SaveFileDialog` zostaną wykorzystane w kolejnych rozdziałach.



## Zadanie 6.3

Wykonaj nowy projekt na podstawie napisanego w tym rozdziale programu. W nowym projekcie muszą się znaleźć wszystkie funkcjonalności, to znaczy lista produktów, edycja wybranego produktu, usuwanie produktów i dodawanie nowych produktów, ale tym razem wszystko powinno być wykonane w jednym oknie.

## 6.9 Wskazówki do zadań

Wskazówki, podobnie jak w innych rozdziałach, zawierają szczegółowe wyjaśnienia i fragmenty rozwiązań.

### Wskazówki do zadania 6.1

W pierwszej kolejności należy się zdecydować na sposób obsługi usuwania: czy służyć ma do tego klawisz (*Delete*), czy osobny przycisk, aczkolwiek równie dobrze można zaimplementować oba te rozwiązania. Przy założeniu, że jest to specjalny przycisk, dodaj w panelu `Grid` jeszcze jeden wiersz, a następnie umieść w tym wierszu przycisk z podpisem *Usuń*. W metodzie obsługującej zdarzenie kliknięcia tego przycisku należy:

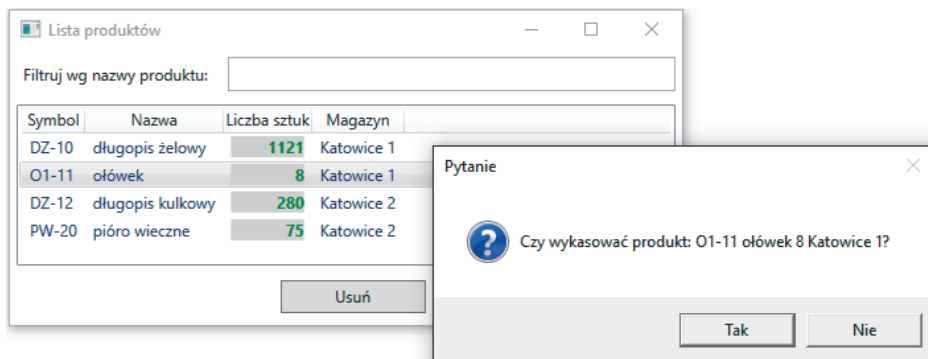
- ♦ Odczytać wybrany produkt z listy z użyciem właściwości `SelectedItem`:

```
Produkt produktZListy = lstProdukty.SelectedItem as Produkt;
```

- ♦ Zadać użytkownikowi pytanie, czy na pewno chce usunąć wybraną pozycję, używając metody `MessageBox.Show`. Na przykład:

```
MessageBoxResult odpowiedz = MessageBox.Show("Czy wykasować produkt: " +  
    produktZListy.ToString() + "?", "Pytanie", MessageBoxButtons.YesNo,  
    MessageBoxIcon.Question);
```

- ♦ Jeśli użytkownik odpowie, że tak, czyli `odpowiedz == DialogResult.Yes`, wówczas należy usunąć produkt z listy `ListaProduktow.Remove(produktZListy);`. Przykładowe działanie programu przedstawia rysunek 6.9.



Rysunek 6.9. Aplikacja Lista produktów z możliwością usuwania wybranego produktu

## Wskazówki do zadania 6.2

Wprowadzenie nowej pozycji do `ListView` można zrealizować na dwa sposoby: albo dodać ją do kolekcji `ObservableCollection<T>` (z domyślnymi wartościami dla wymaganych właściwości) i wówczas za sprawą mechanizmu wiązania będzie widoczna także w `ListView`, albo najpierw pracować na tymczasowym obiekcie klasy `Produkt` i dopiero po zamknięciu okna dialogowego dodać pozycję do kolekcji. Umieszczone tu wskazówki dotyczą tego drugiego sposobu.

W klasie dla okna dodatkowego `Window1` zdefiniuj prywatne pole klasy `czyNowyProdukt` typu `bool`, które będzie ustawiane na `true` w przypadku wprowadzania nowego produktu. Zdefiniuj także prywatne pole `nowyProdukt` typu `Produkt`.

W przeładowanym konstruktorze tej klasy dodaj kolejny argument. Konstruktor po zmianach:

```
public Window1(MainWindow mainWin, bool czyNowy=false)
{
    InitializeComponent();
    mainWindow = mainWin;
    czyNowyProdukt = czyNowy;
    PrzygotujWiazanie();
}
```

W metodzie `PrzygotujWiazanie` należy sprawdzić wartość pola `czyNowyProdukt` i jeśli jest `false`, ma zostać wykonane to, co było w dotychczasowej wersji tej metody; w przeciwnym razie należy przypisać do właściwości `DataContext` nowy obiekt:

```
nowyProdukt = new Produkt("AA-00", "", 0, ""); // Nowy produkt z domyślnymi
                                                // wartościami
gridProdukt.DataContext = nowyProdukt;
```

W metodzie obsługi zdarzenia kliknięcia przycisku *Potwierdź* należy (w przypadku, gdy pole `czyNowyProdukt` ma wartość `true`) dodać nowy produkt do listy:

```
mainWindow.ListaProduktow.Add(nowyProdukt);
```

Powyższa instrukcja będzie wymagała zmiany dostępu do pola `ListProduktow` w klasie `MainWindow` (z `private` na `internal`).

Umieść w panelu w oknie głównym przycisk *Dodaj*. W klasie `MainWindow` należy wykonać obsługę przycisku *Dodaj*, wywołując konstruktor z klasy `Window1` z dwoma argumentami:

```
Window1 okno1 = new Window1(this, true);
okno1.ShowDialog();
```

## Wskazówki do zadania 6.3

Utwórz nowy projekt, powiększ rozmiar okna (np. `Height="350" Width="600"`) i zdefiniuj panel zewnętrzny (np. `DockPanel`). Szkic układu może mieć taką oto postać:

```

<DockPanel LastChildFill="True">
    <Grid x:Name="gridProdukt" Margin="5"
        DataContext="{Binding SelectedItem, ElementName=lstProdukty}">
        <!-- Tu ma być panel Grid, jaki jest w oknie Window! -->
    </Grid>
    <Grid Margin="5">
        <!-- Tu ma być panel Grid z ListView -->
    </Grid>
</DockPanel>

```

W tej wersji programu właściwość `DataContext` dla wybranego produktu jest ustalana w kodzie XAML na podstawie właściwości `SelectedItem`.

Po przekopiowaniu zawartości obu paneli `Grid` z poprzedniej wersji programu w widoku *Design* będzie już widoczna całość okna. Przydatne będą drobne korekty w kodzie XAML, między innymi można do lewego panelu dodać w osobnym wierszu etykietę z opisem (np. „Produkt:”).

Zmień definicję `ListView`, usuwając z niej uchwyt do obsługi zdarzenia `MouseDoubleClick` → `Click`. Obecnie linia rozpoczynająca definicję `ListView` powinna mieć następującą postać:

```

<ListView x:Name="lstProdukty" Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2"
    Margin="5">

```

Jeśli chodzi o zmiany w kodzie w C#, najpierw przekopij zawartość klasy `MainWindow` z poprzedniej wersji programu i dodaj do projektu plik z klasą `Produkt`. Wykasuj metodę `lstProdukty_MouseDoubleClick`.

W metodzie `btnDodaj_Click` umieść jedynie instrukcje dodające produkt do kolekcji (z domyślnymi wartościami):

```

private void btnDodaj_Click(object sender, RoutedEventArgs e)
{
    nowyProdukt = new Produkt("AA-00", "", 0, "");
    ListaProduktow.Add(nowyProdukt);
}

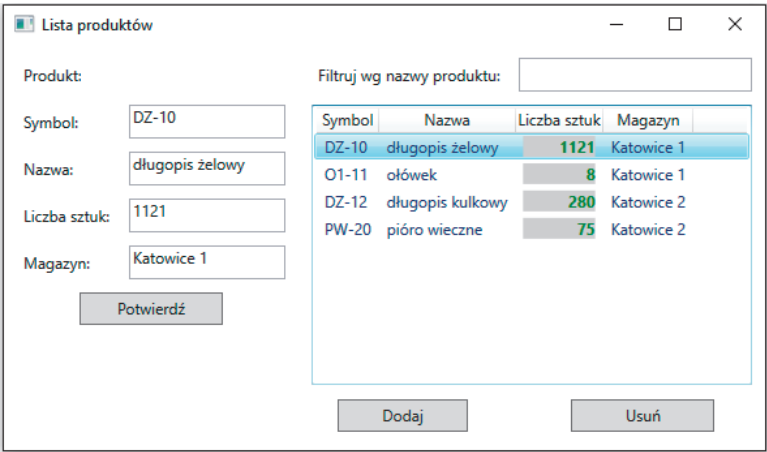
```

W tym rozwiązaniu prywatne pole logiczne `czyNowyProdukt` jest zbędne. W metodzie `btnPotwierdz_Click` należy usunąć dotychczasowy kod.

Tak przygotowany program można już uruchomić (rysunek 6.10).

Program należałoby jeszcze udoskonalić. W obecnej postaci programu przycisk *Potwierdź* nie ma właściwie nic do roboty, bo już nie zamyka żadnych okien ani nie dodaje pozycji do kolekcji. Przydaje się jedynie do zmiany fokusu, która wymusza aktualizację wprowadzanej pozycji w `ListView`. Można go usunąć albo zostawić i w przyszłości wykorzystać do sprawdzania poprawności danych (dla całego wiersza). Wskazana jest zmiana nazwy przycisku umożliwiającego dodanie nowego produktu z *Dodaj* na przykład na *Nowy produkt*. W poprzednich warunkach, z osobnymi oknami, przycisk *Dodaj* był jednoznaczny, a obecnie już niekoniecznie.

**Rysunek 6.10.**  
*Aplikacja Lista  
produktów  
w jednym oknie*



## Rozdział 7.

# Kontrolka DataGrid — aplikacja Edycja produktów

`DataGrid` jest rozbudowaną kontrolką, która przypomina `ListView`, ma jednak więcej możliwości. W niniejszym rozdziale zaprezentuję wybrane spośród tych możliwości.

## 7.1 Kontrolka DataGrid z autogenerowaniem kolumn

Napiszemy program wyświetlający listę produktów w postaci tabeli z użyciem kontrolki `DataGrid`. Otwórz nowy projekt WPF. W oknie *XAML* podmień tytuł okna na „Edycja produktów”. Domyślny rozmiar okna możesz zmienić na przykład na `Height="180" Width="350"`. Następnie w oknie *XAML* podmień kod ze znacznikami `<Grid>` na następujący:

```
<Grid Margin="10">
    <DataGrid Name="gridProdukty"/>
</Grid>
```

W kolejnym kroku stwórz nową klasę w osobnym pliku nazwanym *Produkt.cs*. Należy tam wpisać kod klasy `Produkt` umieszczony na początku podrozdziału 5.3, przy czym trzeba dodać do klasy konstruktor domyślny (bezargumentowy z pustymi klamrami) wymagany przez `DataGrid`:

```
public Produkt()
{ }
```

Następnie w pliku *MainWindow.xaml.cs* dodaj deklarację przestrzeni nazw:

```
using System.Collections.ObjectModel;
```

I podmień kod klasy `MainWindow` na następujący:

```
public partial class MainWindow : Window
{
    private ObservableCollection<Produkt> ListaProduktow = null;

    public MainWindow()
    {
        InitializeComponent();
        PrzygotujWiazanie();
    }

    private void PrzygotujWiazanie()
    {
        ListaProduktow = new ObservableCollection<Produkt>();
        ListaProduktow.Add(new Produkt("O1-11", "ołówek", 8, "Katowice 1"));
        ListaProduktow.Add(new Produkt("PW-20", "pióro wieczne", 75,
            "Katowice 2"));
        ListaProduktow.Add(new Produkt("DZ-10", "długopis żelowy", 1121,
            "Katowice 1"));
        ListaProduktow.Add(new Produkt("DZ-12", "długopis kulkowy", 280,
            "Katowice 2"));
        gridProdukty.ItemsSource = ListaProduktow;
    }
}
```

To wszystko. Program można już uruchomić (rysunek 7.1).

**Rysunek 7.1.**  
*Aplikacja Edycja  
produktów*

Symbol	Nazwa	LiczbaSztuk	Magazyn
O1-11	ołówek	8	Katowice 1
PW-20	pióro wieczne	75	Katowice 2
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2

Kod XAML jest wyjątkowo treściwy w przypadku tego programu — zawartość panelu `Grid` składa się właściwie z jednej linii: `<DataGrid Name="gridProdukty"/>`. Nie ma tu żadnych opisów dotyczących kolumn, jakie mają być wyświetlone. Schemat tabeli został ustalony na podstawie typu danych w kolekcji, z którą została związana kontrolka `DataGrid`. W nagłówkach tabeli widzimy pola klasy `Produkt`, a w poszczególnych wierszach obiekty umieszczone w kolekcji połączonej z `DataGrid`. Po uruchomieniu możemy się przekonać, że dane w tabeli dostępne są do edycji, można także dodać nowy produkt<sup>1</sup> (wpisując jego dane w pustym wierszu) oraz usunąć pozycję z wykazu za pomocą klawisza `Delete`. Wyłączenie możliwości edycji, dodawania nowego obiektu oraz usuwania wymagałoby następującej zmiany definicji kontrolki `DataGrid` w kodzie XAML:

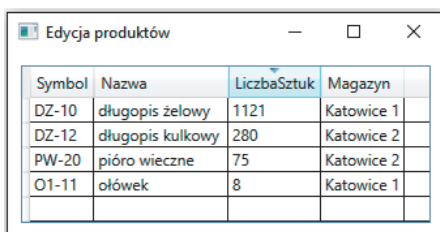
```
<DataGrid Name="gridProdukty" IsReadOnly="True" CanUserAddRows="False"
    CanUserDeleteRows="False"/>
```

<sup>1</sup> Dodawanie i usuwanie pozycji w `DataGrid` jest dostępne dla kolekcji obsługujących dodawanie i usuwanie elementów (tablica nie obsługuje).

Ponadto możliwe jest regulowanie szerokości kolumn i wysokości wierszy oraz zmiana ich kolejności, podobnie jak to miało miejsce w `ListView`. W przeciwieństwie do `ListView` kontrolka `DataGrid` wyposażona jest w możliwość sortowania wykazu według wybranej kolumny (poprzez kliknięcie jej nazwy). Przykładowy wykaz posortowany malejąco według liczby sztuk przedstawia rysunek 7.2. Kolumna wybrana do sortowania jest wyróżniona, a kierunek sortowania (malejący lub rosnący) jest zobrazowany za pomocą ikonki z grotem strzałki (trójkątem) skierowanym w dół lub w górę.

### Rysunek 7.2.

Wykaz produktów  
posortowany  
malejąco według  
liczby sztuk



Symbol	Nazwa	LiczbaSztuk	Magazyn
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2
PW-20	pióro wieczne	75	Katowice 2
O1-11	ołówek	8	Katowice 1

Kontrolka `DataGrid` ma wiele możliwości, wszystkich nie zdołam tu przedstawić, ale zanim zaczniesz poznawać kolejne, warto zauważyć, jak wiele funkcjonalności dostępnych jest automatycznie. Możliwości w zakresie autogenerowania kolumn są bogatsze, niż widzimy to na analizowanym przykładzie. Sposób wyświetlania kolumn jest dopasowany do ich typu. Przykładowo dane typu `bool` wyświetlane są domyślnie w postaci kontrolki `CheckBox`, a dane typu `Uri` wyświetlają się jako linki.

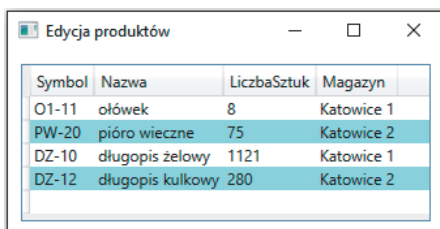
Funkcje automatycznie obsługiwane przez kontrolkę `DataGrid`, takie jak zmiana kolejności, zmiana szerokości kolumn i wysokości wierszy oraz sortowanie, można wyłączyć przez ustawienie właściwości `CanUserReorderColumns`, `CanUserResizeColumns`, `CanUserResizeRows` i `CanUserSortColumns` na wartość `False`. Ustawienie właściwości `HeadersVisibility` na `None` oznacza rezygnację z pokazywania nagłówków<sup>2</sup>.

`DataGrid` ma także inne właściwości, poprzez które możemy wpłynąć na wygląd tabeli. Właściwość `AlternatingRowBackground` pozwala ustawić kolor tła dla wyróżnionego wiersza (domyślnie co drugiego). Ustawiając właściwość `GridLinesVisibility` na `None`, można wyłączyć wyświetlanie linii siatki. Rysunek 7.3 przedstawia wykaz produktów dla następujących ustawień kontrolki `DataGrid`:

```
<DataGrid Name="gridProdukty" AlternatingRowBackground="LightBlue"
          GridLinesVisibility="None"/>
```

### Rysunek 7.3.

Wykaz produktów  
bez linii siatki  
i ze zmianą koloru  
tła w co drugim  
wierszu



Symbol	Nazwa	LiczbaSztuk	Magazyn
O1-11	ołówek	8	Katowice 1
PW-20	pióro wieczne	75	Katowice 2
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2

<sup>2</sup> Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 281).

Istnieje możliwość wyróżniania wiersza (poprzez kolorowanie tła) po dowolnej liczbie pozycji. W tym celu należy ustawić dodatkowo właściwość `AlternationCount`. Przykładowo ustawienie `AlternationCount="3"` spowoduje, że wyróżniona pozycja pojawi w co trzecim wierszu.



Uwaga

W prezentowanych przykładach (w tym rozdziale i w poprzednich) występuje kilka różnych terminów ze słowem „grid”. *Grid* to siatka, czyli układ przypominający tabelę (z wierszami i kolumnami). Poznaliśmy panel `Grid`, który pozwala umieścić inne kontrolki w układzie siatki. Ponadto używaliśmy kontrolki `ListView` w widoku `GridView`. A w tym rozdziale używamy kontrolki `DataGrid`, która udostępnia dane do edycji. To są trzy różne konteksty dla owej „siatki”.

## 7.2 Definiowanie kolumn dla `DataGrid`

W poprzednim podrozdziale wykorzystaliśmy możliwość autogenerowania kolumn w kontrolce `DataGrid`. Nagłówki kolumn zostały automatycznie pobrane na podstawie nazw właściwości klasy `Produkt`. Takie rozwiązanie ma jednak swoje ujemne strony. Przykładowo w nazwach właściwości nie stosuje się spacji. W naszym programie widzimy ten efekt w kolumnie *LiczbaSztuk*. Ponadto nazwy właściwości mogą być w innym języku (np. angielskim, a nie polskim). To jest jeden z powodów, dla których automatycznie wygenerowana „siatka” (tabela) kontrolki `DataGrid` może nam nie wystarczać. Innym powodem może być potrzeba indywidualnego sformatowania wybranych kolumn. Na przykład dla kolumn zawierających liczby oczekiwane zazwyczaj jest wyrównanie do prawej. W takich przypadkach musimy zdefiniować kolumny „ręcznie”. Wykonamy teraz taką wersję programu.

W tym celu w oknie *XAML* podmień kod ze znacznikami `DataGrid` na następujący:

```
<DataGrid Name="gridProdukty" AutoGenerateColumns="False"
AlternatingRowBackground="LightBlue" GridLinesVisibility="None">
  <DataGrid.Columns>
    <DataGridTextColumn Header="Symbol" Binding="{Binding Symbol}" />
    <DataGridTextColumn Header="Nazwa" Binding="{Binding Nazwa}" />
    <DataGridTextColumn Header="Liczba sztuk"
      Binding="{Binding LiczbaSztuk}">
      <DataGridTextColumn.ElementStyle>
        <Style TargetType="{x:Type TextBlock}">
          <Setter Property="HorizontalAlignment" Value="Right" />
        </Style>
      </DataGridTextColumn.ElementStyle>
    </DataGridTextColumn>
    <DataGridTextColumn Header="Magazyn" Binding="{Binding Magazyn}" />
  </DataGrid.Columns>
</DataGrid>
```

Po wykonanej zmianie kodu XAML nagłówki kolumny *Liczba sztuk* zawiera spację i dane umieszczone w tej kolumnie wyrównane są do prawej (rysunek 7.4).



**Rysunek 7.4.**

*Wykaz produktów  
po zmianach  
formatowania*

Symbol	Nazwa	Liczba sztuk	Magazyn
O1-11	ołówek	8	Katowice 1
PW-20	pióro wieczne	75	Katowice 2
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2

W znacznikach `<DataGridTextColumn.ElementStyle>` ustawiono styl dla elementu kolumny *Liczba sztuk* (można osobno zmieniać styl nagłówka). Zagadnienia dotyczące stylów zostaną szerzej omówione w rozdziale 10.

W definicji kolumn został użyty typ kolumny `DataGridTextColumn`. Typ ten stosuje się dla zwykłych kolumn tekstowych korzystających w trybie odczytu z `TextBlock`, a podczas edycji z kontrolki `TextBox`. Kontrolka `DataGrid` obsługuje jeszcze inne typy kolumn: `DataGridHyperLinkColumn` dla hiperłączy, `DataGridCheckBoxColumn` wygodny do prezentacji dla danych typu `bool` oraz `DataGridComboBoxColumn` dla listy rozwijanej (podczas autogenerowania kolumn ten typ przypisany jest domyślnie dla danych typu wyliczeniowego). Jest jeszcze typ `DataGridTemplateColumn`, który pozwala zdefiniować dowolny szablon prezentowania danych (przy użyciu właściwości `CellTemplate` i `CellEditingTemplate`).

## 7.3 Kolumna `DataGridComboBoxColumn`

Wykorzystamy typ `DataGridComboBoxColumn` dla kolumny *Magazyn*. Typ `DataGridComboBoxColumn` jest wygodny dla danych typu wyliczeniowego, ale można go zastosować dla dowolnej kolekcji. My użyjemy go dla listy magazynów. W kodzie XAML podmień znacznik:

```
<DataGridTextColumn Header="Magazyn" Binding="{Binding Magazyn}"/>
```

na następujący:

```
<DataGridComboBoxColumn x:Name="nazwaMagazynu" Header="Magazyn"
    SelectedItemBinding="{Binding Magazyn}"/>
```

W *code-behind* (w klasie `MainWindow`) na końcu metody `PrzygotujWiazanie` dopisz kod:

```
ObservableCollection<string> ListaMagazynow =
    new ObservableCollection<string>() { "Katowice 1", "Katowice 2", "Gliwice 1" };
nazwaMagazynu.ItemsSource = ListaMagazynow;
```

Program można już uruchomić. Podczas edycji danej w kolumnie *Magazyn* wyświetla się lista rozwijana (rysunek 7.5).

**Rysunek 7.5.**

Wykaz produktów  
po zmianie typu  
kolumny dla  
magazynu

Symbol	Nazwa	Liczba sztuk	Magazyn
O1-11	ołówek	8	Katowice 1
PW-20	pióro wieczne	75	Katowice 2
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2

## 7.4 Wiązanie kontrolki DataGrid z dokumentem XML

Dane umieszczane w kontrolce `DataGrid` można bezpośrednio powiązać z plikiem XML. Zaprezentuję prosty przykład obrazujący tę możliwość.

Otwórz nowy projekt WPF. W oknie *XAML* podmień tytuł okna i zmień jego wymiary, to znaczy ustaw atrybuty: `Title="Edycja produktów z pliku XML"` `Height="210"` `Width="360"`.

W oknie *Solution Explorer* podświetl nazwę projektu i prawym klawiszem myszy dodaj folder o nazwie **dane** (*Add/New Folder*). Następnie podświetl nazwę nowego folderu *dane* i prawym klawiszem myszy dodaj plik XML o nazwie **Produkty.xml** (*Add/New Item/XML file*). Do utworzonego pliku *Produkty.xml* wpisz jego nową zawartość:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<ListaProduktow>
  <Produkt>
    <Symbol>O1-11</Symbol>
    <Nazwa>ołówek</Nazwa>
    <LiczbaSztuk>8</LiczbaSztuk>
    <Magazyn>Katowice 1</Magazyn>
  </Produkt>
  <Produkt>
    <Symbol>PW-20</Symbol>
    <Nazwa>pióro wieczne</Nazwa>
    <LiczbaSztuk>75</LiczbaSztuk>
    <Magazyn>Katowice 2</Magazyn>
  </Produkt>
  <Produkt>
    <Symbol>DZ-10</Symbol>
    <Nazwa>długopis żelowy</Nazwa>
    <LiczbaSztuk>1121</LiczbaSztuk>
    <Magazyn>Katowice 1</Magazyn>
  </Produkt>
  <Produkt>
    <Symbol>DZ-12</Symbol>
    <Nazwa>długopis kulkowy</Nazwa>
```

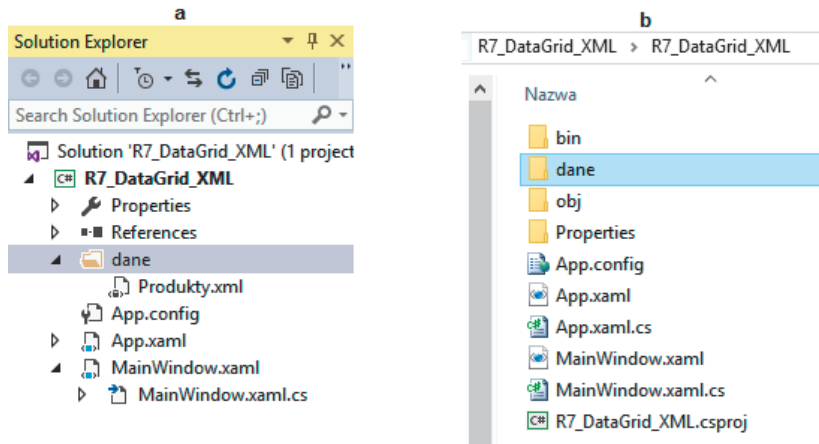
```

<LiczbaSztuk>280</LiczbaSztuk>
<Magazyn>Katowice 2</Magazyn>
</Produkt>
</ListaProduktow>

```

Po wykonaniu opisanych czynności widoczny w oknie *Solution Explorer* folder *dane* będzie zawierał plik *Produkty.xml* (rysunek 7.6a). Jak możemy sprawdzić w eksploratorze plików w systemie Windows, nowy folder *dane* został utworzony w folderze projektu<sup>3</sup> na tym samym poziomie co katalogi *bin* i *obj* (rysunek 7.6b).

**Rysunek 7.6.**  
Umieszczenie folderu *dane* w oknie *Solution Explorer* (a) i w folderze projektu (b)



W kolejnym kroku przejdź do pliku *MainWindow.xaml* i podmień w kodzie XAML puste znaczniki *Grid* na kod:

```

<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>
  <DataGrid x:Name="gridProdukty" AutoGenerateColumns="False" Margin="10"
    AlternatingRowBackground="Lavender" GridLinesVisibility="None"
    Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
    ItemsSource="{Binding Path=Elements[Produkt]}">
    <DataGrid.Columns>
      <DataGridTextColumn Header="Symbol"
        Binding="{Binding Element[Symbol].Value}" />
      <DataGridTextColumn Header="Nazwa"
        Binding="{Binding Element[Nazwa].Value}" />
      <DataGridTextColumn Header="Liczba sztuk"
        Binding="{Binding Element[LiczbaSztuk].Value}" />
    </DataGrid.Columns>
  </DataGrid>
</Grid>

```

<sup>3</sup> Klikając w oknie *Solution Explorer* nazwę projektu, można odczytać w oknie *Properties* ścieżkę do projektu (*Project Folder*).

```

        <DataGridComboBoxColumn x:Name="nazwaMagazynu" Header="Magazyn"
            SelectedItemBinding="{Binding Element[Magazyn].Value}"/>
    </DataGrid.Columns>
</DataGrid>
<Button Grid.Row="1" Grid.Column="1" Margin ="5" MinWidth="120"
    HorizontalAlignment="Right" Height="30" Content="Zapisz"
    Click="btnZapisz_Click"/>
</Grid>

```

W kodzie XAML definiowany jest panel `Grid` mający 2 wiersze i 2 kolumny. W pierwszym wierszu tego panelu umieszczona została kontrolka `DataGrid`. Dla właściwości `ItemsSource` przypisano wiązanie na podstawie elementów dokumentu XML — `Produkt`. Dla każdej kolumny kontrolki `DataGrid` określono wiązanie z wartością danego elementu pliku XML.

W pliku `MainWindow.xaml.cs` należy dodać dyrektywę `using`:

```

using System.Collections.ObjectModel;
using System.Xml.Linq;
using System.IO;

```

i podmienić kod klasy `MainWindow` na następujący:

```

public partial class MainWindow : Window
{
    private string plik1= @"..\..\dane\Produkty.xml"; // Plik źródłowy
    private string plik2 = @"..\..\dane\Produkty2.xml"; // Plik wynikowy
    private XElement wykazProduktow;
    public MainWindow()
    {
        InitializeComponent();
        PrzygotujWiazanie();
    }
    private void PrzygotujWiazanie()
    {
        if(File.Exists(plik1))
            wykazProduktow = XElement.Load(plik1); // Załadowanie danych z pliku
                                                    // źródłowego

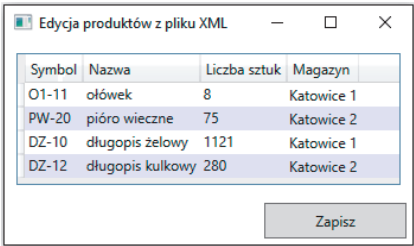
        gridProdukty.DataContext = wykazProduktow;
        ObservableCollection<string> ListaMagazynow =
            new ObservableCollection<string>() {"Katowice 1", "Katowice 2",
            "Gliwice 1"};
        nazwaMagazynu.ItemsSource = ListaMagazynow;
    }
    private void btnZapisz_Click(object sender, RoutedEventArgs e)
    {
        wykazProduktow.Save(plik2); // Zapisanie danych do pliku wynikowego
        MessageBox.Show("Pomyślnie zapisano dane do pliku");
    }
}

```

Kod programu jest krótki i dzięki temu przejrzysty, ale wymagałby dopracowania — brakuje tu między innymi obsługi wyjątków. Dane z pliku XML są ładowane do obiektu klasy `XElement` i obiekt ten jest wiązany z kontrolką `DataGrid`.

Program można uruchomić, a jego przykładowe działanie przedstawia rysunek 7.7. Możemy modyfikować dane o produktach i zapisać je w pliku *Produkty2.xml*. Niestety nie mamy tu dostępnej funkcji dodawania nowych pozycji i ich usuwania. Rozwiązanie takie jest wystarczające do przeglądu i prostej edycji danych w pliku XML, ale jeśli wymagane są operacje dodawania i usuwania rekordów, lepiej jest sięgnąć do rozwiązania z klasą opisującą dane (przedstawionego w poprzednim podrozdziale), uzupełnianego o serializację i deserializację pliku XML.

**Rysunek 7.7.**  
*Wykaz produktów  
załadowanych  
z pliku XML*



Symbol	Nazwa	Liczba sztuk	Magazyn
O1-11	ołówek	8	Katowice 1
PW-20	pióro wieczne	75	Katowice 2
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2

Zapisz

## 7.5 Zadania

Przedstawione zadania stanowią okazję do rozwijania umiejętności w zakresie wykorzystania kontrolki **DataGrid**. W pierwszym zadaniu należy dodać zdjęcia do nowej kolumny, w drugim wyświetlić szczegóły w dodatkowym wierszu, a w ostatnim dokonać grupowania danych prezentowanych w **DataGrid**.

### Zadanie 7.1

Do programu z podrozdziału 7.3, w którym kontrolka **DataGrid** była wiązana z kolekcją produktów, dodaj możliwość prezentowania zdjęć produktów w **DataGrid** (w osobnej kolumnie). Wykorzystaj element **Image**.

### Zadanie 7.2

Do programu z poprzedniego zadania dodaj nową funkcjonalność, umożliwiającą wyświetlenie szczegółów wybranego wiersza przy użyciu właściwości **RowDetailsTemplate**. Dodaj do klasy **Produkt** nową właściwość typu **String**, zawierającą opis produktu. Program ma wyświetlić jej zawartość dla wybranego produktu (z możliwością edycji) w dodatkowym wierszu. Ponadto w dodatkowym wierszu należy umieścić przycisk pozwalający dodać lub zmienić zdjęcie.

### Zadanie 7.3

Do programu z poprzedniego zadania (lub, jeśli wolisz, do zadania bazowego z podrozdziału 7.3) dodaj grupowanie wykazu według magazynu. Wykorzystaj właściwość **GroupDescriptions** interfejsu **ICollectionView**, która działa podobnie jak właściwość **SortDescriptions** (użyta do sortowania w programie opisanym w podrozdziale 6.3). Możemy do właściwości **GroupDescriptions** dodać dowolną liczbę obiektów **PropertyGroupDescription** i utworzyć grupy elementów kolekcji.

## 7.6 Wskazówki do zadań

Wskazówki zawierają szczegółowe wyjaśnienia i większe fragmenty rozwiązań.

### Wskazówki do zadania 7.1

Program wymaga dodania nowej właściwości do klasy `Produkt`:

```
public Uri Zdjecie { get; set; }
```

i uwzględnienia tej zmiany w konstruktorze. W kodzie XAML należy zdefiniować nową kolumnę dla zdjęcia:

```
<DataGridTemplateColumn Header="Zdjęcie" MaxWidth="50" IsReadOnly="True">
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <Image Source="{Binding Path=Zdjecie}" />
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>
```

W kodzie klasy `MainWindow` nie zostało dużo do zrobienia — wystarczy dodać ścieżkę podczas ładowania obiektów do kolekcji, na przykład:

```
ListaProduktow.Add(new Produkt("01-11", "ołówek", 8, "Katowice 1",
    new Uri(sciezka + "olowek.jpg")));
```

gdzie `sciezka` to łańcuch znakowy prezentujący ścieżkę do pliku. Jeżeli wgrasz pliki do osobnego folderu (np. `C:\temp`), wówczas ścieżkę można ustawić jako:

```
string sciezka = @"C:\temp\";
```

Jeżeli wolisz zintegrować pliki z projektem, dodając w oknie *Solution Explorer* do projektu nowy folder, na przykład o nazwie „rysunki” (*Add/New Folder*), a do tego folderu zdjęcia (*Add/Existing Item*), możesz użyć formatu Pack URI<sup>4</sup>. Po dodaniu folderu *rysunki* i plików z grafiką do projektu ich nazwy będą widoczne w oknie *Solution Explorer* (rysunek 7.8). W takim wariancie ścieżkę można zapisać jako:

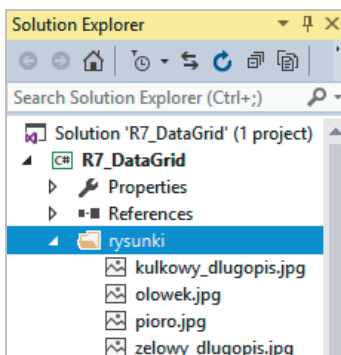
```
string sciezka = "pack://application:,,,/rysunki/"; // Ustalenie ścieżki z użyciem
//formatu Pack URI
```

Po wskazanych zmianach program jest gotowy do uruchomienia. Przykładowy wynik przedstawia rysunek 7.9.

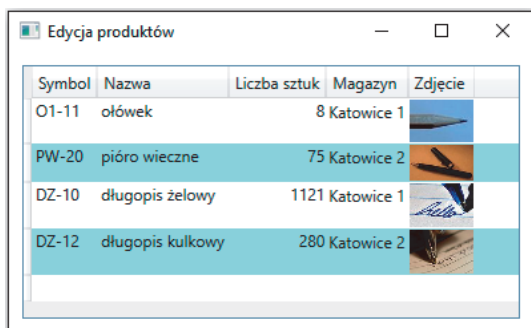
<sup>4</sup> Format Pack URI jest częścią specyfikacji *XML Paper Specification* (XPS). Format ten jest definiowany jako: `pack://URIPakietu/fragment_ścieżki`. Więcej na ten temat znajdziesz w: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 350 – 353); zob. także („Pack URIs in WPF”): [https://msdn.microsoft.com/en-us/library/aa970069\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/aa970069(v=vs.110).aspx).

**Rysunek 7.8.**

Okno Solution Explorer po dodaniu zdjęć do projektu

**Rysunek 7.9.**

Wykaz produktów z kolumną zawierającą zdjęcia produktów

**Wskazówki do zadania 7.2**

Dodaj do klasy `Produkt` nową właściwość `public string Opis { get; set; }`. Zmień odpowiednio konstruktor. W kodzie XAML zwiększ nieco rozmiary okna aplikacji. Na końcu definicji kontrolki `DataGrid` (przed znacznikiem zamykającym `</DataGrid>`) wpisz kod:

```
<DataGrid.RowDetailsTemplate>
  <DataTemplate>
    <StackPanel Orientation="Horizontal">
      <TextBlock Margin="10" Text="Opis produktu:"/>
      <TextBox Margin="10" Text="{Binding Path=Opis}"
        TextWrapping="Wrap" FontWeight="Bold"
        Foreground="BlueViolet" Width="150"/>
      <Button x:Name="btnZdjecie" Margin="10" Click="btnZdjecie_Click"
        Content="Dodaj lub zmień zdjęcie" Height="30"/>
    </StackPanel>
  </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

Właściwość `RowDetailsTemplate` pozwala prezentować dodatkowe dane o obiekcie. W naszym wierszu dodatkowym ma być zawarty opis produktu na podstawie nowej właściwości `Opis` oraz przycisk umożliwiający dodanie zdjęcia lub jego zmianę.

W *code-behind* w klasie `MainWindow` należy uwzględnić nową właściwość z opisem produktu podczas ładowania obiektów do kolekcji, na przykład:

```
ListaProduktow.Add(new Produkt("01-11", "ołówek", 8, "Katowice 1", new
    ↪Uri(sciezka + "olowek.jpg"), "Ołówek z gumką HB"));
```

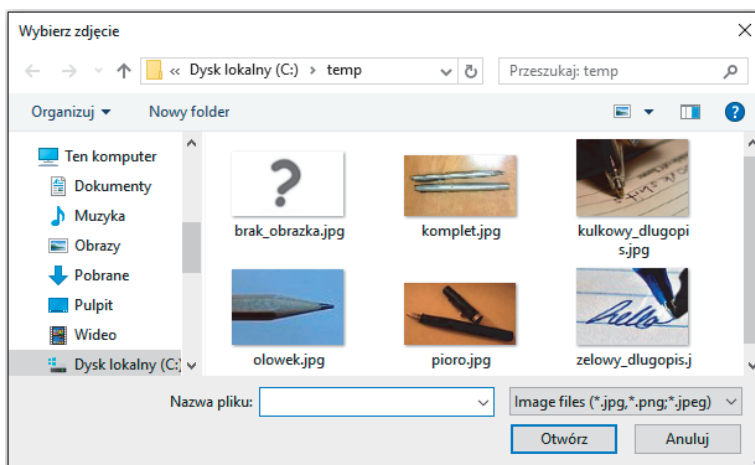
Należy także zaimplementować obsługę zdarzenia kliknięcia przycisku *Dodaj lub zmień zdjęcie*:

```
private void btnZdjecie_Click(object sender, RoutedEventArgs e)
{
    Microsoft.Win32.OpenFileDialog dialog = new Microsoft.Win32.OpenFileDialog();
    dialog.Title = "Wybierz zdjęcie";
    dialog.Filter = "Image files (*.jpg;*.png;*.jpeg)|*.jpg;*.png;*.jpeg|All
    ↪files (*.*)|*.*";
    dialog.InitialDirectory = @"C:\temp\";
    if (dialog.ShowDialog() == true)
    {
        (gridProdukty.SelectedItem as Produkt).Zdjecie = new Uri(dialog.FileName);
        gridProdukty.CommitEdit(DataGridEditingUnit.Cell, true);
        gridProdukty.CommitEdit();
        CollectionViewSource.GetDefaultView(gridProdukty.
        ↪ItemsSource).Refresh();
    }
}
```

Do wskazania zdjęcia wykorzystano klasę `OpenFileDialog`, przypisując dla obiektu tej klasy początkowy folder i określając filtr dla plików z grafiką (podstawowe formaty). Wywołanie metody `dialog.ShowDialog` spowoduje wyświetlenie okna podobnego jak na rysunku 7.10.

### Rysunek 7.10.

*Wybór pliku  
ze zdjęciem*

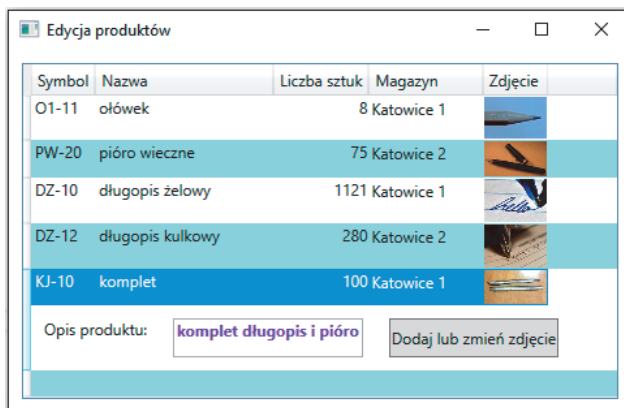


W prezentowanym kodzie przyjęto umiejscowienie plików w osobnym katalogu `C:\temp`. Jeśli użytkownik dokona wyboru pliku, ścieżka do niego (`dialog.FileName`) zostanie przypisana dla obiektu (produktu) prezentowanego przez `gridProdukty.SelectedItem`. W dalszej części kodu jest zatwierdzenie zmian i zakończenie trybu



edycji, a następnie odświeżenie widoku `DataGrid`. Rysunek 7.11 przedstawia działanie programu po opisanych zmianach. Na końcu wykazu znajduje się dodatkowy wiersz ze szczegółami dla nowego produktu.

**Rysunek 7.11.**  
Wykaz produktów  
z dodatkowym  
wierszem



## Wskazówki do zadania 7.3

Wstępne wskazówki zostały już podane w treści zadania. W *code-behind* na końcu metody `PrzygotujWiazanie` dodaj definicję widoku i kryterium grupowania dla tego widoku:

```
ICollectionView widok = CollectionViewSource.GetDefaultView(gridProdukty.  
    ↪ ItemsSource);  
widok.GroupDescriptions.Add(new PropertyGroupDescription("Magazyn"));
```

Pozostały już tylko zmiany w kodzie XAML, w którym trzeba określić sposób wyświetlania zgrupowanych danych. Zaprezentowane zostanie przykładowe proste rozwiązanie. Na końcu definicji kontrolki `DataGrid` (przed znacznikiem zamykającym `</DataGrid>`) dodaj kod:

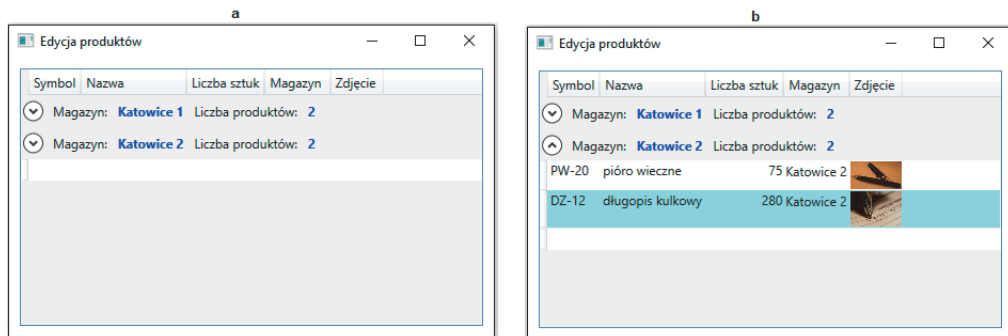
```
<DataGrid.GroupStyle>  
    <GroupStyle>  
        <GroupStyle.ContainerStyle>  
            <Style TargetType="{x:Type GroupItem}">  
                <Setter Property="Template">  
                    <Setter.Value>  
                        <ControlTemplate TargetType="{x:Type GroupItem}">  
                            <Expander>  
                                <Expander.Header>  
                                    <StackPanel Orientation="Horizontal">  
                                        <TextBlock Text="Magazyn:" Margin="5" />  
                                        <TextBlock Text="{Binding Path=Name}"  
                                            Margin="5"  
                                            FontWeight="Bold"  
                                            Foreground="Blue" />  
                                    <TextBlock Text="Liczba produktów:"  
                                        Margin="5"/>  
                                    <TextBlock Text="{Binding  
                                        Path=ItemCount}" />  
                                </Expander.Header>  
                            </Expander>  
                        </Setter.Value>  
                    </Setter>  
                </Style>  
            </GroupStyle.ContainerStyle>  
        </GroupStyle>  
    </DataGrid.GroupStyle>
```

```

        Margin="5" FontWeight="Bold"
        Foreground="Blue"/>
    </StackPanel>
</Expander.Header>
<ItemsPresenter/>
</Expander>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</GroupStyle.ContainerStyle>
</GroupStyle>
</DataGrid.GroupStyle>

```

W prezentowanej definicji pogrupowanego wykazu wykorzystano właściwość klasy `GroupStyle` o nazwie `ContainerStyle`, która pozwala ustalić styl dla tworzonej grupy (można jeszcze osobno określić szablon nagłówka przy użyciu właściwości `HeaderTemplate`). W definicji stylu została umieszczona definicja szablonu<sup>5</sup>, która zawiera `Expander` — kontrolkę służącą do prezentacji danych z możliwością zwiijania i rozwijania szczegółów. W nagłówku kontrolki `Expander` znajdują się cztery pola tekstowe, dwa z nich są związane z utworzonym widokiem dla grupowania, to znaczy nazwa grupy (w naszym programie jest to właściwość `Magazyn`) i liczba produktów w danej grupie. Zawartość kontrolki `Expander` wyświetlana jest za pomocą obiektu `ItemsPresenter`. Efekt działania programu możemy zobaczyć na przykładowych zrzutach ekranu (rysunek 7.12).



**Rysunek 7.12.** Pogrupowany wykaz produktów w wersji prezentującej obie grupy zwiinięte (a) i w wersji z jedną grupą rozwiniętą (b)

Można jeszcze dodać do tego programu jakąś kontrolkę, poprzez którą użytkownik mógłby włączać lub wyłączać grupowanie.

<sup>5</sup> Szczegóły na temat użytych tu konstrukcji (stylów i szablonów) zostaną przedstawione w dalszej części książki (w rozdziałach 10. i 11.).

Kontrolki `Expander` można oczywiście używać niezależnie od `DataGrid` i grupowania. Przetestuj w osobnym projekcie prostą definicję kontrolki `Expander`:

```
<Expander Header="Adres korespondencyjny">
  <StackPanel>
    <TextBlock Text="Miasto:"/>
    <TextBlock Text="Ulica:"/>
    <TextBlock Text="Nr domu:"/>
  </StackPanel>
</Expander>
```



## Rozdział 8.

# Menu — aplikacja Przeglądarka www

W tym rozdziale przedstawię kontrolkę `Menu`, która służy do definiowania menu programu. Napišemy bardzo prostą przeglądarkę stron www. W tworzonej aplikacji wykorzystamy także kontrolkę `ToolBar` i element `WebBrowser`.

## 8.1 Kod XAML

Naszym celem nie jest wykonanie profesjonalnej przeglądarki, lecz zapoznanie się z podstawowymi możliwościami kontrolki `Menu` na przykładzie implementacji prostej przeglądarki www. Nie dla wszystkich opcji tworzonego menu będziemy definiować działanie.

W kodzie XAML ustaw atrybut `Title="Prosta przeglądarka www"`. Zmień także wymiary okna aplikacji na przykładowo: `Height="550" Width="750"`. W oknie *XAML* podmień kod ze znacznikami `<Grid>` i `</Grid>` na następujący:

```
<Grid>
    <!-- Definicja wierszy i kolumn dla panelu Grid-->
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <!-- Menu programu-->
    <Menu Grid.Row="0" Grid.Column="0">
        <MenuItem Header="Plik">
            <MenuItem Header="Zapisz jako" Click="Zapisz_Click" />
            <MenuItem Header="Drukuj" Click="Tmp_Click" />
        </MenuItem>
    </Menu>
</Grid>
```

```

        <Separator/>
        <MenuItem Header="Wyjście" Click="Exit_Click"/>
    </MenuItem>
    <MenuItem Header="Narzędzia">
        <MenuItem Header="Ramka" IsChecked="True" IsCheckable="True"
            Checked="RamkaOn_Click" Unchecked="RamkaOff_Click"/>
        <Separator/>
        <MenuItem Header="Ustawienia" Click="Tmp_Click"/>
        <MenuItem Header="Rozmiar">
            <MenuItem Header="Zwiększ +" Click="Tmp_Click"/>
            <MenuItem Header="Zmniejsz -" Click="Tmp_Click"/>
        </MenuItem>
        <Separator/>
        <MenuItem Header="0 programie" Click="0Programie_Click"/>
    </MenuItem>
</Menu>

<!-- Przeglądarka -->
<DockPanel Grid.Row="1" Grid.Column="0">
    <ToolBar DockPanel.Dock="Top">
        <Button x:Name="btnWstecz" Content="Wstecz" Height="25"
            Background="LightSteelBlue" HorizontalAlignment="Left"
            VerticalAlignment="Top" Width="55" Click="btnWstecz_Click"/>
        <Button x:Name="btnDalej" Content="Dalej" Height="25"
            Background="LightSteelBlue" HorizontalAlignment="Left"
            VerticalAlignment="Top" Width="55" Click="btnDalej_Click"/>
        <Separator/>
        <TextBox x:Name="txtAdres" HorizontalAlignment="Left" Height="25"
            TextWrapping="Wrap" Text="http://" KeyUp="txtAdres_KeyUp"
            VerticalAlignment="Top" MinWidth="400"/>
        <Button x:Name="btnWejdz" Content="Wejdz" Height="25"
            Background="LightSteelBlue" HorizontalAlignment="Left"
            VerticalAlignment="Top" Width="50" Click="btnWejdz_Click"/>
    </ToolBar>
    <Border x:Name="brdRamka" BorderThickness="3" BorderBrush="DarkCyan">
        <WebBrowser x:Name="wbPrzegladarka"
            Navigating="wbPrzegladarka_Navigating" VerticalAlignment="Top"
            Navigated="wbPrzegladarka_Navigated" HorizontalAlignment="Left"/>
    </Border>
</DockPanel>
</Grid>

```

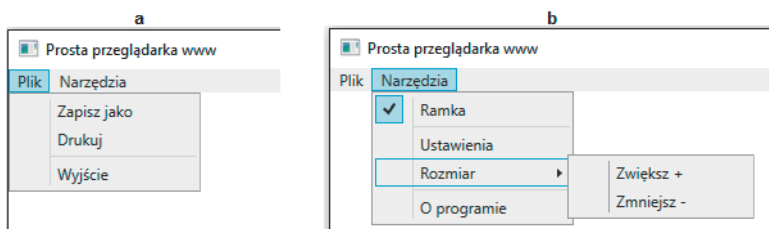
Cały panel `Grid` został podzielony komentarzami na trzy części. W pierwszej części znajduje się definicja panelu, który składa się z dwóch wierszy i jednej kolumny. W dwóch pozostałych częściach pojawiają się nowe kontrolki, dlatego poświęcimy im więcej miejsca.

## Menu programu

W kolejnej części znajduje się kontrolka `Menu`. Poszczególne opcje i podopcje umieszczane są za pomocą elementu `MenuItem`. Dla opcji (takich jak *Plik*), które nie wykonują żadnej operacji, lecz jedynie służą do rozwinięcia podopcji, ustawiamy tytuł (`Header`).

Natomiast dla pozostałych opcji oprócz tytułu określone zostało zdarzenie uruchomienia opcji (**Click**). Dodatkowo w niektórych miejscach struktury menu widzimy element **Separator**, który powoduje tu wyświetlenie poziomej kreski oddzielającej grupy opcji (dla lepszej czytelności struktury **Menu**). Zanim dokończymy omawiać kontrolkę **Menu**, spójrzmy na rysunek 8.1, przedstawiający wygląd zdefiniowanego **Menu** w działaniu, osobno dla rozwiniętej opcji **Plik** i osobno dla opcji **Narzędzia**.

**Rysunek 8.1.**  
Menu programu  
— opcja **Plik** (a)  
i opcja  
**Narzędzia** (b)



W programie są dwie opcje główne: **Plik** i **Narzędzia**. Jak widzimy w kodzie XAML, są one umieszczane na tym samym poziomie bezpośrednio wewnątrz kontrolki **Menu**. Opcja **Plik** zawiera trzy podopcje, a ostatnia z nich (**Wyjście**) jest oddzielona poziomą linią (separator). Opcja **Narzędzia** ma cztery podopcje, z czego jedna, **Rozmiar**, ma dwie kolejne podopcje. W kodzie XAML efekt ten uzyskujemy poprzez odpowiednie zagnieżdżanie znaczników **MenuItem**. Zwróćmy uwagę na opcję **Ramka** — z lewej strony jej nazwy widzimy zaznaczoną kontrolkę podobną do kontrolki **CheckBox**. Efekt ten został uzyskany za pomocą właściwości **IsCheckable**. Ponieważ zakładamy, że domyślnie ma się wyświetlać ramka, dodatkowo w kodzie XAML właściwość **IsChecked** została ustawiona na **True**. Atrybuty **Checked** i **Unchecked** umożliwiają przypisanie nazw metod obsługujących odpowiednio zdarzenie zaznaczenia i odznaczenia opcji.

## Przeglądarka

Przeglądarka została zdefiniowana w panelu **DockPanel1**. Na samym początku panelu znajduje się kontrolka narzędziowa **ToolBar**, służąca do grupowania przycisków (lub innych elementów). Zazwyczaj są to graficzne przyciski, ale niekoniecznie. W naszym programie są to przyciski **Button** z tekstowym opisem i jedno pole tekstowe. Pierwsze dwa przyciski, **Wstecz** i **Dalej**, będą umożliwiać poruszanie się po stronach www. W polu tekstowym będzie można wpisywać adres strony internetowej. Natomiast kliknięcie przycisku **Wejdz** spowoduje otwarcie wskazanej strony. W definicji kontrolki **ToolBar** także można użyć separatora, przy czym w tym przypadku **separator** pojawi się jako pionowa kreska.

W dalszej części kodu widzimy kontrolkę **WebBrowser** umieszczoną wewnątrz ramki (**Border**). Ta ramka została tu dodana nie tyle dla ozdoby, co po to, aby umożliwić proste przetestowanie właściwości **IsCheckable** ustawionej na **True** dla jednej z opcji naszego menu (**Ramka**). Dla kontrolki **WebBrowser** przypisano nazwy metod obsługujących dwa zdarzenia: **Navigating** i **Navigated**. Obsługa pierwszego zdarzenia pozwoli uaktualnić pole tekstowe z adresem po tym, jak użytkownik kliknie jakiś link na stronie. Obsługa drugiego zdarzenia zostanie wykorzystana do ukrycia błędów JavaScriptu (bez tego przeglądanie wielu popularnych stron wymagałoby uciążliwego potwierdzania komunikatów o błędach).

## 8.2 Code-behind

W pliku *MainWindow.xaml.cs* dodaj dyrektywy `using` dla dwóch przestrzeni nazw, jakie będą nam tu potrzebne<sup>1</sup>:

```
using System.Reflection;
using System.IO;
```

W klasie `MainWindow` należy dopisać (pod konstruktorem klasy) nowe metody. Metod tym razem będzie nieco więcej. Zostaną one zaprezentowane w trzech częściach.

### Metody obsługi zdarzeń kliknięcia opcji Menu

W pierwszej kolejności zostaną zaprezentowane metody obsługi wyboru opcji w menu (poprzez kliknięcie):

```
// Metody obsługi zdarzeń kliknięcia opcji Menu
private void RamkaOn_Click(object sender, RoutedEventArgs e) // Włączenie ramki
{
    if(brdRamka != null)
        brdRamka.BorderThickness= new Thickness(3);
}
private void RamkaOff_Click(object sender, RoutedEventArgs e) // Wylączenie ramki
{
    if (brdRamka != null)
        brdRamka.BorderThickness = new Thickness(0);
}
private void Zapisz_Click(object sender, RoutedEventArgs e) // Zapisanie strony
// do pliku
{
    Microsoft.Win32.SaveFileDialog dialog = new Microsoft.Win32.SaveFileDialog();
    dialog.Filter = "WebPage|*.html";
    dialog.DefaultExt = ".html";
    dynamic doc = wbPrzegladarka.Document;
    if (doc != null)
    {
        var htmlText = doc.documentElement.InnerHtml;
        if (dialog.ShowDialog() == true && htmlText != null)
        {
            File.WriteAllText(dialog.FileName, htmlText); // File wymaga
// using System.IO;
        }
    }
}
private void Tmp_Click(object sender, RoutedEventArgs e) // Tymczasowa metoda
// dla niegotowych opcji
{
    MessageBox.Show("Opcja w budowie");
}
```

<sup>1</sup> Te dwie przestrzenie nazw należy dopisać oprócz automatycznie umieszczonych. Łącznie w tej klasie będą wykorzystane następujące przestrzenie nazw: `System.Windows`, `System.Windows.Controls`, `System.Windows.Input`, `System.Windows.Navigation`, `System.Reflection`, `System.IO`.



```

    }
    private void OProgramie_Click(object sender, RoutedEventArgs e) // Informacje
                                                                    // o programie
    {
        MessageBox.Show("Prosta przeglądarka www, Wersja 1.0, Helion 2017");
    }
    private void Exit_Click(object sender, RoutedEventArgs e) // Wyjście (zamknięcie
                                                            // okna aplikacji)
    {
        Close();
    }

```

Dwie pierwsze metody, `RamkaOn_Click` i `RamkaOff_Click`, są uruchamiane w chwili zaznaczenia lub odznaczenia opcji *Ramka*. W ciele tych metod ustawiana jest grubość obwódki na wartość 3 (włączenie ramki) lub 0 (wyłączenie ramki). Metoda `Zapisz_Click` jest podpięta do opcji *Zapisz* i dokonuje zapisu strony do formatu HTML. Jest to jednak tylko „surowa” kopia kodu HTML jednej podstrony. Metoda `Tmp_Click` jest uruchamiana po wyborze opcji, które są niegotowe w obecnej wersji programu: *Drukuj*, *Ustawienia*, *Rozmiar/Zwiększ* oraz *Rozmiar/Zmniejsz*. Metoda powoduje wyświetlenie komunikatu „Opcja w budowie”. Dwie ostatnie metody nie wymagają specjalnego komentarza: jedna z nich wyświetla informacje o programie, a druga zamyka okno aplikacji.

## Metody obsługi zdarzeń dla kontroltek umieszczonych w ToolBar

Następnie należy umieścić metody obsługujące zdarzenia dla kontroltek znajdujących się na pasku narzędziowym (`ToolBar`):

```

// Metody obsługi zdarzeń dla kontroltek umieszczonych w ToolBar
private void btnWejdz_Click(object sender, RoutedEventArgs e)
{
    wbPrzegladarka.Navigate(txtAdres.Text);
}
private void btnWstecz_Click(object sender, RoutedEventArgs e)
{
    if (wbPrzegladarka.CanGoBack)
        wbPrzegladarka.GoBack();
}
private void btnDalej_Click(object sender, RoutedEventArgs e)
{
    if (wbPrzegladarka.CanGoForward)
        wbPrzegladarka.GoForward();
}
private void txtAdres_KeyUp(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
        wbPrzegladarka.Navigate(txtAdres.Text);
}

```

Metoda `btnWejdz_Click` obsługuje zdarzenie kliknięcia przycisku *Wejdz*, czyli realizuje przejście na wskazaną stronę. W tym celu została użyta metoda `Navigate` z klasy `WebBrowser`. W omawianym rozwiązaniu została wykorzystana wersja metody `Navigate` przyjmująca jeden argument `string`. Ale są jeszcze inne przeładowane warianty, w tym

przyjmujący jeden argument typu `Uri`. Ponieważ nie mamy tu walidacji danych, ewentualne pomyłki składniowe w adresie (użycie niedozwolonych znaków) spowodują błąd programu. Pracując samodzielnie nad dalszym rozwojem tej aplikacji, możesz wykorzystać wersję metody `Navigate` z argumentem typu `Uri`.

Metody `btnWstecz_Click` i `btnDalej_Click` realizują przechodzenie do poprzedniej/kolejnej strony. W ich kodzie zostały użyte właściwości pozwalające sprawdzić, czy dana operacja jest możliwa, a jeśli tak, to wywoływana jest odpowiednia metoda z klasy `WebBrowser` dokonująca przejścia.

Ostatnia w tej grupie metoda, `txtAdres_KeyUp`, pozwoli obsłużyć wejście na daną stronę także wówczas, gdy użytkownik potwierdzi klawiszem `Enter` wpisany adres w polu tekstowym.

## Metody obsługi zdarzeń dla kontrolki `WebBrowser`

W dalszej części klasy `MainWindow` należy dołączyć kod metod obsługujących zdarzenia dla kontrolki `WebBrowser`:

```
// Metody obsługi zdarzeń dla kontrolki WebBrowser (Navigating i Navigated)
private void wbPrzegladarka_Navigating(object sender,
    System.Windows.Navigation.NavigatingCancelEventArgs e)
{
    txtAdres.Text = e.Uri.OriginalString;    // Aktualizacja pola tekstowego z adresem
}
private void wbPrzegladarka_Navigated(object sender, NavigationEventArgs e)
{
    HideScriptErrors(wbPrzegladarka, true);    // Wywołanie metody ukrywającej błędy
                                                // JavaScriptu
}
public void HideScriptErrors(WebBrowser wb, bool Hide)
{
    // Ukrycie błędów JavaScriptu, rozwiązanie ze strony MSDN "Suppress Script Errors in
    // Windows.Controls.Webbrowser"
    // Typ wyliczeniowy BindingFlags wymaga przestrzeni nazw using System.Reflection;
    dynamic activeX = this.wbPrzegladarka.GetType().
        InvokeMember("ActiveXInstance",
            BindingFlags.GetProperty | BindingFlags.Instance |
            BindingFlags.NonPublic,
            null, this.wbPrzegladarka, new object[] { });
    activeX.Silent = true;
}
```

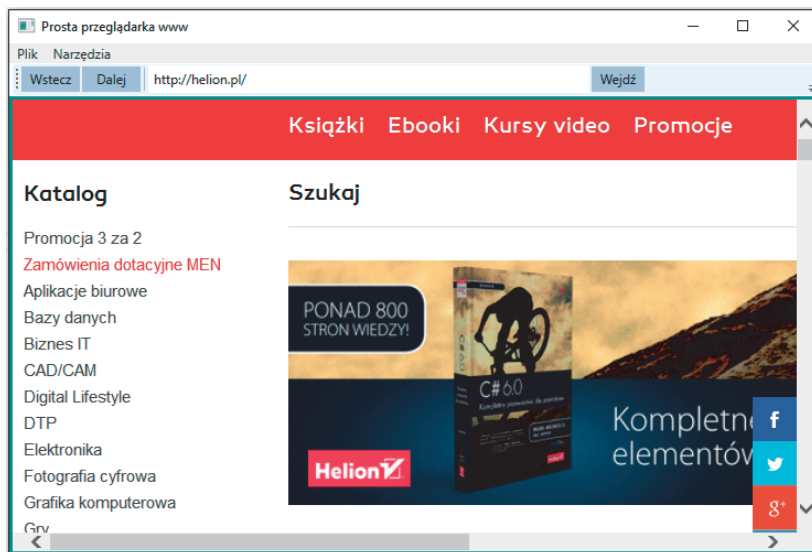
Zdarzenie `Navigating` ma miejsce tuż przed wejściem na daną stronę. Dzięki obsłudze tego zdarzenia będziemy mogli dokonać aktualizacji pola tekstowego po zmianie strony www, to znaczy w chwili, gdy użytkownik kliknie jakiś link, chcąc wejść na podstronę danego serwisu, w polu tekstowym pojawi się adres tej podstrony.

Zdarzenie `Navigated` występuje w chwili, gdy jest pobierana (ładowana) określona strona. Obsługa tego zdarzenia została w naszym programie wykorzystana do ukrycia błędów JavaScriptu. To pozwoli nam testować naszą przeglądarkę bez uciążliwych komunikatów o błędach JavaScriptu zgłaszanych na wielu popularnych stronach www.

Zdefiniowana w programie metoda `HideScriptErrors` dokonuje ukrycia błędów JavaScriptu<sup>2</sup>.

Program można już uruchomić. Rysunek 8.2. przedstawia jego przykładowe działanie. W pasku adresu został wpisany adres strony *helion.pl* i po kliknięciu przycisku *Wejdź* (lub po naciśnięciu klawisza *Enter*) pojawia się wskazana strona.

**Rysunek 8.2.**  
*Działanie prostej  
przeglądarki  
stron www*



Przeglądarka jest bardzo prosta, w działaniu nie jest idealna i daleko jej do profesjonalnych rozwiązań, ale naszym celem przy tworzeniu tego programu było przede wszystkim zapoznanie się z możliwościami kontrolki `Menu`. Obie główne opcje zdefiniowanego `Menu` są widoczne na rysunku 8.2 w postaci zwiniętej, a zatem dopiero kliknięcie opcji *Plik* lub *Narzędzia* pozwoli rozwinąć jej podopcje, co zostało już zaprezentowane na rysunku 8.1.

## 8.3 Zadania

Dwa pierwsze zadania pozwolą Czytelnikowi poznać kolejne możliwości w zakresie tworzenia menu (w tym menu kontekstowego), a ostatnie będzie okazją do poznania kontrolki `TreeView`, definiowanej podobnie jak `Menu`.

<sup>2</sup> Rozwiązanie pochodzi z biblioteki MSDN. Strona ta ma wyjątkowo długi adres, więc wygodniej będzie ją odnaleźć, wpisując w wyszukiwarce tekst „msdn” i tytuł artykułu: „Suppress Script Errors in Windows.Controls.Webbrowser”. Zainteresowane osoby odsyłam do opisu składni metody `InvokeMember`, która została wykorzystana w tym rozwiązaniu („InvokeMember Method”): [https://msdn.microsoft.com/en-us/library/66btctbe\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/66btctbe(v=vs.110).aspx).

## Zadanie 8.1

W programie z przeglądarką www zdefiniuj własne menu kontekstowe (`ContextMenu`) dla pola tekstowego z adresem. W menu, oprócz standardowych opcji dla kontrolki `TextBox` (*Wytnij*, *Kopiuj* i *Wklej*), ma się znaleźć pod separatorem opcja *Dodaj do ulubionych* z dwiema podopcjami: *Do folderu* i *Do paska ulubionych*. Dla obu podopcji możesz przypisać metodę `Tmp_Click` wyświetlającą komunikat „Opcja w budowie”.

## Zadanie 8.2

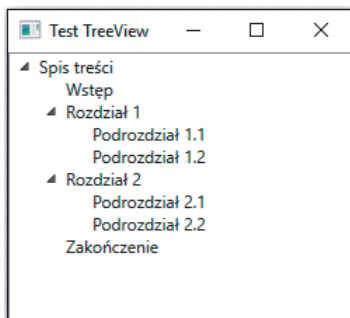
W programie z przeglądarką www na przykładzie jednej z opcji (lub kilku różnych) przetestuj właściwość `Icon` klasy `MenuItem`, pozwalającą wyświetlić obok nazwy opcji mały obrazek. Ponadto za pomocą kontrolki `ToolTip` dodaj do wybranych opcji dodatkowe opisy pojawiające się po najechaniu myszą na nazwę opcji.

## Zadanie 8.3

Wykonaj w osobnym projekcie test kontrolki `TreeView`, służącej do prezentowania danych w sposób hierarchiczny. Tworzenie struktury kontrolki `TreeView` odbywa się podobnie jak w przypadku `Menu`, przy czym używa się w tym celu elementu `TreeViewItem` (zamiast `MenuItem`). Ponadto wykorzystaj dostępną dla elementu `TreeViewItem` właściwość `IsExpanded`, określającą, czy elementy podrzędne mają być rozwinięte. Ustaw tę właściwość na `True`, tak aby po wyświetleniu zawartości kontrolki zostało rozwinięte całe drzewo (rysunek 8.3).

**Rysunek 8.3.**

Test kontrolki  
`TreeView`



# 8.4 Wskazówki do zadań

Wskazówki zawierają szczegółowe wyjaśnienia i większe fragmenty rozwiązań.

## Wskazówki do zadania 8.1

Kontrolka `TextBox` ma domyślne menu kontekstowe, którego działanie możesz sprawdzić, klikając prawym klawiszem myszy w jej obszarze. Stworzymy nowe kontekstowe menu, w którym będą te same opcje, jakie zawiera menu domyślne, a ponadto nowa

opcja z dwiema podopcjami. Do tworzenia menu kontekstowego służy kontrolka `ContextMenu` → `Menu`, która działa podobnie jak `Menu` — tu także konstruujemy strukturę menu poprzez elementy `MenuItem` i `Separator`. Przykładowe rozwiązanie może mieć następującą postać:

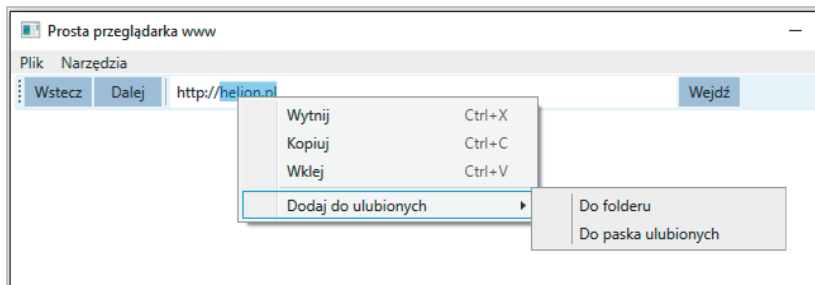
```
<TextBox x:Name="txtAdres" HorizontalAlignment="Left" Height="25"
    TextWrapping="Wrap" Text="http://" KeyUp="txtAdres_KeyUp"
    VerticalAlignment="Top" MinWidth="400">
  <TextBox.ContextMenu>
    <ContextMenu>
      <MenuItem Command="ApplicationCommands.Cut"/>
      <MenuItem Command="ApplicationCommands.Copy"/>
      <MenuItem Command="ApplicationCommands.Paste"/>
      <Separator/>
      <MenuItem Header="Dodaj do ulubionych">
        <MenuItem Header="Do folderu" Click="Tmp_Click"/>
        <MenuItem Header="Do paska ulubionych" Click="Tmp_Click"/>
      </MenuItem>
    </ContextMenu>
  </TextBox.ContextMenu>
</TextBox>
```

Kontrolka `MenuItem` pozwala obsłużyć kilka zdarzeń, w tym `Click`. Przypisanie zdarzenia `Click` nie jest jedynym sposobem dołączenia akcji do `MenuItem`, można w tym celu również wykorzystać polecenia (`Command`). Tak jest w prezentowanym kodzie dla trzech pierwszych opcji, gdzie właściwości `Command` zostały przypisane odpowiednie właściwości klasy `ApplicationCommands`<sup>3</sup>. W tym przypadku właściwość `Header` została automatycznie wypełniona. I tak przykładowo dla polecenia `ApplicationCommands.Cut` nagłówek w języku polskim przyjmuje wartość „Wytnij”. Można zmienić tekst nagłówka, wpisując swój własny tekst, na przykład:

```
<MenuItem Header="Wytnij tekst" Command="ApplicationCommands.Cut"/>
```

Pozostałe opcje menu zostały zdefiniowane podobnie jak w przypadku menu głównego. Rysunek 8.4 przedstawia menu kontekstowe w działaniu.

**Rysunek 8.4.**  
Działanie menu  
kontekstowego dla  
pola tekstowego



<sup>3</sup> Klasa `ApplicationCommands` poprzez statyczne właściwości udostępnia zestaw predefiniowanych poleceń. Opis klasy („ApplicationCommands Class”): [https://msdn.microsoft.com/en-us/library/system.windows.input.applicationcommands\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.input.applicationcommands(v=vs.110).aspx).

## Wskazówki do zadania 8.2

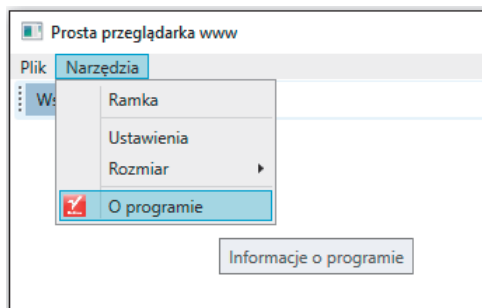
Dodanie rysunku do pozycji menu oraz kontrolki `ToolTip` nie wymaga specjalnych objaśnień. Przykładowe rozwiązanie dla opcji *O programie* może wyglądać tak:

```
<MenuItem Header="O programie" Click="OProgramie_Click">
    <MenuItem.Icon>
        <Image Source="Rysunki/logo.jpg" MaxHeight="20" MaxWidth="20"/>
    </MenuItem.Icon>
    <MenuItem.ToolTip>
        Informacje o programie
    </MenuItem.ToolTip>
</MenuItem>
```

Wygląd menu po zmianach prezentuje rysunek 8.5.

**Rysunek 8.5.**

*Menu po dodaniu  
ikonki dla jednej  
z opcji i opisu  
ToolTip*



## Wskazówki do zadania 8.3

Kontrolkę `TreeView` definiuje się podobnie jak `Menu`. Oto przykładowe rozwiązanie zadania:

```
<TreeView>
    <TreeViewItem Header="Spis treści" IsExpanded="True">
        <TreeViewItem Header="Wstęp" />
        <TreeViewItem Header="Rozdział 1" IsExpanded="True">
            <TreeViewItem Header="Podrozdział 1.1" />
            <TreeViewItem Header="Podrozdział 1.2" />
        </TreeViewItem>
        <TreeViewItem Header="Rozdział 2" IsExpanded="True">
            <TreeViewItem Header="Podrozdział 2.1" />
            <TreeViewItem Header="Podrozdział 2.2" />
        </TreeViewItem>
        <TreeViewItem Header="Zakończenie" />
    </TreeViewItem>
</TreeView>
```

Element `TreeViewItem` ma także właściwość `IsSelected`, określającą, czy dany element został wybrany. W elementach tej kontrolki można umieszczać dowolne inne obiekty. W przypadku kontrolki `TreeView` nierzadko zachodzi konieczność zdefiniowania jej dynamicznie i wówczas korzysta się z klasy `HierarchicalDataTemplate` i wiązania z hierarchiczną kolekcją.

## Rozdział 9.

# Zakładki (TabControl) — aplikacja Odtwarzacz audio

W tym rozdziale wykorzystamy kontrolkę `TabControl`, która pozwala podzielić interfejs na kilka różnych stron zwanych zakładkami. Nagłówki zakładek zazwyczaj umieszczane są w górnej części kontrolki. W wielu popularnych programach zakładki wykorzystywane są w opcjach narzędziowych lub konfiguracyjnych. W prostych programach zakładki mogą pełnić rolę menu i taki właśnie program wykonamy w tym rozdziale. Napišemy prosty odtwarzacz audio.

## 9.1 Kod XAML

W kodzie XAML ustaw atrybut `Title="Odtwarzacz"`. Zmień także wymiary okna aplikacji na przykład na: `Height="300" Width="650"`. W oknie XAML podmień kod ze znacznikami `<Grid>` na następujący:

```
<Grid>
    <TabControl>
        <TabItem Header="Audio">
            <StackPanel Margin="10">
                <WrapPanel>
                    <Button x:Name="btnWybierz" Click="btnWybierz_Click"
                        Content="Wybierz utwór" Padding="5" Margin="20,10" />
                    <TextBlock x:Name="txtUtwor" Text="" MinWidth="150"
                        Margin="10" />
                    <TextBlock x:Name="txtCzas" Text="" Foreground="Blue"
                        Margin="10"/>
                </WrapPanel>
            </StackPanel>
        </TabItem>
    </TabControl>
    <ProgressBar x:Name="pbGra" Margin="20" Height="20"/>
</Grid>
```

```

        <TextBlock Text="{Binding ElementName=pbGra, Path=Value,
            StringFormat={}{0:0}%" HorizontalAlignment="Center"
            VerticalAlignment="Center" />
    </Grid>
    <WrapPanel HorizontalAlignment="Center">
        <Button x:Name="btnPlay" Click="btnPlay_Click"
            Content="Play" Margin="10" Padding="5" Width="50"
            ToolTip="Odtwórz" IsEnabled="False" />
        <Button x:Name="btnPause" Click="btnPause_Click"
            Content="Pause" Margin="10" Padding="5" Width="50"
            ToolTip="Przerwij" IsEnabled="False" />
        <Button x:Name="btnStop" Click="btnStop_Click"
            Content="Stop" Margin="10" Padding="5" Width="50"
            ToolTip="Zakończ" IsEnabled="False" />
    </WrapPanel>
</StackPanel>
</TabItem>
<TabItem Header="Ustawienia">
    <StackPanel>
        <Label x:Name="lblKolor" Content="Wybierz kolor" />
        <RadioButton Content="Niebieski" Checked="radio_Checked"
            IsChecked="True" />
        <RadioButton Content="Zielony" Checked="radio_Checked" />
    </StackPanel>
</TabItem>
</TabControl>
</Grid>

```

Zakładki w kontrolce `TabControl` są domyślnie umieszczane na górze, ale można to zmienić przy użyciu właściwości `TabStripPlacement`, na przykład definicja `<TabControl TabStripPlacement="Bottom">` spowoduje umieszczenie zakładek na dole (można jeszcze użyć wariantu `Left` i `Right`).

Przedstawiony kod XAML zawiera dwie zakładki: pierwsza ma nagłówek *Audio*, a druga *Ustawienia*. Zakładki zadeklarowane są tu jawnie z użyciem elementu `TabItem`.

W zakładce *Audio* w panelu `StackPanel` umieszczone są trzy inne panele. W pierwszym (`WrapPanel`) znajduje się przycisk *Wybierz utwór*, który pozwoli użytkownikowi wybrać plik z nagraniem dźwiękowym. Oprócz przycisku są tu jeszcze dwa bloki tekstowe: jeden będzie zawierał nazwę pliku z wybranym utworem, a drugi — aktualny czas odtwarzania utworu.

W kolejnym panelu, w pojedynczej komórce `Grid`, znajdują się dwa elementy `Progress` ➔ `Bar`, czyli pasek postępu oraz blok tekstu, którego zawartość jest wiązana z kontrolką `ProgressBar`. Tekst ten zostanie umieszczony na pasku postępu i określać będzie aktualny procent odtworzenia utworu.

W ostatnim panelu tej zakładki znajdują się trzy przyciski umożliwiające typowe operacje wykonywane w odtwarzaczach audio: włączenie odtwarzania (*Play*), przerwanie odtwarzania (*Pause*) oraz zakończenie odtwarzania (*Stop*). Przyciski te są domyślnie niedostępne (poprzez ustawienie właściwości `IsEnabled` na `False`). Aktywowane są po wybraniu utworu.



W zakładce *Ustawienia* są tylko dwa przyciski `RadioButton` (wraz z etykietą), które pozwalają zmienić kolor paska postępu.

## 9.2 Code-behind

W pliku *MainWindow.xaml.cs* należy dopisać dyrektywy `using` dla dwóch przestrzeni nazw, jakie tu wykorzystamy<sup>1</sup>:

```
using Microsoft.Win32;
using System.Windows.Threading;
```

Kod klasy `MainWindow` podmień na następujący:

```
public partial class MainWindow : Window
{
    private MediaPlayer mediaPlayer = new MediaPlayer();
    private DispatcherTimer timer;

    public MainWindow()
    {
        InitializeComponent();

        timer = new DispatcherTimer();
        timer.Interval = TimeSpan.FromMilliseconds(500);
        timer.Tick += new EventHandler(timerTick);
    }

    void timerTick(object sender, EventArgs e)
    {
        if (mediaPlayer.Source != null && mediaPlayer.NaturalDuration.HasTimeSpan)
        {
            txtCzas.Text = mediaPlayer.Position.ToString(@"mm\:ss");
            // Ustawienia dla ProgressBar
            TimeSpan ts = mediaPlayer.NaturalDuration.TimeSpan;
            pbGra.Maximum = 100;
            pbGra.Value = ((double) mediaPlayer.Position.TotalMilliseconds /
                ↳ts.TotalMilliseconds)*100;
        }
    }

    private void btnWybierz_Click(object sender, RoutedEventArgs e)
    {
        OpenFileDialog dialog = new OpenFileDialog();
        dialog.Filter = "MP3 files (*.mp3)|*.mp3|All files (*.*)|*.*";
        if (dialog.ShowDialog() == true)
        {
            mediaPlayer.Open(new Uri(dialog.FileName));
            txtUtwor.Text = String.Format("Utwór: {0}", dialog.FileName);
            btnPlay.IsEnabled = true;
        }
    }
}
```

<sup>1</sup> Wymienione przestrzenie nazw należy dopisać oprócz automatycznie umieszczonych. Łącznie w tej klasie będą wykorzystane następujące przestrzenie nazw: `System`, `System.Windows`, `System.Windows.Controls`, `System.Windows.Media`, `Microsoft.Win32`, `System.Windows.Threading`.

```

        btnPause.IsEnabled = true;
        btnStop.IsEnabled = true;
        timer.Start();
    }
}

private void btnPlay_Click(object sender, RoutedEventArgs e)
{
    mediaPlayer.Play();
}
private void btnPause_Click(object sender, RoutedEventArgs e)
{
    mediaPlayer.Pause();
}
private void btnStop_Click(object sender, RoutedEventArgs e)
{
    mediaPlayer.Stop();
}
private void radio_Checked(object sender, RoutedEventArgs e)
{
    var radio = sender as RadioButton;
    string kolor = (radio.Content.ToString() == "Niebieski") ?
        ↪ "LightSkyBlue" : "LightGreen";
    pbGra.Foreground = (SolidColorBrush)new
        ↪ BrushConverter().ConvertFromString(kolor);
}
}

```

Na początku klasy znajduje się definicja obiektu klasy `MediaPlayer`, którego metody pozwolą odtwarzać pliki audio. Kolejnym polem deklarowanym w tej klasie jest `timer` typu `DispatcherTimer`. Klasa ta umożliwia odmierzenie czasu w osobnym wątku, bez zakłócania interfejsu użytkownika. Spójrzmy na schemat kodu:

```

DispatcherTimer timer = new DispatcherTimer();    // Definicja obiektu klasy
                                                    // DispatcherTimer
timer.Interval = TimeSpan.FromMilliseconds(500); // Ustalenie interwału
timer.Tick += new EventHandler(timerTick);        // Zdarzenie
timer.Start();                                    // Uruchomienie wątku

private void timerTick(object sender, EventArgs e)
{
    // Tutaj jest kod wykonywany co 500 milisekund (0,5 sekundy)
}

```

Po utworzeniu obiektu klasy `DispatcherTimer` definiowana jest właściwość `Interval`, określająca, co jaki czas ma być wywoływana metoda (akcja) wskazana w zdarzeniu `Tick`. Metoda `Start` uruchamia przygotowany wątek do pracy. W naszym programie jest wywoływana po wyborze utworu, ale można ją wywołać podobnie jak w powyższym schemacie, czyli zaraz po zdefiniowaniu obiektu. Metoda wskazana dla zdarzenia, nazwana tu `timerTick`, zawiera kod, który będzie wywoływany zgodnie z określonym interwałem.

Spójrzmy zatem teraz na kod metody `timerTick`, która będzie w naszym programie wywoływana co pół sekundy. Aktualizowana jest tam zawartość kilku właściwości, w tym dla tekstu zawierającego aktualny czas odtwarzania utworu oraz aktualną pozycję na odtwarzanej ścieżce (dla `ProgressBar`) prezentowaną procentowo. Informację o całkowitym czasie trwania utworu zawiera właściwość `NaturalDuration`.

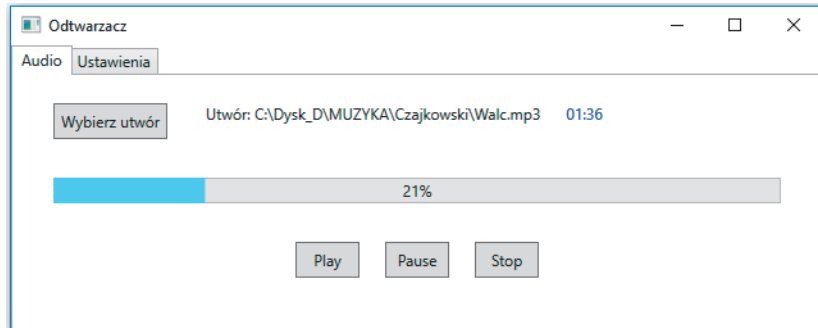
W metodzie `btnWybierz_Click` realizowany jest wybór utworu przy użyciu klasy `OpenFileDialog` (z przestrzeni nazw `Microsoft.Win32`) omawianej w podrozdziale 7.6. Po pomyślnym wyborze pliku wywoływana jest metoda `Open`, która otwiera wskazany plik dźwiękowy. Nazwa pliku zostaje przypisana do właściwości `Text` bloku tekstowego i zostają odblokowane przyciski odtwarzacza. W tym miejscu także jest uruchamiany wątek `timer`.

Trzy kolejne metody obsługujące zdarzenia kliknięcia przycisków (`btnPlay_Click`, `btnPause_Click` oraz `btnStop_Click`) wywołują odpowiednie metody klasy `Media Player` odpowiedzialne za uruchomienie odtwarzania (`Play`), wstrzymanie odtwarzania (`Pause`) oraz zatrzymanie odtwarzania (`Stop`).

Ostatnia w tej klasie metoda, `radio_Checked` (wykorzystywana na zakładce *Ustawienia*), obsługuje zdarzenie kliknięcia `RadioButton` z kolorem paska postępu.

Program można już uruchomić. Rysunek 9.1 przedstawia działanie programu podczas odtwarzania pliku mp3.

**Rysunek 9.1.**  
*Odtwarzacz audio*



Prezentowany prosty program działa zgodnie z założeniami i odtwarza wskazany plik audio. Wygląd aplikacji jest na razie mało efektowny. Po zapoznaniu się w kolejnych rozdziałach z zagadnieniami dotyczącymi stylów i szablonów będzie można dopracować warstwę wizualną odtwarzacza. A już teraz możesz zmienić pewne drobiazgi, na przykład zamienić tekstową zawartość przycisków na graficzne ikonki:

```
<Button x:Name="btnPlay" Click="btnPlay_Click" Margin="10" ToolTip="Odtwórz"
        IsEnabled="False">
    <Image Source="Rysunki/play.jpg" MaxHeight="20" MaxWidth="20"/>
</Button>
```

## 9.3 Zadania

Program wykonany w tym rozdziale można rozwijać o dalsze funkcjonalności. W pierwszym zadaniu należy dodać suwak. Zadanie drugie będzie wymagało drobnych zmian formatowania nagłówków zakładek.

### Zadanie 9.1

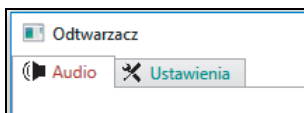
Dodaj do omawianego w rozdziale programu kontrolkę **Slider** (suwak), pozwalającą użytkownikowi zmienić położenie punktu odtwarzania.

### Zadanie 9.2

Zmień kolor dla tekstu w nagłówkach obu zakładek i umieść w nagłówkach ikonki. Rysunek 9.2 przedstawia przykładowy wygląd nagłówków zakładek po zmianach formatowania i dodaniu obrazków.

#### Rysunek 9.2.

*Nagłówki zakładek  
po zmianach  
formatowania*



## 9.4 Wskazówki do zadań

Wskazówki zawierają szczegółowe wyjaśnienia i większe fragmenty rozwiązań (dla pierwszego zadania podaję kompletne rozwiązanie).

### Wskazówki do zadania 9.1

W kodzie XAML pod definicją kontrolki **ProgressBar** wpisz kod dla suwaka:

```
<Slider x:Name="slGra" Thumb.DragStarted="slGra_DragStarted"
Thumb.DragCompleted="slGra_DragCompleted" Margin="20"/>
```

W *code-behind* dodaj dyrektywę **using** dla przestrzeni nazw:

```
using System.Windows.Controls.Primitives;
```

W klasie **MainWindow** zdefiniuj nowe pole prywatne:

```
private bool czySuwakJestPrzesuwany = false;
```

Podmień zawartość metody **timerTick** na następującą:

```
void timerTick(object sender, EventArgs e)
{
    if (mediaPlayer.Source != null && mediaPlayer.NaturalDuration.HasTimeSpan
        && !czySuwakJestPrzesuwany)
```

```

    {
        txtCzas.Text = mediaPlayer.Position.ToString(@"mm:ss");
        // Ustawienia dla ProgressBar
        TimeSpan ts = mediaPlayer.NaturalDuration.TimeSpan;
        pbGra.Maximum = 100;
        pbGra.Value = ((double)mediaPlayer.Position.TotalMilliseconds /
            ↳ts.TotalMilliseconds) * 100;
        // Ustawienia dla Slider
        slGra.Maximum = mediaPlayer.NaturalDuration.TimeSpan.TotalMilliseconds;
        slGra.Value = mediaPlayer.Position.TotalMilliseconds;
    }
}

```

W kodzie tej metody zmieniono warunek aktualizacji i dodano sprawdzenie, czy suwak nie jest przesuwany. Jeśli jest właśnie przesuwany przez użytkownika, to program się wstrzyma z aktualizacją informacji o stanie odtwarzania. Na końcu metody dodano aktualizację bieżącej pozycji suwaka (`slGra.Value`).

Pozostało tylko dodanie metod obsługujących dwa zdarzenia — rozpoczęcia przesuwania suwaka (`Thumb.DragStarted`) i zakończenia przesuwania suwaka (`Thumb.↳DragCompleted`):

```

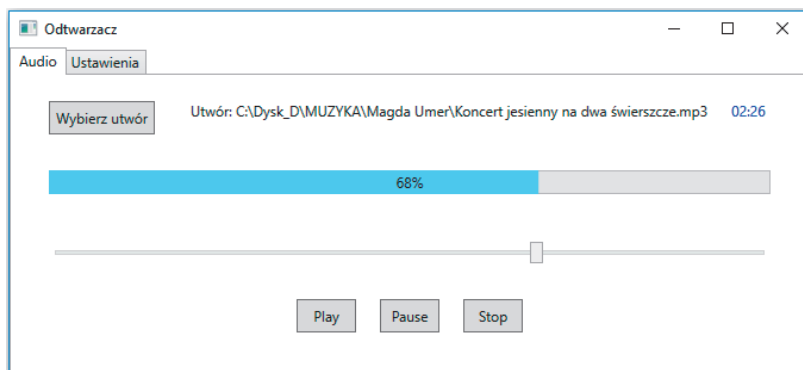
private void slGra_DragStarted(object sender, DragStartedEventArgs e)
{
    czySuwakJestPrzesuwany = true;
}
private void slGra_DragCompleted(object sender, DragCompletedEventArgs e)
{
    czySuwakJestPrzesuwany = false;
    mediaPlayer.Position = TimeSpan.FromMilliseconds(slGra.Value);
}

```

W pierwszej z tych metod flaga (pole logiczne) `czySuwakJestPrzesuwany` ustawiana jest na `true`. Natomiast w metodzie `slGra_DragCompleted` flaga jest zmieniana na `false` i ponadto bieżąca pozycja odtwarzania jest ustawiana według aktualnego położenia suwaka.

Rysunek 9.3 przedstawia przykładowe działanie programu po zmianach.

**Rysunek 9.3.**  
Odtwarzacz audio  
po dodaniu suwaka



Po dodaniu suwaka kontrolkę `ProgressBar` można wyrzucić albo zmniejszoną (i bez procentów) przenieść w inne miejsce.

## Wskazówki do zadania 9.2

W kodzie XAML dla obu zakładek należy zdefiniować nagłówki w osobnych znacznikach. Oto przykładowa definicja jednego z nagłówków:

```
<TabItem.Header>
  <StackPanel Orientation="Horizontal">
    <Image Source="Rysunki/glosnik.png" MaxHeight="16" MaxWidth="16" />
    <TextBlock Text="Audio" Foreground="Brown" Padding="5,0" />
  </StackPanel>
</TabItem.Header>
```

## Rozdział 10.

# Zasoby, style i wyzwalacze

W WPF zasoby dzieli się na dwie grupy: zasoby binarne i zasoby logiczne. Zasobami binarnymi są zazwyczaj obrazki, czcionki czy inne pliki. Natomiast zasoby logiczne wychodzą poza tradycyjne rozumienie pojęcia „zasób” i odnoszą się do fragmentów kodu aplikacji, które można wykorzystać w innych miejscach. W niniejszym rozdziale oprócz zasobów zostaną przedstawione także style, dzięki którym można usprawnić definiowanie wyglądu aplikacji, oraz wyzwalacze, które pozwalają zmieniać ustawienia wyglądu elementów dynamicznie, po spełnieniu określonych warunków. Zasoby (zarówno binarne, jak i logiczne) oraz style pojawiały się już w naszych programach, ale w tym rozdziale uporządkujemy i rozszerzymy wiedzę na ten temat.

## 10.1 Zasoby binarne

Visual Studio obsługuje dla aplikacji WPF kilka różnych akcji budowania (ang. *build action*), czyli sposobów przechowywania zasobów w skompilowanym projekcie. Zasobów binarnych dotyczą:

- ♦ **Resource** — zasób zostaje wbudowany w podzespół (do pliku wynikowego).
- ♦ **Content** — zasób jest włączony do projektu, ale jako osobny plik.

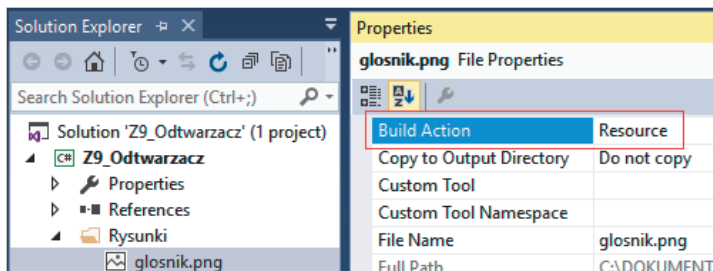
Istnieje jeszcze jeden sposób (wykorzystywany przede wszystkim w Windows Forms), ale nie jest zalecany dla WPF — **Embedded Resource**.

Po włączeniu plików binarnych (np. obrazków) do projektu w Visual Studio (*Add/Existing Item*) domyślnie przyjmują one typ akcji budowania **Resource**, co można sprawdzić we właściwościach pliku w oknie *Properties* (rysunek 10.1).

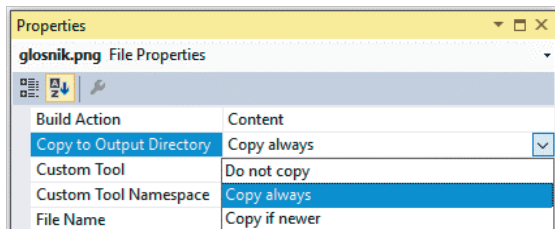
Akcję budowania (*Build Action*) można zmienić (w oknie *Properties*) na **Content**. Aby zasoby zostały przekopiowane w miejsce, gdzie znajduje się plik wynikowy (z zachowaniem struktury katalogów), należy zmienić także ustawienie dla *Copy to Output Directory* na *Copy always* lub *Copy if newer* (rysunek 10.2).

**Rysunek 10.1.**

Właściwości  
wskazanego  
zasobu projektu  
(akcja budowania  
Resource)

**Rysunek 10.2.**

Zmiana właściwości  
wskazanego zasobu  
projektu (akcja  
budowania Content)



Do zasobów **Content** i **Resource** można odwoływać się w kodzie XAML za pomocą URI. Dla przykładowego obrazka umieszczonego w Visual Studio, zarówno dla zasobów typu **Resource** (rysunek 10.1), jak i **Content** (rysunek 10.2), możemy w kodzie XAML odwołać się w następujący sposób:

```
<Image Source="Rysunki/głosnik.png"/>
```

Albo z użyciem formatu Pack URI:

```
<Image Source="pack://application:,,,/Rysunki/głosnik.png"/>
```

Autoryzacja **application** identyfikuje pliki znane już w czasie kompilacji programu i jest niejawnie używana we wszystkich schematach URI z wyjątkiem tych, którym przypisano autoryzację **siteoforigin** (dla osobnych plików)<sup>1</sup>.

Dostęp do zasobów binarnych w *code-behind* może wyglądać następująco:

```
Image rysunek = new Image();
rysunek.Source = new BitmapImage(new Uri("pack://application:,,,/
↳Rysunki/rys1.png "));
```

## 10.2 Zasoby logiczne

Zasoby logiczne są dowolnymi obiektami .NET i zazwyczaj definiowane są w celu ich współdzielenia przez elementy podrzędne. Przydatność takich zasobów wyjaśnimy na prostym przykładzie.

<sup>1</sup> Więcej o formacie Pack URI znajdziesz w: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 350 – 353).



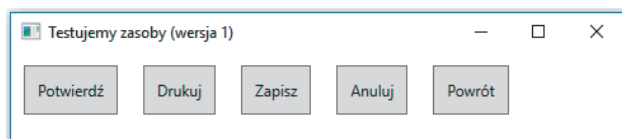
Wykonamy program zawierający pięć przycisków. W kodzie XAML ustaw atrybut `Title="Testujemy zasoby (wersja 1)"`. Zmień także wymiary okna aplikacji na przykład na: `Height="150" Width="500"`. W oknie XAML podmień kod ze znacznikami `<Grid>` i `</Grid>` na następujący:

```
<Grid>
  <WrapPanel>
    <Button Content="Potwierdź" Margin="10" Padding="10"/>
    <Button Content="Drukuj" Margin="10" Padding="10" />
    <Button Content="Zapisz" Margin="10" Padding="10" />
    <Button Content="Anuluj" Margin="10" Padding="10"/>
    <Button Content="Powrót" Margin="10" Padding="10"/>
  </WrapPanel>
</Grid>
```

W panelu `WrapPanel` zdefiniowane są proste przyciski. Po uruchomieniu programu widzimy okno podobne jak na rysunku 10.3.

### Rysunek 10.3.

Wygląd okna aplikacji testującej zasoby logiczne (wersja 1)



Załóżmy jednak, że chcielibyśmy, aby nasze przyciski były bardziej oryginalne i miały gradientowe tło. Moglibyśmy tej zmiany dokonać w następujący sposób:

```
<Button Content="Potwierdź" Margin="10" Padding="10">
  <Button.Background>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
      <GradientStop Color="White" Offset="0"/>
      <GradientStop Color="LightBlue" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

I tak dla wszystkich pięciu przycisków. W ten sposób uzyskamy oczekiwany efekt zmiany tła, ale to rozwiązanie ma wady. Mianowicie wystąpi tu wyraźny przyrost powtarzającego się kodu, to znaczy dodatkowe linie kodu (definiujące tło) dla każdego przycisku będą miały tę samą zawartość. Oczywiście możemy wpisać (przekopiować) ten powtarzający się fragment kodu dla wszystkich przycisków, ale musimy pamiętać o dwóch rzeczach. Po pierwsze w typowej większej aplikacji przycisków może być znacznie więcej niż pięć, a po drugie w każdej chwili może zaistnieć potrzeba zmiany koloru tła. W takiej sytuacji na pewno byłoby wygodniej dokonać modyfikacji w jednym miejscu, a nie we wszystkich, gdzie występują zmieniane elementy.

Wykonamy drugą wersję programu, tym razem z użyciem zasobu logicznego. Wewnątrz elementu `Window`, tuż przed znacznikiem `<Grid>`, wpisz kod definiujący zasób<sup>2</sup>:

<sup>2</sup> W programie zastosowano prosty gradient liniowy. Punkty `StartPoint` i `EndPoint` decydują o położeniu osi gradientu. W prezentowanym kodzie została zdefiniowana pionowa oś gradientu. Poszczególne „warstwy” kolorów są umieszczane prostopadle do osi gradientu. Więcej na temat gradientów liniowych można znaleźć na stronie MSDN („Painting with Solid Colors and Gradients Overview”): [https://msdn.microsoft.com/en-us/library/ms754083\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms754083(v=vs.110).aspx).

```

<Window.Resources>
  <LinearGradientBrush x:Key="tloPrzyciskow" StartPoint="0.5,0" EndPoint="0.5,1">
    <GradientStop Color="White" Offset="0"/>
    <GradientStop Color="LightBlue" Offset="1"/>
  </LinearGradientBrush>
</Window.Resources>

```

A następnie zmień kod definiujący przyciski:

```

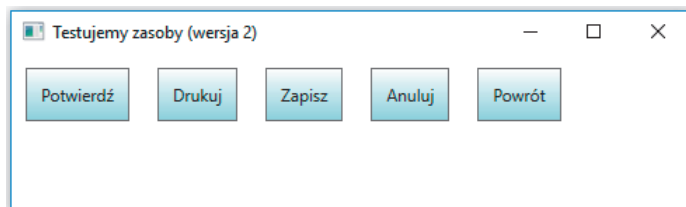
<Button Content="Potwierdź" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
<Button Content="Drukuj" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
<Button Content="Zapisz" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
<Button Content="Anuluj" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
<Button Content="Powrót" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>

```

Obecnie kod definiujący przyciski jest zwięzły. Do właściwości `Background` przypisany został zasób identyfikowany jako `tloPrzyciskow`. Po uruchomieniu programu w nowej wersji widzimy gradientowe tło dla przycisków (rysunek 10.4).

#### Rysunek 10.4.

*Wygląd okna  
aplikacji testującej  
zasoby logiczne  
(wersja 2)*



## Zasięg zasobu

Zwróćmy jeszcze uwagę na zasięg zasobu. W programie zasób został umieszczony wewnątrz elementu `Window`, czyli dotyczy bieżącego okna. Można jednak określać zasięg zasobu inaczej, na przykład dla całej aplikacji<sup>3</sup>, danego elementu lub jego rodzica.

Wykonajmy trzecią wersję programu, która pozwoli lepiej zrozumieć zagadnienie zasięgu zasobu. Usuń definicję zasobu z elementu `Window` i zamień panel `Grid` na następujący:

```

<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <WrapPanel Grid.Row="0" Grid.Column="0">

```

<sup>3</sup> Zasoby dla całej aplikacji można definiować wewnątrz elementu `Application` w pliku `App.xaml`.

```

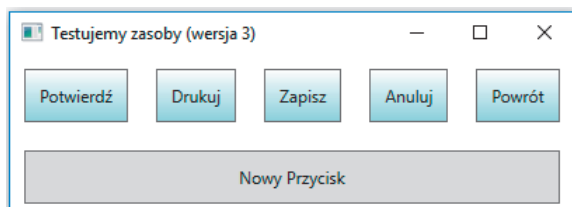
<WrapPanel.Resources>
  <LinearGradientBrush x:Key="tloPrzyciskow" StartPoint="0.5,0"
    EndPoint="0.5,1">
    <GradientStop Color="White" Offset="0"/>
    <GradientStop Color="LightBlue" Offset="1"/>
  </LinearGradientBrush>
</WrapPanel.Resources>
<Button Content="Potwierdź" Margin="10" Padding="10"
  Background="{StaticResource tloPrzyciskow}" />
<Button Content="Drukuj" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
<Button Content="Zapisz" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
<Button Content="Anuluj" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
<Button Content="Powrót" Background="{StaticResource tloPrzyciskow}"
  Margin="10" Padding="10"/>
</WrapPanel>
<Button Grid.Row="1" Grid.Column="0" Content="Nowy Przycisk" Margin="10"
  Padding="10"/>
</Grid>

```

Wygląd okna w nowej wersji programu przedstawia rysunek 10.5.

### Rysunek 10.5.

*Wygląd okna  
aplikacji testującej  
zasoby logiczne  
(wersja 3)*



W nowej wersji programu są dwie zmiany. Przede wszystkim zasób jest teraz definiowany wewnątrz elementu `WrapPanel`, a nie `Window`. Konsekwencją tej zmiany jest to, że dostęp do zasobu mają tylko elementy potomne tego panelu. Drugą zmianą w programie jest dodanie nowego przycisku, będącego poza `WrapPanel`. Ten nowy przycisk nie ma dostępu do zasobu. Jeśli zmienisz jego definicję na poniższą, pojawi się błąd:

```

<Button Grid.Row="1" Grid.Column="0" Content="Nowy Przycisk"
  Background="{StaticResource tloPrzyciskow}" Margin="10" Padding="10"/>

```

Komunikat o błędzie poinformuje nas, że zasób `tloPrzyciskow` nie może być w tym miejscu użyty.



Uwaga

Poszczególne elementy mogą być dowolnie zagnieżdżane, a kod XAML obrazujący to zagnieżdżanie przypomina strukturę drzewiastą. Na różnych poziomach zagnieżdżenia mogą występować definicje zasobów. Dla danego elementu zostanie użyty zasób, który znajduje się bliżej tego elementu w całym drzewie.

Wykonajmy jeszcze jeden test. Umieścimy z powrotem definicję zasobu w elemencie `Window`, ale tym razem z innym kolorem:

```

<Window.Resources>
  <LinearGradientBrush x:Key="tloPrzyciskow" StartPoint="0.5,0" EndPoint="0.5,1">
    <GradientStop Color="White" Offset="1"/>
    <GradientStop Color="LightGray" Offset="0"/>
  </LinearGradientBrush>
</Window.Resources>

```

Następnie ustawmy definicję nowego przycisku z odwołaniem do zasobu:

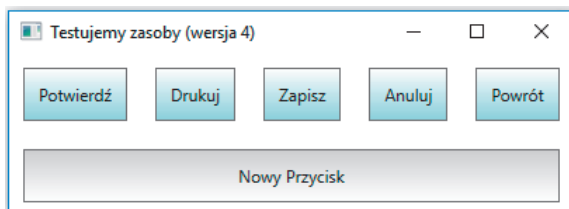
```

<Button Grid.Row="1" Grid.Column="0" Content="Nowy Przycisk"
  Background="{StaticResource tloPrzyciskow}" Margin="10" Padding="10"/>

```

Tym razem błędu nie będzie. Program obecnie ma dwie definicje zasobu na dwóch różnych poziomach (okna `Window` i panelu `WrapPanel`). Dla nowego przycisku najbliższą dostępną definicją zasobu jest ta w elemencie `Window` (rysunek 10.6).

**Rysunek 10.6.**  
Wygląd okna  
aplikacji testującej  
zasoby logiczne  
(wersja 4)



Z kolei dla przycisków będących w panelu `WrapPanel` najbliższą definicją zasobu jest ta w elemencie `WrapPanel` i ten zasób został w ich przypadku zastosowany.

## Zasoby statyczne i dynamiczne

WPF obsługuje dwa sposoby dostępu do zasobu logicznego:

- ◆ Statyczny, definiowany za pomocą rozszerzenia `StaticResource` — taki zasób jest przypisany do danego elementu tylko raz (gdy jest potrzebny po raz pierwszy).
- ◆ Dynamiczny, definiowany za pomocą rozszerzenia `DynamicResource` — taki zasób jest przypisywany za każdym razem, gdy ulegnie zmianie.

Użycie obu sposobów dostępu do zasobów w zakresie ich zasadniczej funkcjonalności nie różni się. Główna różnica między nimi polega na tym, że dostęp dynamiczny pozwala na aktualizację danego elementu tuż po zmianie zasobu (np. gdy użytkownik wybierze inny kolor tła). Ponadto dostęp dynamiczny może być wykorzystywany wyłącznie dla właściwości zależnych, a ograniczenie to nie dotyczy dostępu statycznego.

## Uwagi końcowe

Zasoby logiczne mogą być zdefiniowane w osobnych plikach XAML — wówczas należy je scalić z kodem aplikacji za pomocą właściwości `MergedDictionaries` klasy `ResourceDictionary`, na przykład:

```

<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="nazwaPliku1.xaml"/>
      <ResourceDictionary Source="nazwaPliku2.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>

```

Osobne pliki z definicją zasobów muszą zawierać `ResourceDictionary` jako korzeń (główny element).

W prezentowanym przykładzie jako zasób zdefiniowaliśmy gradient liniowy, można jednak zdefiniować wiele innych właściwości. Poniższy zasób definiuje kolor pędzla, jakiego możemy użyć na przykład do pokolorowania ramki przycisku:

```

<Window.Resources>
  <SolidColorBrush x:Key="kolorRamki">Red</SolidColorBrush>
</Window.Resources>

```

Przykładowe użycie tak zdefiniowanego zasobu może wyglądać następująco:

```

<Button Content="Potwierdź" BorderBrush="{StaticResource kolorRamki}"
  Margin="10" Padding="10"/>

```

W zasobie można zdefiniować właściwie całą kontrolkę. Jeżeli przykładowo zdefiniujemy jako zasób element `Image`:

```

<Window.Resources>
  <Image x:Key="obrazek" Height="25" Source="Rysunki/glosnik.png"/>
</Window.Resources>

```

to możemy użyć tej definicji w taki sposób:

```

<StaticResource ResourceKey="obrazek"/>

```

Program będzie działał poprawnie. Jeżeli jednak chcielibyśmy powielić tę kontrolkę:

```

<WrapPanel>
  <StaticResource ResourceKey="obrazek"/>
  <StaticResource ResourceKey="obrazek"/> // W tej linii zostanie zgłoszony błąd!
</WrapPanel>

```

to taki kod spowoduje błąd. Domyślnie jest używana ta sama instancja obiektu. Możemy mieć w programie dwie identyczne kontrolki, ale jako dwa osobne obiekty. Ustawienie `x:Shared="False"` pozwala na wielokrotne użycie danego zasobu jako niezależnych obiektów<sup>4</sup>. Możemy zatem poprawić ten błąd poprzez zmianę definicji zasobu:

```

<Window.Resources>
  <Image x:Key="obrazek" x:Shared="False" Height="25"
    Source="Rysunki/glosnik.png"/>
</Window.Resources>

```

<sup>4</sup> Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 362).

Prezentowane przykłady obrazują dość wyraźnie korzyści, jakie przynosi stosowanie zasobów logicznych. Kolejnym cennym udogodnieniem WPF jest możliwość grupowania ustawień poprzez tak zwane style.

## 10.3 Style

Zwróćmy uwagę, że w przykładzie z poprzedniego podrozdziału, w kodzie opisującym poszczególne przyciski, wciąż jest powielany ten sam kod z ustawieniami dla `Margin` i `Padding`. Możemy sobie wyobrazić, że takich ustawień jest jeszcze więcej, na przykład dla czcionki tekstu czy grubości ramki itp. Dzięki stylom można zdefiniować cały „pakiet” ustawień w jednym miejscu, a następnie wykorzystać tę definicję podczas tworzenia danej grupy elementów. Style zazwyczaj definiowane są w zasobach i korzystanie z nich odbywa się według tych samych zasad, to znaczy dostępność stylu zależy od jego umiejscowienia w kodzie.

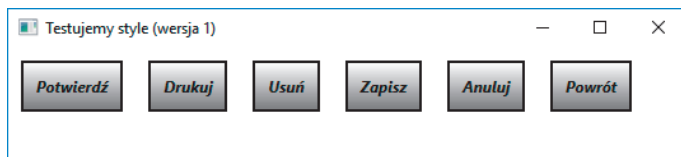
Wykonamy program z użyciem stylu. Utwórz nowy projekt. W kodzie XAML ustaw atrybut `Title="Testujemy style (wersja 1)"`. Zmień także wymiary okna aplikacji na przykład na: `Height="180" Width="550"`. W oknie *XAML* podmień kod ze znacznikami `<Grid>` i `</Grid>` na następujący:

```
<Grid>
  <WrapPanel>
    <WrapPanel.Resources>
      <Style x:Key="stylPrzyciskow">
        <Setter Property="Button.Margin" Value="10"/>
        <Setter Property="Button.Padding" Value="10"/>
        <Setter Property="Button.FontSize" Value="12"/>
        <Setter Property="Button.FontWeight" Value="Bold"/>
        <Setter Property="Button.FontStyle" Value="Italic"/>
        <Setter Property="Button.BorderBrush" Value="Black"/>
        <Setter Property="Button.BorderThickness" Value="2"/>
        <Setter Property="Button.Background">
          <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="White" Offset="0"/>
            <GradientStop Color="Gray" Offset="1"/>
          </LinearGradientBrush>
        </Setter.Value>
      </Setter>
    </Style>
  </WrapPanel.Resources>
  <Button Content="Potwierdź" Style="{StaticResource stylPrzyciskow}"/>
  <Button Content="Drukuj" Style="{StaticResource stylPrzyciskow}"/>
  <Button Content="Usuń" Style="{StaticResource stylPrzyciskow}"/>
  <Button Content="Zapisz" Style="{StaticResource stylPrzyciskow}"/>
  <Button Content="Anuluj" Style="{StaticResource stylPrzyciskow}"/>
  <Button Content="Powrót" Style="{StaticResource stylPrzyciskow}"/>
</WrapPanel>
</Grid>
```

Styl został zdefiniowany jako zasób w panelu `WrapPanel`. Podobnie jak inne zasoby logiczne, można umieścić go w dowolnym innym miejscu (np. w elemencie `Window`). Umieszczenie zasobu decyduje o jego zasięgu. W przykładowym programie wszystkie kontrolki (przyciski) umieszczone zostały w panelu `WrapPanel`, co pozwala zdefiniować styl na tym właśnie poziomie.

Definicja stylu rozpoczyna się od elementu `Style` wraz z kluczem, według którego styl będzie identyfikowany. Następnie w kolejnych liniach definiowane są obiekty klasy `Setter` z właściwościami `Property` i `Value`. I tak w pierwszym elemencie `Setter` jest przypisanie wartości 10 dla marginesu przycisku. W dalszych liniach definiowane są inne ustawienia dla przycisku, takie jak rozmiar czcionki, styl czcionki, kolor i grubość ramki czy kolor tła przycisku. Definicje przycisków zawierają już tylko napisy i odwołanie do stylu `stylPrzyciskow`. Program po uruchomieniu daje rezultat przedstawiony na rysunku 10.7.

**Rysunek 10.7.**  
Wygląd okna  
aplikacji testującej  
style (wersja 1)



Można zdefiniować styl, który będzie współdzielony przez elementy heterogeniczne, np. przycisk, zakładka, pole tekstowe. Wówczas w definicji stylu zostanie użyta klasa `Control`. Przetestujemy nową wersję programu:

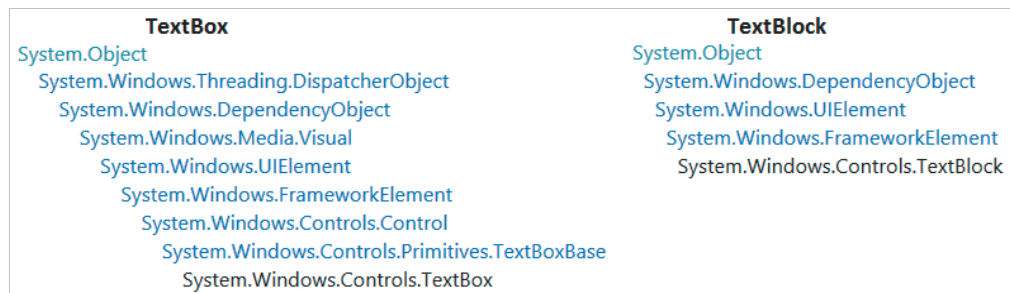
```
<Grid>
  <WrapPanel>
    <WrapPanel.Resources>
      <Style x:Key="stylElementow">
        <Setter Property="Control.Margin" Value="10"/>
        <Setter Property="Control.Padding" Value="10"/>
        <Setter Property="Control.FontSize" Value="12"/>
        <Setter Property="Control.FontWeight" Value="Bold"/>
        <Setter Property="Control.FontStyle" Value="Italic"/>
        <Setter Property="Control.BorderBrush" Value="Black"/>
        <Setter Property="Control.BorderThickness" Value="2"/>
        <Setter Property="Control.Background">
          <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="White" Offset="0"/>
            <GradientStop Color="Gray" Offset="1"/>
          </LinearGradientBrush>
        </Setter.Value>
      </Setter>
    </Style>
  </WrapPanel.Resources>
  <Button Content="Przycisk" Style="{StaticResource stylElementow}"/>
  <TextBox Text="Pole tekstowe" Style="{StaticResource stylElementow}"/>
  <TextBlock Text="TextBlock. A tu nie ma tła zdefiniowanego w stylu"
    Style="{StaticResource stylElementow}"/>
</WrapPanel>
</Grid>
```

Po uruchomieniu widzimy, że zarówno przycisk (**Button**), jak i pole tekstowe (**TextBox**) prezentują się ściśle według określonego stylu. Jedynie **TextBlock** wygląda inaczej: czcionka jest zgodna ze stylem, ale tło i ramka już nie są (rysunek 10.8).

**Rysunek 10.8.**  
Wygląd okna  
aplikacji testującej  
style (wersja 2)



Spójrzmy na diagramy dziedziczenia klas **TextBox** i **TextBlock** (rysunek 10.9).



**Rysunek 10.9.** Diagram dziedziczenia dla klas **TextBox** i **TextBlock** (na podstawie dokumentacji MSDN)

Klasa **TextBox** dziedziczy po klasie **Control**, natomiast klasa **TextBlock** dziedziczy bezpośrednio po klasie **FrameworkElement**. W takiej sytuacji wydawałoby się, że **TextBlock** nie powinien przyjmować żadnych właściwości ustalonych w stylu dla klasy **Control**. Tak się jednak nie dzieje, rozmiar i styl czcionki został ustawiony zgodnie ze stylem. Taka sytuacja ma miejsce, gdy właściwość zależna jest dzielona między różnymi klasami. I jak widzimy, nie dotyczy to między innymi właściwości **Background**, ponieważ tło pozostało domyślne. Używanie uogólnionych stylów wymaga pewnej rozważki i przez niektórych nie jest zalecane<sup>5</sup>.

Możemy ograniczyć użycie stylu, stosując właściwość **TargetType**. Jeżeli w prezentowanym przykładzie (wersja 2) umieścimy tę właściwość w definicji stylu, na przykład:

```
<Style x:Key="stylElementow" TargetType="{x:Type Control}">
```

wówczas będziemy zmuszeni zmienić lub usunąć definicję **TextBlock**.

Właściwość **TargetType** nie wymaga używania przedrostka dla nazwy klasy przy określaniu wartości dla **Property**. Oznacza to, że jeśli przykładowo użyjemy właściwości **TargetType**, ograniczającej stosowanie stylu do klasy **Button**, to zamiast **Property="Button.Margin"** możemy pisać **Property="Margin"**, tak jak to ma miejsce w definicji stylu:

<sup>5</sup> Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 424 – 426).



```

<Style x:Key="stylPrzyciskow" TargetType="{x:Type Button}">
  <Setter Property="Margin" Value="10"/>
  <Setter Property="Padding" Value="10"/>
  <Setter Property="FontSize" Value="12"/>
</Style>

```

Przeanalizujemy definicje stylów, jakie były już wykorzystane w programach omawianych w tej książce. W podrozdziale 7.2 wykorzystaliśmy styl do formatowania kolumny tekstowej w `DataGrid`:

```

<DataGridTextColumn.ElementStyle>
  <Style TargetType="{x:Type TextBlock}">
    <Setter Property="HorizontalAlignment" Value="Right" />
  </Style>
</DataGridTextColumn.ElementStyle>

```

Właściwość `ElementStyle` klasy `DataGridTextColumn` pozwala ustawić styl dla komórki w kolumnie, która nie jest w trybie edycji<sup>6</sup>. Ponieważ ustawienie dotyczyło tylko jednej kontrolki (`DataGrid`), można było zdefiniować styl dla właściwości `ElementStyle` bez używania zasobów. W przeciwnym razie warto byłoby skorzystać z definicji zasobu, na przykład:

```

<Window.Resources>
  <Style x:Key="stylDlaLiczb" TargetType="{x:Type TextBlock}">
    <Setter Property="HorizontalAlignment" Value="Right"/>
    <Setter Property="Foreground" Value="Blue"/>
  </Style>
</Window.Resources>

```

Wówczas definicja kolumny *Liczba sztuk* w `DataGrid` miałaby postać:

```

<DataGridTextColumn Header="Liczba sztuk" Binding="{Binding LiczbaSztuk}"
  ElementStyle="{StaticResource stylDlaLiczb}"/>

```

Spójrzmy także na definicję stylu w podrozdziale 6.5:

```

<Window.Resources>
  <Style TargetType="ListViewItem">
    <Setter Property="HorizontalContentAlignment" Value="Stretch" />
  </Style>
</Window.Resources>

```

Należy zwrócić uwagę, że w definicji tego stylu brakuje klucza ustawianego przez `x:Key`. Pominięcie klucza w definicji stylu powoduje, że styl ten zostaje niejawnie przypisany dla elementów typu określonego przez `TargetType`. Przypisanie to odbywa się w zasięgu wynikającym z umiejscowienia definicji danego stylu. Przykładowo tak zdefiniowany styl jako zasób w elemencie `Window` będzie niejawnie przypisany dla wszystkich elementów określonego typu w danym oknie (tu dla `ListViewItem`). W powyższym przykładzie zostało pominięte także rozszerzenie znaczników `x:Type`, ale to akurat nie

<sup>6</sup> Styl dla komórki w trybie edycji ustawiany jest za pomocą właściwości `EditingElementStyle`.

wpływa na działanie programu. Oba zapisy, to znaczy `<Style TargetType="ListViewItem">` i `<Style TargetType="{x:Type ListViewItem}">`, są równoważne<sup>7</sup>.

Można zrezygnować z niejawnego przypisania stylu dla danego typu poprzez ustawienie właściwości `Style` na `Null`. Definiując przykładowy styl dla `TextBox`:

```
<Window.Resources>
  <Style TargetType="{x:Type TextBox}">
    <Setter Property="Foreground" Value="Blue"/>
  </Style>
</Window.Resources>
```

możemy dla wybranych kontroltek `TextBox` zrezygnować z przypisania tego stylu jako domyślnego:

```
<StackPanel>
  <TextBox Text="Wpisz imię..." />
  <TextBox Text="Wpisz nazwisko..." />
  <TextBox Style="{x:Null}" Text="Wpisz numer..." />
</StackPanel>
```

Dwie pierwsze kontrolki `TextBox` wyświetlą tekst w kolorze niebieskim (zgodnie z definiowanym stylem), natomiast ostatnia, zawierająca ustawienie `Style="{x:Null}"`, wyświetli tekst w kolorze czarnym (czyli bez uwzględnienia definicji tego stylu).

Temat zasobów logicznych i stylów, ściśle związany z dziedziczeniem właściwości w drzewie elementów, zostanie uzupełniony przykładami prezentującymi dziedziczenie **właściwości zależnych** bez definiowania zasobów i stylów. Przypominam, że nie chodzi tu o dziedziczenie w hierarchii klas, lecz w drzewiastej strukturze kodu XAML (której poświęcimy więcej miejsca w kolejnym rozdziale). Programując w WPF, musimy pamiętać o tych dwóch perspektywach dziedziczenia.

Popatrzmy na przykład kodu XAML, w którym element główny `Window` zawiera właściwości `FontSize` i `FontWeight`:

```
<Window x:Class="WlasciosciZalezne.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Main Window" Height="350" Width="525" FontSize="25"
  FontWeight="Bold">
  <StackPanel>
    <TextBlock Text="Potwierdź lub anuluj" FontSize="15"/>
    <WrapPanel>
      <Button Content="Zapisz" Margin="10"/>
      <Button Content="Anuluj" Margin="10"/>
    </WrapPanel>
  </StackPanel>
</Window>
```

<sup>7</sup> Więcej na temat możliwości przypisania do właściwości `TargetType` łańcucha znakowego z nazwą danego typu można znaleźć na stronie dokumentacji MSDN („x:Type Markup Extension”): [https://msdn.microsoft.com/en-us/library/ms753322\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms753322(v=vs.110).aspx).

Potomkiem elementu `Window` jest panel `StackPanel`. Zawarty w panelu `TextBlock` przyjmie czcionkę pogrubioną (**bold**) o rozmiarze 15 (bo dla tego elementu została przesłonięta definicja właściwości `FontSize`). Natomiast zawartość obu przycisków będących wewnątrz panelu `WrapPanel` będzie wyświetlona według ustawień ustalonych w elemencie `Window` (czcionka pogrubiona o rozmiarze 25)<sup>8</sup>.

Klasa `Window` udostępnia właściwości `FontSize` i `FontWeight` (obie dziedziczone po klasie `Control`), więc mogliśmy ich użyć w definicji `Window`. Ale możliwa jest także inna sytuacja. Dzięki **właściwościom dołączonym** możemy w definicji elementów nadrzędnych przypisywać wartość właściwości, jakich te elementy nie mają, na przykład w panelu `WrapPanel` możemy przypisać ustawienia dla czcionki:

```
<WrapPanel TextElement.FontSize="50" TextElement.FontWeight="Light">
  <Button Content="Zapisz" Margin="10"/>
  <Button Content="Anuluj" Margin="10"/>
</WrapPanel>
```

W definiowanym panelu zostały użyte dwie właściwości dołączone — dla rozmiaru czcionki i grubości (poprzedzając ich nazwy klasą dostawcy). Zostało to zrobione z myślą o elementach podrzędnych. Zawartość obu przycisków zostanie wyświetlona zgodnie z ustawieniami właściwości dołączonych.

## 10.4 Wyzwalacze

Wyzwalacze umożliwiają definicję stylów, które zostaną zastosowane warunkowo. Wyróżnia się trzy rodzaje wyzwalaczy<sup>9</sup>:

- ♦ Wyzwalacze właściwości (ang. *property triggers*) — wywoływane są po zmianie wartości właściwości zależnej.
- ♦ Wyzwalacze danych (ang. *data triggers*) — wywoływane są po zmianie wartości każdej właściwości .NET (nie tylko zależnej).
- ♦ Wyzwalacze zdarzeń (ang. *event triggers*) — wywoływane są w momencie wygenerowania zdarzenia kierowanego. Takie wyzwalacze mają zastosowanie w animacjach, których w tym podręczniku nie opisuję.

<sup>8</sup> W tym przykładzie wygląda to przejrzysto, ale nie zawsze tak jest. Dziedziczenie właściwości zależnych jest podporządkowane zasadom, które wyznaczają ostateczną wartość właściwości na podstawie różnych źródeł. Więcej informacji na ten temat można znaleźć w: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 81 – 84).

<sup>9</sup> Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 430).

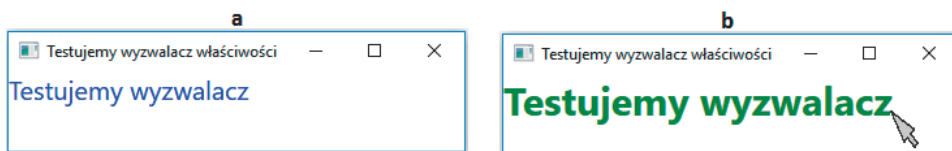
## Wyzwalacze właściwości

Przykład wyzwalacza właściwości prezentuje poniższy prosty kod:

```
<Window.Resources>
  <Style TargetType="{x:Type TextBlock}">
    <Setter Property="FontSize" Value="20"/>
    <Setter Property="Foreground" Value="Blue"/>
    <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="FontSize" Value="30"/>
        <Setter Property="Foreground" Value="Green"/>
        <Setter Property="FontWeight" Value="Bold"/>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
<StackPanel>
  <TextBlock Text="Testujemy wyzwalacz"/>
</StackPanel>
```

Wyzwalacze tej grupy zaimplementowane są w klasie `Trigger`. Właściwości `Property` i `Value` klasy `Trigger` określają warunek, który uruchamia wyzwalacz. W prezentowanym programie wyzwalacz zostanie aktywowany w chwili, gdy właściwość zależna `IsMouseOver` będzie mieć wartość `True`.

Po uruchomieniu programu pojawi się tekst „Testujemy wyzwalacz” (rysunek 10.10a). Po najechaniu myszą na obszar napisu zmieni się rozmiar tekstu, jego kolor oraz grubość czcionki (rysunek 10.10b). Po wycofaniu myszy z obszaru `TextBlock` tekst wyświetli się według poprzednich ustawień.



Rysunek 10.10. Testowanie wyzwalacza właściwości

## Wyzwalacze danych

Wyzwalacze danych są reprezentowane przez klasę `DataTrigger`, której działanie jest podobne do wyzwalacza właściwości. Akcje tego wyzwalacza są wykonywane dla dowolnej właściwości obiektu .NET. W pierwszej kolejności przetestujemy ten rodzaj wyzwalacza dla zmiany właściwości kontrolki `CheckBox`.

W nowym projekcie WPF ustaw tytuł i rozmiar okna aplikacji na `Title="Wyzwalacze danych" Height="150" Width="300"`. Następnie wpisz kod:

```

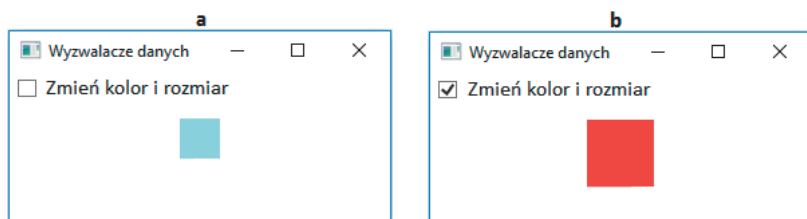
<StackPanel>
  <StackPanel.Resources>
    <Style x:Key="stylKwadrat" TargetType="Rectangle">
      <Setter Property="Margin" Value="10"/>
      <Setter Property="Fill" Value="LightBlue"/>
      <Setter Property="Width" Value="30"/>
      <Setter Property="Height" Value="30"/>
      <Style.Triggers>
        <DataTrigger Binding="{Binding ElementName=cb0n,
          Path=IsChecked}" Value="True">
          <Setter Property="Fill" Value="Red" />
          <Setter Property="Width" Value="50"/>
          <Setter Property="Height" Value="50"/>
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </StackPanel.Resources>
  <CheckBox x:Name="cb0n" Content="Zmień kolor i rozmiar" Margin="5"/>
  <Rectangle Style="{StaticResource stylKwadrat}" />
</StackPanel>

```

W przypadku klasy `DataTrigger` do ustawienia warunku „odpalającego” wyzwalacz używa się właściwości `Binding`. W prezentowanym programie w panelu `StackPanel` definiowany jest zasób ze stylem dla kwadratu z użyciem elementu `Rectangle`. W stylu został określony margines, kolor wypełnienia oraz wymiary kwadratu (rysunek 10.11a). W dalszej części definiowany jest wyzwalacz danych `DataTrigger` dla właściwości `IsChecked` kontrolki `CheckBox`. Jeśli użytkownik ustawi tę właściwość na `True`, to zmieni się kolor kwadratu i jego wymiary. Działanie programu po zmianie właściwości `IsChecked` przedstawia rysunek 10.11b.

### Rysunek 10.11.

Wygląd kwadratu przed zmianą właściwości `IsChecked` (a) i po zmianie (b)



Dodamy do programu definicję drugiego stylu. Przed znacznikiem zamykającym zasoby panelu `</StackPanel.Resources>` wpisz kod:

```

<Style x:Key="stylPolaTekstowego" TargetType="TextBox">
  <Setter Property="MaxLength" Value="11"/>
  <Setter Property="Width" Value="80"/>
  <Setter Property="Background" Value="LightPink"/>
  <Style.Triggers>
    <DataTrigger Binding="{Binding RelativeSource={RelativeSource Self},
      Path=Text.Length}" Value="11">
      <Setter Property="Background" Value="White"/>
    </DataTrigger>
  </Style.Triggers>
</Style>

```

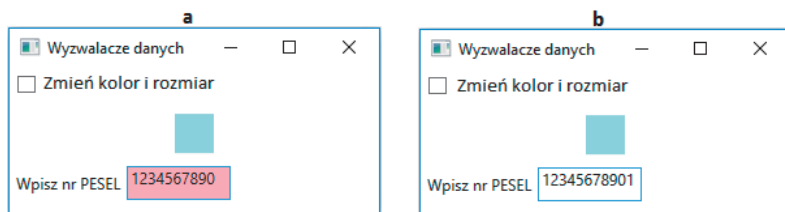
Zdefiniowany styl pozwoli zmienić kolor tła w chwili, gdy użytkownik wpisze oczekiwaną liczbę znaków przewidzianą dla danego pola tekstowego (w przykładzie jest 11).

Użycie właściwości `RelativeSource` z klasy `Binding` pozwala określić źródło w sposób względny. Jeżeli chcemy powiązać jedną właściwość elementu z inną właściwością tego samego elementu (tu pola tekstowego), możemy dla `RelativeSource` przypisać wariant `RelativeSource Self`.

Musimy jeszcze dodać pole tekstowe, w którym użyjemy stylu `stylPolaTekstowego`. Pod definicją elementu `Rectangle` wpisz kod:

```
<WrapPanel>
  <Label Content="Wpisz nr PESEL"/>
  <TextBox Style="{StaticResource stylPolaTekstowego}"/>
</WrapPanel>
```

Numer PESEL ma 11 znaków. W chwili, gdy użytkownik wpisze ostatni, jedenasty znak, kolor tła zmieni się z jasnoróżowego, mającego sygnalizować, że brakuje znaków (rysunek 10.12a), na biały (rysunek 10.12b). Na potrzeby testu wpisano kolejne cyfry, które nie stanowią prawidłowego numeru PESEL.



**Rysunek 10.12.** Pole tekstowe nie zmienia koloru tła, jeśli liczba znaków jest mniejsza niż 11 (a), natomiast po wpisaniu 11 znaków kolor tła zmienia się na biały (b)

Wykonamy jeszcze jeden program z wyzwalaczem danych, tym razem dla właściwości `LiczbaSztuk` klasy `Produkt`. Będziemy bazować na programie z rozdziału 7. Utwórz nowy projekt. Zawartość pliku `MainWindow.xaml.cs` i klasy `Produkt` przygotuj na podstawie podrozdziału 7.1. Natomiast korzystając z podrozdziału 7.2, wpisz kod XAML. Następnie zmień definicję stylu dla kolumny `Liczba sztuk` na następującą:

```
<Style TargetType="{x:Type TextBlock}">
  <Setter Property="HorizontalAlignment" Value="Right" />
  <Style.Triggers>
    <DataTrigger Binding="{Binding Path=LiczbaSztuk}" Value="0">
      <Setter Property="Foreground" Value="Red"/>
      <Setter Property="FontWeight" Value="Bold"/>
    </DataTrigger>
  </Style.Triggers>
</Style>
```

Nowa definicja stylu zawiera wyzwalacz danych wiązany z właściwością `LiczbaSztuk`, a jego akcja jest uruchamiana w chwili, gdy wartość tej właściwości będzie równa 0 — wówczas kolor tekstu zostanie zmieniony na czerwony, a czcionka zostanie pogrubiona. W ten sposób pozycje z zerowym stanem zostaną w wykazie wyróżnione.

Początkowo wszystkie produkty mają niezerowy stan, ale gdy wpisujemy zero w kolumnie *Liczba sztuk* dla wybranych produktów, to od razu zobaczymy w wykazie zmienne ustawienia formatowania (rysunek 10.13).

**Rysunek 10.13.**  
*Użycie wyzwalacza  
 danych w DataGrid  
 do wyróżnienia  
 wartości zerowych  
 w kolumnie Liczba sztuk*

Symbol	Nazwa	Liczba sztuk	Magazyn
O1-11	ołówek	0	Katowice 1
PW-20	pióro wieczne	0	Katowice 2
DZ-10	długopis żelowy	1121	Katowice 1
DZ-12	długopis kulkowy	280	Katowice 2

## Warunki logiczne w wyzwalaczach

Podczas formułowania warunków dla wyzwalaczy (wszystkich rodzajów) możemy używać zarówno alternatywy (OR), jak i koniunkcji (AND), uwzględniając kilka różnych warunków. Alternatywę budujemy, definiując poszczególne wyzwalacze na tym samym poziomie, na przykład:

```
<Style TargetType="{x:Type TextBlock}" ">
  <Setter Property="HorizontalAlignment" Value="Right" />
  <Style.Triggers>
    <DataTrigger Binding="{Binding Path=LiczbaSztuk}" Value="0">
      <Setter Property="Foreground" Value="Red"/>
      <Setter Property="FontWeight" Value="Bold"/>
    </DataTrigger>
    <DataTrigger Binding="{Binding Path=LiczbaSztuk}" Value="1">
      <Setter Property="Foreground" Value="Red"/>
      <Setter Property="FontWeight" Value="Bold"/>
    </DataTrigger>
  </Style.Triggers>
</Style>
```

Tak zdefiniowany styl zapewni zmianę formatowania wskazanej kolumny, gdy wystąpi jeden z warunków. W prezentowanym kodzie oba warunki odwołują się do tej samej właściwości (*LiczbaSztuk*), ale mogą oczywiście wystąpić odwołania do różnych właściwości.

Koniunkcję warunków dla wyzwalaczy buduje się przy użyciu specjalnych klas: *MultiTrigger* (dla wyzwalaczy właściwości) i *MultiDataTrigger* (dla wyzwalaczy danych). Konstrukcja koniunkcji w obu klasach wygląda podobnie. Przykładowa koniunkcja warunków dla stylu użytego w *DataGrid* może mieć następującą postać:

```
<Style TargetType="{x:Type TextBlock}" ">
  <Setter Property="HorizontalAlignment" Value="Right" />
  <Style.Triggers>
    <MultiDataTrigger>
      <MultiDataTrigger.Conditions>
```

```
        <Condition Binding="{Binding Path=LiczbaSztuk}" Value="0"/>
        <Condition Binding="{Binding Path=Magazyn}" Value="Katowice 1"/>
    </MultiDataTrigger.Conditions>
    <Setter Property="Foreground" Value="Red"/>
    <Setter Property="FontWeight" Value="Bold"/>
</MultiDataTrigger>
</Style.Triggers>
</Style>
```

Tym razem zmiana formatowania kolumny *Liczba sztuk* wymaga jednoczesnego spełnienia dwóch warunków zdefiniowanych przy użyciu właściwości `Conditions`, zawierającej kolekcję obiektów typu `Condition`: właściwość `LiczbaSztuk` musi mieć wartość `"0"`, a właściwość `Magazyn` musi mieć wartość `"Katowice 1"`.



## Rozdział 11.

# Szablony danych, konwertery i szablony kontroltek

W tym rozdziale omówię kolejne rozwiązania WPF pozwalające na jeszcze większą elastyczność podczas tworzenia aplikacji: szablony danych, konwertery oraz szablony kontroltek. Ponieważ wyjaśnienia dotyczące szablonów wymagają użycia terminu „drzewo prezentacji”, w pierwszej kolejności zajmiemy się przedstawieniem tego zagadnienia.

## 11.1 Drzewo logiczne i drzewo prezentacji

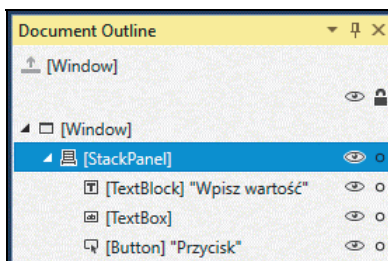
W WPF interfejs użytkownika ma budowę hierarchiczną, którą możemy zobrazować w postaci struktury drzewiastej. Spójrzmy na przykładowy kod XAML:

```
<Window x:Class="R11_VisualTree.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Przykład drzewa wizualnego" Height="200" Width="200">
    <StackPanel>
        <TextBlock Text="Wpisz wartość" Foreground="Gray" Margin="10"/>
        <TextBox Text="100" Margin="10"/>
        <Button Content="Przycisk" Margin="10"/>
    </StackPanel>
</Window>
```

**Drzewo logiczne** jest widoczne w kodzie XAML (o ile cały interfejs jest tu definiowany). Możemy zobaczyć je także w postaci rozwijanego konspektu w oknie *Document Outline* (opcja *View/Other Windows/Document Outline*), co przedstawia rysunek 11.1.

**Rysunek 11.1.**

Drzewo logiczne  
w oknie Document  
Outline



Korzeniem drzewa logicznego prezentowanej aplikacji jest okno, czyli element `Window`. W oknie znajduje się panel `StackPanel`, a w nim trzy elementy: `TextBlock`, `TextBox` i `Button`. Struktura drzewa logicznego wyznacza nam dostępność zasobów, o czym była mowa w poprzednim rozdziale. Ma także wpływ na mechanizm dziedziczenia właściwości.

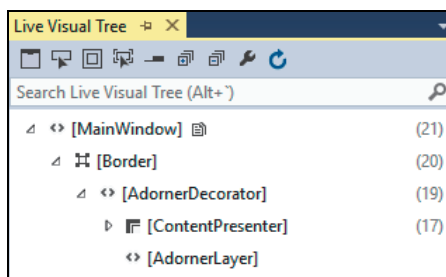
Drzewo logiczne nie pokazuje nam jednak wszystkich elementów potrzebnych do prezentacji poszczególnych komponentów interfejsu. Niektóre elementy widoczne w drzewie logicznym jako pojedyncze węzły mają tak naprawdę złożoną strukturę. Całe drzewo zawierające wszystkie elementy składowe nazywane jest **drzewem prezentacji** lub **drzewem wizualnym** (ang. *visual tree*).

Przeanalizujemy teraz drzewo wizualne prezentowanego programu. Utwórz nowy projekt WPF, w kodzie XAML umieść panel `StackPanel` na podstawie kodu z początku podrozdziału. Uruchom program z debuggerem. W podrozdziale 1.1 znajdują się zalecane ustawienia dla parametru *Enable UI Debugging Tools for XAML* w opcji *Tools/Options/Debugging/General*. Upewnij się, czy pozycja *Enable UI Debugging Tools for XAML* jest zaznaczona (czyli włączona). Jeżeli okno *Live Visual Tree* zostało zamknięte, można je wyświetlić przy użyciu opcji *Debug/Windows/Live Visual Tree*.

Po uruchomieniu programu w oknie *Live Visual Tree* powinna być widoczna zawartość podobna do prezentowanej na rysunku 11.2.

**Rysunek 11.2.**

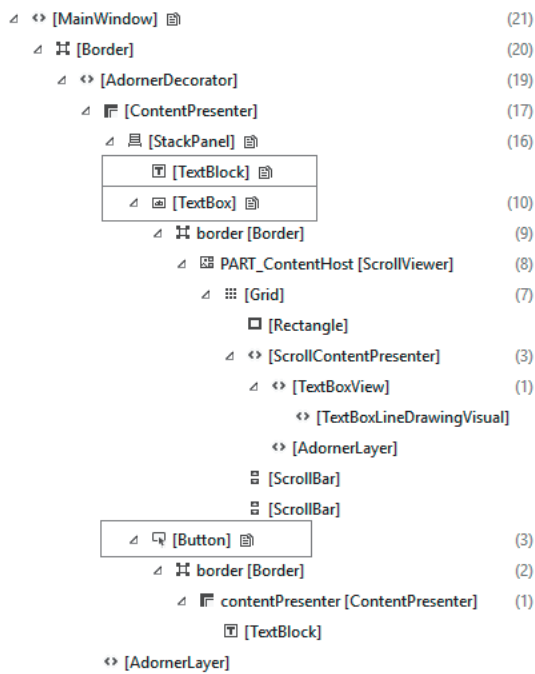
Okno Live Visual  
Tree po uruchomieniu  
aplikacji



Wśród przycisków dostępnych w górnej części okna jest kwadrat z plusem (*Expand All*), który należy kliknąć, aby rozwinąć wszystkie pozycje drzewa wizualnego (rysunek 11.3).

**Rysunek 11.3.**

*Okno Live Visual Tree  
po rozwinięciu pozycji  
drzewa*



Wartości w nawiasach okrągłych znajdujące się po prawej stronie węzłów drzewa informują o liczbie potomków danego węzła. Dodatkowe prostokąty na rysunku zostały dorysowane w celu lepszego uwidocznienia węzłów wynikających z drzewa logicznego, to znaczy elementów panelu `StackPanel`. Na podstawie budowy drzewa prezentacji widzimy, że `TextBlock` jest węzłem atomowym, czyli podobnie jak w drzewie logicznym, tak i tu nie zawiera elementów podrzędnych. Natomiast dwa pozostałe składniki panelu (`TextBox` i `Button`) mają strukturę złożoną. I tak przykładowo przycisk składa się z ramki (`Border`) oraz zawartości (`ContentPresenter`), domyślnie definiowanej jako tekst, czyli element `TextBlock`.

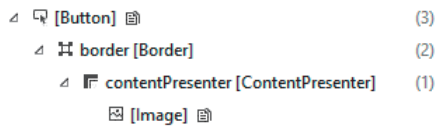
Jeśli zmienimy zawartość przycisku z tekstowej na rysunek:

```
<Button Margin="10">
    <Image Source="Rysunki/glosnik.png" MaxHeight="20" MaxWidth="20"/>
</Button>
```

to drzewo prezentacji dla tego przycisku będzie miało postać przedstawioną na rysunku 11.4.

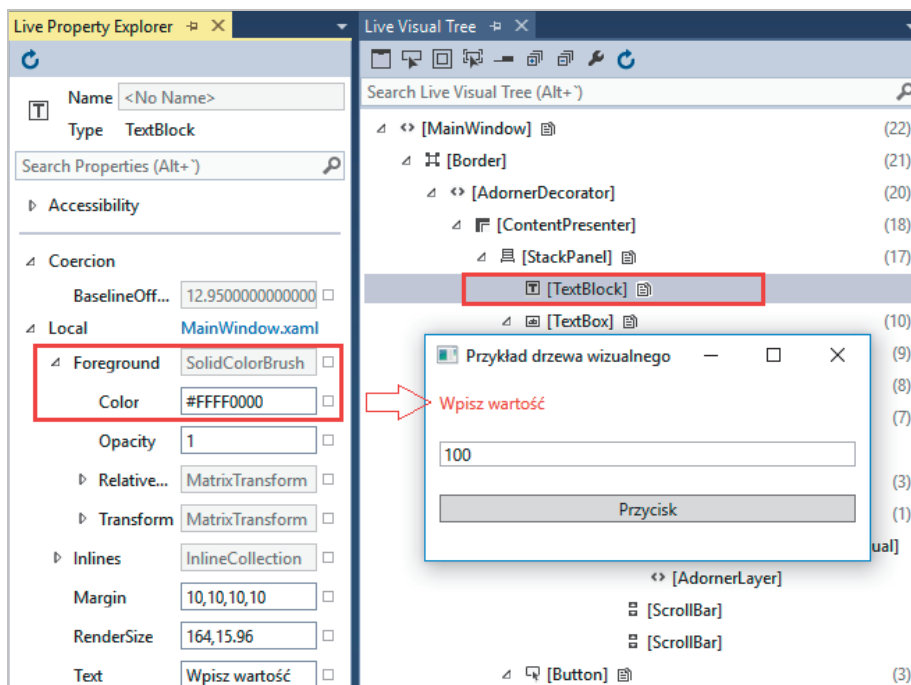
**Rysunek 11.4.**

*Drzewo wizualne  
dla przycisku  
z rysunkiem*



Przy okazji korzystania z okna *Live Visual Tree*, które wyświetla drzewo wizualne w trakcie działania aplikacji, zaprezentuję możliwość, jaką daje ono we współpracy z oknem *Live Property Explorer*. Przywróć poprzednią zawartość programu z tekstowym

przyciskiem, a następnie uruchom program. W oknie *Live Visual Tree* kliknij przycisk z kluczem *Show Properties* (lub wybierz opcję *Debug/Windows/Live Property Explorer*). Następnie rozwiń całe drzewo w oknie *Live Visual Tree*, odszukaj węzeł dla *Text* → *Block* (pierwszy element w *StackPanel*) i zaznacz go w tym oknie. Wówczas w oknie *Live Property Explorer* pokażą się właściwości tego elementu. Zmień jedną z nich, na przykład kolor czcionki. W tym celu rozwiń pozycję dla koloru czcionki *Foreground*, wpisz w polu *Color* nową wartość (np. *Red* lub szesnastkowy kod koloru *#FFFF0000*). Po tej zmianie spójrz na okno z uruchomioną aplikacją — kolor czcionki tekstu „Wpisz wartość” zmieni się z szarego na czerwony (rysunek 11.5). Zmianę definicji elementów w uruchomionej aplikacji można zrobić zmieniając kod XAML, o ile zaznaczono opisany w podrozdziale 1.1. parametr *Enable XAML Edit and Continue*, który pozwala uwidoczniać efekty zmiany kodu XAML w uruchomionej aplikacji.



**Rysunek 11.5.** Użycie okna *Live Property Explorer* — zmiana właściwości dla koloru czcionki jest od razu widoczna w uruchomionej aplikacji

Pracę tę można dodatkowo usprawnić za pomocą przycisku *Enable selection in the running application* (a także innych przycisków dostępnych w górnej części okna *Live Visual Tree*). Jeśli jest włączony, to wskazane myszą komponenty w oknie aplikacji otaczane są czerwoną ramką i po kliknięciu danego elementu zostaną wyświetlone informacje o tym elemencie w obu oknach — z drzewem wizualnym i właściwościami.

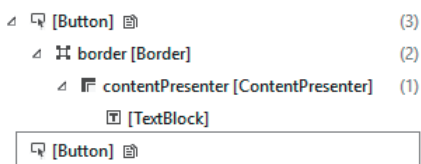
Wracamy do głównego tematu rozdziału, czyli szablonów. Przeprowadzimy ostatni eksperyment z naszym programem. Do bazowej wersji przykładu w tym podrozdziale (z tekstowym przyciskiem) dodaj w kodzie XAML wewnątrz panelu *StackPanel* nowy przycisk:

```
<Button Content="Widmo">
  <Button.Template>
    <ControlTemplate></ControlTemplate>
  </Button.Template>
</Button>
```

Po uruchomieniu programu z debuggerem zobaczymy w drzewie prezentacji „pusty” węzeł dla nowego przycisku, widoczny w dolnej części rysunku 11.6.

### Rysunek 11.6.

*Drzewo wizualne dla dwóch przycisków. Drugi przycisk (otoczony ramką) zawiera pusty szablon*



W oknie uruchomionej aplikacji widzimy, że przycisk się nie pokazuje, choć ma wpisaną zawartość (`Content="Widmo"`). Spójrzmy na kod tego przycisku. Przypisując do właściwości `Template` pusty szablon, `<ControlTemplate></ControlTemplate>`, usunęliśmy drzewo prezentacji dla tego przycisku i dopóki nie podmienimy na inne drzewo, przycisk *Widmo* nie pokaże się w aplikacji.

W tym rozdziale będziemy mówić o dwóch rodzajach szablonów: szablonach danych i szablonach kontroltek. Oba rodzaje szablonów pozwalają ingerować w domyślne drzewo prezentacji.

## 11.2 Szablony danych — aplikacja Lista zadań

Szablony danych pozwalają określić sposób wyświetlania poszczególnych części obiektu źródłowego, dla którego (jako całości) zdefiniowano wiązanie (*Binding*). Wiele kontroltek WPF ma właściwość typu `DataTemplate`, która pozwala dołączyć odpowiedni szablon danych.

Na stronie MSDN<sup>1</sup> jest czytelny przykład, obrazujący wykorzystanie szablonu danych w kontrolce `ListBox`. Wykonamy własną wersję tego programu. W programie wykorzystamy klasę `ObservableCollection`. Całą kolekcję zdefiniujemy w klasie kolekcji i jej zawartość wskażemy w zasobach.

Utwórz nowy projekt WPF. Ustaw tytuł na `Title="Lista zadań"`, a wymiary okna na przykład na `Height="200" Width="420"`.

<sup>1</sup> Przykład użycia szablonu danych („Data Templating Overview”): [https://msdn.microsoft.com/en-us/library/ms742521\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms742521(v=vs.110).aspx).

Zdefiniuj w pliku *Zadanie.cs* klasę:

```
class Zadanie
{
    public string Nazwa { get; set; }
    public string Opis { get; set; }
    public int Priorytet { get; set; }
}
```

Następnie w nowym pliku utwórz klasę *KolekcjaZadan*. Dodaj w pliku dla tej klasy dyrektywę dla przestrzeni nazw:

```
using System.Collections.ObjectModel;
```

i wpisz definicję klasy *KolekcjaZadan*:

```
class KolekcjaZadan : ObservableCollection<Zadanie>
{
    public KolekcjaZadan()
    {
        Add(new Zadanie
        {
            Nazwa = "Zamówienie",
            Opis = "Zamówić 100 długopisów żelowych",
            Priorytet = 1
        });
        Add(new Zadanie
        {
            Nazwa = "Zaproszenie",
            Opis = "Zaprosić kontrahentów na pokaz nowego produktu",
            Priorytet = 2
        });
        Add(new Zadanie
        {
            Nazwa = "Sprząatanie",
            Opis = "Posprzątać magazyn",
            Priorytet = 3
        });
    }
}
```

Klasa *KolekcjaZadan* dziedziczy po kolekcji *ObservableCollection<Zadanie>*, co oznacza, że będzie kolekcją (zbiorem obiektów). W konstruktorze klasy znajdują się trzy wywołania metody *Add*, dodające przykładowe obiekty do kolekcji<sup>2</sup>.

W kodzie klasy *MainWindow* tym razem nic nie zmieniamy. Pozostaje zawartość wpisana automatycznie.

Wracamy do kodu XAML. Wpisz tam kod (wewnątrz lub zamiast znaczników *Grid*):

---

<sup>2</sup> Jeżeli nie znasz użytej tu konstrukcji do inicjalizacji właściwości/pól obiektów, przejrzyj przykłady na stronie („Object and Collection Initializers”): <https://msdn.microsoft.com/en-us/library/bb384062.aspx>.

```

<StackPanel>
  <StackPanel.Resources>
    <local:KolekcjaZadan x:Key="listaZadanDoWykonania"/>
  </StackPanel.Resources>
  <TextBlock FontSize="14" Text="Lista zadań:" Margin="10,0"/>
  <ListBox Margin="10" ItemsSource="{Binding Source={StaticResource
    listaZadanDoWykonania}}"/>
</StackPanel>

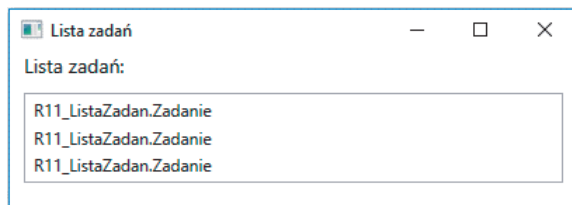
```

W panelu definiowany jest jako zasób dostęp do kolekcji `KolekcjaZadan`. Zasób ten jest wskazywany jako źródło podczas wiązania danych dla `ListBox`.

Uruchomimy program w obecnej wersji. Prezentacja danych listy raczej nas nie zadowoli, ponieważ wyświetlają się jedynie nazwy klasy obiektów, jakie są w kolekcji (rysunek 11.7).

### Rysunek 11.7.

*Lista zadań  
w pierwszej wersji  
— brak informacji  
o zadaniach*



Możemy dodać do klasy `Zadanie` nadpisanie metody `ToString`, przykładowo w postaci:

```

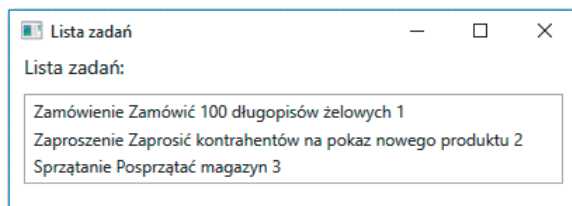
public override string ToString()
{
    return String.Format("{0} {1} {2}", Nazwa, Opis, Priorytet);
}

```

Wówczas lista zadań będzie wyglądała znacznie lepiej, co widzimy na rysunku 11.8.

### Rysunek 11.8.

*Lista zadań  
w drugiej wersji  
— z informacjami  
o zadaniach*



Wprawdzie moglibyśmy jeszcze dopracować prezentację danych przy użyciu `String.Format` w metodzie `ToString` (np. dodać znak nowej linii pomiędzy właściwościami), ale jeżeli zależałoby nam na tym, aby poszczególne właściwości klasy `Zadanie` były wyświetlane w różny sposób (np. w innym kolorze), to już musimy sięgnąć do szablonu danych.

Możesz nieznacznie zwiększyć wysokość okna aplikacji. Następnie podmień definicję kontrolki `ListBox` na następującą:

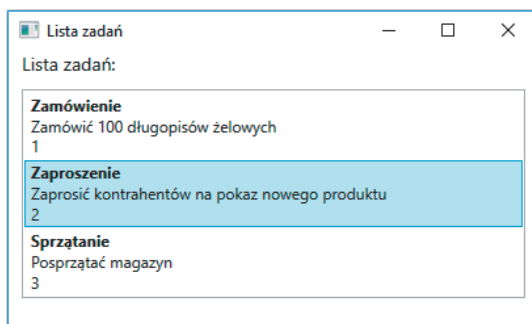
```

<ListBox Margin="10" ItemsSource="{Binding Source={StaticResource
    listaZadanDoWykonania}}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding Path=Nazwa}" FontWeight="Bold"/>
        <TextBlock Text="{Binding Path=Opis}" />
        <TextBlock Text="{Binding Path=Prioritytet}" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>

```

W obecnej wersji programu, dzięki użyciu szablonu danych, mamy możliwość określenia sposobu wyświetlania poszczególnych właściwości klasy `Zadanie`. Dla właściwości `ItemTemplate` klasy `ListBox` przypisano szablon zawierający panel z trzema elementami `TextBlock`. Nazwa zadania zostanie wyświetlona czcionką pogrubioną (rysunek 11.9).

**Rysunek 11.9.**  
*Lista zadań*  
 w kolejnej wersji  
 — z użyciem  
 szablonu danych



Możemy cały szablon zdefiniować jako zasób, a następnie przypisać go do właściwości `ItemTemplate` (lub innej typu `DataTemplate`). W omawianym programie mogłoby to wyglądać następująco (jeśli chcesz przetestować ten wariant, podmień cały panel `StackPanel`):

```

<StackPanel>
  <StackPanel.Resources>
    <local:KolekcjaZadan x:Key="listaZadanDoWykonania"/>
    <DataTemplate x:Key="szablonZadan">
      <StackPanel>
        <TextBlock Text="{Binding Path=Nazwa}" FontWeight="Bold" />
        <TextBlock Text="{Binding Path=Opis}" />
        <TextBlock Text="{Binding Path=Prioritytet}" />
      </StackPanel>
    </DataTemplate>
  </StackPanel.Resources>
  <TextBlock FontSize="14" Text="Lista zadań:" Margin="10,0"/>
  <ListBox Margin="10" ItemsSource="{Binding Source={StaticResource
    listaZadanDoWykonania}}" ItemTemplate="{StaticResource szablonZadan}" />
</StackPanel>

```

Program w obu prezentowanych wersjach będzie działał tak samo.



Na koniec podrozdziału spójrzmy jeszcze na szablon użyty we wskazówkach do zadania z podrozdziału 7.6:

```
<DataGridTemplateColumn Header="Zdjęcie" MaxWidth="50" IsReadOnly="True">
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <Image Source="{Binding Path=Zdjecie}" />
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>
```

W tym przypadku szablon został zastosowany w komórce elementu `DataGrid`, przypisano go do właściwości `CellTemplate`. Szablon ten był potrzebny do wyświetlenia zdjęcia. `DataGrid` oferuje kilka typów kolumny: tekstowy, dla hiperłączy, dla danych typu `bool`, dla listy rozwijanej oraz typ „uniwersalny” `DataGridTemplateColumn`, który pozwala zdefiniować dowolny szablon prezentowania danych (przy użyciu właściwości `CellTemplate` i `CellEditingTemplate`), co już wyjaśniłam w podrozdziale 7.2.

## 11.3 Konwertery wartości

Mechanizm wiązania danych czasami nie wystarcza. Przykładowo obiekt źródłowy może być dostępny w innej postaci, niż jest potrzebna w obiekcie docelowym. Nie musimy w takich przypadkach rezygnować z wygody wiązania danych, wystarczy użyć w tym celu konwerterów wartości.

W podrozdziale 5.5, we wskazówkach do jednego z zadań, zaleciłam, aby do listy rozwijanej wpisać angielskie nazwy kolorów. Testowaliśmy wówczas wiązanie danych. Użytkownik miał możliwość zmiany rozmiaru tekstu, koloru oraz zawartości. Teraz wykonamy podobne zadanie, tym razem jednak użytkownik będzie zmieniał tylko kolor. Kolory zostaną zapisane w języku polskim.

Zacniemy od stworzenia konwertera. Konwerter jest klasą, która musi implementować interfejs `IValueConverter`. Interfejs ten wymaga definicji dwóch metod:

- ♦ `Convert`, która dokonuje konwersji obiektu źródłowego do obiektu docelowego dla podanego typu;
- ♦ `ConvertBack`, która wykonuje operację odwrotną.

Stwórz w projekcie w nowym pliku klasę `ColorPlToColorEnConverter`. Dodaj dyrektywy `using`:

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

Następnie wpisz kod klasy:

```
class ColorPlToColorEnConverter: IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
```

```

{
    if (targetType != typeof(Brush))
    {
        throw new InvalidOperationException("Celem powinien być typ Brush");
    }
    string kolorPL = value.ToString();    // Kolor w języku polskim jako string
    Dictionary<string, Brush> kolory = new Dictionary<string, Brush>();
    kolory.Add("Czarny", Brushes.Black);
    kolory.Add("Czerwony", Brushes.Red);
    kolory.Add("Żółty", Brushes.Yellow);
    kolory.Add("Zielony", Brushes.Green);
    kolory.Add("Niebieski", Brushes.Blue);
    if (kolory.ContainsKey(kolorPL))    // Czy dany kolor jest w słowniku
        return kolory[kolorPL];        // Zwraca kolor jako Brush
    else
        return Brushes.LightGray;
}
public object ConvertBack(object value, Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
{
    throw new NotImplementedException();
}
}

```

Ponieważ w naszym programie ma miejsce wiązanie jednostronne (*OneWay*), implementacja dotyczy przede wszystkim metody *Convert*. Kolor podany w języku polskim jako *string* zostanie przekonwertowany do koloru typu *Brush*. Kolory zostały wpisane do słownika (kolekcji *Dictionary*). Gdyby w programie było więcej odwołań do kolorów, należałoby ten słownik zdefiniować w innym miejscu.

W kodzie XAML wpisz tytuł i rozmiary okna, na przykład *Title="Testujemy konwerter"* *Height="150"* *Width="500"*. Dodaj zasób z konwerterem dla okna:

```

<Window.Resources>
    <local:ColorPLToColorEnConverter x:Key="colorPLToColorEnConverter"/>
</Window.Resources>

```

Następnie w znacznikach *Grid* wpisz kod:

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*/>
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.Column="0" Content="Kolor" Margin="5"/>
    <ComboBox x:Name="cmbKolor" Grid.Row="0" Grid.Column="1"
        SelectedIndex="0" HorizontalAlignment="Left" Width="145" Margin="5">
        <ComboBoxItem Content="Czarny"></ComboBoxItem>
        <ComboBoxItem Content="Czerwony"></ComboBoxItem>
        <ComboBoxItem Content="Żółty"></ComboBoxItem>
    </ComboBox>

```

```

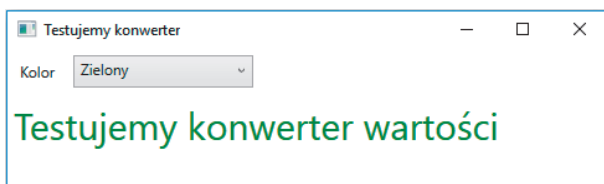
<ComboBoxItem Content="Zielony"></ComboBoxItem>
<ComboBoxItem Content="Niebieski"></ComboBoxItem>
</ComboBox>
<TextBlock Grid.Row ="1" Grid.Column ="0" Grid.ColumnSpan="2" Margin="5"
    Text="Testujemy konwerter wartości" FontSize="30"
    Foreground="{Binding Path=SelectedItem.Content, ElementName=cmbKolor,
        Converter={StaticResource colorPIToColorEnConverter}}"/>
</Grid>

```

Dla koloru tekstu elementu `TextBlock` przypisano wiązanie z listą rozwijaną wraz ze zdefiniowanym konwerterem, udostępnianym przez zasób `colorPIToColorEnConverter`. Przykładowe działanie programu przedstawia rysunek 11.10.

**Rysunek 11.10.**

*Test aplikacji  
z konwerterem  
dla nazw kolorów*



Uważny Czytelnik może zapytać, dlaczego nie była potrzebna konwersja w wersji programu z angielskimi nazwami kolorów. Przecież łańcuch znakowy `"Black"` to nie to samo co właściwość `Brushes.Black` typu `SolidColorBrush`. Nie musieliśmy wówczas robić konwersji, ponieważ WPF ma wbudowane **konwertery typów**, dzięki czemu definicje w kodzie XAML takie jak poniższa:

```

<Button Content="Zapisz" FontSize="20" Foreground="Blue"/>

```

nie wymagają określenia typu dla właściwości, które nie są typu `string`, jak na przykład rozmiar czcionki czy jej kolor.

## 11.4 Szablony kontrolki

Poznane dotąd techniki (style, wyzwalacze) pozwalają zmienić wizualizację kontrolki, ale tylko w pewnym zakresie, to znaczy podstawowy wygląd kontrolki pozostaje bez zmian. Przykładowo bez względu na to, w jaki sposób zmienimy standardowe właściwości przycisku, nadal jest to prostokątny przycisk, który pokazuje określoną zawartość. WPF pozwala na jeszcze większą elastyczność w zakresie modyfikacji wyglądu kontrolki poprzez zastosowanie **szablonów kontrolki**. Zamiast ustawiania poszczególnych właściwości można dla danej kontrolki zastąpić drzewo prezentacji, zachowując przy tym jej podstawową funkcjonalność. W ten sposób możemy zdefiniować owalny przycisk czy pasek postępu w kształcie koła.

Wykonamy prosty program z szablonem kontrolki dla przycisku. W nowym projekcie WPF zmien tytuł okna i jego wymiary na przykład na `Title="Szablon przycisku"` `Height="150"` `Width="350"`. Następnie wpisz kod definicji szablonu przycisku jako zasób okna:

```

<Window.Resources>
    <ControlTemplate x:Key="szablonPrzycisku" TargetType="Button">
        <Grid>
            <Ellipse Fill="Lavender" Width="100" Height="50" Stroke="Black"/>
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
                Text="Zapisz" FontSize="16" FontWeight="Bold"/>
        </Grid>
    </ControlTemplate>
</Window.Resources>

```

W komórce panelu `Grid` zdefiniujemy przycisk według szablonu `szablonPrzycisku`:

```

<Grid>
    <Button Template="{StaticResource szablonPrzycisku}"
        Click="Button_Click"/>
</Grid>

```

W *code-behind*, w klasie `MainWindow`, dopisz obsługę zdarzenia kliknięcia przycisku:

```

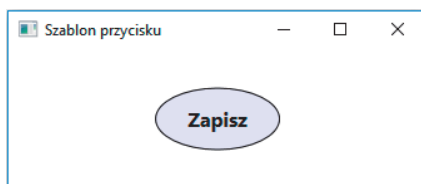
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Zapisano dane!");
}

```

Program możemy uruchomić i przyjrzeć się nowemu przyciskowi (rysunek 11.11).

### Rysunek 11.11.

*Test szablonu  
przycisku*



Przycisk mimo zmiany wyglądu zachował swoją standardową funkcjonalność — możemy go klikać i uruchamiać działanie zdefiniowane dla zdarzenia `Click`.

Obecna postać szablonu ma jednak pewne ograniczenia, mianowicie nie pozwala na zmianę właściwości przycisku określonych w szablonie. Nie możemy zmienić zawartości przycisku (`Content`) i rozmiaru czcionki tekstu poprzez prostą definicję:

```

<Button Content="Zatwierdź" Template="{StaticResource szablonPrzycisku}"
    Click="Button_Click" FontSize="30"/>

```

Program nie pokaże nowej nazwy przycisku *Zatwierdź*, wciąż będzie na przycisku tekst, jaki wynika z szablonu, czyli *Zapisz*. Rozmiar czcionki `FontSize="30"` także nie zostanie uwzględniony, tekst będzie miał rozmiar wynikający z szablonu, czyli `16`. Możemy takie zachowanie szablonu zmienić za pomocą klasy `ContentPresenter`. Podmień kod XAML na następujący:

```

<Window.Resources>
    <ControlTemplate x:Key="szablonPrzycisku" TargetType="Button">
        <Grid>
            <Ellipse Fill="Lavender" Width="100" Height="50" Stroke="Black"/>

```

```

        <ContentPresenter HorizontalAlignment="Center"
                        VerticalAlignment="Center"/>
    </Grid>
</ControlTemplate>
</Window.Resources>
<Grid>
    <Button Content="Zatwierdź" FontSize="16" FontWeight="Bold"
            Template="{StaticResource szablonPrzycisku}" Click="Button_Click"/>
</Grid>

```

Przeanalizujemy zmiany. Obecnie w szablonie przycisku zamiast elementu `TextBlock` jest `ContentPresenter`. `ContentPresenter` prezentuje wartość, która jest związana z właściwością `Content` kontrolki `Button` (lub innej dziedziczącej po klasie `ContentControl`). Nie widzimy w kodzie tego wiązania, ponieważ został użyty skrócony zapis z wbudowanym niejawnie wiązaniem `Content={TemplateBinding Content}`. Natomiast w definicji przycisku `Button` mamy teraz ustawienie właściwości `Content` oraz rozmiaru czcionki i jej grubości. Właściwość `Content` mogłaby zostać, w końcu przyciski w aplikacji, nawet jeśli mają w pełni ujednolicony wygląd, to różnią się właśnie zawartością. Pozostałe jednak właściwości, których może być znacznie więcej, należałoby ustawić w stylu.

Wykonamy kolejną wersję tego programu, tym razem z szablonem umieszczonym w stylu. Wpisz kod zasobu:

```

<Window.Resources>
    <Style x:Key="stylPrzycisku" TargetType="{x:Type Button}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="Button">
                    <Grid>
                        <Ellipse Fill="Lavender" Width="100" Height="50"
                                Stroke="Black"/>
                        <ContentPresenter HorizontalAlignment="Center"
                                VerticalAlignment="Center"/>
                    </Grid>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
        <Setter Property="FontSize" Value="16"/>
        <Setter Property="FontWeight" Value="Bold"/>
    </Style>
</Window.Resources>

```

Szablon został umieszczony w stylu i opisuje kształt przycisku, czego nie mogliśmy zrobić tradycyjnie (poprzez standardowe właściwości kontrolki). Natomiast właściwości przycisku (`FontSize` i `FontWeight`) zostały opisane w stylu (poza szablonem). Definicja samego przycisku może wyglądać tak:

```

<Grid>
    <Button Content="Zatwierdź" Style="{StaticResource stylPrzycisku}"
            Click="Button_Click"/>
</Grid>

```

Definiowany przez nas wygląd przycisku (zwykła elipsa) nie był wyszukany. Podczas definiowania szablonów kontrolki można wspomóc się narzędziami graficznymi, takimi jak Microsoft Blend, które pozwalają uzyskać znacznie ciekawsze efekty<sup>3</sup>.

Obok bardziej złożonych szablonów pomijamy tu kwestie tak zwanych skórek i motywów. WPF nie ma specjalnego mechanizmu przewidzianego dla skórek, ale można uzyskać taką funkcjonalność przy użyciu dynamicznych zasobów, o których jest wzmianka w podrozdziale 10.2. Jeśli chodzi o motywy, warto byłoby poznać na dalszym etapie nauki sposoby zachowania spójności między wyglądem aplikacji a wyglądem systemu operacyjnego<sup>4</sup>.

## 11.5 Zadania

Przedstawione zadania dotyczą programów napisanych w bieżącym rozdziale i w poprzednim.

### Zadanie 11.1

Dodaj do szablonu danych w programie z podrozdziału 11.2 (z listą zadań) wyzwalacz danych, który dla zadań mających priorytet równy 1 (najwyższy) ustawi kolor czcionki na **Red** (czerwony).

### Zadanie 11.2

Dodaj do szablonu w programie z podrozdziału 11.2 (z listą zadań) konwerter, który dla zadań mających priorytet równy 1 (najwyższy) ustawi kolor czcionki dla nazwy zadania (samej nazwy) na **Red**.

### Zadanie 11.3

W programie z rozdziału 11.4 z szablonem przycisku (w wersji bez stylu) zmień szablon, dodając do niego dwa wyzwalacze: jeden ma zmienić kolor tła elipsy na jasnoniebieski w chwili najechania myszą na obszar przycisku (**IsMouseOver**), a drugi ma zmienić kolor tła przycisku na czerwony w momencie, gdy przycisk zostaje wciśnięty (**IsPressed**).

---

<sup>3</sup> Przykład tworzenia szablonu przycisku z użyciem starszej wersji programu Microsoft Expression Blend („Create a Button by Using Microsoft Expression Blend”): [https://msdn.microsoft.com/en-us/library/bb613598\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/bb613598(v=vs.110).aspx).

<sup>4</sup> Skórki i motywy zostały opisane m.in. w: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 462 – 473).

## 11.6 Wskazówki do zadań

W bieżących zadaniach nie ma nic wymagającego dodatkowych wyjaśnień względem zawartości tego rozdziału i poprzedniego. Jeżeli chcemy dodać wyzwalacz do szablonu danych, korzystamy z właściwości `DataTemplate.Triggers` (jak w zadaniu 11.1). W przypadku dodawania wyzwalacza do szablonu kontrolki korzystamy z właściwości `ControlTemplate.Triggers` (jak w zadaniu 11.3).

### Wskazówki do zadania 11.1

Na samym początku definicji szablonu można napisać kod wyzwalacza:

```
<DataTemplate.Triggers>
  <DataTrigger Binding="{Binding Path=Priorytet}" Value="1">
    <Setter Property="TextBlock.Foreground" Value="Red"/>
  </DataTrigger>
</DataTemplate.Triggers>
```

### Wskazówki do zadania 11.2

Wykorzystaj kod z podrozdziału 11.2 bez zmian wprowadzonych w zadaniu poprzednim. Utwórz nowy plik dla konwertera i dodaj dyrektywy `using`:

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

Kod konwertera może mieć następującą postać:

```
class PriorityToForegroundConverter: IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        if (targetType != typeof(Brush))
        {
            throw new InvalidOperationException("Celem powinien być typ Brush");
        }
        int priorytet = int.Parse(value.ToString());
        return (priorytet == 1 ? Brushes.Red : Brushes.Black);
    }
    public object ConvertBack(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

W kodzie XAML należy dodać konwerter do definicji zasobu, na przykład:

```
<local:PriorityToForegroundConverter x:Key="priorytetKonwerter"/>
```

Natomiast w definicji `TextBlock` dla nazwy zadania należy dokonać wiązania z właściwością `Priorytet` oraz przypisać konwerter:

```
<TextBlock Text="{Binding Path=Nazwa}" FontWeight="Bold" Foreground="{Binding
    Priorytet, Converter={StaticResource priorytetKonwerter}}" />
```

### Wskazówki do zadania 11.3

Wskazówki zostały podane w treści zadania (nazwy właściwości), tu umieszczam przykładowe rozwiązanie:

```
<Window.Resources>
    <ControlTemplate x:Key="szablonPrzycisku" TargetType="Button">
        <Grid>
            <Ellipse x:Name="elipsa" Fill="Lavender" Width="100" Height="50"
                Stroke="Black"/>
            <ContentPresenter HorizontalAlignment="Center"
                VerticalAlignment="Center"/>
        </Grid>
        <ControlTemplate.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
                <Setter Property="Fill" Value="LightBlue" TargetName="elipsa"/>
            </Trigger>
            <Trigger Property="IsPressed" Value="True">
                <Setter Property="Fill" Value="Red" TargetName="elipsa"/>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
</Window.Resources>
```

Definicja przycisku bez zmian, czyli:

```
<Button Content="Zatwierdź" FontSize="16" FontWeight="Bold"
    Template="{StaticResource szablonPrzycisku}" Click="Button_Click"/>
```



## Rozdział 12.

# Walidacja danych

Sprawdzanie poprawności danych wydaje się oczywistym etapem podczas tworzenia większości programów i nie poświęcalibyśmy temu tematowi osobnego rozdziału, gdyby nie fakt, że w kontekście wiązania danych walidacja wymaga specjalnego podejścia. W przypadku, gdy użytkownik wpisze nieprawidłowe wartości, oczekujemy odpowiednich reakcji programu, nie tracąc przy tym komfortu automatyzmu, jaki zapewnia wiązanie danych. WPF dostarcza w tym celu wbudowane mechanizmy obsługi błędów i umożliwia stworzenie własnych reguł walidacji.

## 12.1 Wbudowane mechanizmy walidacji

W tym podrozdziale Czytelnik pozna niektóre wbudowane mechanizmy walidacji danych. W podrozdziale 5.3 omówiłam wiązanie danych na przykładzie programu z klasą *Produkt*. W programie tym można było zaobserwować jeden z przejawów wbudowanego mechanizmu walidacji, mianowicie wpisanie w polu *Liczba sztuk* wartości innej niż liczba powodowało wyświetlenie ramki pola tekstowego na czerwono. Takie działanie zapewnia nam domyślny szablon. Wykonamy teraz prostszy wariant tego programu, zawierający mniej kontrolek, i skupimy się na walidacji danych.

W nowym projekcie WPF ustaw w kodzie XAML tytuł i rozmiary okna na przykład na: `Title="Walidacja danych (wersja 1)" Height="100" Width="400"`. Możesz także ustawić `Language="pl-PL"` (dla formatowania wartości liczbowej z przecinkiem dziesiętnym w polu tekstowym). Następnie wpisz kod:

```
<Window.DataContext>
    <local:Towar Nazwa="ořówek" Cena="21"/>
</Window.DataContext>
<WrapPanel>
    <Label Content="Nazwa:"/>
    <TextBox Margin="5" Text="{Binding Nazwa}" Width="100"/>
    <Label Content="Cena:"/>
    <TextBox Margin="5" Text="{Binding Cena, UpdateSourceTrigger=
        PropertyChanged, StringFormat={}{0:F2}}" Width="100"/>
</WrapPanel>
```

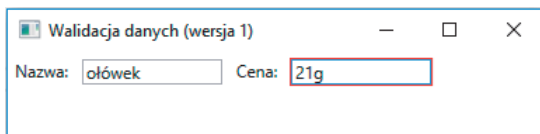
W programie tym ustawienie kontekstu dla wiązania danych zostało wykonane w kodzie XAML poprzez przypisanie obiektu klasy `Towar` do właściwości `DataContext` dla okna. Klasę `Towar` zdefiniuj w osobnym pliku:

```
class Towar
{
    public string Nazwa { get; set; }
    public double Cena { get; set; }
}
```

Program można uruchomić. Po wpisaniu wartości tekstowej w polu `Cena` ramka pola tekstowego wyświetli się w kolorze czerwonym (rysunek 12.1).

### Rysunek 12.1.

*Automatyczne wyróżnianie pola z błędną wartością ceny (niezgodną z typem właściwości `Cena`)*



Przedstawiane w tym rozdziale techniki walidacji dotyczą zmian przechodzących z elementów docelowych do źródła danych. W podrozdziale 5.5 pisałem o sposobach aktualizacji wiązanych danych od celu do źródła. Decyduje o tym właściwość `UpdateSource` → `Trigger`. Domyślnym ustawieniem sposobu aktualizacji dla pola tekstowego jest `LostFocus`. W programie zostało to zmienione na `PropertyChanged`, dzięki czemu uzyskamy sygnalizację błędnej wartości od razu po wpisaniu znaku, który uniemożliwia konwersję do typu `double` (bo taki typ ma właściwość `Cena`)<sup>1</sup>.

## Komunikat o błędzie jako `ToolTip`

Wyróżnienie pola czerwoną ramką nie informuje jednak użytkownika o rodzaju błędu. Informacja o tym błędzie jest dostępna w programie, musimy ją jedynie odczytać i zaprezentować we wskazanym miejscu. W pierwszej kolejności wyświetlimy komunikat o błędzie w postaci „dymku”, czyli z użyciem właściwości `ToolTip` dla pola tekstowego. Wykorzystamy w tym celu wyzwalacz. Na początku definicji panelu `WrapPanel` dopisz kod:

```
<WrapPanel.Resources>
    <Style TargetType="TextBox">
        <Style.Triggers>
            <Trigger Property="Validation.HasError" Value="true">
                <Setter Property="ToolTip" Value="{Binding
                    RelativeSource={x:Static RelativeSource.Self},
                    Path=(Validation.Errors).CurrentItem.ErrorContent}" />
            </Trigger>
        </Style.Triggers>
    </Style>
</WrapPanel.Resources>
```

<sup>1</sup> Takie działanie będzie wygodniejsze podczas testowania walidacji. W programach pisanych dla użytkowników lepszym rozwiązaniem jest zazwyczaj to z domyślnym ustawieniem, powodujące reakcję dopiero po utracie fokusu, np. poprzez kliknięcie innego pola. Przewaga aktualizacji w trybie `LostFocus` polega głównie na nierozpraszaniu użytkownika komunikatami, gdy ten jeszcze nie ukończył wprowadzania danej wartości.

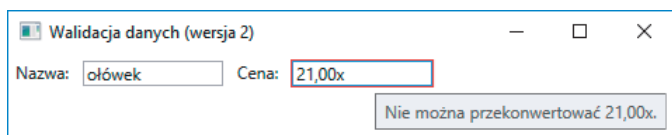
```

</Style.Triggers>
</Style>
</WrapPanel.Resources>

```

Po dodaniu wyzwalacza można już testować nową wersję programu. Wpisanie wartości, która nie może być konwertowana do typu `double`, spowoduje wyświetlenie ramki na czerwono, a ponadto najechanie myszą w obszar pola tekstowego wyświetli „dymek” z komunikatem o błędzie (rysunek 12.2).

**Rysunek 12.2.**  
Wyświetlenie komunikatu  
o błędzie przy użyciu  
ToolTip



## Definicja szablonu dla Validation.ErrorTemplate

Wykonamy kolejną wersję programu, tym razem komunikat o błędzie zostanie wyświetlony za pomocą szablonu zdefiniowanego dla właściwości `Validation.ErrorTemplate`. Usuń lub zakomentuj zasób z wyzwalaczem zdefiniowany w poprzedniej wersji programu. W kolejnym kroku podmień definicję pola tekstowego dla ceny na następującą:

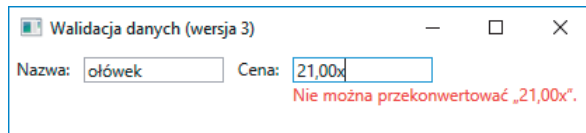
```

<TextBox Text="{Binding Cena, UpdateSourceTrigger=PropertyChanged,
    StringFormat={}{0:F2}}" Margin="5" Width="100">
    <Validation.ErrorTemplate>
        <ControlTemplate>
            <StackPanel>
                <AdornedElementPlaceholder/>
                <TextBlock Text="{Binding CurrentItem.ErrorContent}"
                    Foreground="Red"/>
            </StackPanel>
        </ControlTemplate>
    </Validation.ErrorTemplate>
</TextBox>

```

Zdefiniowanie własnego szablonu pozwala umiejscowić pewne dodatkowe elementy, takie jak kolorowe ramki, wykrzykniki czy tekst komunikatu, względem kontrolki z błędną wartością. `AdornedElementPlaceholder` określa miejsce, w którym znajduje się element z walidowaną wartością (tu ceną) względem innych elementów szablonu. W prezentowanym szablonie jest tylko jeszcze jeden element, mianowicie komunikat o błędzie, który wyświetli się pod walidowaną wartością (rysunek 12.3).

**Rysunek 12.3.**  
Wyświetlenie komunikatu  
o błędzie konwersji przy  
użyciu zdefiniowanego  
szablonu



W tej wersji programu ramka pola tekstowego nie została wyróżniona na czerwono tak, jak to było w domyślnej wersji szablonu. Jeżeli chcielibyśmy ramkę wyróżnić, należałoby ją narysować wokół elementu `AdornedElementPlaceholder`, na przykład:

```
<Border BorderBrush="Red" BorderThickness="1" Width="100">
  <AdornedElementPlaceholder/>
</Border>
```

Oczywiście nie ma przeszkód, aby szablon walidacji umieścić w definicji stylu. I taki wariant zostanie zastosowany w kolejnym przykładzie.

## Implementacja interfejsu IDataErrorInfo

Sprawdzenie typu danej to tylko część przeprowadzanej walidacji. Przeważnie w programach sprawdza się, w zależności od typu i znaczenia danej, liczbę cyfr, zakres liczbowy, wielkość liter czy dopuszczalne znaki. Jednym z rozwiązań WPF pozwalających na walidację z określeniem warunków jest użycie klasy `DataErrorValidationRule`. Skorzystanie z tego mechanizmu wymaga implementacji interfejsu `IDataErrorInfo`.

W nowym projekcie stwórz plik dla klasy `Towar.cs` i dodaj tam polecenie z przestrzenią nazw:

```
using System.ComponentModel;
```

Następnie zdefiniuj klasę `Towar`:

```
class Towar : IDataErrorInfo
{
    public string Nazwa { get; set; }
    public double Cena { get; set; }

    public string Error // „Pusta” implementacja właściwości Error
    {
        get
        {
            throw new NotImplementedException();
        }
    }

    public string this[string nazwaWlasciwosciTowaru] // Implementacja
                                                    // indeksatora
    {
        get
        {
            string komunikat = String.Empty;
            switch (nazwaWlasciwosciTowaru)
            {
                case "Nazwa":
                    if (string.IsNullOrEmpty(Nazwa))
                        komunikat = "Nazwa musi być wpisana!";
                    else if (Nazwa.Length < 3)
                        komunikat = "Nazwa musi mieć minimum 3 znaki!";
                    break;
                case "Cena":
                    if ((Cena < 0.1) || (Cena > 1000))
                        komunikat = "Cena musi być z zakresu od 0,10 do 1000";
                    break;
            }
        }
    }
}
```

```

    };
    return komunikat;
}
}
}

```

Pierwsze dwie linie klasy zawierają właściwości `Nazwa` i `Cena`, podobnie jak to miało miejsce w poprzednim programie. Reszta kodu w tej klasie stanowi implementację interfejsu `IDataErrorInfo`, zawierającego właściwość `Error` i indeksator. Właściwość `Error` została przewidziana dla błędów, które dotyczą całego obiektu (tu nie wykorzystamy tej możliwości). Indeksatory (definiowane przy użyciu słowa kluczowego `this`) pozwalają indeksować instancje klasy podobnie jak tablicę. Składniowo indeksatory przypominają zarówno tablice, jak i właściwości<sup>2</sup>. Użyty w omawianym programie indeksator jest indeksowany nazwami właściwości klasy. W przypadku wprowadzenia błędnej wartości zwrócony zostanie odpowiedni komunikat o błędzie. Spójrzmy na warunki walidacji dla obu właściwości. W przypadku właściwości `Nazwa` zakładamy, że wartość musi być wpisana i ponadto ma mieć minimum 3 znaki. Natomiast właściwość `Cena` ma być liczbą z zakresu od 0,1 do 1000 zł.

W kodzie XAML ustaw tytuł, rozmiary okna i język na: `Title="Walidacja danych (wersja 4)" Height="100" Width="450" Language="pl-PL"`. Następnie wpisz kod:

```

<Window.DataContext>
    <local:Towar Nazwa="ołów" Cena="21"/>
</Window.DataContext>
<Window.Resources>
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="Margin" Value="5"/>
        <Setter Property="Width" Value="100"/>
        <Setter Property="Validation.ErrorTemplate">
            <Setter.Value>
                <ControlTemplate>
                    <StackPanel>
                        <AdornedElementPlaceholder/>
                        <TextBlock Text="{Binding CurrentItem.ErrorContent}"
                                Foreground="Red"/>
                    </StackPanel>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
<WrapPanel>
    <Label Content="Nazwa:"/>
    <TextBox Text="{Binding Nazwa, UpdateSourceTrigger=PropertyChanged,
        ValidatesOnDataErrors=True}"/>
    <Label Content="Cena:"/>
    <TextBox Text="{Binding Cena, UpdateSourceTrigger=PropertyChanged,
        StringFormat={}, ValidatedOnDataErrors=True}"/>
</WrapPanel>

```

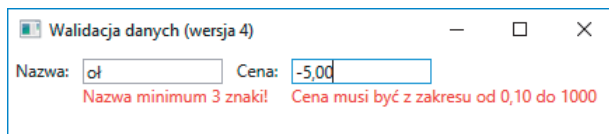
<sup>2</sup> Indeksatory („Using Indexers”): <https://msdn.microsoft.com/en-us/library/2549tw02.aspx>.

Szablon został umieszczony w stylu dla `TextBox` jako zasób okna. Zawartość samego szablonu względem poprzedniego programu nie uległa zmianie. W definicji wiązania dla obu pól tekstowych właściwość `ValidatesOnDataErrors` została ustawiona na `True`, co oznacza uwzględnienie walidacji za pomocą klasy `DataErrorValidationRule`.

Program można uruchomić. Na rysunku 12.4 znajduje się zrzut ekranu po wprowadzeniu błędnych wartości dla obu pól.

#### Rysunek 12.4.

*Wyświetlenie komunikatu o błędach z użyciem klasy `DataErrorValidationRule`.*



Począwszy od wersji platformy .NET 4.5, dostępny jest jeszcze jeden interfejs umożliwiający walidację — `INotifyDataErrorInfo`. Interfejs ten zawiera właściwość logiczną `HasErrors`, metodę `GetErrors` oraz zdarzenie `ErrorsChanged`. Rozwiązanie z użyciem `INotifyDataErrorInfo` w przeciwieństwie do `IDataErrorInfo` może działać asynchronicznie. Ponadto pozwala także na raportowanie wielu błędów dla jednej właściwości<sup>3</sup>.

## 12.2 Definiowanie własnych reguł walidacji

Poprawność danych można sprawdzić także poprzez stworzenie własnych reguł walidacji. Na stronie MSDN<sup>4</sup> znajduje się prosty i czytelny przykład takiej walidacji. Stworzymy własną wersję tego programu, bardziej uniwersalną.

Wykonamy tym razem walidację tylko dla pola z ceną. Styl z szablonem dedykowany dla `TextBox` pozostaje bez zmian. Zmieniamy jedynie mechanizm walidacji, a nie sposób wyświetlania komunikatów o błędach. W kodzie XAML ustaw tytuł, rozmiary okna i język na: `Title="Walidacja danych (wersja 5)" Height="100" Width="570" Language="pl-PL"`. Następnie wpisz kod XAML z poprzedniego zadania (wersja 4), podmieniając jedynie zawartość panelu `WrapPanel` na następującą:

```
<WrapPanel>
    <Label Content="Nazwa:"/>
    <TextBox Text="{Binding Nazwa}"/>
    <Label Content="Cena:"/>
    <TextBox
        <TextBox.Text>
```

<sup>3</sup> Interfejs `INotifyDataErrorInfo` („INotifyDataErrorInfo Interface”): [https://msdn.microsoft.com/en-us/library/system.componentmodel.inotifydataerrorinfo\(v=vs.105\).aspx](https://msdn.microsoft.com/en-us/library/system.componentmodel.inotifydataerrorinfo(v=vs.105).aspx).

<sup>4</sup> Przykład z użyciem własnych reguł walidacji („How to: Implement Binding Validation”): [https://msdn.microsoft.com/en-us/library/ms753962\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms753962(v=vs.110).aspx).

```

        <Binding Path="Cena" UpdateSourceTrigger="PropertyChanged"
                StringFormat="{0:F2}">
            <Binding.ValidationRules>
                <local:ValidatorLiczba Min="0.1" Max="1000"/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
</WrapPanel>

```

W prezentowanym kodzie XAML zwróćmy uwagę na „podłączenie” obiektu klasy `ValidatorLiczba` (z naszymi regułami walidacji) do pola tekstowego z ceną. Wykorzystana została w tym celu właściwość `ValidationRules` klasy `Binding`.

Klasa `Towar` ma mieć pierwotną postać:

```

class Towar
{
    public string Nazwa { get; set; }
    public double Cena { get; set; }
}

```

Do wykonania została nam już tylko klasa, w której będą zapisane reguły walidacji. Taka klasa musi dziedziczyć po klasie `ValidationRule`. Reguły walidacji implementuje się w nadpisanej metodzie `Validate`. Utwórz nowy plik dla klasy `ValidatorLiczba` i dodaj w nim dyrektywę `using`:

```
using System.Windows.Controls;
```

Następnie wpisz kod klasy `ValidatorLiczba`:

```

class ValidatorLiczba: ValidationRule
{
    public double Min { get; set; }
    public double Max { get; set; }

    public override ValidationResult Validate(object value,
        System.Globalization.CultureInfo cultureInfo)
    {
        double sprawdzanaLiczba = 0;
        try
        {
            if (value.ToString().Length > 0)
                sprawdzanaLiczba = Double.Parse(value.ToString());
        }
        catch (FormatException e)
        {
            return new ValidationResult(false, "Niedozwolone znaki - " +
                e.Message);
        }

        if (sprawdzanaLiczba < Min || sprawdzanaLiczba > Max)
        {

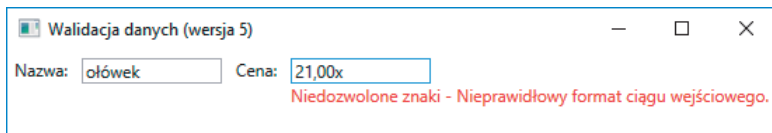
```

```
        return new ValidationResult(false, "Wprowadź liczbę z zakresu: " +  
            Min + " - " + Max);  
    }  
    else  
    {  
        return new ValidationResult(true, null);  
    }  
}
```

W metodzie `Validate` wprowadzana wartość jest w pierwszej kolejności konwertowana do typu `double`. Gdyby to się nie udało, jako rezultat zostanie zwrócony obiekt klasy `ValidationResult` z odpowiednim komunikatem o błędzie. Jeżeli wartość można przekonwertować do liczby `double`, sprawdzany jest jej zakres na podstawie właściwości tej klasy `Min` i `Max`. Przykładowe działanie programu przedstawia rysunek 12.5.

**Rysunek 12.5.**

*Wyświetlenie komunikatu o błędzie z użyciem własnych reguł walidacji*



## 12.3 Wyrażenia regularne

Do sprawdzania poprawności danych możemy wykorzystać wyrażenia regularne — specjalny język opisujący wzorzec występowania dozwolonych znaków w danym łańcuchu tekstowym. W środowisku .NET w przestrzeni nazw `System.Text.RegularExpressions` dostępna jest klasa `Regex`, która umożliwia pracę z wyrażeniami regularnymi<sup>5</sup>. Klasę `Regex` można wykorzystać do operowania na tekstach. My jednak skupimy się na jej walidacyjnym zastosowaniu, poprzez które możemy wzbogacić zaprezentowane w tym rozdziale mechanizmy walidacji dostarczane przez WPF.

Wszystkie znaki używane w wyrażeniach regularnych moglibyśmy podzielić na trzy grupy<sup>6</sup>:

- ◆ Znaki dopasowań — określają dozwolone znaki, np. `[0-9]` oznacza dowolną cyfrę z zakresu od 0 do 9. Identyczne znaczenie ma znak `\d`. Znaki dopasowań zawiera tabela 12.1.
- ◆ Znaki pozycyjne — określają pozycję wzorca w łańcuchu: czy łańcuch musi się zaczynać od wzorca (znak `^`) i czy musi się kończyć wzorcem (znak `$`). Brak tych znaków oznacza, że sprawdzane będzie zawieranie wzorca w tekście, czyli sam tekst będzie mógł zawierać jeszcze inne znaki. Znaki pozycyjne zawiera tabela 12.2.

<sup>5</sup> Klasa `Regex` nie jest klasą WPF, omawiana jest tu ze względu na jej przydatność podczas walidacji danych.

<sup>6</sup> Perry S.C., *Core C# i .NET*, Helion, Gliwice 2006 (str. 242).



- ♦ Znaki powtórzeń i opcjonalne — określają występowanie znaku (lub grupy znaków) przed danym symbolem: czy wystąpi przynajmniej jeden raz, zero razy, wiele razy lub konkretną liczbę razy. Moglibyśmy te znaki porównać (w dużym uroszczeniu) do instrukcji sterujących, w szczególności do pętli. Znaki powtórzeń i opcjonalne zawiera tabela 12.3.

**Tabela 12.1.** Składnia wyrażeń regularnych — znaki dopasowań

Symbol	Opis	Przykłady wzorca
.	Dowolny znak.	<code>.ala</code> — dopasowany może być łańcuch, który zawiera takie słowa jak „sala”, „lala” itp.
[ ]	Dowolny znak spośród wymienionych w nawiasach kwadratowych. Można użyć znaku „-” dla zakresu, np. <code>[A-Z]</code> oznacza duże litery od A do Z.	<code>[A-Zasz]</code> — dopasowany może być łańcuch, który zawiera albo dowolną dużą literę (z zakresu A – Z), albo jedną z trzech małych liter: a, s lub z, np. „Ania”, „sosna”.  <code>k[aio]t</code> — dopasowany może być łańcuch, który zawiera słowo „kat”, „kot” lub „kit”.  <code>Apollo1[0-7]</code> — dopasowany może być łańcuch, który zawiera słowo „Apollo10”, „Apollo11” lub (...) do „Apollo17”.
[^ ]	Dowolny znak spośród niewymienionych w nawiasach kwadratowych, np. <code>[^0-9]</code> oznacza, że znak nie może być cyfrą.	<code>[^A-Z]</code> — dopasowany zostanie łańcuch, który nie zawiera dużych liter.
\d	Dowolna cyfra (odpowiednik <code>[0-9]</code> ).	<code>\dZZ\d</code> — dopasowany zostanie łańcuch, który zawiera słowo złożone z jednej cyfry, liter ZZ oraz jeszcze jednej cyfry, np. „1ZZ5”.
\D	Dowolny znak inny niż cyfra (odpowiednik <code>[^0-9]</code> ).	<code>\D\d</code> — dopasowany zostanie łańcuch, który zawiera słowo złożone z jednego znaku niebędącego cyfrą oraz jednej cyfry, np. „A1”, „%3”, „z8”.
\s	Dowolny znak biały (np. spacja, tabulacja).	<code>\d\s\d</code> — dopasowany zostanie łańcuch, który zawiera ciąg znaków złożony z jednej cyfry, jednego znaku białego i jeszcze jednej cyfry, np. „3 6”.
\S	Dowolny znak, który nie jest znakiem białym.	<code>\S[5-9]</code> — dopasowany zostanie łańcuch, który zawiera ciąg znaków złożony z jednego znaku, który nie jest znakiem białym, oraz jednej cyfry z zakresu 5 – 9, np. „X6”, „%8”.
\w	Dowolny znak alfanumeryczny (cyfry i litery) z ewentualnymi znakami podkreślenia (odpowiednik <code>[a-zA-Z0-9_]</code> ).	<code>\w\s\w</code> — dopasowany zostanie łańcuch, który zawiera ciąg znaków złożony z jednego znaku alfanumerycznego, jednego znaku białego oraz jeszcze jednego znaku alfanumerycznego, np. „A 7”, „3 2”.
\W	Dowolny znak, który nie jest znakiem alfanumerycznym ani znakiem podkreślenia (odpowiednik <code>[^a-zA-Z0-9_]</code> ).	<code>\W\W</code> — dopasowany zostanie łańcuch, który zawiera dwa znaki niealfanumeryczne, np. „##”, „!*”.

Tabela 12.1. Składnia wyrażeń regularnych — znaki dopasowań (ciąg dalszy)

Symbol	Opis	Przykłady wzorca
<code>\p{name}</code>	Dopasowany zostanie znak Unicode zgodny z kategorią określoną przez <code>{name}</code> , np. <code>\p{Lu}</code> oznacza dopasowanie dużej litery. Wykaz kategorii można znaleźć na stronie MSDN <sup>7</sup> .	<code>\p{Ll}</code> — dopasowany zostanie łańcuch zawierający małą literę.
<code>\P{name}</code>	Dopasowany zostanie znak, który nie jest zgodny z kategorią określoną przez <code>{name}</code> .	<code>\P{Lu}</code> — dopasowany zostanie łańcuch, który nie zawiera dużej litery.

Zwróćmy uwagę na to, że w objaśnieniach do przedstawionych przykładów pojawia się słowo „zawiera”. Dla przykładowych wzorców będzie sprawdzane zawieranie. Na przykład wzorec `.ala` zostanie dopasowany do słowa „sala”, ale także do frazy „wolna sala” (ponieważ dowolny znak i „ala” zawierają się w tej frazie) i do łańcucha „balanga” (bo tu też się zawiera podany wzorec). Podczas walidacji mamy również do czynienia ze sprawdzaniem zawierania, ale w typowych polach (jak np. kod pocztowy) zazwyczaj potrzebujemy wzorca, który jest zgodny z analizowanym łańcuchem. Wówczas dany wzorec zarówno zaczyna się, jak i kończy w danym tekście. Takie zdefiniowanie wzorca jest możliwe dzięki użyciu znaków pozycjonowania, które przedstawia tabela 12.2.

Tabela 12.2. Składnia wyrażeń regularnych — znaki pozycjonowania

Symbol	Opis	Przykłady wzorca
<code>^</code>	Wzorec musi się znajdować na początku łańcucha tekstowego.	<code>^d</code> — dopasowany zostanie łańcuch, który zaczyna się od cyfry, np. „4you”, „153xcw”.
<code>\$</code>	Wzorec musi się znajdować na końcu łańcucha tekstowego.	<code>ski\$</code> – dopasowany zostanie łańcuch, który kończy się na „ski”, np. „Kowalski”, „miski”. <code>^być\$</code> — dopasowany zostanie tylko łańcuch „być”. Łańcuch „być albo nie być” nie będzie dopasowany (ponieważ są inne znaki między jednym a drugim wystąpieniem słowa „być”). <code>^[0-9][0-9]\$</code> — dopasowany zostanie łańcuch mający dwie cyfry i nic więcej, np. „78”.
<code>\b</code>	Dopasowanie musi się znajdować na początku lub końcu słowa.	<code>\bd\w+</code> — wzorec pozwala sprawdzić, czy w tekście jest słowo zaczynające się na literę „d”, np. „miły dom”. <code>\w+y\b</code> — wzorec pozwala sprawdzić, czy w tekście jest słowo kończące się na literę „y”, np. „miły dom”.
<code>\B</code>	Dopasowanie nie może się znajdować na początku lub końcu słowa.	<code>\Bgra\B</code> — wzorec pozwala sprawdzić, czy w środku słowa jest łańcuch „gra”, np. „programowanie”. <code>\Bnie</code> — dopasowane zostaną teksty, w których jest słowo zawierające łańcuch „nie”, ale nie na początku, np. „programowanie” lub „mniej” (ale nie będzie pasować „niebo”).

<sup>7</sup> Wykaz kategorii znaków Unicode używanych przez symbole `\p{name}` i `\P{name}` („Supported Unicode General Categories”): [https://msdn.microsoft.com/en-us/library/20bw873z\(v=vs.110\).aspx#SupportedUnicodeGeneralCategories](https://msdn.microsoft.com/en-us/library/20bw873z(v=vs.110).aspx#SupportedUnicodeGeneralCategories).

Dotychczasowe przykłady nie zawierały zbyt wielu powtórzeń. Wzorec dla dwóch cyfr zapisany jako `[0-9][0-9]` lub `\d\d` nie jest jeszcze kłopotliwy do napisania. Ale już wzorec dla numeru PESEL, który ma 11 cyfr, byłby niewygodny do zakodowania bez symboli sterujących powtórzeniami. Symbole z tej grupy zawiera tabela 12.3.

**Tabela 12.3.** Składnia wyrażeń regularnych — znaki powtórzeń i opcjonalne

Symbol	Opis	Przykłady wzorca
<code>?</code>	Zero lub jedno wystąpienie znaku znajdującego się przed tym symbolem.	<code>Mo?c</code> — dopasowany zostanie łańcuch, który zaczyna się od litery „M”, następnie występuje w nim raz lub nie występuje w ogóle litera „o”, a na końcu ma literę „c”, np. „Moc”, „Mc”.
<code>+</code>	Jedno lub więcej wystąpień znaku znajdującego się przed tym symbolem.	<code>[a-z]+</code> — dopasowany zostanie łańcuch, który składa się przynajmniej z jednej małej litery.
<code>*</code>	Zero lub więcej wystąpień znaku znajdującego się przed tym symbolem.	<code>[A-Z][1-5]*</code> — dopasowany zostanie łańcuch, który zawiera jedną dużą literę oraz zero lub więcej cyfr z zakresu 1 – 5, np. „A”, „B121”.  <code>A.*</code> — dopasowany zostanie łańcuch, który zawiera jedną literę „A” oraz opcjonalnie dowolny ciąg znaków.
<code>{n}</code>	n wystąpień znaku znajdującego się przed tym symbolem.	<code>\d{3}</code> — dopasowany zostanie łańcuch, który zawiera trzy dowolne cyfry.
<code>{n,}</code>	Przynajmniej n wystąpień znaku znajdującego się przed tym symbolem.	<code>ab{2,}a</code> — dopasowany zostanie łańcuch, który zawiera „abba” lub „abbba” i podobne teksty z powieloną literą „b” (litera „b” musi wystąpić przynajmniej dwa razy).
<code>{n,m}</code>	Przynajmniej n wystąpień znaku znajdującego się przed tym symbolem, ale nie więcej niż m.	<code>ab{2,3}a</code> — dopasowany zostanie łańcuch, który zawiera „abba” lub „abbba” (litera „b” musi wystąpić przynajmniej dwa razy, ale maksymalnie trzy razy).

Jeśli chodzi o znaki powtórzeń i opcjonalne, należy zwrócić uwagę na to, że wszystkie te symbole sterują tym znakiem (lub grupą znaków), który jest umieszczony PRZED nim. Grupy znaków ujmuje się w nawiasy okrągłe. Przykładowo wzorec `^(http://)?www.*$` oznacza, że grupa znaków „http://” jest opcjonalna (może wystąpić 1 raz lub 0 razy), natomiast musi się znaleźć sekwencja „www”, za którą mogą być (ale nie muszą) dowolne znaki.

Prezentowane tabele nie zawierają wszystkich symboli wyrażeń regularnych, pozostałe można znaleźć na stronie MSDN<sup>8</sup>. Wśród niewymienionych w tabelach symboli, o których warto wspomnieć, jest znak pionowej kreski (`|`), oznaczający alternatywę, oraz lewy ukośnik (`\`), nadający znakom dosłowne znaczenie. Dla przykładowego wzorca `ski|ska` zostanie dopasowany łańcuch, który zawiera znaki „ski” lub znaki „ska”, np. „Kowalski”, „skała”. Natomiast lewy ukośnik poprzedza znak, który ma być traktowany jako znak dosłowny. Stosujemy ten symbol, gdy chcemy we wzorcu uzyskać dosłowną kropkę (`\.`), plus (`\+`) czy inny znak pełniący w wyrażeniach regularnych

<sup>8</sup> Symbole wyrażeń regularnych („Regular Expression Language”): [https://msdn.microsoft.com/en-us/library/az24scfc\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/az24scfc(v=vs.110).aspx).

jakaś funkcję. Dla wzorca `\w\.pl` dopasowany zostanie łańcuch, który zawiera ciąg znaków złożony z jednego znaku alfanumerycznego, jednej kropki oraz liter „pl”, np. „a.pl”, „l.pl”.

W celu sprawdzenia poprawności danych możemy wykorzystać metodę `IsMatch` klasy `Regex`, która sprawdza dopasowanie wzorca do łańcucha. Przykładowy warunek z użyciem metody `IsMatch` może mieć następującą postać:

```
Regex wzorzec = new Regex("[0-9]{11}$");  
if(!wzorzec.IsMatch(Pesel))  
    komunikat = "Numer PESEL musi mieć 11 znaków";
```

gdzie `Pesel` to sprawdzany łańcuch znakowy.

Klasa `Regex` dostarcza także metodę `IsMatch` w wersji statycznej:

```
if (!Regex.IsMatch(Pesel, "[0-9]{11}$"))  
    komunikat = "Numer PESEL musi mieć 11 znaków";
```

Ponieważ niektóre symbole wyrażeń regularnych zawierają lewy ukośnik (np. `\d`), ten sam, który jest wykorzystywany w znakach specjalnych języka C# (takich jak znak nowej linii `\n`), nie możemy podać wzorca w taki sposób:

```
Regex wzorzec = new Regex("^\\d{11}$"); // Tu będzie błąd!
```

Musimy dać dwa ukośniki (co oznacza dosłowny ukośnik), na przykład:

```
Regex wzorzec = new Regex(@"^\\d{11}$");
```

Drugim rozwiązaniem, wygodniejszym, gdy we wzorcu znajduje się więcej ukośników, jest umieszczenie przed pierwszym znakiem cudzysłowu znaku `@`, na przykład:

```
Regex wzorzec = new Regex(@"^\\d{11}$");
```

Musimy pamiętać, że w przypadku wyrażeń regularnych mamy do czynienia z „językiem w języku”. Wpisanie znaku `@` lub podwójnego ukośnika ma zapewnić nam dosłowne traktowanie lewego ukośnika w tym „zewnętrznym języku” i pozostawienie jego interpretacji w gestii „wewnętrznego języka” (czyli wyrażeń regularnych).

## Przykład walidacji danych z użyciem wyrażeń regularnych

Możemy przystąpić do napisania programu z użyciem klasy `Regex`. Wykorzystamy rozwiązanie omawiane w podrozdziale 12.1 z implementacją interfejsu `IDataErrorInfo`.

W nowym projekcie stwórz plik dla definicji klasy `Osoba.cs`. Dodaj tam dyrektywę `using`:

```
using System.ComponentModel;  
using System.Text.RegularExpressions;
```

Następnie zdefiniuj klasę `Osoba`:

```

class Osoba: IDataErrorInfo
{
    public string Nazwisko { get; set; }
    public string Imiona { get; set; }
    public string Pesel { get; set; }
    public string KodPocztowy { get; set; }

    public string Error
    {
        get
        {
            throw new NotImplementedException();
        }
    }

    public string this[string nazwaWlasosciOsoby]
    {
        get
        {
            string komunikat = String.Empty;
            switch (nazwaWlasosciOsoby)
            {
                case "Nazwisko":
                    if (string.IsNullOrEmpty(Nazwisko))
                        komunikat = "Nazwisko musi być wpisane!";
                    else if (!Regex.IsMatch(Nazwisko, @"^[A-Z][a-z]+$"))
                        komunikat = "Nazwisko z dużej litery i minimum 2 znaki!";
                    break;
                case "Imiona":
                    if (string.IsNullOrEmpty(Imiona))
                        komunikat = "Należy wpisać imię lub imiona!";
                    else if (!Regex.IsMatch(Imiona, @"^[A-Z][a-z]
↪+([s[A-Z][a-z]+)*$"))
                        komunikat = "Imiona z dużej litery i minimum 2 znaki";
                    break;
                case "Pesel":
                    if (string.IsNullOrEmpty(Pesel))
                        komunikat = "Pesel musi być wpisany!";
                    else if (!Regex.IsMatch(Pesel, @"^\d{11}$"))
                        komunikat = "Numer PESEL musi mieć 11 znaków";
                    break;
                case "KodPocztowy":
                    if (string.IsNullOrEmpty(KodPocztowy))
                        komunikat = "Kod pocztowy musi być wpisany!";
                    else if (!Regex.IsMatch(KodPocztowy, @"^\d{2}-\d{3}$"))
                        komunikat = "Kod pocztowy ma mieć format 99-999";
                    break;
            };
            return komunikat;
        }
    }
}

```

Klasa ma cztery właściwości: `Nazwisko`, `Imiona`, `Pesel` oraz `KodPocztowy`. Dla właściwości `Nazwisko` został przyjęty wzorzec `^[A-Z][a-z]+$`, który oznacza, że nazwisko musi się zaczynać od dużej litery, a następnie powinno mieć przynajmniej jedną małą literę. Taki wzorzec przyjmie tylko jednoczłonowe nazwisko. Dla właściwości `Imiona` zdefiniowano bardziej złożony wzorzec: `@^[A-Z][a-z]+(\s[A-Z][a-z]+)*$`, który przyjmie dowolną liczbę imion oddzielonych spacją, przy czym każde z nich musi się składać z jednej dużej litery i przynajmniej jednej małej litery. Zakładamy, że musi być przynajmniej jedno imię (bez znaków białych), pozostałe imiona, poprzedzone spacją, to znaczy grupa symboli w nawiasach `(\s[A-Z][a-z]+)`, są opcjonalne, ponieważ znajduje się za tą grupą symboli gwiazdka, oznaczająca 0 lub więcej razy. Dla numeru PESEL przewidziano prosty wzorzec, sprawdzający, czy numer ma 11 cyfr: `@^\d{11}$`. Natomiast kod pocztowy będzie weryfikowany za pomocą wzorca: `@^\d{2}-\d{3}$`, który sprawdza, czy kod pocztowy składa się z dwóch cyfr, łącznika i trzech cyfr.

W kodzie XAML ustaw tytuł, rozmiary okna i język na: `Title="Walidacja danych (Regex)" Height="180" Width="450"`. Następnie wpisz kod:

```
<Window.DataContext>
    <local:Osoba Nazwisko="Kowalski" Imiona ="Jan Andrzej"
        Pesel="01234567890" KodPocztowy="40-000"/>
</Window.DataContext>
<Window.Resources>
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="Margin" Value="5"/>
        <Setter Property="Width" Value="120"/>
        <Setter Property="Validation.ErrorTemplate">
            <Setter.Value>
                <ControlTemplate>
                    <WrapPanel>
                        <AdornedElementPlaceholder/>
                        <TextBlock Text="{Binding CurrentItem.ErrorContent}"
                            Foreground="Red" Margin="5,0"/>
                    </WrapPanel>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Label Content="Nazwisko:" Grid.Row="0" Grid.Column="0"/>
    <TextBox Text="{Binding Nazwisko, UpdateSourceTrigger=PropertyChanged,
        ValidatesOnDataErrors=True}" Grid.Row="0" Grid.Column="1"/>
    <Label Content="Imiona:" Grid.Row="1" Grid.Column="0"/>
```

```

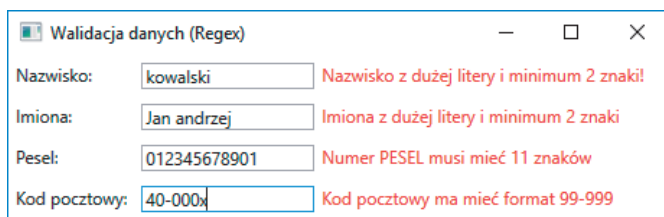
<TextBox Text="{Binding Imiona, UpdateSourceTrigger=PropertyChanged,
    ValidatesOnDataErrors=True}" Grid.Row="1" Grid.Column="1"/>
<Label Content="Pesel:" Grid.Row="2" Grid.Column="0"/>
<TextBox Text="{Binding Pesel, UpdateSourceTrigger=PropertyChanged,
    ValidatesOnDataErrors=True}" Grid.Row="2" Grid.Column="1"/>
<Label Content="Kod pocztowy:" Grid.Row="3" Grid.Column="0"/>
<TextBox Text="{Binding KodPocztowy, UpdateSourceTrigger=PropertyChanged,
    ValidatesOnDataErrors=True}" Grid.Row="3" Grid.Column="1"/>
</Grid>

```

W szablonie dla właściwości `Validation.ErrorTemplate` nieznacznie zmieniony został sposób wyświetlania komunikatów o błędach względem poprzednich programów — tym razem komunikaty będą wyświetlane obok pola z błędną wartością, a nie poniżej (panel `StackPanel` został zmieniony na `WrapPanel`). W panelu `Grid` umieszczone są pola tekstowe dla wszystkich czterech właściwości klasy `Osoba`.

Program można uruchomić. Rysunek 12.6 przedstawia przykładowe działanie programu po wpisaniu błędnych wartości.

**Rysunek 12.6.**  
Walidacja danych  
z użyciem klasy  
`Regex`



W programie dla uproszczenia pominięta została kwesta polskich znaków diakrytycznych (takich jak ą, ę, ć, ś itd.). Oczywiście nie ma przeszkód, aby uwzględnić te znaki. Można dodać polskie znaki do wykazu znaków w nawiasach kwadratowych — przykładowo wzorec dla jednocłonowego nazwiska zaczynającego się od dużej litery i składającego się z co najmniej dwóch znaków mógłby wyglądać tak<sup>9</sup>:

```

if (!Regex.IsMatch(Nazwisko, @"^[A-ZĄĆĘŁŃŚŻŹ][a-ząćęłńśżź]+$")) // Komunikat
                                                                    // o błędzie

```

Można także w tym celu użyć symbolu `\p{name}` opisanego w tabeli 12.1:

```

if (!Regex.IsMatch(Nazwisko, @"^\p{Lu}\p{Ll}+$")) // Komunikat o błędzie

```

## 12.4 Zadania

Do wykonania mamy trzy zadania. W pierwszym należy zmienić wzorec wyrażenia regularnego. Zadanie drugie będzie polegało na „odświeżeniu” jednego z poprzednich programów i dodaniu walidacji danych. W ostatnim zadaniu należy wykonać walidację danych dla całego wiersza w kontrolce `DataGrid`.

<sup>9</sup> W części wzorca dla dużej pierwszej litery nazwiska można rozważyć usunięcie liter „A” i „E”, chyba że planujemy w programie wpisywać fikcyjne nazwiska (np. Edward ącki).

## Zadanie 12.1

W programie z podrozdziału 12.3 zmień wzorec wyrażenia regularnego dla nazwiska. Przyjmij możliwość wpisania nazwisk maksymalnie trójczłonowych, z których każdy człon składa się z pierwszej dużej litery i minimum jednej małej litery. Człony mogą być oddzielone łącznikiem lub spacją. Językoznawcy odróżniają termin „myślnik” od terminu „łącznik”: łącznik jest to krótka kreska, która łączy wyrazy bez spacji, np. Kowalska-Nowak. Dla nas będzie to oczywiście „minus” bez okolicznych spacji.

## Zadanie 12.2

Napisz program na podstawie programu z wykazem produktów stworzonego w podrozdziałach 7.1 – 7.3 z użyciem `DataGrid`. W pierwszej kolejności przenieś definicję wiązania kolekcji z *code-behind* do XAML. Następnie wykonaj walidację danych w `DataGrid` przy użyciu implementacji interfejsu `IDataErrorInfo`. Wykonaj sprawdzanie dla dwóch właściwości klasy `Produkt`. `Symbol` ma składać się z dwóch członów oddzielonych łącznikiem (minusem): pierwszy człon może się składać albo z dwóch dużych liter, albo z dużej litery i cyfry, natomiast drugi człon ma się składać z dwóch cyfr. Drugą sprawdzaną właściwością ma być `LiczbaSztuk`, dla której należy przyjąć zakres wartości `<0,10000>`.

## Zadanie 12.3

Do programu tworzonego w poprzednim zadaniu dodaj walidację całego wiersza. Poprawność danych sprawdza się nie tylko poprzez weryfikację każdego pola z osobna, ale także badanie spójności danych i ich wzajemnych zależności. Przykładowo w programie, w którym wpisuje się dwie daty: zatrudnienia i zwolnienia, można dokonać weryfikacji dla upewnienia się, czy data zatrudnienia jest wcześniejsza od daty zwolnienia. W naszym programie nie mamy tak wyraźnego powodu do walidacji całego obiektu (produktu), ale możemy sobie przyjąć założenie, które tego wymaga. Założmy, że produkty, których symbol zaczyna się od litery „A”, muszą mieć stan magazynowy minimum 10. Cały wiersz, w którym nie jest spełniony ten warunek, ma być wyróżniony poprzez narysowanie czerwonej ramki. W programie wykorzystaj implementację właściwości `Error`.

# 12.5 Wskazówki do zadań

Wskazówki, podobnie jak w innych rozdziałach, zawierają szczegółowe wyjaśnienia i fragmenty rozwiązań.

## Wskazówki do zadania 12.1

Spójrzmy najpierw na wzorec dla właściwości `Imiona`:

```
if (!Regex.IsMatch(Imiona, @"^[A-Z][a-z]+(\s[A-Z][a-z]+)*$")) // Komunikat...
```



Wzorzec dla nazwiska ma się różnić w dwóch aspektach: po pierwsze zamiast nieograniczonej liczby członów mają być maksymalnie trzy, a po drugie mamy uwzględnić dwa separatory, mianowicie spację i łącznik (minus), czyli dwa człony nazwiska oddzielone mogą być albo jedną spacją, albo jednym łącznikiem. W wersji bez polskich znaków mogłoby to wyglądać tak:

```
if (!Regex.IsMatch(Nazwisko, @"^[A-Z][a-z]+((\s|-)[A-Z][a-z]+){0,2}$"))
    // Komunikat...
```

Natomiast w wersji uwzględniającej polskie znaki wzorzec może mieć następującą postać:

```
if (!Regex.IsMatch(Nazwisko, @"^\p{Lu}\p{Ll}+((\s|-)\p{Lu}\p{Ll}+){0,2}$"))
    // Komunikat...
```

W obu wersjach wzorca pierwszy człon jest obowiązkowy (i nie ma w nim żadnych separatorów), natomiast dwa kolejne człony wraz z poprzedzającym separatorem są opcjonalne, to znaczy może nie być żadnego z nich, może być jeden albo mogą być dwa.

## Wskazówki do zadania 12.2

Zacznijmy od pierwszego etapu, mianowicie przeniesienia definicji wiązania z kolekcją do kodu XAML. Można uprościć definicję samej klasy **Produkt**:

```
class Produkt : IDataErrorInfo
{
    public string Symbol { get; set; }
    public string Nazwa { get; set; }
    public int LiczbaSztuk { get; set; }
    public string Magazyn { get; set; }
}
```

Stwórz kolekcję dla magazynów:

```
class Magazyny : ObservableCollection<string>
// Dodaj using System.Collections.ObjectModel;
{
    public Magazyny()
    {
        Add("Katowice 1");
        Add("Katowice 2");
        Add("Gliwice 1");
    }
}
```

I dla produktów:

```
class KolekcjaProduktow : ObservableCollection<Produkt>
// Dodaj using System.Collections.ObjectModel;
{
    public KolekcjaProduktow()
    {
        Add(new Produkt
        {
            Symbol = "01-11",
            Nazwa = "ołówki",
        })
    }
}
```

```

        LiczbaSztuk = 8,
        Magazyn = "Katowice 1"
    });
    Add(new Produkt
    {
        Symbol = "PW-20",
        Nazwa = "pióro wieczne",
        LiczbaSztuk = 75,
        Magazyn = "Katowice 2"
    });
    Add(new Produkt
    {
        Symbol = "DZ-10",
        Nazwa = "długopis żelowy",
        LiczbaSztuk = 1121,
        Magazyn = "Katowice 1"
    });
    Add(new Produkt
    {
        Symbol = "DZ-12",
        Nazwa = "długopis kulkowy",
        LiczbaSztuk = 280,
        Magazyn = "Katowice 2"
    });
    }
}

```

*Code-behind* z definicją klasy `MainWindow` ma mieć taką postać, jaka jest automatycznie definiowana dla nowego projektu WPF, to znaczy:

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}

```

W kodzie XAML dodaj zasób z utworzonymi kolekcjami, na przykład:

```

<Window.Resources>
    <local:KolekcjaProduktow x:Key="wykazProduktow"/>
    <local:Magazyny x:Key="listaMagazynow"/>
</Window.Resources>

```

Kod dla `DataGrid` należy nieznacznie zmienić względem programu z podrozdziału 7.3, tak aby po zmianach miał postać:

```

<DataGrid AutoGenerateColumns="False" AlternatingRowBackground="LightBlue"
    GridLinesVisibility="None"
    ItemsSource="{Binding Source={StaticResource wykazProduktow}}">
    <DataGrid.Columns>
        <DataGridTextColumn Header="Symbol" Binding="{Binding Symbol}"/>
        <DataGridTextColumn Header="Nazwa" Binding="{Binding Nazwa}"/>
    </DataGrid.Columns>
</DataGrid>

```

```

<DataGridTextBoxColumn Header="Liczba sztuk" Binding="{Binding LiczbaSztuk}">
  <DataGridTextBoxColumn.ElementStyle>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="HorizontalAlignment" Value="Right" />
    </Style>
  </DataGridTextBoxColumn.ElementStyle>
</DataGridTextBoxColumn>
<DataGridComboBoxColumn x:Name="nazwaMagazynu" ItemsSource="{Binding
  Source={StaticResource listaMagazynow}}" Header="Magazyn"
  SelectedItemBinding="{Binding Magazyn}" />
</DataGrid.Columns>
</DataGrid>

```

Zmiany są w dwóch miejscach. Pierwsza zmiana polega na tym, że w definicji `DataGrid` jest przypisane wiązanie z kolekcją produktów dla właściwości `ItemsSource`. Druga zmiana zaś dotyczy definicji kolumny dla nazwy magazynu, gdzie znajduje się obecnie definicja wiązania z kolekcją magazynów.

Po dokonanych zmianach program można uruchomić — powinien działać tak samo jak w wersji z rozdziału 7. Prezentowane zmiany nie były konieczne dla walidacji danych, były jedynie okazją do przećwiczenia innego sposobu kodowania, z przewagą po stronie kodu XAML.

Teraz można przystąpić do drugiego etapu i wykonania walidacji. Proponuję zacząć od implementacji interfejsu `IDataErrorInfo` w klasie `Produkt`.

Najważniejszą część tej implementacji powinna mieć postać:

```

public string this[string nazwaWlasciwosciProduktu]
{
    get
    {
        string komunikat = String.Empty;
        switch (nazwaWlasciwosciProduktu)
        {
            case "Symbol":
                if (string.IsNullOrEmpty(Symbol))
                    komunikat = "Symbol musi być wpisany!";
                else if (!Regex.IsMatch(Symbol, @"^[A-Z][A-Z0-9]-[0-9]{2}$"))
                    komunikat = "Symbol ma mieć format XX-99 lub X9-99
                    ↳ (X-litera, 9-cyfra)";
                break;
            case "LiczbaSztuk":
                if (LiczbaSztuk < 0 || LiczbaSztuk > 10000)
                    komunikat = "Liczba sztuk ma być z zakresu <0,10000>";
                break;
        };
        return komunikat;
    }
}

```

W kodzie XAML dodaj do zasobów styl pokazujący „dymek” z opisem błędu:

```
<Style x:Key="stylTextBlock" TargetType="{x:Type TextBlock}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource={x:Static
          RelativeSource.Self},
          Path=(Validation.Errors)[0].ErrorContent}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Użycie `ToolTip` do wyświetlenia komunikatów o błędach jest dość wygodne w przypadku `DataGrid` ze względu na sąsiadujące dane w tabeli.

Zmienić należy jeszcze definicję obu kolumn. W przypadku kolumny dla symbolu produktu może mieć ona postać:

```
<DataGridTextColumn Header="Symbol" ElementStyle="{StaticResource
  stylTextBlock}" Binding="{Binding Symbol, ValidatesOnDataErrors=True}" />
```

Dodano tu właściwość `ElementStyle`, dla której przypisany został utworzony styl.

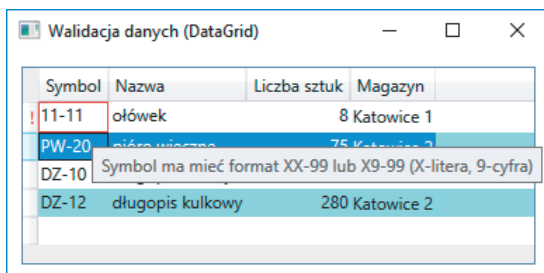
W przypadku zmiany definicji dla liczby sztuk musimy uwzględnić fakt, że dla tej kolumny już jest definiowany styl dla elementu `TextBlock` (z wyrównaniem do prawej). W tym celu dokonamy **dziedziczenia stylu**, na co pozwala właściwość `BasedOn`:

```
<DataGridTextColumn Header="Liczba sztuk" Binding="{Binding LiczbaSztuk,
  ValidatesOnDataErrors=True}">
  <DataGridTextColumn.ElementStyle>
    <Style BasedOn="{StaticResource stylTextBlock}"
      TargetType="{x:Type TextBlock}">
      <Setter Property="HorizontalAlignment" Value="Right" />
    </Style>
  </DataGridTextColumn.ElementStyle>
</DataGridTextColumn>
```

Program można uruchomić. Przykładowy zrzut ekranu z przeprowadzonej walidacji przedstawia rysunek 12.7.

**Rysunek 12.7.**

*Walidacja  
w DataGrid*



`DataGrid` prezentuje dane w trybie edycji (dla kolumn tekstowych `TextBox`) i przeglądu (`TextBlock`). W obecnej wersji programu komunikaty o błędach zostały przypisane tylko dla `TextBlock`. Możemy je zobaczyć dopiero po wyjściu z trybu edycji i podświetleniu błędnego pola. Możesz zdefiniować podobny styl także dla kontrolki `TextBox` — wówczas będzie można zobaczyć automatycznie generowany komunikat w kolumnie *Liczba sztuk*, gdy zostanie wpisana wartość, której nie da się konwertować do typu `int` (przy tym błędnie dana komórka pozostaje w trybie edycji). Nie ma także przeszkód, aby dla kontrolki `DataGrid` zdefiniować swoje własne reguły walidacji w klasie dziedziczącej po `ValidationRule`<sup>10</sup>. Można też zmienić sposób oznaczania błędnej komórki. Domyślny szablon rysuje czerwoną ramkę wokół komórki i umieszcza czerwony wykrzyknik w nagłówku wiersza.

## Wskazówki do zadania 12.3

W klasie `Produkt` zamień „pustą” implementację właściwości `Error` na następujący kod:

```
public string Error
{
    get
    {
        string komunikat = String.Empty;
        if (Symbol.Substring(0,1)=="A" && LiczbaSztuk < 10)
        {
            komunikat = "Wymagana liczba produktów o symbolu A to min.10";
        }
        return komunikat;
    }
}
```

Należy zdefiniować styl dla elementu `DataGridRow`, na przykład:

```
<Style TargetType="{x:Type DataGridRow}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
      <Setter Property="BorderThickness" Value="1"/>
      <Setter Property="BorderBrush" Value="Red"/>
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource={x:Static RelativeSource.Self},
          Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

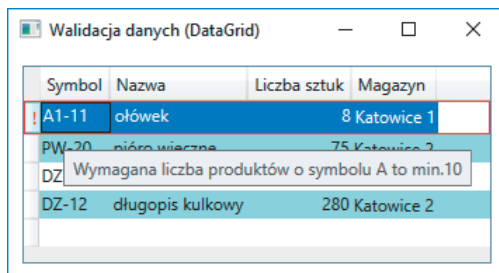
Pozostało już tylko włączyć walidację dla wiersza. Na początku definicji `DataGrid` wpisz:

```
<DataGrid.RowValidationRules>
  <DataErrorValidationRule/>
</DataGrid.RowValidationRules>
```

<sup>10</sup> Przykład użycia własnych reguł walidacji w `DataGrid` („Implement Validation with the DataGrid Control”): [https://msdn.microsoft.com/en-us/library/ee622975\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee622975(v=vs.110).aspx).

Program po uruchomieniu wyświetla komunikat dla wiersza, w którym wpisujemy dane niezgodne z przyjętym założeniem, co pokazuje rysunek 12.8.

**Rysunek 12.8.**  
*Walidacja całego  
wiersza w DataGrid*



## Rozdział 13.

# Wprowadzenie do wzorca projektowego MVVM

Wzorce projektowe wykorzystuje się w celu zastosowania uniwersalnych rozwiązań, często pojawiających się w praktyce programowania. Dla programistów WPF polecany jest wzorzec MVVM (ang. *Model-View-ViewModel*, czyli model – widok – model widoku).

Jeżeli celem Czytelnika było wykonanie kilku programów w WPF i na jakiś czas przerwywa on swoją przygodę z tą technologią, to może na razie pominąć lekturę tego rozdziału. Natomiast polecam jego treść wszystkim tym, którzy zamierzają uczyć się dalej i w przyszłości pracować w tym środowisku. Każde ulepszenie, nie tylko w programowaniu, wiąże się zazwyczaj z kosztem odczuwanym najmocniej na samym początku „inwestycji”. Do przykręcenia kilku niedużych śrubek wystarczy zwykły śrubokręt. Jeśli dana osoba skręca co jakiś czas meble, to ma powody, aby pomyśleć o zakupie prostej wkrętarki. Fachowiec może potrzebować lepszego sprzętu. Podobnie jest w programowaniu ze stosowaniem wzorców projektowych wspieranych przez różne frameworki: przynoszą nam zauważalne korzyści wraz ze wzrostem naszej aktywności zawodowej.

Zaczęliśmy programowanie w WPF od podejścia korzystającego z *code-behind*, zwanego **autonomicznym widokiem** (ang. *autonomous view* — AV). Podejście to nie daje dostatecznej elastyczności i utrudnia testowalność. W dalszej części książki odchodziliśmy powoli od tego stylu programowania, a *code-behind* w naszych ostatnich przykładach nie zawierał dodatkowego kodu. To jest dobry moment na wprowadzenie do wzorca MVVM, który pozwala na uporządkowaną separację logiki programu i sposobu wyświetlania danych. Rozwiązanie to ułatwia zarówno testowanie aplikacji, jak i zmianę wyglądu interfejsu. W tym rozdziale omówię podstawowe zasady wzorca MVVM i wykonamy prosty przykład.

## 13.1 Model-View-ViewModel

Nazwa wzorca (ang. *Model-View-ViewModel* — MVVM) zawiera poszczególne warstwy wzorca, ale nie są one wymienione w takiej kolejności, w jakiej ze sobą współpracują. Faktyczna kolejność warstw ma postać:

**View — ViewModel — Model**

Zaprezentuję warstwy wzorca MVVM, począwszy modelu.

### Model

*Model* jest „najniższą” warstwą, najściślej związaną z danymi, umieszczanymi zazwyczaj w bazie danych. Musimy pamiętać, że wzorce projektowe stanowią pewną propozycję zaleceń i nie oznacza to, że w praktyce programistycznej funkcjonuje jednolita szkoła, ściśle trzymająca się tych samych zasad. Jeśli chodzi o klasy modeli, niektórzy zalecają, aby były możliwie proste, nie obciążone żadnymi konkretnymi mechanizmami platformy .NET<sup>1</sup>. Inni natomiast dopuszczają takie implementacje modelu, które mają udogodnienia wykorzystywane podczas wiązania z widokiem<sup>2</sup>. Do tych różnic jeszcze wrócimy w trakcie omawiania przykładu.

### View

*View* (widok) to kod XAML, w którym definiujemy interfejs użytkownika. Zgodnie z założeniami MVVM nie należy dla widoku używać zaplecza w postaci *code-behind*. Przykładowa definicja przycisku:

```
<Button x:Name="btnStart" Content="Start" Click="btnStart_Click"/>
```

wraz z kodem dla zdarzenia `btnStart_Click` w *code-behind* nie jest zgodna z zasadami MVVM. Zamiast zdarzeń używa się poleceń (ang. *commands*) i znanego nam mechanizmu wiązania. Prawidłowa definicja przycisku w MVVM może mieć postać:

```
<Button Content="Start" Command="{Binding Start}"/>
```

Użyte jest tu wiązanie dla polecenia (*Command*). Można by zadać pytanie: gdzie ma być zdefiniowany kod dla tego polecenia, jeśli nie w *code-behind*? Tym miejscem jest warstwa pośrednia *ViewModel*.

<sup>1</sup> Zgodnie z tym zaleceniem warstwa modelu powinna być tworzona w metodologii projektowania domenowego (ang. *domain-driven design*, DDD), zob.: Matulewski J., *MVVM i XAML w Visual Studio 2015*, Helion, Gliwice 2016 (str. 25 – 26), oraz Garofalo R., *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*, Microsoft Corporation 2011 (str. 61 – 90).

<sup>2</sup> Implementacja wzorca MVVM z użyciem biblioteki PRISM („Implementing the MVVM Pattern Using the Prism Library 5.0 for WPF”): [https://msdn.microsoft.com/en-us/library/gg405484\(v=pandp.40\).aspx#sec4](https://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx#sec4).



## ViewModel

*ViewModel* (model widoku) zajmuje się udostępnianiem modelu dla widoku, zatem pełni rolę pośrednika między warstwami *Model* i *View*. Model widoku pozwala zaimplementować funkcjonalności, jakie przy stosowaniu autonomicznego widoku umieszcza się w *code-behind*. Połączenie między *ViewModel* a *View* opiera się na mechanizmie wiązania danych. Mechanizm ten obejmuje także polecenia, co obrazuje prezentowana wyżej definicja przycisku: `Command="{Binding Start}"`.

## 13.2 Budujemy widok dla przykładowej aplikacji

Zacniemy omawiać aplikację od widoku, ponieważ ta warstwa prawie nic nowego nie wnosi względem tego, co już znamy z poprzednich rozdziałów, i dzięki temu kod powinien być zrozumiały. Ponadto, analizując widok, będzie można omówić oczekiwane działanie aplikacji.

Utwórz nowy projekt. W kodzie XAML zmień tytuł i rozmiary okna na przykład na: `Title="Studenci" Height="210" Width="300"`. Następnie wpisz kod:

```
<Window.DataContext>
    <local:StudentViewModel/>
</Window.DataContext>
<StackPanel>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <Label Content="Imię:" Grid.Row="0" Grid.Column="0"/>
        <TextBox Text="{Binding Kursant.Imie}" Grid.Row="0" Grid.Column="1"
            Width = "120" Margin = "5"/>
        <Label Content="Nazwisko:" Grid.Row="1" Grid.Column="0"/>
        <TextBox Text="{Binding Kursant.Nazwisko}" Grid.Row="1" Grid.Column="1"
            Width = "120" Margin = "5"/>
        <Label Content="Rok rozpoczęcia studiów:" Grid.Row="2" Grid.Column="0"/>
        <TextBox Text="{Binding Kursant.RokPrzyjeciaNaStudia,
            UpdateSourceTrigger=PropertyChanged}" Grid.Row="2" Grid.Column="1"
            Width = "120" Margin = "5"/>
        <Label Content="Ile lat studiuje:" Grid.Row="3" Grid.Column="0"/>
        <TextBlock Text="{Binding Kursant.CzasStudiowania, Mode = OneWay}"
            Grid.Row="3" Grid.Column="2" Width = "120" Margin = "5"/>
    </Grid>
</StackPanel>
```

```

</Grid>
<Button Content="Wyczyść dane" Height="20" Width="100"
        VerticalAlignment="Bottom" HorizontalAlignment="Right"
        Margin="20" Command="{Binding Wyczyszc}" />
</StackPanel>

```

W oknie mamy prosty formularz z trzema polami tekstowymi i jednym elementem `TextBlock`. Pola tekstowe mają służyć do wprowadzenia danych o studencie: imienia, nazwiska oraz roku rozpoczęcia studiów. Dla każdego z tych elementów zdefiniowane są wiązania, a w zasobach jest określony kontekst danych — instancja klasy `Student` → `ViewModel`. Właściwość `Kursant.CzasStudiowania` nie będzie zmieniana. Jej wartość jest obliczana na podstawie aktualnego roku i roku przyjęcia na studia.

W definicji przycisku jest wiązanie do polecenia: `Command="{Binding Wyczyszc}"` — i to jest jedyne miejsce, w którym kod XAML różni się od wcześniej omawianych. Jednym z założeń wzorca MVVM jest minimalizacja *code-behind*. Wszystkie komponenty konkretyzujące omawiany widok zostały z nim połączone poprzez mechanizm wiązania danych. Oznacza to, że prezentowany kod XAML nie wymaga żadnych dodatkowych akcji w *code-behind* i pozostaje on w następującej postaci:

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}

```

## 13.3 Implementacja modelu

Jak już wspomniałam, wzorce projektowe stanowią propozycję zaleceń i nie wszyscy muszą stosować się ściśle do tych samych zasad w zakresie określonych szczegółów implementacji. Co się tyczy budowania klas modelu i modelu widoku, istnieją pewne warianty. W podręcznikach szerzej opisujących wzorec MVVM można znaleźć kilka różnych wariantów definicji tych klas<sup>3</sup>. W niniejszym podrozdziale zaprezentuję tylko jeden wariant, w którym zarówno model, jak i model widoku implementuje interfejs dla powiadomień oraz w modelu widoku tworzony jest obiekt modelu<sup>4</sup>.

W tworzonej aplikacji będzie niewiele plików (klas) i dla uproszczenia nie będziemy tworzyć osobnych folderów w projekcie (w praktyce oczywiście stworzymy foldery i odpowiednio porządkujemy pliki projektu).

<sup>3</sup> Matulewski J., *MVVM i XAML w Visual Studio 2015*, Helion, Gliwice 2016 (str. 29 – 50).

<sup>4</sup> Fakt, że istnieją różne warianty, zapewne nie dziwi Czytelnika. Zasięgając w różnych źródłach opinii w celu zbadania, który z nich jest lepszy i bardziej godny zalecania, możemy napotkać niezgodne zdania. Nie będziemy rozstrzygać tych kwestii na stronach niniejszej książki. Zainteresowanych odsyłam do innych podręczników i źródeł internetowych, w tym do filmu *MVVM Best Practices*, Microsoft TechNet Channel 9, <https://channel9.msdn.com/Shows/Visual-Studio-Toolbox/MVVM-Best-Practices>.

Stwórz w projekcie plik `Student.cs` i dodaj przestrzeń nazw:

```
using System.ComponentModel;
```

Następnie wpisz tam kod klasy `Student`:

```
public class Student : INotifyPropertyChanged
{
    private string imie;
    private string nazwisko;
    private int rokPrzyjeciaNaStudia;

    public string Imie
    {
        get
        {
            return imie;
        }
        set
        {
            if (imie != value)
            {
                imie = value;
                OnPropertyChanged("Imie");
            }
        }
    }

    public string Nazwisko
    {
        get
        {
            return nazwisko;
        }
        set
        {
            if (nazwisko != value)
            {
                nazwisko = value;
                OnPropertyChanged("Nazwisko");
            }
        }
    }

    public int RokPrzyjeciaNaStudia
    {
        get
        {
            return rokPrzyjeciaNaStudia;
        }
        set
        {
            if (rokPrzyjeciaNaStudia != value)
            {
```

```

        rokPrzyjeciaNaStudia = value;
        OnPropertyChanged("RokPrzyjeciaNaStudia");
        OnPropertyChanged("CzasStudiowania");
    }
}

public string CzasStudiowania
{
    get
    {
        return (DateTime.Now.Year - RokPrzyjeciaNaStudia).ToString();
    }
}

// Implementacja interfejsu INotifyPropertyChanged
public event PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged(string property)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(property));
    }
}
}

```

Klasa `Student` definiuje cztery właściwości: `Imie`, `Nazwisko`, `RokPrzyjeciaNaStudia` i `CzasStudiowania`. Ponadto klasa ta implementuje interfejs `INotifyPropertyChanged`, który umożliwia przesyłanie powiadomień (ang. *notify*) o zmianach właściwości. Dzięki takim powiadomieniom nie będziemy musieli dokonywać „ręcznego” odświeżania wartości kontrolki po ewentualnych zmianach obiektów, jak to miało miejsce w niektórych innych naszych programach. Interfejs ten wymaga implementacji zdarzenia `PropertyChanged`, z którego korzysta mechanizm wiązania danych. Zgłaszamy to zdarzenie wówczas, gdy zmieniany jest stan jednej z właściwości. Wykorzystamy w tym celu metodę pomocniczą, nazwaną tu `OnPropertyChanged`. Jej argumentem jest nazwa właściwości, której dotyczy zmiana. Metodę tę wywołujemy podczas definicji akcesorów `set` właściwości danej klasy. Przykładowo w definicji akcesora `set` dla imienia wywołano tę metodę z argumentem „Imie”, to znaczy `OnPropertyChanged("Imie")`. W przypadku zmiany roku rozpoczęcia studiów posyłamy dwa powiadomienia, dotyczące zarówno tej właściwości, jak i czasu studiowania (ponieważ czas studiowania obliczany jest na podstawie roku rozpoczęcia studiów):

```

OnPropertyChanged("RokPrzyjeciaNaStudia");
OnPropertyChanged("CzasStudiowania");

```

## 13.4 Implementacja modelu widoku

Utwórz w projekcie nowy plik z klasą `StudentViewModel`. Dodaj dyrektywy `using` dla przestrzeni nazw:

```
using System.ComponentModel;
using System.Windows;
```

Następnie wpisz kod klasy:

```
class StudentViewModel: INotifyPropertyChanged
{
    private Student _kursant;
    public Student Kursant
    {
        get { return _kursant; }
        set
        {
            _kursant = value;
            OnPropertyChanged("Kursant");
        }
    }
    public MyCommand Wyczysc { get; set; }

    public StudentViewModel()
    {
        Kursant = new Student { Imie = "Jan", Nazwisko = "Kowalski",
                                RokPrzyjeciaNaStudia = 2014 };
        Wyczysc = new MyCommand(WyczyscDane);
    }

    private void WyczyscDane()
    {
        if (MessageBox.Show("Czy wyczyścić dane studenta?", "Pytanie",
                            MessageBoxButton.YesNo, MessageBoxImage.Question) ==
            MessageBoxResult.Yes)
        {
            Kursant.Imie = String.Empty;
            Kursant.Nazwisko = String.Empty;
            Kursant.RokPrzyjeciaNaStudia = DateTime.Now.Year;
        }
    }

    // Implementacja interfejsu INotifyPropertyChanged
    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string property)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(property));
        }
    }
}
```

W klasie `StudentViewModel` także implementujemy interfejs `INotifyPropertyChanged`. W klasie tej definiowana jest właściwość `Kursant` typu `Student` oraz właściwość `Wyczysc` typu `MyCommand`. W tym miejscu odniesiemy się na chwilę do fragmentu kodu XAML w widoku, mianowicie do definicji przycisku:

```
<Button Content="Wyczyść dane" Height="20" Width="100"
        VerticalAlignment="Bottom"
        HorizontalAlignment="Right" Margin="20" Command="{Binding Wyczysc}"/>
```

Jak widać, dla właściwości `Command` przypisane jest wiązanie z właściwością modelu widoku `Wyczysc`. Właściwość ta jest deklarowana w omawianej klasie `StudentViewModel`:

```
public MyCommand Wyczysc { get; set; }
```

oraz definiowana w konstruktorze tej klasy:

```
Wyczysc = new MyCommand(WyczyscDane);
```

Musimy napisać kod klasy `MyCommand`. Stwórz osobny plik `MyCommand.cs` i dodaj przestrzeń nazw:

```
using System.Windows.Input;
```

Następnie wpisz kod klasy:

```
public class MyCommand : ICommand
{
    Action _execute;

    public MyCommand(Action executeMethod)
    {
        _execute = executeMethod;
    }
    public void Execute(object parameter)
    {
        if (_execute != null)
        {
            _execute();
        }
    }
    public bool CanExecute(object parameter)
    {
        return true;
    }
    public event EventHandler CanExecuteChanged;
}
```

Obsługa poleceń w WPF wymaga implementacji interfejsu `ICommand`. Interfejs ten zawiera deklarację dwóch metod `Execute` i `CanExecute` oraz zdarzenie `CanExecuteChanged`. Metoda `Execute` umożliwia wykonanie polecenia. W naszej uproszczonej implementacji interfejsu `ICommand` metoda ta wywoła akcję (`Action`) będącą argumentem konstruktora tej klasy, czyli metodę delegata lub wyrażenia lambda<sup>5</sup>. Metoda `CanExecute` pozwala

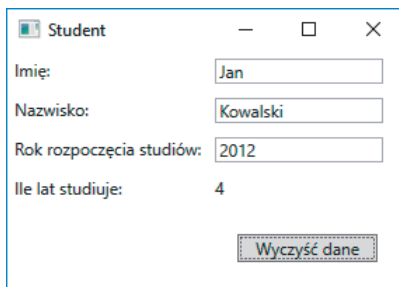
<sup>5</sup> Delegat `Action` („Action Delegate”): [https://msdn.microsoft.com/en-us/library/system.action\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.action(v=vs.110).aspx).

sprawdzić, czy polecenie jest możliwe. W tworzonym programie zakładamy, że zawsze jest możliwe, co stanowi uproszczenie zazwyczaj stosowanych implementacji interfejsu `ICommand`. Zadaniem zdarzenia `CanExecuteChanged` jest powiadamianie o zmianie możliwości wykonania polecenia. Tu nie będziemy korzystać z tych powiadomień, bardziej uniwersalną implementację `ICommand` zaprezentuję w kolejnym podrozdziale.

Program można już uruchomić. Rysunek 13.1 przedstawia okno programu.

#### Rysunek 13.1.

*Okno programu  
Student*



Po wciśnięciu przycisku *Wyczyść dane* pojawi się pytanie *Czy wyczyścić dane studenta?* i jeśli użytkownik odpowie *Tak*, to dane o nazwisku i imieniu zostaną wyczyszczone, a właściwość `RokPrzyjeciaNaStudia` przyjmie wartość na podstawie aktualnego roku.

Uruchomiony program wymaga pewnych dodatkowych uwag, które wraz ze wskazówkami dotyczącymi dalszej nauki wzorca projektowego MVVM omówię w kolejnym podrozdziale.

## 13.5 Przed dalszą nauką MVVM

Nietrudno zauważyć, że kod prezentowanego programu jest wyraźnie dłuższy niż kod niejednego programu tworzonego w tym podręczniku bez wzorca MVVM. A w dodatku sugerowałam, że jest to wersja „light”, to znaczy uproszczona względem faktycznie stosowanych. Typowy problem autora podręcznika polega na tym, że życiowy przykład najlepiej pokazujący przydatność jakiegoś złożonego usprawnienia jest zbyt skomplikowany, aby nadawał się do przystępnego opisanie. Z konieczności zaczyna się zatem od przykładów trywialnych, które pozwalają przedstawić ideę rozwiązania, ale za to mogą rodzić w Czytelniku odczucie przerostu formy nad treścią.

Kod faktycznie stosowanych rozwiązań jest nie tylko dłuższy, ale w oczach początkujących programistów także mało „przyjazny”. Wynika to z dążenia do maksymalnego uogólnienia poszczególnych implementacji. Przykładowo, jeśli klasę `MyCommand` chcielibyśmy wykorzystać dla innych poleceń, takich, dla których sprawdzenie możliwości wykonania polecenia ma znaczenie, wówczas musielibyśmy definicję tej klasy rozszerzyć. Kod, który ma być uniwersalny i zostać użyty w różnych wariantach poleceń, z konieczności musi być wyposażony w odpowiednie konstrukcje, umożliwiające „elastyczne” działanie, na przykład typy generyczne czy typy delegatów. Przydatności takiego rozwiązania raczej nie docenimy w tak prostym przykładzie jak omawiany w tym rozdziale.

Wdrażając do swoich programów różne uniwersalne rozwiązania, możemy wykorzystać proponowane w podręcznikach i internetowych poradnikach „gotowce”. Przykładowo uogólnioną klasę dla poleceń przedstawia w swoim podręczniku na temat wzorca MVVM Jacek Matulewski<sup>6</sup>:

```
using System;
using System.Diagnostics;
using System.Windows.Input;
public class RelayCommand : ICommand
{
    readonly Action<object> _execute;
    readonly Predicate<object> _canExecute;
    public RelayCommand(Action<object> execute, Predicate<object>
↳ canExecute = null)
    {
        if (execute == null) throw new ArgumentNullException("execute");
        _execute = execute;
        _canExecute = canExecute;
    }
    [DebuggerStepThrough]
    public bool CanExecute(object parameter)
    {
        return _canExecute == null ? true : _canExecute(parameter);
    }
    public event EventHandler CanExecuteChanged
    {
        Add
        {
            if (_canExecute != null) CommandManager.RequerySuggested += value;
        }
        remove
        {
            if (_canExecute != null) CommandManager.RequerySuggested -= value;
        }
    }
    public void Execute(object parameter)
    {
        _execute(parameter);
    }
}
```

Możemy także skorzystać z bibliotecznych implementacji, na przykład klasy `Delegate` `↳Command<T>` z biblioteki Prism<sup>7</sup>.

Spójrzmy teraz na klasę `Student`. Jej kod jest wyraźnie dłuższy z powodu implementacji interfejsu `INotifyPropertyChanged`. Dla eksperymentu zakomentuj linię z poleceniem `OnPropertyChanged("Imie")`; i sprawdź działanie przycisku *Wyczyść dane* po tej zmianie. Wówczas kontrolka `TextBox` z imieniem nie zostanie wyczyszczona.

<sup>6</sup> Matulewski J., *MVVM i XAML w Visual Studio 2015*, Helion, Gliwice 2016 (str. 67 – 68).

<sup>7</sup> Klasa `DelegateCommand`, implementująca metody wymagane przez interfejs `ICommand` („DelegateCommand Class”): [https://msdn.microsoft.com/en-us/library/gg431410\(v=pandp.50\).aspx](https://msdn.microsoft.com/en-us/library/gg431410(v=pandp.50).aspx).



Implementacja interfejsu dla powiadomień jest konieczna, jeśli oczekujemy automatycznych aktualizacji powiązanych danych. Tu także można usprawnić sobie pracę, stosując uogólnione klasy pomocnicze, na przykład abstrakcyjną klasę obserwacji, w której będzie definicja zdarzenia `PropertyChangedEventHandler` i metody wywoływanej podczas powiadomień<sup>8</sup>. Taka klasa przydałaby się już w naszym programie, ponieważ w dwóch miejscach mamy implementację interfejsu dla powiadomień.

Do definiowania kolekcji zaleca się używanie klasy `ObservableCollection<T>`, która była stosowana w programach omawianych w tym podręczniku. Klasa ta ma szczególne znaczenie dla wzorca MVVM, ponieważ implementuje zarówno interfejs powiadomień dla kolekcji `INotifyCollectionChanged`, jak i dla właściwości `INotifyPropertyChanged`<sup>9</sup>.

Innym cennym udogodnieniem może być stosowanie automatycznego lokalizatora modeli widoków dla poszczególnych widoków. Każdy widok musi mieć swój model widoku; w naszym programie jawnie go przypisaliśmy do widoku (definiując kontekst):

```
<Window.DataContext>
    <local:StudentViewModel/>
</Window.DataContext>
```

Można to jednak zrobić inaczej, ustawiając jedynie odpowiednią właściwość tak, aby klasa lokalizatora automatycznie podłączyła stosowny model widoku<sup>10</sup>.

Nie pozostaje mi nic innego jak zachęcić Czytelnika do rozpoznania wymienionych oraz pozostałych rozwiązań wspierających wzorec MVVM. Zagłębiając się w tę tematykę, można dostrzec nakładanie się różnych komplementarnych koncepcji i wzorców w ramach wzorca MVVM. Stosowanie wzorca projektowego MVVM wspomagają frameworki takie jak Prism, MVVM Light Toolkit czy Caliburn.Micro.

---

<sup>8</sup> Kod klasy `ObservableObject` jest umieszczony w: Matulewski J., *MVVM i XAML w Visual Studio 2015*, Helion, Gliwice 2016 (str. 50). W bibliotece Prism dostępna jest klasa `BindableBase`, implementująca `INotifyPropertyChanged` („BindableBase Class”): [https://msdn.microsoft.com/en-us/library/microsoft.practices.prism.mvvm.bindablebase\(v=pandp.50\).aspx](https://msdn.microsoft.com/en-us/library/microsoft.practices.prism.mvvm.bindablebase(v=pandp.50).aspx).

<sup>9</sup> Wskazane jest, aby Czytelnik pogłębił wiedzę na temat klasy `ObservableCollection` i interfejsów, jakie ona implementuje (`INotifyCollectionChanged` i `INotifyPropertyChanged`). Klasa `ObservableCollection` w dokumentacji MSDN („ObservableCollection Class”): [https://msdn.microsoft.com/en-us/library/ms668604\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms668604(v=vs.110).aspx).

<sup>10</sup> Lokalizator modelu widoku w bibliotece Prism („Implementing the MVVM Pattern Using the Prism Library”): [https://msdn.microsoft.com/en-us/library/gg405484\(v=pandp.40\).aspx#sec20](https://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx#sec20). Przykład z użyciem lokalizatora modelu widoku: [https://www.tutorialspoint.com/mvvm/mvvm\\_hooking\\_up\\_viewmodel.htm](https://www.tutorialspoint.com/mvvm/mvvm_hooking_up_viewmodel.htm).



## Rozdział 14.

# Trochę teorii na temat WPF

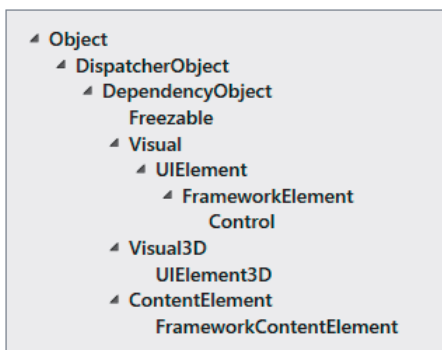
Niniejsza książka wspomaga wstępny etap nauki WPF. Ograniczanie teoretycznych informacji na samym początku i przejście do praktycznego poznawania WPF pozwala się oswoić z tą zaawansowaną technologią. To powinno pomóc podczas dalszej nauki, która wymagać będzie zarówno pogłębiania teoretycznych aspektów, jak i doskonalenia praktycznych umiejętności. W tym rozdziale omawiam hierarchię klas WPF, jej najważniejsze klasy, ze szczególnym uwzględnieniem klasy `Control`. Na koniec wskażę dalsze kierunki nauki dla Czytelnika tej książki.

## 14.1 Hierarchia klas WPF

Jak już wspominałam w rozdziale 1., diagram dziedziczenia klas WPF ma dość rozbudowaną strukturę. W pierwszej kolejności warto zapoznać się z najczęściej używanymi klasami. Rysunek 14.1 przedstawia uproszczony diagram klas WPF, które omówię w dalszej części.

### Rysunek 14.1.

*Uproszczony diagram  
dziedziczenia klas WPF*



**Object** — po tej klasie dziedziczą wszystkie klasy .NET i oczywiście nie jest to klasa specyficzna dla WPF.

**DispatcherObject** — obiekt dziedziczący po tej klasie jest dostępny jedynie dla tego wątku, w którym został utworzony. Klasa ta jest związana z klasą **Dispatcher**, która zarządza kolejką komunikatów.

**DependencyObject** — ta klasa zapewnia funkcjonowanie właściwości zależnych, mających kluczową rolę dla WPF.

**Freezable** — klasa ta pozwala na zamrażanie obiektów w celach wydajnościowych. Obiekty będące w stanie zamrożenia mogą być współdzielone przez różne wątki. Zamraża się takie elementy graficzne jak pióra, pędzle, geometrie i animacje<sup>1</sup>.

**Visual** — klasa ta zapewnia wsparcie dla renderowania w WPF. Jest bazowa dla obiektów 2D.

**UIElement** — klasa bazowa dla obiektów wizualnych 2D z obsługą kierowanych zdarzeń, wiązania poleceń, układów graficznych<sup>2</sup>.

**FrameworkElement** — klasa udostępnia rozszerzenia względem klasy **UIElement**. Do możliwości sterowania układem dodaje takie właściwości jak **HorizontalAlignment**, **VerticalAlignment** czy **Margin**. Klasa definiuje właściwość **Style** i wspiera mechanizm wiązania danych. Ponadto udostępnia takie funkcjonalności jak podpowiedzi **ToolTip** („dymki”) i menu kontekstowe. Uproszczony diagram dziedziczenia dla tej klasy przedstawia rysunek 14.2.

**Control** — klasa bazowa dla kontroltek. Wykorzystaliśmy w naszych programach kilkanaście różnych kontroltek, na przykład **Button**, **CheckBox**, **Label**, **ComboBox**, **ListBox**, **ListView**, **DataGrid**. Do tematu kontroltek wrócimy jeszcze w kolejnym podrozdziale. Klasa **Control** dodaje do **FrameworkElement** takie właściwości jak **FontSize**, **Padding**, **Background** czy **FontStyle**, a także wsparcie dla szablonów omówionych w podrozdziale 11.4.

**Visual3D** — klasa podstawowa dla obiektów 3D.

**UIElement3D** — klasa ta zapewnia podobną funkcjonalność jak klasa **UIElement**, ale dla obiektów 3D.

**ContentElement** — klasa ta ma funkcjonalność podobną jak klasa **UIElement**, ale tylko dla elementów związanych z dokumentami, które nie mają własnej reprezentacji graficznej.

**FrameworkContentElement** — klasa ta dodaje do klasy bazowej **ContentElement** wsparcie dla dodatkowych funkcji API (w tym podpowiedzi **ToolTip** i menu kontekstowe), kontekst danych dla wiązania danych oraz wsparcie dla stylów. Pozwala zapewnić elementom dokumentu funkcje, jakie mają zwykłe kontrolki<sup>3</sup>.

<sup>1</sup> Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 67 – 68).

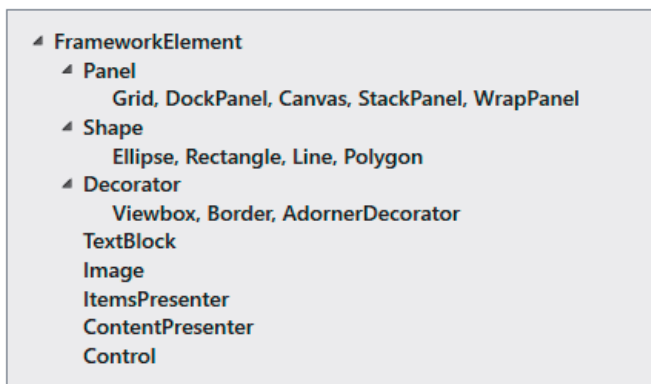
<sup>2</sup> *Ibid.*

<sup>3</sup> Cisek J., *Tworzenie nowoczesnych aplikacji graficznych w WPF*, Helion, Gliwice 2012 (str. 59 – 61).

W niniejszym podręczniku skupialiśmy się na elementach wizualnych, które wywodzą się z klasy `FrameworkElement`. Rysunek 14.2 zawiera tylko niektóre klasy wywodzące się od klasy `FrameworkElement`, większość z nich była wykorzystywana w programach w tej książce.

**Rysunek 14.2.**

*Uproszczony diagram  
dziedziczenia dla klasy  
`FrameworkElement`*



Wśród klas potomnych klasy `FrameworkElement` znajdują się klasy paneli, kształtów (np. `Rectangle`), `Decorator`, `TextBlock` oraz `Image`. W kolejnym podrozdziale przyjrzymy się bliżej kontrolkom, czyli elementom wywodzącym się od klasy `Control`.

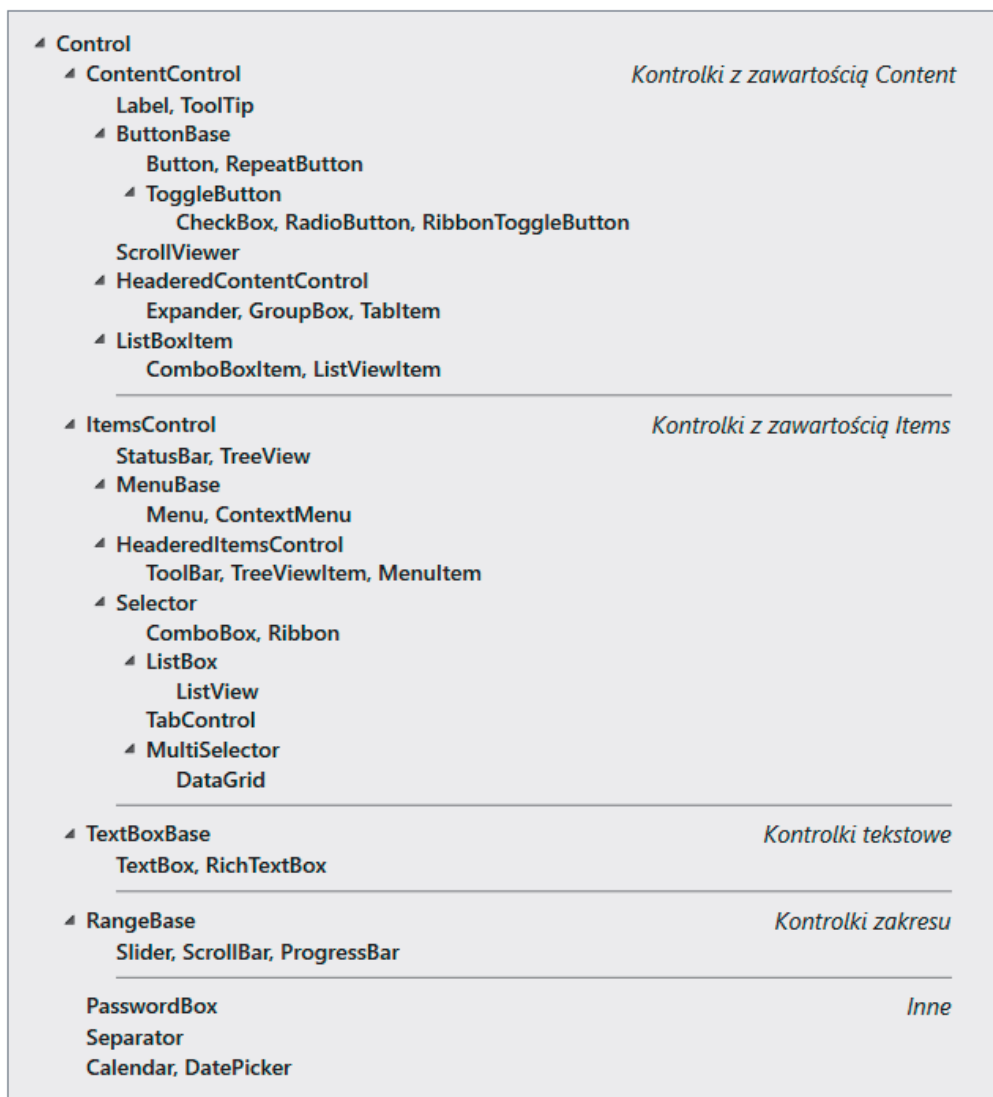
## 14.2 Kontrolki

W programach pisanych w tym podręczniku Czytelnik miał okazję poznać kilkanaście różnych kontroltek, a w tym podrozdziale uporządkujemy wiedzę na ten temat. Podczas dalszej nauki WPF i pisania coraz bardziej zaawansowanych programów przydatna będzie znajomość powiązań między kontrolkami wynikających z hierarchii klas. Rysunek 14.3 przedstawia uproszczony diagram dziedziczenia dla klasy `Control`.

Wszystkie kontrolki można podzielić na kilka grup:

- ♦ kontrolki z zawartością wpisywaną do właściwości `Content`;
- ♦ kontrolki z zawartością wpisywaną do właściwości `Items`;
- ♦ kontrolki tekstowe;
- ♦ kontrolki zakresu;
- ♦ inne.

W dalszej części omówione zostaną poszczególne grupy kontroltek.



**Rysunek 14.3.** Uproszczony diagram dziedziczenia dla klasy *Control*

## Kontrolki z zawartością wpisywaną do właściwości Content

Właściwość `Content` pozwala zapisać tylko jeden obiekt. Przykładowo przycisk `Button` może być zdefiniowany jako: `<Button Content="Zapisz"/>` czy `<Button>Zapisz</Button>`. Możemy też wykorzystać właściwość `Content` i wpisać do niej jakiś inny element, na przykład `Label`, `Image` lub `TextBlock`, ale bezpośrednio tylko jeden:

```

<Button>
  <Button.Content>
    <TextBlock Text="Zapisz" Background="White"/>
  </Button.Content>
</Button>

```

Gdybyśmy chcieli mieć na przycisku dwa elementy, na przykład obrazek i tekst, to musielibyśmy najpierw te dwa elementy scalić (opakować) w „coś jednego”, na przykład za pomocą `StackPanel`:

```

<Button MaxHeight="50" MaxWidth="150">
  <Button.Content>
    <StackPanel Orientation="Horizontal">
      <Image Source="Rysunki/glosnik.png" MaxHeight="20" MaxWidth="20"/>
      <TextBlock Text="Play" Margin="5"/>
    </StackPanel>
  </Button.Content>
</Button>

```

Kontrolki w tej grupie można podzielić na trzy podgrupy:

- ♦ kontrolki informacyjne;
- ♦ przyciski;
- ♦ kontenery z nagłówkiem.

Omówimy tu także jeszcze jedną kontrolkę, która nie należy do żadnej z wymienionych grup, mianowicie `ScrollView`.

## Kontrolki informacyjne

Do tej grupy kontrolerek należą `Label` (etykieta) i `ToolTip` („dymek” z podpowiedzią). Obie kontrolki były kilkakrotnie użyte w tworzonych w tej książce programach<sup>4</sup> i nie będziemy ich tu przedstawiać. Warto może zwrócić uwagę na pewne podobieństwo funkcjonalne kontrolki `Label` do elementu `TextBlock`. Przy czym `TextBlock` wywodzi się bezpośrednio z klasy `FrameworkElement`, a nie z klasy `Control`. Formalnie nie jest zatem kontrolką (ale bywa tak nazywany). Popatrzmy na kod XAML:

```

<StackPanel>
  <TextBlock Text="Dowolny tekst"/>
  <Label Content="Inny tekst"/>
</StackPanel>

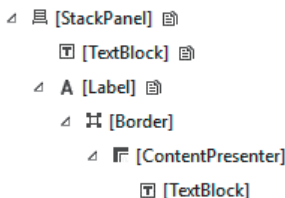
```

Prezentowany kod XAML stanowi jednocześnie drzewo logiczne dla opisywanych elementów: panel `StackPanel` zawiera dwa elementy `TextBlock` i `Label`. Drzewo wizualne natomiast pokazuje nam wszystkie szczegóły (rysunek 14.4).

<sup>4</sup> Przykład użycia etykiety `Label` znajduje się m.in. w podrozdziale 3.1. Natomiast przykład użycia `ToolTip` można znaleźć w podrozdziale 9.1.

**Rysunek 14.4.**

Drzewo wizualne dla panelu *StackPanel* zawierającego dwa elementy *TextBlock* i *Label*



Jak widać, *Label* w swoim drzewie wykorzystuje *TextBlock*, ale „opakowuje” go w pewne dodatkowe komponenty. Kontrolka *Label* może być oczywiście wypełniona dowolną zawartością, niekoniecznie blokiem tekstu. Kontrolki *Label* i *ToolTip* wraz kontrolką *Frame* należą do tak zwanych kontenerów prostych<sup>5</sup>.

## Przyciski (kontrolki wywodzące się z klasy *ButtonBase*)

Wszystkie przyciski wywodzą się z klasy *ButtonBase*. Oprócz przycisku *Button* jest jeszcze kilka innych, w tym *CheckBox* i *RadioButton* (dwa ostatnie wywodzą się bezpośrednio z klasy *ToggleButton*). Wszystkie wymienione, to znaczy *Button*, *CheckBox* oraz *RadioButton*, były wykorzystywane w naszych programach<sup>6</sup>. Po klasie *ButtonBase* dziedziczy także przycisk *RepeatButton*, który zachowuje się podobnie jak *Button*, z tą jednak różnicą, że wywołuje zdarzenie *Click* tak długo, jak długo jest pozostaje wciśnięty<sup>7</sup>.

## Kontenery z nagłówkiem

Kontenery z nagłówkiem mają oprócz zawartości także nagłówek (ang. *header*). WPF dostarcza dwa takie kontenery (dziedziczą po klasie *HeaderedContentControl*), mianowicie *GroupBox* i *Expander*. Program z użyciem klasy *Expander* został omówiony w podrozdziale 7.6, natomiast prosty przykład obrazujący działanie kontrolki *GroupBox* przedstawimy tutaj. *GroupBox* wykorzystuje się do grupowania innych elementów. Musimy jednak pamiętać, że jest to kontrolka zawartości i może zawierać tylko „coś jednego”. Zatem grupę elementów należy umieścić za pomocą innego elementu, na przykład panelu. Popatrzmy na przykład:

```

<GroupBox Header="Opcje zapisywania:" Margin="10">
  <StackPanel>
    <CheckBox>Włącz autozapisywanie</CheckBox>
    <CheckBox>Sprawdzaj przed zapisem</CheckBox>
    <CheckBox>Zapisuj tylko bieżące zadanie</CheckBox>
  </StackPanel>
</GroupBox>

```

<sup>5</sup> Kontrolka *Frame* (nie ma jej na rysunku przedstawiającym uproszczony diagram dziedziczenia klasy *Control*) dziedziczy bezpośrednio po klasie *ContentControl*. Opis klasy *Frame* („Frame Class”): [https://msdn.microsoft.com/en-us/library/system.windows.controls.frame\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.controls.frame(v=vs.110).aspx).

<sup>6</sup> Przycisk *Button* był wykorzystywany w większości zadań. Przykład użycia *CheckBox* można znaleźć w podrozdziale 3.2, natomiast *RadioButton* w podrozdziale 9.1.

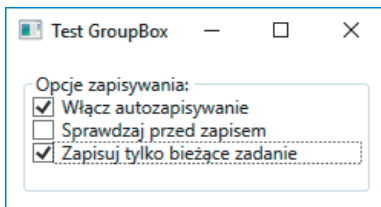
<sup>7</sup> Opis klasy *RepeatButton* wraz z przykładem („RepeatButton Class”): [https://msdn.microsoft.com/en-us/library/system.windows.controls.primitives.repeatbutton\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.controls.primitives.repeatbutton(v=vs.110).aspx).



Tu kontener zawiera kilka kontrolki `CheckBox`. Tak zdefiniowany kontener `GroupBox` będzie miał wygląd zaprezentowany na rysunku 14.5.

**Rysunek 14.5.**

Test kontrolki  
`GroupBox`



Zakończymy omawiać grupę kontrolki z zawartością `Content` na kontrolce `ScrollViewer`, która umożliwia przewijanie zawartości przy użyciu pasków przewijania. Kontrolka ta ma właściwości decydujące o widoczności paska przewijania (realizowanego przez kontrolkę `ScrollBar`), osobno w pionie i poziomie: `VerticalScrollBarVisibility` i `HorizontalScrollBarVisibility`. Zarówno jedna, jak i druga właściwość może przyjąć jedną z czterech wartości:

- ♦ **Visible** — pasek przewijania jest widoczny (to ustawienie jest domyślne dla `VerticalScrollBarVisibility`);
- ♦ **Auto** — pasek przewijania jest widoczny tylko wówczas, gdy zawartość się nie mieści;
- ♦ **Hidden** — pasek przewijania jest niewidoczny, ale logicznie istnieje, więc można przewijać zawartość za pomocą klawiszy strzałek (to ustawienie jest domyślne dla `HorizontalScrollBarVisibility`);
- ♦ **Disabled** — pasek przewijania jest niewidoczny i nie działa przewijanie.

Spójrzmy na przykład kontrolki `ScrollViewer`:

```
<ScrollViewer HorizontalScrollBarVisibility="Auto">
  <StackPanel>
    <TextBlock Text="Długi tekst pozwalający przetestować właściwość
      HorizontalScrollBarVisibility" Margin="10"/>
    <TextBlock Text="Długi tekst pozwalający przetestować właściwość
      HorizontalScrollBarVisibility" Margin="10"/>
  </StackPanel>
</ScrollViewer>
```

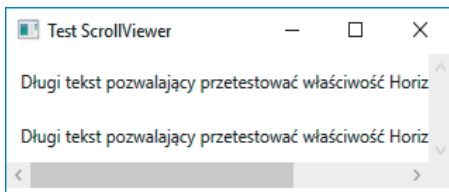
Kontrolka `ScrollViewer` zawiera tu panel z dwoma dłuższymi tekstami, które są przewijane w poziomie. Oczywiście kontrolka ta może zawierać różne obiekty (nie tylko teksty). Jak każda kontrolka z zawartością `Content` może przechować tylko jeden element, zatem kilka obiektów należy umieścić w jednym elemencie (w przykładzie użyto `StackPanel`). Rysunek 14.6 przedstawia wygląd okna dla prezentowanego przykładu.

Zamiast przewijania zawartości elementów stosować można skalowanie. Prosty mechanizm skalowania dostarcza element `Viewbox`, który nie jest kontrolką (dziedziczy po klasie `Decorator`)<sup>8</sup>.

<sup>8</sup> Skalowanie za pomocą elementu `Viewbox`: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 137 – 140).

**Rysunek 14.6.**

Panel z tekstami  
umieszczony  
w kontrolce  
ScrollView



Wracamy do głównego węzła w diagramie dziedziczenia przedstawionym na rysunku 14.3, mianowicie do klasy `Control`. Omówiłam kontrolki zawartości wywodzące się z klasy `ContentControl`, teraz przejdziemy do kontrolki wywodzących się z klasy `ItemsControl`.

## Kontrolki z zawartością `Items`

W zawartości kontrolki dziedziczących po klasie `ItemsControl` można umieszczać więcej danych (np. kolekcje). Zawartość kontrolki z tej grupy umieszczana jest we właściwości `Items`, która jest typu `ItemCollection`. Poszczególne elementy takiej kontrolki mogą być dowolnymi obiektami.

Właściwość `Items` jest tylko do odczytu (ma zdefiniowany sam akcesor `get`). W klasie `ItemsControl` dostępna jest także właściwość `ItemsSource`, która umożliwia ustawienie zawartości na podstawie istniejącej kolekcji. Właściwość `ItemsSource` używaliśmy w tym podręczniku do wiązania danych. Kontrolki klasy `ItemCollection` mają także przydatną właściwość `HasItems` (typu `bool`), która zwraca wartość `True` w przypadku braku pozycji (czyli pustej kolekcji).

Kontrolki z zawartością `Items` podzielimy na trzy grupy:

- ◆ selektory;
- ◆ kontrolki menu;
- ◆ inne kontrolki z zawartością `Items`.

### Selektory

Selektory to kontrolki dziedziczące po klasie `Selector`. Klasa ta pozwala na indeksowanie elementów oraz dokonywanie wyboru. Klasa `Selector` ma właściwości specyficzne dla operacji wykonywanych w tej grupie kontrolki. `SelectedIndex` ustawia lub zwraca numer zaznaczonego elementu (wartość `-1` w przypadku braku zaznaczenia). Właściwość `SelectedItem` pozwala ustawić lub zwrócić zaznaczoną pozycję (`null` w przypadku braku zaznaczenia).

Wśród selektorów jest lista rozwijana `ComboBox`. Używaliśmy tej kontrolki (m.in. w podrozdziale 3.2), ale nie wszystkich jej funkcjonalności. Przykładowo można sterować sposobem wyboru elementów z listy za pomocą właściwości `IsEditable`, której domyślnym ustawieniem jest `False`. Jeśli ustawimy ją na `True`, to wówczas pojawi się pole,

w którym będzie można wpisywać tekst i na tej podstawie dokonać zaznaczenia (wyboru) pozycji<sup>9</sup>.

Kolejnym selektorem jest znana nam kontrolka `ListBox` (przykład jej użycia znajduje się w podrozdziale 11.2). Kontrolka ta umożliwia wybór zarówno pojedynczego elementu, jak i większej liczby pozycji. Decyduje o tym właściwość `SelectionMode`, mogąca przyjąć trzy wartości: `Single`, `Multiple` oraz `Extended`. Jeżeli jest ustawiona na `Single` (tak jest domyślnie), to wówczas można wybrać tylko jedną pozycję z listy. W przypadku ustawień `Multiple` i `Extended` można zaznaczyć więcej pozycji, które są dodawane do kolekcji `SelectedItems`. Oba ostatnie tryby różnią się sposobem oznaczania pozycji: w trybie `Multiple` każde kliknięcie pozycji niezaznaczonej dodaje ją do kolekcji `SelectedItems`, natomiast w trybie `Extended` zbiorowe zaznaczenie możemy uzyskać, używając dodatkowo klawiszy `Ctrl` (pojedyncze dodawanie do kolekcji) lub `Shift` (zaznaczenie zakresu).

Jak możemy zauważyć na rysunku 14.3, po klasie `ListBox` dziedziczy klasa `ListView`. Kontrolkę `ListView` stosowaliśmy w rozdziale 6. i tam została ona dość dokładnie omówiona.

Podobna w działaniu do `ListView` jest kontrolka `DataGrid`. Klasa `DataGrid` dziedziczy po klasie `MultiSelector`. Klasa ta ma więcej funkcjonalności niż `ListView`, ważniejsze spośród nich były prezentowane w rozdziale 7.

Selektorem jest także kontrolka `TabControl`, pozwalająca tworzyć zakładki. Domyślnie wybrana jest zakładka pierwsza (o indeksie 0), co można zmienić przy użyciu właściwości `SelectedItem`. Przykład zastosowania `TabControl` znajduje się w rozdziale 9.

Do selektorów należy również kontrolka `Ribbon`, której nie używaliśmy w tym podręczniku. `Ribbon` pozwala tworzyć interfejs w postaci wstążki zawierającej pasek narzędzi szybkiego dostępu<sup>10</sup>.

## Kontrolki Menu

W grupie kontrolki z zawartością `Items` dziedziczących po klasie `ItemsControl` znajdują się także kontrolki menu. WPF obsługuje dwa rodzaje menu: menu aplikacji i menu podręczne (kontekstowe). Za pierwsze z nich odpowiada klasa `Menu`, a za drugie klasa `ContextMenu`. Obie te klasy dziedziczą po klasie `MenuBase`. Przykłady zastosowania obu rodzajów menu zostały przedstawione w rozdziale 8. Poszczególne opcje i podopcje umieszczane są za pomocą elementu `MenuItem`. Klasa `MenuItem` udostępnia właściwość `IsCheckable`, dzięki której pozycja menu może przypominać wyglądem i funkcjonalnością kontrolkę `CheckBox`. Ma także właściwość `Icon`, która umożliwia wstawienie obrazka obok tytułu elementu menu.

<sup>9</sup> Takie działanie właściwości `IsEditable` może zostać zmienione przez ustawienie innej właściwości — `IsReadOnly`. Jeśli ta druga właściwość ma wartość `True`, to nie ma możliwości wpisywania tekstu.

<sup>10</sup> Opis kontrolki `Ribbon` można znaleźć m.in. w: Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015 (str. 285 – 298), oraz na stronie dokumentacji MSDN („Ribbon Class”): [https://msdn.microsoft.com/en-us/library/system.windows.controls.ribbon.ribbon\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.controls.ribbon.ribbon(v=vs.110).aspx).

## Inne kontrolki z zawartością Items

Wśród pozostałych kontrolki w tej grupie możemy wymienić kontrolki `TreeView` i `StatusBar` dziedziczące bezpośrednio po klasie `ItemsControl` oraz kontrolkę `ToolBar` wywodzącą się bezpośrednio od `HeaderedItemsControl`. Kontrolka `TreeView` umożliwia prezentację danych mających hierarchiczną strukturę, a przykład jej zastosowania został omówiony w podrozdziale 8.4. Definiuje się ją podobnie jak `Menu`. W elementach tej kontrolki można umieszczać dowolne inne obiekty. Kontrolka ta pod pewnymi względami zachowuje się podobnie jak selektor, ale nie jest selektorem (nie dziedziczy po klasie `Selector`). Umożliwia jednak wybór elementu i ma właściwości `SelectedItem` i `SelectedValue` (przy czym nie ma właściwości `SelectedIndex`).

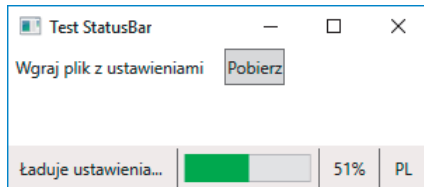
`StatusBar` to pasek stanu, który służy do prezentowania stanu aplikacji. Kontrolka ta nie była wykorzystywana w podręczniku. Prosty przykład definicji `StatusBar` przedstawię tutaj:

```
<DockPanel>
  <StatusBar DockPanel.Dock="Bottom">
    <StatusBarItem>
      <Label Content="Ładuje ustawienia..." />
    </StatusBarItem>
    <Separator />
    <ProgressBar Width="90" Height="20" Value="51" />
    <Separator />
    <StatusBarItem>
      <Label Content="51%" />
    </StatusBarItem>
    <Separator />
    <StatusBarItem>
      <Label Content="PL" />
    </StatusBarItem>
  </StatusBar>
</DockPanel>
<WrapPanel>
  <Label Content="Wgraj plik z ustawieniami" />
  <Button Content="Pobierz" Margin="10,0" />
</WrapPanel>
</DockPanel>
```

Tak zdefiniowany pasek stanu będzie widoczny w dolnej części okna, co pokazuje rysunek 14.7.

### Rysunek 14.7.

Przykładowy  
wygląd kontrolki  
`StatusBar`



Elementami kontrolki `StatusBar` mogą być dowolne obiekty. Jawne umieszczanie poszczególnych elementów w `StatusBarItem` nie jest konieczne (bez tego zostałyby niejawnie umieszczone), ale w niektórych sytuacjach jest to przydatne ze względu na

możliwość korzystania z właściwości dołączanych dostępnych dla `StatusBarItem`. W kontrolce `StatusBar` korzysta się podobnie jak w `Menu` z separatora, przy czym dla `StatusBar` separator jest linią pionową.

Kontrolka `ToolBar` pozwala zdefiniować pasek narzędzi, a jej elementami zazwyczaj są małe przyciski lub inne kontrolki (np. lista rozwijana). Prostej paska narzędziowego `ToolBar` użyliśmy w podrozdziale 8.1.

Ponownie wracamy do głównego korzenia w diagramie dziedziczenia z rysunku 14.3, to znaczy do klasy `Control`. Omówiliśmy dwie bardziej złożone „rodziny” kontrolki, wywodzące się od klas `ContentControl` i `ItemsControl`. Teraz przejdziemy do klasy `TextBoxBase`, która jest bazowa dla kontrolki pozwalających wpisywać tekst.

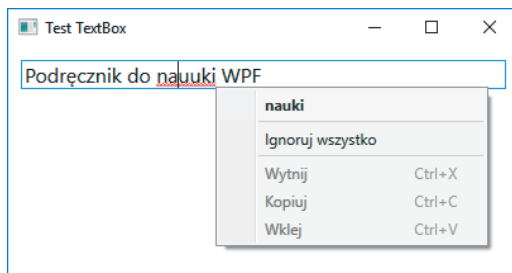
## Kontrolki tekstowe

Po klasie `TextBoxBase` dziedziczą dwie kontrolki tekstowe: `TextBox` i `RichTextBox`. Pole tekstowe `TextBox` było w tym podręczniku używane wielokrotnie, począwszy od rozdziału 3. Można by tu dodać jedynie, że `TextBox` pozwala sprawdzać pisownię tekstu. Należy w tym celu ustawić właściwość `SpellCheck.IsEnabled` na `True`. Przykładowe pole tekstowe:

```
<TextBox SpellCheck.IsEnabled="True" Text="Podręcznik do nauki WPF"
        Language="pl-pl" />
```

będzie obsługiwać sprawdzanie pisowni w języku polskim. Przy użyciu menu kontekstowego można wskazać poprawną wersję danego wyrazu, co przedstawia rysunek 14.8.

**Rysunek 14.8.**  
*Sprawdzanie pisowni  
w kontrolce TextBox*



Można zawijać wiersz w polu tekstowym, odpowiednio ustawiając właściwość `TextWrapping` (przykład takiego pola tekstowego znajduje się w podrozdziale 7.6).

Kontrolka `RichTextBox` także pozwala wprowadzać tekst, ale ma więcej możliwości, przede wszystkim umożliwia formatowanie tekstu. Tekst w przypadku tej kontrolki jest przechowywany jako obiekt klasy `FlowDocument` we właściwości `Document`. W kontrolce tej można umieszczać nie tylko zwykły tekst, ale także takie obiekty jak obrazki. Nie stosowaliśmy w naszych programach tej kontrolki i tu przedstawię jej prosty przykład:

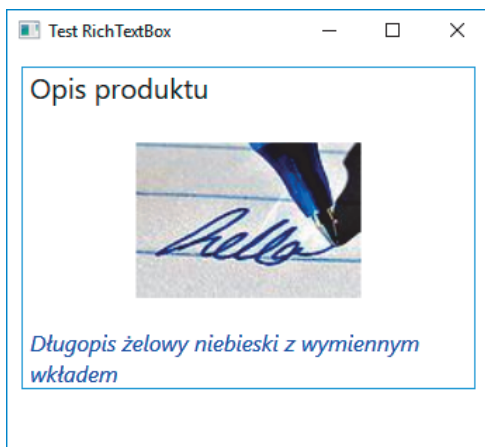
```
<RichTextBox Margin="10">
    <FlowDocument>
        <Paragraph FontSize="20">0pis produktu</Paragraph>
```

```
<Paragraph FontStyle="Italic" FontSize="16" Foreground="Blue">  
    Długopis żelowy niebieski z wymiennym wkładem  
</Paragraph>  
</FlowDocument>  
</RichTextBox>
```

Rysunek 14.9 przedstawia wyżej zdefiniowany element `RichTextBox`, przy czym po uruchomieniu programu wkopiewano obrazek pomiędzy dwoma akapitami.

#### Rysunek 14.9.

Przykład użycia  
`RichTextBox`



Po raz kolejny wracamy do węzła głównego w diagramie dziedziczenia z rysunku 14.3, czyli do klasy `Control`. Zostały nam jeszcze do omówienia kontrolki zakresu i pozostałe.

## Kontrolki zakresu

Kontrolki zakresu dziedziczą po klasie `RangeBase`. Pozwalają prezentować zakres dla wykonywanej operacji za pomocą trzech właściwości: `Value`, `Minimum` oraz `Maximum`. Pierwsza z nich dotyczy aktualnej wartości, natomiast dwie pozostałe określają wartości brzegowe dla zakresu. W tej grupie są trzy kontrolki: `ProgressBar`, `Slider` oraz `ScrollBar`.

`ProgressBar` pozwala prezentować pasek postępu. Zastosowaliśmy klasyczny pasek postępu w rozdziale 9. Domyślnie `ProgressBar` jest ułożony poziomo, można to zmienić za pomocą właściwości `Orientation`. Kontrolka `Slider` (suwak) pozwala użytkownikowi zmieniać położenie uchwyty. Przykłady użycia suwaka można znaleźć w podrozdziałach 5.1 i 9.4. Kontrolka `ScrollBar` udostępnia pasek przewijania przydatny do oglądania obiektów niemieszczących się w zadanym obszarze. Samodzielnie jest coraz rzadziej stosowana, częściej pośrednio, poprzez użycie omówionej już kontrolki `ScrollView`.

## Pozostałe kontrolki

W ostatniej grupie kontroltek omówione zostaną kontrolki `PasswordBox`, `Separator` oraz dwie związane z datą: `Calendar` i `DatePicker`. Wszystkie one dziedziczą bezpośrednio po klasie `Control`.

Kontrolka `PasswordBox` jest elementem podobnym do pola tekstowego, ale służy do wprowadzania tekstu bez jego pokazywania, co zazwyczaj wykorzystuje się do wprowadzania hasła. W miejsce każdego wpisywanego znaku na ekranie wyświetla się znak maskujący. Domyślnie jest to znak punktora („•”), ale można go zmienić na inny znak (stosując właściwość `PasswordChar`).

Przykładowa definicja kontrolki `PasswordBox`:

```
<PasswordBox MaxLength="20" PasswordChar="*" />
```

pozwoли wpisać hasło składające się z maksymalnie 20 znaków. W miejsce każdego znaku zostanie wyświetlona gwiazdka (\*).

Element `Separator` był w naszych programach kilkakrotnie używany (m.in. w rozdziale 8.1), a służy on do wizualnego oddzielenia innych elementów w postaci linii poziomej lub pionowej.

Kontrolka `Calendar` pokazuje kalendarz. Najprostszy kalendarz możemy wyświetlić na przykład następującym poleceniem:

```
<Calendar Margin="10" />
```

Wówczas pojawi się aktualny kalendarz (rysunek 14.10).

**Rysunek 14.10.**

*Kalendarz*



Domyślnie kalendarz wyświetla się jak na rysunku. Można zmienić tryb kalendarza, ustawiając odpowiednio właściwość `DisplayMode` na jedną z wartości: `Year` lub `Decade` (domyślnie jest `Month`). Jest też możliwość określenia sposobu wybierania dat przy użyciu właściwości `SelectionMode`. Właściwość ta może mieć jedną z wartości:

- ♦ `SingleDate` — pozwala wybrać jedną datę (tak jest domyślnie) i będzie ona przechowywana we właściwości `SelectedDate`.
- ♦ `SingleRange` — pozwala wybrać kilka dat z ciągłego zakresu. Dаты z podanego zakresu przechowywane są we właściwości `SelectedDates`.

- ◆ **MultipleRange** — pozwala wybrać dowolne daty. Wskazane daty przechowywane są we właściwości **SelectedDates**.
- ◆ **None** — nie można wybierać dat.

Można wskazać zakres (lub zakresy) dat w kalendarzu, jakie mają być niemożliwe do wyboru. Stosuje się w tym celu właściwość **BlackoutDates**<sup>11</sup>.

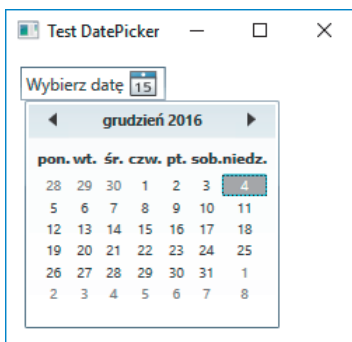
Kontrolka **DatePicker** wyświetla pole tekstowe do wprowadzenia daty z możliwością rozwinięcia okna kalendarza. Przykładowa definicja kontrolki może mieć następującą postać:

```
<DatePicker HorizontalAlignment="Left" MaxWidth="140"/>
```

Po uruchomieniu programu pojawia się pole tekstowe, gdzie można wpisać lub wybrać datę z kalendarza. Tę drugą możliwość przedstawia rysunek 14.11.

**Rysunek 14.11.**

Kontrolka  
*DatePicker*



W tym podrozdziale dokonaliśmy uporządkowania wiedzy na temat kontroltek, uzupełniając ją o krótkie prezentacje kilku kontroltek niewykorzystanych w naszych programach. Domyślny wygląd kontroltek można dość istotnie zmienić, stosując szablony kontroltek (wprowadzenie do tego tematu zawiera podrozdział 11.4).

## 14.3 Kierunki dalszej nauki WPF

Głównym zadaniem tego podręcznika było łagodne i praktyczne wprowadzenie do programowania z użyciem WPF. Przedstawiłam tu jedynie część możliwości, jakich dostarcza to rozbudowane narzędzie. Przede wszystkim skupiłam się na podstawowych funkcjonalnościach, takich jak tworzenie standardowych kontroltek i paneli, wykorzystanie mechanizmu wiązania danych, definiowanie stylów, szablonów i wyzwalaczy. Takie umiejętności mogą wystarczyć do napisania całkiem ciekawej aplikacji o charakterze biznesowym, w wielu jednak przypadkach może wystąpić konieczność wzbogacenia wizualnych walorów aplikacji o bardziej zaawansowane możliwości w zakresie

<sup>11</sup> Opis i przykład użycia właściwości **BlackoutDates** na stronie MSDN („Calendar.BlackoutDates Property”): [https://msdn.microsoft.com/en-us/library/system.windows.controls.calendar.blackoutdates\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.controls.calendar.blackoutdates(v=vs.110).aspx).



**grafiki 2D i 3D oraz animacji.** To właśnie jest jeden z ważniejszych kierunków dalszej nauki WPF dla Czytelnika tej książki. Podczas omawiania przykładu kontrolki `RichTextBox` wspomniałam, że jej zawartość jest przechowywana jako obiekt klasy `FlowDocument`<sup>12</sup>, ale tematyka **dokumentów** tworzonych przy użyciu tej klasy nie była tu poruszana, co wymagałoby uzupełnienia, podobnie jak możliwości WPF dotyczące **transformacji i skalowania**. Należałoby także pogłębić wiedzę na temat **właściwości zależnych** i wyznaczania ostatecznej wartości właściwości na podstawie wielu źródeł. Ponadto konieczne będzie bliższe zaznajomienie się z zagadnieniami dotyczącymi **zdarzeń** w kontekście WPF. W początkowym etapie nauki może nie było potrzeby zastanawiać się, jak to się dzieje, że mamy do dyspozycji jedno proste zdarzenie `Click` dla przycisku `Button`, o którym obecnie już wiemy, że składa się z różnych elementów. To wygodne rozwiązanie zapewnia mechanizm **zdarzeń kierowanych**. Tworzenie aplikacji bardziej złożonych niż prezentowane w tej książce wymaga lepszego poznania struktury i rodzajów aplikacji WPF. W treści podręcznika, w miejscach, w których dany temat nie został w pełni wyczerpany, umieściłam krótkie wzmianki o pozostałych możliwościach. Dotyczy to między innymi tworzenia bardziej złożonych szablonów, wykorzystania innych rozwiązań w zakresie walidacji danych czy nieomówionych tu kontrolek.

Wymienione tematy moglibyśmy ująć jako poszerzony zakres podstawowy WPF i oczywiście można go dalej rozszerzać o elementy bardziej zaawansowane, w zależności od dalszych kierunków zainteresowań Czytelnika i jego potrzeb zawodowych. W szczególności kierunki te mogą dotyczyć współpracy z innymi technologiami, narzędziami pomocniczymi czy frameworkami wspierającymi stosowanie wzorca projektowego MVVM.

---

<sup>12</sup> Klasa `FlowDocument` wywodzi się z klasy `FrameworkContentElement`, przedstawionej na rysunku 14.1 (na początku rozdziału).



# Literatura

Cisek J., *Tworzenie nowoczesnych aplikacji graficznych w WPF*, Helion, Gliwice 2012.

Garofalo R., *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*, Microsoft Corporation 2011.

Kempa A., Staś T., *Wstęp do programowania w C#. Łatwy podręcznik dla początkujących*, Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice 2014.

Matulewski J., *MVVM i XAML w Visual Studio 2015*, Helion, Gliwice 2016.

Nathan A., *WPF 4.5. Księga eksperta*, Helion, Gliwice 2015.

Petzold C., *Windows 8. Programowanie aplikacji z wykorzystaniem C# i XAML*, Helion, Gliwice 2013.

## Źródła internetowe i materiały multimedialne

Biblioteka MSDN — scentralizowana baza oficjalnych dokumentów dla programistów i deweloperów zawierająca dokumentacje techniczne: <http://msdn.microsoft.com/library> (podstawowe źródło).

<http://www.wpf-tutorial.com> — tutorial WPF (język angielski).

<http://www.altcontroldelete.pl/tag/wpf-tutorial> — tutorial WPF (język polski).

<https://www.tutorialspoint.com/mvvm/index.htm> — tutorial MVVM (język angielski).

<https://www.wpfutorial.net> — tutorial WPF (język angielski).

Kempa A., kurs wideo *Język C#. Kurs video. Poziom pierwszy. Programowanie dla początkujących*, dostępny na stronie [videopoint.pl](http://videopoint.pl).

[https://commons.wikimedia.org/wiki/Main\\_Page](https://commons.wikimedia.org/wiki/Main_Page) — repozytorium wolnych zasobów. W omawianych programach zostały wykorzystane następujące obrazki pochodzące z tego repozytorium:

- ◆ <https://commons.wikimedia.org/wiki/File:Bläistift.jpg> (ołówek, wykorzystano fragment rysunku)
- ◆ [https://commons.wikimedia.org/wiki/File:2004-02-29\\_Ball\\_point\\_pen\\_writing.jpg](https://commons.wikimedia.org/wiki/File:2004-02-29_Ball_point_pen_writing.jpg) (długopis kulkowy)
- ◆ <https://commons.wikimedia.org/wiki/File:GelPen.jpg> (długopis żelowy)
- ◆ [https://commons.wikimedia.org/wiki/File:Calligraphy\\_Fountain\\_Pen.jpg](https://commons.wikimedia.org/wiki/File:Calligraphy_Fountain_Pen.jpg) (pióro)
- ◆ <https://commons.wikimedia.org/wiki/File:Loudspeaker-rtl.png> (głośnik)
- ◆ <https://commons.wikimedia.org/wiki/File:Octicons-tools.svg> (narzędzia)

Zdjęcie kompletu piśmiennego (pióro i długopis) — własność autorki.

# Skorowidz

## A

autonomiczny widok, 175

## C

code-behind, 21, 25, 27

## D

DataGrid, 85

- autogenerowanie kolumn, 85

- definiowanie kolumn, 88, 94

- kolumna z listą rozwijaną, 89

- wiązanie z XML, 90

diagram dziedziczenia klas WPF, 21, 187

dokument XML, 22, 90

dokumentacja MSDN, 20

drzewo

- logiczne, 21, 137

- prezentacji, 21, 137

- wizualne, *patrz* prezentacji

dyrektywa using, 20

dziedziczenie

- stylu, 172

- właściwości zależnych, 130

## E

element

- główny (XML), 22

- Window (XAML), 25

element, *patrz także* kontrolka

- Ellipse, 148

- Image, 52, 94, 110, 120

- Rectangle, 48, 133

- TextBlock, 61, 62, 75, 191

etykieta, 45

## F

filtrowanie, 76

fokus, 69

format Pack URI, 94

formatowanie, 74

## H

hierarchia klas WPF, 187

## I

instalacja środowiska, 13

interfejs, 18

- ICommand, 182, 183

- ICollectionView, 93, 97

- IDataErrorInfo, 156, 171

- INotifyDataErrorInfo, 158

- INotifyPropertyChanged, 180

- IValueConverter, 145

## J

jednostka px, 35

język

- C#, 15

- XAML, 22, 23

- XML, 22

## K

klasa

- Application, 28

- ApplicationCommands, 109

- Binding, 61, 69

- Brushes, 42

- ContentElement, 188

- ContentPresenter, 148, 149

- Control, 188, 189

## klasa

- DataErrorValidationRule, 156
- DataTrigger, 132
- DependencyObject, 188
- DispatcherObject, 188
- DispatcherTimer, 114
- FrameworkContentElement, 188
- FrameworkElement, 188
- Freezable, 188
- MainWindow, 27
- MultiDataTrigger, 135
- MultiTrigger, 135
- Object, 188
- ObservableCollection, 72, 185
- OpenFileDialog, 96
- Regex, 160
- Selector, 194
- Setter, 126, 127
- Trigger, 132
- UIElement, 188
- UIElement3D, 188
- ValidationResult, 159, 160
- ValidationRule, 159
- Visual, 188
- Visual3D, 188
- Window, 27
- XElement, 92

## kolory w WPF, 42

## komentarze w XAML, 24

## kontekst danych, 63

## kontrolka, 189

- Button, 45
- Calendar, 199
- CheckBox, 48
- ComboBox, 48, 49
- ContextMenu, 109, 195
- DataGrid, 85
- DatePicker, 200
- Expander, 97, 99
- GroupBox, 192
- Label, 45
- ListBox, 141
- ListView, 71
- Menu, 101, 195
- MenuItem, 109
- PasswordBox, 199
- ProgressBar, 111, 112
- RadioButton, 51, 55
- RepeatButton, 192
- Ribbon, 195
- RichTextBox, 197, 198
- ScrollBar, 198
- ScrollViewer, 193
- Separator, 199
- Slider, 61, 117

## StatusBar, 196

## TabControl, 111

## TextBox, 45, 108, 197

## ToolBar, 102, 103

## ToolTip, 110

## TreeView, 110

## WebBrowser, 102, 103, 106

## konwertery

## typów, 147

## wartości, 145

## korzeń dokumentu XML, 22

## kształty, 189

## elipsa, 148

kwadrat, *patrz* prostokąt

## prostokąt, 48, 133

**L**

## lista rozwijana, 49

## w DataGrid, 89

## ListView, 71

## filtrowanie, 76

## formatowanie, 74

## sortowanie, 73

## wyrównanie, 75

**M**

## mechanizmy walidacji, 153

## metoda

## CanExecute, 182

## Convert, 145

## ConvertBack, 145

## Close, 79

## Execute, 182

## GoBack, 105

## GoForward, 105

## IndexOf, 77, 78

## IsMatch, 164

## MessageBox.Show, 38, 81, 181

## Navigate, 105

## Open, 113, 115

## Pause, 114, 115

## Play, 114, 115

## Show, 78

## ShowDialog, 80

## Stop, 114, 115

## ToString, 65

## TryParse, 47

## Validate, 159

## Microsoft Blend, 26, 150

## Model, 176, 178

## MSDN, 20

## MVVM, Model-View-ViewModel, 175, 183

**O**

obrazek, 52, 94, 110, 120  
obsługa zdarzenia  
  kliknięcia, 47, 49, 104  
  zmiany w polu tekstowym, 47  
odtwarzacz audio, 111  
okno  
  aplikacji WPF, 33  
  dialogowe własne, 80  
  MessageBox, 38, 81, 181  
  OpenFileDialog, 96, 113, 115  
  SaveFileDialog, 104

**P**

panel  
  Canvas, 53  
  DockPanel, 56  
  Grid, 57  
  StackPanel, 54  
  WrapPanel, 56  
pasek postępu, 112, 198  
piksel, 35  
polecenie, 178, 182, 184  
powiadomienia o zmianach, 180  
pozycja elementów, 29  
przeglądarka, 101  
przestrzeń nazw, 20  
przeźroczystość, 41  
przycisk, 34, 192  
  Button, 45  
  CheckBox, 48  
  RadioButton, 51, 55  
  RepeatButton, 192

**R**

reguły walidacji, 158  
rozmiar elementów, 29  
rozszerzenia znaczników, 28  
rysunek, *patrz* obrazek

**S**

scalanie komórek w Grid, 59  
selektory, 194  
siatka, 59  
słowa kluczowe XAML, 26  
sortowanie, 73  
struktura Color, 42  
style, 126  
suwak, 62, 116

szablony  
  danych, 141  
  kontrolerek, 147

**T**

tryb wiązania danych  
  OneWay, 69  
  TwoWay, 69

**U**

układ okien aplikacji, 34  
uruchomienie aplikacji, 14

**V**

View, 176  
ViewModel, 177

**W**

walidacja danych, 153  
  pola tekstowego, 66  
  w DataGrid, 170, 171, 173  
  wyrażenia regularne, 160  
warstwa prezentacji, 33  
wiązanie  
  danych, 61  
  kolekcji danych, 71  
widok, 176, 177  
własne reguły walidacji, 158  
właściwości, 16  
  automatyczne, 17  
  dołączane, 21, 54, 131  
  zależne, 21, 130  
właściwość  
  AlternatingRowBackground, 87  
  AlternationCount, 88  
  Background, 75  
  CanGoBack, 105  
  CanGoForward, 105  
  CellTemplate, 145  
  Command, 109, 182  
  ContainerStyle, 98  
  Content, 190  
  DataContext, 63, 65, 83  
  DialogResult, 80  
  DisplayMemberBinding, 71, 72  
  ElementName, 62, 63  
  ElementStyle, 129, 172  
  Error, 156  
  Fill, 48

## właściwość

- Filter, 77
- FlowDirection, 32
- FontSize, 36
- Foreground, 75
- Grid.Column, 59
- Grid.ColumnSpan, 59, 60
- Grid.Row, 59
- Grid.RowSpan, 59, 60
- GridLinesVisibility, 87
- GroupDescriptions, 93, 97
- GroupName, 51
- Header, 71, 88, 99, 102, 110, 111, 192
- Height, 29
- HorizontalAlignment, 30
- HorizontalContentAlignment, 31
- Icon, 108, 110
- Interval, 114
- IsCheckable, 102, 103
- IsChecked, 102, 103, 133
- IsEnabled, 41, 46, 112
- IsExpanded, 110
- ItemHeight, 56
- ItemsSource, 73, 171
- ItemWidth, 56
- LastChildFill, 56, 57
- Margin, 29
- MaxHeight, 29
- MaxWidth, 29
- MinHeight, 29
- MinWidth, 29
- Mode, 69
- Name, 27
- NaturalDuration, 113, 115
- Opacity, 41, 43, 49
- Orientation, 55, 56
- Padding, 29
- Path, 62, 63
- RelativeSource, 134
- RowDetailsTemplate, 95
- SelectedIndex, 49
- SelectedItem, 79, 96, 194
- SortDescriptions, 74
- Source, 52, 94, 120
- StringFormat, 68, 70
- Stroke, 48
- TabStripPlacement, 112
- TargetType, 128
- Template, 141
- TextWrapping, 95, 102, 197
- ToolTip, 110, 154
- Triggers, 151
- UpdateSourceTrigger, 69
- ValidatesOnDataErrors, 157, 158
- Validation.ErrorTemplate, 155
- ValidationRules, 159
- VerticalAlignment, 30
- VerticalContentAlignment, 31
- Visibility, 51
- Width, 29
- ZIndex, 54
- WPF, Windows Presentation Foundation, 9, 187
- wrażenia regularne, 160
- wyrównywanie, 30, 31
  - tekstu w ListView, 75
- wyzwalacze 131
  - danych, 131, 132
  - warunki logiczne, 135
  - właściwości, 131
  - zdarzeń, 131
- wzorce projektowe, 175
- wzorec MVVM, 175

**X**

XAML, 22, 23

XML, 22

**Z**

zakładki, 111

zasięg zasobu, 122

zasoby

- binarne, 119

- logiczne, 120

- statyczne i dynamiczne, 124

zdarzenie

- CanExecuteChanged, 182

- Checked, 51, 102, 103, 112

- Click, 38

- KeyUp, 105, 106

- MouseDoubleClick, 78

- MouseEnter, 39

- MouseLeave, 39

- Navigated, 106

- Navigating, 106

- PropertyChanged, 180

- TextChanged, 47

- Tick, 114

zmiana właściwości zasobu, 120

znacznik 22

- otwierający, 22

- rozszerzenia znaczników, 28

- zamykający, 22