Część IV

Tworzenie aplikacji

W części IV zrobisz kolejny wielki krok w kierunku zostania pełnoprawnym programistą. Po tej lekturze będziesz w stanie tworzyć programy, które będziesz mógł sprzedawać. Pamiętaj, że jedyna różnica między naszymi przykładami a „prawdziwymi" programami polega na tym, że za te drugie się płaci.

Zaczniemy teraz odchodzić od korzystania z frameworku Snaps, który ukrywa pewne podstawowe złożoności związane z tworzeniem oprogramowania dla platformy Windows 10. Nie porzucimy jednak tego frameworku całkowicie. Dowiesz się, jak działają elementy Snaps i jak używać ich funkcjonalności w pisanych przez Ciebie programach. Zaczniesz od nauki projektowania interfejsów użytkownika, a następnie przejdziesz do kwestii związanych z tym, jak nowoczesne programy obsługują zewnętrzne zdarzenia.

16.

Tworzenie interfejsu użytkownika z wykorzystaniem obiektów

Czego nauczysz się w tym rozdziale?

Interfejs użytkownika to witryna sklepowa programu. Podobnie jak okno wystawowe może skusić Cię, żeby wejść do sklepu i zakupić jakieś produkty, dobrze zaprojektowany interfejs użytkownika może zachęcić użytkowników do korzystania z oprogramowania. W tym rozdziale zastanowimy się nad tym, jak tworzone są interfejsy użytkownika nowoczesnych aplikacji. Dowiesz się, jak wykorzystywać komponenty oprogramowania do reprezentowania elementów, z którymi użytkownik wchodzi w interakcje. Następnie zobaczysz, w jaki sposób programy współpracują z tymi elementami i jak te elementy reagują na działania podejmowane przez użytkownika. W całym rozdziale będziemy pracować z językiem o nazwie XAML (ang. *Extensible Application Markup Language*, wymawiaj„zamel"), czyli rozszerzalnym językiem znaczników dla aplikacji, który można stosować do opisywania projektu interfejsu użytkownika. Zobaczysz, jak możesz używać języka XAML ze środowiskiem Visual Studio w celu tworzenia pierwszorzędnego doświadczenia użytkownika..

Tworzenie maszyny dodającej

Po sukcesie gadającego samouczka tabliczki mnożenia, który utworzyliśmy w rozdziale 6., zostałeś poproszony przez znajomych o opracowanie maszyny dodającej, której będzie można używać do ćwiczenia dodawania lub w jakiś inny sposób. Twoi znajomi powiedzieli Ci, że potrzebują czegoś podobnego do programu pokazanego na rysunku 16.1. W tym programie użytkownicy wpisują w dwóch polach składniki dodawania, a następnie naciskają przycisk *równa się*, aby zobacz wynik.

Rysunek 16.1. Maszyna dodająca

Jest to prosta aplikacja dla systemu Windows, ale tak naprawdę wykracza poza to, co możemy zrobić za pomocą biblioteki Snaps. Programy, które do tej pory pisaliśmy, wykorzystywały zachowania z tej biblioteki w celu zapewnienia interakcji z użytkownikami. Metody Snaps działają bez zarzutu, ale nie są zbyt elastyczne. Aby utworzyć maszynę do nauki dodawania, musimy dowiedzieć się, jak tworzyć graficzne interfejsy użytkownika (ang. *graphical user interfaces* — GUI).

Graficzny interfejs użytkownika wykorzystujący XAML

Interfejs użytkownika to elegancka nazwa tego, co widzą użytkownicy podczas korzystania z programu. Składa się z przycisków, pól tekstowych, etykiet i obrazów, z którymi pracują użytkownicy, aby wykonać swoje zadania. Do obowiązków programisty należy między innymi tworzenie tego frontendu i zapewnianie mu odpowiednich zachowań, aby użytkownicy mogli sterować programem i uzyskiwać od niego to, czego potrzebują.

W tym rozdziale nie chodzi o programowanie w języku C#, ale o tworzenie interfejsów użytkownika za pomocą języka XAML, który został zaprojektowany przez firmę Microsoft w celu

ułatwienia programistom budowania dobrze prezentujących się aplikacji. Języka XAML używamy do opisywania tego, co jest wyświetlane, a za pomocą C# zapewniamy wymagane zachowania. (Programy korzystające z interfejsu użytkownika opartego na języku XAML możesz tworzyć także za pomocą innych języków programowania, takich jak Visual Basic). Aby zrozumieć ten proces, musisz nauczyć się kilku rzeczy na ten temat sposobu działania języków znaczników, ale ta wiedza jest niezwykle przydatna, ponieważ mnóstwo nowoczesnych systemów interfejsu użytkownika działa w podobny sposób.

Rozszerzalny język znaczników dla aplikacji umożliwia wykorzystanie reguł języka do tworzenia konstrukcji, które zasadniczo mogą opisać wszystko. Angielski jest pod tym względem bardzo podobny. Istnieją litery i znaki interpunkcyjne, które są symbolami języka angielskiego. Mamy również reguły (gramatykę), które określają sposób używania słów i tworzenia zdań, oraz różne rodzaje słów — rzeczowniki opisujące rzeczy i czasowniki opisujące działania. W przypadku pojawienia się czegoś nowego, wymyślamy nowy zestaw słów, żeby to opisać. Kiedy wynaleziono *komputer*, ktoś musiał wymyślić to słowo, wraz z wyrażeniami, takimi jak *bootowanie*, *awaria systemu*, *działa za wolno*, *skasowało całą moją pracę* itp.

Języki oparte na języku XML są rozszerzalne pod tym względem, że możemy wymyślać nowe słowa i frazy zgodne z regułami danego języka i wykorzystywać te nowe konstrukcje do opisywania dowolnych rzeczy. Są to tak zwane **języki znaczników** (ang. *markup languages*), ponieważ mogą być wykorzystywane do opisywania rozmieszczenia elementów na stronie. Angielskie słowo *markup* oznaczało pierwotnie instrukcje używane w drukowaniu, gdy chciano powiedzieć na przykład: „Wydrukuj nazwisko Rob Miles bardzo dużą czcionką". Najbardziej znanym językiem znaczników jest prawdopodobnie HTML (ang. *Hypertext Markup Language*), który jest używany w sieci WWW do opisywania formatów stron internetowych. Programiści często wymyślają własne formaty przechowywania danych, wykorzystując do tego celu język XML. XAML przyjmuje reguły rozszerzalnego języka znaczników i używa ich do utworzenia języka opisującego komponenty, które mają być wyświetlane na stronie.

Spójrz na opis XAML pola tekstowego:

```
<TextBox Name="firstNumberTextBox" Width="100" Margin="4" TextAlignment="Center">
</TextBox>
```

W tym przykładzie widać, że projektanci języka XAML utworzyli właściwości, których nazwy określają ich zastosowanie. Element `TextBox` może otrzymać nazwę (`Name`), musi mieć określone szerokość (`Width`) oraz margines (`Margin`), pozwala także określić sposób wyrównania tekstu w polu (`TextAlignment`).

XAML i projekt strony

Gdy używasz programu Visual Studio do tworzenia zupełnie nowej uniwersalnej aplikacji systemu Windows, otrzymujesz stronę zawierającą tylko kilka elementów. W miarę dokładania

do strony kolejnych części, rozmiar plik powiększa się wraz z każdym dodawanym opisem. Niektóre elementy, takie jak pole tekstowe, mogą być samodzielne. Inne działają jak kontenery. Oznacza to, że mogą przechowywać inne elementy. Kontenery są bardzo przydatne, gdy chcesz opracować układ strony. Element `StackPanel` może na przykład przechowywać zestaw innych elementów ułożonych jeden na drugim. Plik XAML może również zawierać opisy animacji i efektów przejścia, które mogą być zastosowane do elementów na stronie, aby uzyskać naprawdę spektakularny interfejs użytkownika.

Nie zamierzam poświęcać zbyt wiele czasu na aspekty XAML dotyczące układu strony; wystarczy powiedzieć, że za pomocą tego języka możesz tworzyć imponujące frontendy dla programów. Okazuje się jednak, że wielu programistów (dotyczy to również mnie, ale nie musi dotyczyć Ciebie) niezbyt dobrze radzi sobie z projektowaniem atrakcyjnych interfejsów użytkownika. W prawdziwym życiu firmy często zatrudniają grafików, którzy tworzą artystycznie wyglądające frontendy. Rolą programisty jest przygotowanie kodu dla tych wyświetlaczy, aby wykonać wymagane zadanie.

XAML został opracowany z uwzględnieniem tej kwestii. Wymusza silną separację między projektem wyświetlacza ekranowego i sterowanym przez niego kodem. Ułatwia to programistom tworzenie początkowego interfejsu użytkownika, który jest następnie zmieniany przez projektanta, aby stał się bardziej atrakcyjny. Możliwe jest również, żeby programista wziął pełny projekt interfejsu użytkownika, a potem dopasował zachowania do każdego z komponentów wyświetlacza.

Opisywanie elementów XAML

Zobaczmy, w jaki sposób język XAML pozwala projektować aplikacje. W tym celu użyjemy tego języka do utworzenia naszej maszyny dodającej. Aby przypomnieć sobie, jak będzie wyglądał ten interfejs użytkownika, spójrz na rysunek 16.1. Ten wyświetlacz składa się z sześciu komponentów:

1. Tytuł *Maszyna dodająca*. Ten blok tekstu jest nieco większy niż reszta tekstu, aby się wyróżniał.
2. Górne pole tekstowe, w którym użytkownik wprowadza liczbę.
3. Element tekstowy zawierający znak +.
4. Dolne pole tekstowe, w którym użytkownik wprowadza drugą liczbę.
5. Przycisk, który użytkownik naciska w celu wykonania dodawania.
6. Pole tekstowe wyniku, które zmienia się, aby pokazać wynik po naciśnięciu przycisku. (W tej chwili to pole tekstowe jest puste, ponieważ nie wykonaliśmy jeszcze żadnego sumowania).

W terminologii XAML każdy pojedynczy element na ekranie nazywa się `UIElement` (czyli element interfejsu użytkownika). Każdy ma określoną pozycję na ekranie, określony rozmiar etykiety i wiele innych właściwości. Aktualizując XAML opisujący stronę, możesz na przykład zmienić kolor tekstu w polu tekstowym lub wyrównanie tekstu: do lewej, do prawej albo na środku. Kod XAML, którego użyłem do opisania wyświetlacza dla maszyny dodającej, wygląda następująco:

```
<StackPanel>
  <TextBlock Text="Maszyna dodająca" TextAlignment="Center" Margin="8"
      FontSize="16"></TextBlock>
  <TextBox Name="firstNumberTextBox" Width="100" Margin="8" TextAlignment="Center">
      </TextBox>
  <TextBlock Text="+" TextAlignment="Center" Margin="8"></TextBlock>
  <TextBox Name="secondNumberTextBox" Width="100" Margin="8"
      TextAlignment="Center"></TextBox>
  <Button Content="równa się" Name="equalsButton" HorizontalAlignment="Center"
      Margin="8"></Button>
  <TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="8">
      </TextBlock>
</StackPanel>
```

ANALIZA KODU

Badanie kodu XAML

Jeśli przyjrzysz się uważnie, możesz skojarzyć każdy z elementów pokazanych na rysunku 16.1 i wymienionych na powyższej liście z elementami w pliku XAML. Jest jednak kilka rzeczy, którym powinniśmy przyjrzeć się dokładniej.

Pytanie: Do czego służy element `StackPanel`?

> **Odpowiedź:** `StackPanel` jest bardzo prosty i bardzo przydatny. Pozwala nam ułożyć serię elementów wyświetlacza w stos — oznacza to, że nie musisz definiować pozycji na ekranie dla każdego z elementów indywidualnie. Domyślnym ustawieniem jest układanie w stos kolejnych elementów idąc w dół ekranu, ale można także układać w stos elementy poruszając się w poprzek ekranu. Możesz umieszczać jeden element `StackPanel` wewnątrz drugiego elementu `StackPanel`, aby utworzyć stos rzędów. Zagnieżdżanie elementów jest powtarzającym się motywem w dokumentach XAML.

Pytanie: Jaka jest różnica między `TextBox` a `TextBlock`?

> **Odpowiedź:** `TextBox` to element wyświetlacza, w którym użytkownik może wpisywać tekst. Dwie dodawane liczby będą wprowadzane w polach tekstowych o nazwach `firstNumberTextBox` i `secondNumberTextBox`. Natomiast `TextBlock` to po prostu blok tekstu, który jest wyświetlany. Element `TextBlock` wykorzystujemy do przekazywania użytkownikowi różnych informacji. W tym przypadku używamy elementów `TextBlock`, aby wyświetlić tytuł aplikacji, znak + dla dodawania, a także wynik obliczeń. Użytkownik nie może wchodzić w interakcje z zawartością `TextBlock`.

Pytanie: Aby sprawdzić, czy rozumiesz, jak to działa, spróbuj wskazać, jaki tekst jest aktualnie wyświetlany w `resultTextBlock`.

> **Odpowiedź:** Możesz to ustalić, przeglądając kod XAML w celu odnalezienia elementu `TextBlock` o nazwie `resultTextBlock` i jego właściwości `Text`. Okazuje się, że po uruchomieniu programu właściwość `Text` jest ustawiana na "", czyli na pusty łańcuch znaków.

Z PUNKTU WIDZENIA PROGRAMISTY

Korzystaj z automatycznego układu w jak największym stopniu

Zawsze się niepokoję, kiedy zaczynam ustawiać elementy na ekranie w pozycjach bezwzględnych. Gdy tak robisz, przyjmujesz założenia dotyczące rozmiaru używanego przez Ciebie ekranu i wymiarów elementu. Nowoczesne komputery są dostarczane z bardzo różnymi rozmiarami ekranów, a ponadto użytkownicy mogą zmieniać rozmiar tekstu na ekranie, aby powiększyć widok. Mogą również w trakcie korzystania z Twojego programu zmieniać orientację ekranu z poziomej na pionową. Jeśli ustawisz na sztywno pozycję elementów na ekranie, będzie to oznaczało, że może się to sprawdzać na jednym konkretnym urządzeniu, ale będzie wyglądać bardzo niezręcznie na innym. Dlatego powinieneś używać funkcji automatycznego układu, takich jak `StackPanel`, aby dynamicznie pozycjonowały elementy za Ciebie. Znacznie zmniejsza to ryzyko, że Twój program będzie miał problemy z wyświetlaniem.

Elementy XAML i obiekty oprogramowania

Z punktu widzenia programowania każdy element XAML na ekranie jest w rzeczywistości obiektem w programie. Widziałeś już, że obiekty to świetny sposób na reprezentowanie różnych rzeczy, z którymi chcemy pracować. Okazuje się, że obiekty świetnie nadają się również do reprezentowania innych rzeczy — na przykład elementów na wyświetlaczu. Jeśli się nad tym zastanowić, można dojść do wniosku, że pole wyświetlające tekst na ekranie będzie miało takie właściwości, jak jego pozycja na ekranie, kolor tekstu, sam tekst i tak dalej.

Podczas kompilowania programu korzystającego z interfejsu użytkownika XAML, system „kompiluje" również opis XAML w celu utworzenia zestawu obiektów C#, z których każdy reprezentuje element interfejsu użytkownika. W maszynie dodającej istnieją trzy różne typy elementów:

- `TextBox`. Umożliwia użytkownikowi wprowadzanie tekstu do programu.
- `TextBlock`. Blok tekstu, który po prostu przekazuje informacje.
- `Button`. Coś, co możemy nacisnąć, aby wywołać jakieś zdarzenia w naszym programie.

Nasz program będzie jednak manipulował tymi elementami tak, jakby były obiektami C#, chociaż w rzeczywistości są one zdefiniowane w pliku źródłowym XAML. Może to działać, ponieważ kiedy program jest budowany, system XAML tworzy obiekty odpowiadające elementom opisanym w pliku źródłowym XAML.

Zarządzanie elementami za pomocą ich nazw

Kiedy chcesz używać w programie elementów interfejsu użytkownika, potrzebujesz sposobu odwoływania się do nich. Jeśli spojrzysz na XAML, który opisuje maszynę dodającą, zobaczysz, że niektóre komponenty mają właściwość nazwy (`Name`):

```
<TextBox Name="firstNumberTextBox" Width="100" Margin="4" TextAlignment="Center">
    </TextBox>
```

Nadałem temu polu tekstowemu nazwę `firstNumberTextBox`. (Nigdy nie zgadniesz, jak nazywa się drugie pole tekstowe). Zwróć uwagę, że w tym kontekście nazwa elementu zdefiniuje tak naprawdę nazwę zmiennej `TextBox`, zadeklarowanej wewnątrz programu maszyny dodającej. Innymi słowy, gdzieś w programie znajdzie się teraz następująca instrukcja:

```
TextBox firstNumberTextBox;
```

Te deklaracje są tworzone przez Visual Studio podczas budowania programu, więc nie musisz się martwić, gdzie znajduje się ta instrukcja. Musisz tylko o tym pamiętać, że tak właśnie działa program.

ANALIZA KODU

Nazwy elementów XAML

Pytanie: Dlaczego nie wszystkie elementy wyświetlacza mają nazwę?

Odpowiedź: Nazwy musimy nadać jedynie tym elementom, z którymi nasz program musi wchodzić w interakcje. Nie ma sensu nadawanie nazwy elementowi `TextBlock`, który zawiera znak +, ponieważ podczas działania programu użytkownicy nigdy nie będą musieli wchodzić z nim w interakcje — przechowuje on po prostu coś, co będzie wyświetlane użytkownikowi. Oczywiście, jeśli później zechcesz zmienić ten element (na przykład, aby umożliwić użytkownikowi wybór takiej wersji programu, która wykona odejmowanie), będziesz mógł nadać mu nazwę.

Właściwości w elementach

Niektóre właściwości elementu `TextBlock` są ustawiane w kodzie XAML, który go deklaruje. Inne muszą zostać zmienione przez program podczas jego działania. Wszystkie właściwości mogą być ustawiane w deklaracji i wszystkie mogą być później zmieniane przez program.

Oto XAML opisujący tę część ekranu, w której wyświetla się wynik dodawania:

```
<TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="4">
    </TextBlock>
```

Zwróć uwagę, że mamy właściwość `Text`, która jest obecnie ustawiona na pusty łańcuch znaków. Kiedy mówimy o „właściwościach" elementów XAML na stronie, tak naprawdę mówimy o wartościach właściwości w klasie, która implementuje `TextBlock`. Aby to wyjaśnić, załóżmy, że program zawiera taką instrukcję:

```
resultTextBlock.Text = "0";
```

Ta instrukcja spowoduje pojawienie się tekstu `0` w polu `resultTextBlock` na wyświetlaczu. Spowoduje również uruchomienie przypisania właściwości `Set` wewnątrz obiektu `result-TextBlock`, co ustawi odpowiednią wartość dla tekstu w `TextBlock`. Innymi słowy, właściwość `Text` obiektu `TextBlock` jest tą samą właściwością, niezależnie od tego, czy jej wartość zostanie ustawiona w kodzie XAML, czy w programie C#.

Projektowanie stron za pomocą języka XAML

XAML okazuje się bardzo przydatny. Gdy zdobędziesz już wiedzę niezbędną do opisywania komponentów, znacznie szybciej będziesz dodawał elementy do strony i przesuwał je, dokonując po prostu edycji tekstu w pliku XAML, zamiast przeciągać pola tekstowe i inne elementy po ekranie. Uważam, że jest to szczególnie przydatne, gdy potrzebujesz dużej liczby podobnych elementów na ekranie. Visual Studio zna składnię wykorzystywaną do opisywania każdego typu elementu i zapewnia w tym zakresie wsparcie IntelliSense.

Jeśli zapoznasz się ze specyfikacją XAML, dowiesz się z niej, że elementom możesz nadawać właściwości graficzne, które mogą czynić je przezroczystymi, dodawać obrazy do ich tła, a nawet animować je. W tym momencie wychodzisz poza programowanie i wkraczasz w dziedzinę projektowania graficznego — życzę Ci powodzenia.

Skoro wiesz już, że elementy na ekranie są w rzeczywistości graficzną realizacją obiektów oprogramowania, w następnej kolejności musisz dowiedzieć się, jak uzyskać kontrolę nad tymi obiektami i zmusić je do robienia użytecznych rzeczy w aplikacji. W tym celu musisz dodać trochę kodu C#, który wykona obliczenia potrzebne maszynie dodającej. Ale najpierw zbudujmy samą aplikację.

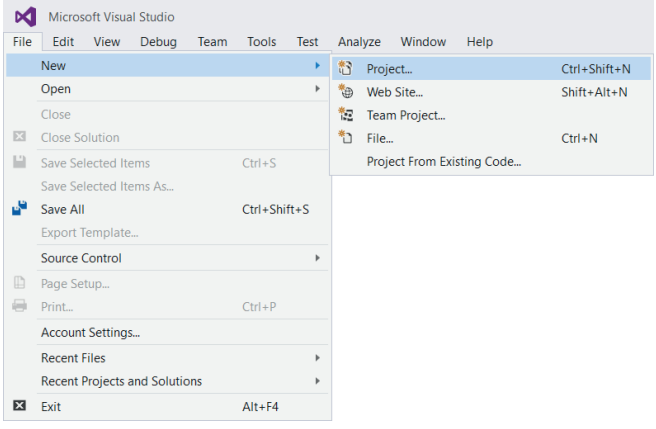
ZRÓB TO SAM

Zbudujmy naszą pierwszą uniwersalną aplikację Windows

W tym momencie tworzymy naszą pierwszą uniwersalną aplikację systemu Windows (ang. *Universal Windows Application*). Do tej pory budowaliśmy wszystko w środowisku frameworku Snaps. Już za chwilę utworzymy zupełnie nową, całkowicie pustą aplikację. Ten sam krok wykonuje każdy programista aplikacji, gdy decyduje się zbudować nowy program.

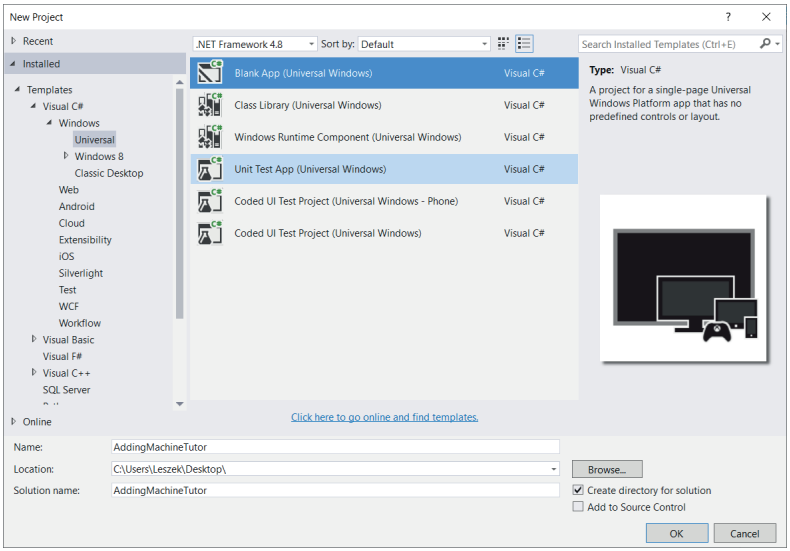
Tworzenie nowej aplikacji

Najpierw uruchom program Visual Studio 2015. Następnie w menu tego programu wybierz *File/New/Project*, jak pokazano na rysunku 16.2, aby otworzyć okno dialogowe *New Project*.



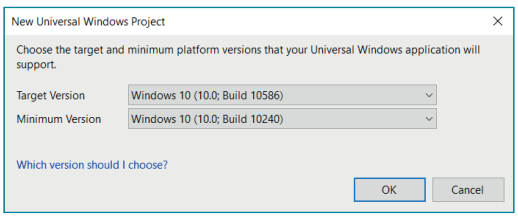
Rysunek 16.2. Menu tworzenia nowego projektu programu Visual Studio

Możesz utworzyć wiele różnych rodzajów projektów. My chcemy wybrać opcję *Blank App (Universal Windows)*. Po lewej stronie rysunku 16.3 możesz zobaczyć, jak nawigować do *Templates/VisualC#/Windows/Universal*, aby dotrzeć do tego zestawu szablonów projektu.



Rysunek 16.3. Nazywanie naszego nowego projektu AddingMachineTutor

Podczas procesu tworzenia rozwiązania może zostać wyświetlone zapytanie, z którą wersją systemu Windows ma współpracować aplikacja, jak pokazano na rysunku 16.4. Kliknij po prostu *OK*, aby użyć wersji domyślnych.



Rysunek 16.4. Gdy pojawi się pytanie dotyczące wybrania wersji systemu Windows, zachowaj ustawienia domyślne

CO MOGŁO PÓJŚĆ NIE TAK?

Nie widzisz szablonów Universal Windows Application

Jeżeli w oknie dialogowym *New Project* nie możesz znaleźć żadnych szablonów Uniwersal Windows, prawdopodobnie używasz niewłaściwej wersji Visual Studio. Musisz używać programu Visual Studio 2015. Jeśli używasz tej wersji Visual Studio, ale nadal nie widzisz tego typu projektu, upewnij się, że masz zainstalowany pakiet narzędzi Universal Tools dla aplikacji Visual Studio 2015. Zazwyczaj te narzędzia są instalowane jako część ogólnej instalacji programu Visual Studio 2015, ale możesz mieć starszą wersję instalatora, która ich nie ma. Zajrzyj do instrukcji online (zobacz link w rozdziale 1.), które opisują, jak to naprawić.

Po znalezieniu wymaganego typu projektu wprowadź nazwę tego projektu (ja nazwałem swój *AddingMachineTutor*), a następnie kliknij przycisk *OK*. Domyślnie Visual Studio utworzy nowe rozwiązanie w podfolderze katalogu *Dokumenty*. Miejsce zapisywania rozwiązania możesz zmienić, klikając *Browse* i przed kliknięciem *OK* wybierając inną lokalizację na Twojej maszynie.

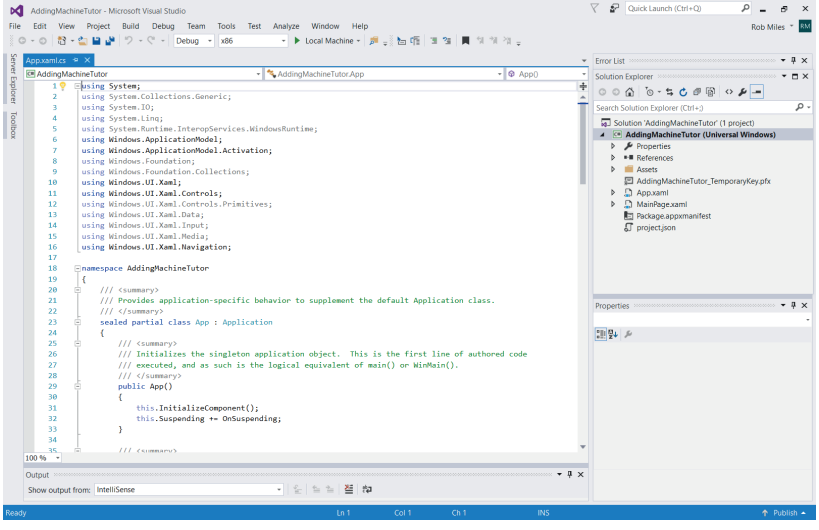
CO MOGŁO PÓJŚĆ NIE TAK?

Wybór niewłaściwego szablonu nigdy nie kończy się dobrze

Przyznanie się do tego jest nieco krępujące, ale w przeszłości byłem znany z wybierania złego szablonu na początku projektu. Zwykle robię to nadal podczas demonstracji dla co najmniej 200 studentów. Powoduje to moje zmieszanie i wywołuje rozbawienie wśród słuchaczy, więc radzę Ci dokładnie sprawdzić, czy wybrałeś właściwy szablon, chyba że chcesz wyglądać tak głupio jak ja.

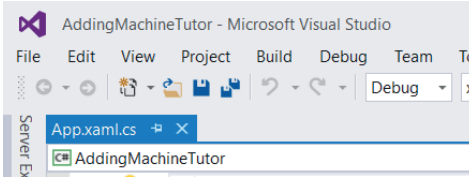
Tworzenie pustego programu

Po kliknięciu *OK* program Visual Studio 2015 zabiera się do pracy i tworzy dla Ciebie nowy, pusty projekt. Rysunek 16.5 pokazuje, co wyświetla Visual Studio po utworzeniu nowej aplikacji.



Rysunek 16.5. Pusta aplikacja uniwersalna Windows

Wygląda to bardzo dezorientująco. Visual Studio pokazuje zawartość należącego do aplikacji pliku o nazwie *App.xaml.cs*. Jest to ważny plik — stanowi tę część aplikacji, która przejmuje kontrolę, gdy program zostanie uruchomiony — ale na razie zostawimy go bez wprowadzania jakichkolwiek zmian. (Zawsze możemy wrócić do niego później, jeśli będziemy musieli dokonać modyfikacji). Aby zamknąć widok pliku *App.xaml.cs*, kliknij *X* obok nazwy pliku, jak pokazano na rysunku 16.6.



Rysunek 16.6. Zamykanie okna

Możemy uruchomić naszą pustą aplikację w taki sam sposób, jak uruchamialiśmy aplikacje biblioteki Snaps. Kliknij przycisk uruchamiania (zieloną strzałkę) w górnym rzędzie kontrolek (upewnij się, że obok przycisku napisane jest *Local Machine*). Po kliknięciu tego przycisku program jest kompilowany, ładowany do pamięci, a następnie może zostać uruchomiony. Rezultat pokazuje rysunek 16.7.

Rysunek 16.7. Uruchamianie pustego programu

Pusty program wygląda mniej więcej tak, jak można się tego spodziewać, chociaż zaintrygować mogą Cię dwie liczby wyświetlane w lewym górnym rogu okna aplikacji. Są to liczniki wydajności, które informują o wymaganiach stawianych przez aplikację komputerowi, który jest hostem, oraz częstotliwość aktualizacji wyświetlacza. W tej chwili nie są one szczególnie ważne, więc możesz je zignorować.

Biorąc pod uwagę to, że na razie udało nam się utworzyć jedynie pusty program, mamy niezbyt wiele funkcjonalności. Możemy przeciągać okno po ekranie, zmieniać jego rozmiar, maksymalizować i minimalizować je oraz zamknąć aplikację. Moglibyśmy nawet przesłać to rozwiązanie do sprzedaży w Sklepie Windows, ale jest mało prawdopodobne, że zostałoby ono zatwierdzone, ponieważ obecnie nie robi nic.

ANALIZA KODU

Zatrzymywanie aplikacji Windows

Pytanie: Jak zatrzymać aplikację Windows?

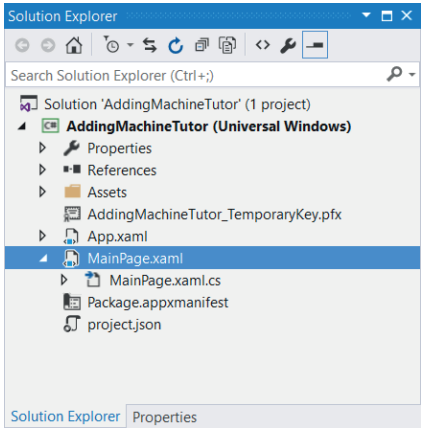
> **Odpowiedź:** W programach, które do tej pory pisaliśmy, na początku działania programu wywoływana była metoda `StartProgram` z biblioteki Snaps, a gdy ta metoda kończyła wykonywanie, kończył się program. Jednak aplikacja Windows 10 nie działa w ten sposób. Ten program będzie „działał", dopóki użytkownik go nie zamknie lub nie zostanie wyłączony komputer.

Oczywiście aplikacja systemu Windows 10 nie działa w ten sam sposób, co nasza wcześniejsze programy, ponieważ przez większość czasu pozostaje uśpiona, dopóki użytkownik czegoś nie zrobi. Nasz program maszyny dodającej przez większość czasu będzie oczekiwał na wpisanie przez użytkownika jakichś liczb lub naciśnięcie przycisku uruchamiającego obliczanie.

Jeśli chcesz kontrolować zamykanie programu przez użytkownika — na przykład w celu zapisania pewnych danych — możesz podłączyć jakąś metodę do zdarzenia uruchamianego, gdy użytkownik zamyka program.

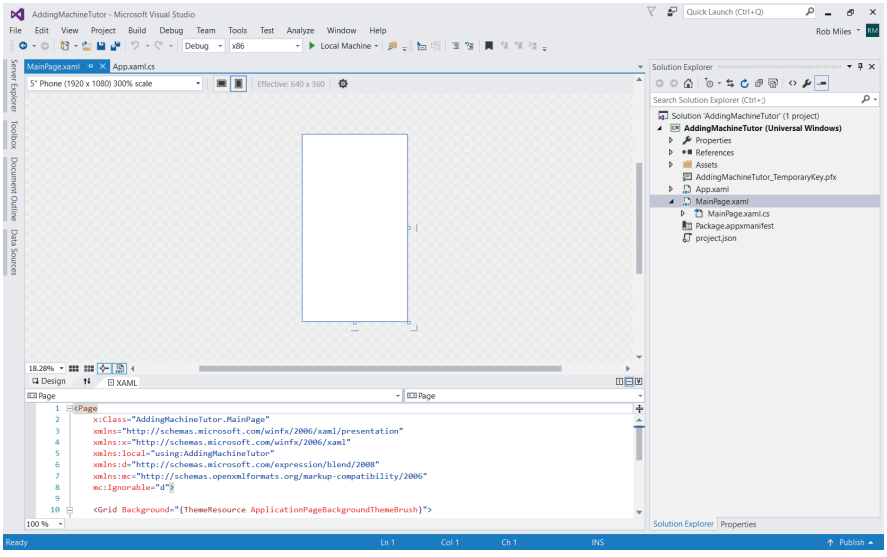
Tworzenie interfejsu użytkownika za pomocą XAML

Następnym krokiem jest dodanie do interfejsu użytkownika nowych elementów. W tym celu dodamy trochę kodu do zawartości pliku XAML, który opisuje stronę. Kod XAML opisujący stronę główną aplikacji znajduje się w pliku o nazwie *MainPage.xaml*. Po pierwsze, musisz zatrzymać aplikację, jeśli nadal działa. Następnie powinieneś otworzyć plik *MainPage.xaml*, klikając go dwukrotnie w oknie narzędzia *Solution Explorer*, jak pokazano na rysunku 16.8.



Rysunek 16.8. Otwieranie pliku MainPage.xaml

Otwarcie pliku powoduje wyświetlenie jego zawartości w widoku edycji, co pozwala zobaczyć kod XAML definiujący projekt oraz podgląd strony, jak widać na rysunku 16.9.



Rysunek 16.9. Edytowanie kodu XAML w Visual Studio

Widok edycji jest podzielony na dwa obszary. U góry znajduje się podgląd interfejsu, w takiej postaci, jak zobaczy go użytkownik. Na dole mamy kod XAML, który opisuje elementy na stronie. Zaczniemy od edycji pliku XAML.

Na samym dole pliku XAML znajduje się element `Grid` zawierający elementy, które pojawiają się na ekranie. Początkowy wiersz opisu `Grid` możesz dostrzec na samym dole rysunku 16.9. Możemy dodać jakiś element do ekranu, dodając kod XAML do elementu `Grid`:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel>
        <TextBox Name="firstNumberTextBox" Width="100" Margin="8"
            TextAlignment="Center"></TextBox>
    </StackPanel>
</Grid>
```

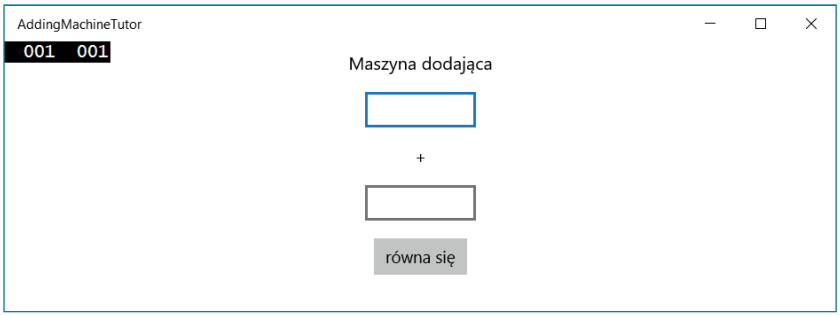
Jest to definicja pierwszego pola `TextBox` na wyświetlaczu. Jeśli ponownie uruchomimy aplikację, to pole `TextBox` zostanie wyświetlone w górnej części okna aplikacji, jak pokazano na rysunku 16.10.

Rysunek 16.10. Pojedyncze pole TextBox na ekranie

W polu `TextBox` możesz wpisywać tekst. Wpisany tekst jest wyśrodkowywany w polu `TextBox`, ponieważ właściwość `TextAlignment` dla `TextBox` została ustawiona na `Center`. Teraz możesz śmiało dodać inne elementy do wyświetlacza.

```
<StackPanel>
    <TextBlock Text="Maszyna dodająca" TextAlignment="Center" Margin="8"
        FontSize="16"></TextBlock>
    <TextBox Name="firstNumberTextBox" Width="100" Margin="8"
        TextAlignment="Center"></TextBox>
    <TextBlock Text="+" TextAlignment="Center" Margin="8"></TextBlock>
    <TextBox Name="secondNumberTextBox" Width="100" Margin="8"
        TextAlignment="Center"></TextBox>
    <Button Content="równa się" Name="equalsButton" HorizontalAlignment="Center"
        Margin="8"></Button>
    <TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="8">
        </TextBlock>
</StackPanel>
```

Te instrukcje XAML definiują elementy składające się na wyświetlacz maszyny dodającej. Powyższy XAML przypomina trochę program, ale w rzeczywistości jest czymś zupełnie innym. Jest to deklaracja wszystkich elementów wyświetlacza, których program będzie używał. Visual Studio odczyta zawartość tego pliku i podczas kompilacji programu utworzy elementy wyświetlacza. Jeżeli wstawisz ten kod do pliku *MainPage.xaml*, zobaczysz elementy pojawiające się w oknie podglądu nad tekstem. Gdy uruchomisz program, zobaczysz te elementy wyświetlone na ekranie zgodnie ze specyfikacją, jak widać na rysunku 16.11.



Rysunek 16.11. Elementy maszyny dodającej

Jeśli zmienisz rozmiar okna programu, przeciągając jego narożnik po ekranie, zauważysz, że elementy zawsze pozostają na środku okna. Windows automatycznie zaktualizuje układ, jeśli zmieni się rozmiar zawierającego go okna. W tego rodzaju projekcie przyjemne jest to, że będzie wyświetlany poprawnie na dowolnym urządzeniu, od 84-calowego ekranu Surface Hub po znacznie mniejszy Windows Phone.

ANALIZA KODU

Pozycjonowanie elementów na wyświetlaczu XAML

Radziłem Ci używać scenorysów do ustalania, jak powinien wyglądać program, a XAML zapewnia świetny sposób na wbudowanie scenorysowych projektów w aplikację. Gdy będziesz planował swoje projekty, musisz jednak wiedzieć, jak w XAML określa się wymiary.

Pytanie: Jak mierzy się elementy w projekcie XAML?

Odpowiedź: Proste pytanie, skomplikowana odpowiedź. W dawnych czasach łatwo było określać wymiary na wyświetlaczach. Wszystko było mierzone w pikselach (ang. *pixel*). (Angielskie słowo *pixel* to skrót od określenia *picture element*, oznaczającego element obrazu, który jest pojedynczą adresowalną kropką na wyświetlaczu). Wymiar 100 oznaczał 100 rzeczywistych kropek na ekranie. To podejście dobrze się sprawdzało, ponieważ ekrany miały niską rozdzielczość, a w użyciu nie było wielu różnych rozmiarów. Ponadto program był przeznaczony do działania tylko na jednej platformie.

Obecnie istnieje wiele różnych rozmiarów wyświetlaczy, od małych tabletów po ogromne ekrany naścienne. Co gorsza, same wyświetlacze różnią się znacznie pod względem liczby zawieranych pikseli. Niektóre urządzenia mają panele LCD z setkami pikseli na cal, podczas gdy inne obywają się ze znacznie niższą rozdzielczością. Mierzenie wszystkiego w pikselach przestało działać.

Wymiary używane w uniwersalnych aplikacjach Windows zostały utworzone, aby aplikacje były przenośne na różne ekrany. Gdy rozważaliśmy rozmiar duszków w naszych grach, zauważyliśmy, że wartości „pikselowe" używane w XAML są skalowane przez system Windows w celu odzwierciedlenia bazowego sprzętu, aby 96 z nich odpowiadało jednemu calowi na wyświetlaczu. Jest to jednak nieco bardziej skomplikowane. Wartości pikselowe, zwane **pikselami efektywnymi**, są w rzeczywistości skalowane w celu uwzględnienia odległości oglądania, rozmiaru i rozdzielczości wyświetlacza, aby „wyglądały odpowiednio". Oznacza to, że coś o szerokości określonej jako100 pikseli można narysować przy użyciu 150, 175 czy 200 fizycznych pikseli, w zależności od docelowego urządzenia.

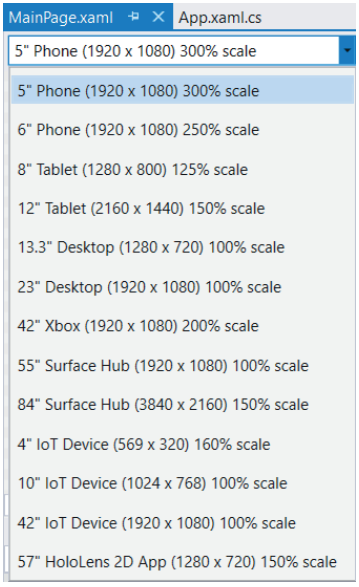
Oprócz pikseli efektywnych system Windows 10 zapewnia także funkcjonalność zwaną **adaptowalnym interfejsem użytkownika** (ang. *Adaptive User Interface*), która pozwala tworzyć alternatywne projekty dla różnych rozmiarów wyświetlaczy. Chodzi o to, że kiedy program jest uruchamiany, automatycznie wybiera projekt, który działa odpowiednio na używanym wyświetlaczu. Na dużym ekranie użytkownik zobaczy wiele paneli, a na mniejszym zostanie użyty projekt z jednym panelem, a użytkownik będzie nawigował między panelami.

Pytanie: Czy pole `firstNumberTextBox` znajduje się na środku ekranu, ponieważ ustawiłem w nim właściwość `TextAlignment="Center"`?

> **Odpowiedź:** Nie. Właściwość `TextAlignment` odnosi się do wyrównania tekstu wewnątrz pola tekstowego. Ustawienie tej właściwości na `Center` oznacza, że gdy użytkownik będzie wpisywał tekst w polu `TextBox`, kursor zostanie umieszczony na środku tego pola. Okazuje się, że jeśli nie określisz inaczej, elementy są naturalnie wyśrodkowywane przez XAML. Możesz jednak dodać do elementu właściwość, która wypozycjonuje go inaczej. Jeśli dodasz na przykład właściwość `HorizontalAlignment= "Left"` do elementu `firstNumberTextBox`, przesunie się on do lewej krawędzi ekranu. Istnieje wiele interesujących właściwości, które można ustawić dla elementów. Możesz się o nich dowiedzieć, korzystając z funkcjonalności IntelliSense edytora XAML, która będzie sugerować właściwości, a także wartości, które mogą przyjmować.

Przegląd rozmiarów ekranu XAML

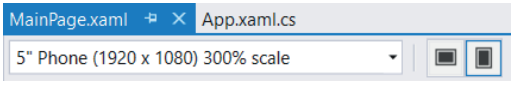
Jeśli spojrzysz na ekran podglądu w Visual Studio i na ekran otrzymywany po uruchomieniu aplikacji, zauważysz, że te strony mają zupełnie inne kształty i rozmiary. Pole `TextBox` również wygląda na większe na ekranie podglądu. Visual Studio zapewnia wiele środowisk podglądu o różnych rozmiarach. Możesz je zobaczyć, gdy rozwiniesz pole wyboru znajdujące się w lewym górnym rogu edytora XAML. Rysunek 16.12 pokazuje typy urządzeń, które są skonfigurowane w Visual Studio.



Rysunek 16.12. Opcje rozmiaru wyświetlacza w XAML

Bardzo chciałbym napisać program, który będzie wyświetlany na 84-calowym ekranie Surface Hub. Lub taki, który będzie działał na konsoli Xbox One. Jeśli masz na myśli konkretne urządzenie dla swojej aplikacji, możesz je wybrać, aby odzwierciedlić podgląd w edytorze.

Zwróć uwagę, że po prawej stronie tej listy znajduje się kilka przydatnych informacji na temat używanego okna podglądu. Jak widać na rysunku 16.13, możesz odczytać efektywną liczbę pikseli dostępnych w ustawieniach ekranu wybranych dla strony podglądu. Oznacza to, że gdybym utworzył pole `TextBox` o szerokości 360 pikseli, zajęłoby ono cały ekran.



Rysunek 16.13. Efektywne rozmiary w XAML

Z PUNKTU WIDZENIA PROGRAMISTY

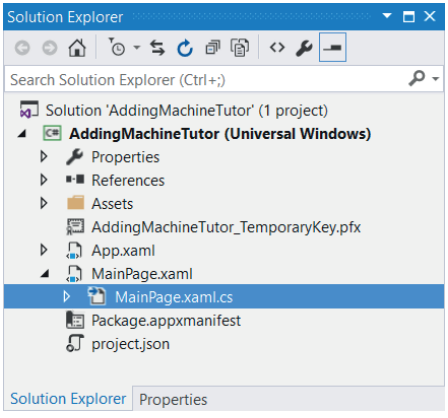
Projektuj interfejsy użytkownika pod kątem maksymalnej elastyczności

Ten materiał na temat rozmiarów i projektów wykracza poza faktyczny zakres naszej książki o programowaniu, ale myślę, że należy tutaj podkreślić jedną bardzo ważną kwestię — wymiary i pozycje ustalone na sztywno nie są tak naprawdę Twoimi przyjaciółmi. Możesz użyć funkcjonalności projektanta w Visual Studio do precyzyjnego rozmieszczania elementów na ekranie, ale myślę, że jest to bardzo zły pomysł w świecie z tyloma różnymi formatami urządzeń. Jeśli ustawisz elementy

na ekranie za pomocą zbyt wielu ustawionych na sztywno wartości, będziesz miał bardzo nieelastyczny wyświetlacz. W którymś momencie będziesz musiał zmierzyć się ze zirytowanym użytkownikiem, który będzie narzekał, że nie widzi przycisku *Wyślij*, gdy używa programu na swoim tablecie PC. Ja do układania prostych elementów używam kontenera `StackPanel` i Tobie sugeruję to samo.

Dodawanie zachowań programu

Języka XAML możesz używać do tworzenia bogatych i interesujących interfejsów użytkownika, ale żaden z nich tak naprawdę nie będzie robił niczego dla swoich użytkowników. Aby uzyskać tę funkcjonalność, musisz uruchomić jakiś kod programu. Ilekroć Visual Studio tworzy plik XAML, który opisuje stronę na wyświetlaczu, tworzy również plik programu C#, zwany plikiem ***code-behind***, i właśnie tutaj możemy umieścić kod, który sprawi, że nasza aplikacja będzie działać. Plik programu C# dla danej strony XAML znajduje się pod jej wpisem w oknie *Solution Explorer*. Możesz go otworzyć, klikając strzałkę po lewo od nazwy strony, a następnie dwukrotnie klikając nazwę pliku źródłowego, jak pokazano na rysunku 16.14.



Rysunek 16.14. Lokalizowanie pliku code-behind z kodem C#

Jeśli zajrzysz do pliku o nazwie *MainPage.xaml.cs*, przekonasz się, że raczej nie zawiera zbyt dużo kodu:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
```

```
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;
// Szablon elementu pustej strony został opisany na stronie http://go microsoft com/fwlink/?LinkId=402352&clcid=0x409

namespace AddingMachineTutor
{
    /// <summary>
    /// Pusta strona, która może być użyta samodzielnie lub można do niej nawigować za pomocą metod Frame
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Przestrzenie nazw, które zawierają obiekty XAML.

Przestrzeń nazw, która zawiera naszą aplikację.

Page jest klasą nadrzędną dla klasy MainPage.

Konstruktor dla MainPage.

Wywołanie tej metody powoduje skonfigurowanie strony.

Większa część tego pliku składa się z instrukcji `using`, które pozwalają programowi na bezpośrednie korzystanie z klas XAML bez konieczności podawania pełnej nazwy każdej z nich. Zamiast pisać na przykład `Windows.UI.Xaml.Controls.Button`, możemy napisać po prostu `Button`, ponieważ plik zawiera instrukcję `using Windows.UI.Xaml.Controls`.

ANALIZA KODU

Rzut oka na klasę MainPage

Chociaż klasa `MainPage` wygląda bardzo podobnie do innych klas, które widziałeś, ma pewne nowe elementy.

Pytanie: Co oznacza przestrzeń nazw?

Odpowiedź: Przestrzenie nazw pozwalają nam nadawać unikatowe nazwy elementom w aplikacjach. Widziałeś, że słowo kluczowe `using` języka C# pozwala kompilatorowi przeszukać przestrzeń nazw pod kątem elementów, których używasz w swoich programach. Instrukcja `namespace` w tym programie pokazuje, jak tworzona jest przestrzeń nazw. Efektem wykonania tej instrukcji jest to, że pełną nazwą klasy `MainPage` staje się `AddingMachineTutor.MainPage`. Visual Studio używa nazwy tworzonej aplikacji jako przestrzenni nazw zawierającej wszystkie klasy, które są częścią tej aplikacji.

Pytanie: Dlaczego definicja klasy `MainPage` jest tak skomplikowana?

Odpowiedź: Kiedy tworzyliśmy klasę, było to bardzo proste. Pisaliśmy po prostu słowo kluczowe `class`, po którym następowała nazwa, jaką wybraliśmy dla danej klasy. Zrobiliśmy tak, rozpoczynając deklarację klasy `Contact`:

```
class Contact
```

Jeśli spojrzysz na deklarację klasy `MainPage`, zobaczysz wiele dodatkowego kodu:

```
public sealed partial class MainPage : Page
```

Pierwszym godnym uwagi elementem jest `: Page` na końcu deklaracji. Jest to informacja dla kompilatora C#, że klasa `MainPage` jest rozszerzeniem klasy `Page`. Widziałeś to już wcześniej, kiedy tworzyliśmy zestaw klas duszków dla gry *Space Rockets in Space*. Różne rodzaje duszków tworzyliśmy poprzez rozszerzenie typu nadrzędnego. Tutaj tworzymy nowy rodzaj strony XAML poprzez rozszerzenie nadrzędnej klasy `Page`. Klasa `Page` jest dostarczana jako część zestawu zasobów służących do tworzenia uniwersalnych aplikacji systemu Windows.

Pytanie: Co oznacza `sealed`?

Odpowiedź: Widziałeś, że można tworzyć nowe klasy, rozszerzając klasy nadrzędne. Jednak klasa oznaczona za pomocą modyfikatora `sealed` nie może być w ten sposób rozszerzana. Nie ma powodu, by ktokolwiek rozszerzał klasę `MainPage`, dlatego jest ona oznaczona jako `sealed`, żeby coś takiego nie mogło się zdarzyć.

Pytanie: Co oznacza `partial`?

Odpowiedź: Poznałeś już słowo kluczowe `partial`, gdy badałeś, jak tworzone są zachowania biblioteki Snaps. Słowo kluczowe `partial` informuje kompilator, że w aplikacji mogą istnieć inne części tej klasy, przechowywane w osobnych plikach źródłowych. Innymi słowy, ten plik przechowuje część klasy `MainPage`. Klasy częściowe ułatwiają nawigację po dużych klasach, które mogą być rozłożone na kilka krótszych plików, a nie przechowywane w jednym dużym pliku. Podczas budowania programu, zawartość pliku *code-behind* z kodem C# jest łączona z C# wygenerowanym na podstawie opisu strony XAML, aby utworzyć obiekt C# reprezentujący tę stronę na ekranie.

Jedyną metodą w tym programie jest konstruktor klasy `MainPage`. Jak wiesz, konstruktor klasy jest wywoływany, gdy tworzona jest instancja klasy. Jedynym zadaniem tego konstruktora jest wywoływanie metody `InitializeComponent`. Jeśli zajrzysz do metody `InitializeComponent`, znajdziesz kod, który tworzy instancje elementów wyświetlacza. Ten kod jest tworzony automatycznie przez Visual Studio na podstawie kodu XAML opisującego stronę. Ważne jest, abyś pozostawił to wywołanie w obecnej postaci i nie zmieniał zawartości samej metody, ponieważ prawdopodobnie spowodowałoby to uszkodzenie Twojego programu. Zagłębiliśmy się bardzo w wewnętrzne mechanizmy działania XAML. Piszę Ci o tym wszystkim, abyś mógł zrozumieć, że tak naprawdę nie ma tutaj żadnej magii. Z naszego punktu widzenia przyjemne jest to, że nie

musimy się przejmować, w jaki sposób te obiekty są tworzone i wyświetlane; aby zaprojektować i zbudować nasz wyświetlacz, możemy po prostu użyć wysokopoziomowych narzędzi lub napisać instrukcje w XAML.

Obliczanie wyniku

W tej chwili nasz program wygląda dobrze, ale tak naprawdę niczego nie robi. Potrzebujemy napisać trochę kodu, który wykona wymagane obliczenia i wyświetli wynik. Może to być coś takiego:

```
private void displayResult()
{
    float v1 = float.Parse(firstNumberTextBox.Text);
    float v2 = float.Parse(secondNumberTextBox.Text);

    float.result = v1 + v2;

    resultTextBlock.Text = result.ToString();
}
```

Metoda wywoływana w celu wyświetlenia wyniku.

Pobiera wartości, które dodajemy.

Pobiera wynik, a następnie go wyświetla.

Elementy `TextBox` wyświetlacza udostępniają właściwość o nazwie `Text`. Można ją odczytywać lub można w niej zapisywać. Ustawienie wartości dla właściwości `Text` spowoduje zmianę tekstu w polu `TextBox` na ekranie. Odczytywanie właściwości `Text` umożliwia naszemu programowi odczytanie tego, co zostało wpisane w polu `TextBox`.

Tekst jest podawany jako łańcuch znaków, który należy przekonwertować na wartość liczbową, jeśli nasz program ma wykonać jakiekolwiek obliczenia dla tej wartości. Widziałeś już wcześniej metodę `Parse`. Pobiera ona łańcuch znaków i zwraca liczbę opisaną przez ten łańcuch. Każdy z typów liczbowych (`int`, `float`, `double` itp.) ma zachowanie `Parse`, które pobiera łańcuch znaków i zwraca opisaną przez niego wartość liczbową. Tworzona przez nas maszyna dodająca może działać z liczbami zmiennoprzecinkowymi, więc ta metoda parsuje tekst z każdego wejściowego pola tekstowego, a następnie oblicza wynik, dodając wartości.

Na koniec ta metoda przyjmuje obliczoną liczbę, konwertuje ją na łańcuch znaków, a na koniec ustawia ten łańcuch jako tekst pola `resultTextBlock`. Metoda `ToString` jest odwrotnością metody `Parse` („antyparsowaniem", jeśli takie określenie bardziej Ci się podoba). Zapewnia tekst opisujący zawartość obiektu. W przypadku typu zmiennoprzecinkowego jest to tekst opisujący daną wartość.

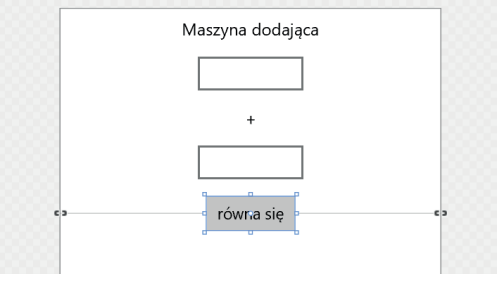
Mamy teraz kod, który ustala odpowiedź. Musimy tylko znaleźć sposób na uruchomienie tego kodu, gdy użytkownik naciśnie przycisk *równa się*.

Zdarzenia i programy

W rozdziale 15. dowiedziałeś się, że programy składają się z komponentów, które wysyłają do siebie nawzajem komunikaty. Gdy kosmita uderzał w statek kosmiczny, wysyłał do niego komunikat, który mówił: „Musisz teraz przyjąć obrażenia". Robił to, wywołując metodę `TakeDamage` dla statku kosmicznego. Zdarzenie jest formą komunikatu. Jeżeli omówisz tę kwestię z komputerowcami, każdy z nich będzie miał swoje zdanie na temat różnicy między zdarzeniem a komunikatem. Ja uważam, że zdarzenie jest rodzajem komunikatu, który mogą subskrybować komponenty programu.

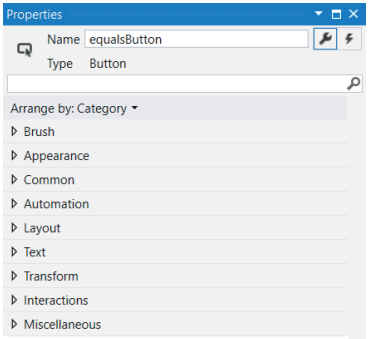
Jeśli jest to dla Ciebie nieco dezorientujące, pomyśl o problemie, który próbujemy rozwiązać. Chcemy, żeby program wyświetlał wynik dodawania, gdy użyty zostanie przycisk wyświetlony na ekranie. Przycisk jest rodzajem elementu wyświetlacza, który może generować zdarzenia (w tym przypadku: „Zostałem kliknięty"), a nasz program chciałby otrzymywać te zdarzenia. Metody, które są uruchamiane w reakcji na takie zdarzenia, nazywane są **procedurami obsługi zdarzeń**. Z punktu widzenia języka C# w tych metodach nie ma nic wyjątkowego. Istotną różnicą między metodą obsługi zdarzeń i „zwykłą" metodą jest to, że procedura obsługi zdarzeń jest wywoływana tylko wtedy, kiedy nastąpi jakieś zdarzenie.

Okazuje się, że podłączenie przycisku do metody obsługi zdarzeń jest w rzeczywistości całkiem łatwe, a większość pracy wykona za nas Visual Studio. Zacznijmy od powrotu do edytora graficznego XAML w Visual Studio i wybrania elementu przycisku (przez kliknięcie go myszą), jak pokazano na rysunku 16.15. Po wybraniu przycisku możesz poruszać nim po ekranie lub zmienić jego rozmiar, ale my chcemy edytować właściwości przycisku i dodać procedurę obsługi zdarzeń dla zdarzenia `Click` — innymi słowy, określić metodę, która będzie uruchamiana przy każdym kliknięciu przycisku.



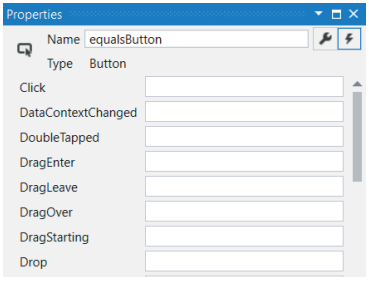
Rysunek 16.15. Wybranie przycisku w edytorze XAML

Visual Studio udostępnia panel *Properties* (właściwości), w którym można zmieniać wygląd i zachowanie każdego obiektu na ekranie. Ten panel jest zwykle wyświetlany w prawym dolnym rogu okna programu Visual Studio. Wybranie przycisku w edytorze powoduje wybranie tego elementu również w panelu *Properties*. (Możesz przekonać się, że panel *Properties* wyświetla poprawne właściwości, ponieważ wartość pola *Name* u góry panelu to *equalsButton*, a *Type* to *Button*). Rysunek 16.16 pokazuje właściwości elementu *equalsButton*.



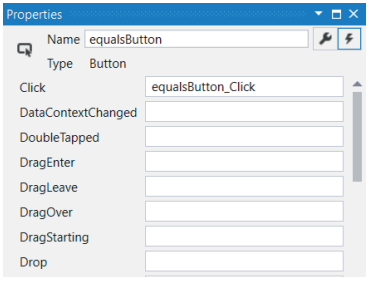
Rysunek 16.16. Właściwości przycisku

Te właściwości są podzielone na kategorie. Możesz ich użyć do zmiany wyglądu przycisku, ale my chcemy zarządzać zdarzeniami, które może generować, kliknij więc ikonę błyskawicy w prawym górnym rogu panelu *Properties*, aby przejść do panelu zdarzeń, pokazanego na rysunku 16.17. Wyświetla on zdarzenia, które może generować ten przycisk.



Rysunek 16.17. Lista zdarzeń przycisku

Jeśli chcesz, możesz podłączyć procedurę obsługi zdarzeń do każdego z tych zdarzeń, ale tak naprawdę nie mamy teraz na to czasu. Zdarzenie, do którego chcemy się podłączyć, znajduje się na samym początku listy — jest to zdarzenie *Click*. Procedurę obsługi zdarzeń możemy podłączyć do tego zdarzenia klikając dwukrotnie puste pole tekstowe znajdujące się obok nazwy zdarzenia *Click*. Po dwukrotnym kliknięciu zobaczysz nazwę metody, jak pokazano na rysunku 16.18.



Rysunek 16.18. Dodano procedurę obsługi zdarzenia Click do przycisku „równa się”

Dzieją się tutaj jeszcze dwie inne rzeczy. Po pierwsze, Visual Studio dodaje opis procedury obsługi zdarzeń do kodu XAML dla przycisku *równa się*:

```
<Button Content="Equals" Name="equalsButton" HorizontalAlignment="Center" Margin="4"
Click="equalsButton_Click" ></Button>
```

Po drugie, Visual Studio dodaje pustą metodę obsługi zdarzeń do kodu działającego „za stroną" (ang. *code-behind*):

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

Podczas kompilowania programu zdarzenie opisane w XAML zostaje powiązane z metodą obsługi zdarzeń, dzięki czemu po naciśnięciu przycisku uruchamiana jest metoda obsługi zdarzeń. Aby kalkulator działał, metoda obsługi zdarzeń musi po prostu wywoływać metodę, która wykona obliczenia.

```
private void equalsButton_Click(object sender, RoutedEventArgs e)
{
    displayResult();
}

Demo 16-01 AddingMachineTutor
```

Gdy użytkownik kliknie przycisk *równa się*, metoda `equalsButton _ Click` zostanie uruchomiona i wywoła metodę `displayResult` w celu wyświetlenia wyniku.

ANALIZA KODU

Metody obsługi zdarzeń

Ta metoda obsługi zdarzeń jest dostarczana automatycznie, ale warto zastanowić się nad kilkoma kwestiami.

Pytanie: Jakie są parametry metody obsługi zdarzeń?

Odpowiedź: Ta metoda obsługi zdarzeń ma dwa parametry. Pierwszy, o nazwie `sender`, to referencja do obiektu na ekranie, który faktycznie wywołuje zdarzenie. W przypadku maszyny dodającej, ta referencja odwołuje się do obiektu `equalsButton`, klikniętego przez użytkownika.

Drugi parametr o nazwie `e` zawiera informacje o zdarzeniu, które wystąpiło. W przypadku kliknięcia przycisku nie można dodać zbyt wiele informacji, ale jeśli zdarzenia są generowane przez inne działania — na przykład ruch wskaźnika — parametr może dostarczyć szczegółowe informacje, takie jak pozycja wskaźnika w momencie wygenerowania zdarzenia.

Pytanie: Kiedy nasz program wywołuje metodę obsługi zdarzeń?

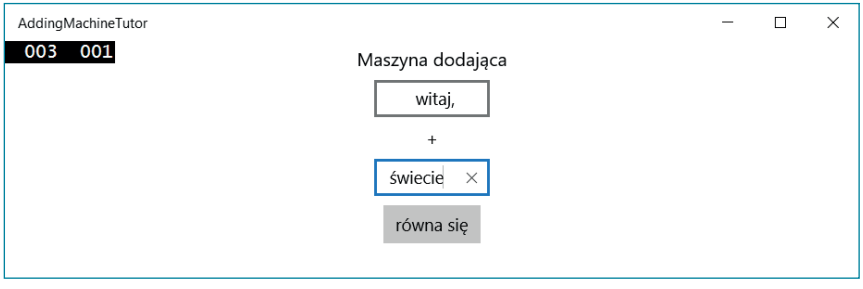
Odpowiedź: Metoda obsługi zdarzeń nigdy nie jest wywoływana przez nasz program. Jeśli użytkownik kliknie przycisk, ta metoda jest wywoływana, ale nasz program nigdy tak naprawdę sam jej nie wywołuje. Jest to zupełnie inny model niż ten, do którego jesteś przyzwyczajony. Przez większość czasu nasz program maszyny dodającej nie będzie nic robił; będzie tylko czekał, aż użytkownik wykona jakąś akcję.

Pytanie: Co się stanie, jeśli wykonywanie metody obsługi zdarzeń będzie zajmować dużo czasu?

Odpowiedź: Bardzo ważne jest, aby procedura obsługi zdarzeń zakończyła się tak szybko, jak to możliwe. Dopóki działa procedura obsługi zdarzeń, reszta interfejsu użytkownika nie jest w stanie czegokolwiek robić. Wszyscy doświadczyliśmy tego okropnego poczucia utraty kontroli, gdy aplikacja przestaje odpowiadać. Przyczyną tego braku reakcji jest zawsze jedna z metod obsługi zdarzeń w przyblokowanej aplikacji.

Obsługa błędów

Opracowane przez nas rozwiązanie działa i zapewnia w pełni sprawną maszynę dodającą. Nie jest to jednak program zbyt przyjazny dla użytkownika. Użytkownik może na przykład wpisać tekst w pola liczbowe, jak pokazano na rysunku 16.19.

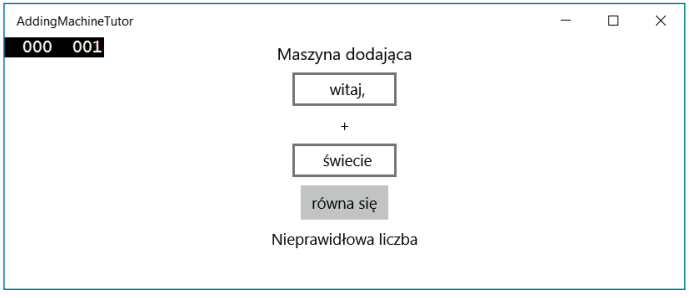


Rysunek 16.19. Wprowadzono niepoprawne liczby

Wiemy już, że spowoduje to problemy z metodą `Parse`, która natychmiast rzuci wyjątek i zatrzyma program. Wiemy również, że możemy przechwytywać wyjątki i obsługiwać je (widziałeś to w rozdziale 11.). W przypadku programu maszyny dodającej najlepszym sposobem radzenia sobie z tego rodzaju błędami jest wyświetlenie komunikatu:

```
private void displayResult()
{
    try
    {
        float v1 = float.Parse(firstNumberTextBox.Text);
        float v2 = float.Parse(secondNumberTextBox.Text);

        float result = v1 + v2;
        resultTextBlock.Text = result.ToString();
    }
    catch
    {
        resultTextBlock.Text = "Nieprawidłowa liczba";
    }
}

Demo 16-02 AddingMachineTutorErrors
```

Jeśli którekolwiek z wywołań metody `Parse` z bloku `try` rzuci wyjątek, program przekaże wykonywanie do bloku `catch`, który w polu `resultTextBlock` wyświetli komunikat *Nieprawidłowa liczba*, jak pokazano na rysunku 16.20.

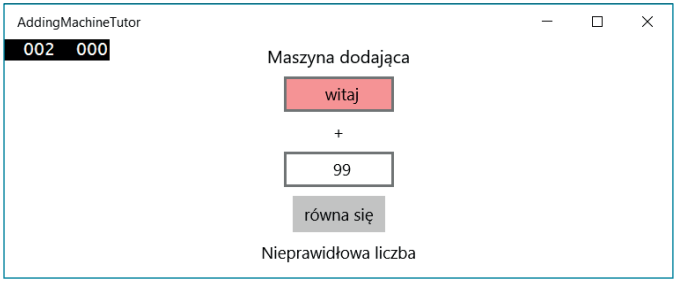


Rysunek 16.20. Wyświetlanie błędu „Nieprawidłowa liczba"

Użycie właściwości TextBox w celu ulepszenia interfejsu użytkownika

Obsługa błędów w naszym programie maszyny dodającej nie jest doskonała. Chociaż podczas działania program wyłapuje błędy, w rzeczywistości nie informuje użytkownika, która z wartości jest nieprawidłowa. Możemy ulepszyć ten interfejs użytkownika, korzystając z kilku dodatkowych właściwości elementu `TextBox`. Jest mnóstwo rzeczy, które moglibyśmy zrobić.

Zmienimy więc tło pola tekstowego z niepoprawnym wpisem, aby było podświetlane na czerwono. To pole tekstowe będzie wtedy wyglądać tak, jak pokazano na rysunku 16.21.



Rysunek 16.21. Podświetlanie niepoprawnych wpisów poprzez użycie właściwości

Jeśli wpis jest nieprawidłowy, zostanie podświetlony na czerwono. We wszechświecie XAML elementy wyświetlane na ekranie są rysowane za pomocą obiektu pędzla (ang. *brush*). Dostępnych jest wiele różnych klas pędzli; klasa `Brush` XAML jest tak naprawdę dobrym przykładem projektu opartego na klasach. `Brush` jest klasą nadrzędną wielu różnych rodzajów pędzli, które mogą rysować wzory lub obrazy. My użyjemy pędzla `SolidColorBrush`. Możemy go wykorzystać do utworzenia pędzla o określonym kolorze:

```
Brush errorBrush = new SolidColorBrush(Colors Red);
```

Tworzona jest tutaj referencja do klasy `Brush` o nazwie `errorBrush`, która jest ustawiana na jednolity czerwony pędzel. Możemy teraz ustawić, aby tło pola `TextBox` było rysowane za pomocą tego pędzla:

```
firstNumberTextBox.Background = errorBrush;
```

Ta instrukcja ustawia tło pola `firstNumberTextBox` na czerwone, aby wskazać, że wykryto błąd. Może to wyglądać na bardzo okrężny sposób ustawiania wartości koloru, ale w rzeczywistości zapewnia to dużą elastyczność. Mógłbyś na przykład utworzyć obraz reprezentujący błąd, a następnie użyć go jako tła pola `TextBox`.

Kolor tła pola tekstowego możemy ustawić, gdy metoda `Parse` rzuci wyjątek, ale istnieje również sposób przekonwertowania tekstu na liczbę, który nie korzysta z wyjątków, aby wskazać wystąpienie błędu. Metoda `TryParse` próbuje parsować łańcuch znaków i zwraca wartość `false`, jeśli próba się nie powiedzie. Poniższe instrukcje pokazują, jak jej używać:

```
float v1;
if (float.TryParse(firstNumberTextBox Text, out v1) == false)
```

Dodanie out oznacza, że metoda musi umieścić jakąś wartość w tym argumencie.

```
{
    firstNumberTextBox.Background = errorBrush;
}
```

Jeśli wykonanie TryParse nie powiedzie się, kolor tła dla pola TextBlock jest ustawiany na czerwony.

Ten kod ustawi kolor tła pola `TextBox` na czerwony, jeśli wprowadzony tekst nie będzie prawidłową liczbą.

```
Brush errorBrush = new SolidColorBrush(Colors Red);

private void displayResult()
{

    float v1;
    float v2;
    bool validValues = true;

    if (float.TryParse(firstNumberTextBox.Text, out v1)==false)
    {
        validValues = false;
        firstNumberTextBox.Background = errorBrush;
    }

    if (float.TryParse(secondNumberTextBox.Text, out v2)==false)
    {
        validValues = false;
        secondNumberTextBox.Background = errorBrush;
    }

    if (validValues)
    {
        float result = v1 + v2;
        resultTextBlock.Text = result.ToString();
    }
    else
    {
        resultTextBlock.Text = "Nieprawidłowa liczba";
    }
}

Demo 16-03 AddingMachineTutorFaultyErrorDisplay
```

Ta flaga wskazuje, czy wszystkie wartości są prawidłowe.

Próbuje parsować pierwszą wartość.

Jeśli parsowanie się nie powiedzie, ustawia flagę, aby wskazać, że dane wejściowe są nieprawidłowe.

Powtarzanie dla drugiej wartości.

ANALIZA KODU

Błędy w metodzie displayResult

Ta metoda `displayResult` wygląda na działającą, ale ma — niestety — poważny błąd.

Pytanie: Co jest nie tak z metodą `displayResult`?

Odpowiedź: Problem z `displayResult` nie jest widoczny przy pierwszym użyciu. Jeśli przetestujesz program tylko raz, przekonasz się, że jeśli wprowadzisz prawidłowe informacje, program wyświetli poprawny wynik. Jeżeli uruchomisz program ponownie i wprowadzisz nieprawidłowe wartości, odpowiednie pole `TextBox` zmieni kolor na czerwony, wskazując błąd. Jednak gdy wpiszesz jakieś prawidłowe wartości po wypróbowaniu nieprawidłowych, tło pozostanie czerwone. Nie jest to szczególnie zaskakujące, ponieważ w programie nie ma nic, co resetowałoby tła pól tekstowych po tym, gdy stały się czerwone, aby wskazać, że wartość jest nieprawidłowa.

Pytanie: Jak naprawić kolory tła?

Odpowiedź: Możemy naprawić ten błąd, przywracając kolor zwykły tła, jeśli wartość w polu `TextBox` będzie prawidłowa. Musimy tu jednak uważać, ponieważ nie możemy po prostu założyć, że wszyscy używają koloru białego jako tła tekstu. Niektórzy mogą używać różnych fantazyjnych schematów kolorów na swoich komputerach. Dobra wiadomość jest taka, że my możemy odczytywać i zapisywać oryginalny kolor tła pola `TextBox` i po prostu przywrócić oryginalny pędzel tła, gdy wartość zostanie uznana za prawidłową. Program może testować oryginalny kolor tła pola `TextBox` w konstruktorze strony, a następnie użyć tego do ustawienia koloru tła dla prawidłowych wpisów.

```
Brush errorBrush = new SolidColorBrush(Colors Red);

Brush correctBrush;                          Pędzel używany, aby wskazywać poprawność.
public MainPage()
{
    this.InitializeComponent();

    correctBrush = firstNumberTextBox.Background;   Kopiuje oryginalny kolor tła do
}                                                   poprawnego pędzla.

private void displayResult()
{
    float v1;
    float v2;
```

```
    bool validValues = true;

    if (float.TryParse(firstNumberTextBox.Text, out v1) == false)
    {
        validValues = false;
        firstNumberTextBox.Background = errorBrush;
    }
    else
        firstNumberTextBox.Background = correctBrush;

    if (float.TryParse(secondNumberTextBox.Text, out v2) == false)
    {
        validValues = false;
        secondNumberTextBox.Background = errorBrush;
    }
    else
        secondNumberTextBox.Background = correctBrush;

    if (validValues)
    {
        float result = v1 + v2;
        resultTextBlock.Text = result.ToString();
    }
    else
    {
        resultTextBlock.Text = "Nieprawidłowa liczb";
    }
}

Demo 16-04 AddingMachineTutorFixedErrorDisplay
```

Ten kod jest uruchamiany, jeśli wartość jest prawidłowa.

Ustawia kolor tła, aby wskazać, że wartość jest prawidłowa.

ZRÓB TO SAM

Utwórz kilka różnych maszyn liczących

Wzorca maszyny dodającej możesz użyć do utworzenia programów wykonujących odejmowanie, mnożenie i dzielenie. Możesz nawet utworzyć „maszynę wielozadaniową", która będzie miała różne obszary ekranu, przeznaczone do wykonywania poszczególnych działań arytmetycznych, albo opracować taką wersję programu, która będzie przyjmować dwie liczby i wyświetlać ich sumę, iloczyn i różnicę.

Czego się nauczyłeś?

W tym rozdziale porzuciłeś środowisko Snaps i utworzyłeś swoją pierwszą aplikację uniwersalną dla systemu Windows 10. Zobaczyłeś, jak używać języka XAML do opisywania rozmieszczenia i właściwości elementów na stronie aplikacji, oraz jak wykorzystać Visual Studio do edycji i przeglądania projektów XAML. Odkryłeś, że elementy opisane w pliku XAML ujawniają się jako obiekty w pliku programu C#, który kryje się za interfejsem użytkownika. Programy mogą aktualizować elementy wyświetlacza poprzez zapisywanie wartości we właściwościach elementów oraz odczytywać informacje z wyświetlacza poprzez odczytywanie elementów. Na początek omówiliśmy trzy elementy języka XAML: `TextBlock`, który wyświetla text, `TextBox`, w którym użytkownicy mogą wprowadzać surowy tekst, oraz element `Button`, który może być używany do wyzwalania zdarzeń.

Element `Button` zawiera zdarzenia, których można użyć do uruchamiania metod C# w reakcji na działania użytkownika. W przeciwieństwie do poprzednich programów, w których nasz kod był uruchamiany, gdy zaczynał działać program, aplikacja XAML ma kod C#, który jest powiązany ze zdarzeniami generowanymi przez elementy na wyświetlaczu. XAML opisujący przycisk może zawierać właściwość `Click`, identyfikującą metodę C#, która zostanie uruchomiona po kliknięciu przycisku. Zobaczyłeś, jak można tworzyć działające w ten sposób programy, i jak można generować dane wyjściowe w odpowiedzi na działania użytkownika.

Na koniec przyjrzeliśmy się prostej obsłudze błędów i zrobiliśmy pierwsze kroki w kierunku tworzenia odpowiednio przyjaznych dla użytkownika interfejsów.

Oto kilka pytań dotyczących interfejsów użytkownika opartych na kodzie XAML, nad którymi możesz się zastanowić.

Czym różnią się nasze programy od profesjonalnych?

Programy, które teraz piszemy, wykorzystują te same techniki i elementy wyświetlacza, co pełnowymiarowe, profesjonalne aplikacje. Jak wspomniałem na początku książki, jedyna różnica między pisanymi przez nas programami a programami profesjonalnymi polega na tym, że my nie sprzedajemy jeszcze naszych programów. Wykorzystanie języka XAML do zaprojektowania programu oraz sposób reagowania przez metody w kodzie na zdarzenia generowane przez elementy XAML wyświetlacza są dokładnie takie same w naszych programach, jak w tych większych.

Czy mogę tworzyć własne elementy i umieszczać je na wyświetlaczu?

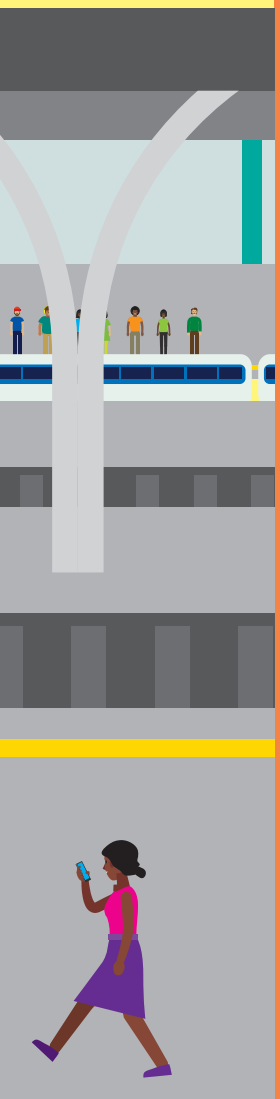
Tak. Chociaż omówienie tych kwestii wykracza poza zakres tej książki, faktem jest, że wszystkie elementy na ekranie zostały zbudowane na podstawie hierarchii klas, a to oznacza, że możesz je rozszerzać, aby dodawać własne zachowania. Możesz także tworzyć swoje własne niestandardowe kontrolki, zawierające wiele elementów sterowania, którymi można następnie manipulować za pomocą Visual Studio.

Jak mogę tworzyć naprawdę dobrze wyglądające interfejsy użytkownika?

XAML, który utworzyliśmy do tej pory, jest bardzo użyteczny. Wykonuje wyznaczone zadanie, ale nie jest zbyt ładny. Na szczęście istnieje narzędzie o nazwie Blend, które jest dostarczane jako część instalacji programu Visual Studio. Zapewnia ono przyjazny dla projektanta widok projektu interfejsu użytkownika. Narzędzia Blend można użyć do tworzenia efektów graficznych i animacji stosowanych do komponentów, a także do tworzenia i wykorzystywania szablonów wyświetlacza, które można zastosować do elementów w projektach interfejsu użytkownika. Powinieneś zapamiętać, że ostatecznie dane wyjściowe będą nadal miały postać pliku tekstowego, który zawiera opisy XAML elementów wyświetlacza. Język XAML zapewnia niesamowity poziom kontroli nad sposobem wyświetlania elementów, a narzędzie Blend doskonale nadaje się do pracy nad projektem. Możesz zbadać działanie narzędzia Blend (chociaż jest to bardzo złożony program), klikając prawym przyciskiem myszy dowolne pliki XAML w oknie *Solution Explorer* rozwiązań i wybierając opcję *Design in Blend*.

17.

Aplikacje i obiekty



Czego nauczysz się w tym rozdziale?

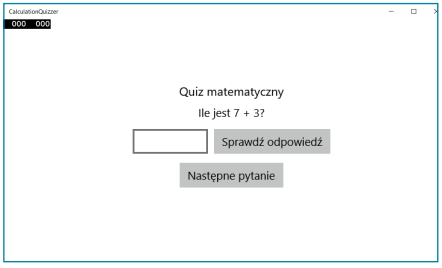
W ostatnim rozdziale utworzyliśmy bardzo prostą maszynę dodającą, która pokazała, jak aplikacje mogą komunikować się ze swoimi użytkownikami. Zobaczyłeś, jak korzystać z obiektów do reprezentowania elementów w interfejsie użytkownika, a także w jaki sposób program może zmieniać właściwości obiektów, aby aktualizować informacje wyświetlane użytkownikom.

W tym rozdziale zbadasz, jak można zmusić do interakcji interfejsy użytkownika i obiekty w dobrze skonstruowanej aplikacji. Trochę zabawy zapewni Ci również dodawanie zdjęć i dźwięków do aplikacji uniwersalnej. Ponadto dowiesz się, jak używać elementu ComboBox, aby umożliwić użytkownikowi dokonywanie wyborów.

Tworzenie aplikacji quizu matematycznego

Maszyna dodająca, którą utworzyliśmy w rozdziale 16., całkiem dobrze sprawdza się jako program do wykonywania obliczeń i przeglądania wyników, ale nie jest to zbyt dobre narzędzie do ćwiczenia dodawania. Do tego celu lepiej nadawałby się program, który wyświetla pytanie, prosi o odpowiedź, a następnie informuje, czy jest ona poprawna.

Jako punktu wyjścia moglibyśmy użyć projektu aplikacji pokazanego na rysunku 17.1. W górnej części okna wyświetlane jest pytanie, pod nim znajduje się pole tekstowe, w którym użytkownik wpisuje odpowiedź, a przycisk *Sprawdź odpowiedź* służy do weryfikacji podanego wyniku (gdy przycisk zostanie naciśnięty, program informuje, czy odpowiedź jest poprawna). Aby przejść do kolejnego pytania, użytkownik może nacisnąć przycisk *Następne pytanie*.



Rysunek 17.1. Aplikacja quizu matematycznego

Ten program wygląda podobnie do maszyny dodającej, nad którą pracowaliśmy w ostatnim rozdziale, więc wydaje się, że powinien być łatwy do napisania. Musimy tylko zmienić sposób, w jaki będzie reagował na wciskanie przycisków wyświetlonych na stronie.

Obiekty i wyświetlacze użytkownika

W aplikacji *Rejestr czasu pracy* używaliśmy obiektu `Contact` do przechowywania informacji o każdym kontakcie. Obiekt `Contact` przechowywał na przykład nazwę kontaktu. Klasa `Contact` nie zawiera jednak żadnego kodu, który wyświetlałby ten obiekt na ekranie. Programy są często skonstruowane tak, aby wyraźnie rozróżniać między obiektami, które zarządzają informacjami biznesowymi (takimi jak nazwa kontaktu), oraz obiektami przyjmującymi dane wejściowe i generującymi dane wyjściowe. Obiekty takie jak `Contact` nazywamy **obiektami biznesowymi**, ponieważ ich zadaniem jest przechowywanie informacji biznesowych, a nie interakcja z użytkownikiem.

W przypadku *Quizu matematycznego* zachowanie „quizowania" możemy potraktować jako formę obiektu biznesowego i oddzielić je od tej części programu, która steruje wyświetlaczem. Wyświetlacz prosi obiekt quizu: „Podaj mi pytanie". Następnie, gdy użytkownik wpisze odpowiedź, wyświetlacz pyta obiekt quizu: „Czy ta odpowiedź jest poprawna?". Z takiego sposobu

działania aplikacji quizowej wynikają dwie ogromne korzyści: aplikacja staje się łatwiejsza do testowania i jest znacznie bardziej elastyczna.

Testowalność

Jeśli włączymy zachowania quizu do strony wyświetlacza, nie będziemy mogli testować ich automatycznie. Lubię programy, które potrafią same się testować, a nie takie, w przypadku których muszę siedzieć przy komputerze, wpisywać różne rzeczy, naciskać przyciski i sprawdzać, co zostanie zwrócone. W przypadku naszego prostego programu quizowego nie ma wiele do przetestowania, ale gdybyśmy pracowali nad bardziej skomplikowaną aplikacją — na przykład dla banku — nie chcielibyśmy wykonywać ręcznie tysięcy transakcji, aby sprawdzić, czy program działa poprawnie. Chcielibyśmy sprawdzić saldo konta, wpłacić pieniądze, a następnie ponownie sprawdzić, czy saldo zostało poprawnie zaktualizowane, i to bez konieczności robienia tego przy ekranie komputera. Testowanie tej funkcjonalności byłoby znacznie łatwiejsze, gdyby transakcje na rachunku bankowym były przeprowadzane przez obiekt zewnętrzny w stosunku do strony wyświetlacza, ponieważ wtedy moglibyśmy napisać kod, który wykonywałby dużą liczbę transakcji i sprawdzał dla nas wyniki.

Analogicznie, jeśli wiemy, co powinien robić obiekt quizu matematycznego, możemy poprosić go o pytanie, samemu przeprowadzić ewaluację odpowiedzi, a następnie sprawdzić, czy obiekt quizu uznaje tę odpowiedź za poprawną. Wszystko to można wykonać automatycznie.

Elastyczność

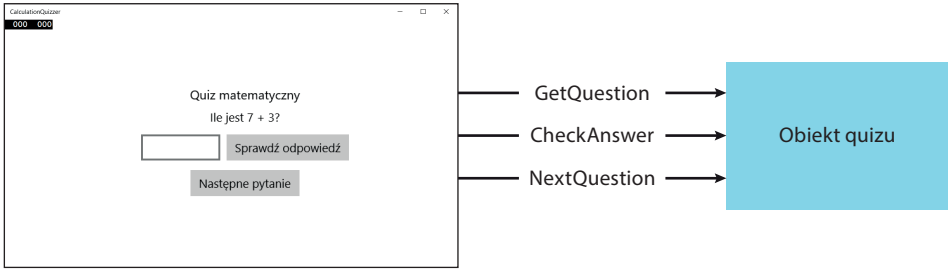
Kiedy użytkownicy popracują chwilę z naszym programem, przekonają się, że bardzo ułatwia on ćwiczenie dodawania. Może się okazać, że będą wtedy chcieli, aby program sprawdzał również ich wiedzę w zakresie odejmowania oraz mnożenia. A potem będą chcieli ponownego rozszerzenia programu o takie dziedziny wiedzy jak historia. Jeśli zachowanie testowe będzie wbudowane w wyświetlacz, będziemy musieli stworzyć nowy wyświetlacz dla każdego rodzaju quizu. Jeśli jednak quiz i wyświetlacz będą osobnymi obiektami, będziemy mogli podmieniać jeden obiekt quizu na drugi, a projekt interfejsu użytkownika będzie mógł pozostać taki sam. Jeżeli dobrze wykorzystamy interfejsy języka C#, czyli mechanizm, który poznałeś w rozdziale 15., będziemy mogli tworzyć wiele nowych rodzajów quizów i po prostu podłączać je do naszej aplikacji. Będziemy mogli utworzyć nawet quiz z wiedzy ogólnej, który będzie wykorzystywał pytania ze wszystkich naszych obiektów quizu.

Tworzenie obiektu quizu

Po uruchomieniu programu quizu strona wyświetlacza utworzy obiekt quizu, a następnie użyje tego obiektu do wyświetlenia użytkownikowi pytań i odpowiedzi. Relacje pomiędzy obiektem quizu a stroną wyświetlacza możemy rozważać w kategoriach komunikatów, które są wysłane, gdy odbywa się quiz. Strona wyświetlacza musi robić trzy rzeczy:

- musi pobrać tekst do wyświetlenia w na stronie w polu zapytania,
- musi sprawdzić odpowiedź podaną przez użytkownika,
- musi przejść do następnego pytania.

Te akcje odpowiadają z grubsza przyciskom na stronie, jak pokazano na rysunku 17.2.



Rysunek 17.2. Obiekt quizu i jego relacja ze stroną wyświetlacza

Możemy wyrazić te akcje w interfejsie C#:

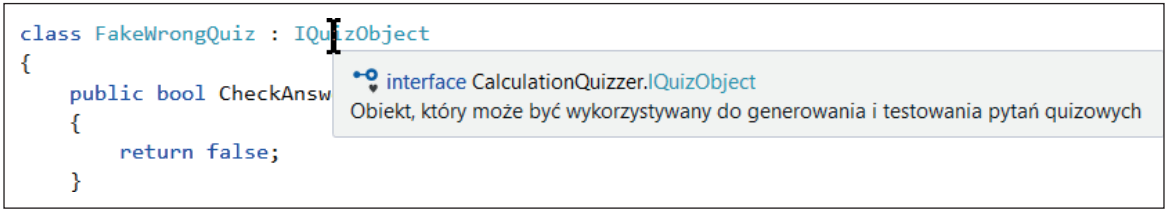
```
/// <summary>
/// Obiekt, który może być wykorzystany do generowania
/// i testowania pytań quizowych
/// </summary>
interface IQuizObject
{
    /// <summary>
    /// Pobiera tekst dla pytania
    /// </summary>
    /// <returns>tekst pytania</returns>
    string GetQuestion();

    /// <summary>
    /// Sprawdza, czy odpowiedź jest prawidłowa
    /// </summary>
    /// <param name="answer">odpowiedź do przetestowania</param>
    /// <returns>true, jeśli odpowiedź jest prawidłowa</returns>
    bool CheckAnswer(string answer);

    /// <summary>
    /// Przechodzi do następnego pytania
    /// </summary>
    void NextQuestion();
}
```

Poznałeś już interfejsy. Klasa może zaimplementować ten interfejs, a następnie być postrzegana w kategoriach zdefiniowanej przez niego funkcjonalności. Innymi słowy, możemy wziąć dowolną klasę i nadać jej te trzy metody zdefiniowane w interfejsie, a wtedy będzie można się do niej odwoływać poprzez referencję typu `IQuizObject`.

Profesjonalne komentarze

Wcześniej w tej książce zastanawialiśmy się, co czyni program „profesjonalnym". Sądzę, że jedną z rzeczy, które wyróżniają profesjonalny program, jest poziom i jakość komentarzy w kodzie źródłowym. Jak widać, interfejs `IQuizObject` ma tak naprawdę więcej komentarzy niż instrukcji języka C#. Same komentarze są napisane w formacie przypominającym XML, wprowadzonym w rozdziale 8., w podrozdziale „Dodawanie do metod komentarzy IntelliSense". Jak możesz pamiętać, komentarze zostały zaprojektowane do odczytu przez Visual Studio i służą dostarczaniu informacji IntelliSense, które wyświetlają się podczas edytowania programu.

Na rysunku 17.3 używam edytora kodu Visual Studio do utworzenia nowej klasy o nazwie `FakeWrongQuiz`, która implementuje interfejs `IQuizObject`. Kiedy umieszczam kursor nad nazwą interfejsu, wyskakuje okienko zawierające informacje o interfejsie, które zostały dodane w komentarzu.



Rysunek 17.3. Pomoc IntelliSense stanowi część profesjonalnego programu

Dostępność takich informacji znacznie ułatwia programiście tworzenie obiektów oraz używanie obiektów opracowanych przez innych. Pamiętaj, że Visual Studio tworzy dla Ciebie nawet szablony komentarzy; musisz jedynie wpisać trzy ukośniki (///) w edytorze Visual Studio nad metodą lub klasą, którą chcesz udokumentować. Dodając komentarze do metod, możesz zapewnić podsumowanie metody, opis każdego parametru oraz tego, co dana metoda zwraca. Kiedy widzę program zawierający takie komentarze, zaczynam uważać go za „profesjonalny" kawał roboty.

Tworzenie atrapy obiektu

Klasa `FakeWrongQuiz`, pokazana w poniższym kodzie, zapewnia implementacje wszystkich metod zdefiniowanych w interfejsie `IQuizObject`, ale te metody nie robią zbyt wiele. Pytaniem jest zawsze ten sam łańcuch znaków, a odpowiedź jest zawsze błędna.

```
class FakeWrongQuiz : IQuizObject
{
    public bool CheckAnswer(string answer)
    {
        return false;
    }

    public string GetQuestion()
    {
        return "Odpowiedź na to pytanie jest zawsze nieprawidłowa";
    }

    public void NextQuestion()
    {
    }
}
```

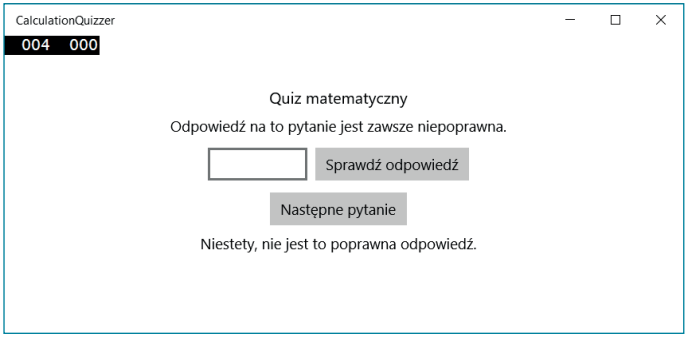
Implementuje IQuizObject.

Zawsze zwraca false.

Zawsze zwraca to samo pytanie.

Nie ma następnego pytania, więc ta metoda nie robi nic.

Ta klasa nie wygląda na zbyt przydatną, ale w rzeczywistości taka jest. Programista może jej użyć, aby przetestować rozwijany interfejs użytkownika. Może przygotować tę atrapę quizu zanim jeszcze ukończy prawdziwe obiekty quizu, co oznacza, że strona wyświetlacza i quizy mogą być rozwijane w tym samym czasie. Grupa Twoich przyjaciół może na przykład tworzyć obiekty quizu, a Ty możesz budować interfejs użytkownika. Kiedy wszyscy skończycie pisać kod, możesz połączyć ze sobą te klasy i otrzymać działający program. Rysunek 17.4 pokazuje użycie atrapy quizu.



Rysunek 17.4. Do testowania można użyć atrapy obiektu

Moglibyśmy również utworzyć atrapę obiektu quizu `FakeRightQuiz`, która mogłaby zostać użyta do przetestowania zachowania programu, gdy użytkownik poda prawidłową odpowiedź.

Tworzenie obiektu quizu dodawania

Obiekt quizu dodawania pobiera dwie liczby losowe, które następnie wykorzystuje do wygenerowania pytania. Liczby losowe są wytwarzane przez generator liczb losowych, który jest składową tego obiektu.

```
/// <summary>
/// Obiekt quizu, który implementuje quiz dodawania
/// </summary>
class AdditionQuizObject : IQuizObject
{
    /// <summary>
    /// Generator liczb losowych używany przez quiz
    /// </summary>
    private Random rand = new Random();

    /// <summary>
    /// Bieżące pytanie przekazywane przez ten obiekt w postaci łańcuhca znaków
    /// </summary>
    private string currentQuestion;

    /// <summary>
    /// Wartość odpowiedzi, która jest liczbą całkowitą
    /// </summary>
    private int currentAnswer;

    public string GetQuestion()
    {
        //Zwraca po prostu bieżące pytanie
        return currentQuestion;
    }

    public bool CheckAnswer(string answer)
    {

        int answerValue;

        // Konwertuje parametr na liczbę
        if (int.TryParse(answer, out answerValue))
        {
            // jeśli konwersja na liczbę zakończyła się sukcesem,
            // porównuje z odpowiedzią
            if (answerValue == currentAnswer)
                // zwraca true, jeśli odpowiedź jest prawidłowa
```

```
                return true;
            }
            // odpowiedź była zła albo użytkownik
            // nie wpisał liczby. Zwraca false
            return false;
        }

        public void NextQuestion()
        {
            // Generuje dwie liczby z przedziału od 0 do 9
            int firstNum = rand.Next(0, 10);
            int secondNum = rand.Next(0, 10);

            // Przechowuje łańcuch znaków zapytania
            currentQuestion = "Ile jest " + firstNum + " " + secondNum + "?";

            // Przechowuje prawidłową odpowiedź
            currentAnswer = firstNum + secondNum;
        }

        public AdditionQuizObject()
        {
            // Kiedy obiekt jest tworzony,
            // konfiguruje pierwsze pytanie
            NextQuestion();
        }
}
```

ANALIZA KODU

Rzut oka na klasę AdditionQuizObject

Pytanie: Dlaczego metoda `CheckAnswer` sprawdza łańcuch znaków, a nie liczbę?

Odpowiedź: Metoda `CheckAnswer` z obiektu quizu dostaje odpowiedzi udzielone przez użytkownika w celu porównania ich z poprawnymi wynikami. Jeśli dostarczona odpowiedź nie zgadza się z poprawną, metoda zwraca `false`, aby wskazać, że ta odpowiedź jest niepoprawna. Ten obiekt implementuje quiz matematyczny, który działa z liczbami, ale metoda `CheckAnswer` sprawdza łańcuch znaków zamiast liczby. Nie jest to jednak pomyłka. Oznacza to, że moglibyśmy zrobić quiz, który akceptowałby odpowiedzi tekstowe — na przykład nazwisko pierwszego prezydenta Stanów Zjednoczonych — oraz liczby. Przysparza to nieco więcej pracy obiektowi quizu dodawania, ponieważ musi on przekonwertować sprawdzany tekst na liczbę, ale dzięki temu quiz jest znacznie bardziej elastyczny.

Pytanie: Czy coś może powstrzymać przebiegłego programistę przed zerknięciem do odpowiedzi na pytania z quizu?

Odpowiedź: Tak. Tekst pytania i nader ważna odpowiedź zostały zaprojektowane jako reprezentujące dane prywatne składowe klasy `AdditionQuizObject`. Tylko metody działające w obiekcie `AdditionQuizObject` mają do nich dostęp, co oznacza, że podstępny programista nie może w żaden sposób odczytać odpowiedzi z obiektu quizu. Jest to jedna z sytuacji, w których ochrona danych w obiekcie jest bardzo dobrym pomysłem.

Pytanie: Jak możemy utrudnić zadania z dodawania?

Odpowiedź: Jednym ze sposobów byłoby rozszerzenie zakresu liczb generowanych przez program. Jest to kontrolowane przez zakres liczb losowych używanych przez metodę `NextQuestion`. W tej chwili ta metoda wykorzystuje wartości z zakresu od 0 do 9, ale możesz rozszerzyć zakres (i nawet zacząć od liczby ujemnej), jeśli chcesz utrudnić ćwiczenie odejmowania.

Tworzenie strony wyświetlacza quizu

Strona wyświetlacza quizu jest opisana w pliku XAML. Możemy użyć kontenera `StackPanel`, aby ułożyć w stos elementy wyświetlane na ekranie. Możemy również użyć poziomego kontenera `StackPanel`, żeby można było umieścić odpowiedź wprowadzaną przez użytkownika i przycisk *Sprawdź odpowiedź* w tej samej linii na ekranie.

```
<StackPanel VerticalAlignment="Center">
    <TextBlock Text="Quiz matematyczny" TextAlignment="Center" Margin="4"
     FontSize="16"></TextBlock>
    <TextBlock Name="questionTextBlock" Text="" TextAlignment="Center" Margin="4">
    </TextBlock>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center" Margin="4">
        <TextBox Name="answerTextBox" Width="100" Margin="4"
         TextAlignment="Center"></TextBox>
        <Button Content="Sprawdź odpowiedź" Name="checkAnswerButton"
         HorizontalAlignment="Center"  Margin="4"
         Click="checkAnswerButton_Click" ></Button>
    </StackPanel>
    <Button Content="Następne pytanie" Name="getNextQuestionButton"
     HorizontalAlignment="Center"  Margin="4"
     Click="getNextQuestionButton_Click" ></Button>
    <TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="4">
     </TextBlock>
</StackPanel>
```

Kod kryjący się za stroną XAML konfiguruje quiz w konstruktorze klasy, a następnie dostarcza procedury obsługi zdarzeń dla dwóch przycisków widocznych na ekranie:

```
public sealed partial class MainPage : Page
{
    IQuizObject activeQuiz;

    public MainPage()
    {
        this.InitializeComponent();

        activeQuiz = new FakeWrongQuiz();
        questionTextBlock.Text = activeQuiz.GetQuestion();
    }

    private void checkAnswerButton_Click(object sender, RoutedEventArgs e)
    {
        if (activeQuiz.CheckAnswer(answerTextBox.Text))
        {
            resultTextBlock.Text = "Odpowiedź prawidłowa! Dobra robota.";
        }
        else
        {
            resultTextBlock.Text = "Niestety, nie jest to poprawna odpowiedź.";
        }
    }

    private void getNextQuestionButton_Click(object sender, RoutedEventArgs e)
    {
        activeQuiz.NextQuestion();
        string nextQuestion = activeQuiz.GetQuestion();
        questionTextBlock.Text = nextQuestion;
        answerTextBox.Text = "";
        resultTextBlock.Text = "";
    }
}

Demo 17-01 Simple Quiz
```

Quiz używany przez tę stronę wyświetlacza.

Konstruktor

Konfiguruje aktywny quiz.

Umieszcza pytanie w polu bloku tekstowego.

Uruchamiane przy sprawdzaniu odpowiedzi.

Sprawdza, czy wprowadzono poprawną odpowiedź.

Przechodzi do następnego pytania.

Ustawia tekst dla następnego pytania.

ANALIZA KODU

Kompletny program

Ten kod jest dość zwięzły, ale wciąż interesujący.

Pytanie: Z jakiego konkretnego quizu korzysta ten program?

Odpowiedź: Zmienna `activeQuiz` jest ustawiana na aktywny w danym momencie quiz. Kiedy uruchomiona zostanie ta wersja quizu, zmienna `activeQuiz` zostanie ustawiona na instancję `FakeWrongQuiz`.

Pytanie: Jak zmienić to na quiz dotyczący dodawania?

Odpowiedź: Musimy tylko ustawić, żeby zmienna `activeQuiz` odwoływała się do instancji `AdditionQuiz`.

Dodawanie dźwięku i obrazów

Program quizu działa bardzo dobrze, ale wyświetlacz jest trochę nudny. Na początek możesz dodać jakiś dźwięk, a może nawet trochę obrazów.

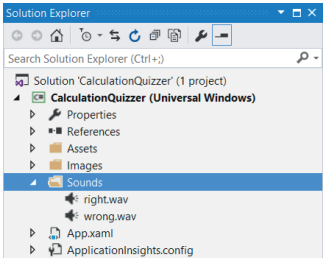
Dodawanie dźwięku

Zacznijmy od dźwięku. W przeszłości do wydawania dźwięków używaliśmy Snapów, ale teraz dowiesz się, jak dodać efekty dźwiękowe do uniwersalnej aplikacji systemu Windows 10. Zakres elementów wyświetlacza XAML nie jest ograniczony do tych, które mogą wyświetlać obrazy; mamy również elementy, które mogą wydawać dźwięki, a nawet odtwarzać filmy.

Oto XAML, który tworzy element multimedialny o nazwie `soundMediaElement`. Możemy go wykorzystać do odtwarzania dźwięków w naszych programach. Możemy umieścić go w dowolnym miejscu na ekranie, ponieważ zapewnia po prostu dane wyjściowe w postaci dźwięku. Typu `MediaElement` możesz także użyć do odtwarzania plików wideo, ale nie musimy tego robić w tym programie.

```
<MediaElement Name="soundMediaElement"></MediaElement>
```

Dźwięki dodajemy do aplikacji Windows 10 w taki sam sposób, jak w przeszłości dodawaliśmy dźwięki do programów biblioteki Snaps. Używamy plików dźwiękowych o rozszerzeniu *.wav* i przeciągamy je do zasobów naszego programu, jak pokazano na rysunku 17.5.



Rysunek 17.5. Doda wanie elementów dźwiękowych do Quizu matematycznego

Skoro mamy już element dźwiękowy i element XAML do wydawania dźwięku, musimy utworzyć kod C#, który odtworzy ten dźwięk.

```
private void checkAnswerButton_Click(object sender, RoutedEventArgs e)
{
    if (activeQuiz.CheckAnswer(answerTextBox.Text))
    {
        resultTextBlock.Text = "Odpowiedź prawidłowa! Dobra robota.";
        Uri soundsource = new Uri("ms-appx:///Sounds/right.wav");
        soundMediaElement.Source = soundsource;
        soundMediaElement.Play();
    }
    else
    {
        resultTextBlock.Text = "Niestety, nie jest to poprawna odpowiedź.";
        Uri soundsource = new Uri("ms-appx:///Sounds/wrong.wav");
        soundMediaElement.Source = soundsource;
        soundMediaElement.Play();
    }
}

Demo 17-02 Simple Quiz with Sound
```

Tworzy Uri, ustawia źródło i odtwarza element multimedialny.

Używa innego dźwięku dla niepoprawnych odpowiedzi.

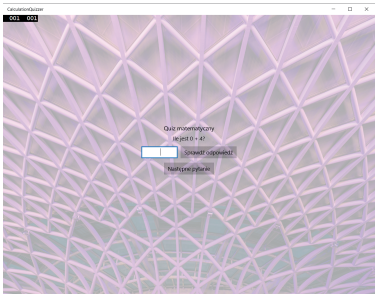
Kluczem do zrozumienia, jak działa dźwięk, jest zrozumienie użycia `Uri` (czyli ujednoliconego wskaźnika zasobów). Jest to element odwołujący się do określonego zasobu. Program może korzystać z wielu rodzajów zasobów, w tym dźwięków, obrazów i plików. Te zasoby mogą być umieszczone lokalnie na komputerze, przechowywane w samej aplikacji lub dostępne przez połączenie sieciowe. Ujednolicony wskaźnik zasobów to sposób na tworzenie referencji do tych zasobów, a te referencji mogą być używane przez elementy XAML. `Uri` może być utworzony z łańcucha znaków określającego lokalizację zasobu. Dodanie do lokalizacji zasobu przedrostka `ms-appx://` oznacza, że ten zasób jest zawartością przechowywaną w aplikacji.

Na rysunku 17.5 widać, że umieściłem dźwięki w folderze o nazwie *Sounds* w projekcie, więc adres moich dźwięków w programie musi zawierać ten folder w pliku ścieżki do zasobu.

Gdy mam `Uri`, który zapewnia adres zasobu, mogę ustawić ten adres jako wartość właściwości `Source` elementu `soundMediaElement`. (Niektóre elementy XAML mają właściwość `Source`, która określa, skąd pobierają swoją zawartość). Wartość `Source` jest ustawiana na `Uri` elementu, który ma być odtwarzany przez `MediaElement`. Finalna deklaracja, czyli wywołanie metody `Play`, powoduje, że `MediaElement` odtwarza element multimedialny, który jest przypisany do jego źródła. W tym przypadku odtwarzany jest odpowiedni efekt dźwiękowy. Możesz zmienić odtwarzany dźwięk, zmieniając po prostu element źródłowy.

Dodawanie obrazów

Do stron XAML możesz dodawać obrazy za pomocą elementu `Image`. Mógłbyś po prostu dodać obraz do interfejsu użytkownika, ale chciałbym raczej dodać tło dla całej aplikacji, aby strona wyglądała tak, jak pokazano na rysunku 17.6.



Rysunek 17.6. Dodawanie tła do strony

Tło powinno wypełnić cały ekran. Mogę to osiągnąć, wykorzystując sposób, w jaki element wyświetlacza `Grid` działa w XAML. Możemy to wykorzystać do ułożenia elementów na ekranie. Jest to nieco trudniejsze niż użycie kontrolki `StackPanel`, ale daje większą kontrolę nad tym, która część wyświetlacza jest przypisywana do poszczególnego elementu.

Gdy tworzysz projekt XAML, wyświetlacz składa się z siatki zawierającej tylko jedną komórkę, która formuje cały ekran. Wszystkie elementy w tej siatce są rysowane jeden na drugim w kolejności, w jakiej są podane w elemencie `Grid`. Jeśli więc utworzę element `Image` i rozszerzę go, aby wypełniał cały ekran, zapewni to dość ładne tło. To wszystko mogę tak naprawdę zrobić z poziomu projektu XAML. Oto sposób dodawania obrazu tła do strony:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Image Source="ms-appx:///Images/kingscross.jpg"
    Opacity="0.3"
    Stretch="Fill">
  </Image>
  <StackPanel VerticalAlignment="Center">
  </StackPanel>
</Grid>

Demo 17-03 Simple Quiz with Background
```

StackPanel, który tworzy wyświetlacz aplikacji.

Wartość `Opacity`, która jest podawana w zakresie od 0 do 1, jest całkiem użyteczna, ponieważ kontroluje jak duża część tła jest wyświetlana na obrazie. Jeśli umieścisz po prostu obraz za kontrolkami, staną się one trudne do zobaczenia na tle obrazu. Im mniejsza wspomniana wartość, tym słabiej widoczny jest obraz. Ustawienie `Opacity` na 1 daje jednolity obraz, ale

wartość 0 sprawi, że obraz będzie całkowicie przezroczysty. Z moich testów wynika, że dobrze sprawdza się wartość 0,3. Obraz jest widoczny, ale nadal można zobaczyć kontrolki.

ZRÓB TO SAM

Przygotuj imponujący quiz

Możesz użyć obrazów i dźwięków, aby utworzyć bardzo interesujący wyświetlacz quizu. Źródło dla `Image` możesz zmienić zarówno z poziomu programu, jak i projektu XAML, możesz więc zmienić tło programu, jeśli użytkownik udzieli złej odpowiedzi na pytanie. Możesz także ułożyć obrazy jeden na drugim i używać ich ustawień stopnia nieprzejrzystości, aby je scalić. Spróbuj utworzyć naprawdę imponującą wersję programu quizu lub użyj tych technik, aby ulepszyć jakieś inne napisane przez Ciebie programy.

Obsługa wielu quizów

Jak można się było spodziewać, Twoi znajomi chcą teraz wersji programu obsługującej różne typy quizów. Możemy zacząć od zbudowania quizu służącego do ćwiczenia trzech kolejnych działań arytmetycznych: odejmowania, mnożenia i dzielenia.

Tworzenie klas quizów

Możesz uznać, że dobrym pomysłem byłoby rozszerzenie klasy `AddQuizObject` w celu utworzenia nowych klas, które mogą wykonywać te funkcje, ale nie wydaje mi się, żeby był to dobry plan. Rozszerzamy klasę, a następnie nadpisujemy jej metody, aby tworzyć mniej abstrakcyjną wersję klasy potomnej, a nie zupełnie inną.

W aplikacji bankowej można na przykład rozszerzyć klasę `Account` (rachunek), aby otrzymać klasę `CheckingAccount` (rachunek bieżący), a następnie rozszerzyć `CheckingAccount`, żeby wygenerować `CheckingAccountWithOverdraft` (rachunek bieżący z kredytem odnawialnym). Wszystkie te klasy robią tę samą fundamentalną rzecz; chodzi po prostu o to, że klasy potomne służą nieco innym scenariuszom. Jednak rozszerzenie obiektu dodawania, aby utworzyć obiekt odejmowania, wydaje mi się niewłaściwe, ponieważ ta klasa potomna robi coś zupełnie innego niż rodzic i nie zapewnia bardziej wyspecjalizowanej wersji rodzica. Tak naprawdę musimy być bardziej abstrakcyjni i utworzyć klasę `CalculationQuizObject`, która będzie następnie rozszerzana w celu uzyskania różnych rodzajów quizów matematycznych. Rysunek 17.7 przedstawia projekt hierarchii klas dla quizów.

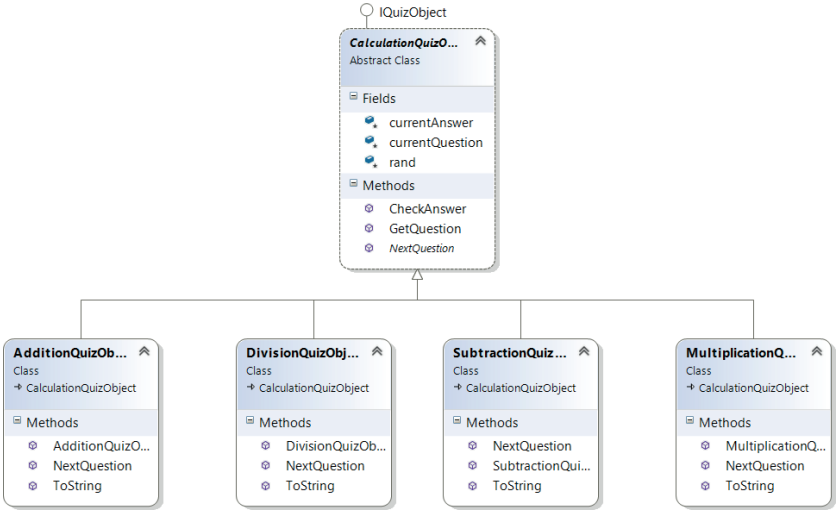
Z PUNKTU WIDZENIA PROGRAMISTY

Prawdopodobnie najlepszego projektu nie obmyślisz już na samym początku

Jeśli chodzi o kompilator C#, nieistotne jest dla niego znaczenie Twoich klas oraz to, czy projekt jest idealny. Tak samo nieistotne będą te kwestie dla użytkowników Twojego programu. Dopóki będą otrzymywali działający quiz, będą zadowoleni. Silny argument przemawia za tym, aby myśleć zgodnie z zasadą: „Gdy tylko wymyślisz projekt, który potencjalnie będzie mógł zadziałać, możesz przestać projektować program i zacząć go budować. Przynajmniej w ten sposób klient coś otrzyma".

Z mojego doświadczenia wynika, że mogę poświęcić dużo czasu na poszukiwanie idealnego projektu dla rozwiązania, a potem, kiedy zaczynam tworzyć program, wpadam na dużo lepszy pomysł. Dotyczy to wielu dziedzin, w których prototypy służą do testowania idei i przekazywania informacji do procesu produkcyjnego. Programiści mają technikę zwaną „refaktoryzacją", która polega na zmienianiu układu klas w programach w trakcie ich tworzenia. Bardziej zaawansowane wersje Visual Studio zapewniają potężne narzędzia, które mogą być w tym pomocne.

Nie ma nic złego w ulepszaniu projektu oprogramowania w trakcie jego tworzenia, ale dla mnie najważniejsze jest to, abyś przed rozpoczęciem budowania proponowanego projektu bardzo starannie upewnił się, że będzie on rzeczywiście działał. Omówiliśmy już, jak bolesne może być przeżycie niemiłego zaskoczenia; najlepiej robić wszystko, aby tego uniknąć.

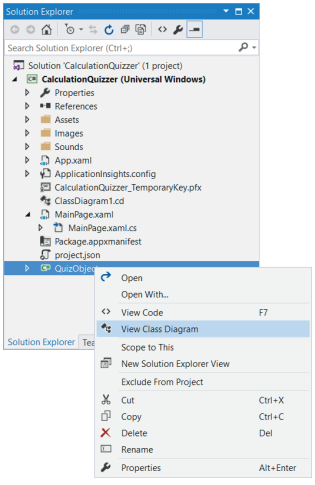


Rysunek 17.7. Hierarchia klas quizów

Jedyną metodą, która jest różna w każdej z klas potomnych, jest metoda `NextQuestion`, która ustawia tekst pytania i oblicza odpowiedź. Poniżej pokazano metodę `NextQuestion` z klasy `MultiplicationQuizObject`. Nadpisuje ona metodę abstrakcyjną z klasy `CalculationQuizObject` w celu zapewnienia zachowania dla zadań z mnożeniem.

```
public override void NextQuestion()
{
    int firstNum = rand.Next(0, 10);
    int secondNum = rand.Next(0, 10);
    currentQuestion = "Ile jest " + firstNum + " * " + secondNum + "?";
    currentAnswer = firstNum * secondNum;
}
```

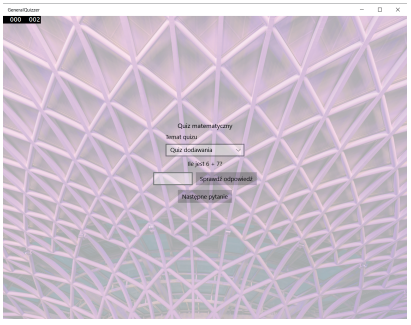
Diagram klas pokazany na rysunku 17.7 został w rzeczywistości wygenerowany przez Visual Studio 2015 z napisanego przeze mnie programu. Rysunek 17.8 pokazuje, jak to zrobić: kliknij prawym przyciskiem myszy dowolny z plików źródłowych w oknie *Solution Explorer*, wybierz opcję *View Class Diagram*, a Visual Studio narysuje diagram przedstawiający wzajemne relacje klas. Możesz użyć takiego diagramu do tworzenia nowych klas w Twoim rozwiązaniu.



Rysunek 17.8. Użyj tych opcji, aby wyświetlić diagram klas

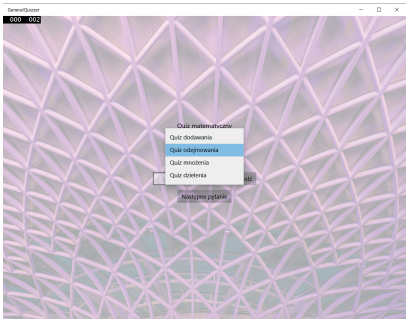
Wybieranie rodzaju quizu za pomocą kontrolki ComboBox

Skoro mamy już zestaw klas obliczeniowych quizu, potrzebujemy sposobu, aby użytkownicy mogli wybierać rodzaj pytań, na które chcą odpowiadać. Możemy użyć kontrolki XAML o nazwie `ComboBox`, żeby umożliwić użytkownikowi wybór tematu quizu. Rysunek 17.9 pokazuje, jak to wygląda w uruchomionym programie.



Rysunek 17.9. Pole selektora quizu z wybranym quizem dodawania

Jeżeli użytkownik będzie chciał przejść do innego quizu, może rozwinąć pole wyboru i kliknąć nowy typ quizu. Rysunek 17.10 pokazuje opcje przy rozwiniętym polu wyboru.



Rysunek 17.10. Użytkownicy wybierają typ quizu za pomocą pola wyboru

Użytkownikom spodoba się pomysł zastosowania tej opcji, ponieważ widzieli ją w wielu innych programach. Teraz musimy dodać pole wyboru do naszego programu. Poniżej pokazano XAML opisujący kontrolkę `ComboBox`. Do kontrolki `ComboBox` odwołujemy się w naszym programie za pomocą nazwy `quizTopicComboBox`.

```
<ComboBox Header="Temat quizu"
    Name="quizTopicComboBox"
    Width="200" Margin="4"
    HorizontalAlignment="Center"
    SelectionChanged="quizTopicComboBox_SelectionChanged">
</ComboBox>
```

Zdarzenie, które jest uruchamiane, gdy zmienia się wybór.

Nadaje nagłówek polu wyboru.

Teraz musimy skonfigurować kontrolkę `ComboBox` z opcjami, które powinni mieć do wyboru użytkownicy.

```
public MainPage()
{
    this InitializeComponent();
    quizTopicComboBox.Items Add(new AdditionQuizObject());
    quizTopicComboBox.Items Add(new SubtractionQuizObject());
    quizTopicComboBox.Items Add(new MultiplicationQuizObject());

    quizTopicComboBox.SelectedIndex = 0;
}
```

Dodaje obiekt quizu do pola wyboru ComboBox.

Wybiera początkowy obiekt quizu.

Kontrolka `ComboBox` przechowuje kolekcję elementów (`Items`) i pozwala użytkownikowi wybrać jeden element z tej kolekcji. Program może dodawać elementy do kolekcji, z której może wybierać je użytkownik. W przypadku aplikacji *GeneralQuizzer* elementy to różne obiekty quizu. Przy uruchamianiu programu musimy dodać każdy typ quizu do elementów w `quizTopicComboBox`. Najlepszym miejscem, w którym możemy to zrobić, jest konstruktor strony. Elementy w `ComboBox` działają dokładnie tak samo, jak kolekcje `List`, które widziałeś wcześniej. Program może dodawać elementy do listy i będą one zarządzane przez kontrolkę `ComboBox`.

Po skonfigurowaniu elementów z `quizTopicComboBox` musimy się upewnić, że przy uruchamianiu programu jeden z elementów będzie wybrany. Program może to zrobić poprzez ustawienie wartości właściwości `SelectedIndex` kontrolki `quizTopicComboBox`. W powyższym kodzie ustawiłem wartość `SelectedIndex` dla `quizTopicComboBox` na pierwszy element (ten o indeksie 0). Efekt tego działania jest taki sam, jakby użytkownik otworzył `quizTopicComboBox` na ekranie, a następnie wybrał element znajdujący się na początku listy opcji. Powoduje to również wystąpienie zdarzenia `SelectionChanged` w `quizTopicComboBox`.

```
IQuizObject activeQuiz;

private void quizTopicComboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    activeQuiz = (IQuizObject) quizTopicComboBox.SelectedItem;
    setupNextQuestion();
}
```

Aktualnie aktywny plik.

Pobiera zaznaczony element i używa go do ustawienia wartości activeQuiz.

Pobiera następne pytanie dla tego quizu.

ANALIZA KODU

Zdarzenia zmiany wyboru

Nie pracowaliśmy wcześniej z elementem `ComboBox`. Rodzi to kilka interesujących pytań, które musimy rozważyć.

Pytanie: Kiedy uruchamiana jest metoda `SelectionChanged`?

Odpowiedź: Odpowiedź na to pytanie jest zarówno prosta, jak i skomplikowana. Ta metoda jest uruchamiana, gdy użytkownik zmienia zaznaczenie w polu `ComboBox` podczas działania programu. Jednak metoda `SelectionChanged` jest również uruchamiana, jeśli cokolwiek w naszym programie zmieni zaznaczenie w kontrolce `ComboBox`. Używamy tego mechanizmu z dobrym skutkiem, gdy program jest uruchamiany i musimy wybrać początkowy quiz. Jedna z sytuacji, które zawsze kończą się źle (i ja popełniłem ten błąd), to taka, w której metoda obsługi zdarzeń `SelectionChanged` zmienia wybrany element w kontrolce `ComboBox`, wywołując kolejne zdarzenie `SelectionChanged`. Inicjuje to nieskończoną sekwencję zmian, która w rezultacie blokuje komputer.

Pytanie: Skąd pochodzą nazwy elementów?

Odpowiedź: Na zrzutach ekranu z programu wygląda na to, że kontrolka `ComboBox` pokazuje nazwy poszczególnych rodzajów quizu i pozwala użytkownikowi wybrać jeden z nich. Skąd jednak pochodzą te nazwy? Odpowiedź jest taka, że `ComboBox` używa zachowania `ToString` do uzyskania tekstów reprezentujących każdy z elementów, spośród których użytkownik może wybierać. Poznałeś już metodę `ToString`; za jej pomocą instruujemy dowolny obiekt, by podał łańcuch znaków opisujący jego zawartość. Wewnątrz klasy `MultiplicationQuizObject` mamy na przykład następującą metodę `ToString`:

```
public override string ToString()
{
   return "Quiz mnożenia";
}
```

Pytanie: W jaki sposób element z `ComboBox` jest używany do ustawienia aktywnego quizu?

Odpowiedź: Za pomocą klasy `ComboBox` można wybrać dowolny element. Przechowuje ona listę obiektów, z których może wybierać użytkownik. W przypadku `quizTopicComboBox` wszystkie elementy z listy są obiektami, które implementują interfejs `IQuizObject` (czyli możemy używać ich do quizów). Jednak lista elementów w `ComboBox` musi być listą obiektów. Oznacza to, że program musi przekonwertować wybrany element (którym jest referencja do obiektu) na referencję do obiektu `IQuizObject`, którego można użyć w quizie. Do wykonania tej konwersji używa rzutowania. Korzystaliśmy wcześniej z tej techniki, kiedy chcieliśmy poinformować kompilator, że można konwertować jeden typ na drugi. Typ, którego chcemy użyć, jest podany w nawiasach przed wartością, którą chcemy przekonwertować.

```
activeQuiz = (IQuizObject) quizTopicComboBox SelectedItem;
```

ZRÓB TO SAM

Utwórz program quizu z wiedzy ogólnej przeprowadzanego na czas

Możesz teraz napisać kompletny program quizu. Możesz utworzyć zestaw różnych obiektów quizu obsługujących różne tematy, a następnie pozwolić użytkownikowi wybierać spośród nich. Możesz nawet ustawić określony w sekundach czas, w którym użytkownik musi odpowiedzieć na pytanie. Program quizu może rejestrować, w którym momencie wyświetlone zostało pytanie, a następnie sprawdzać czas naciśnięcia przycisku odpowiedzi. Im szybciej zostanie udzielona odpowiedź, tym wyższy wynik. Możliwe jest odjęcie jednej wartości `DateTime` od drugiej i wygenerowanie wartości `TimeSpan`, której program może użyć określenia długości interwału czasowego.

Czego się nauczyłeś?

W tym rozdziale nauczyłeś się właściwego sposobu tworzenia interfejsu użytkownika dla rozwiązania obiektowego. Dowiedziałeś się, że interfejs użytkownika powinien być bardzo cienką warstwą kodu, która dostarcza komunikaty do obiektu zapewniającego zachowania programu. Taki tryb pracy ma wiele zalet. Dużo łatwiej jest przetestować obiekt implementujący aplikację, ponieważ można napisać programy symulujące działania interfejsu użytkownika i sprawdzać odpowiedzi obiektu, dzięki któremu to działa. Możesz także utworzyć „atrapowe" interfejsy użytkownika lub obiekty biznesowe, które mogą być wykorzystywane do testowania systemu w trakcie jego opracowywania. Umożliwia to także pracę nad interfejsem użytkownika i zachowaniami biznesowymi w tym samym czasie. Przekonałeś się, że mechanizm interfejsu C# jest bardzo przydatny w takiej sytuacji, ponieważ można go użyć do określenia charakteru wywołań metod, które będą używane do przekazywania komunikatów.

Zobaczyłeś także inne zastosowanie dla hierarchii klas, kiedy tworzyliśmy zbiór różnych obiektów liczbowych quizu, które różniły się tylko samodzielnym generowaniem rzeczywistego pytania i były w stanie udostępnić wszystkie pozostałe zachowania. Jeśli chodzi o interfejs użytkownika, nauczyłeś się dodawać obrazy i dźwięki za pomocą elementów `MediaElement` i `Image`. Na koniec zobaczyłeś, jak użytkownik może wybierać spośród szeregu opcji za pomocą pola wyboru `comboBox`.

Pytanie: Dlaczego słowo „interfejs" jest używane w tak mylący sposób?

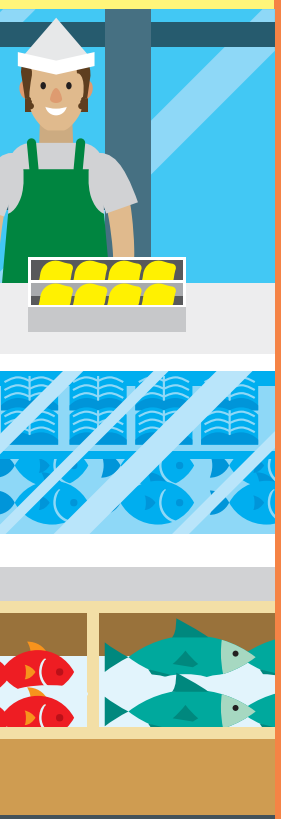
Mam prawdziwy problem z tym pytaniem. Ilekroć mówimy o interfejsach, musimy uważać na to, co naprawdę mamy na myśli. Interfejs użytkownika to zbiór elementów, które składają się na doświadczenie użytkownika podczas interakcji z programem. Interfejs języka C# to zestaw zachowań, które klasa może implementować. Widzę, dlaczego to słowo może mieć zastosowanie w obu sytuacjach, ale szkoda, że to samo słowo jest ostatecznie używane w obu kontekstach.

Czy moje obiekty biznesowe muszą komunikować się z interfejsem użytkownika?

Nie. To właśnie piękno tworzenia obiektów, które działają w ten sposób. Po zdefiniowaniu kanałów, których można używać do instruowania obiektu, by wykonał pewne czynności, możesz używać tych kanałów na różne sposoby. Dostęp do obiektu mogą uzyskać komunikaty pochodzące z połączenia sieciowego lub programu udającego użytkownika siedzącego przy klawiaturze.

18.

Zaawansowane zagadnienia aplikacji



Czego nauczysz się w tym rozdziale?

Przeszedłeś długą drogę od czasu, gdy korzystając z frameworku Snaps tworzyłeś timery i aplikacje do anonsowania gości przybywających na przyjęcie. Działanie Twoich pierwszych programów polegało na tym, że pobierały pewne dane, przetwarzały je w jakiś sposób, a następnie generowały dane wyjściowe. Ten schemat pozostaje niezmienny, ale teraz myślisz w kategoriach obiektów, które przyjmują komunikaty, wykonują na ich podstawie pewne działania, a następnie wysyłają komunikaty do innych obiektów. Programowanie to nie tylko szukanie rozwiązań dla zadanych problemów. Chodzi również o nadanie tym rozwiązaniom odpowiedniej struktury, aby łatwo było je testować, budować, wdrażać i utrzymywać.

Ten rozdział będzie dobrym punktem startowym w Twojej karierze programisty aplikacji. Wykorzystasz aplikację *Quiz matematyczny* z poprzedniego rozdziału i dowiesz się, jak wiązać dane przechowywane w obiektach z elementami na wyświetlaczu. Zobaczysz również, jak programiści tworzą obiekty, których jedyną rolą jest zapewnienie widoku danych w systemie. Są to dość skomplikowane rzeczy, ale mają ogromny potencjał. Jeśli poczujesz się zdezorientowany, spróbuj przypomnieć sobie, dlaczego właściwie wykonujemy daną akcję i przeczytaj sekcje „Analiza kodu" dla tego przykładu. Ten materiał może być trudny do opanowania, ale zdobyta wiedza bardzo Ci się przyda, jeśli kiedykolwiek będziesz ubiegać się o pracę jako programista. Na początek poznasz kilka sprytnych funkcjonalności języka C#, których można używać do przyspieszania procesu pisania programu.

Przyspieszanie pisania kodu C#

Jest to dobry moment, aby wspomnieć o kilku sztuczkach, które można wykorzystać w celu przyspieszenia i ułatwienia procesu pisania kodu C#. Nie musisz ich używać we wszystkich swoich programach, ale możesz uznać je za przydatne. Prawdopodobnie napotkasz je też w programach napisanych przez innych programistów.

Skracanie instrukcji

Do skracania instrukcji możesz używać różnych operatorów. Dotychczas przyglądaliśmy się operatorom, które pojawiały się w wyrażeniach i działały na dwóch operandach, na przykład:

```
age = age + 1;
```

W tym przypadku operatorem jest znak plusa (+). Operuje on na zmiennej `age` i wartości 1. Celem tej instrukcji jest dodanie 1 do zmiennej `age`. Jest to jednak wyrażone w dość rozwlekły sposób, zarówno pod względem tego, co musisz wpisać, jak i tego, co faktycznie będzie musiał zrobić komputer, gdy uruchomi ten program. Język C# pozwala skrócić zapis tej operacji do następującej linii kodu:

```
age++;
```

Dzięki temu możesz wyrazić się bardziej zwięźle, a kompilator może wygenerować wydajniejszy kod, ponieważ teraz wie, że dodajesz liczbę 1 do określonej zmiennej. Ten podwójny znak plusa (++; jest to operator inkrementacji) nazywany jest operatorem **jednoargumentowym**, gdyż działa tylko na jednym operandzie. Powoduje to zwiększenie o 1 wartości znajdującej się w tym operandzie. Istnieje też odpowiadający mu operator `--`, którego można użyć do zmniejszenia (dekrementacji) wartości zmiennych.

Kolejny przykład skrótu związany jest z dodawaniem określonej wartości do zmiennej. Gdybyśmy mieli program, w którym uzyskiwalibyśmy różne wyniki za zabicie poszczególnych rodzajów kosmitów, moglibyśmy napisać tak:

```
totalScore = totalScore + alienValue;
```

Jest to całkowicie w porządku, ale znowu jest dość rozwlekłe. Język C# ma kilka dodatkowych operatorów, które pozwalają skracać tego typu instrukcje do postaci takiej jak:

```
totalScore += alienValue;
```

Operator += łączy dodawanie i przypisywanie, dzięki czemu wartość w `totalScore` jest zwiększana o wartość `alienValue`. Tabela 18.1 pokazuje kilka przykładów operatorów dla skrótów:

Tabela 18.1. Operatory skrótów

OPERATOR	EFEKT
`a += b`	Wartość w *a* jest zastępowana przez *a* + *b*
`a -= b`	Wartość w *a* jest zastępowana przez *a* – *b*
`a /= b`	Wartość w *a* jest zastępowana przez *a* : *b*
`a *= b`	Wartość w *a* jest zastępowana przez *a* · *b*

Istnieją jeszcze inne kombinacje operatorów. Znalezienie ich pozostawiam Tobie. Na początek wyszukaj `msdn operatory C#`.

Instrukcje i wartości

W języku C# instrukcje mają wartość, której możesz użyć w swoim programie. Poniższa instrukcja przypisuje na przykład wartość 0 do zmiennej `score` (wynik).

```
score = 0;
```

Możesz użyć tej instrukcji na początku gry wideo, aby ustawić wartość wyniku na 0. Jednak sama instrukcja ma wartość 0, więc jeśli chcesz, możesz napisać tak:

```
hits = score = 0;
```

Ta instrukcja ustawia wartość zmiennej `hits` (trafienia) na rezultat wykonania instrukcji `score = 0`. Innymi słowy, zmienna `hits` również jest ustawiana na 0. Jeżeli potrzebujesz robić tego rodzaju rzeczy (a przyznaję, że nie jestem ich zwolennikiem), radziłbym Ci używać nawiasów, w sposób pokazany poniżej, aby było to bardziej przejrzyste.

```
hits = (score = 0);
```

Użycie wyników z operatorów jednoargumentowych w testach

Podczas korzystania z operatorów takich jak ++, może wystąpić niejednoznaczność polegająca na tym, że nie będziesz wiedzieć, czy otrzymasz wartość instrukcji przed inkrementacją, czy po.

Język C# zapewnia sposób uzyskania każdej z tych wartości, w zależności od żądanego efektu. Zmieniając pozycję znaków ++ możesz określić, czy chcesz zobaczyć wartość przed, czy po wykonaniu dodawania:

- `i++` oznacza „podaj wartość przed inkrementacją",
- `++i` oznacza „podaj wartość po inkrementacji".

Poniższy kod dałby wartość `j` równą `3`.

```
int i = 2, j ;
j = ++i ;
```

Wszystkie pozostałe operatory specjalne, czyli += itd., zwracają wartość po wykonaniu instrukcji operatora.

Z PUNKTU WIDZENIA PROGRAMISTY

Zawsze dąż do prostoty

Nie daj się ponieść. Fakt, że możesz tworzyć kod taki jak pokazano poniżej, nie znaczy, że powinieneś to robić.

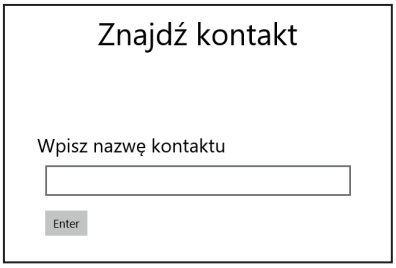
```
height = width = speed = count = size = 0 ;
```

Gdy obecnie piszę jakiś program, w pierwszej kolejności zastanawiam się, czy będzie on łatwy do zrozumienia. Nie traktuję tego jako obowiązku wobec komputera, ale względem osób, które będą musiały czytać mój kod. Nie wydaje mi się, aby powyższa instrukcja była bardzo łatwa do prześledzenia, więc bez względu na to, w jak dużym stopniu może zwiększać wydajność, i tak nie stosuję podobnych konstrukcji.

Tworzenie edytora kontaktów dla systemu Windows 10

Wróćmy do aplikacji menedżera kontaktów, którą zbudowaliśmy w rozdziale 10. Prawniczka, dla której pracowałeś, widziała inne aplikacje będące jej zdaniem lepsze niż Twój program, więc oczekiwałaby kilku ulepszeń. Na początek chciałaby zmienić sposób wyszukiwania kontaktu, z którym potrzebuje pracować. W tej chwili prawniczka wpisuje nazwę kontaktu, a program jej szuka.

Rysunek 18.1 pokazuje, jak to działa. Po naciśnięciu przycisku *Enter* program szuka kontaktu o podanej nazwie.



Rysunek 18.1. Wyszukiwanie kontaktów

Działa to bez zarzutu i prawniczka korzysta z tego już od jakiegoś czasu, ale teraz chce czegoś nieco łatwiejszego w użyciu. Oczekiwałaby aplikacji, w której może wybierać nazwy kontaktów z listy, jak pokazano na rysunku 18.2. Prawniczka wpisywałaby szukany łańcuch znaków, a następnie naciskała przycisk *Szukaj*. Program wyświetlałby wszystkie kontakty zawierające w nazwie szukany łańcuch znaków, a prawniczka mogłaby wybrać ten, z którym chce pracować. W tym przykładzie przed naciśnięciem *Szukaj* wpisała *r*. Lista pokazuje wszystkie kontakty, których nazwa zawiera literę *r*.

Rysunek 18.2. Ulepszony interfejs wyszukiwania

Wydaje się, że czeka nas dobra zabawa przy budowaniu tej funkcjonalności, a przy okazji możesz poćwiczyć swoje nowo nabyte umiejętności XAML, więc spróbujmy to zrobić.

ZRÓB TO SAM

Napisz aplikację do zarządzania danymi

To dobry moment, żebyś zaczął tworzyć własne, w pełni funkcjonalne aplikacje oparte na danych. Kilka następnych stron książki poświęcimy na zbadanie, jak utworzyć profesjonalną aplikację do zarządzania kontaktami, przeznaczoną dla platformy Windows 10. Możesz użyć dokładnie tych samych technik, aby zbudować aplikację działającą z dowolnym rodzajem danych.

Przechowywanie szczegółowych informacji o kontaktach

Aby przechowywać w aplikacji dane o kontaktach, użyliśmy klasy o nazwie `Contact`. Przechowywała ona nazwę, adres, numer telefonu i liczbę minut dla danego kontaktu. Wszystkie dane były przechowywane w składowych klasy `Contact`. Tutaj użyjemy tego samego wzorca, ale wprowadzimy jedną zmianę w sposobie działania naszego oldschoolowego projektu opartego na klasach.

W nowej wersji klasy `Contact` wszystkie elementy danych w klasie będą przechowywane jako właściwości publiczne. Właściwości użyliśmy w rozdziale 10., gdzie odkryłeś, że są one świetnym sposobem na uzyskanie kontroli, gdy program wchodzi w interakcję z danymi w obiekcie. Swoją wiedzę na temat właściwości możesz odświeżyć, przyglądając się niewielkiej klasie `Contact`, którą pokazano poniżej. Zawiera ona tylko jedną składową — właściwość `Name` kontaktu. Kompletną klasę ze wszystkimi właściwościami danych zobaczysz nieco później.

```
public class Contact
{
    // Prywatny łańcuch znaków przechowujący wartość nazwy ze zmiennej name
    private string name;

    // Publiczny łańcuch znaków nazwy
    public string Name
    {
        // Pobiera wartość nazwy
        get
        {
            // Zwraca wartość nazwy
            return name;
        }
```

```
        // Ustawia wartość nazwy
        set
        {
            // Ustawia wartość nazwy na value
            name = value;
        }
    }
}
```

Ten kod jest uruchamiany, gdy program zapisuje coś w zmiennej.

Skoro mamy już naszą klasę, możemy używać jej w programach w następujący sposób:

```
Contact rob = new Contact();
rob Name = "Rob";
```

Te dwie instrukcje tworzą nowy obiekt `Contact`, a potem ustawiają nazwę kontaktu na `Rob`. Kiedy wykonane zostanie przypisanie do właściwości `Name`, uruchamiane jest zachowanie `set` wewnątrz klasy, które ustawia wartość nazwy na `Rob`. To bardzo ważny aspekt naszego nowego projektu. Wiąże się to z tym, że kod w naszym obiekcie może przejąć kontrolę, gdy coś się stanie z danymi w tym obiekcie. Kolejnym ważnym aspektem tej zmiany jest to, że obiekty danych mogą wchodzić w interakcje z systemem wyświetlania, wykorzystując udostępniane przez siebie właściwości.

W języku C# możesz pisać znacznie krótsze definicje właściwości, jeśli chcesz jedynie, żeby wartość stała się właściwością. Są to tak zwane **automatycznie implementowane** właściwości — kompilator automatycznie tworzy dla Ciebie prywatną zmienną kryjącą się za właściwością.

```
public class Contact
{
    public string Name { get; set; }
}
```

Ten kod sprawia, że składowa `Name` klasy `Contact` staje się właściwością, zapewnia także zachowania `get` i `set`. Jeśli jednak chcesz uzyskać kontrolę, kiedy właściwość jest używana, musisz rozszerzyć tę właściwość do pełnej wersji, którą widziałeś wcześniej. Oto kompletna klasa `Contact`, której będziemy używać w naszej nowej wersji aplikacji *Rejestr czasu pracy*, ze wszystkimi elementami danych w postaci automatycznie zaimplementowanych właściwości.

```
public class Contact
{
    // Wartości reprezentujące dane jako automatycznie implementowane właściwości

    public string Name { get; set; }
    public string Address { get; set; }
```

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

    public int MinutesSpent { get; set; }

    // Konstruktor instancji kontaktu
    public Contact(string name, string address, string phone)
    {
        // Kopiuje przychodzące wartości do właściwości
        this.Name = name;
        this.Address = address;
        this.Phone = phone;
    }
}
```

Klasa ma również konstruktor, który służy do konfigurowania instancji z nazwą, adresem i numerem telefonu.

```
Contact rob = new Contact(name:"Rob", address:"Rob's house", phone: "0000 11111
2222");
```

Ta instrukcja tworzy nowy kontakt o nazwie `rob` z wartości, które wprowadziłem w kodzie programu. Ukończony program będzie odczytywał te informacje z elementów `TextBox`.

Zapisywanie wielu kontaktów

Utworzony przez nas wcześniej program przechowywał wszystkie elementy kontaktów w obiekcie `List`, który był przechowywany wewnątrz aplikacji. Tym razem utworzymy klasę, która będzie zarządzać magazynem danych. Będzie ona zawierała listę kontaktów (obiekt `List`), zapewniała zachowania umożliwiające programom zarządzanie kontaktami i wyszukiwanie ich, a także utworzy testowy zestaw danych, abyśmy mogli się nim pobawić.

```
public class ContactStore
{
    // Zapisywanie listy kontaktów
    private List<Contact> contacts = new List<Contact>();

    // Zapisywanie kontaktu w magazynie danych
    public void StoreContact(Contact contact)
    {
        // Dodawanie kontaktu do listy
        contacts.Add(contact);
```

```
    }

    // Usuwanie kontaktu z magazynu danych
    public void RemoveContact(Contact contact)
    {
        // Usuwanie kontaktu z listy
        contacts.Remove(contact);
    }
}
```

Powyższy kod pokazuje zachowania `ContactStore`, które pozwalają programowi dodawać kontakty do pamięci i usuwać je z niej. Pokazana poniżej metoda `StoreContact` otrzymuje referencję do kontaktu, który ma zostać zapisany, i dodaje go do listy. Metoda `RemoveContact` usuwa dany kontakt z listy.

```
// Tworzy nowy magazyn kontaktów
ContactStore store = new ContactStore();
// Tworzy nowy kontakt
Contact rob = new Contact(name: "Rob", address: "Rob's house", phone: "0000 11111
2222");
// Umieszcza kontakt w magazynie
store.StoreContact(contact: rob);
```

ANALIZA KODU

Odpowiedzialności klas

Pytanie: W klasie `ContactStore` nie ma żadnych poleceń pozwalających programiście edytować zawartość kontaktu. Czy to jest poprawne?

> **Odpowiedź:** Tak. Chodzi o właściwy podział odpowiedzialności. Klasa `ContactStore` nie ponosi odpowiedzialności za dane w kontaktach; opiekowanie się wszystkimi danymi dotyczącymi kontaktu w programie jest zadaniem obiektu `Contact`.

Pytanie: Jak trudno byłoby zmodyfikować klasę `ContactStore`, aby mogła współpracować z innymi rodzajami danych?

> **Odpowiedź:** Ze względu na nasz rozsądny projekt, byłoby to bardzo łatwe. Wystarczyłoby zmienić typ listy oraz parametry metod `StoreContact` i `RemoveContact` — podstawowe zachowania byłyby dokładnie takie same.

Tworzenie kontaktów testowych

Bardzo zależy mi, abyśmy mogli przetestować nasz program bez konieczności wpisywania całego mnóstwa kontaktów. W tym celu dodałem do klasy `ContactStore` metodę, która utworzy magazyn kontaktów wypełniony już kontaktami.

```
public class ContactStore
{
    // Statyczna metoda, która zwraca testowy magazyn kontaktów
    public static ContactStore GetTestStore()
    {
        // To jest obiekt ContactStore, który będzie przechowywał wynik
        ContactStore result = new ContactStore();

        // Tablica testowych łańcuchów znaków nazw
        string[] testNames = {
            "Rob", "Marysia", "Dawid", "Joanna", "Krzyś"
            "Szymon", "Karol", "Helena", "Nela",
            "Amanda", "Sandea", "Renata", "Roman" };

        // Przechodzi przez każdą nazwę na liście
        foreach (string name in testNames)
        {
            // Tworzy testowy kontakt z tej nazwy
            Contact newContact = new Contact(name: name,
                address: name + " dom",
                phone: name + " dom");
            // Dodaje kontakt do wyniku
            result contacts Add(newContact);
        }

        // Zwraca nowy magazyn kontaktów
        return result;
    }
}
```

Wyszukiwanie kontaktów

Mamy teraz klasę `ContactStore`, której możemy używać do przechowywania kontaktów, ale nie mamy jeszcze sposobu, aby nasza znajoma prawniczka mogła wyszukiwać kontakt, z którym potrzebuje pracować.

Pamiętaj, że prawniczka chce wpisywać część nazwy — na przykład *Ro* — a program ma wtedy wyświetlać listę wszystkich kontaktów, które zawierają *Ro* w nazwie, aby mogła następnie wybrać spomiędzy *Robert*, *Roman*, *Roksana* i tak dalej. Oznacza to, że metoda `Find` nie może znajdować tylko jednego kontaktu; jako rezultat swojego wyszukiwania musi zwracać kolekcję. Czegoś takiego jeszcze nie robiliśmy. Do tej pory nasze metody zwracały pojedyncze elementy, ale nie ma powodu, dla którego metoda nie mogłaby zwracać listy lub tablicy.

```
public class ContactStore
{
    // Zwraca listę kontaktów, których nazwa zawiera wyszukiwaną nazwę
    public List<Contact> FindContactsWithName(string searchName)
    {
        // Konwertuje wyszukiwaną nazwę na wielkie litery
        searchName = searchName.ToUpper();

        // Tworzy listę kontaktów, która zostanie zwrócona
        List<Contact> result = new List<Contact>();

        // Wykonuje pętlę przez wszystkie kontakty w magazynie
        foreach (Contact contact in contacts)
        {
            // Tworzy wersję nazwy zapisaną wielkimi literami
            string contactName = contact.Name.ToUpper();
            // Sprawdza, czy nazwa zawiera wyszukiwany łańcuch znaków
            if (contactName.Contains(searchName))
            {
                // Dodaje kontakt do listy, jeśli znaleziono dopasowanie
                result Add(contact);
            }
        }
        // Zwraca listę kontaktów z pasującymi nazwami
        return result;
    }
}
```

Metodę `FindContactsWithName` można potraktować jako rodzaj filtra. Przyjmuje ona listę wszystkich kontaktów i tworzy nową listę, zawierającą tylko kontakty pasujące do kryteriów wyszukiwania.

ANALIZA KODU

Metoda FindContactsWithName

Pytanie: Co się stanie, jeśli w kontaktach nie zostanie znaleziony szukany łańcuch znaków?

Odpowiedź: W takim przypadku wynikiem będzie lista, która nie zawiera żadnych elementów, co jest w porządku. Oznaczałoby to, że na liście nie ma żadnego kontaktu z pasującymi znakami w nazwie. Jeśli wyszukiwany łańcuch znaków pojawi się w nazwie każdego kontaktu, wynikiem będzie kopia wszystkich kontaktów z magazynu danych, co też jest w porządku.

Pytanie: Do czego służą te wszystkie metody `ToUpper()`?

Odpowiedź: Widzieliśmy już wcześniej tę metodę. Zwraca ona wersję łańcucha znaków zapisaną wielkimi literami. Innymi słowy, z łańcucha znaków *Rob* otrzymalibyśmy *ROB*. Bardzo ważne jest, żeby porównania łańcuchów znaków były wykonywane przy użyciu znaków tej samej wielkości, ponieważ prawniczka złoży reklamację, jeśli będzie wyszukiwać *Rob*, a program nie znajdzie tego kontaktu, zapisanego w programie jako *rob*.

Pytanie: Czy tworzenie nowej listy przy każdym wyszukiwaniu nie jest mało wydajne?

Odpowiedź: Niezupełnie. Pamiętaj, że lista zawiera referencje do kontaktów, a nie same kontakty, a referencje to w rzeczywistości bardzo małe ilości danych. Biblioteki stanowiące bazę naszych programów bardzo dobrze radzą sobie z tworzeniem list.

Wyświetlanie listy znalezionych kontaktów

Metoda `FindContactsWithName` zwraca listę kontaktów, którą prawniczka życzy sobie przeglądać. Teraz musimy wyświetlić tę listę, aby mogła wybrać kontakt, z którym chce pracować. W tym celu możemy użyć niezwykle potężnej funkcjonalności XAML, a mianowicie **wiązania danych**. Wiązanie danych robi to, co sugeruje ten termin — tworzy połączenie między fragmentem danych (w tym przypadku listą kontaktów) i elementem wyświetlacza.

Tworzenie szablonu wiązania danych

Chcielibyśmy powiązać `Contact` z elementem wyświetlacza, ale nie możemy powiązać go bezpośrednio, ponieważ wartość kontaktu obejmuje kilka różnych komponentów. Wartość obiektu `Contact` zawiera wartości `Name`, `Address` i `PhoneNumber`. Potrzebujemy więc zaprojektować szablon, który będzie określał sposób wyświetlania kontaktu. Pozwoli nam to wybierać, które części kontaktu chcemy wyświetlać na ekranie (możemy pominąć na przykład numer telefonu), oraz określać sposób formatowania danych. Może to być coś takiego:

```
<DataTemplate>
    <StackPanel Margin="4">
        <TextBlock Text="{Binding Name}"/>
        <TextBlock Text="{Binding Address}"/>
    </StackPanel>
</DataTemplate>
```

Szablon danych to fragment kodu XAML, który opisuje sposób wyświetlania pewnych danych. Tutaj wskazujemy, że chcemy wyświetlać dane w postaci elementu `StackPanel`, który zawiera dwa elementy: nazwę i adres. Powinno to spowodować wyświetlenie czegoś takiego, jak pokazano poniżej, z dwoma łańcuchami znaków ułożonymi jeden pod drugim:

```
Rob Miles
Rob Miles dom
```

Zwróć uwagę, że na liście wyszukiwania chcemy wyświetlać tylko nazwę i adres kontaktu. Nie ma wiązania danych dla wartości numeru telefonu, więc nie będzie ona wyświetlana.

Użycie elementu DataTemplate w elemencie ListBox

Szablonu danych używamy do wskazania, jak mają być wyświetlane właściwości obiektu. Zamierzamy użyć tego szablonu, aby wyświetlić każdy z kontaktów znajdujących się na liście. W tym celu umieszczamy element `DataTemplate` wewnątrz elementu `ItemTemplate` należącego do `ListBox`, który wyświetli nasze kontakty:

```
<ListBox Name="ContactListBox" Margin="4" Height="300">
    <ListBox ItemTemplate>
        <DataTemplate>
            <StackPanel Margin="4">
                <TextBlock Text="{Binding Name}"/>
                <TextBlock Text="{Binding Address}"/>
            </StackPanel>
        </DataTemplate>
    </ListBox ItemTemplate>
</ListBox>
```

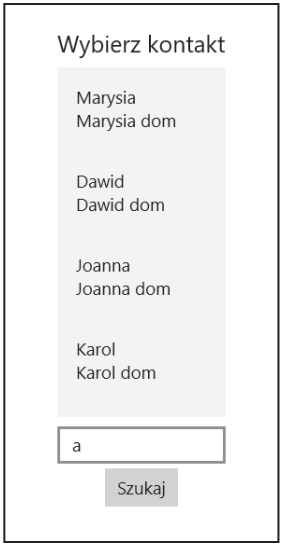
Czas się trochę cofnąć. Element wyświetlacza `ListBox` o nazwie `ContactListBox` ma wyświetlać listę kontaktów zwracanych przez `FindContactsWithName`. Aby to zrobić, musi wiedzieć, jak wyświetlić pojedynczy kontakt. `ListBox` używa elementu `ItemTemplate`, aby uzyskać projekt elementu listy; w tym przypadku będzie używał do tego celu elementu `DataTemplate`.

Gdy przygotujemy już ten XAML, `ListBox` musi zostać po prostu poinformowany, jaką kolekcję danych powinien wyświetlić:

```
// Procedura obsługi zdarzeń uruchamiana po kliknięciu przycisku wyszukiwania
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    // Pobiera łańcuch znaków wyszukiwania z pola testowego wyszukiwania
    string searchName = searchTextBox.Text;

    // Pobiera listę pasujących kontaktów
    List<Contact> foundList = contacts.FindContactsWithName(searchName);

    // Wyświetla listę
    ContactListBox.ItemsSource = foundList;
}

Demo 18-01 Finding Contacts
```

To jest punkt wiązania dla listy.

Ten kod jest uruchamiany, gdy użytkownik naciśnie przycisk *Szukaj* na wyświetlaczu. Metoda `FindContactsWithName` zwraca listę kontaktów do wyświetlenia, a następnie wyświetla ją w `ContactListBox`. `ListBox` wyświetli każdy obiekt z kolekcji jako element na liście. Rysunek 18.3 pokazuje, jak to działa. Prawniczka użyła do wyszukiwania łańcucha znaków *a*, a program wyświetlił każdy kontakt z literą *a* w nazwie.



Rysunek 18.3. Filtr wyszukiwania

Aby wygenerować ten wyświetlacz, nasz program nie musi robić niczego poza ustawieniem właściwości `ItemsSource` obiektu `ListBox`. Cała reszta jest wykonywana przez wyświetlacz. `ListBox` zapewnia nawet automatycznie paski przewijania; jeśli lista jest za długa, żeby zmieścić się na ekranie, użytkownik może przewijać ją w górę i w dół.

ANALIZA KODU

ListBox i ItemSource

Pytanie: Jak to działa? Co robi system Windows, aby wyświetlić te elementy?

> **Odpowiedź:** Jeśli uruchomisz przykładowy kod dla tego programu, będzie to prawie jak magia. Wpisujesz jakieś litery, naciskasz przycisk *Szukaj*, a lista kontaktów magicznie zapełnia się pasującymi kontaktami. Jak to naprawdę działa? Kluczem jest to przypisanie:

```
ContactListBox ItemsSource = foundList;
```

Po prawej stronie przypisania znajduje się kolekcja, która zawiera referencje do wszystkich pasujących kontaktów. Po lewej stronie znajduje się właściwość `ItemsSource` obiektu `ListBox`, który pokaże listę kontaktów. Właściwość `ItemsSource` oczekuje, że otrzyma kolekcję rzeczy do pokazania. Gdy `ContactListBox` otrzymuje jakąś kolekcję, przechodzi przez jej poszczególne elementy, dodając je do listy, która ma być wyświetlona. Wykorzystuje element `DataTemplate` elementu `ListBox`, aby określić, co należy wyświetlić na ekranie. Elementy w `DataTemplate` są dopasowywane do właściwości obiektów dodawanych do listy, a następnie wyświetlane. W tym przypadku lokalizowane są właściwości `Name` i `Addres`, które są wyświetlane.

Mylące może być to, że `ContactListBox` nigdy nie jest wyraźnie informowany, że wyświetla wartość `Contact`. `ContactListBox` otrzymuje coś, co akurat zawiera właściwości `Name` i `Address`, które są następnie wyświetlane zgodnie z szablonem danych z XAML.

Osiąga się to poprzez zastosowanie technologii języka C# zwanej **refleksją** (ang. *reflection*). Refleksja pozwala programowi zapytać obiekt: „Jakie masz właściwości?". Obiekt odpowiada wtedy listą właściwości, a te mogą być dopasowywane do wiązań w szablonie.

Oznacza to, że używany przez nas szablon elementu `ListBox` będzie działał z każdą listą zawierającą jakieś elementy, które mają właściwości `Name` i `Address`, a nie tylko z obiektami `Contact`.

Pytanie: Co się stanie, jeśli szablon będzie odwoływał się do właściwości, które nie istnieją na liście obiektów?

> **Odpowiedź:** Szablon elementu `ListBox` instruuje go, co wyświetlać dla każdego elementu na liście:

```
<DataTemplate>
    <StackPanel Margin="4">
        <TextBlock Text="{Binding Name}"/>
        <TextBlock Text="{Binding Address}"/>
    </StackPanel>
</DataTemplate>
```

Gdy wyświetlana jest lista, proces refleksji dopasowuje szablon, wiążąc nazwy z właściwościami w obiektach. Jeśli jednak użyjemy `Binding Namey` — czyli popełnimy błąd w szablonie — program będzie działał zupełnie dobrze; po prostu nie uda mu się wyświetlić niczego dla `Name`. Może to być źródłem irytujących błędów, więc jeśli coś nie jest wyświetlane poprawnie, sprawdź, czy nazwy szablonów są prawidłowe.

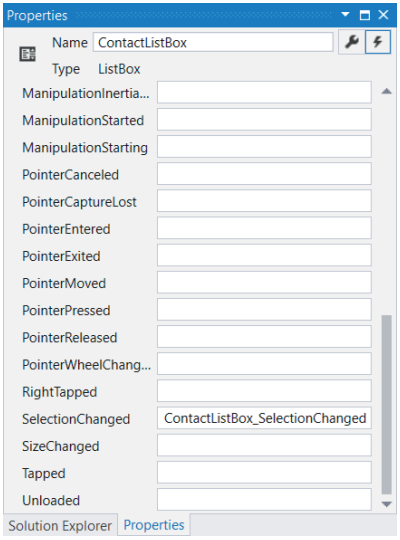
Pytanie: Jak sprawić, by lista wyglądała ładniej?

Odpowiedź: Możesz uatrakcyjnić wyświetlanie listy, używając różnych kolorów i rozmiarów tekstu dla elementów nazwy i adresu pola `TextBox`. Możesz również dodać do szablonu wszelkie inne elementy stylizujące, na przykład obrazy i kształty.

Edycja kontaktu

Teraz możemy bardzo łatwo wyświetlić listę elementów. Następną rzeczą, którą chcemy zrobić, jest utworzenie aplikacji do edycji danych. Gdy prawniczka wybierze jeden z elementów na liście, chcemy wyświetlić ekran umożliwiający edycję tego elementu. Aby to zadziałało, musimy wykrywać, kiedy zmienia się wybrany element na liście. W tym celu możemy użyć kodu bardzo podobnego do tego, który napisaliśmy dla `ComboBox` — służył on do wybierania quizów w poprzednim rozdziale. Nasz program będzie mógł wykrywać, kiedy zmieni się wybrany element w `ListBox`, dzięki podłączeniu pewnego kodu do zdarzenia, które jest generowane w przypadku wystąpienia zmiany.

Okno *Properties* dla `ContactListBox` pokazane na rysunku 18.4 ma zdarzenie `Selection-Changed` powiązane z metodą, która jest uruchamiana, gdy użytkownik wybierze jeden z elementów z `ListBox`.



Rysunek 18.4. Procedura obsługi zmiany wyboru dla ContactListBox

Oto kod tej metody:

```
// Procedura obsługi zdarzeń uruchamiana, gdy użytkownik zmieni wybór na liście kontaktów
private void ContactListBox_SelectionChanged(object sender,
     SelectionChangedEventArgs e)
{
    // Jeśli zmiana wymaga odznaczenia elementu, po prostu wraca z wykonywania
    if (ContactListBox.SelectedItem == null)
        return;

    // Pobiera wybrany element jako kontakt
    Contact contact = (Contact) ContactListBox.SelectedItem;
    // Ładuje wybrany kontakt do edytora
    selectContactForEdit(contact);
}
```

ANALIZA KODU

Wybieranie elementów

Pytanie: Dlaczego musimy sprawdzić, czy `SelectedItem` ma wartość `null`?

Odpowiedź: Zastanów się, co się dzieje podczas używania `ListBox`. Istnieją dwie sytuacje, w których wybrany element może się zmienić. Pierwsza jest oczywista — użytkownik wybiera

element z listy. Gdy to zrobi, uruchomi się procedura obsługi zdarzeń i element `SelectedItem` w `ListBox` będzie odwoływał się do wybranego elementu.

Drugim kontekstem, w którym wybrany element może ulec zmianie, jest ładowanie `ListBox` z nową listą do wyświetlenia. Kiedy tak się dzieje, nic nie jest zaznaczone, ponieważ na tym etapie użytkownik niczego jeszcze nie wybrał. W tym momencie referencja `SelectedItem` ma wartość `null`, którą referencja może mieć wtedy, gdy nie odwołuje się do niczego przydatnego. Ponieważ zmiana na `null` to też zmiana, uruchamia się metoda `SelectionChanged`.

Gdy użytkownik wybiera kontakt, program wywołuje metodę `selectContactForEdit`. Następnym krokiem jest dodanie do tej metody zachowania, które pozwoli prawniczce edytować wybrany kontakt.

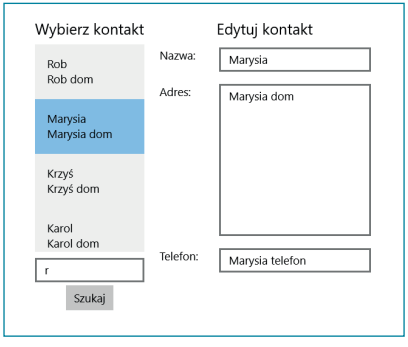
Z PUNKTU WIDZENIA PROGRAMISTY

Mechanizm wyboru ListBox jest bardzo wszechstronny

Właśnie nauczyłeś się bardzo ważnej zasady dotyczącej projektowania interfejsu użytkownika. Wiesz jak wyświetlić listę elementów i umożliwić użytkownikowi wybieranie kontaktów z tej listy. Zobaczysz to zachowanie w wielu aplikacjach, od mediów społecznościowych po odtwarzacze muzyczne. `ListBox` możesz użyć do wyświetlenia dowolnego rodzaju elementu, który można wyrazić za pomocą XAML, możesz ponadto wiązać dowolne właściwości elementów w szablonie z elementami Twoich danych.

Edytor kontaktów

Rozmawiałeś z prawniczką o projektowaniu interfejsu użytkownika — chociaż w tym przypadku nazwanie tego rozmową byłoby eufemizmem. Dyskusja stała się gorąca, co nie jest zaskakujące. Z mojego doświadczenia wynika, że użytkownicy mają wyrobione zdanie na temat tego, jak powinny działać ich programy, a programiści nie zawsze podzielają te opinie. Udało Ci się jednak uzgodnić projekt interfejsu użytkownika, pokazany na rysunku 18.5.



Rysunek 18.5. Interfejs użytkownika służący do edycji kontaktu

Kiedy prawniczka wybierze kontakt, zostanie on wyświetlony po prawej stronie okna aplikacji. Prawniczka może edytować kontakt, a jeśli przejdzie do innego kontaktu, zmiany zostaną automatycznie zapisane. Moim zdaniem słusznie zwróciłeś uwagę na to, że jeśli użytkownik programu spróbuje opuścić kontakt, który edytował, program powinien wyświetlić monit, taki jak: „Czy chcesz opuścić edycję tego elementu? Wprowadziłeś zmiany. Czy chcesz je zachować?". Jednak prawniczce nie podoba się ten pomysł; mówi, że zawsze poprawnie wprowadza zmiany i może je łatwo wycofać w razie potrzeby. Prawniczka nie chce być nękana komunikatami ostrzegawczymi, więc poprosiła Cię, abyś ich nie wyświetlał.

Z PUNKTU WIDZENIA PROGRAMISTY

Klient nie zawsze ma rację, ale zawsze to klient

Przeprowadzałem tego typu dyskusje z klientami. Projektowałem coś, co uważałem za idealne interfejsy, tylko po to, żeby usłyszeć: „Nie tak to powinno działać". To boli. Jeśli znajdziesz się w takiej sytuacji, sądzę, że powinieneś pamiętać o trzech rzeczach.

Po pierwsze, nigdy nie dopuszczaj do sytuacji, w której pokażesz klientowi coś, co zbudowałeś, a on stwierdzi, że mu się to nie podoba. Naprawdę ważne rzeczy, takie jak interfejs użytkownika, powinny być projektowane we współpracy z klientem. Interfejsy powinny być produktem debaty, a nie prezentacji.

Po drugie, z etycznego punktu widzenia, jeśli uważasz, że coś jest niepoprawne w niebezpieczny sposób, powinieneś wskazać to klientowi i formalnie poprosić go o wzięcie odpowiedzialności za dane zachowanie. Ten program nie należy do tej kategorii, ale jeśli piszesz aplikację o przeznaczeniu medycznym, naprawdę musisz wskazać wszelkie błędy projektowe, które Twoim zdaniem mogą powodować wyświetlanie nieprawidłowych danych.

Wreszcie, nawet jeśli klient się myli, nadal jest klientem. Płaci Ci, żebyś zrobił to, czego potrzebuje. W ostatecznym rozrachunku chcesz, aby był zadowolony z tego, co zrobiłeś. A czasem, jak podpowiada mi moje doświadczenie, klient może mieć rację.

Po wybraniu kontaktu panel edycji jest wypełniany szczegółowymi informacjami o kontakcie, które możesz wykorzystać lub edytować. Oto kod dla metody `selectContactForEdit`:

```
// To jest aktualnie wybrany kontakt, początkowo null
Contact selectedContact = null;

// Wybiera konkretny kontakt do edycji
private void selectContactForEdit(Contact contactToEdit)
{
    // Sprawdza, czy aktualnie edytujemy kontakt
    if (selectedContact != null)
```

Wybiera inny kontakt do edycji.

```
    {
        // Zamierzamy opuścić kontakt — zapisz go
        saveContactFromPage(selectedContact);
    }
    // Wyświetla kontakt, do którego przechodzimy
    placeContactOnPage(contactToEdit);
    // Zapamiętuje wybrany kontakt, żeby można go było zapisać później
    selectedContact = contactToEdit;
}
```

Ważne jest zrozumienie kontekstu, w którym uruchamiana jest metoda `selectContactForEdit`:

1. Użytkownik szuka kontaktu (wpisuje na przykład *Ro* w polu wyszukiwania i naciska przycisk *Szukaj*). Lista wypełnia się kontaktami, które zawierają w nazwie szukany łańcuch znaków.
2. Użytkownik wybiera kontakt w elemencie `ListBox` (klika *Rob* na wyświetlonej liście kontaktów).
3. Uruchamiana jest metoda `selectionChanged`, ponieważ jest ona połączona ze zdarzeniem zmiany wyboru w elemencie `ListBox`, a my właśnie wybraliśmy inny kontakt.
4. Metoda `selectionChanged` wywołuje następnie metodę `selectContactForEdit`, aby przygotować się do edytowania.

Metoda `selectContactForEdit` musi wykonać dwa zadania. Musi zapisać aktualnie edytowany kontakt (jeśli jakiś jest edytowany), a następnie musi wybrać nowy kontakt do edytowania. Zmienna `selectedContact` jest używana przez program do śledzenia aktualnie edytowanego kontaktu. Po uruchomieniu programu ta zmienna jest ustawiana na `null`, co wskazuje, że żaden kontakt nie jest edytowany.

Następnie mamy metody służące do przenoszenia danych między kontaktem a elementami `TextBox` na stronie, którą widzi użytkownik. Obie te metody otrzymują kontakt jako parametr. Pierwsza z metod przenosi informacje kontaktowe z kontaktu na ekran (`placeContactOnPage`), a druga przenosi informacje kontaktowe ze strony z powrotem do kontaktu (`saveContactFromPage`).

```
// Umieszcza kontakt na stronie i przygotowuje go do edycji
private void placeContactOnPage(Contact source)
{
    // Kopiuje dane ze źródłowego kontaktu do elementów wyświetlacza
    NameTextBox.Text = source.Name;
```

```
    AddressTextBox.Text = source.Address;
    PhoneTextBox.Text = source.Phone;
}
// Ładuje dane kontaktu z elementów wyświetlacz do docelowego kontaktu
private void saveContactFromPage(Contact destination)
{
    // Kopiuje element wyświetlacza do docelowego kontaktu
    destination.Name = NameTextBox.Text;
    destination.Address = AddressTextBox.Text;
    destination.Phone = PhoneTextBox.Text;
}

Demo 18-02 Contact Editor
```

ANALIZA KODU

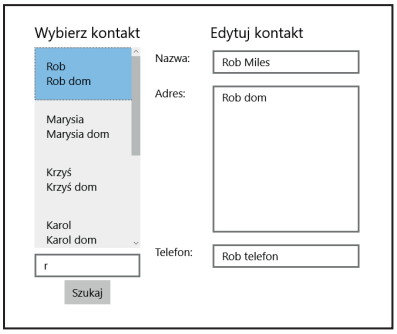
Edytowanie kontaktów

Pytanie: Czy w tym procesie jest jakakolwiek walidacja danych?

Odpowiedź: Te dwie metody, `placeContactOnPage` i `saveContactFromPage`, nie przeprowadzają żadnej walidacji danych. Jeśli prawniczka pozostawi puste którekolwiek pole `TextBox` lub wprowadzi nieprawidłowe informacje, program nie wyświetli żadnych błędów. Prawniczce to odpowiada, ale jeśli Twoja aplikacja musi upewnić się, że przechowywane dane są prawidłowe, rozsądnie byłoby wstawić tutaj pewną walidację.

Wiązanie danych w interfejsie użytkownika

Jeśli uruchomisz ten edytor kontaktów, który do tej pory napisaliśmy, przekonasz się, że działa całkiem dobrze. Możesz wybrać kontakt, edytować go, przejść do innego kontaktu, a następnie wrócić do pierwotnego. Przekonasz się, że Twoje zmiany są zapisywane w kontaktach, ale interfejs użytkownika nie jest tak dobry, jak mógłby być. Rysunek 18.6 pokazuje problem.



Rysunek 18.6. Wyświetlanie aktualizacji

Zmieniłem tutaj nazwę pierwszego kontaktu na Rob Miles, ale wpis na liście wyboru po lewej stronie nie zmienił się, aby to odzwierciedlić. Jesteśmy raczej pewni, że nasza klientka prawniczka zwróci na to uwagę i nie będzie zadowolona. Musimy to więc naprawić.

Obiekty obserwowalne

Aby usunąć problem, musimy sprawić, by system wyświetlania był „świadomy" zmian w danych, które muszą być aktualizowane. System wyświetlania listy musi zostać poinformowany o zmianie nazwy, aby mógł zaktualizować `ListBox`. W XAML robimy to poprzez ustawienie obiektów jako „obserwowalnych".

W kontekście XAML obserwowalny oznacza: „Mogę poprosić Cię o poinformowanie mnie, kiedy zostaniesz zmieniony". Większość obiektów C# nie jest obserwowalna, ponieważ nie ma żadnego podmiotu, który chciałby wiedzieć, czy ich wartość się zmieniła. Lista kontaktów wyświetlanych przez `ContactListBox` nie jest obserwowalna. Jest to w porządku, jeśli chcemy jedynie przeglądać zawartość listy, ale jak widziałeś, w przypadku zmiany któregokolwiek elementu z listy, wyświetlacz nie ma możliwości, by to wykryć i dokonać aktualizacji. Potrzebujemy bardziej wyspecjalizowanej formy kolekcji, która może być „obserwowana" przez system wyświetlania, aby można było odzwierciedlać zmiany w kolekcji na wyświetlaczu.

Klasa ObservableCollection

Właśnie po to stworzona została klasa `ObservableCollection`. Może ona służyć do przechowywania kolekcji elementów i zapewnia obsługę powiadomień, dzięki czemu w przypadku zmiany zawartości kolekcji, system wyświetlania może zostać o tym powiadomiony. Bardzo łatwo możemy utworzyć obserwowalną kolekcję kontaktów:

```
// Obserwowalna wersja listy kontaktów
ObservableCollection<Contact> contactList;
```

```
// Procedura obsługi zdarzeń dla przycisku wyszukiwania
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    // Pobiera nazwę do wyszukania z pola wyszukiwania
    string searchName = searchTextBox Text;

    // Pobiera listę pasujących kontaktów
    List<Contact> foundList = contacts FindContactsWithName(searchName);

    // Używa pobranej listy do utworzenia obserwowalnej kolekcji
    contactList = new ObservableCollection<Contact>(foundList);

    // Łączy kolekcję z polem kontaktu w celu jego wyświetlania
    ContactListBox ItemsSource = contactList;
    // Czyści wyświetlacz edytora
    clearContactEdit();
}
```

ANALIZA KODU

ObservableCollection

Pytanie: Jaka jest różnica między `ObservableCollection` a `List`?

Odpowiedź: Z punktu widzenia użytkownika nie ma żadnej różnicy między tymi dwiema klasami kolekcji. Działają w ten sam sposób. Różnica polega na tym, że system zarządzania platformą Windows może łączyć się ze zdarzeniami generowanymi przez `ObservableCollection`, gdy zawartość listy ulega zmianie. Wskazówka tkwi w nazwie. Pierwsza z tych klas może być obserwowana, a druga nie.

Pytanie: Dlaczego nie czynimy wszystkich kolekcji obserwowalnymi?

Odpowiedź: Moglibyśmy używać klasy `ObservableCollection` do wszystkich przechowywanych danych, ale prawdopodobnie spowolniłoby to nasze programy. Kolekcja obserwowalna musi sprawdzać, czy ktoś nie jest przypadkiem zainteresowany jakimikolwiek zmianami, które wprowadza w zarządzanych przez siebie danych. Spowalnia to nieco działanie kolekcji i dlatego ten typ jest zarezerwowany dla elementów, które wyświetlamy.

Niestety, jeśli dodamy nową wersję procedury obsługi zdarzeń przycisku *Szukaj*, pokazaną w poprzednim kodzie, program nadal nie odzwierciedla pożądanych zmian. Jeżeli użytkownik zaktualizuje nazwę w kontakcie, nie znajdzie to odzwierciedlenia w tekście na liście. Dzieje się tak, ponieważ `ObservableCollection` reaguje na zmiany w zawartości listy, a nie na zmiany danych w elemencie z listy. Gdybyśmy dodali lub usunęli kontakt, lista by się zmieniła, ale zmiana zawartości elementu znajdującego się liście nie jest wykrywana. Innymi słowy, lista nie ma obecnie możliwości uzyskania informacji o zmianach wprowadzanych w przechowywanych w niej danych.

Kontakty obserwowalne

Każdy z elementów wyświetlacza XAML, które składają się na interfejs użytkownika aplikacji *Rejestr czasu pracy*, wyświetla widok obiektu danych z naszego programu. Edytor utworzył ten widok przez ustawianie wartości właściwości w elementach wyświetlacza.

Pokazana poniżej metoda `placeContactOnPage` przyjmuje informacje o nazwie, adresie i numerze telefonu z kontaktu, do którego odwołuje się jako `source`, i ustawia właściwości `Text` odpowiednich wartości `TextBox`, aby wyświetlić dane kontaktowe. Szkopuł w tym, że jest to operacja jednorazowa. Jeśli zmieni się cokolwiek w obiekcie `Contact`, wyświetlacz nie zostanie zaktualizowany, ponieważ w tej chwili klasa `Contact` nie jest obserwowalna. Musimy nadać klasie `Contact` pewne zachowania, które umożliwiają powiadamianie innych obiektów (w tym przypadku systemu wyświetlania) o zmianach właściwości w obiekcie `Contact`.

```
// umieszcza kontakt na stronie
private void placeContactOnPage(Contact source)
{
    // Kopiuje informacje o kontakcie z kontaktu do komponentów wyświetlacza
    NameTextBox.Text = source.Name;
    AddressTextBox.Text = source.Address;
    PhoneTextBox.Text = source.Phone;
}
```

Środowisko XAML ma protokół, za pomocą którego odbywa się to powiadamianie. Nazywa się on `INotifyPropertyChanged`. W przypadku używania interfejsu `INotifyPropertyChanged` obiekt zawiera „listę osób do powiadamiania", jeśli coś zmieni się w obiekcie. Przypomina to trochę planowanie przyjęcia. Masz listę zaproszonych osób. Jeśli potrzebujesz zmienić miejsce lub czas rozpoczęcia imprezy (innymi słowy, jeśli zmieni się którakolwiek z „właściwości" Twojej imprezy), przeglądasz tę listę, dzwonisz do umieszczonych na niej osób i mówisz im na przykład, że impreza zaczyna się teraz o 20:00, a nie o 20:30.

W naszym obiekcie „lista osób do powiadamiania" to lista **obiektów delegowanych**. Delegat to funkcjonalność języka C#, której jeszcze nie poznałeś. Jest ona bardzo wszechstronna. Delegata możesz potraktować jako obiekt zawierający referencje do metody z jakiegoś obiektu. Jeżeli delegat zostanie „wywołany", wywołana zostanie również metoda, do której delegat się odwołuje.

ANALIZA KODU

Delegaty

Pytanie: Gdzie mogliśmy już widzieć delegaty?

> **Odpowiedź:** Widzieliśmy delegaty w akcji, ale sami ich nie tworzyliśmy. Delegaty zapewniają mechanizm, za pomocą którego przycisk XAML jest podłączony do metody w naszym programie. Przycisk *Szukaj* w programie Rejestr czasu pracy „wie", którą metodę wywołać, gdy zostanie kliknięty, ponieważ otrzymał delegata, który odwołuje się do tej metody. Rozmawialiśmy o tym, że zdarzenie jest tylko wywołaniem metody — delegowanie umożliwia jednemu obiektowi (odbiorcy) przekazywanie delegata drugiemu obiektowi (przekaźnikowi), aby w razie zdarzenia przekaźnik mógł wywołać daną metodę.

Delegata możesz potraktować jako „wizytówkę" metody. Jeśli jakiś inny obiekt ma wizytówkę, może wywołać tę metodę. Po uruchomieniu aplikacji *Rejestr czasu pracy* system XAML tworzy delegata dla metody `SearchButton _ Click`, a następnie przekazuje go obiektowi `SearchButton`. W ten sposób wywoływana jest nasza metoda po kliknięciu przycisku. Gdy używane są powiadomienia o zmianach właściwości, system wyświetlania daje „wizytówkę" elementowi danych, aby ten wiedział, jaką metodę wywołać, jeśli zmienią się przechowywane w nim dane.

Delegaty zapewniają sposób wiązania obiektów między sobą i tworzenia ścieżek dla komunikatów podczas działania programu.

Tak właśnie wygląda „lista osób do powiadamiania" w obiekcie, który chce stać się obserwowalny. Typ `event` to typ `delegate` wbudowany w C# specjalnie dla obsługi zdarzeń. Wszystkie elementy, które są zainteresowane zmianami w danym obiekcie — czyli obiekty, które chcą go obserwować — mogą dołączyć delegata do elementu `PropertyChanged` w klasie. My w rzeczywistości nigdy nie przypisujemy niczego do tej zmiennej, ale system wyświetlania — tak.

```
public event PropertyChangedEventHandler PropertyChanged;
```

Skoro wiesz już, jak utrzymywać „listę osób do powiadamiania", musimy wymyślić, jak faktycznie powiadamiać o tym, że wartość się zmieniła. Robimy to, wywołując delegata:

```
PropertyChanged(this, new PropertyChangedEventArgs("Address"));
```

Metoda, która jest połączona z delegatem `PropertyChanged`, przyjmuje dwa argumenty. Informacje te są wprowadzane do systemu wyświetlania, aby poinstruować wyświetlacz, że coś się zmieniło i należy to ponownie narysować. Pierwszym elementem danych jest referencja do rzeczywistego obiektu, zawierającego zmienianą właściwość. Drugim jest wartość `PropertyChangedEventArgs`, która jest ustawiana na nazwę zmienianej właściwości. Powyższy kod informuje obserwatora, że zmieniana jest właściwość `Address` kontaktu.

Chcemy, aby ten kod był uruchamiany za każdym razem, gdy zmieni się adres w kontakcie, więc umieszczamy go w zachowaniu `set` właściwości adresu.

```
// Klasa kontaktu, która implementuje metody interfejsu INotifyPropertyChanged
public class Contact :INotifyPropertyChanged
{
    // Prywatna wartość dla adresu
    private string address;

    // Punkt wiązania dla obiektów, które chcą otrzymywać powiadomienia, gdy ta właściwość się zmieni
    public event PropertyChangedEventHandler PropertyChanged;

    // Publiczna właściwość adresu
    public string Address
    {
        // Odczytuje właściwość
        get
        {
            // Jeśli właściwość została odczytana, zwraca po prostu wartość adresu
            return address;
        }

        // Ustawia właściwość
        set
        {
            // Zapisuje przychodzącą wartość adresu
            address = value;
            // Testuje, czy coś jest podłączone do zdarzenia zmiany właściwości
            if(PropertyChanged != null)
            {
                // Uruchamia zdarzenie zmiany właściwości
                PropertyChanged(this, new PropertyChangedEventArgs("Address"));
            }
        }
    }
}

Demo 18-03 Updating Contact Editor
```

ANALIZA KODU

INotifyPropertyChanged

Jest to prawdopodobnie najbardziej skomplikowany fragment kodu C#, jaki do tej pory widziałeś. Oczywiście masz w związku z tym parę pytań.

Pytanie: Powiedz mi jeszcze raz, dlaczego to robimy?

> **Odpowiedź:** Ten kod zapewnia sposób informowania systemu wyświetlania, kiedy dane w naszym obiekcie zmieniają się. System wyświetlania musi wiedzieć, kiedy wartość się zmienia, aby móc aktualizować widok danych. Jest to bardzo wszechstronna funkcjonalność. Oznacza to, że jeśli nasz program dokona zmiany w wyświetlanym kontakcie — na przykład przypisując nowy adres do kontaktu — wyświetlacz zostanie zaktualizowany automatycznie, a my nie musimy robić nic więcej.

Pytanie: Dlaczego `PropertyChanged` nie zapewnia nowej wartości właściwości?

> **Odpowiedź:** Wydaje się trochę głupie, że metoda `PropertyChanged` nie zapewnia nowej wartości właściwości. To trochę tak, jakbym zadzwonił do Ciebie i powiedział: „Zmieniła się godzina rozpoczęcia imprezy", a następnie zakończył połączenie. Oczywiście, gdybym tak zrobił, natychmiast byś do mnie oddzwonił i zapytał: „No dobra, a jaka jest nowa godzina?". Tak właśnie powinno się stać w przypadku systemu wyświetlacza. Zdarzenie `PropertyChanged` nie dostarcza nowej wartości; zamiast tego uruchamia system wyświetlania, żeby pobrał zaktualizowaną wartość i wyświetlił ją w pewnym momencie w przyszłości.

Pytanie: Co się stanie, jeśli źle podam nazwę właściwości?

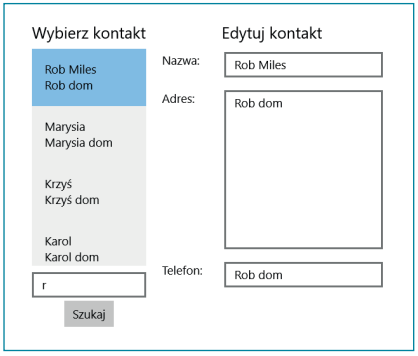
> **Odpowiedź:** Jeżeli program w parametrach zdarzenia `PropertyChanged` używa `Adress` zamiast `Address`, będzie działał bez błędu, ale do aktualizacji nie dojdzie.

Pytanie: Jak klasa może zaimplementować interfejs, który wydaje się nie zawierać żadnych metod?

> **Odpowiedź:** Klasa `Contact` musi implementować interfejs `INotifyPropertyChanged`, aby system wyświetlania wiedział, że można go obserwować. Interfejs to „lista zakupów" metod. Każda klasa implementująca interfejs musi zawierać metody, które on opisuje — z tym, że klasa `Contact` nie wydaje się implementować żadnych dodatkowych metoda, a jedynie zdarzenie `PropertyChanged`.

Interfejs może zawierać zarówno zdarzenia, jak i metody, co naprawdę ma sens. Jeśli używasz interfejsów do opisywania, jak można podłączać obiekty, powinna istnieć możliwość użycia do tego celu delegatów zdarzeń.

Jeżeli sprawimy, że wszystkie właściwości będą wysyłać odpowiednie powiadomienia, otrzymamy aplikację do edycji danych, która faktycznie poprawnie się aktualizuje, jak widać na rysunku 18.7.



Rysunek 18.7. Prawidłowe aktualizacje

Jeśli uruchomisz aplikację demo, znajdującą się w folderze *Demo 18-03 Updating Contact Editor*, i popracujesz z nią trochę, zauważysz, że tekst w polu `ListBox` nie aktualizuje się, dopóki użytkownik nie przejdzie do innego elementu na liście. Wynika to z faktu, że zawartość obiektu `Contact` jest aktualizowana tylko w tym momencie i to jest akcja, która wyzwala aktualizację zawartości `ListBox`. Możliwe jest utworzenie wersji, która aktualizuje listę za każdym razem, gdy zmienia się tekst w kontakcie, ale myślę, że używanie jej byłoby raczej rozpraszające.

ANALIZA KODU

Obserwowalność

Pytanie: Dlaczego musieliśmy uczynić listę klasą `ObservableCollection`, skoro tak naprawdę wyświetlane muszą być zmiany danych na tej liście?

> **Odpowiedź:** Ta lista musi być obserwowalna, ponieważ wiązania dla „obserwowalności" są przekazywane z kontenera do obiektów w nim zawartych. System wyświetlania jest powiązany z elementem `ListBox`. Ale kiedy kontakty są dodawane do `ListBox`, każdy z nich jest wiązany z `ObservableCollection` w `ListBox`, dzięki czemu zmiany są propagowane poprawnie. Możesz potraktować to jako „łańcuch dowodzenia", który musi być stale połączony od góry do dołu.

Projektowanie oprogramowania i Rejestr czasu pracy

Twój młodszy brat jest pod wrażeniem programu *Rejestr czasu pracy* i planuje przekształcić go w „rejestr receptur na babeczki" dla swojego przyjaciela zajmującego się produkcją babeczek.

Przeczytał jednak w internecie kilka artykułów na temat tworzenia oprogramowania i uznał, że *Rejestr czasu pracy* nie jest zbyt dobrym programem, ponieważ „nie używa wzorca Model--Widok-WidokModel". Podejrzewasz, że tak naprawdę nie ma pojęcia, co to znaczy, ale równie dobrze możemy przyjrzeć się tej technologii, skoro już o niej wspomniał. Okazuje się, że ma rację, i istnieje wiele powodów, dla których warto zbadać wzorzec Model-Widok-WidokModel (ang. *Model-View-ViewModel* — MVVM).

Model-Widok-WidokModel

Program z graficznym interfejsem użytkownika może być bardzo trudny do przetestowania. Jedynym sposobem na sprawdzenie, czy wszystko na wyświetlaczu działa poprawnie, jest wpisywanie różnych rzeczy, naciskanie przycisków i obserwowanie, co się będzie działo. Możesz przetestować swoje programy raz czy dwa, płacąc młodszemu bratu za wykonanie sekwencji testowej, ale jeśli potrzebujesz dość często testować program, może okazać się to bardzo kosztowne. Testowanie interfejsów użytkownika to duży problem dla programistów, ponieważ usilnie dążą do tego, aby ich testy były całkowicie automatyczne. Nasz edytor kontaktów jest trudny do przetestowania dlatego, że zachowania edycji są wymieszane na stronie z elementami wyświetlacza i nie ma separacji między wyświetlaczem a procesami działającymi za nim. Aby przetestować ten system, musimy napisać kod, który komunikuje się z wieloma różnymi elementami w programie.

Z PUNKTU WIDZENIA PROGRAMISTY

Poznaj wzorce projektowe

Wzorzec projektowy jest sposobem na ustrukturyzowanie rozwiązania. Każda profesja ma swoje własne wzorce projektowe, wypracowane na podstawie doświadczenia. Jeśli na przykład malujesz pokój, jednym z „wzorców" będzie pomalowanie najpierw ściany, a dopiero potem ram drzwi i okien. Wzorce projektowe powstały na podstawie doświadczeń wielu programistów i zbudowanych projektów. Z profesjonalnego punktu widzenia powinieneś przyjrzeć się wzorcom projektowym w trakcie budowania swoich umiejętności programistycznych. Model-Widok-WidokModel to nie jedyny wzorzec, o którym powinieneś mieć jakieś pojęcie.

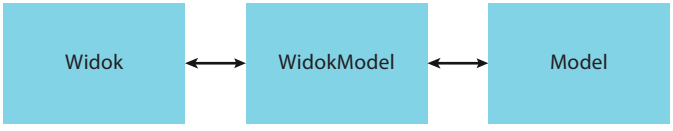
Jeszcze tylko kilka słów ostrzeżenia. Tak samo jak można znaleźć dekoratorów, którzy będą przysięgać, że powinieneś najpierw pomalować ramy drzwi i okien, można również znaleźć programistów z wyrobionymi różnymi opiniami na temat poszczególnych wzorców. Najlepsza rada, jaka przychodzi mi do głowy w tej sytuacji, jest taka, żebyś budował swoje doświadczenie i wiedzę, by móc wyrobić sobie własne zdanie na ten temat. I pamiętaj, że Twoja opinia jest co najmniej tak samo cenna, jak opinie innych osób.

Nasz program byłby o wiele łatwiejszy do przetestowania, gdybyśmy mieli jeden obiekt, który zajmuje się tylko procesem edycji. W tym celu zaprojektowany został wzorzec MVVM. Rozwiązanie zbudowane przy użyciu wzorca MVVM ma klasę model-widok, która zapewnia łącze między kodem XAML, dostarczającym widoku danych, oraz klasą `Contact` (w naszym programie), zapewniającą model danych, z którymi pracujemy.

Klasę model-widok możesz potraktować jako kelnera w bardzo eleganckiej restauracji. Gość (widok) siedzi przy stole i wybiera potrawy do zjedzenia, a kucharz (model) jest w kuchni i przygotowuje jedzenie. Gość poprosi kelnera (model-widok) o niektóre pozycje z menu, a kelner utworzy prośbę do kucharza o wydanie posiłku. Kiedy posiłek jest gotowy, kelner zabiera go z kuchni i serwuje gościowi. Gość nie ma pojęcia, jak przygotowuje się jedzenie; po prostu wie, co chce zjeść. Kelner zna polecenia, które rozumie kucharz, i przekłada na nie żądania gościa.

Ta struktura ma wiele zalet. Posiłki mogą zamawiać różni goście (widoki). Kucharza można zastąpić innym kucharzem, o ile nowy kucharz i kelner będą używali tych samych poleceń do komunikacji, a restauracja będzie dalej funkcjonować. Ogromną zaletą z punktu widzenia programisty jest to, że jednym z naszych widoków może być widok „testowy", który wysyła żądania, a następnie sprawdza odpowiedzi z modelu-widoku.

Rysunek 18.8 pokazuje sposób komunikowania się obiektów. Połączenie między widokiem a widokiem-modelem (gościem i kelnerem) uzyskuje się przez wiązanie danych. Widok jest stroną XAML, a widok-model zawiera cały kod, który faktycznie steruje interfejsem użytkownika. Zadaniem modelu jest zapewnienie dostępu do danych, którymi zarządza system.



Rysunek 18.8. Wzorzec Model-Widok-WidokModel

Wiązanie danych we wzorcu Model-Widok-WidokModel

We wcześniejszych wersjach menedżera kontaktów XAML nasz program w celu przeprowadzania edycji bezpośrednio pobierał dane z elementów wyświetlacza i zapisywał je w nich. Istnieje jednak o wiele łatwiejszy sposób, czyli użycia wiązania danych XAML do połączenia właściwości z klasy widok-model z elementami na ekranie.

```
<TextBox Name="NameTextBox" Width="200" Margin="4" Text="{Binding
Name,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"></TextBox>
```

Ten kod XAML opisuje powiązane pole `TextBox` o nazwie `NameTextBox`, w którym tekst znajdujący się w polu jest wiązany z właściwością `Name` z klasy widok-model. Zabrzmiało to w dość wyrafinowany sposób, prawda? Powtórzmy to jeszcze raz w zwolnionym tempie. Możemy zacząć od sposobu, w jaki do tej pory przeprowadzaliśmy edycję.

```
<TextBox Name="NameTextBox" Width="200" Margin="4" Text="Robert"></TextBox>
```

Powyższy XAML definiuje pole `TextBox` o nazwie `NameTextBox`, które zawiera ustalony tekst `"Robert"`. Po uruchomieniu, program wyświetla *Robert* w polu `TextBox`. Jest to doskonały sposób na wyświetlenie mojego imienia, ale nie to chcemy zrobić. Chcemy wyświetlić nazwę kontaktu, z którym pracuje prawniczka. Możemy operować na nazwie kontaktu, ładując ją do tego pola `TextBox`:

```
private void placeContactOnPage(Contact source)

{

    NameTextBox.Text = source.Name;

}
```

Używamy metody `placeContactOnPage`, aby kontakt był widoczny do edycji. Powyżej widać jej skróconą wersję. Instrukcja w metodzie umieszcza nazwę kontaktu źródłowego w polu `NameTextBox`, aby użytkownik mógł zobaczyć i edytować nazwę. Pod koniec edycji program musi odłożyć edytowany tekst z powrotem na miejsce, aby wszelkie wprowadzone zmiany zostały odzwierciedlone w edytowanym kontakcie.

```
private void saveContactFromPage(Contact destination)
{
    destination.Name = NameTextBox.Text;
}
Demo 18-02 Contact Editor
```

Po zakończeniu edycji wywołujemy metodę `saveContactFromPage`. Ta skrócona wersja zawiera instrukcję, która pod koniec edycji umieszcza edytowaną nazwę z powrotem w docelowym obiekcie `Contact`. Ten kod możesz zobaczyć w akcji w programie *Demo 18-02 Contact Editor*.

Musimy to zrobić dla wszystkich edytowanych elementów. Możemy jednak wyeliminować potrzebę pisania tego kodu, jeśli użyjemy wiązania danych w XAML. Określamy źródło właściwości XAML jako powiązane z właściwością w klasie widok-model.

To jest wersja `NameTextBox` z wiązaniem danych:

```
<TextBox Name="NameTextBox" Width="200" Margin="4"
Text="{Binding Name,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"></TextBox>
```

Wszystko w `TextBox` jest takie samo, z wyjątkiem tego, że tekst dla nazwy pochodzi z powiązanej właściwości z klasy widok-model. Zobaczymy tę właściwość `Name` w następnym punkcie rozdziału.

Zbadajmy szczegóły wiązania, przyglądając się kolejno poszczególnym elementom.

```
Text="{Binding Name,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"
```

Pierwszy element, `Binding`, identyfikuje właściwość w klasie, z którą dokonujemy wiązania. Moglibyśmy mieć również wiązanie z właściwością `Address` dla pola `TextBox`, które pozwala użytkownikowi edytować adres kontaktu.

Drugi element, `Mode`, określa *tryb* wiązania. Tryb może mieć wartość `OneTime` (jednorazowy), `OneWay` (jednokierunkowy) lub `TwoWay` (dwukierunkowy). Jeśli trybem jest `OneTime`, dane źródłowe są odczytywane raz, podczas początkowego wyświetlania pola `TextBox`. Jeśli dane w `TextBox` zostaną zaktualizowane przez program, nie zostanie to odzwierciedlone na ekranie.

Wiązanie `OneWay` oznacza, że kontrolka XAML zmieni się, jeśli zmieni się właściwość (innymi słowy, jeśli program zmieni zawartość właściwości `Name`, wyświetlacz zmieni się odpowiednio), ale jeżeli użytkownik wpisze coś w polu `TextBox`, ta zmiana nie zostanie odzwierciedlona w klasie widok-model. Nam potrzebny jest tryb `TwoWay`. Oznacza to, że jeśli program zmieni dane, wyświetlacz zaktualizuje się, a jeżeli użytkownik zmieni wyświetlaną nazwę, te dane również zostaną zaktualizowane.

Ostatni element określa, kiedy aktualizowany jest element źródłowy (właściwość w naszym widoku-modelu). Myślę, że jest to bardzo źle nazwany element. Wydaje Ci się przez to, że definiujesz coś, co zaktualizuje źródłowy wyzwalacz. W rzeczywistości ta instrukcja oznacza: „To jest wyzwalacz, który spowoduje aktualizację właściwości `source`". W przypadku naszego edytora chcemy, aby menedżer wyświetlania zaktualizował informacje w klasie widok-model za każdym razem, gdy zmieni się właściwość (to znaczy, gdy użytkownik wpisze coś w polu `TextBox` lub program zmieni właściwość), więc używamy ustawienia `PropertyChanged`.

ANALIZA KODU

Jak zrozumieć wiązanie danych?

Pytanie: Wiązanie danych wygląda na skomplikowane. Czy możesz mi jeszcze raz przypomnieć, dlaczego to robimy?

Odpowiedź: Implementujemy wiązanie danych, ponieważ chcemy, aby elementy widoku w naszym projekcie nie zawierały żadnego kodu programu. Chcemy, aby zmiany w wyświetlaczu były odzwierciedlane we właściwościach w naszym widoku-modelu bez naszej pomocy. Innymi słowy, wartość właściwości `Name` wewnątrz naszej klasy widoku-modelu powinna stawać się automatycznie widoczna na ekranie dla użytkownika, a zmiany dokonywane w wyświetlaczu powinny być odzwierciedlane w zawartości właściwości.

„Wiązanie" to bardzo dobry termin na to, co się tutaj dzieje. Gdy dwa elementy zostają związane, zmiany w jednym obiekcie są automatycznie odzwierciedlane w drugim, bez konieczności robienia czegokolwiek; jest to zachowanie ze wszech miar pożądane.

Pytanie: Jak faktycznie działa wiązanie danych?

Odpowiedź: Wiązanie danych można traktować jako zestaw instrukcji dla środowiska wyświetlania, które ustanawiają połączenie między dwoma obiektami. Widzieliśmy już podstawowy element procesu w mechanizmie `INotifyPropertyChanged`. Zapewnia sposób na to, aby jeden obiekt mógł powiedzieć drugiemu: „Hej! Zmieniłem się!", a ten drugi może następnie wykorzystać te informacje, aby zaktualizować coś w systemie. Wiązanie danych wykorzystuje interfejs `INotifyPropertyChanged` w celu spowodowania, żeby zmiany w jednym elemencie propagowały się przez obiekty w systemie.

Pytanie: Czy można wiązać tylko z właściwościami tekstowymi?

Odpowiedź: Nie. W rzeczywistości jest to jeden z najpotężniejszych aspektów wiązania danych. Właściwości programu można powiązać z wieloma innymi właściwościami elementów na wyświetlaczu, w tym z ich pozycją, kolorem, a nawet rozmiarem. Proces jest dokładnie taki sam...

Klasa widok-model

Klasa widok-model zarządza interakcją między widokiem a obiektami danych (zwanymi również modelem). Właściwość, z którą połączony jest XAML, jest definiowana w klasie widoku-modelu, a widok-model zawiera również wszystkie zachowania, które sprawiają, że edycja działa.

```
public class ContactManagerViewModel : INotifyPropertyChanged
{
    // Prywatny łańcuch znaków nazwy
    private string name;

    // Publiczna właściwość nazwy
    public string Name
    {
        get
        {
            // Zwraca prywatną wartość dla get
            return name;
        }
        set
        {
            // Ustawia prywatną właściwość na przychodzącą wartość
```

```
            name = value;

            if (PropertyChanged != null)
            {
                // Jeśli obiekt zarejestrował swoje zainteresowanie właściwością,
                // wywołuje zdarzenie PropertyChanged, aby wskazać, że zmieniła się nazwa
                PropertyChanged(this, new PropertyChangedEventArgs("Name"));
            }
        }
    }
}
```

Powyższy kod pokazuje właściwość `Name` w klasie widok-model. Jeśli wydaje Ci się, że widziałeś już ten układ wcześniej, to dobrze Ci się wydaje. Mamy ten sam kod wewnątrz obiektu `Contact`, służący temu, aby system wyświetlania mógł wiązać się z kontaktem i wykrywać, kiedy zmienia się nazwa kontaktu. Rzeczywisty widok-model w naszej aplikacji będzie miał takie właściwości dla każdego elementu danych, który widok-model prezentuje widokowi. W naszym przypadku są to `Address`, numer `Phone`, pole `ListBox` wyszukiwania i wyszukiwany tekst.

Łączenie widoku-modelu z widokiem

Mamy teraz widok XAML oraz widok-model C#. Następnym krokiem jest ich połączenie. Robimy to, ustawiając `DataContext` (kontekst danych) dla strony edycji na klasę widok-model. Ten kontekst danych definiuje obiekt, na który zmapowane będą wszystkie wiązania. W przypadku naszego edytora chcemy, aby tym kontekstem była instancja klasy `ContactManagerViewModel`.

```
<Page
    x:Class="ContactManager.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ContactManager"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Page DataContext>
        <local:ContactManagerViewModel x:Name="contactManagerViewModel"/>
    </Page DataContext>
```

Standardowy nagłówek XAML dla strony edytora.

Początek informacji DataContext dla strony.

Identyfikuje klasę, która ma być użyta, oraz lokalną nazwę.

Część opisu `local:ContactManagerViewModel` identyfikuje klasę, która ma zostać utworzona. Natomiast część opisu `x:Name="contactManagerViewModel"` nazywa tworzoną instancję.

ANALIZA KODU

Korzystanie z DataContext

Pytanie: Czym właściwie jest `DataContext`?

Odpowiedź: Wiązania XAML zawierają nazwy elementów, z którymi chcemy się wiązać. Wiążemy na przykład nazwę (`Name`) naszego kontaktu z polem `TextBox` na wyświetlaczu. `DataContext` to łącze między XAML a obiektem C#, który faktycznie zawiera właściwość `Name`, przechowującą wartość, którą będziemy wiązać. Jest to „kontekst, w którym istnieją dane". Jako kontekstu danych (`DataContext`) dla strony XAML moglibyśmy użyć dowolnej klasy, o ile klasa będzie zawierać właściwości pasujące do tych na stronie.

Pytanie: W jaki sposób wykorzystywany jest `DataContext` podczas działania programu?

Odpowiedź: Po załadowaniu strony system tworzy instancję klasy widoku-modelu. W tym przypadku tworzy instancję klasy `ContactManagerViewModel`, która będzie się nazywała `contactManagerViewModel`. Widzieliśmy już wcześniej tego rodzaju tworzenie obiektów. Każdy z elementów wyświetlacza w XMAL jest również tworzony podczas ładowania strony.

Pytanie: Gdzie konfigurowane jest połączenie z danymi?

Odpowiedź: Klasa widok-model ma za zadanie połączyć widok z danymi. Dzieje się to podczas tworzenia instancji widoku-modelu. Właśnie wtedy aplikacja załaduje dane z dowolnego używanego magazynu danych, a następnie przygotuje je do użycia przez edytor.

Przekazywanie poleceń do widoku-modelu

Wiązanie danych pozwala nam połączyć właściwości z klasy widok-model z elementami wyświetlacza i widoku. Nie zapewnia jednak środków, za pomocą których moglibyśmy przekazywać polecenia do widoku-modelu. Musimy wprowadzić do naszego widoku-modelu jedno polecenie — to inicjujące przeszukiwanie zapisanych kontaktów w celu znalezienia kontaktów o nazwie, która zawiera określony łańcuch znaków. We wcześniejszej aplikacji utworzyliśmy procedurę obsługi zdarzeń, która uruchamia się po naciśnięciu przycisku *Szukaj*. To samo możemy zrobić w naszej aplikacji widoku-modelu, z tą różnicą, że tym razem procedura obsługi zdarzeń wywołuje po prostu metodę w widoku-modelu:

```
namespace ContactManager
{
    /// <summary>
    /// Pusta strona, która może być użyta samodzielnie lub można do niej nawigować za pomocą metod Frame
    /// </summary>
    public sealed partial class MainPage : Page
    {
        // Konstruktor strony
        public MainPage()
        {
            this.InitializeComponent();
        }

        // Zachowanie wyszukiwania powiązane z przyciskiem
        private void SearchButton_Click(object sender, RoutedEventArgs e)
        {
            // Prosi widok-model o przeprowadzenie wyszukiwania
            contactManagerViewModel.DoSearch();
        }
    }
}
```

Gdy użytkownik naciśnie przycisk *Szukaj*, procedura obsługi zdarzeń wywołuje metodę `DoSearch` wewnątrz klasy widok-model.

```
public class ContactManagerViewModel : INotifyPropertyChanged
{
    public void DoSearch()
    {
        List<Contact> foundList = contacts FindContactsWithName(SearchText);

        FoundList = new ObservableCollection<Contact>(foundList);
    }
}
```

Jest to cała zawartość metody `DoSearch` w widoku-modelu. Prosi ona model (magazyn kontaktów), aby utworzył obiekt `List` wszystkich kontaktów zawierających w nazwie wyszukiwany tekst. Następnie tworzy z tej listy obiekt `ObservableCollection` i ustawia właściwość `FoundList` na znalezioną listę. Ten kod działa dokładnie tak samo jak w poprzedniej wersji edytora, ale jest znacznie prostszy.

ANALIZA KODU

Magia wzorca MVVM

Pytanie: W jaki sposób procedura obsługi zdarzeń znajduje klasę widok-model do wywołania?

Odpowiedź: Kiedy ustawiamy `DataContext` w kodzie XAML, mówimy, że „wszystkie wiązania danych w tym projekcie są powiązane z właściwościami w klasie o nazwie `ContactManagerViewModel`". Instancja tej klasy, którą zamierzamy utworzyć, nazywa się `contactManagerViewModel`. Ta instancja jest tworzona podczas ładowania strony i można z niej później korzystać w kodzie działającym w klasie widoku.

Pytanie: W jaki sposób metoda `DoSearch` przechwytuje łańcuch znaków do wyszukania?

Odpowiedź: W poprzedniej wersji program musiał znaleźć pole `TextBox`, które zawierało część nazwy do wyszukania. W takim przypadku używamy po prostu właściwości `SearchText`. Ta właściwość jest powiązana z widokiem i dostarcza tekst, który chcemy wyszukać. Widok-model nie wie, gdzie odbywa się wiązanie, ani nawet z jaką kontrolką powiązana jest właściwość; wie po prostu, że ta właściwość będzie przechowywać tekst do wyszukania. To ilustruje magię wzorca MVVM.

Pytanie: Jak ustawić zawartość elementu `ListBox` na wyniki wyszukiwania?

Odpowiedź: To raczej magia widoku-modelu. Właściwość `FoundList` w widoku-modelu jest powiązana z właściwością `ItemsSource` elementu `ListBox`, który wyświetla listę kontaktów. Gdy przypisanie zostanie wykonane przez drugą instrukcję w metodzie `DoSearch`, `ListBox` zaktualizuje się automatycznie.

Jeśli dowiesz się nieco więcej o wzorcu MVVM, znajdziesz sposób użycia mechanizmu poleceń, który umożliwia bezpośrednie wiązanie zdarzeń z metodami z klasą widok-model. Jednak dla naszych celów ten projekt jest wystarczający.

Wykrywanie zmian w widoku

Program działa w taki sposób, że prawniczka będzie wybierać kontakty z `ListView`, kiedy będzie chciała z nimi pracować. Możemy użyć wiązania danych, aby wykryć, kiedy wybrany kontakt się zmieni.

```
<ListBox ItemsSource="{Binding FoundList}" Name="ContactListBox" Margin="4"
Height="273" SelectedItem="{Binding SelectedContact,Mode=TwoWay}" >
```

Gdy prawniczka wybiera element z `ListBox`, zmienia się wybrany element. Wiązanie spowoduje zmianę właściwości `SelectedContact` w klasie widok-model.

```
public class ContactManagerViewModel : INotifyPropertyChanged
{
    private Contact selectedContact;

    public Contact SelectedContact
    {
        get
        {
            return selectedContact;
        }
        set
        {
            if (selectedContact != null)
                saveContactFromPage (selectedContact);

            selectedContact = value;

            if (selectedContact != null)
            {
                placeContactOnPage(SelectedContact);
            }

            if (PropertyChanged != null)
            {
                PropertyChanged(this,
                        new PropertyChangedEventArgs("SelectedContact"));
            }
        }
    }
}

Demo 18-04 MVM ContactEditor
```

Zachowanie `set` we właściwości pozwala programowi przejąć kontrolę, gdy właściwość jest zmieniana. Gdy kontakt zostanie wybrany, chcemy zapisać ten, który wcześniej edytowaliśmy, a następnie wybrać nowy do edycji. Tak właśnie zachowuje się powyższy kod. Jeśli uruchomisz program demo, przekonasz się, że działa dokładnie tak, jak powinien, ale wszystkimi zachowaniami zarządza się w klasie widok-model.

Z PUNKTU WIDZENIA PROGRAMISTY

Niektórych rzeczy uczysz się, studiując kod

Zasada Model-Widok-WidokModel jest istotna podczas tworzenia programów. Nie jest to jednak coś, co będziesz mógł w pełni pojąć czytając tylko opisy jej działania. Jedynym sposobem, aby naprawdę zrozumieć, jak to działa, jest zastanowienie się, co programista próbuje zrobić, a następnie przeanalizowanie kodu, który to robi.

Studiując kod napisany przez innych programistów, odebrałem kilka świetnych lekcji programowania. Zwykle podczas czytania cudzego kodu zaczynam od: „Dlaczego, u licha, oni to zrobili?", aż dochodzę do: „Kurczę blade, to jest całkiem sprytne". Jeśli kilka poprzednich punktów rozdziału było dla Ciebie trudne do ogarnięcia (co jest całkowicie zrozumiałe), możesz się wiele nauczyć ze studiowania kodu.

Pamiętaj jednak, że nie musisz jedynie przeglądać kodu. Możesz wykonywać na nim „eksperymenty", wstawiając punkty przerwania za pomocą Visual Studio, a następnie przechodząc przez kod w trakcie jego działania.

ZRÓB TO SAM

Zbadaj MVVM

Teraz, gdy zaczynasz rozumieć, jak działa MVVM, poświęć trochę czasu, aby zbudować coś prostego przy użyciu tego wzorca. Możesz wziąć istniejący program i spróbować odgadnąć, co powinien zawierać widok, co będzie w modelu-widoku i co powinno się znaleźć w samym modelu. Możesz także zbudować eksperymentalną aplikację, która będzie łączyć wejściowe pole `TextBox` z wyjściowym polem `TextBlock` przy użyciu wiązania danych.

Pamiętaj, że wiązania danych możesz używać do kontrolowania każdego aspektu elementów na stronie XAML, także obrazów i dźwięków.

Czego się nauczyłeś?

W tym rozdziale poświęciłeś dużo czasu na poznawanie wzorca projektowego Model-Widok--WidokModel, służącego do budowania interfejsów użytkownika, które są łatwe do testowania i zarządzania. Dowiedziałeś się, że istnieją **wzorce**, które wyznaczają sposób robienia pewnych rzeczy. Odkryłeś również zasadę wiązania danych, dzięki której za kulisami można użyć kodu, który sprawi, że zmiany w jednym obiekcie będą powodować zmiany w drugim. Przekonałeś się, że właściwości są ważną częścią wiązania danych, gdyż pozwalają uruchamiać określone działania kodu, gdy zmieniają się dane w klasie. Przyjrzałeś się także **delegatom**, czyli obiektom, które pozwalają programowi manipulować referencjami do metod.

To był świetny rozdział do ćwiczenia umiejętności przy użyciu obiektów. Umożliwił Ci również docenienie znaczenia projektu. A na koniec oczywiście kilka pytań.

Czy Model-Widok-WidokModel jest jedynym sposobem na organizowanie struktury aplikacji?

Nie. Istnieje wiele sposobów tworzenia aplikacji z interfejsem użytkownika. Wzorzec MVVM został utworzony z myślą o języku XAML, służącym do opisywania stron. Zasada wiązania danych jest stosowana w wielu różnych aplikacjach. Zrozumienie, jak to działa, jest bardzo przydatne. Jeśli jednak chcesz tworzyć wszystkie aplikacje przy użyciu technik „kod na stronie", które widziałeś w poprzednim rozdziale, to dla mnie jest to całkowicie w porządku i zapewne będzie takie również dla Twoich użytkowników.

Jak właściwie testujemy aplikacje zbudowane przy użyciu widoków-modeli?

Gdy masz już klasę widoku-modelu, możesz ją przetestować w taki sam sposób, jak traktujesz każdy inny obiekt oprogramowania. Możesz wyzwalać zachowania w obiekcie i porównywać to, co się dzieje z tym, co powinno się dziać. Możesz na przykład przetestować zachowanie naszego edytora w zakresie wyszukiwania, tworząc zestaw danych testowych, ustawiając jakiś tekst we właściwości `SearchText`, wywołując metodę `DoSearch`, a następnie przeglądając zawartości kolekcji `FoundList`. Piękno tego procesu polega na tym, że w żadnym momencie nie trzeba patrzeć na ekran, żeby testować, i nie ma szans na jakiekolwiek błędy w kodzie widoku, ponieważ widok w ogóle nie zawiera żadnego kodu.

Czy program może zawierać więcej niż jeden widok-model?

Tak, może. Dany widok-model jest zwykle kojarzony z określoną aktywnością w programie. W aplikacji bankowej możesz mieć różne widoki-modele dla poszczególnych rodzajów transakcji, które będzie wykonywał system.

Dlaczego delegaty są tak przydatne?

Delegata możesz potraktować jako referencję, która może odwoływać się do metod. Widziałeś, że służyły do implementacji przycisków w interfejsie użytkownika. Obiekt `Button` otrzymuje obiekt delegowany, który identyfikuje metodę do wywołania po aktywowaniu przycisku. Mechanizm `INotifyPropertyChanged` również używa obiektów delegowanych; są one jak zwykły obiekt i mogą zarejestrować zainteresowanie zmianami we właściwości. Obserwowany obiekt podaje właściwości delegata, którego można wywołać, jeśli nastąpi jakaś zmiana. Jednak moc delegatów wykracza poza te dwa zastosowania.

Listę referencji delegatów możesz zamienić w „program w programie". Poszczególne delegaty mogą być wywoływane po kolei w celu zapewnienia sekwencji działań. Użyłem tego z bardzo dobrym efektem w grach, aby zapewnić „skrypty" do wykonywania przez elementy gry. Zrozumienie, jak działają delegaty i jak z nich korzystać, zapewnia Ci naprawdę wartościową umiejętność.