Twoja przepustka do świata C#!

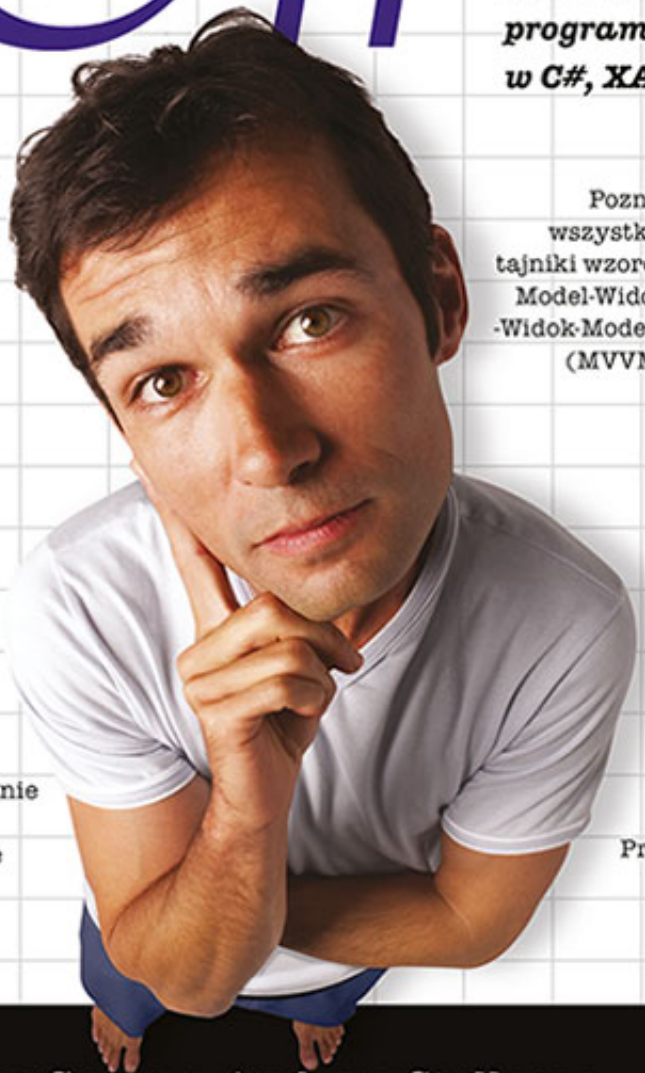
Wydanie III

Rusz głową!

C#

Doskonały podręcznik do nauki praktycznego programowania w C#, XAML i .NET

Zarządzaj swoimi obiektami dzięki wykorzystaniu abstrakcji i dziedziczenia



Poznaj wszystkie tajniki wzorca Model-Widok-Widok-Modelu (MVVM)

Napisz w pełni funkcjonalną staromodną grę wideo

Dowiedz się, w jaki sposób programowanie asynchroniczne pozwoliło Krystynie spełnić wymagania klientów

Przekonaj się, jak Janek zastosował kolekcje i LINQ, by ujarzmić niesforną kolekcję komiksów

O'REILLY®

Jennifer Greene, Andrew Stellman

Helion

Tytuł oryginału: Head First C#, 3rd Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-8311-6

Authorized Polish translation of the English edition of **Head First C#, 3rd Edition**
ISBN 9781449343507 © 2013 Jennifer Greene, Andrew Stellman

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/cshru3.zip

Spis treści (skrócony)

Spis treści (z prawdziwego zdarzenia)

Wstęp

Przygotuj się na C#. Właśnie sobie siedzisz i próbujesz się czegoś nauczyć, ale mózg wciąż powtarza Ci, że cała ta nauka *nie jest ważna*. Twój umysł mówi: „Lepiej wyjdź z pokoju i zajmij się ważniejszymi sprawami, takimi jak to, których dzikich zwierząt unikać, oraz to, że strzelanie z łuku na golasa nie jest dobrym pomysłem". W jaki sposób oszukać mózg, tak aby myślał, że Twoje życie naprawdę zależy od nauki C#?

Zacznij pisać programy w C#

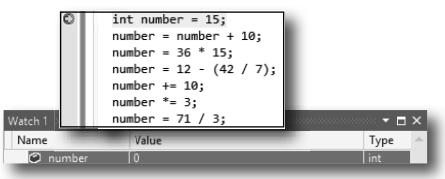
1 Napisz coś fajnego, i to szybko!

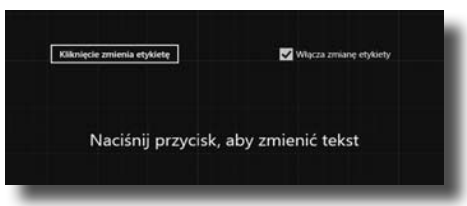
Czy chcesz tworzyć wspaniałe programy naprawdę szybko? Wraz z C# dostajesz do ręki **potężny język programowania** i wartościowe narzędzie. Dzięki **Visual Studio IDE** do historii przejdą sytuacje, w których musiałeś pisać jakiś nędzny kod, by ponownie zapewnić prawidłowe działanie przycisku. I to nie wszystko. Dodatkowo będziesz mógł **skupić się na faktycznym wykonywaniu naprawdę fajnych programów,** zamiast starać się zapamiętać, który parametr metody odpowiadał za *nazwę* przycisku, a który za *wyświetlany na nim tekst*. Brzmi zachęcająco? Przewróć zatem stronę i przystąpmy do programowania.

To tylko kod

2 Pod maską

Jesteś programistą, nie jedynie użytkownikiem IDE. IDE może wykonać za Ciebie wiele pracy, ale na razie jest to wszystko, co może dla Ciebie zrobić. Oczywiście istnieje wiele **powtarzalnych czynności** podczas pisania aplikacji i IDE okazuje się tu bardzo pomocne. Praca z nim to jednak *dopiero początek*. Możesz wycisnąć ze swoich programów znacznie więcej — **pisanie kodu C#** to właśnie droga, która doprowadzi Cię do tego celu. Kiedy osiągniesz mistrzowski poziom w kodowaniu, nie będzie *żadnej* rzeczy, której Twój program nie umiałby zrobić.

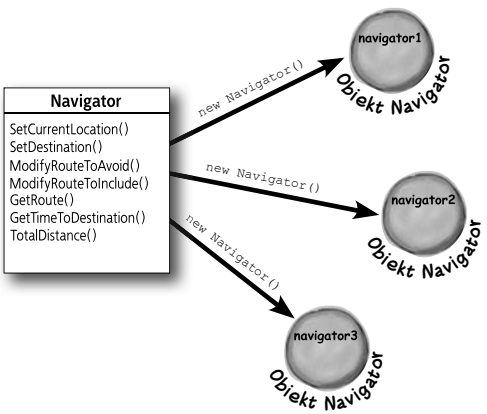


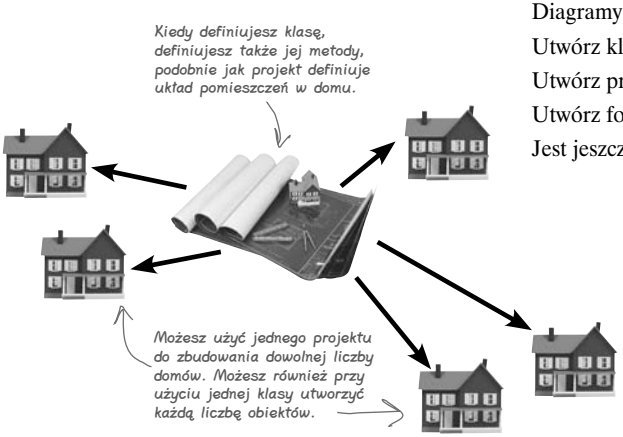


Obiekty: zorientuj się!

3 Tworzenie kodu ma sens

Każdy pisany przez Ciebie program rozwiązuje jakiś problem. Rozpoczynając pisanie programu, zawsze warto zastanowić się, jaki *problem* ma on rozwiązywać. Właśnie do tego przydają się **obiekty**. Pozwalają one tworzyć strukturę kodu tak, by odpowiadała ona rozwiązywanemu problemowi, dzięki czemu będziesz mógł *skoncentrować się na nim samym*, a nie na mechanice tworzenia kodu. Prawidłowe użycie obiektów spowoduje, że proces pisania kodu stanie się bardziej *intuicyjny*, a jego późniejsza analiza i modyfikacja — znacznie łatwiejsze.





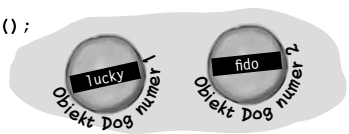
Typy i referencje

4 Jest 10:00. Czy wiesz, gdzie są Twoje dane?

Typ danych, baza danych, dane komandora porucznika… wszystko to są ważne rzeczy. Bez danych Twoje programy są bezużyteczne. Potrzebujesz informacji dostarczanych przez użytkowników. Na ich podstawie wyszukujesz lub tworzysz nową **informację** i zwracasz im ją. W rzeczywistości prawie wszystko, co robisz podczas programowania, sprowadza się do **pracy z danymi** w taki czy w inny sposób. W tym rozdziale dowiesz się o różnych aspektach **typów danych** C#, nauczysz się pracować z danymi w programie, a nawet odkryjesz kilka pilnie strzeżonych sekretów **obiektów** (*psssst… obiekty to także dane*).


```
Dog fido;
Dog lucky = new Dog();
```


```
fido = new Dog();
```

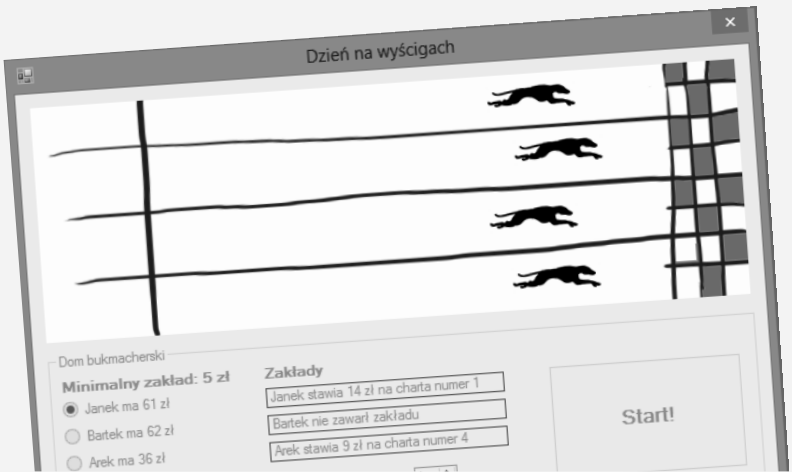



```
lucky = null;
```

Laboratorium C# numer 1

Dzień na wyścigach

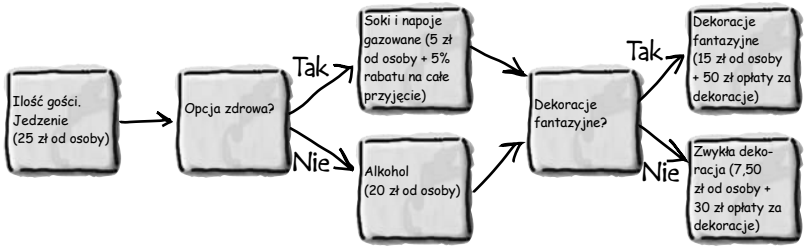
Janek, Bartek i Arek uwielbiają chodzić na tor wyścigowy, ale ciągła utrata pieniędzy powoduje u nich frustrację. Potrzebują symulatora, aby mogli określić zwycięzcę, zanim wyłożą pieniądze na zakłady. Jeśli dobrze wywiążesz się z zadania, będziesz miał procenty z ich wygranych.



Hermetyzacja

5 Co ma być ukryte... niech będzie ukryte

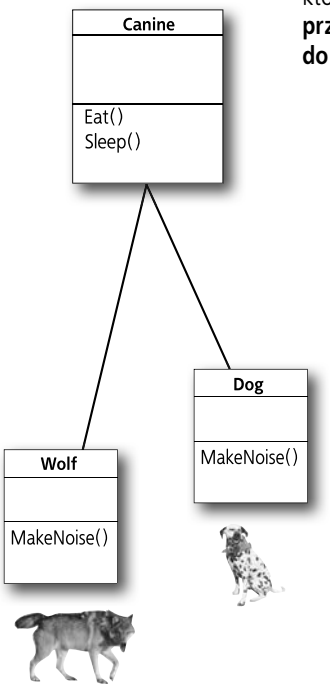
Czy kiedykolwiek marzyłeś o odrobinie prywatności? Czasami Twoje obiekty czują się tak samo. Na pewno nie lubisz sytuacji, w których ktoś, komu nie ufasz, czyta Twój pamiętnik lub przegląda wykazy Twoich operacji bankowych. Dobre obiekty nie pozwalają ***innym*** obiektom na oglądanie swoich pól. W tym rozdziale nauczysz się wykorzystywać potęgę **hermetyzacji**. **Sprawisz, że dane obiektów będą prywatne**, i dodasz metody, które umożliwią Ci **zabezpieczenie dostępu do danych**.

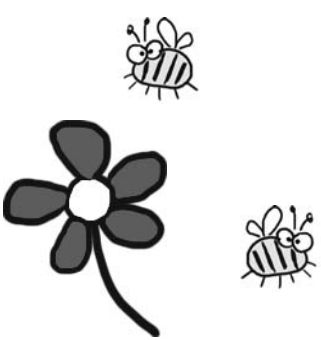


Dziedziczenie

6 Drzewo genealogiczne Twoich obiektów

Czasami CHCIAŁBYŚ być dokładnie taki sam jak Twoi rodzice. Czy kiedykolwiek natknąłeś się na obiekt, który robiłby ***prawie*** wszystko, czego byś sobie od niego życzył? Czy kiedykolwiek znalazłeś się w takiej sytuacji, że gdybyś ***zmienił dosłownie kilka rzeczy***, obiekt byłby doskonały? Cóż, to tylko jeden z wielu powodów, które sprawiają, że **dziedziczenie** zalicza się do najważniejszych koncepcji i technik w języku C#. Kiedy skończysz czytać ten rozdział, będziesz wiedział, jak **rozszerzać** obiekty, by móc wykorzystywać ich zachowania i jednocześnie dysponować **elastycznością**, która pozwoli Ci te zachowania modyfikować. Unikniesz **wielokrotnego pisania kodu**, **przedstawisz prawdziwy świat** znacznie dokładniej, a w efekcie otrzymasz kod **łatwiejszy do zarządzania**.

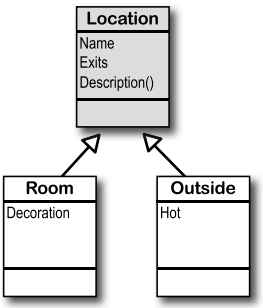


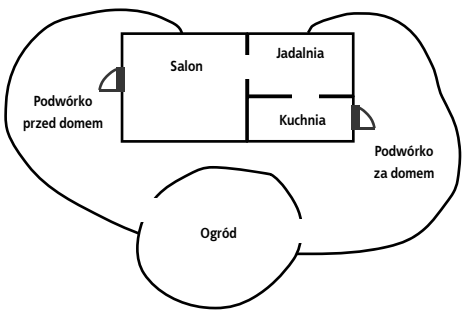


Interfejsy i klasy abstrakcyjne

7 Klasy, które dotrzymują swoich obietnic

Czyny potrafią powiedzieć więcej niż słowa. Czasami potrzebujesz pogrupować swoje obiekty na podstawie tego, **co robią**, zamiast tego, po jakiej klasie dziedziczą. To jest moment, w którym należy powiedzieć o **interfejsach**. Pozwalają one na pracę z każdą klasą, która jest w stanie wykonać daną czynność. Jednak wraz z **wielkimi możliwościami przychodzi wielka odpowiedzialność** i każda klasa, która implementuje interfejs, **musi wypełnić wszystkie swoje obowiązki**... albo kompilator połamie Ci nogi, zrozumiałeś?

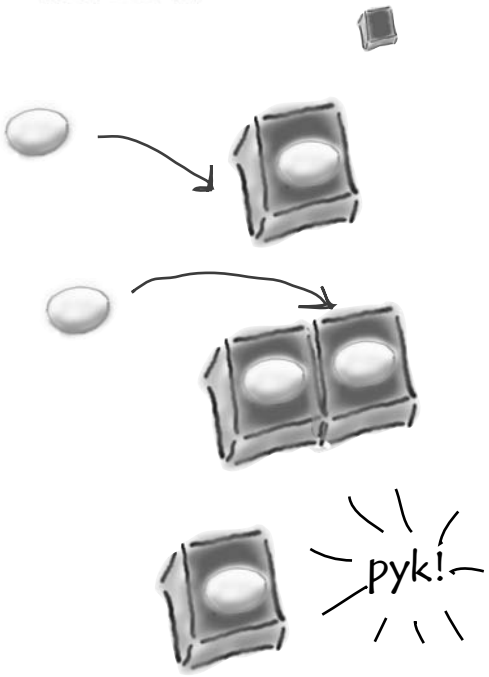




Typy wyliczeniowe i kolekcje

8 Przechowywanie dużej ilości danych

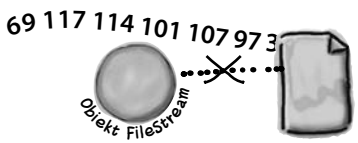
Z deszczu pod rynnę. W rzeczywistym świecie nie musisz się zwykle zajmować danymi w małych ilościach i w niewielkich fragmentach. Nie, Twoje dane przychodzą do Ciebie w **grupach, stosach, pękach, kopach**. Potrzebujesz jakiegoś potężnego narzędzia do ich zorganizowania. Nadszedł czas, aby przedstawić **kolekcje**. Pozwalają one **przechowywać i sortować dane, a także zarządzać** tymi z nich, które Twój program musi przeanalizować. Dzięki temu możesz myśleć o pisaniu programów do pracy z danymi, a samo ich przechowywanie zostawić kolekcjom.



Odczyt i zapis plików

9 Zachowaj te bajty dla mnie!

Czasem opłaca się być trwałym. Do tej pory wszystkie programy były krótkotrwałe. Uruchamiały się, działały przez chwilę i były zamykane. Czasami nie jest to wystarczające, zwłaszcza jeżeli zajmujesz się ważnymi danymi. Musisz mieć możliwość **zapisania swojej pracy**. W tym rozdziale pokażemy sposób **zapisywania danych do pliku**, a następnie **wczytywania tych informacji z powrotem** do programu. Dowiesz się co nieco o **klasach strumieni** .NET i zetkniesz się z tajemnicami systemów **szesnastkowego** i **dwójkowego**.



Laboratorium C# numer 2

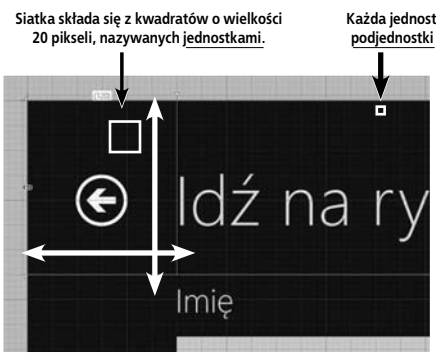
Wyprawa

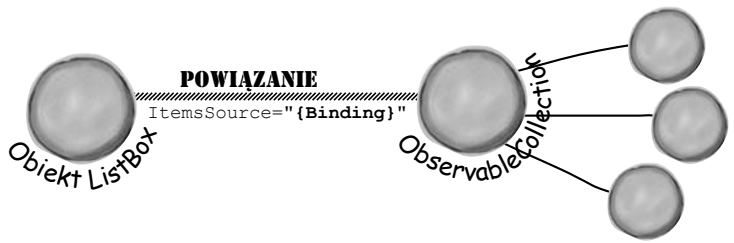
Twoim zadaniem jest stworzenie gry przygodowej, w której potężny wojownik wyrusza na wyprawę i dzielnie walczy, poziom za poziomem, ze śmiertelnie niebezpiecznymi wrogami. Stworzysz system turowy. Oznacza to, że najpierw gracz wykonuje jeden ruch, a następnie ruch wykonuje przeciwnik. Gracz może przesunąć się lub zaatakować; potem możliwość ruchu i ataku dostaje każdy z wrogów. Gra toczy się do czasu, aż gracz pokona wszystkich przeciwników na wszystkich siedmiu poziomach lub zginie.

Projektowanie aplikacji dla Sklepu Windows z użyciem XAML

10 Przenosząc swoje aplikacje na wyższy poziom

Jesteś już gotów, by wkroczyć do zupełnie nowego świata tworzenia aplikacji. Korzystanie z technologii WinForms do tworzenia klasycznych aplikacji dla systemu Windows jest doskonałym sposobem nauki ważnych rozwiązań języka C#, niemniej jednak możesz pójść *znacznie dalej*. W tym rozdziale dowiesz się, jak używać języka **XAML** do projektowania aplikacji przeznaczonych dla Sklepu Windows, nauczysz się tworzyć aplikacje **działające na dowolnych urządzeniach, integrować** dane ze stronami przy użyciu **wiązania danych** i używać Visual Studio do ujawniania tajemnic stron XAML poprzez badanie obiektów tworzonych na podstawie kodu XAML.





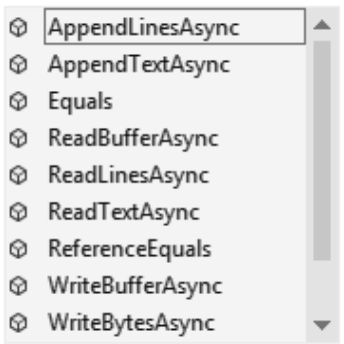
Async, await i serializacja kontraktu danych

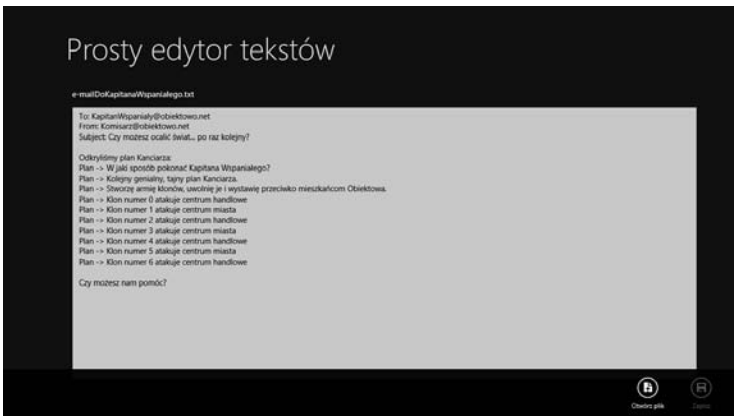
11 Przepraszam, że przerywam

Nikt nie lubi być zmuszanym do oczekiwania… zwłaszcza użytkownicy.

Komputery są doskonałe w wykonywaniu wielu rzeczy jednocześnie, nie ma zatem żadnego powodu, aby Twoje aplikacje nie mogły tego robić. W tym rozdziale dowiesz się, jak sprawić, by dzięki **zastosowaniu metod asynchronicznych** Twoje aplikacje reagowały błyskawicznie. Nauczysz się także korzystać z **wbudowanych narzędzi do wybierania plików**, wyświetlać **okienka z komunikatami** oraz **asynchronicznie zapisywać i odczytywać dane z plików** bez „zawieszania" aplikacji. Połączysz te wszystkie możliwości z serializacją kontraktu danych i opanujesz tworzenie bardzo nowoczesnych aplikacji.


```
FileIO.
```



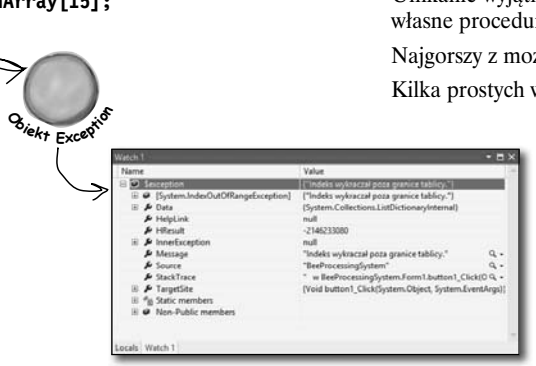


Obsługa wyjątków

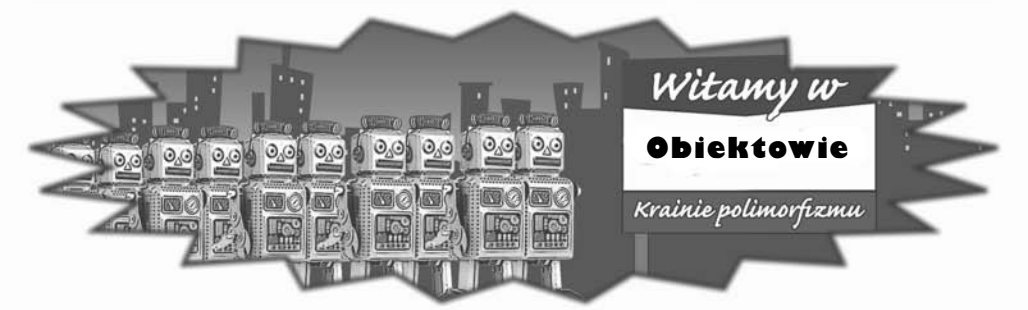
12 Gaszenie pożarów nie jest już popularne

Programiści nie mają być strażakami. Pracowałeś jak wół, przebrnąłeś przez dokumentacje techniczne i kilka zajmujących książek *Rusz głową!*, wspiąłeś się na szczyt swoich możliwości: jesteś **mistrzem programowania**. W dalszym ciągu musisz jednak odrywać się od pracy, ponieważ **program wyłącza się** lub **nie zachowuje się tak, jak powinien**. Nic nie wybija Cię z rytmu tak, jak obowiązek naprawienia dziwnego błędu... Z **obsługą wyjątków** możesz jednak napisać kod, który **poradzi sobie z pojawiającymi się problemami**. Jest nawet lepiej, możesz bowiem zareagować na ich pojawienie się i sprawić, że wszystko **będzie dalej działało**.


```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```



13 Kapitan Wspaniały
Śmierć obiektu



Przeszukiwanie danych i tworzenie aplikacji przy użyciu LINQ

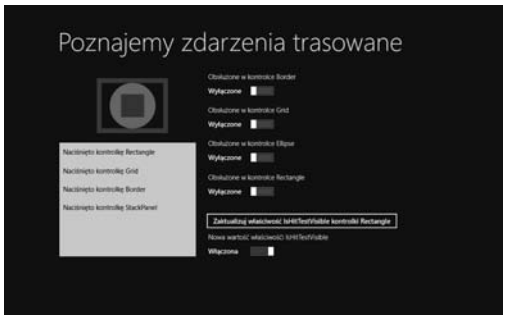
14 Przejmij kontrolę nad danymi

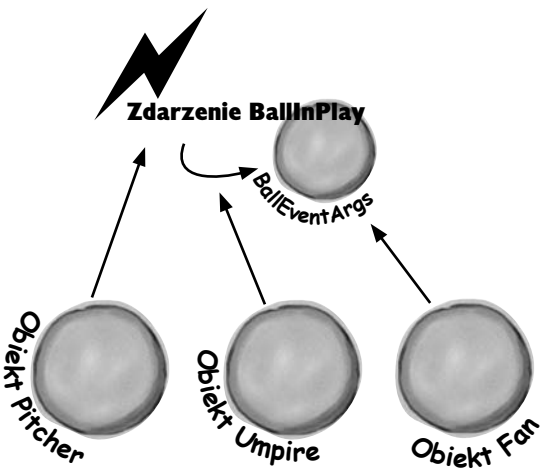
To świat przepełniony danymi... Lepiej, żebyś wiedział, jak w nim żyć. Czasy, gdy mogłeś programować kilka dni, a nawet kilka tygodni, bez konieczności pracy **z ogromem danych**, minęły już bezpowrotnie. Nadeszła epoka, w której ***wszystko opiera się na nich***. W tym miejscu do akcji wkracza **LINQ**. To nie tylko sposób na **pobieranie danych** w prosty, intuicyjny sposób. Pozwala on także **grupować** i **łączyć dane pochodzące z różnych źródeł**. A kiedy już podzielisz dane na fragmenty, którymi można łatwo zarządzać, Twoje aplikacje dla Sklepu Windows będą **korzystać z kontrolek do nawigowania**, pozwalających poruszać się po danych, przeglądać je, a nawet powiększać i wyświetlać szczegółowe informacje na ich temat.

Zdarzenia i delegaty

15 Co robi Twój kod, kiedy nie patrzysz

Twoje obiekty zaczynają myśleć o sobie. Nie możesz zawsze kontrolować tego, co one robią. Czasami różne rzeczy... zdarzają się. Kiedy to następuje, chciałbyś, aby Twoje obiekty były wystarczająco sprytne i odpowiednio **reagowały**. To miejsce, w którym do akcji wkraczają zdarzenia. Jeden obiekt *udostępnia* zdarzenie, inny je *obsługuje* i wszystko pracuje razem, aby całość działała sprawnie. Jest to wspaniałe, o ile nie chcesz, by Twój obiekt mógł kontrolować, kto będzie mógł nasłuchiwać jego zdarzeń. Wtedy bardzo pomocne okazują się **funkcje zwrotne**.

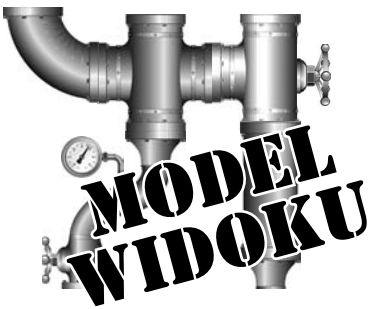


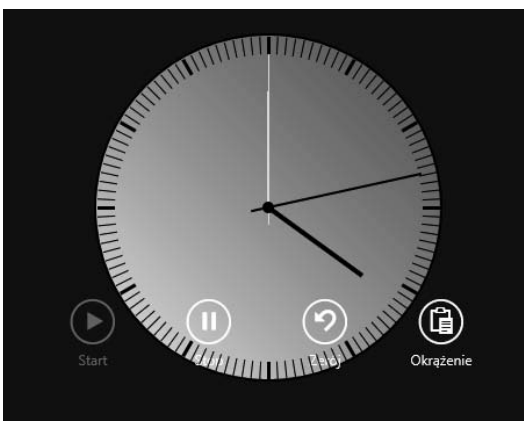


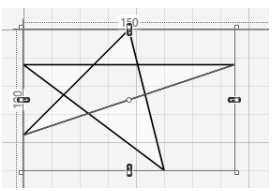
Projektowanie aplikacji według wzorca MVVM

16 Świetne aplikacje od zewnątrz i od środka

Twoje aplikacje muszą być nie tylko wspaniałe wizualnie. Kiedy mówimy o *projekcie*, co Ci przychodzi do głowy? Przykład jakiejś wspaniałej architektury budowlanej? Doskonale rozplanowana strona WWW? Produkt zarówno estetyczny, jak i dobrze zaprojektowany? Dokładnie te same zasady odnoszą się do aplikacji. W tym rozdziale poznasz **wzorzec Model-View-ViewModel** (MVVM, model-widok-widok modelu) i dowiesz się, jak używać go do tworzenia dobrze zaprojektowanych aplikacji o luźnych powiązaniach. Przy okazji poznasz **animacje, szablony kontrolek** używane do projektowania wyglądu aplikacji, nauczysz się stosować **konwertery**, by ułatwiać korzystanie z techniki wiązania danych, a w końcu zobaczysz, jak połączyć te wszystkie elementy, by ***stworzyć solidne podstawy*** do tworzenia dowolnych aplikacji **w języku C#**.







Laboratorium C# numer 3

Invaders

Dzięki temu laboratorium oddasz hołd jednej z najbardziej popularnych, czczonych i powielanych ikon w historii gier komputerowych. Nie potrzebuje ona żadnego wprowadzenia. Czas utworzyć grę Invaders.

Projekt dodatkowy!

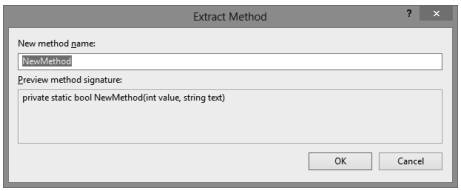
17 Napisz aplikację Windows Phone

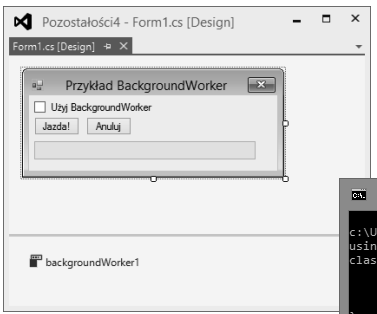
Klasy, obiekty, XAML, hermetyzacja, dziedziczenie, polimorfizm, LINQ, MVVM... dysponujesz już wszystkimi narzędziami niezbędnymi do pisania wspaniałych aplikacji dla Sklepu Windows oraz tradycyjnych aplikacji okienkowych. Czy jednak wiesz, że **tych samych narzędzi możesz użyć do pisania aplikacji dla *Windows Phone?*** Tak, to prawda! W tym dodatkowym projekcie poznasz proces tworzenia gry dla systemu Windows Phone. Jeśli nie posiadasz odpowiedniego urządzenia, to i tak nie masz się czym przejmować — będziesz mógł w nią grać na **emulatorze Windows Phone.** A zatem zaczynajmy!

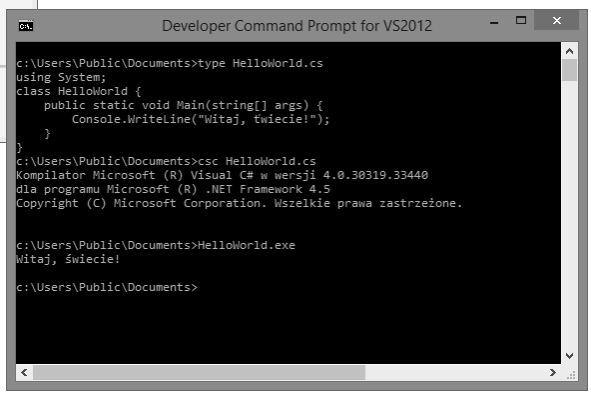
Pozostałości

A 11 najważniejszych rzeczy, które chcieliśmy umieścić w tej książce

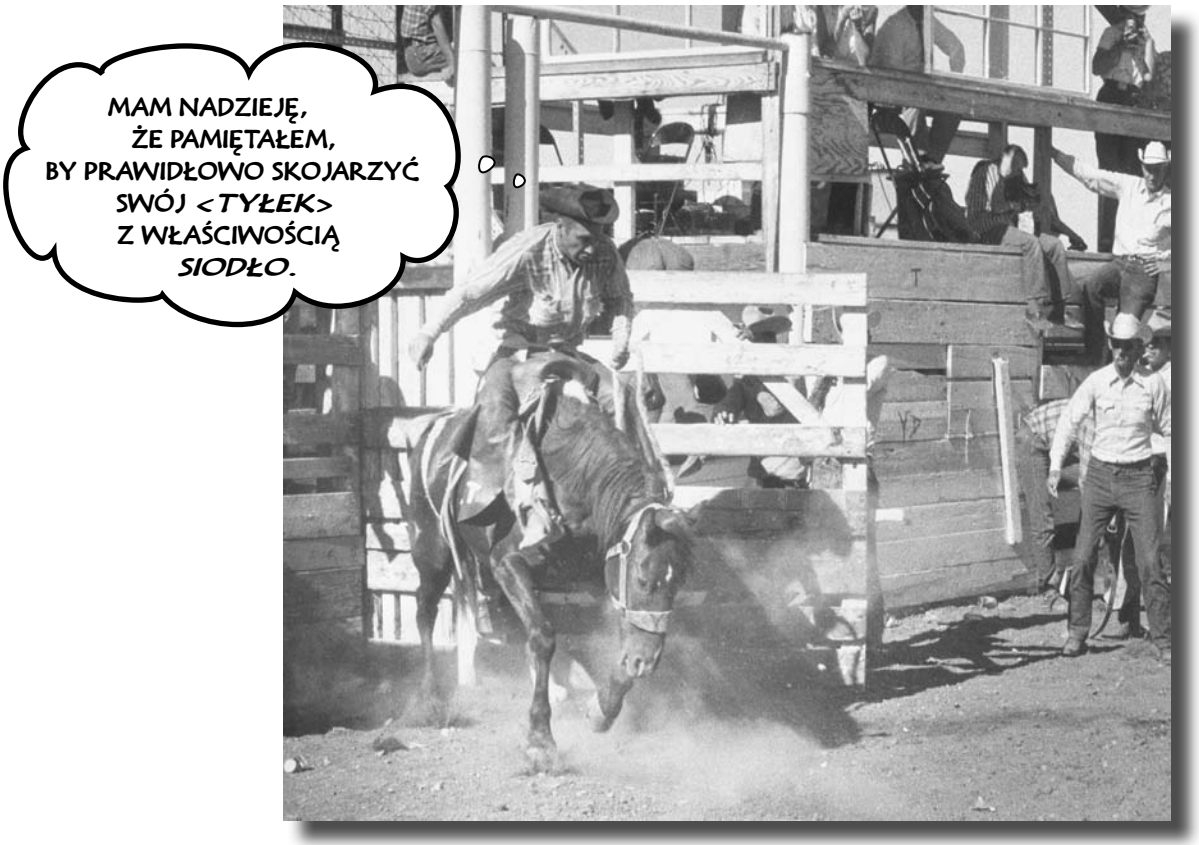
Zabawa dopiero się zaczyna! Pokazaliśmy Ci mnóstwo wspaniałych narzędzi do tworzenia naprawdę **potężnych programów** w C#. Nie było jednak możliwe, abyśmy w tej książce zmieścili **każde narzędzie, technologię i technikę** — nie ma ona po prostu tylu stron. Musieliśmy podjąć *naprawdę przemyślaną decyzję*, co umieścić, a co pominąć. Oto kilka tematów, których nie mogliśmy przedstawić. Pomimo tego, że nie zajęliśmy się nimi, w dalszym ciągu myślimy, że są one **ważne i przydatne**. Należałoby więc chociaż o nich wspomnieć. Tak też zrobiliśmy.







Przenieś swoje aplikacje na wyższy poziom

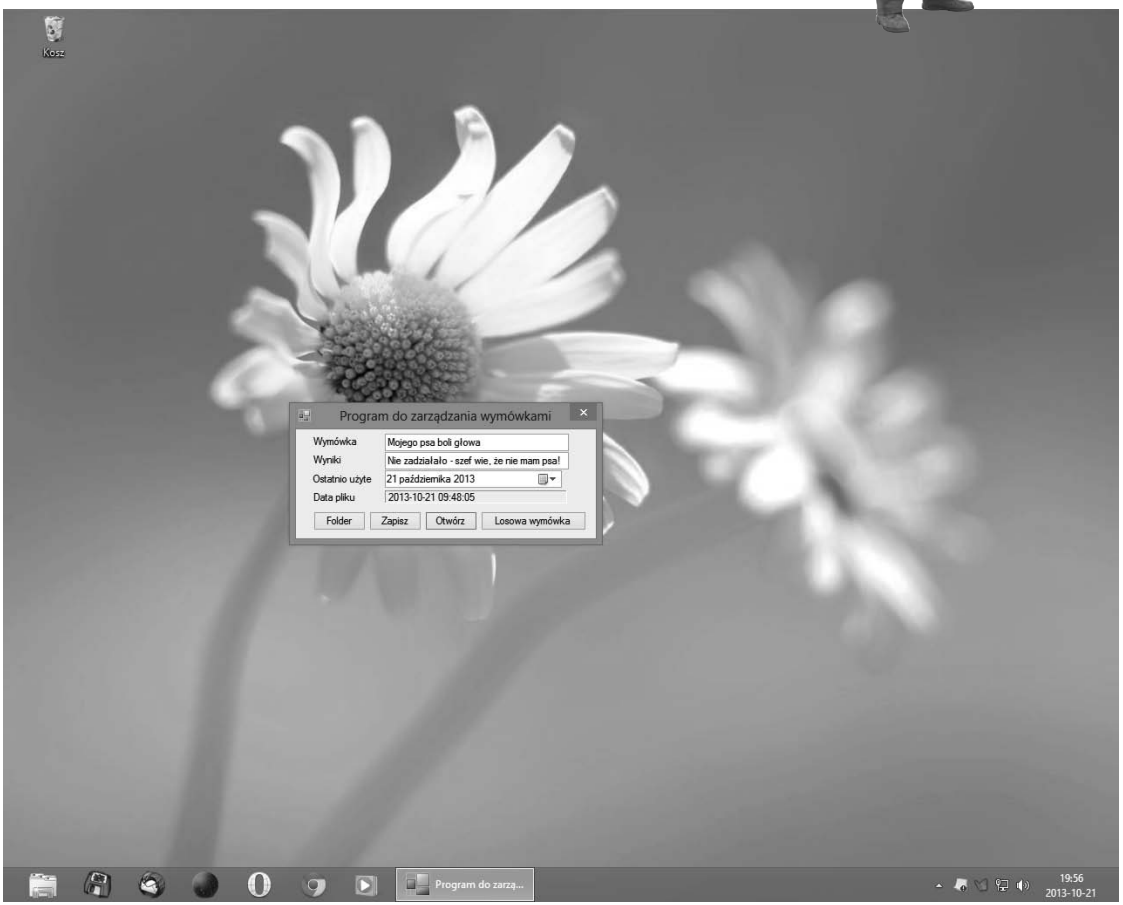


Jesteś już gotów, by wkroczyć do zupełnie nowego świata tworzenia aplikacji. Korzystanie z technologii WinForms do tworzenia klasycznych aplikacji dla systemu Windows jest doskonałym sposobem nauki ważnych rozwiązań języka C#, niemniej jednak możesz pójść *znacznie dalej*. W tym rozdziale dowiesz się, jak używać języka **XAML** do projektowania aplikacji przeznaczonych dla Sklepu Windows, nauczysz się tworzyć aplikacje **działające na dowolnych urządzeniach, integrować** dane ze stronami przy użyciu **wiązania danych** i używać Visual Studio do ujawniania tajemnic stron XAML poprzez badanie obiektów tworzonych na podstawie kodu XAML.

Damian używa Windows 8

Staromodne aplikacje Damiana dla systemu Windows wyglądają archaicznie! Damian ma już dosyć ciągłego klikania malutkich pól wyboru. A chciałby, żeby jego program do zarządzania wymówkami był aplikacją z prawdziwego zdarzenia. Czy możemy mu w tym pomóc?

Program do zarządzania wymówkami Damiana działa, jednak przestarzały klasyczny program dla systemu Windows nie jest żadną konkurencją dla prawdziwej, fantastycznej aplikacji, w stu procentach przeznaczonej dla Sklepu Windows.



Za kulisami

Zrób to!

Chcesz szybko wtłoczyć te pomysły do swojej głowy? Zatem **zrób opisane poniżej rzeczy**, zanim zaczniesz czytać ten rozdział!

Aplikacje przeznaczone dla Sklepu Windows są bardziej skomplikowane niż programy WinForms. Dlatego też programy WinForms są naprawdę efektywnym narzędziem do nauki, jednak nie radzą sobie równie dobrze, gdy chodzi o napisanie niezwykle odlotowej aplikacji. Czasami warto cofnąć się o krok i pomyśleć o tym, jak się uczymy, gdyż to pomoże nam robić to bardziej efektywnie. A zatem spróbujmy to zrobić teraz.

Stworzyłeś już sobie doskonałe podstawy, zdobywając podstawową wiedzę o języku C#, obiektach, kolekcjach i innych narzędziach .NET. Teraz jednak wrócimy do tworzenia aplikacji dla Sklepu Windows przy wykorzystaniu języka XAML. Na kilku następnych stronach spróbujemy skorzystać z możliwości IDE do badania obiektów, które tworzą i którymi zarządzają programy WinForms. Kiedy później użyjesz IDE do badania obiektów tworzonych w aplikacjach dla Sklepu Windows pisanych przy użyciu języka XAML, staraj się wychwytywać różnice — oraz, co jest równie ważne, podobieństwa pomiędzy nimi.

Wykonaj poniższe czynności, zanim zaczniesz dalszą lekturę tego rozdziału

Cały rozdział 1. i znaczną część rozdziału 2. przeznaczyliśmy na przedstawienie tworzenia aplikacji przeznaczonych dla Sklepu Windows przy wykorzystaniu języka XAML oraz platformy .NET Framework for Windows Store. Teraz wykorzystamy wiedzę zdobytą w tamtych dwóch rozdziałach. Jeśli chcesz w możliwie jak największym stopniu skorzystać z lektury tego rozdziału, sugerujemy, żebyś wykonał kilka przedstawionych poniżej czynności.

- Dla niektórych przesiadka z WinForms na stosowanie języka XAML jest całkowicie bezproblemowa, innych natomiast może drażnić. **Te czynności przygotowawcze pomogą Ci szybciej przyswoić sobie najważniejsze idee**.
- Wróć do rozdziału 1. i ponownie, od samego początku stwórz aplikację *Ratuj ludzi*. Tym razem upewnij się, że wpisałeś ręcznie cały kod.
- I nie zapomnij ***przeanalizować kodu aplikacji***! Wciąż znajdują się w nim fragmenty, którymi jeszcze nie zajmowaliśmy się szczegółowo, choć powinieneś już rozpoznawać je na tyle dobrze, byś mógł rozpocząć tworzenie mentalnych podstaw.
- Spróbuj dobrze zrozumieć tajniki działania gry. Nie staraj się jednak robić tego za wszelką cenę. Jak już wspominaliśmy — wciąż pozostało sporo zagadnień, którymi w tej książce jeszcze nie zajmowaliśmy się.
- Zwróć szczególną uwagę na zmiany wprowadzane w szablonie *Basic Page* oraz na sposób, w jaki zastąpiliśmy nim domyślną stronę *MainPage.xaml* — w tym rozdziale będziesz bowiem to robił kilka razy.
- **Jeszcze raz wykonaj** projekt XAML z rozdziału 2. Nie zapomnij zrobić także ćwiczenia. Teraz powinieneś być gotowy!

I jeszcze jedno...

W tej książce nie przedstawiliśmy wszystkich możliwości aplikacji WinForms. W rzeczywistości korzystają one z silnika graficznego określanego jako GDI+, potrafiącego generować zaskakująco dobry interfejs graficzny i materiały do druku oraz doskonale obsługiwać interakcję z użytkownikiem (choć wymaga ona znacznie większego nakładu pracy niż w przypadku korzystania z języka XAML). Jednym z najważniejszych sposobów nauki podstawowych zasad programowania jest przeanalizowanie tej samej rzeczy wykonanej na dwa różne sposoby.

Detektywistyczne poszukiwania!

Od czasu pierwszego projektu, *Ratuj ludzi*, przedstawionego w rozdziale 1., poznałeś już całkiem sporo ważnych pojęć związanych z językiem C# i nabyłeś dużo praktyki w posługiwaniu się nimi. Teraz nadszedł czas, by założyć kapelusz, wziąć lupę i przetestować swoje detektywistyczne umiejętności. Sprawdź, czy będziesz w stanie odszukać wszystkie elementy C# w kodzie aplikacji *Ratuj ludzi*. Aby ułatwić Ci początki poszukiwań, podaliśmy jedną z odpowiedzi. Czy jesteś w stanie znaleźć pozostałe?

(Na niektóre z pytań jest więcej niż jedna prawidłowa odpowiedź).

☐ **Zastosowanie łańcucha znaków jako klucza podczas pobierania obiektu z kolekcji `Dictionary`.**

.....

.....

.....

☐ **Zastosowanie inicjalizatora.**

.....

.....

.....

☐ **Dodanie do tej samej kolekcji obiektów dwóch różnych typów.**

.....

.....

.....

☐ **Wywołanie metody statycznej.**

.....

.....

.....

☐ **Zastosowanie metody procedury obsługi zdarzeń.**

.....

.....

.....

☐ **Użycie słowa kluczowego as w celu rzutowania obiektu w dół.**

.....

.....

.....

☐ **Przekazanie do metody referencji do obiektu.**

.....

.....

.....

☑ **Utworzenie instancji obiektu i wywołanie jednej z jego metod.**

W metodzie AnimateEnemy() tworzony jest obiekt klasy StoryBoard, a następnie wywoływana jest jego metoda Begin().

☐ **Użycie typu wyliczeniowego w celu przypisania wartości.**

.....

.....

.....

Detektywistyczne poszukiwania! Rozwiązanie

Od czasu pierwszego projektu, *Ratuj ludzi*, przedstawionego w rozdziale 1., poznałeś już całkiem sporo ważnych pojęć związanych z językiem C# i nabyłeś dużo praktyki w posługiwaniu się nimi. Teraz nadszedł czas, by założyć kapelusz, wziąć lupę i przetestować swoje detektywistyczne umiejętności. Sprawdź, czy będziesz w stanie odszukać wszystkie elementy C# w kodzie aplikacji *Ratuj ludzi*. Aby ułatwić Ci początki poszukiwań, podaliśmy jedną z odpowiedzi. Czy jesteś w stanie znaleźć pozostałe?

(Na niektóre z pytań jest więcej niż jedna prawidłowa odpowiedź).

Poniżej zamieściliśmy te, które nam udało się znaleźć; Ty mogłeś znaleźć zupełnie inne!

☑ **Zastosowanie łańcucha znaków jako klucza podczas pobierania obiektu z kolekcji `Dictionary`.**

W drugim wierszu metody AddEnemy() używany jest łańcuch znaków „EnemyTemplate", przy użyciu którego ze słownika Resources pobierany jest obiekt ControlTemplate.

☑ **Zastosowanie inicjalizatora.**

W metodzie AnimateEnemy() podczas inicjalizacji obiektu DoubleAnimation określane są wartości trzech właściwości: From, To oraz Duration.

☑ **Dodanie do tej samej kolekcji obiektów dwóch różnych typów.**

W metodzie StartGame() do kolekcji playArea.Children dodawany jest obiekt StackPanel (człowiek) oraz obiekt Rectangle (docelowy portal).

☑ **Wywołanie metody statycznej.**

W procedurze obsługi zdarzeń target_PointerEntered() wywoływane są dwie statyczne metody klasy Canvas: SetLeft() oraz SetTop().

☑ **Zastosowanie metody procedury obsługi zdarzeń.**

W oknie Properties została określona procedura obsługi zdarzeń PointerPressed obiektu StackPanel reprezentującego człowieka.

☑ **Użycie słowa kluczowego as w celu rzutowania obiektu w dół.**

Słownik Resources zwraca obiekt typu <object, object>, a zatem wartość odwołania Resources["EnemyTemplate"] jest rzutowana w dół na konkretny typ ControlTemplate.

☑ **Przekazanie do metody referencji do obiektu.**

Referencja do obiektu ContentControl jest przekazywana jako pierwszy parametr metody AnimateEnemy().

☑ **Utworzenie instancji obiektu i wywołanie jednej z jego metod.**

W metodzie AnimateEnemy() tworzony jest obiekt klasy StoryBoard, a następnie wywoływana jest jego metoda Begin().

☑ **Użycie typu wyliczeniowego w celu przypisania wartości.**

W metodzie EndTheGame() jest używany typ wyliczeniowy Visibility, a konkretnie właściwości startButton.Visibility jest przypisywana wartość Visibility.Visibile.

Technologia Windows Forms korzysta z grafu obiektów stworzonego przez IDE

Podczas tworzenia klasycznej aplikacji przeznaczonej dla systemu Windows IDE przygotowuje formularz i generuje jego kod, który umieszcza w pliku *Form1.Designer.cs*. Ale co właściwie jest umieszczane w tym pliku? Nie jest to bynajmniej nic tajemniczego. Już wiesz, że wszystkie kontrolki umieszczane na formularzu są obiektami, wiesz także, że referencje do obiektów można zapisywać w polach. A zatem w jakimś miejscu wygenerowanego kodu musi się znaleźć deklaracja pola reprezentującego każdy z obiektów formularza, kod służący do utworzenia tego obiektu oraz kod do wyświetlenia go na formularzu. Spróbujmy odszukać te wiersze kodu, byśmy mogli dokładnie sprawdzić, co się dzieje w formularzu.

1. Otwórz projekt *Prosty edytor tekstów*, który napisałeś w rozdziale 9., i otwórz plik *Form1.Designer.cs*. Przewiń jego zawartość w dół i odszukaj deklaracje pól. Powinieneś ujrzeć po jednej deklaracji pola dla każdej kontrolki umieszczonej na formularzu.

```
Windows Form Designer generated code

private System.Windows.Forms.OpenFileDialog openFileDialog1;
private System.Windows.Forms.SaveFileDialog saveFileDialog1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button open;
private System.Windows.Forms.Button save;
private System.Windows.Forms.TableLayoutPanel tableLayoutPanel1;
private System.Windows.Forms.FlowLayoutPanel flowLayoutPanel1;
```

2. Rozwiń sekcję kodu wygenerowanego przez projektanta formularzy i odszukaj kod tworzący kontrolki:

IDE mogło wygenerować te wiersze kodu w nieco innej kolejności, niemniej jednak na pewno znajdzie się tu po jednym wierszu kodu dla każdej kontrolki umieszczonej na formularzu.

```
private void InitializeComponent()
{
    this.openFileDialog1 = new System.Windows.Forms.OpenFileDialog();
    this.saveFileDialog1 = new System.Windows.Forms.SaveFileDialog();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.open = new System.Windows.Forms.Button();
    this.save = new System.Windows.Forms.Button();
    this.tableLayoutPanel1 = new System.Windows.Forms.TableLayoutPanel();
    this.flowLayoutPanel1 = new System.Windows.Forms.FlowLayoutPanel();
```

3. Kontrolki takie jak `TableLayoutPanel` oraz `FlowLayoutPanel`, mogące zawierać inne kontrolki, dysponują właściwością o nazwie **`Controls`**. Jest to obiekt typu `ControlCollection`, który pod bardzo wieloma względami przypomina obiekt typu `List<Control>`. Każda kontrolka umieszczana na formularzu jest obiektem klasy pochodnej klasy `Control`, a dodanie jej do kolekcji `Controls` panelu spowoduje, że zostanie ona wyświetlona wewnątrz niego. Przewiń zawartość pliku w dół, do miejsca, w którym do panelu `FlowLayoutPanel` dodawane są przyciski *Zapisz* i *Otwórz*:

```
this.flowLayoutPanel1.Controls.Add(this.save);
this.flowLayoutPanel1.Controls.Add(this.open);
```

Panel `FlowLayoutPanel` został z kolei umieszczony w komórce panelu `TableLayoutPanel`. Odszukaj miejsce kodu, w którym on oraz pole tekstowe są dodawane do panelu zewnętrznego:

```
this.tableLayoutPanel1.Controls.Add(this.textBox1, 0, 0);
this.tableLayoutPanel1.Controls.Add(this.flowLayoutPanel1, 0, 1)
```

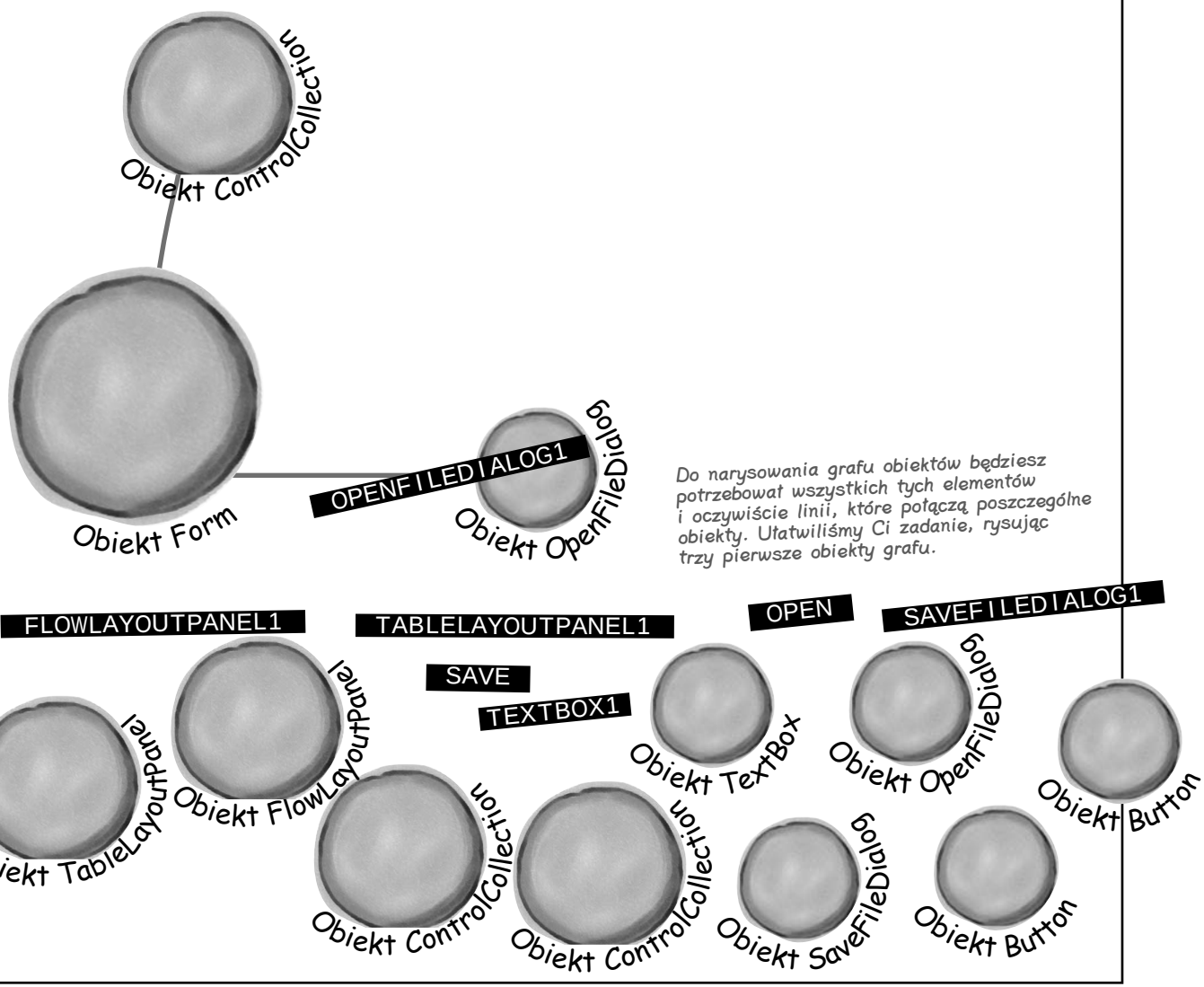
Także sam obiekt `Form` jest pojemnikiem, do którego zostaje dodany obiekt panelu `TableLayoutPanel`:

```
this.Controls.Add(this.tableLayoutPanel1);
```

Będziesz musiał otworzyć plik Form1.Designer.cs z projektu Prostego edytora tekstów, który napisałeś w rozdziale 9.

Zaostrz ołówek

Przyjrzyj się instrukcjom new oraz wywołaniom metody `Controls.Add()` wygenerowanym przez IDE w formularzu Prostego edytora tekstu i narysuj graf obiektów tworzonych podczas wykonywania tego kodu.

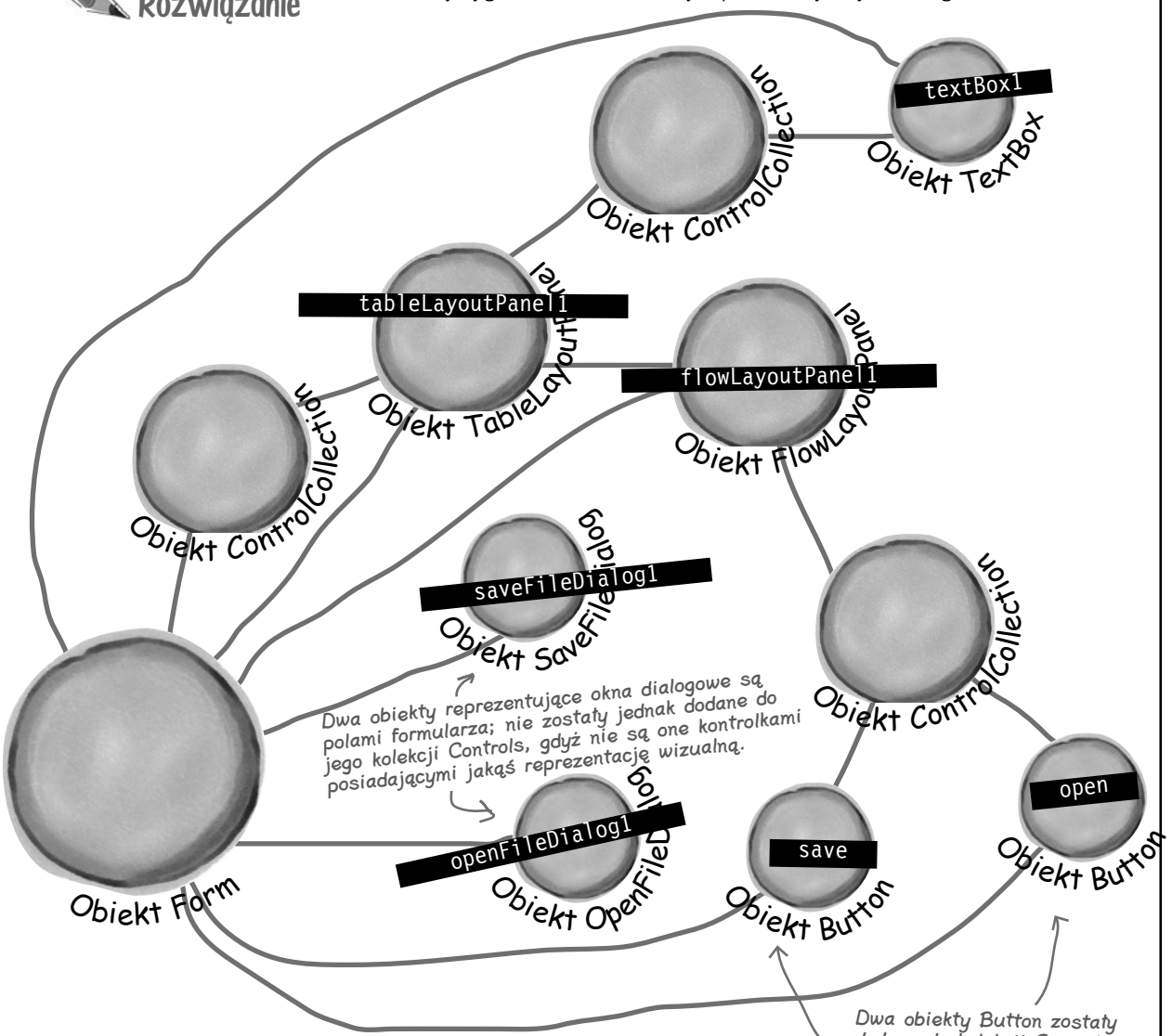


Do narysowania grafu obiektów będziesz potrzebował wszystkich tych elementów i oczywiście linii, które połączą poszczególne obiekty. Ułatwiliśmy Ci zadanie, rysując trzy pierwsze obiekty grafu.

Zaostrz ołówek

Rozwiązanie

Przyjrzyj się instrukcjom new oraz wywołaniom metody `Controls.Add()` wygenerowanym przez IDE w formularzu Prostego edytora tekstu i narysuj graf obiektów tworzonych podczas wykonywania tego kodu.



To Ci się naprawdę przyda, kiedy będziesz wykonywał ćwiczenia. Przyjrzyj się także drugiej metodzie dostępnej w klasie Debug.

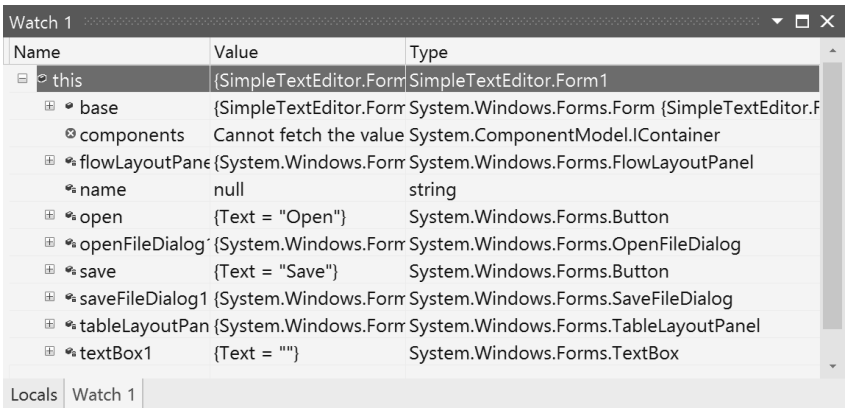
Tools

Podpowiedź do debugowania

Metoda `System.Diagnostics.Debug.WriteLine()` pozwala wyświetlać tekst w oknie wyników w trakcie sesji debugowania. Można jej używać tak samo jak metody `Console.WriteLine()` w aplikacjach Windows Forms.

Użyj IDE do przejrzenia grafu obiektów

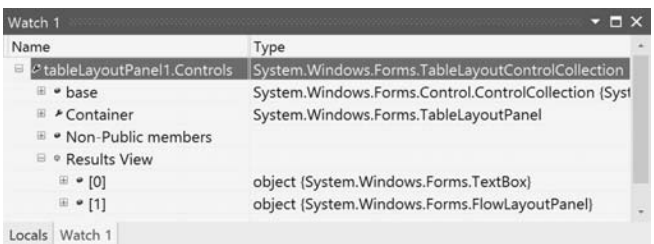
Wróć do projektu Prostego edytora tekstu i umieść punkt przerwania na wywołaniu metody `InitializeComponent()`, w konstruktorze formularza. Następnie uruchom debugowanie programu. Gdy zostanie ono przerwane w punkcie przerwania, naciśnij klawisz *F10*, by wejść do metody, po czym **przejdź do okna *Watch* i wpisz `this`**, by wyświetlić i przejrzeć graf obiektów.



Kliknij przycisk „+” widoczny z lewej strony „this”, by rozwinąć zawartość obiektu i wyświetlić wszystkie jego pola, wygenerowane przez IDE w celu przechowywania referencji do kontrolek umieszczonych na formularzu.

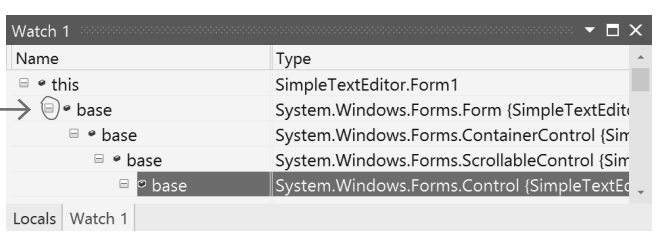
Dodaj do okna *Watch* pole `tableLayoutPanel1.Controls` i rozwiń pozycję *Results View*, by zobaczyć obiekty dodane do panelu:

Kolekcja Controls kontrolki TableLayouPanel zawiera dwie kontrolki: TextBox oraz FlowLayoutPanel zawierającą przyciski Zapisz i Otwórz.



Okazuje się, że klasa `System.Windows.Form` nie posiada właściwości `Controls`. Właściwość ta jest dziedziczona po jej klasie bazowej, `ContainerControl`, która z kolei dziedziczy ją po swojej klasie bazowej, `ScrollableControl`, a ta od swojej klasy bazowej `Control`. Rozwijaj pozycję ⊞ base w oknie *Watch*, aby przejść hierarchię dziedziczenia, docierając aż do klasy `System.Windows.Forms.Control`. (To właśnie po niej klasa `Form` dziedziczy kolekcję `Controls`). Następnie rozwiń pozycję *Results View* kolekcji `Controls`. Znajdziesz w niej po jednym obiekcie dla każdej kontrolki umieszczonej na formularzu!

Rozwiń pozycję „base”, by wyświetlić właściwości, które obiekt odziedziczył po swojej klasie bazowej:

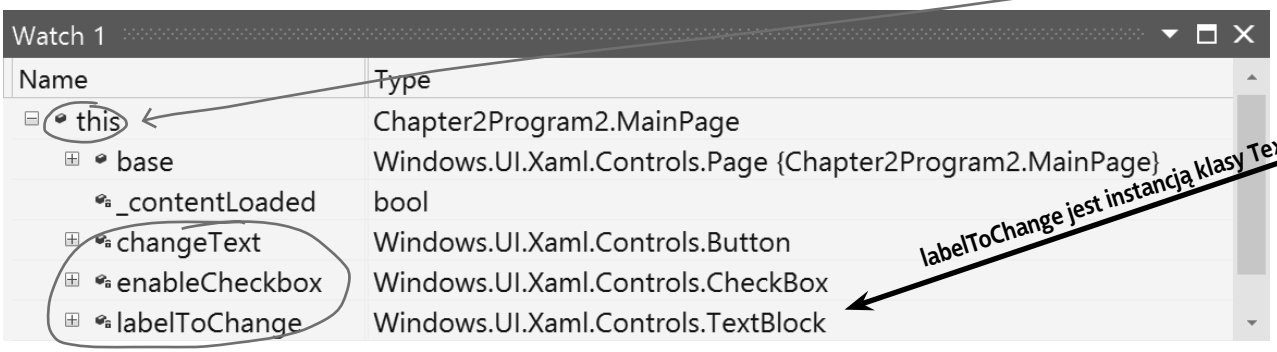


To już jest Twoje ostatnie spotkanie z aplikacjami WinForms na kilka najbliższych rozdziałów! Wrócimy do nich jeszcze kilka razy, gdyż są one naprawdę doskonałymi narzędziami do nauki i poznawania języka C#.

Aplikacje dla Sklepu Windows używają XAML do tworzenia obiektów interfejsu użytkownika

Używając kodu XAML do tworzenia interfejsu użytkownika aplikacji przeznaczonych dla Sklepu Windows, w rzeczywistości tworzymy graf obiektów. Podobnie jak w aplikacjach WinForms, także i w tym przypadku można skorzystać z IDE oraz jego okna *Watch* do przeglądnięcia tych obiektów. Otwórz program, którego w **rozdziale 2. używaliśmy do „zabaw z instrukcją if-else”**. Następnie otwórz plik *MainPage.xaml.cs*, umieść punkt przerwania w konstruktorze, w wierszu zawierającym wywołanie metody `InitializeComponent()`, a następnie **skorzystaj z możliwości IDE, by zbadać utworzone w aplikacji obiekty interfejsu użytkownika**.

1 Uruchom debugowanie, a następnie naciśnij klawisz *F10*, aby wejść do metody `InitializeComponent()`. Visual Studio 2012 for Windows 8 posiada nieco inny układ okien od Visual Studio przeznaczonego do tworzenia tradycyjnych aplikacji, gdyż dysponuje większymi możliwościami; na przykład pozwala na otwieranie wielu okien *Watch* (co przydaje się, kiedy chcemy śledzić większą liczbę danych). Wyświetl to okno, wybierając z menu opcję *DEBUG/Windows/Watch/Watch 1*, a następnie wpisz w nim `this`:



2 A teraz przyjrzyj się kodowi XAML definiującemu postać strony:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Kliknięcie zmienia etykietę"
        HorizontalAlignment="Center" Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Włącza zmianę etykiety"
          HorizontalAlignment="Center" IsChecked="true" Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
           Text="Naciśnij przycisk, aby zmienić tekst"
           HorizontalAlignment="Center" VerticalAlignment="Center"
           Grid.ColumnSpan="2"/>
</Grid>
```

Kod XAML definiujący kontrolki umieszczone na stronie zostaje przekształcony w obiekt Page, którego pola i właściwości zawierają referencje do kontrolek interfejsu użytkownika.

3 Dodaj do okna *Watch* kilka właściwości kontrolki `labelToChange`:

Watch 1

Name	Value	Type
labelToChange.Text	"Naciśnij przycisk, aby zmienić tekst"	string
labelToChange.HorizontalAlignment	Center	Windows.UI.Xaml.HorizontalAlig
labelToChange.VerticalAlignment	Center	Windows.UI.Xaml.VerticalAlignm
labelToChange.TextWrapping	Wrap	Windows.UI.Xaml.TextWrapping

Aplikacja automatycznie określi wartości właściwości na podstawie kodu XAML:

```
<TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
       Text="Naciśnij przycisk, aby zmienić tekst"
       HorizontalAlignment="Center" VerticalAlignment="Center"
       Grid.ColumnSpan="2"/>
```

Spróbuj jednak dodać do okna *Watch* właściwość `labelToChange.Grid` lub `labelToChange.ColumnSpan`. Kontrolka `labelToChange` jest typu `Windows.UI.Controls.TextBlock`, a klasa ta nie deklaruje żadnej z tych właściwości. Czy jesteś w stanie odgadnąć, co się dzieje w tym kodzie XAML?

4 Zatrzymaj program, otwórz plik *MainPage.xaml.cs* i odszukaj w nim deklarację klasy `MainPage`. Przyjrzyj się jej deklaracji — jak widać, dziedziczy ona po klasie `Page`. Umieść wskaźnik myszy nad słowem `Page`, tak by IDE wyświetliło pełną nazwę klasy:

Przesuń wskaźnik myszy nad słowo Page, by wyświetlić nazwę klasy.

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
    }
```

class Windows.UI.Xaml.Controls.Page
Encapsulates a page of content that can be navigated to.

Teraz ponownie uruchom program i naciśnij klawisz *F10*, by wejść do metody `InitializeComponent()`. Przejdź do okna *Watch* i rozwiń elementy `this`, następnie `base` i jeszcze raz `base`, przechodząc tym samym nieco w górę hierarchii dziedziczenia.

Watch 1

Name	Type
this	Chapter2Program2.MainPage
base	Windows.UI.Xaml.Controls.Page {Chapter2Program2.MainPage}
base	Windows.UI.Xaml.Controls.UserControl {Chapter2Program2.MainPa
base	Windows.UI.Xaml.Controls.Control {Chapter2Program2.MainPage}
Content	Windows.UI.Xaml.UIElement {Windows.UI.Xaml.Controls.Grid}
Static members	

Rozwiń te elementy, by wyświetlić klasy bazowe.

Rozwiń element Content i sprawdź jego węzeł [Windows.UI.Xaml.Controls.Grid].

Poświęć chwilę, by dokładnie zbadać obiekty wygenerowane na podstawie kodu XAML. Przyjrzymy się im nieco dokładniej w dalszej części książki. Na razie po prostu je przejrzyj, byś uświadomił sobie, jak wiele obiektów tworzy Twoją aplikację.

Przeprojektuj formularz Idź na ryby!, zmieniając go w aplikację dla Sklepu Windows

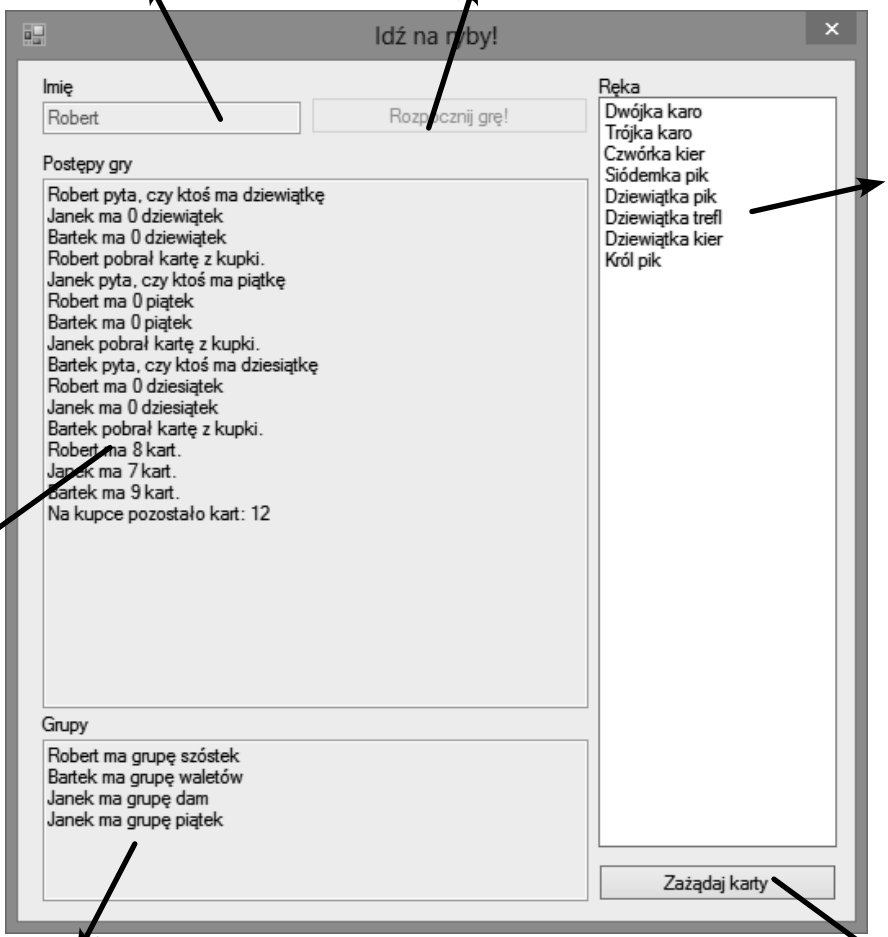
Gra Idź na ryby!, którą napisałeś w rozdziale 8., byłaby fantastyczną aplikacją dla Sklepu Windows. Uruchom zatem Visual Studio 2012 for Windows 8 i utwórz w nim nowy projekt **Windows Store** (taki sam jak w przypadku aplikacji *Ratuj ludzi*). Na kilku następnych stronach przeprojektujesz ten formularz w formie strony XAML, która będzie potrafiła dostosować się do urządzeń z ekranami o różnych rozmiarach. Zamiast korzystać z kontrolek Windows Forms umieszczanych na formularzu, skorzystasz z kontrolek charakterystycznych dla aplikacji przeznaczonych dla Sklepu Windows, umieszczanych na stronie.

Zrób to!

To zostaje przekształcone w element <TextBox/>

To zostaje przekształcone w element <Button/>

Użyjemy panelu StackPanel o układzie poziomym, aby zgrupować kontrolki TextBox oraz Button, tak by można je było wyświetlić w jednej komórce układu.



To zostaje przekształcone w element <ListBox/>

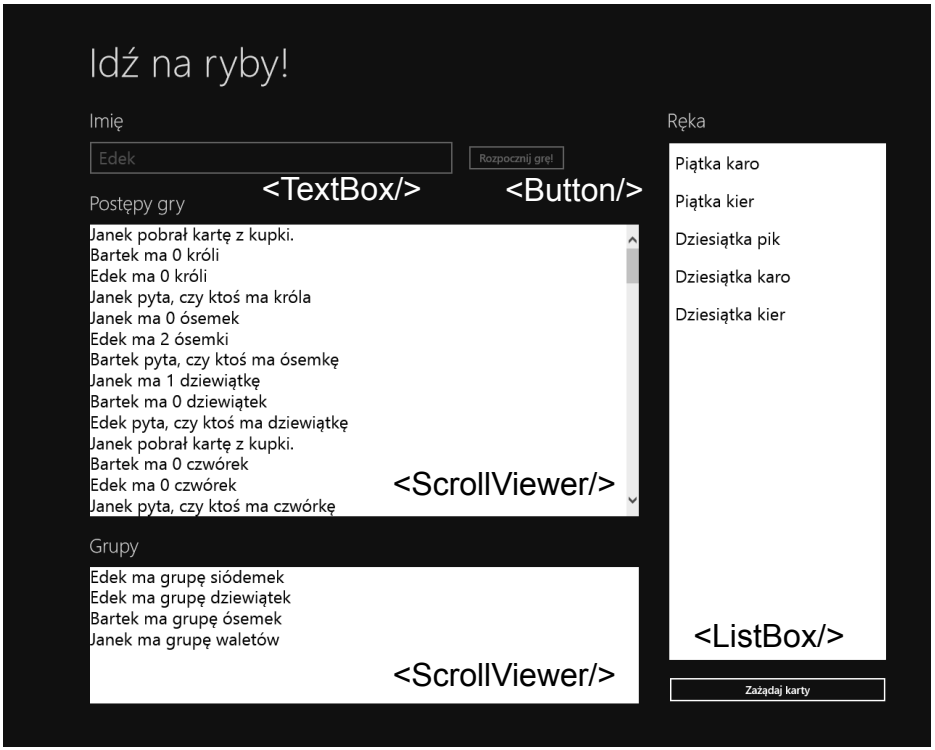
To zostaje przekształcone w element <ScrollViewer/>

To zostaje przekształcone w element <ScrollViewer/>

To jest kolejna kontrolka dostępna w przyborniku. Prezentuje ona sekwencję łańcuchów znaków, dodając do nich pionowe i poziome paski przewijania, jeśli tekst będzie większy od rozmiarów kontrolki.

To zostaje przekształcone w element <Button/>

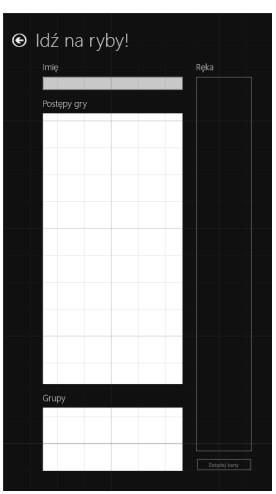
Tak te kontrolki będą wyglądać na głównej stronie aplikacji:

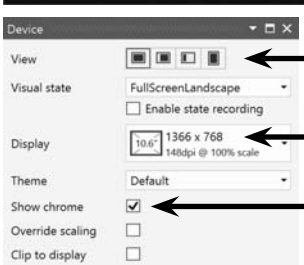


Większość kodu obsługującego przebieg gry pozostanie w niezmienionej postaci, zmieni się natomiast kod obsługujący interfejs aplikacji.

Kontrolki te zostaną umieszczone w siatce, której wiersze i kolumny będą się zmniejszać i powiększać w zależności od wielkości ekranu. Dzięki temu gra będzie mogła się zmniejszać i powiększać, dostosowując się do ekranu. Do określania różnych konfiguracji ekranu możesz skorzystać z okna *Device* dostępnego w IDE.

Grą będzie się można bawić niezależnie od rozmiarów strony.





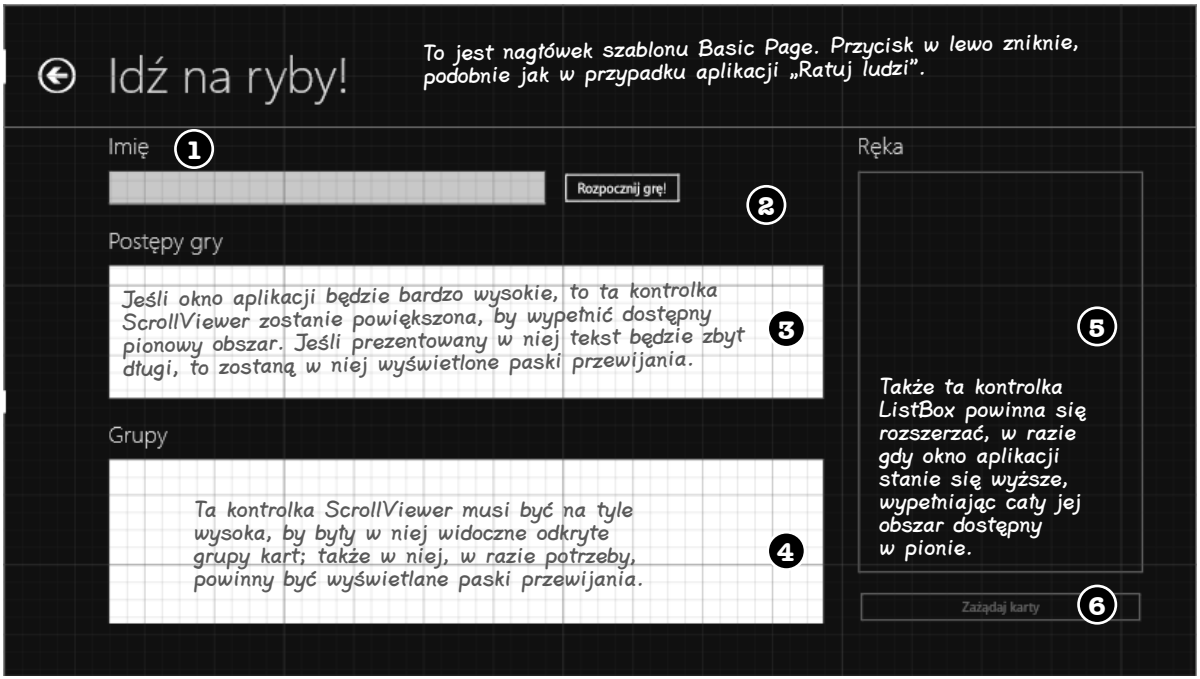
Przyciski View pozwalają wyświetlić stronę w układzie poziomym, pionowym oraz w układzie podzielonego ekranu.

Różne opcje dostępne na liście Display pozwalają wyświetlać stronę przy wykorzystaniu różnych rozdzielczości oraz proporcji ekranu.

Usuń zaznaczenie pola „Show chrome", by ukryć obrazek reprezentujący oprawę ekranu urządzenia.

Określanie postaci strony rozpoczyna się od dodania kontrolek

Technologie XAML oraz WinForms mają jedną wspólną cechę: w obu do określania postaci interfejsu użytkownika używane są kontrolki. Strona gry Idź na ryby! posiada dwa przyciski, kontrolkę `ListBox` służącą do wyświetlania kart w ręce, pole `TextBox`, w którym użytkownik może wpisać swoje imię, oraz cztery etykiety `TextBlock`. Interfejs użytkownika dopełniają dwie kontrolki `ScrollViewer` o białym tle, służące do prezentowania postępów gry oraz odłożonych grup.



Szablon *Basic Page* zawiera siatkę składającą się z dwóch wierszy. Jej górny wiersz zawiera nagłówek z nazwą aplikacji. Z kolei w drugim wierszu umieszczony jest obszar treści, zdefiniowany przez poniższą siatkę. Cała ta siatka jest umieszczona w wierszu 1. układu (oraz w jego jedynej kolumnie o numerze 0).

```
<Grid Grid.Row="1" Margin="120,0,60,60">
    <TextBlock Text="Imię" Margin="0,0,0,20"
1       Style="{StaticResource SubheaderTextStyle}"/>
```

Oto znacznik otwierający, rozpoczynający siatkę.

Marginesy określają wcięcie siatki, dzięki czemu jest odsunięta od krawędzi strony. Lewy margines zawsze wynosi 120 pikseli.

Zastosujemy kontrolkę typu `StackPanel`, aby w jednej komórce siatki umieścić pole `TextBox` do podania nazwy gracza oraz przycisk rozpoczynający grę:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
    <TextBox x:Name="playerName" FontSize="24"
        Width="500" MinWidth="300" />

2   <Button x:Name="startButton" Margin="20,0"
        Content="Rozpocznij grę!"/>
</StackPanel>
```

Ta właściwość dodaje wokół kontrolek TextBox i Button margines o szerokości 20 pikseli. Kiedy właściwość Margin zawiera dwie liczby, określają one, odpowiednio: marginesy w poziomie (lewy i prawy) oraz w pionie (górny i dolny).

Każda etykieta umieszczona na stronie („Imię”, „Postępy gry” itd.) jest kontrolką `TextBlock` posiadającą niewielki margines u góry oraz u dołu, jak również określoną wartość właściwości `SubHeaderTextStyle`:

```
<TextBlock Text="Postępy gry"
    Style="{StaticResource SubheaderTextStyle}"
    Margin="0,20,0,20" Grid.Row="2"/>
```

Kontrolka `ScrollViewer` wyświetla postępy gry i dysponuje paskami przewijania, które zostaną wyświetlone, kiedy tekst stanie się zbyt długi i przestanie się mieścić w obszarze kontrolki:

3
```
<ScrollViewer Grid.Row="3" FontSize="24"
    Background="White" Foreground="Black" />
```

Oto kolejne kontrolki, `TextBlock` oraz `ScrollViewer`, służące do wyświetlania grup kart. Domyślną wartością wyrównania kontrolek `ScrollViewer` w pionie oraz w poziomie jest `Stretch` i okaże się, że jest ona naprawdę przydatna. Skonfigurujemy wiersze i kolumny siatki w taki sposób, by kontrolki `ScrollViewer` rozszerzały się, dostosowując się do ekranów o różnych wielkościach.

```
<TextBlock Text="Grupy" Style="{StaticResource SubheaderTextStyle}"
    Margin="0,20,0,20" Grid.Row="4"/>
```

4
```
<ScrollViewer FontSize="24" Background="White" Foreground="Black"
    Grid.Row="5" Grid.RowSpan="2" />
```

Pośrodku siatki dodaliśmy kolumnę o szerokości 40 pikseli, by zapewnić odstęp pomiędzy kontrolkami w obu kolumnach. Oznacza to także, że pozostałe kontrolki — `ListBox` oraz `Button` — muszą zostać umieszczone w trzeciej kolumnie. Kontrolka `ListBox` zajmuje wiersze do 2. do 6., a zatem zastosowaliśmy w niej właściwości `Grid.Row="1"` oraz `Grid.RowSpan="5"`. Dzięki temu rozwiązaniu kontrolka ta będzie się powiększała i zmniejszała, dostosowując do wielkości strony.

5
```
<TextBlock Text="Ręka" Style="{StaticResource SubheaderTextStyle}"
    Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>

<ListBox x:Name="cards" Background="White" FontSize="24" Height="Auto"
    Margin="0,0,0,20" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"/>
```

Pamiętaj, że numeracja wierszy i kolumn rozpoczyna się od zera, a zatem w przypadku kontrolek umieszczonych w trzeciej kolumnie należy użyć właściwości Grid.Column="2".

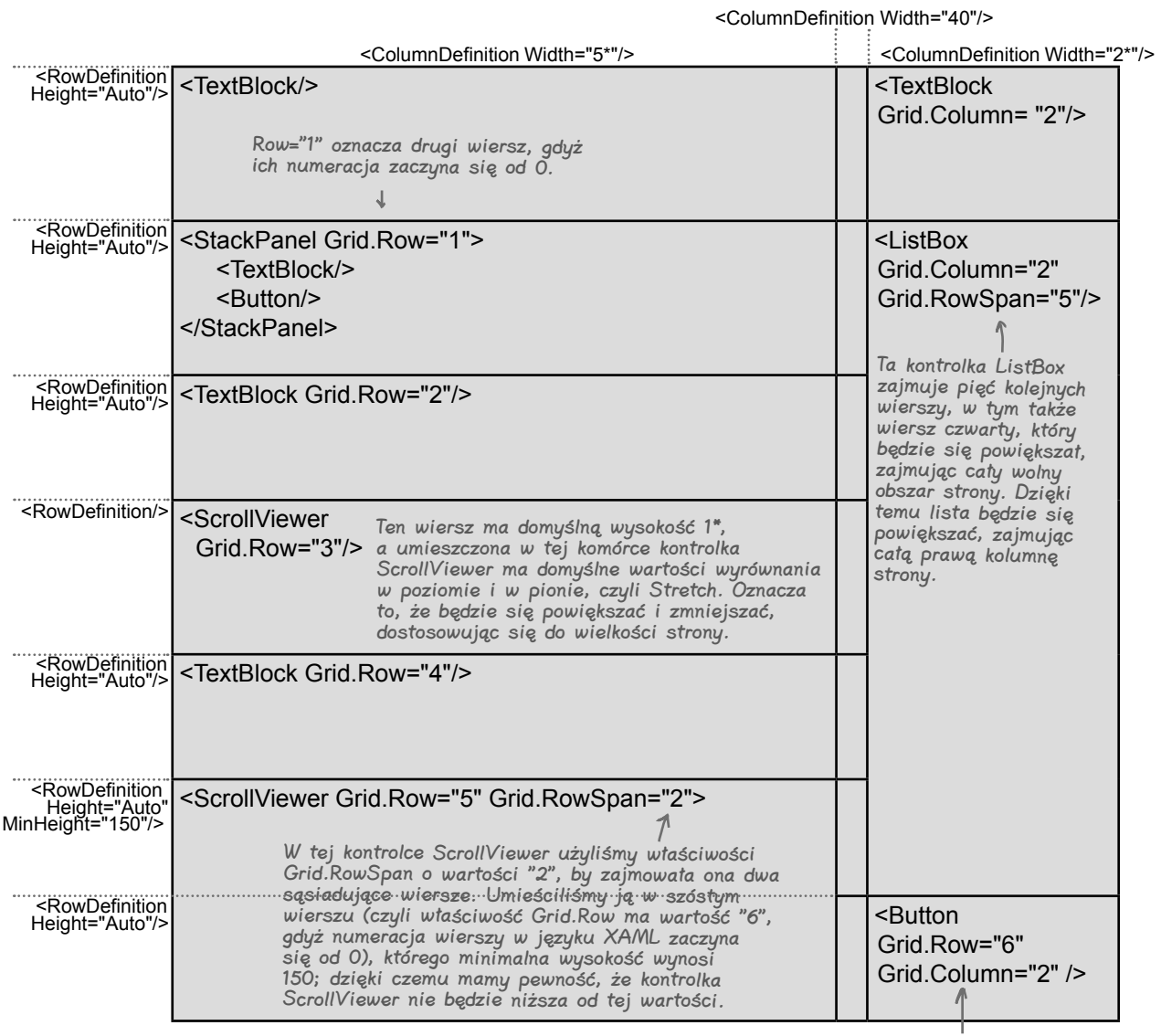
W przycisku *Zażądaj karty*, we właściwościach określających wyrównanie w pionie i w poziomie, zastosowaliśmy wartość `Stretch`, dzięki czemu wypełni on cały obszar komórki. Dodany do listy margines dolny o wysokości 20 pikseli zapewni niewielki odstęp pomiędzy listą i przyciskiem.

6
```
<Button x:Name="askForACard" Content="Zażądaj karty"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Grid.Row="6" Grid.Column="2"/>
```

Tworzenie siatki zakończymy na następnej stronie. → jesteś tutaj ▸

Wiersze i kolumny mogą zmieniać wielkość, dostosowując się do rozmiarów strony

Siatki są niezwykle efektywnym narzędziem do określania układu stron, gdyż ułatwiają projektowanie stron, które mogą być wyświetlane na wielu różnych urządzeniach. Wysokości oraz szerokości zakończone znakiem * **są automatycznie dostosowywane** do ekranów o różnej geometrii. Strona gry Idź na ryby! składa się z trzech kolumn. Pierwsza ma szerokość 5*, a trzecia 2*, a zatem będą się **proporcjonalnie** kurczyły i rozszerzały, zachowując zawsze proporcje 5:2. Druga kolumna ma ustaloną szerokość wynoszącą 40 pikseli i służy do wizualnego oddzielenia dwóch pozostałych kolumn. Poniżej zamieściliśmy schemat przedstawiający układ wierszy i kolumn strony (oraz kontrolki umieszczone w komórkach siatki):



Numeracja wierszy i kolumn w języku XAML rozpoczyna się od 0, dlatego też w tej kontrolce Button wiersz został określony jako 6, a kolumna jako 2 (aby pominąć kolumnę środkową). Właściwości określające wyrównanie kontrolki w pionie i poziomie mają wartość Stretch, dzięki czemu przycisk zajmie cały obszar komórki. Wysokość wiersza została określona jako Auto, dzięki czemu zostanie ona dostosowana do wymiarów zawartości (czyli przycisku oraz jego marginesów).

Oto jak wygląda definicja wierszy i kolumn dla takiego układu strony:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="5*"/>
    <ColumnDefinition Width="40"/>
    <ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto" MinHeight="150" />
    <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

Pierwsza kolumna zawsze będzie 2,5 razy szersza od trzeciej (proporcja 5:2), a obie będą od siebie oddzielone drugą kolumną o stałej szerokości 40 pikseli. W kontrolkach `ScrollViewer` oraz `ListBox`, używanych do prezentacji danych, właściwości `HorizontalAlignment` przypisano wartość `"Stretch"`, dzięki czemu wypełnią one całą szerokość kolumn.

Czwarty wiersz ma domyślną wysokość 1*, dzięki czemu będzie się powiększał i zmniejszał, zajmując cały dostępny obszar, który nie został zajęty przez pozostałe wiersze. Kontrolki `ListBox` oraz `ScrollViewer` są wyświetlone także w tym wierszu, dzięki czemu także one będą się powiększać i zmniejszać.

Wysokość niemal wszystkich wierszy ma wartość `Auto`. Tylko jeden wiersz siatki będzie się powiększał i zmniejszał i wszystkie kontrolki wyświetlone w tym wierszu będą się zachowywały podobnie.

Definicje wierszy i kolumn można dodawać wewnątrz siatki, powyżej lub poniżej tego kodu. Tym razem dodaliśmy je poniżej.

```
</Grid>
```

Oto znacznik zamykający siatki. Cały kod XAML strony połączysz w całość pod koniec rozdziału, kiedy zakończysz przerabianie gry na aplikację dla Sklepu Windows.

Skorzystaj z systemu siatki, by określić układ stron aplikacji

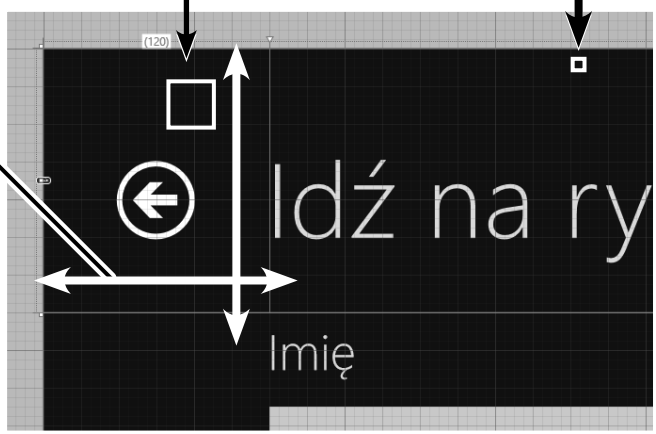
Jeśli już w rozdziale 1. nie użyłeś tych przycisków, umieszczonych u dołu okna Designer i służących, odpowiednio, do wyświetlania linii siatki, przyciągania oraz przyciągania do siatki, to skorzystaj z nich teraz.

Czy kiedykolwiek zauważyłeś, że różne aplikacje przeznaczone dla Sklepu Windows mają podobny wygląd? Dzieje się tak dlatego, że korzystają z **systemu siatki**, by nadać aplikacjom coś, co projektanci z firmy Microsoft określają mianem „spójnego zarysu". Siatka składa się z kwadratów, nazywanych *jednostkami* (ang. *units*) oraz *podjednostkami* (ang. *subunits*) — miałeś już okazję się z nimi spotkać, gdyż są one wykorzystywane w Visual Studio IDE.

Siatka składa się z kwadratów o wielkości 20x20 pikseli, nazywanych jednostkami.

Każda jednostka jest podzielona na podjednostki o wielkości 5x5 pikseli.

Nagłówek strony powinien mieć wysokość 7 jednostek, tekst powinien się zaczynać 6 jednostek od lewej krawędzi strony, a jego dolna krawędź powinna się znajdować na wysokości 5 jednostek poniżej górnej krawędzi.



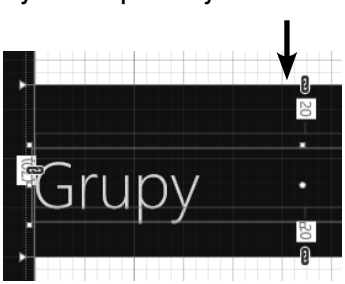
Wiersz nagłówka jest automatycznie dodawany do szablonu Basic Page, który zastosujesz w dalszej części rozdziału, kiedy gra Idź na ryby! będzie już działać.

W aplikacji Idź na ryby! w kontrolce `<Grid>` zawierającej wszystkie pozostałe kontrolki aplikacji skorzystamy z właściwości `Margin`, by stworzyć odstęp od krawędzi strony. Wartością tej właściwości może być: jedna liczba (określająca wielkość wszystkich czterech marginesów: lewego, górnego, prawego i dolnego), dwie liczby (pierwsza z nich określa wielkość marginesu lewego i prawego, a druga: górnego i dolnego) bądź też cztery liczby określające, odpowiednio, wielkość marginesu lewego, górnego, prawego i dolnego.

Twoja aplikacja Idź na ryby! ma lewy margines o wielkości 120 pikseli (czyli 6 jednostek), nie ma górnego marginesu, natomiast prawy i dolny margines mają wielkość 60 pikseli (3 jednostek):

```
<Grid Grid.Row="1" Margin="120,0,60,60">
```

Strona używa marginesów, by dodać odstęp o wielkości jednej jednostki pomiędzy etykietami i pozostałymi elementami.



Oprócz tego pomiędzy etykietami oraz innymi elementami strony także zostały dodane marginesy o wielkości jednej jednostki:

```
<TextBlock Text="Books"
           Style="{StaticResource SubheaderTextStyle}"
           Margin="0,20,0,20" Grid.Row="4"/>
```

Nie istnieją głupie pytania

P: Jaki jest efekt przypisania wysokości wiersza lub szerokości kolumny wartości „Auto”?

O: Przypisanie wartości `Auto` właściwości `Height` wiersza lub właściwości `Width` kolumny powoduje, że wielkość tego wiersza lub kolumny będzie się zmieniać i dostosowywać do wymiarów jego zawartości. Samemu możesz spróbować, jak to działa. Utwórz nowy projekt Blank App i zmodyfikuj siatkę na stronie *MainPage.xaml*, dodając do niej kilka wierszy i kolumn, których wysokości i szerokości zostały określone jako `Auto`. W oknie *Designer* nie zobaczysz niczego, gdyż wiersze i kolumny są puste, więc zostały zmniejszone do wysokości i szerokości 0 pikseli. Kiedy jednak dodasz jakieś kontrolki do różnych wierszy i kolumn, przekonasz się, że powiększyły się one, dostosowując się do wymiarów tych kontrolek.

P: A zatem czym to się różni od określenia wysokości wiersza lub szerokości kolumny jako 1*, 2* lub 5*?

O: Zastosowanie * w określeniu wysokości wiersza lub szerokości kolumny sprawia, że wiersze lub kolumny będą powiększane **proporcjonalnie,** wypełniając cały obszar siatki. Jeśli siatka zawiera trzy kolumny o szerokościach 3*, 3* i 4*, to każda z dwóch pierwszych kolumn będzie zajmowała 30% szerokości całej siatki pomniejszonej o szerokości kolumn o ustalonej lub automatycznie wyznaczanej (`Auto`) szerokości; trzecia kolumna (4*) będzie zajmowała 40% szerokości siatki.

Jest pewien powód, dla którego domyślna szerokość i wysokość określona jako 1*, ma sens. Jeśli wszystkie wiersze i kolumny będą miały te domyślne wartości, to niezależnie od wielkości siatki wielkości poszczególnych wierszy i kolumn zawsze będą równe.

P: „Piksele”. Ciągle używacie tego słowa. Nie sądzę, że oznacza ono to, co wy uważacie, że ono oznacza.

O: Wiele osób korzystających z języka XAML używa terminu *piksel*, ale masz rację — z technicznego punktu widzenia nie używasz tych samych pikseli, które widzisz na ekranie. Technicznym określeniem liczb podawanych jako wartości właściwości `Margin`, `Height`, `Width` oraz innych jest: **jednostki niezależne od urządzenia**. Aplikacje przeznaczone dla Sklepu Windows muszą działać na ekranach o różnych wielkościach i kształtach, a zatem niezależnie od tego, jak duży lub mały jest ekran urządzenia, na którym została uruchomiona aplikacja, jedna jednostka niezależna od urządzenia zawsze będzie miała wielkość 1/96 cala. Kwadrat pięć na pięć takich jednostek tworzy jedną podjednostkę układu strony, a kwadrat cztery na cztery podjednostki tworzą jedną jednostkę układu strony. Te wszystkie jednostki układu oraz jednostki niezależne od urządzenia są nieco mylące, dlatego też te drugie będziemy nazywali *pikselami*.

Wszystkie wysokości oraz szerokości w języku XAML można także wyrazić w innych jednostkach, dodając do liczb odpowiednie określenia: `in` (cale), `cm` (centymetry) lub `pt` (punkty, chodzi tu o punkty typograficzne o wielkości 1/72 cala). Spróbuj określić układ strony, używając cali lub centymetrów. A następnie weź linijkę i sprawdź, czy system Windows nadał kontrolkom prawidłowe wymiary.

P: Czy jest jakiś prosty sposób, by mieć pewność, że moja aplikacja będzie dobrze wyglądać na różnych monitorach?

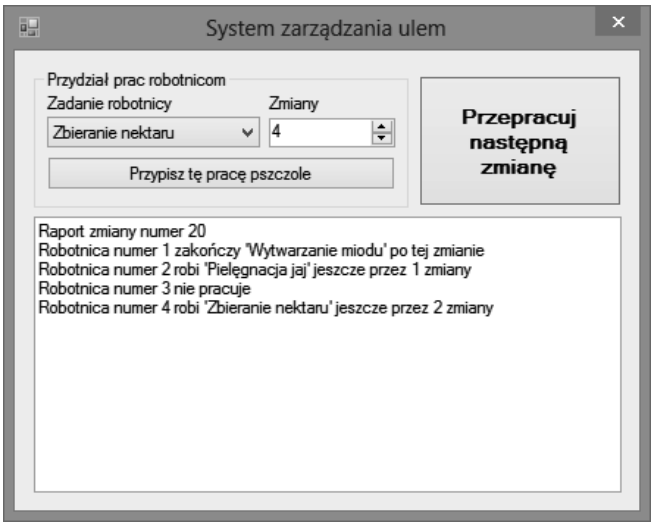
O: Tak, IDE udostępnia przydatne narzędzie, które zapewnia takie możliwości. Okno *Designer* pozwala sprawdzać, jak projektowane strony XAML będą wyglądały na różnych urządzeniach, i to na kilka sposobów. Można skorzystać z okna *Device*, by wyświetlić stronę w różnych rozdzielczościach oraz trybach podziału ekranu. W dalszej części książki pokażemy, jak można uruchamiać aplikację w symulatorze, zapewniającym możliwość prowadzenia z nią interakcji w symulowanych urządzeniach wyposażonych w ekrany o różnej wielkości.

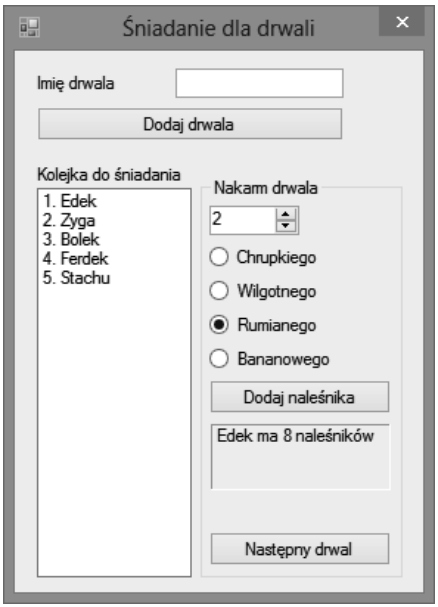
Kiedy szerokość wiersza lub wysokość kolumny ma wartość Auto, oznacza to, że ich wymiary będą dostosowywane do zawartości.

Więcej informacji na temat określania układów stron można znaleźć w witrynie MSDN, na stronie *http://msdn.microsoft.com/pl-pl/library/windows/apps/hh872191.aspx*.

Ćwiczenia

Użyj języka XAML, by przeprojektować każdą z klasycznych aplikacji Windows na aplikację przeznaczoną dla Sklepu Windows. Dla każdej z nich utwórz nowy projekt Blank App i zmień stronę główną na stronę utworzoną przy użyciu szablonu Basic Page (dokładnie tak samo jak w aplikacji *Ratuj ludzi*). Następnie zmodyfikuj strony, wprowadzając odpowiednie zmiany w siatce i dodając do niej kontrolki. Te nowe aplikacje nie muszą działać. Wystarczy, że utworzysz odpowiedni kod XAML, tak by pasowały one wyglądem do poniższych formularzy.





Odszukaj odpowiednie miejsce w kodzie XAML strony Basic Page i dodaj tam nową kontrolkę `Grid` lub `StackPanel`, w której umieścisz pozostałe kontrolki strony. Tę nową siatkę powinieneś umieścić w drugim wierszu (`Grid.Row="1"`) nowo utworzonej strony Basic Page.

```
<Grid Style="{StaticResource LayoutRootStyle}">
    <Grid.RowDefinitions>
        <RowDefinition Height="140"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Grid Grid.Row="1" Margin="120,0"...>

    <!-- Back button and page title -->
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
```

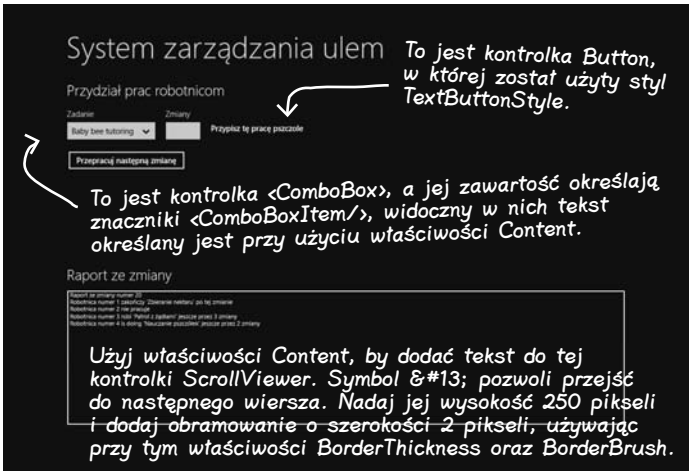
Oto kontrolka <Grid> umieszczona na tej stronie. Zwinęliśmy ją w edytorze XAML.

Odszukaj ten komentarz, a następnie, powyżej niego, dodaj nową kontrolkę <Grid>.

XAML dość elastycznie podchodzi do kolejności znaczników.

Poprosiliśmy Cię, byś dodawał kod XAML nowego układu strony poniżej definicji wierszy, gdyż dzięki temu łatwiej go będzie odnaleźć w pliku. Niektórzy programiści lubią zapisywać kod XAML w takiej samej kolejności, w jakiej poszczególne kontrolki są umieszczane na stronie. Mogą je umieszczać za zamykającym znacznikiem </Grid>, kończącym siatkę zawierającą przycisk ze strzałką w lewo oraz tytuł aplikacji. Radzimy, żebyś poeksperymentował, zapisując kod w różnych miejscach, gdyż warto, byś samemu określił, które miejsce będzie Ci się wydawało najbardziej intuicyjne.

Zastosuj kontrolki `StackPanel`, by zaprojektować układ tego formularza. Składa się on z dwóch grup kontrolek. W nagłówkach tych grup został zastosowany styl `GroupHeaderTextStyle`, same grupy oddziela od siebie margines o wysokości 40 pikseli, a poniżej nagłówków został dodany margines o wysokości 20 pikseli. W etykietach umieszczonych nad kontrolkami został zastosowany styl `BodyTextStyle`, a nad kontrolkami dodano margines o wysokości 10 pikseli. Pomiędzy kontrolkami dodano poziomy margines o szerokości 20 pikseli.




```
<StackPanel Grid.Row="1" Margin="120,0">
<TextBlock/>
  <StackPanel Orientation="Horizontal">
    <StackPanel>
      <TextBlock/>
      <ComboBox>
        <CombiboxItem/>
        <CombiboxItem/>
        ... 4 dodatkowe ...
      </ComboBox>
    </StackPanel>
    <StackPanel>
      <TextBlock/>
      <TextBox/>
    </StackPanel>
    <Button/>
<Button/>
<TextBlock/>
<ScrollViewer/>
</StackPanel>
```

Właściwości SelectedIndex kontrolki ComboBox przypisz wartość 0, by został wyświetlony pierwszy element listy.

Do zaprojektowania tego formularza użyj kontrolki `Grid`. Ma mieć osiem wierszy, a we wszystkich właściwość `Height` ma mieć wartość `Auto`, żeby dostosowywały się do zawartości. Zastosuj kontrolki `StackPanel`, żeby umieścić kilka kontrolek w jednym wierszu.


```
<Grid Grid.Row="1" Margin="120,0">
  <TextBlock/>
  <TextBox/>
  <TextBlock/>
  <ListBox>
    <ListBoxitem/>
    <ListBoxitem/>
    ... 4 dodatkowe ...
  </ListBox>
  <TextBlock>
  <StackPanel Orientation="Horizontal">
    <TextBox/>
    <ComboBox> ... 4 elementy ... </ComboBox>
    <Button/>
  </StackPanel>
  <ScrollViewer/>
  <StackPanel Orientation="Horizontal">
    <Button/>
    <Button/>
  </StackPanel>
```

Zadbaj o to, by strony wyglądały tak samo jak na tych zrzutach, dodając do kontrolek **przykładowe dane**, które w normalnej aplikacji zostałyby określone przy użyciu metod i właściwości klas.

Może się zdarzyć, że kiedy spróbujesz wybrać opcję *New Item...*, by dodać do projektu nową stronę Basic Page, IDE wyświetli w oknie *Designer* komunikat o błędzie — Visual Studio będzie bowiem oczekiwać, że projekt zostanie zbudowany. W takim przypadku wybierz opcję Rebuild Solution, a błąd zniknie.

Rozwiązania ćwiczeń

Użyj języka XAML by przeprojektować każdą z klasycznych aplikacji Windows na aplikację przeznaczoną dla Sklepu Windows. Dla każdej z nich utwórz nowy projekt Blank App i zmień stronę główną na stronę utworzoną przy użyciu szablonu Basic Page (dokładnie tak samo jak w aplikacji *Ratuj ludzi*). Następnie zmodyfikuj strony wprowadzając odpowiednie zmiany w siatce i dodając do niej kontrolki. Te nowe aplikacje nie muszą działać. Wystarczy, że utworzysz odpowiedni kod XAML, tak pasowały one wyglądem do poniższych formularzy.

To jest margines, który miałeś zastosować. Opuszczenie wartości prawego i dolnego marginesu sprawi, że będą one miały takie same wielkości jak marginesy lewy i górny (120 i 0).

```
<StackPanel Grid.Row="1" Margin="120,0">
    <TextBlock Text="Przydział prac robotnicom"
               Style="{StaticResource GroupHeaderTextStyle}"/>
    <StackPanel Orientation="Horizontal" Margin="0,20,0,0">
        <StackPanel Margin="0,0,20,0">
            <TextBlock Text="Zadanie robotnicy" Margin="0,0,0,10"
                       Style="{StaticResource BodyTextStyle}"/>
            <ComboBox SelectedIndex="0">
                <ComboBoxItem Content="Nauczanie pszczółek"/>
                <ComboBoxItem Content="Pielęgnacja jaj"/>
                <ComboBoxItem Content="Utrzymywanie ula"/>
                <ComboBoxItem Content="Wytwarzanie miodu"/>
                <ComboBoxItem Content="Zbieranie nektaru"/>
                <ComboBoxItem Content="Patrol z żądłami"/>
            </ComboBox>
        </StackPanel>
        <StackPanel>
            <TextBlock Text="Shifts" Margin="0,0,0,10"
                       Style="{StaticResource BodyTextStyle}"/>
            <TextBox/>
        </StackPanel>
        <Button Content="Przypisz to zadanie robotnicy" Margin="20,20,0,0"
                Style="{StaticResource TextButtonStyle}" />
    </StackPanel>
    <Button Content="Przepracuj następną zmianę" Margin="0,20,0,0" />

    <TextBlock Text="Raport ze zmiany" Margin="0,40,0,20"
               Style="{StaticResource GroupHeaderTextStyle}"/>
    <ScrollViewer BorderThickness="2" BorderBrush="White" Height="250"
                  Content="
Raport zmiany numer 20&#13;
Robotnica numer 1 robi 'Zbieranie nektaru' jeszcze przez 2 zmiany&#13;
Robotnica numer 2 zakończyła swoje zadanie&#13;
Robotnica numer 2 nie pracuje&#13;
Robotnica numer 3 robi 'Patrol z żądłami' jeszcze przez 3 zmiany&#13;
Robotnica numer 4 robi 'Nauczanie pszczółek' jeszcze przez 4 zmiany
                  "/>
</StackPanel>
```

Czy Twój kod XAML wygląda inaczej niż ten? XAML pozwala na tworzenie stron o bardzo podobnym (lub nawet identycznym) wyglądzie na wiele różnych sposobów.

To jest nagłówek drugiej grupy, powyżej niego jest margines o wysokości 40 pikseli, a poniżej margines o wysokości 20 pikseli.

Oto przykładowy tekst, który umieściliśmy w polu raportu. Właściwość Content ignoruje znaki nowych wierszy dodawane w kodzie XAML — dodaliśmy je tutaj, by poprawić czytelność rozwiązania.

```
<Grid Grid.Row="1" Margin="120,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

    <TextBlock Text="Imię drwala" Margin="0,0,0,10"
               Style="{StaticResource BodyTextStyle}"/>
    <TextBox Grid.Row="1"/>

    <TextBlock Grid.Row="2" Text="Kolejka do śniadania" Margin="0,20,0,10"
               Style="{StaticResource BodyTextStyle}"/>
    <ListBox Grid.Row="3">
        <ListBoxItem Content="1. Edek"/>
        <ListBoxItem Content="2. Zyga"/>
        <ListBoxItem Content="3. Bolek"/>
        <ListBoxItem Content="4. Ferdek"/>
        <ListBoxItem Content="5. Stachu"/>
        <ListBoxItem Content="6. Robert"/>
    </ListBox>

    <TextBlock Grid.Row="4" Text="Nakarm drwala" Margin="0,20,0,10"
               Style="{StaticResource BodyTextStyle}"/>
    <StackPanel Grid.Row="5" Orientation="Horizontal">
        <TextBox Text="2" Margin="0,0,20,0"/>
        <ComboBox SelectedIndex="0" Margin="0,0,20,0">
            <ComboBoxItem Content="chrupkiego"/>
            <ComboBoxItem Content="wilgotnego"/>
            <ComboBoxItem Content="rumianego"/>
            <ComboBoxItem Content="bananowego"/>
        </ComboBox>
        <Button Content="Dodaj naleśniki" Style="{StaticResource TextButtonStyle}"/>
    </StackPanel>

    <ScrollViewer Grid.Row="6" Margin="0,20,0,0" Content="Edek ma 7 naleśników"
                  BorderThickness="2" BorderBrush="White"/>

    <StackPanel Grid.Row="7" Orientation="Horizontal" Margin="0,40,0,0">
        <Button Content="Dodaj drwala" Margin="0,0,20,0" />
        <Button Content="Następny drwal" />
    </StackPanel>

</Grid>
```

Z tych definicji usunęliśmy znaki nowych wierszy, by cały kod rozwiązania zmieścił się na jednej stronie.

Żeby wszystko było jasne: te przykładowe elementy listy miałeś dodać ***w ramach ćwiczenia****, żeby strona przypominała wygląd używanej aplikacji. W dalszej części książki dowiesz się, jak powiązać kontrolki takie jak ListBox z właściwościami swoich klas.*

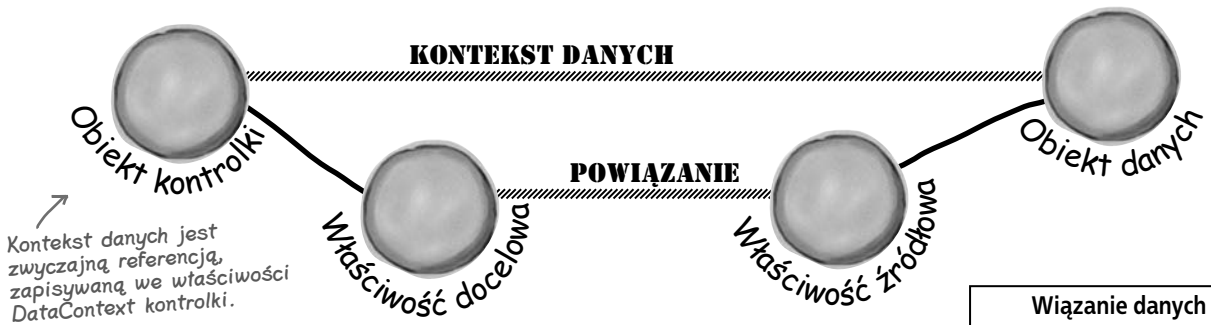
Kolejna przykładowa zawartość...

WYSIL SZARE KOMÓRKI

Co sądzisz o układzie tej strony? Czy nie byłoby lepiej przenieść przyciski *Dodaj drwala* i *Następny drwal* na standardowy pasek aplikacji dostępny w systemie Windows 8?

Wiązanie danych kojarzy strony XAML z klasami

Kontrolki `TextBlock`, `ScrollViewer`, `TextBox` oraz wiele innych zostały stworzone po to, by wyświetlać dane. W aplikacjach WinForms wyświetlanie danych w polach oraz dodawanie elementów do list wymagało korzystania z właściwości. W podobny sposób można także postępować w aplikacjach korzystających z języka XAML, choć istnieje także inne rozwiązanie: można skorzystać z **wiązania danych** (ang. *data binding*), by automatycznie zapisywać dane w kontrolkach umieszczonych na stronie. Co więcej, można nawet skorzystać z wiązania danych, by zapisywać w klasach dane wpisywane w kontrolkach.



Wiązanie danych operuje wyłącznie na właściwościach. Jeśli spróbujemy wykorzystać w tym celu publiczne pole, w kontrolce nie pojawią się żadne dane — co więcej, nie zostanie także wyświetlony żaden błąd!

Kontekst, ścieżka i wiązanie

Wiązanie danych w języku XAML jest relacją pomiędzy ***właściwością źródłową*** obiektu dostarczającego danych dla kontrolki oraz ***właściwością docelową*** kontrolki prezentującej te dane. Aby określić takie wiązanie, w **kontekście danych** kontrolki należy zapisać referencję do obiektu z danymi. Z kolei **wiązanie** musi zawierać **ścieżkę wiązania**, czyli właściwość obiektu zawierającą dane. Kiedy wszystkie te informacje zostaną określone, kontrolka będzie automatycznie odczytywać i wyświetlać wartość właściwości źródłowej.

Ścieżką wiązania zastosowaną w tej kontrolce TextBlock jest właściwość Cash. Kontrolka będzie wyświetlać wartość tej właściwości, odczytywaną z dowolnego obiektu, z którym zostanie powiązana.

Aby określić wiązanie danych w kodzie XAML, należy określić właściwość źródłową w formie **`{Binding Ścieżka}`**:

```
<TextBlock x:Name="walletTextBlock" Text="{Binding Cash}"/>
```

Oprócz tego potrzebny jest obiekt, z którym kontrolka zostanie powiązana — w tym przypadku jest to obiekt `Guy` zapisany w zmiennej `joe`, którego właściwość `Cash` ma wartość `325.50`. Kontekst jest określany przez zapisanie referencji do obiektu `Guy` we właściwości `DataContext` kontrolki.

```
Guy joe = new Guy("Józek", 47, 325.50M);
walletTextBlock.DataContext = joe;
```

Kontekstem danych dla tej kontrolki TextBlock jest referencja do obiektu Guy. Kontrolka ta odczyta z tego obiektu wszystkie powiązane właściwości.

Teraz powiązanie zostało już określone! Podałeś kontekst danych będący instancją klasy `Guy` i określiłeś ścieżkę powiązania odwołującą się do właściwości `Cash`. Skoro zastosowane powiązanie ma postać `Cash`, zatem kontrolka `TextBlock` przejrzy obiekt danych, **poszukując w nim właściwości o nazwie `Cash`**.

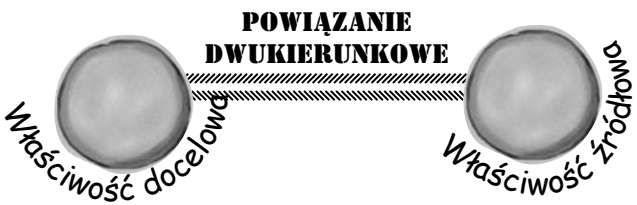
Można także pominąć określanie ścieżki powiązania i zapisać we właściwości kontrolki jedynie wyrażenie `{Binding}`. W takim przypadku wyświetlony zostanie wynik zwrócony przez wywołanie metody `ToString()` klasy `Guy`.

Powiązanie dwukierunkowe pozwala odczytywać i zapisywać wartość właściwości źródłowej

Wiązanie danych pozwala odczytywać wartość z obiektu danych. Jednak, korzystając z **powiązania dwukierunkowego**, można także modyfikować wartość właściwości źródłowej:

```
<TextBox x:Name="ageTextBox" Text="{Binding Age, Mode=TwoWay}"/>
```

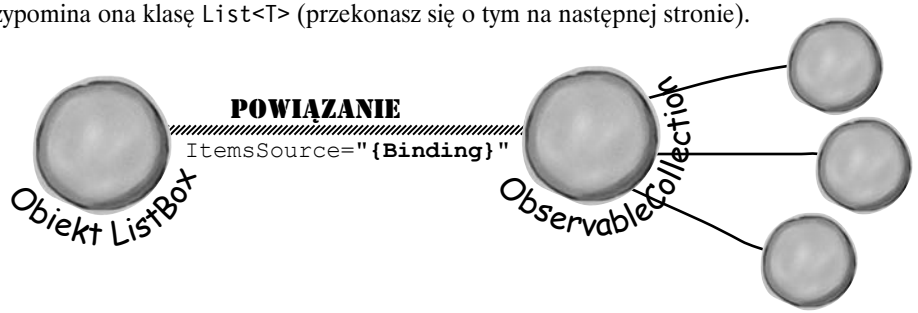
Ścieżka powiązania zastosowana w tej kontrolce `TextBox` odwołuje się do właściwości `Age`, natomiast samo powiązanie działa w trybie dwukierunkowym. W momencie wyświetlania strony kontrolka odczyta i wyświetli wartość właściwości `Age` dowolnego obiektu, z którym została związana. Jeśli jednak zmienimy wartość tej kontrolki, wywoła ona akcesor `set` właściwości `Age` i zaktualizuje jej wartość.



Mechanizm wiązania danych został opracowany w taki sposób, by przysparzać nam jak najmniej problemów. Jeśli ścieżka powiązania będzie się odwoływać do właściwości, która nie jest dostępna w kontekście danych, to kontrolka nie będzie wyświetlać ani aktualizować danych, a jednocześnie nie wystąpią żadne problemy w działaniu programu.

ObservableCollection pozwala tworzyć powiązania z kolekcjami

Niektóre kontrolki, takie jak `TextBlock` lub `TextBox`, wyświetlają łańcuchy znaków. Inne, takie jak `ScrollViewer`, wyświetlają zawartość obiektu. Niemniej jednak spotkałeś się już z kontrolkami wyświetlającymi kolekcje, takimi jak `ListBox` oraz `ComboBox`. To właśnie z tego względu .NET Framework udostępnia klasę `ObservableCollection<T>`, reprezentującą kolekcję opracowaną specjalnie pod kątem mechanizmów wiązania danych. Pod względem sposobu działania przypomina ona klasę `List<T>` (przekonasz się o tym na następnej stronie).



Utworzenie powiązania między właściwością ItemsSource kontrolki ListBox oraz kolekcją ObservableCollection sprawi, że kontrolka wyświetli wszystkie elementy kolekcji.

Utwórz powiązanie w kodzie (bez użycia kodu XAML!)

Jeśli spróbujemy sprawdzić kontrolkę, nie znajdziemy w niej właściwości o nazwie `Binding`. C# nie pozwala pobierać referencji do właściwości obiektu, a jedynie do całego obiektu. Okazuje się, że definiując powiązanie danych w kodzie XAML, określamy je, tworząc **obiekt klasy `Binding`**, zawierający nazwę właściwości źródłowej podaną w formie łańcucha znaków. Oto kod, który tworzy obiekt `Guy` i wiąże go z kontrolką `TextBlock` o nazwie `walletTextBlock`, w taki sposób, że właściwość `Text` kontrolki będzie powiązana z właściwością `Cash` obiektu:

```
Guy joe = new Guy("Joe", 47, 325.50M);
Binding cashBinding = new Binding();
cashBinding.Path = new PropertyPath("Cash");
cashBinding.Source = joe;
walletTextBlock.SetBinding(TextBlock.TextProperty, cashBinding);
```

Istnieje klasa o nazwie DependencyProperty, a klasa TextBlock definiuje całą masę statycznych właściwości będących obiektami tej klasy. Jedna z nich nosi nazwę TextProperty.

Kontrolki XAML mogą zawierać tekst... i nie tylko

Porozmawiajmy trochę o **kodzie znacznikowym** XAML (w końcu to jego reprezentuje litera M w skrócie XAML, a odnosi się on do wszystkich znaczników definiujących stronę) oraz o **kodzie ukrytym** (ang. *code-behind*; czyli kodzie C# stanowiącym uzupełnienie kodu XAML i umieszczanym w plikach *.cs*).

Kiedy używasz kontrolek `Grid` lub `StackPanel`, inne, umieszczane wewnątrz nich kontrolki są zapisywane pomiędzy ich znacznikiem otwierającym i zamykającym. W taki sam sposób można także postępować, używając innych kontrolek. Na przykład wartość właściwości `Text` kontrolek `TextBox` lub `TextBlock` można określić, zapisując łańcuch znaków pomiędzy ich znacznikiem otwierającym i zamykającym:

```
<TextBlock>To jest prezentowany tekst</TextBlock>
```

Ten zapis jest odpowiednikiem określenia właściwości Text.

W takim przypadku nowe wiersze można tworzyć, używając znaczników `<LineBreak/>`, a nie symboli `&#13;`. W obu przypadkach sprowadza się to do zastosowania znaku Unicode o wartości U+0013, który jest interpretowany jako znak nowego wiersza. Można go także zapisać w formie szesnastkowej — `&#xD;`; z kolei symbol `&#xA3;` pozwala umieścić w łańcuchu znak `£` (pamiętasz program Tablica znaków?).

```
<TextBlock>Pierwszy wiersz<LineBreak/>Drugi wiersz</TextBlock>
```

Spróbuj dodać tę kontrolkę `TextBlock` do kodu strony, a następnie wybierz opcję *Edit Text*, by zmienić jej zawartość i naciśnij kombinację klawiszy *Shift+Enter*, by dodać nowy wiersz. W efekcie IDE doda do strony następujący fragment kodu XAML:

```
<TextBlock>
    <Run Text="Pierwszy wiersz"/>
    <LineBreak/>
    <Run Text="Drugi wiersz"/>
</TextBlock>
```

Każde z tych trzech rozwiązań będzie wyglądało na ekranie tak samo, lecz jednocześnie spowoduje wygenerowanie innego grafu obiektów. Każdy znacznik `<Run>` jest przekształcany w odrębny obiekt łańcucha znaków, a każdy z tych obiektów może mieć swoją własną nazwę:

```
<Run Text="Pierwszy wiersz" x:Name="firstLine" />
```

Tej nazwy można następnie użyć w kodzie C# obsługującym stronę XAML do zmiany wyświetlanego łańcucha:

```
firstLine.Text = "To jest nowa zawartość pierwszego wiersza";
```

Kontrolki z zawartością, takie jak `ScrollViewer`, dysponują właściwością `Content` (a nie `Text`), która działa w inny sposób — może zawierać dowolne kontrolki. A takich kontrolek z treścią jest całkiem sporo. Jedną z bardziej przydatnych jest kontrolka `Border`, której można używać, by dodawać tło lub obramowanie do innych kontrolek, które normalnie ich nie mają, takich jak `TextBlock`:

```
<Border Background="Blue"
    BorderBrush="Green" BorderThickness="3">
</Border>
```

Kontrolka ScrollViewer dziedziczy po ContentControl, czyli tej samej kontrolce, której użyłeś, by stworzyć obcych w grze Ratuj ludzi. Utworzona przez Ciebie kontrolka ContentControl zawierała kontrolkę Grid, a w niej trzy kontrolki Ellipse.

ZACZYNAM WYOBRAŻAĆ SOBIE, JAK ZAPROJEKTUJĘ PROGRAM DO ZARZĄDZANIA WYMÓWKAMI. ALE W JAKI SPOSÓB KONTROLKI BĘDĄ ODCZYTYWAĆ I MODYFIKOWAĆ DANE PRZECHOWYWANE W OBIEKTACH?

Nie istnieją głupie pytania

P: Moja strona zawiera siatkę, w której jest umieszczona inna siatka zawierająca StackPanel. Czy istnieje ograniczenie liczby kontrolek, które można umieszczać w innych kontrolkach?

O: Nie. Można umieszczać kontrolki wewnątrz innych kontrolek, a z kolei te w jeszcze innych. W dalszej części rozdziału nauczysz się tworzyć własne kontrolki, zaczynając od utworzenia kontenera i dodając do niego dalsze kontrolki. Na przykład siatkę można umieścić w *dowolnej* innej kontrolce z treścią — zrobiłeś już to w grze *Ratuj ludzi,* tworząc wroga z kontrolki `Grid` i trzech kontrolek `Ellipse`. To jedna z mocnych stron stosowania języka XAML do projektowania aplikacji — pozwala ona na tworzenie złożonych stron przy użyciu prostych kontrolek.

P: Gdybym mógł określić układ strony, używając kontrolki Grid lub StackPanel, to której z nich powinienem użyć?

O: To w dużym stopniu zależy od sytuacji. Na to pytanie nie ma jednej dobrej odpowiedzi: czasami lepszym rozwiązaniem będzie użycie kontrolki `StackPanel`, czasami `Grid`, a czasami połączenia ich obu. A nie są to bynajmniej nasze jedyne możliwości. Można skorzystać z kontrolki `Canvas` (której użyłeś w grze *Ratuj ludzi*), pozwalającej na wyświetlanie innych kontrolek w miejscach określonych przy użyciu właściwości `Canvas.Left` oraz `Canvas.Right`. Wszystkie te trzy kontrolki są klasami pochodnymi klasy `Panel`, a jedną z odziedziczonych po niej możliwości jest dodawanie i prezentowanie dowolnej liczby innych kontrolek.

P: Czy to oznacza, że są kontrolki, w których można umieszczać tylko jedną inną kontrolkę?

O: Owszem. Spróbuj umieścić na stronie kontrolkę `ScrollViewer`. A następnie spróbuj umieścić wewnątrz niej dwie inne kontrolki. Oto co zobaczysz:

```
<ScrollViewer>
    <TextBox/>
    <Button/>
</Sc
```

The property 'Content' is set more than once.

Dzieje się tak dlatego, że w tym przypadku XAML określa wartość właściwości `Content` kontrolki `ScrollViewer`, a ta jest typu `object`. Spróbuj jednak zastąpić kontrolkę `ScrollViewer` kontrolką `Grid`:

```
<Grid>
    <TextBox/>
    <Button/>
</Grid>
```

Okaże się, że wszystko jest w porządku. W tym przypadku zagnieżdżone kontrolki są bowiem dodawane do kolekcji `Children`. (W grze *Ratuj ludzi* używałeś tej kolekcji od dodawania wrogów).

P: Dlaczego niektóre kontrolki, takie jak `TextBlock`, mają właściwość `Text`, a niektóre właściwość `Content`?

O: Ponieważ mogą wyświetlać wyłącznie tekst; właśnie dlatego dysponują właściwością `Text` typu `String`, a nie właściwością `Content` typu `object`. Jest to tak zwana **domyślna właściwość** kontrolki. W przypadku kontrolek takich jak `Grid` oraz `StackPanel` tą domyślną właściwością jest kolekcja `Children`.

P: Czy powinienem wpisywać kod XAML ręcznie, czy raczej używać okna Designer IDE i przeciągać kontrolki z przybornika?

O: Warto spróbować obu tych sposobów i wybrać ten, który Ci bardziej odpowiada. Wielu programistów korzysta niemal wyłącznie z okna *Designer*, choć jest też sporo takich, którzy go niemal wcale nie używają, gdyż przekonali się, że wpisywanie kodu XAML jest szybsze. Dzięki technologii IntelliSense wpisywanie kodu XAML jest faktycznie wyjątkowo łatwe.

P: Przypomnijcie mi jeszcze raz, dlaczego miałem poznać technologię WinForms? Czemu nie zacząłem od razu uczyć się języka XAML i tworzenia aplikacji dla Sklepu Windows?

O: Ponieważ istnieje wiele pojęć, których znajomość znacznie ułatwia zrozumienie języka XAML. Na przykład przyjrzyjmy się kolekcji `Children`. Gdybyś nie rozumiał, czym są kolekcje, to czy odpowiedź na trzecie pytanie zamieszczone na tej stronie miałaby dla Ciebie jakikolwiek sens? Może. Jednak znacznie łatwiej ją zrozumieć, wiedząc czym są kolekcje. Z drugiej strony przeciąganie kontrolek z przybornika i umieszczanie ich na formularzu jest naprawdę łatwe. Korzystanie z technologii WinForms wymaga naprawdę znacznie mniej wiedzy niż projektowanie stron z użyciem języka XAML (co jest zrozumiałe, gdyż XAML jest znacznie nowszą i znacznie bardziej elastyczną technologią). Dzięki temu, że poświęciliśmy kilka rozdziałów, prezentując WinForms, nabrałeś nieco praktyki w projektowaniu aplikacji z graficznym interfejsem użytkownika i tworzeniu interesujących projektów. A to z kolei pozwoliło Ci przyswoić sobie wiele ważnych pojęć. Oprócz tego, poznanie dwóch sposobów utworzenia tych samych projektów jest bardzo wartościowe. To właśnie z tego powodu wróciliśmy do kilku projektów z poprzednich rozdziałów: poznając dwa sposoby napisania tej samej aplikacji, będziesz mógł lepiej zrozumieć zarówno technologię WinForms, jak i aplikacje dla Sklepu Windows.

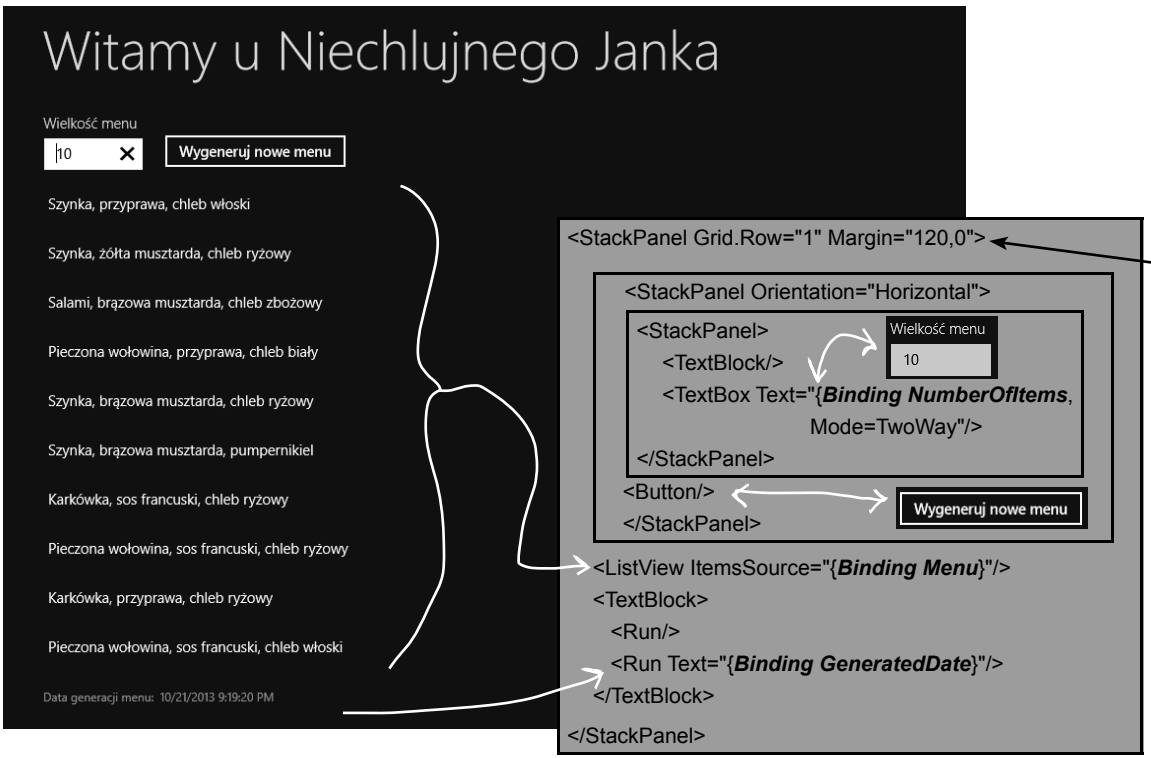
Technologia WinForms jest doskonałym narzędziem do nauki i poznawania języka C#, jednak język XAML jest znacznie lepszym narzędziem do tworzenia elastycznych i efektywnych aplikacji.

Użyj wiązania danych, by usprawnić aplikację Niechlujnego Janka

Pamiętasz program generujący menu dla Niechlujnego Janka, który napisałeś w rozdziale 4? Cóż, Janek zainstalował sobie teraz system Windows 8 i chciałby, żeby jego aplikacja do generowania menu została przystosowana dla Sklepu Windows. Zróbmy zatem to, o co prosi.

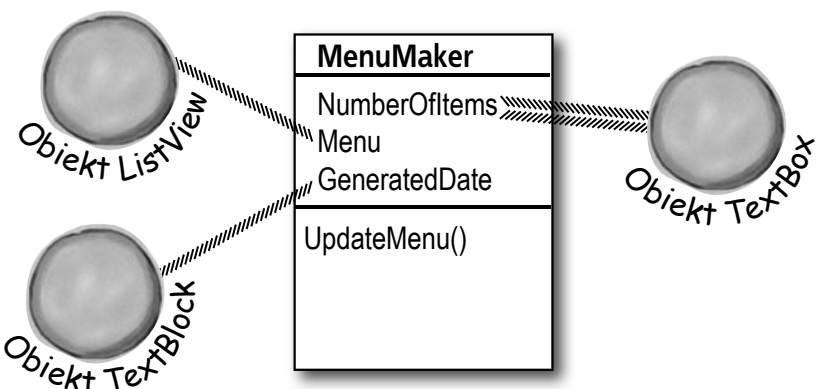
Oto strona, którą mamy zamiar stworzyć.

Będzie ona korzystać z jednokierunkowego wiązania danych, by wyświetlać informacje w kontrolce `ListView` oraz w jednym z obiektów `Run` umieszczonych wewnątrz kontrolki `TextBlock` oraz dwukierunkowego wiązania danych w kontrolce `TextBox`.



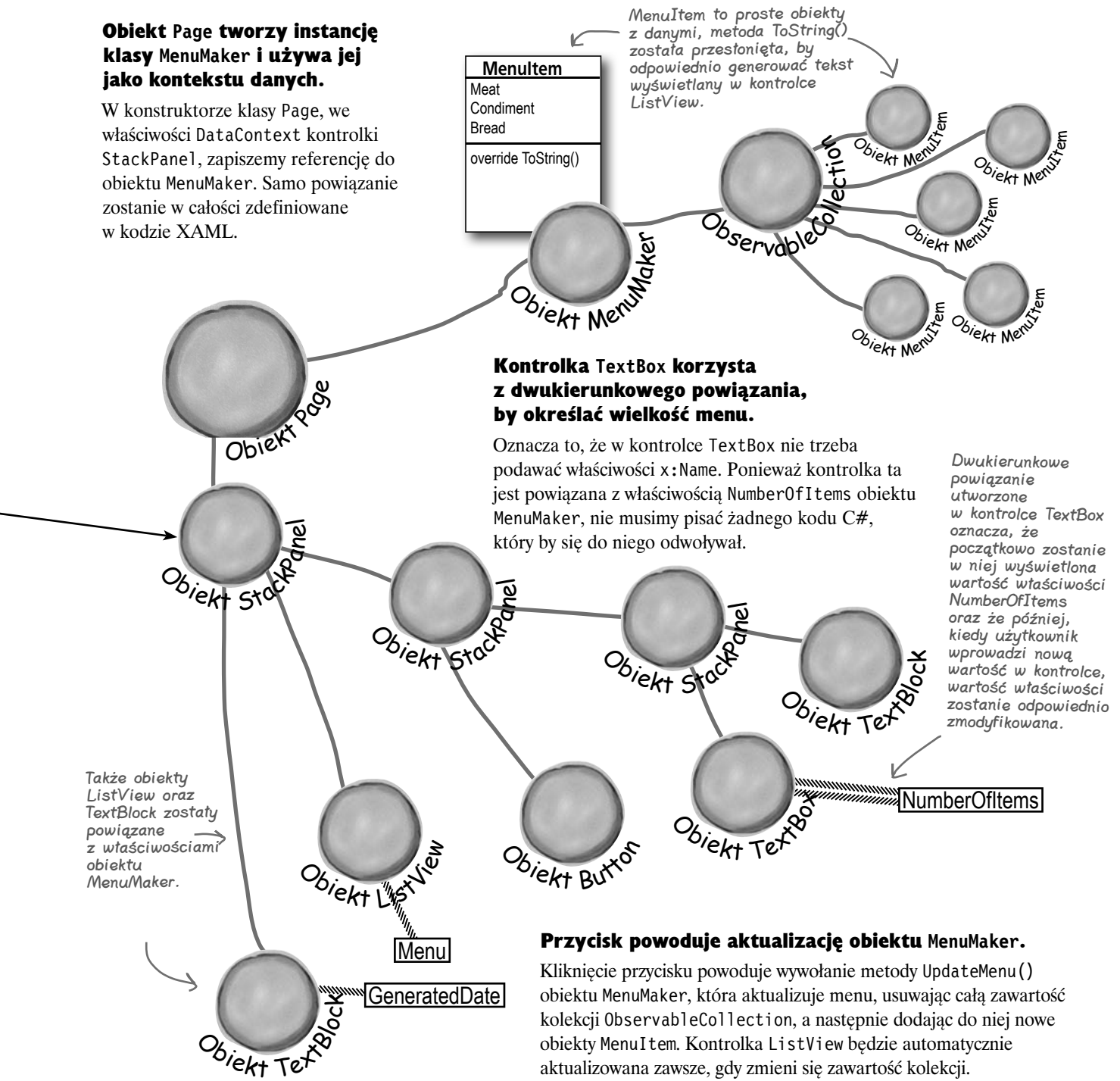
Będzie nam potrzebny obiekt, którego właściwości użyjemy do zdefiniowania powiązania.

Obiekt `Page` będzie zawierał instancję klasy `MenuMaker`, która z kolei dysponuje trzema właściwościami publicznymi: właściwością `NumberOfItems` typu `int`, kolekcją `Menu` typu `ObservableCollection` zawierającą menu oraz właściwością `GeneratedDate` typu `DateTime`.



Obiekt `Page` tworzy instancję klasy `MenuMaker` i używa jej jako kontekstu danych.

W konstruktorze klasy `Page`, we właściwości `DataContext` kontrolki `StackPanel`, zapiszemy referencję do obiektu `MenuMaker`. Samo powiązanie zostanie w całości zdefiniowane w kodzie XAML.



Kontrolka `TextBox` korzysta z dwukierunkowego powiązania, by określać wielkość menu.

Oznacza to, że w kontrolce `TextBox` nie trzeba podawać właściwości `x:Name`. Ponieważ kontrolka ta jest powiązana z właściwością `NumberOfItems` obiektu `MenuMaker`, nie musimy pisać żadnego kodu C#, który by się do niego odwoływał.

Przycisk powoduje aktualizację obiektu `MenuMaker`.

Kliknięcie przycisku powoduje wywołanie metody `UpdateMenu()` obiektu `MenuMaker`, która aktualizuje menu, usuwając całą zawartość kolekcji `ObservableCollection`, a następnie dodając do niej nowe obiekty `MenuItem`. Kontrolka `ListView` będzie automatycznie aktualizowana zawsze, gdy zmieni się zawartość kolekcji.

Oto wyzwanie dla programistów. Bazując na podanych do tej pory informacjach, powiedz, jak wiele z nowej, poprawionej aplikacji dla Niechlujnego Janka jesteś w stanie napisać bez zaglądania na następną stronę?

Zrób to!

1 Utwórz nowy projekt i zastąp stronę MainPage.xaml stroną utworzoną według szablonu Basic Page.

Utwórz nowy projekt **aplikacji typu *Windows Store***. Następnie usuń z niego plik *MainPage.xaml*, **utwórz nową stronę na podstawie szablonu** *Basic Page* **i nadaj jej nazwę** *MainPage.xaml*. Po wprowadzeniu tych zmian będziesz musiał ponownie zbudować projekt. To są dokładnie te same czynności, które wykonałeś, tworząc aplikację *Ratuj ludzi* (zajrzyj do rozdziału 1., jeśli musisz sobie odświeżyć pamięć).

2 Dodaj nową, poprawioną klasę MenuMaker.

Przebyłeś już długą drogę od momentu, gdy zakończyłeś lekturę rozdziału 4. Napiszmy zatem prawidłowo hermetyzowaną klasę, która dzięki zastosowaniu właściwości pozwoli nam zapisywać i pobierać określone informacje. W jej konstruktorze utworzymy kolekcję `ObservableCollection` obiektów typu `MenuItem`, która będzie aktualizowana za każdym razem, gdy zostanie wywołana metoda `UpdateMenu()`. Metoda ta będzie także aktualizować właściwość `GeneratedDate` typu `DateTime`, określającą czas generacji bieżącego menu. Dodaj poniższą klasę `MenuMaker` do projektu:

Kliknij prawym przyciskiem myszy nazwę projektu wyświetloną w oknie Solution Explorer i dodaj nową klasę, tak samo jak robiłeś to w innych projektach.

Będziesz potrzebował tej instrukcji using, gdyż klasa ObservableCollection<T> została zdefiniowana w tej przestrzeni nazw.

```
using System.Collections.ObjectModel;

class MenuMaker {
    private Random random = new Random();
    private List<String> meats = new List<String>()
        { "Pieczona wołowina", "Salami", "Indyk", "Szynka", "Karkówka" };
    private List<String> condiments = new List<String>() {"żółta musztarda",
        "brązowa musztarda", "musztarda miodowa","majonez","przyprawa","sos francuski"};
    private List<String> breads = new List<String>() {"chleb ryżowy",
        "chleb biały", "chleb zbożowy", "pumpernikiel", "chleb włoski", "bułka"};
    public ObservableCollection<MenuItem> Menu { get; private set; }
    public DateTime GeneratedDate { get; private set; }
    public int NumberOfItems { get; set; }
    public MenuMaker() {
        Menu = new ObservableCollection<MenuItem>();
        NumberOfItems = 10;
        UpdateMenu();
    }
    private MenuItem CreateMenuItem() {
        string randomMeat = meats[random.Next(meats.Count)];
        string randomCondiment = condiments[random.Next(condiments.Count)];
        string randomBread = breads[random.Next(breads.Count)];
        return new MenuItem(randomMeat, randomCondiment, randomBread);
    }
    public void UpdateMenu() {
        Menu.Clear();
        for (int i = 0; i < NumberOfItems; i++) {
            Menu.Add(CreateMenuItem());
        }
        GeneratedDate = DateTime.Now;
    }
}
```

Wartości tych właściwości będziesz wyświetlał na stronie, używając wiązania danych. W przypadku właściwości NumberOfItems użyjesz powiązania dwukierunkowego.

Nowa metoda CreateMenuItem() zwraca obiekt MenuItem, a nie łańcuch znaków. W ten sposób łatwiej nam będzie zmieniać wygląd elementów listy, gdyby pojawiła się taka konieczność.

Przyjrzyj się dokładnie, jak działa ta metoda. Okazuje się, że nie tworzy ona nowej kolekcji obiektów MenuItem, a jedynie modyfikuje istniejącą kolekcję, usuwając najpierw jej dotychczasową zawartość, a następnie dodając do niej nowe elementy.

Co się stanie, kiedy właściwości NumberOfItems zostanie przypisana wartość mniejsza od zera?

Używaj typu DateTime, by operować na datach

Spotkałeś się już z typem DateTime pozwalającym przechowywać daty. Można go także używać do tworzenia i modyfikowania dat i czasu. Definiuje on statyczną właściwość o nazwie Now, która zwraca bieżącą datę i godzinę. Udostępnia także metody, takie jak AddSeconds(), pozwalające na dodawanie i konwersję sekund, milisekund, dni itd. oraz właściwości takie jak Hour oraz DayOfWeek pozwalające na odczyt poszczególnych elementów dat i czasu. W sam raz na czasie!

3 Dodaj klasę `MenuItem`.

Przekonałeś się już, że przechowywanie danych w obiektach i klasach, a nie w łańcuchach znaków, pozwala tworzyć bardziej elastyczne programy. Oto prosta klasa służąca do przechowywania jednej kanapki dostępnej w menu. Dodaj ją do swojego projektu:

```
class MenuItem {
    public string Meat { get; private set; }
    public string Condiment { get; private set; }
    public string Bread { get; private set; }

    public MenuItem(string meat, string condiment, string bread) {
        Meat = meat;
        Condiment = condiment;
        Bread = bread;
    }

    public override string ToString() {
        return Meat + ", " + Condiment + ", " + Bread;
    }
}
```

Trzy łańcuchy znaków określające kanapkę są przekazywane do konstruktora i zapisywane w automatycznych właściwościach przeznaczonych tylko do odczytu.

Przesłoń metodę ToString(), aby obiekty MenuItem wiedziały, w jaki sposób chcemy je wyświetlać.

4 Utwórz stronę XAML.

Oto zrzut ekranu aplikacji. Czy potrafiłbyś ją utworzyć, korzystając z kontrolek `StackPanel`? Kontrolka `TextBox` ma szerokość 100 pikseli. W dolnej kontrolce `TextBlock` został zastosowany styl `BodyTextStyle`; zawiera ona dwa znaczniki `<Run>` (w drugim jest wyświetlana wyłącznie data i godzina).

Tym razem nie dodawaj żadnych danych przykładowych. Informacje do prezentacji zostaną dostarczone przez powiązanie danych.

Czy potrafisz samodzielnie stworzyć taką stronę, bazując wyłącznie na tym zrzucie ekranu?

5 Do kodu XAML dodaj nazwy obiektów i pozostałe parametry wiązania danych

Oto kod XAML, który należy dodać do strony *MainPage.xaml*. Upewnij się, że umieścisz go w **zewnętrznej siatce bezpośrednio przed komentarzem XAML** `<!-- Back button and page title -->`, dokładnie tak samo jak na stronie głównej aplikacji Ratuj ludzi. Przyciskowi nadaliśmy nazwę `newMenu`. Ponieważ w kontrolkach `ListView`, `TextBlock` oraz `TextBox` wykorzystaliśmy mechanizm wiązania danych, zatem nie musieliśmy określać ich nazw. *(Dodatkowe ułatwienie. Tak naprawdę nie musieliśmy nawet określać nazwy przycisku; zrobiliśmy to tylko po to, by IDE mogło automatycznie dodać do niego procedurę obsługi zdarzeń o nazwie newMenu_Clicked, w wyniku dwukrotnego kliknięcia tej kontrolki. Przekonaj się sam!).*

```
<StackPanel Grid.Row="1" Margin="120,0" x:Name="pageLayoutStackPanel">
    <StackPanel Orientation="Horizontal" Margin="0,0,0,20">
        <StackPanel Margin="0,0,20,0">
            <TextBlock Style="{StaticResource BodyTextStyle}"
                       Text="Wielkość menu" Margin="0,0,0,10" />
            <TextBox Width="100" HorizontalAlignment="Left"
                       Text="{Binding NumberOfItems, Mode=TwoWay}" />
        </StackPanel>
        <Button x:Name="newMenu" VerticalAlignment="Bottom" Click="newMenu_Click"
                Content="Wygeneruj nowe menu" Margin="0,0,20,0"/>
    </StackPanel>
    <ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0" />
    <TextBlock Style="{StaticResource CaptionTextStyle}">
        <Run Text="Data generacji menu: " />
        <Run Text="{Binding GeneratedDate}"/>
    </TextBlock>
</StackPanel>
```

Do odczytu i ustawiania liczby kanapek w menu przy użyciu kontrolki TextBox konieczne było zastosowanie powiązania dwukierunkowego.

Oto kontrolka ListView. Spróbuj zastąpić ją kontrolką ListBox i sprawdź, co się zmieni na stronie.

To właśnie w takich miejscach przydają się znaczniki <Run>. Dzięki nim wystarczyło użyć jednej kontrolki TextBlock i zdefiniować powiązanie jedynie z fragmentem tekstu.

Do pliku MainPage.xaml.cs dodaj kod obsługujący stronę.

W konstruktorze strony tworzona jest kolekcja zawierająca całe menu, obiekt `MenuMaker` oraz określany kontekst danych kontrolek wykorzystujących mechanizm wiązania danych. Wymaga to dodatkowo zdefiniowania pola typu `MenuMaker` o nazwie `menuMaker`.

```
MenuMaker menuMaker = new MenuMaker();

public MainPage() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
}
```

W pliku MainPage.xaml.cs pojawiło się pole typu MenuMaker, które posłuży jako kontekst danych dla kontrolki StackPanel zawierającej wszystkie powiązane kontrolki prezentujące dane na stronie.

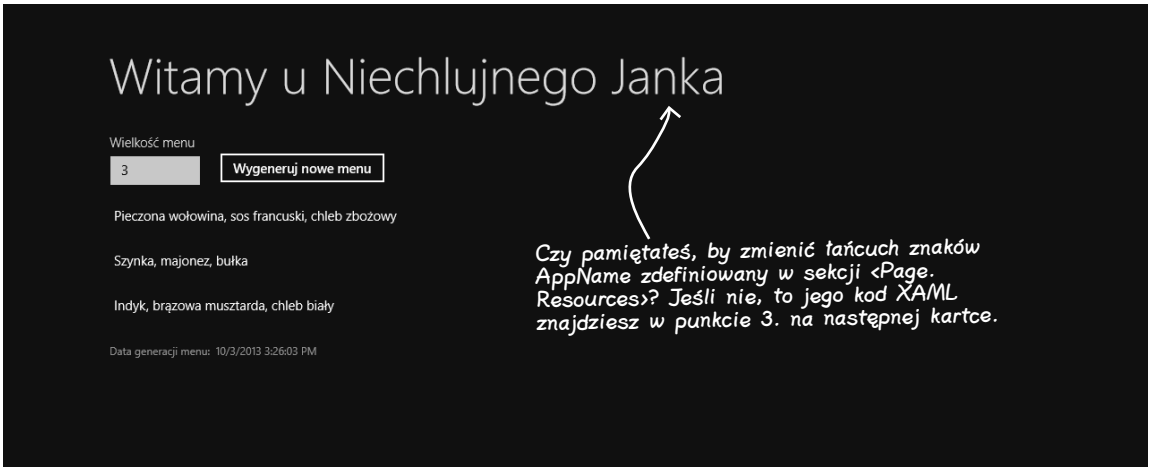
Musisz określić kontekst danych dla kontrolki `StackPanel`. Dzięki temu zostanie on przekazany do wszystkich kontrolek umieszczonych wewnątrz niej.

W końcu, dwukrotnie kliknij przycisk, aby wygenerować szkielet procedury obsługi zdarzeń `Click`. Oto cały kod obsługi tych zdarzeń — ogranicza się on do zaktualizowania menu:

```
private void newMenu_Click(object sender, RoutedEventArgs e) {
    menuMaker.UpdateMenu();
}
```

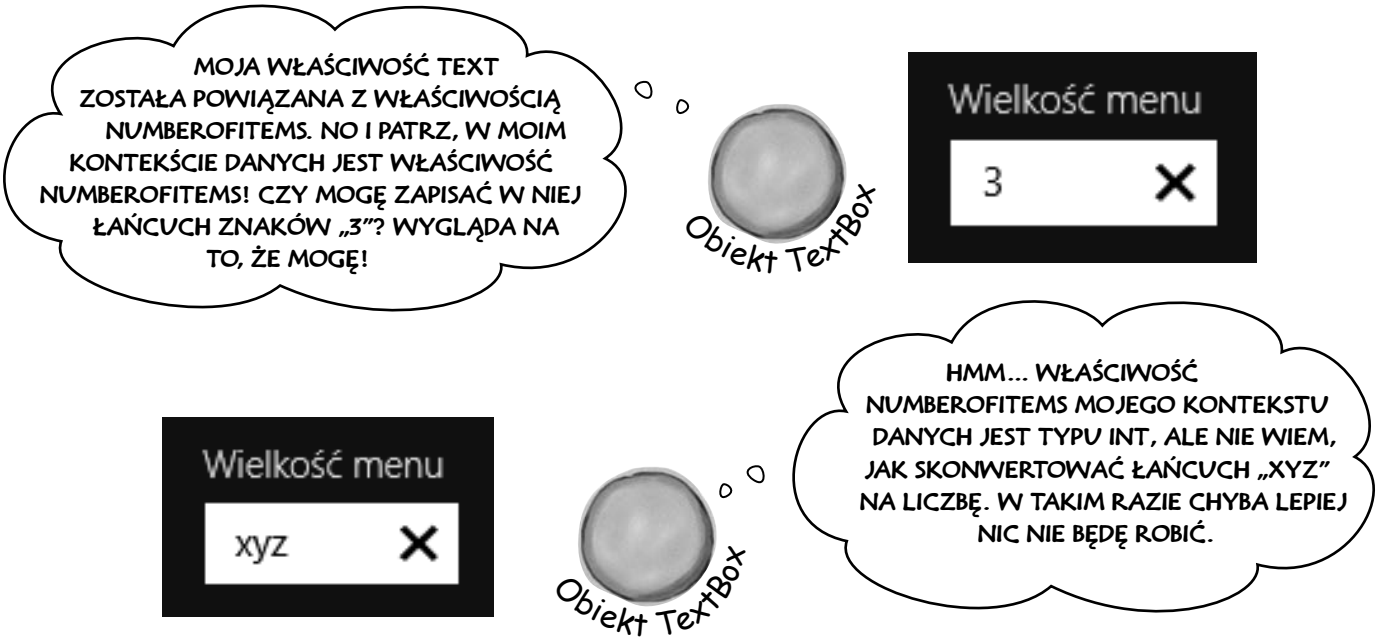
Jest pewien sposób pozwalający na łatwą zmianę nazwy procedury obsługi zdarzeń, tak by jednocześnie zaktualizować zarówno kod C# jak i XAML. Zajrzyj do punktu 8. dodatku „Pozostałości”, by dowiedzieć się więcej o narzędziach refaktoryzacji dostępnych w Visual Studio IDE.

A teraz uruchom program! Spróbuj zmienić liczbę w polu `TextBox`. Wpisz w nim wartość 3 i kliknij przycisk — program wygeneruje nowe menu zawierające trzy kanapki.



Teraz możesz już wypróbować powiązania danych, by przekonać się, jak bardzo są one elastyczne. Spróbuj wpisać w polu tekstowym „xyz” lub zostawić je puste. Nic się nie stanie! Wpisując cokolwiek w kontrolce `TextBox`, umieszczamy w niej łańcuch znaków. Kontrolka całkiem inteligentnie określa, co ma z tym łańcuchem zrobić. Wie, że ścieżka powiązania ma postać `NumberOfItems`, zatem przegląda kontekst danych, sprawdzając, czy jest w nim dostępna jakakolwiek właściwość o tej nazwie, a następnie stara się w możliwie najlepszy sposób skonwertować łańcuch znaków na typ tej właściwości.

Zwróć uwagę na wygenerowaną datę. Nie zmienia się ona, choć menu jest aktualizowane. Cóż, chyba wciąż pozostaje nam jeszcze coś do zrobienia.



Korzystaj z zasobów statycznych, by deklarować obiekty w kodzie XAML

Tworząc stronę w języku XAML, w rzeczywistości tworzysz graf obiektów, posługując się przy tym takimi obiektami jak `StackPanel`, `Grid`, `TextBlock` czy też `Button`. Jak się już przekonałeś, nie ma w tym niczego magicznego ani tajemniczego — dodanie do kodu XAML znacznika `<TextBox>` sprawi, że w obiekcie strony pojawi się pole typu `TextBox`, w którym zostanie zapisana referencja do instancji klasy `TextBox`. Dodając do tego znacznika właściwość `x:Name`, sprawimy, że w kodzie ukrytym będziemy mogli używać tej nazwy, by odwoływać się do kontrolki.

Dokładnie w ten sam sposób można tworzyć instancje *niemal wszystkich* klas i zapisywać je w polach obiektu strony. Pozwalają na to tak zwane **zasoby statyczne** dodawane do kodu XAML. Co więcej, mechanizm wiązania danych bardzo dobrze współpracuje z takimi zasobami statycznymi, zwłaszcza jeśli wykorzystamy także okno Designer IDE. Wróć zatem do programu dla Niechlujnego Janka i przekształć obiekt `MenuMaker` w zasób statyczny.

1 Z pliku kodu ukrytego usuń pole MenuMaker.

Masz zamiar utworzyć obiekt klasy `MenuMaker` oraz kontekst danych w kodzie XAML, a zatem usuń z kodu C# dwa wyróżnione wiersze:

```
~~MenuMaker menuMaker = new MenuMaker();~~

public MainPage() {
    this.InitializeComponent();

    ~~pageLayoutStackPanel.DataContext = menuMaker;~~
}
```

Nie istnieją głupie pytania

P: Ale zaraz! Ta aplikacja nie ma przycisku Zamknij! W jaki sposób mogę z niej wyjść?

O: Aplikacje przeznaczone dla Sklepu Windows domyślnie nie mają żadnego przycisku do zamykania, gdyż zazwyczaj z większości spośród nich nigdy się nie wychodzi. Aplikacje tego typu działają zgodnie z **cyklem życia aplikacji**, który definiuje trzy stany: nie działa, działa, wstrzymana. Aplikacje mogą zostać wstrzymane, jeśli użytkownik z nich wyjdzie lub gdy system Windows wykryje, że w urządzeniu zaczyna brakować prądu. Oprócz tego, jeśli system będzie musiał odzyskać pamięć, to może zakończyć działanie aplikacji. W dalszej części książki nauczysz się tworzyć aplikacje działające zgodnie z tym cyklem.

2 Sprawdź, jak wygląda przestrzeń nazw twojej aplikacji.

Spójrz na sam początek kodu XAML strony, a przekonasz się, że w jej znaczniku otwierającym jest umieszczonych kilka właściwości `xmlns`. Każda z nich definiuje przestrzeń nazw. Poszukaj takiej, która zaczyna się od `xmlns:local` i odpowiada przestrzeni nazw projektu. Powinna wyglądać mniej więcej tak:

To jest właściwość przestrzeni nazw XML. Zawiera ona łańcuch „xmlns:" zakończony identyfikatorem, którym w tym przypadku jest „local".

Jeśli wartość przestrzeni nazw rozpoczyna się od „using:", oznacza to, że odwołuje się ona do jednej z przestrzeni nazw dostępnych w projekcie. Może się także rozpoczynać od „http://", co będzie oznaczać standardową przestrzeń nazw XAML.

```
xmlns:local="using:NiechlujnyJanekRozdzial10"
```

Będziesz używał tego identyfikatora podczas tworzenia obiektów w przestrzeni nazw projektu.

Ponieważ nasza aplikacja nosi nazwę NiechlujnyJanekRozdzial10, zatem IDE utworzył tę przestrzeń nazw na nasze potrzeby. Odszukaj przestrzeń nazw używaną w Twojej aplikacji, gdyż to właśnie w niej będzie istniał obiekt MenuMaker.

3 DODAJ DO PLIKU XAML ZASÓB STATYCZNY I OKREŚL KONTEKST DANYCH.

Odszukaj w kodzie strony sekcję `<Page.Resources>` i wpisz **`<local:`** — spowoduje to wyświetlenie okienka *IntelliSense*:

```
<Page.Resources>

    <local:
            App
            MenuItem
            MenuMaker

    <!-- TODO: Delete this line if the key AppName is declared in App.xaml -->
    <x:String x:Key="AppName">Witamy u Niechlujnego Janka</x:String>
</Page.Resources>
```

Zasoby statyczne można tworzyć wyłącznie w przypadku, gdy ich klasa definiuje konstruktor bezparametrowy. To ma sens! Gdyby konstruktor miał parametry, to skąd strona XAML miałaby wiedzieć, jakie argumenty przekazać w jego wywołaniu?

W okienku zostaną wyświetlone wszystkie klasy dostępne w przestrzeni nazw, których możesz używać. Wybierz `MenuMaker` i nadaj jej nazwę `menuMaker`:

```
<local:MenuMaker x:Name="menuMaker"/>
```

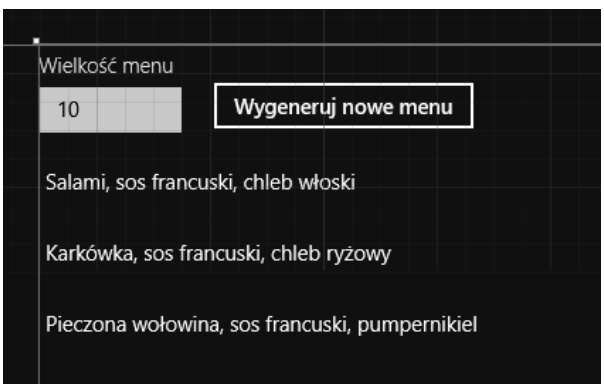
Teraz strona zawiera statyczny zasób typu `MenuMaker` o nazwie `menuMaker`.

4 OKREŚL KONTEKST DANYCH KONTROLKI STACKPANEL I WSZYSTKICH KONTROLEK WEWNĄTRZ NIEJ.

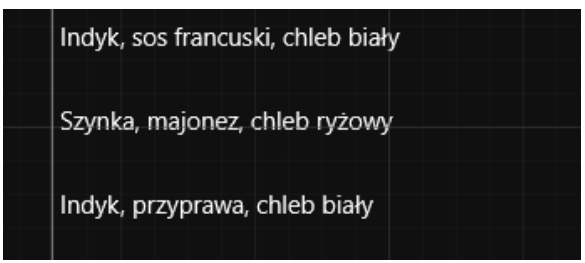
Odszukaj zewnętrzną kontrolkę `StackPanel` i określ jej właściwość `DataContext` w następujący sposób:

```
<StackPanel Grid.Row="1" Margin="120,0"
      DataContext="{StaticResource ResourceKey=menuMaker}">
```

Twój program będzie działał dokładnie tak samo jak wcześniej. Ale czy zauważyłeś, co się stało w IDE, kiedy do kodu XAML dodałeś kontekst danych? Gdy tylko to zrobiłeś, IDE utworzyło instancję klasy `MenuMaker` i użyło jej właściwości do określenia zawartości powiązanych kontrolek. W oknie *Designer* natychmiast pojawiło się wygenerowane menu — jeszcze zanim zdążyłeś uruchomić aplikację. Niezłe!



W oknie Designer natychmiast zostanie wyświetlone wygenerowane menu, jeszcze zanim zdążysz uruchomić aplikację.



Hm... coś tu nie jest w porządku. Została wyświetlona liczba kanapek w menu, samo menu, ale nie data. O co chodzi?

Wyświetlaj obiekty, używając szablonów danych

Podczas wyświetlania elementów listy prezentowana jest zawartość kontrolek `ListViewItem` (używanych w kontrolkach `ListView`), `ListBoxItem` lub `ComboBoxItem`, powiązanych z obiektami zapisanymi w kolekcji `ObservableCollection`. Każda kontrolka `ListViewItem` w aplikacji Niechlujnego Janka jest powiązana z obiektem `MenuItem` przechowywanym w kolekcji `Menu`. Domyślnie obiekty `ListViewItem` wywołują metodę `ToString()` obiektów `MenuItem`, możesz jednak zastosować **szablon danych**, który skorzysta z mechanizmu wiązania danych, by wyświetlać informacje z właściwości powiązanych obiektów.

Zmodyfikuj znacznik `<ListView>`, dodając do niego prosty szablon danych. Używa on prostego wyrażenia `{Binding}`, by wywoływać metodę `ToString()` elementów listy.

To naprawdę prosty szablon danych, co więcej, wygląda on bardzo podobnie do standardowego szablonu używanego do wyświetlania elementów ListViewItem.

```
<ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding}"/>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Nie zmieniaj nic wewnątrz znacznika ListView, tylko zastąp /> znakiem > i dodaj poniżej zamykający znacznik </ListView>. Następnie pomiędzy znacznikami ListView dodaj znacznik ListView.ItemTemplate zawierający szablon danych.

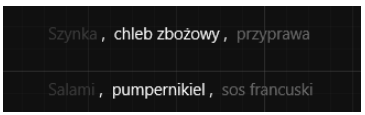
Dodanie {Binding} bez żadnej ścieżki powoduje wywołanie metody ToString() powiązanego obiektu.

Zmień szablon danych, dodając do niego trochę kolorów.

Zmień znacznik <DataTemplate> wraz z zawartością, lecz resztę znacznika ListView pozostaw bez zmian.

```
<DataTemplate>
    <TextBlock>
        <Run Text="{Binding Meat}" Foreground="Blue"/><Run Text=", "/>
        <Run Text="{Binding Bread}" FontWeight="Light"/><Run Text=", "/>
        <Run Text="{Binding Condiment}" Foreground="Red" FontWeight="ExtraBold"/>
    </TextBlock>
</DataTemplate>
```

Możesz powiązać poszczególne znaczniki Run. Możesz w nich zmieniać czcionkę, kolor oraz inne właściwości.



Zaszalej! Szablon danych może zawierać zupełnie dowolne kontrolki.

```
<DataTemplate>
    <StackPanel Orientation="Horizontal">
        <StackPanel>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
        </StackPanel>
        <Ellipse Fill="DarkSlateBlue" Height="Auto" Width="10" Margin="10,0"/>
        <Button Content="{Binding Condiment}" FontFamily="Segoe Script"/>
    </StackPanel>
</DataTemplate>
```

Właściwość Content obiektu DataTemplate może zawierać tylko jeden obiekt, a zatem, chcąc umieścić w szablonie danych więcej kontrolek, będziesz musiał użyć jakiegoś kontenera, takiego jak StackPanel.

Nie istnieją głupie pytania

P: A zatem do określania układu mogę używać kontrolek StackPanel lub Grid. Mogę tworzyć zasoby statyczne w kodzie XAML lub używać pól definiowanych w kodzie ukrytym. Mogę ustawiać właściwości kontrolek lub używać wiązania danych. Po co jest tyle sposobów na realizację tego samego celu?

O: Ponieważ C# oraz XAML są niezwykle elastycznymi narzędziami do tworzenia aplikacji. Ta elastyczność pozwala projektować bardzo szczegółowe strony działające na wielu różnych urządzeniach z różnymi ekranami. Dzięki nim dysponujesz obszernym zestawem narzędzi, których możesz używać do tworzenia *odpowiednich* stron. Nie postrzegaj zatem tej sytuacji jako źródła zamieszania, wyobraź sobie, że dysponujesz wieloma opcjami, spośród których możesz wybrać te najlepsze.

P: Wciąż nie do końca rozumiem, jak działają zasoby statyczne. Co się dzieje, kiedy umieszczę znacznik wewnątrz sekcji <Page.Resources>?

O: Dodając tam znacznik, aktualizujesz obiekt `Page`. Odszukaj zasób `AppName`, w którym podałeś nową wartość, by zmienić nagłówek strony:

```
<x:String x:Key="AppName">Witamy u Niechlujnego Janka</x:String>
```

A teraz przejrzyj kod, który IDE dodało podczas tworzenia szablonu Basic Page, i sprawdź, gdzie ten zasób jest używany:

```
<TextBlock x:Name="pageTitle" Grid.Column="1"
    Text="{StaticResource AppName}"
    Style="{StaticResource PageHeaderTextStyle}"/>
```

Strona używa go, by określić tekst. A zatem, co się tak naprawdę dzieje? Możesz się o tym przekonać, używając IDE. Umieść punkt przerwania w procedurze obsługi kliknięć przycisku, uruchom aplikację i kliknij przycisk. Do okna *Watch* dodaj wyrażenie `this.Resources["AppName"]` — zauważysz, że zawiera ono referencję do łańcucha znaków. W taki sposób działają wszystkie zasoby statyczne — dodanie takiego zasobu do kodu powoduje utworzenie obiektu i zapisanie referencji do niego w kolekcji o nazwie `Resources`.

P: Czy mogę używać zapisu {StaticResource} we własnym kodzie, czy tylko w szablonach, takich jak Basic Page?

O: Oczywiście — możesz tworzyć i używać zasobów w taki sposób. W szablonie Blank Page nie ma niczego szczególnego, podobnie jak w innych szablonach używanych w tej książce. Zawierają one zwyczajny kod C# i XAML i nie robią niczego, czego nie mógłbyś zrobić samemu.

P: Użyłem właściwości x:Name, by określić nazwę zasobu MenuMaker, natomiast zasób AppName używa właściwości x:Key. Czym one się różnią?

O: Zastosowanie właściwości `x:Key` powoduje, że zasób statyczny jest dodawany do kolekcji `Resources` i kojarzony z podanym kluczem, natomiast nie jest tworzone pole (a zatem nie możesz użyć identyfikatora `AppName` w kodzie C#, musisz odwoływać się do takiego zasobu, używając kolekcji `Resources`). W razie użycia właściwości `x:Name` zasób jest dodawany do kolekcji `Resources`, a oprócz tego jest dodawany jako pole do obiektu `Page`. To dzięki temu byłeś w stanie wywołać metodę `UpdateMenu()` na rzecz statycznego zasobu `MenuMaker`.

P: Czy ścieżka powiązania musi wskazywać na właściwość zawierającą łańcuch znaków?

O: Nie. Można wiązać właściwości dowolnych typów. O ile tylko można skonwertować typ właściwości źródłowej i docelowej, to powiązanie będzie działać. W przeciwnym razie dane zostaną zignorowane. Pamiętaj także, że nie wszystkie właściwości kontrolek są łańcuchami znaków. Załóżmy, że w kontekście danych jest dostępna właściwość logiczna o nazwie `EnableMyObject`. Możesz ją powiązać z dowolną właściwością typu logicznego, taką jak `IsEnabled`. W takim przypadku kontrolka będzie aktywowana i dezaktywowana zależnie od wartości właściwości `EnableMyObject`:

```
IsEnabled="{Binding EnableMyObject}"
```

Oczywiście, jeśli powiążesz ją z właściwością tekstową, to będzie wyświetlany jedynie tekst `True` lub `False` (co, jeśli się nad tym nieco zastanowić, będzie całkowicie zrozumiałe).

P: Dlaczego IDE wyświetla dane na formularzu, kiedy utworzę zasób statyczny i powiązanie w kodzie XAML, lecz nie kiedy zrobię to w kodzie C#?

O: Ponieważ IDE rozumie kod XAML, zawierający wszystkie informacje niezbędne do utworzenia obiektów koniecznych do wyświetlenia strony. Kiedy tylko utworzyłeś w kodzie XAML zasób `MenuMaker`, IDE utworzyło instancję tej klasy. Nie było w stanie zrobić tego na podstawie instrukcji `new` w konstruktorze, gdyż mógł on zawierać wiele innych instrukcji i także one musiałyby zostać wykonane. IDE wykonuje kod ukryty C# wyłącznie w wyniku uruchomienia aplikacji. Jeśli jednak dodasz do strony zasób statyczny, to IDE go utworzy, tak samo jak tworzy instancje kontrolek `TextBlock`, `StackPanel` i wszystkich innych. Ustawia także właściwości tych kontrolek, by je wyświetlić w oknie *Designer*, a zatem, kiedy określisz kontekst danych oraz ścieżki powiązań, także one zostaną użyte i w efekcie menu pojawi się w oknie IDE.

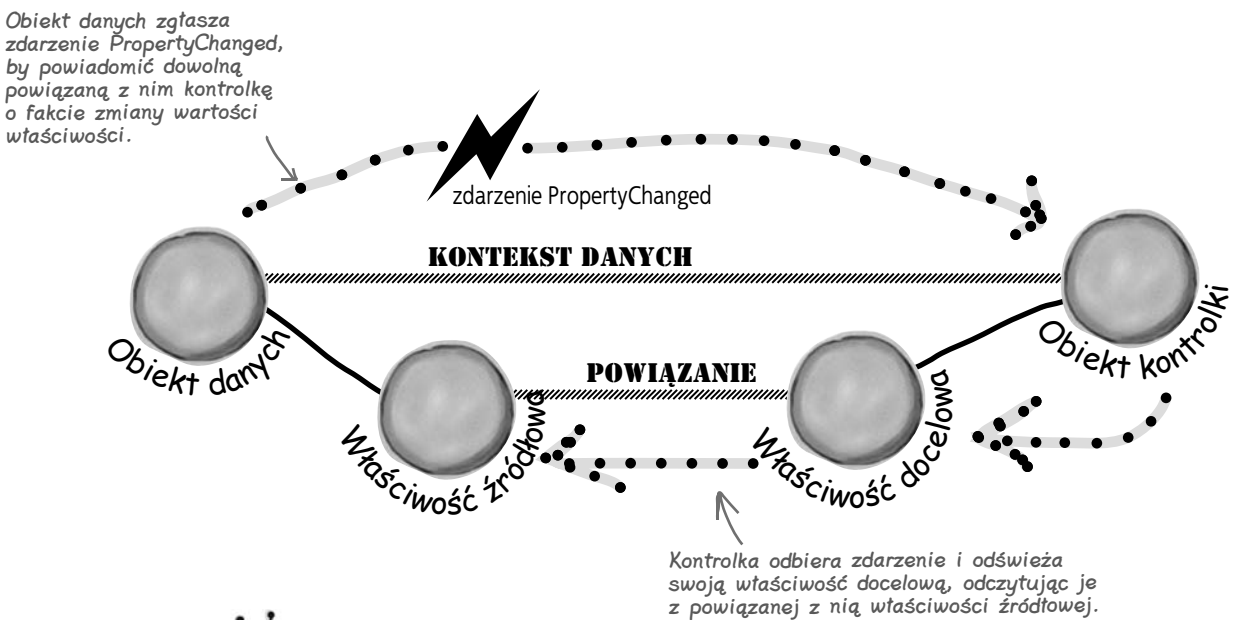
Zasoby statyczne dodawane do strony są tworzone w momencie wczytywania strony, a później mogą być używane przez dowolne obiekty aplikacji.

Nazwa „zasób statyczny" jest nieco myląca. Są one tworzone dla każdej instancji i w żadnym przypadku nie przypominają pól statycznych!

Interfejs INotifyPropertyChanged pozwala powiązanym obiektom przesyłać aktualizacje

Kiedy klasa `MenuMaker` aktualizuje menu kanapek, powiązana z nią kontrolka `ListView` zostaje zaktualizowana. Jednak dokładnie w tym samym momencie klasa `MenuMaker` zmienia także wartość właściwości `GeneratedDate`. Dlaczego zatem nie jest aktualizowana powiązana z nią kontrolka `TextBlock`? Wynika to z faktu, że każda zmiana kolekcji typu `ObservableCollection` powoduje **zgłoszenie zdarzenia**, które informuje dowolną powiązaną kontrolkę o pojawieniu się zmian. W analogiczny sposób kliknięcie przycisku powoduje zgłoszenie zdarzenia `Click`, a upłynięcie czasu odmierzanego przez obiekt `Timer` powoduje zgłoszenie zdarzenia `Tick`. Zawsze gdy dodasz lub usuniesz jakieś elementy kolekcji `ObservableCollection`, zgłosi ona odpowiednie zdarzenie.

Możesz zadbać, by obiekty danych informowały o swoich zmianach właściwości docelowe i powiązane kontrolki. Należy w tym celu **zaimplementować interfejs `INotifyPropertyChanged`**, zawierający tylko jedno zdarzenie: `PropertyChanged`. Wystarczy zgłosić to zdarzenie po każdej zmianie wartości właściwości i obserwować, jak powiązana kontrolka zostanie automatycznie zaktualizowana.



Kolekcje działają niemal w ten sam sposób co obiekty danych

W rzeczywistości klasa `ObservableCollection<T>` nie implementuje interfejsu `INotifyPropertyChanged`. Zamiast tego implementuje ściśle z nim związany interfejs **`INotifyCollectionChanged`**, *który zamiast zdarzeń `PropertyChanged` generuje zdarzenia `CollectionChanged`. Kontrolka wie, że powinna oczekiwać na te zdarzenia, gdyż klasa `ObservableCollection` implementuje interfejs `INotifyCollectionChanged`. Zapisanie we właściwości `DataContext` obiektu implementującego ten interfejs sprawi, że kontrolka będzie reagować na zdarzenia `CollectionChanged`.*

Zmodyfikuj klasę MenuMaker, by informowała Cię, gdy zmieni się właściwość GeneratedDate

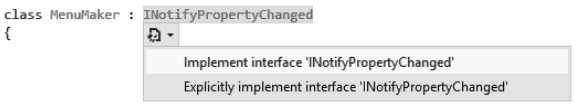
To pierwszy raz, kiedy zgłaszasz zdarzenia.

Procedury obsługi zdarzeń piszesz już od samego początku tej książki, jednak pierwszy raz masz okazję samemu zgłaszać zdarzenia. Wszystkiego na temat zgłaszania zdarzeń oraz tego, jak one działają, dowiesz się w rozdziale 15. Jak na razie wystarczy, żebyś wiedział, że interfejs może deklarować zdarzenia oraz że Twoja metoda `OnPropertyChanged()` jest zgodna ze standardowym wzorcem przekazywania zdarzeń do innych obiektów, stosowanym w C#.

Interfejs `INotifyPropertyChanged` został zdefiniowany w przestrzeni nazw `System.ComponentModel`, a zatem powinieneś zacząć od dodania na samym początku pliku klasy `MenuMaker` następującej instrukcji `using`:

```
using System.ComponentModel;
```

Zmodyfikuj klasę `MenuMaker`, dodając do jej deklaracji interfejsu `INotifyPropertyChanged`, a następnie skorzystaj z możliwości IDE, by go zaimplementować:



Będzie to wyglądało nieco inaczej od przykładów, które widziałeś w rozdziałach 7. i 8. Nie będziesz musiał dodawać żadnych metod ani właściwości. Zamiast tego dodasz zdarzenie:

```
public event PropertyChangedEventHandler PropertyChanged;
```

A następnie dodasz poniższą metodę `OnPropertyChanged()`, której będziesz używał do zgłaszania zdarzenia `PropertyChanged`:

```
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

To jest standardowy wzorzec zgłaszania zdarzeń stosowany w .NET Framework.

Teraz jedyną rzeczą, którą musisz zrobić, by poinformować powiązaną kontrolkę o zmianie właściwości, jest wywołanie metody `OnPropertyChanged()` i przekazanie do niej nazwy właściwości, która uległa zmianie. Chcemy, by po każdej zmianie menu była aktualizowana kontrolka `TextBox` powiązana z właściwością `GeneratedDate`; w tym celu wystarczy, że dodamy do metody `UpdateMenu()` jeden wiersz kodu:

```
public void UpdateMenu() {
    Menu.Clear();
    for (int i = 0; i < NumberOfItems; i++) {
        Menu.Add(CreateMenuItem());
    }
    GeneratedDate = DateTime.Now;

    OnPropertyChanged("GeneratedDate");
}
```

Teraz po wygenerowaniu nowego menu powinna się także zmienić data.

Nie zapomnij zaimplementować interfejsu `INotifyPropertyChanged`.

Mechanizm wiązania danych działa tylko wtedy, gdy kontrolki implementują ten interfejs. Jeśli do deklaracji klasy nie dodasz `: INotifyPropertyChanged`, to powiązane kontrolki nie będą aktualizowane — i to nawet jeśli obiekt danych będzie zgłaszał zdarzenia `PropertyChanged`.

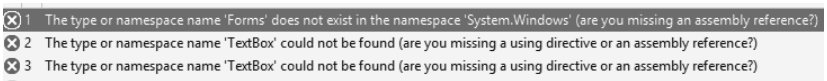
Ćwiczenia

Zakończ przerabianie gry *Idź na ryby!* do postaci aplikacji dla Sklepu Windows. Będziesz musiał zmodyfikować kod XAML przygotowany wcześniej w tym rozdziale, dodając do niego powiązania danych, skopiować wszystkie klasy i typy wyliczeniowe z aplikacji napisanej w rozdziale 8. (lub pobrać je z serwera FTP wydawnictwa Helion) i zaktualizować klasy `Player` oraz `Game`.

1 Dodaj istniejące pliki klas i dostosuj deklarowaną w nich przestrzeń nazw do bieżącej aplikacji.

Do swojego projektu dodaj następujące pliki, pochodzące z gry *Idź na ryby!*, którą napisałeś w rozdziale 8.: *Values.cs*, *Suits.cs*, *Card.cs*, *Deck.cs*, *CardComparer_bySuit.cs*, *CardComparer_byValue.cs*, *Game.cs* oraz *Player.cs*. Możesz skorzystać z opcji *Add Existing Item* dostępnej w oknie *Solution Explorer*, jednak w każdym z tych plików będziesz musiał **zmienić przestrzeń nazw,** dostosowując ją do przestrzeni używanej w projekcie (jak już robiłeś w przypadku kilku innych projektów w tej książce).

Spróbuj zbudować projekt. W plikach *Game.cs* oraz *Player.cs* powinny pojawić się błędy, przypominające te przedstawione poniżej:



2 Usuń odwołania do wszystkich klas i obiektów `WinForms` i dodaj do klasy `Game` odpowiednie instrukcje `using`.

Nie znajdujemy się już w świecie WinForms, dlatego też musisz usunąć z plików *Game.cs* oraz *Player.cs* instrukcje `using System.Windows.Forms`. Będziesz także musiał usunąć wszystkie odwołania do kontrolek `TextBox`. Oprócz tego konieczne będzie dodanie do klasy `Game` interfejsu `INotifyPropertyChaned` oraz kolekcji `ObservableCollection<T>`, a to oznacza, że na początku pliku *Game.cs* będziesz musiał dodać następujące instrukcje `using`:

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

3 Dodaj instancję klasy `Game` jako zasób statyczny i określ kontekst danych.

Zmodyfikuj kod XAML strony, dodając do niego instancję klasy `Game` jako zasób statyczny, i użyj jej jako kontekstu danych dla siatki zawierającej stronę Idź na ryby!, którą przygotowałeś we wcześniejszej części rozdziału. Oto kod XAML definiujący zasób statyczny: `<local:Game x:Name="game"/>`. Będziesz także potrzebował nowego konstruktora w klasie `Game`, gdyż zasoby mogą być tworzone tylko wtedy, gdy klasa definiuje konstruktor bezargumentowy:

```
public Game() {
    PlayerName = "Edek";
    Hand = new ObservableCollection<string>();
    ResetGame();
}
```

4 Do klasy `Game` dodaj publiczne właściwości, których użyjesz do zdefiniowania powiązań danych z kontrolkami.

Oto właściwości, które powiążesz z kontrolkami na stronie:

```
public bool GameInProgress { get; private set; }
public bool GameNotStarted { get { return !GameInProgress; } }
public string PlayerName { get; set; }
public ObservableCollection<string> Hand { get; private set; }
public string Books { get { return DescribeBooks(); } }
public string GameProgress { get; private set; }
```

5 Użyj mechanizmu wiązania danych, by aktywować i dezaktywować kontrolki `TextBox`, `ListBox` oraz `Button`.

Chcesz, by pole tekstowe *Imię* oraz przycisk *Rozpocznij grę* były aktywne wyłącznie wtedy, gdy gra jeszcze nie została rozpoczęta, oraz by kontrola `ListBox` *Ręka* i przycisk *Zażądaj karty* były aktywne tylko wtedy, gdy gra została już rozpoczęta. Dodasz do klasy `Game` kod określający wartość właściwości `GameInProgress`. Przyjrzyj się właściwości `GameNotStarted`. Przekonaj się, jak ona działa, a następnie dodaj następujące powiązania danych do kontrolek `TextBox`, `ListBox` oraz dwóch kontrolek `Button`.

Każdej z nich będziesz musiał użyć dwa razy.

```
IsEnabled="{Binding GameInProgress}"      IsEnabled="{Binding GameNotStarted}"
IsEnabled="{Binding GameInProgress}"      IsEnabled="{Binding GameNotStarted}"
```

6 Zmodyfikuj klasę `Player`, by zmuszała klasę `Game` do wyświetlania postępów w grze.

Wcześniejsza wersja klasy `Player`, używana w aplikacji WinForms, używała kontrolki `TextBox` przekazywanej w wywołaniu jej konstruktora. Zmień ją, by przekazywana była referencja typu `Game`, zapisywana następnie w polu prywatnym. (Spójrz na przedstawioną poniżej metodę `StartGame()`, by zobaczyć, jak ma być używany ten nowy konstruktor podczas dodawania graczy). Odszukaj wiersze zawierające odwołania do kontrolki `TextBox` i zastąp je wywołaniami metody `AddProgress()` obiektu `Game`.

7 Zmodyfikuj klasę `Game`.

Zmień typ wynikowy metody `PlayOneRound()` z `bool` na `void`, a następnie zmodyfikuj jej kod tak, by postępy w grze były wyświetlane przez metodę `AddProgress()`, a nie poprzez odwoływanie się do kontrolki `TextBox`. Jeśli gracz wygrał, wyświetl postęp gry, przywróć ją do stanu początkowego i zakończ działanie metody. W przeciwnym razie odśwież kolekcję `Hand` i wyświetl informacje o kartach gracza.

Będziesz także musiał dodać lub zaktualizować poniższe cztery metody i określić, do czego one służą i jak mają działać.

```
public void StartGame() {
    ClearProgress();
    GameInProgress = true;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    Random random = new Random();
    players = new List<Player>();
    players.Add(new Player(PlayerName, random, this));
    players.Add(new Player("Bartek", random, this));
    players.Add(new Player("Janek", random, this));
    Deal();
    players[0].SortHand();
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
    OnPropertyChanged("Books");
}

public void AddProgress(string progress)
{
    GameProgress = progress +
            Environment.NewLine +
            GameProgress;
    OnPropertyChanged("GameProgress");
}
```

```
public void ClearProgress() {
    GameProgress = String.Empty;
    OnPropertyChanged("GameProgress");
}
```

> Będziesz **także** musiał zaimplementować interfejs `INotifyPropertyChanged` i dodać tę samą metodę `OnPropertyChanged()`, którą dodałeś do klasy `MenuMaker`. Używają jej zaktualizowane metody i będzie jej także używać metoda `PullOutBooks()`.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}
```

Rozwiązania ćwiczeń

Poniżej przedstawiliśmy cały kod ukryty, który miałeś napisać:

```
private void startButton_Click(object sender, RoutedEventArgs e) {
    game.StartGame();
}
private void askForACard_Click(object sender, RoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
private void cards_DoubleTapped(object sender, DoubleTappedRoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
```

A oto zmiany, które miałeś wprowadzić w klasie `Player`:

```
class Player {
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private Game game;
    public Player(String name, Random random, Game game) {
        this.name = name;
        this.random = random;
        this.game = game;
        this.cards = new Deck(new Card[] { });
        game.AddProgress(name + " przyłączył się do gry");
    }
    public Deck DoYouHaveAny(Values value)
    {
        Deck cardsIHave = cards.PullOutValues(value);
        game.AddProgress(Name + " ma " + cardsIHave.Count + " " + Card.Plural(value, cardsIHave.Count));
        return cardsIHave;
    }
    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
        game.AddProgress(Name + " pyta czy ktoś ma " + Card.Plural(value, 1));
        int totalCardsGiven = 0;
        for (int i = 0; i < players.Count; i++) {
            if (i != myIndex) {
                Player player = players[i];
                Deck CardsGiven = player.DoYouHaveAny(value);
                totalCardsGiven += CardsGiven.Count;
                while (CardsGiven.Count > 0)
                    cards.Add(CardsGiven.Deal());
            }
        }
        if (totalCardsGiven == 0) {
            game.AddProgress(Name + " pobrał kartę z kupki.");
            cards.Add(stock.Deal());
        }
    }

    // ... reszta kodu klasy Player nie została zmieniona ...
```

Oto zmiany w kodzie XAML strony:

```
<Grid Grid.Row="1" Margin="120,0,60,60" DataContext="{StaticResource ResourceKey=game}" >
    <TextBlock Text="Imię" Margin="0,0,0,20"
        Style="{StaticResource SubheaderTextStyle}"/>
    <StackPanel Orientation="Horizontal" Grid.Row="1">
        <TextBox x:Name="playerName" FontSize="24" Width="500" MinWidth="300"
         Text="{Binding PlayerName, Mode=TwoWay}" IsEnabled="{Binding GameNotStarted}" />
        <Button x:Name="startButton" Margin="20,0" IsEnabled="{Binding GameNotStarted}"
            Content="Rozpocznij grę!" Click="startButton_Click" />
    </StackPanel>
    <TextBlock Text="Postępy gry"
        Style="{StaticResource SubheaderTextStyle}" Margin="0,20,0,20" Grid.Row="2" />
    <ScrollViewer Grid.Row="3" FontSize="24" Background="White" Foreground="Black"
            Content="{Binding GameProgress}" />
    <TextBlock Text="Grupy" Style="{StaticResource SubheaderTextStyle}"
            Margin="0,20,0,20" Grid.Row="4"/>
    <ScrollViewer FontSize="24" Background="White" Foreground="Black"
            Grid.Row="5" Grid.RowSpan="2" Content="{Binding Books}" />
    <TextBlock Text="Ręka" Style="{StaticResource SubheaderTextStyle}"
            Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>
    <ListBox Background="White" FontSize="24" Height="Auto" Margin="0,0,0,20"
            x:Name="cards" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"
            ItemsSource="{Binding Hand}" IsEnabled="{Binding GameInProgress}"
            DoubleTapped="cards_DoubleTapped" />
    <Button x:Name="askForACard" Content="Zażądaj karty" HorizontalAlignment="Stretch"
            VerticalAlignment="Stretch" Grid.Row="6" Grid.Column="2"
            Click="askForACard_Click" IsEnabled="{Binding GameInProgress}" />
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="5*"/>
        <ColumnDefinition Width="40"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto" MinHeight="150" />
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
</Grid>
```

Kontekstem danych dla siatki jest klasa Game, gdyż wszystkie powiązania odwołują się do właściwości tej klasy.

Ta kontrolka TextBox używa dwukierunkowego powiązania z właściwością PlayerName.

To jest procedura obsługi zdarzeń Click przycisku, który rozpoczyna grę.

Kontrolki ScrollViewer prezentujące postępy gry oraz grupy są powiązane z właściwościami Progress oraz Books.

Właściwość IsEnabled aktywuje i dezaktywuje kontrolkę. Jest to właściwość typu Boolean, można ją zatem powiązać z właściwością tego samego typu dostępną w kontrolce, by włączać ją i wyłączać na podstawie wartości właściwości.

Rozwiązania ćwiczeń

Poniżej przedstawiliśmy wszystkie zmiany, które należało wprowadzić w kodzie klasy Game, włącznie z kodem, który podaliśmy już w instrukcjach do ćwiczenia.

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

class Game : INotifyPropertyChanged {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;
    public bool GameInProgress { get; private set; }
    public bool GameNotStarted { get { return !GameInProgress; } }
    public string PlayerName { get; set; }
    public ObservableCollection<string> Hand { get; private set; }
    public string Books { get { return DescribeBooks(); } }
    public string GameProgress { get; private set; }

    public Game() {
        PlayerName = "Edek";
        Hand = new ObservableCollection<string>();
        ResetGame();
    }

    public void AddProgress(string progress) {
        GameProgress = progress + Environment.NewLine + GameProgress;
        OnPropertyChanged("GameProgress");
    }

    public void ClearProgress() {
        GameProgress = String.Empty;
        OnPropertyChanged("GameProgress");
    }

    public void StartGame() {
        ClearProgress();
        GameInProgress = true;
        OnPropertyChanged("GameInProgress");
        OnPropertyChanged("GameNotStarted");
        Random random = new Random();
        players = new List<Player>();
        players.Add(new Player(PlayerName, random, this));
        players.Add(new Player("Bartek", random, this));
        players.Add(new Player("Janek", random, this));
        Deal();
        players[0].SortHand();
        Hand.Clear();
        foreach (String cardName in GetPlayerCardNames())
            Hand.Add(cardName);
        if (!GameInProgress)
            AddProgress(DescribePlayerHands());
        OnPropertyChanged("Books");
    }
```

Te instrukcje są niezbędne, by móc korzystać z interfejsu INotifyPropertyChanged i klasy ObservableCollection.

Te właściwości są używane przez mechanizm wiązania danych.

To jest nowy konstruktor klasy Game. Tworzymy tylko jedną kolekcję i czyścimy ją w momencie przywracania początkowego stanu gry. Gdybyśmy tworzyli nowy obiekt, to formularz straciłby odwołanie do niego, a aktualizacje zostałyby przerwane.

Te metody zapewniają działanie mechanizmu wiązania danych w grze. Nowe wiersze są dodawane na górze, dzięki czemu wcześniejsze zdarzenia są przesuwane ku dołowi kontrolki ScrollViewer.

Oto metoda StartGame(), którą pokazaliśmy już wcześniej. Czyści ona postępy gry, tworzy graczy, rozdaje karty, a następnie aktualizuje postępy gry i grupy.

Wszystkie programy, które napisałeś podczas lektury tej książki, można przerobić na aplikacje dla Sklepu Windows napisane przy użyciu języka XAML. Można je jednak napisać na bardzo wiele sposobów, co jest szczególnie prawdziwe w przypadku korzystania z języka XAML! To właśnie dlatego pokazaliśmy Ci tak dużo kodu w ramach opisu ćwiczenia.

Ta metoda wcześniej zwracała wartość typu bool, dzięki której formularz mógł aktualizować przebieg gry. Teraz musi ona jedynie wywoływać metodę AddProgress(), a o resztę zatroszczy się mechanizm wiązania danych.

```
public void PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            AddProgress(players[i].Name + " ma nową grupę");
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        OnPropertyChanged("Books");
        players[0].SortHand();
        if (stock.Count == 0) {
            AddProgress("Na kupce nie ma już żadnych kart. Gra skończona!");
            AddProgress("Zwycięzcą jest... " + GetWinnerName());
            ResetGame();
            return;
        }
    }
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
}

public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}

public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}

// ... reszta kodu klasy Game nie została zmieniona ...
```

Grupy uległy zmianie, a formularz musi o tym wiedzieć, by mógł zaktualizować kontrolki ScrollViewer.

Oto zmiany wprowadzone w metodzie PlayOneRound(), które aktualizują postęp gry, gdy została ona zakończona, bądź aktualizują rękę i grupy, jeśli gra jeszcze trwa.

A to jest metoda ResetGame() z instrukcji do ćwiczenia. Czyści ona grupy, rękę i kupkę kart.

To jest standardowy wzorzec użycia właściwości PropertyChanged, przedstawiony już wcześniej w tym rozdziale.

Skorowidz

H

I

J

K

N

O

P

Z

Twoja przepustka do świata C#!

C#. Rusz głową!

C# to odpowiedź firmy Microsoft na odnoszący sukcesy język Java. Za pomocą C# możesz pisać przenośny kod, który Twoi klienci uruchomią w dowolnym systemie. Jest tylko jeden warunek — muszą mieć dostęp do środowiska uruchomieniowego .NET Framework, Mono lub DotGNU. Innymi słowy, C# spełnił marzenia programistów — raz stworzony kod można uruchomić bez dodatkowych nakładów na różnych platformach.

Najnowsze wydanie tej książki, należącej do cenionej serii „Rusz głową!", zostało zaktualizowane, poprawione oraz rozszerzone o dodatkowy projekt na platformę Windows Phone. Gdy pochłoniesz te kilkaset stron, zaczniesz swobodnie korzystać z języka C#, jego konstrukcji i możliwości. Książka charakteryzuje się doskonałą przejrzystością oraz przystępnie przedstawioną wiedzą. Znajdziesz tu elementy programowania obiektowego, operacje na plikach, obsługę wyjątków oraz pracę z wieloma wątkami — to tylko niektóre z poruszanych zagadnień. Ponadto przekonasz się, jak sprawnie stworzyć atrakcyjny interfejs użytkownika, oraz zrozumiesz, do czego służy język LINQ. Jest to wyśmienita pozycja dla wszystkich czytelników chcących rozpocząć przygodę z językiem C# oraz platformą .NET.

Poznaj możliwości języka C#!

Dzięki tej książce:

- przygotujesz swoje środowisko pracy
- poznasz składnię oraz konstrukcję języka C#
- zaznajomisz się z elementami programowania obiektowego
- wykorzystasz możliwości języka LINQ
- stworzysz projekt dla Windows Phone
- wykorzystasz potencjał platformy .NET

sięgnij po WIĘCEJ

KOD KORZYŚCI

helion.pl
księgarnia internetowa

Nr katalogowy: 20272

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-8311-6

9 788324 683116

Cena 99,00 zł

Informatyka w najlepszym wydaniu