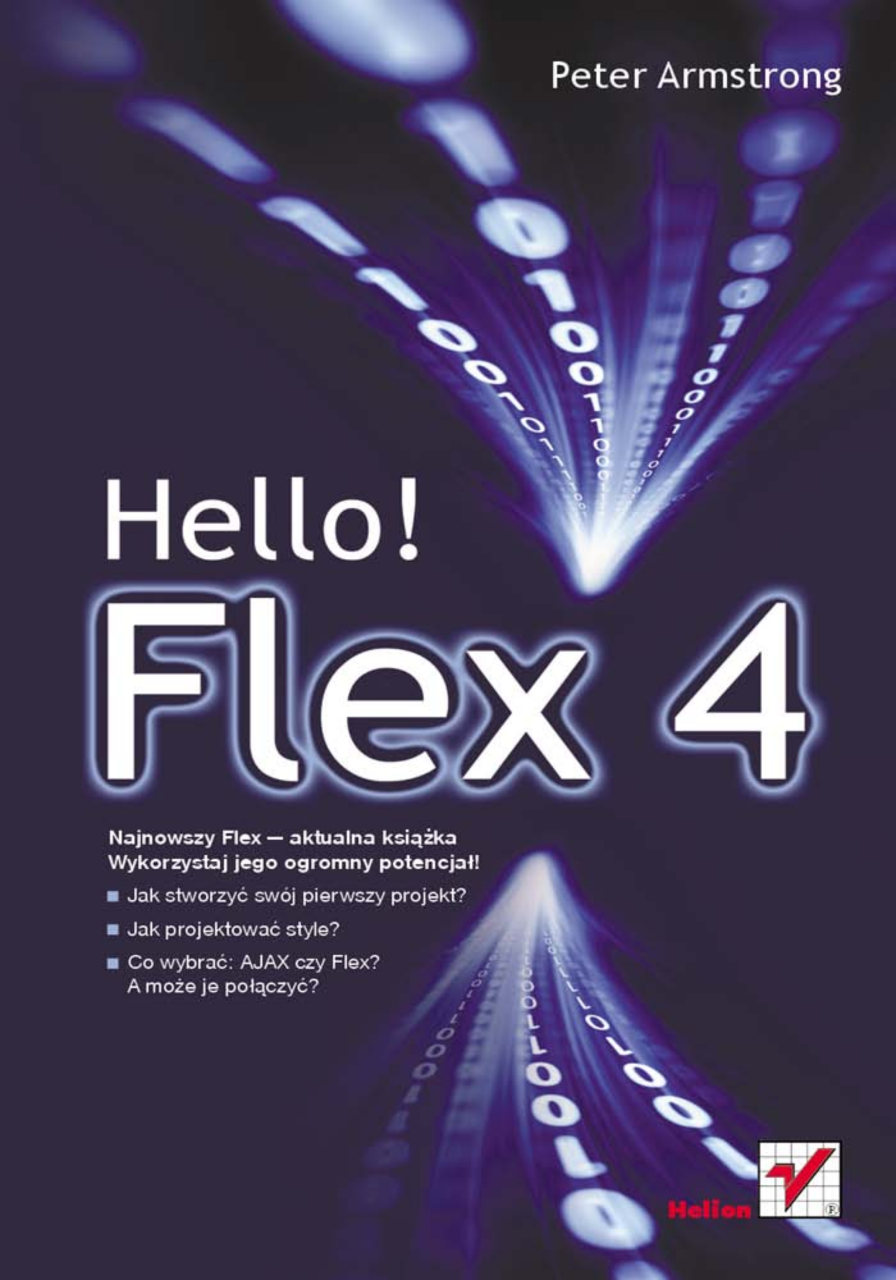


Peter Armstrong

Hello! Flex 4

Najnowszy Flex — aktualna książka
Wykorzystaj jego ogromny potencjał!

- Jak stworzyć swój pierwszy projekt?
- Jak projektować style?
- Co wybrać: AJAX czy Flex?
  A może je połączyć?

Helion

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

Hello! Flex 4

Autor: Peter Armstrong
Tłumaczenie: Krzysztof Sawka
ISBN: 978-83-246-2881-0
Tytuł oryginału: Hello! Flex 4
Format: 158×235, stron: 272

Najnowszy Flex – aktualna książka. Wykorzystaj szansę!

- Jak stworzyć swój pierwszy projekt?
- Jak projektować style?
- Co wybrać: AJAX czy Flex? A może je połączyć?

Flex w wersji 4 to najnowszy krzyk mody na rynku aplikacji webowych. Pozwala na tworzenie rozwiązań internetowych w niczym nieprzypominających tych, z którymi mamy do czynienia na co dzień. Są one atrakcyjne pod względem graficznym, charakteryzują się wysoką interakcyjnością oraz nieprawdopodobnie ogromnymi możliwościami. Jeżeli wspomnieć jeszcze o tym, że jest to środowisko zupełnie darmowe… pora przygotować się na rewolucję w świecie aplikacji internetowych!

Od marca 2010 roku dostępna jest najnowsza wersja Fleksa, oznaczona numerem 4. Warto poznawać tę wersję z podręcznikiem „Hello! Flex 4" w dłoni. Znajdziesz tu wszechstronną wiedzę na temat języków ActionScript, XML, E4X, obiektów pierwotnych oraz zasad projektowania stylów. Ponadto dowiesz się, jak tworzyć efekty, animacje oraz jak wykorzystać obiekty typu DataGrid. Jednak zanim dojdziesz do tych interesujących, lecz złożonych zagadnień, będziesz miał okazję zapoznać się z samym środowiskiem oraz najlepszymi praktykami i metodami pracy. Jeżeli chcesz rozpocząć przygodę z najnowszą wersją środowiska Flex – nie mogłeś lepiej trafić! Rozpocznij ją już dziś!

- Pierwsze kroki we Fleksie – omówienie platformy
- Nasłuchiwanie zdarzeń
- Wiązanie danych – adnotacja Bindable
- Struktura aplikacji pisanych w środowisku Flex
- Języki ActionScript, XML, E4X
- Definiowanie zmiennych i przestrzenie nazw
- Obiekty, tablice oraz sterowanie przepływem
- Dziedziczenie i inne pojęcia obiektowe w środowisku Flex
- Składniki typu Spark
- Stany widoku
- Projektowanie stylów
- Tworzenie animacji i efektów specjalnych
- Wykorzystanie obiektów typu DataGrid
- Zastosowanie pojemników nawigacyjnych oraz elementów wyskakujących
- Projektowanie i tworzenie formularzy

Rozpocznij przygodę z Flex 4 bez najmniejszych problemów!

Spis treści

6

Tworzenie łatwych w obsłudze formularzy za pomocą formaterów i analizatorów poprawności w środowisku Flex

W tym rozdziale nauczymy się obsługi formaterów i analizatorów poprawności, dzięki którym tworzenie formularzy do wprowadzania danych staje się przyjemnością — przynajmniej w porównaniu z innymi środowiskami programistycznymi. Formatery środowiska Flex są używane głównie do formatowania danych wyświetlanych użytkownikowi w kontrolkach takich jak przedstawiony w rozdziale 5. obiekt `DataGrid`. Mogą one również być wykorzystywane do pobierania danych wprowadzanych przez użytkownika i przekształcania ich w poprawnie sformatowane dane wejściowe. Analizatory poprawności (ang. *validator*) służą w tym środowisku do sprawdzania poprawności wprowadzanych danych oraz wyświetlania odpowiednich komunikatów w przypadku wprowadzenia nieprawidłowych informacji.

Dokumentacja interfejsów API dotycząca formaterów i analizatorów poprawności jest pod wieloma względami znakomita, jednak jej słabym punktem zawsze było przedstawienie jednoczesnego zastosowania wspomnianych elementów dla jednej kontrolki wykorzystywanej przez użytkownika. Istnieje pewne dobre usprawiedliwienie: wcale nie jest tak łatwo przeprowadzić taką czynność poprawnie! Jeżeli jednak zależy nam, aby formularze były jak najbardziej przydatne, jest to droga, na którą prędzej czy później musimy wstąpić. Jeśli tak się stanie, czas, jaki zaoszczędzicie po przeczytaniu tego rozdziału, może być wart ceny całego podręcznika.

Już od czasów środowiska Flex 1.0 obsługa formatowania była w nim przemyślana dobrze, a obsługa analizatorów poprawności — jeszcze lepiej. Największą innowacją od tamtej pory jest wprowadzenie do klas analizatorów poprawności właściwości `id` oraz możliwość powiązania danych z ich właściwościami. Zatem proces tworzenia aplikacji integrującej formatowanie i analizę poprawności w tych samych składnikach został ułatwiony.

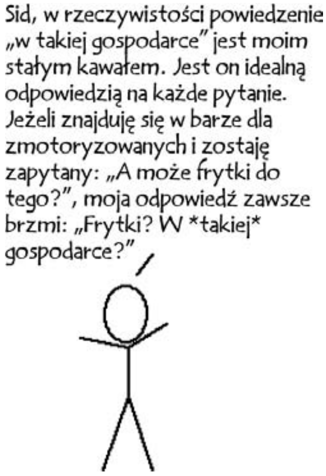
W takiej gospodarce warto mieć asa w rękawie!
W takiej gospodarce? Nie możecie przestać marudzić, prawda? No cóż, przynajmniej moje pokolenie będzie miało emeryturę! MUA, HA, HA, HA, HA...

Rozpoczniemy od utworzenia malutkiej aplikacji próbnej, która będzie bezpośrednio wykorzystywała wbudowane formatery i analizatory poprawności. Naszym zadaniem będzie zaobserwowanie, w jaki sposób te obiekty działają bez naszej dodatkowej pomocy. Przejdziemy następnie do nieco bardziej skomplikowanego zadania i napiszemy formularz `AddressForm`, w którym zostanie zaprezentowana współpraca formaterów i analizatorów poprawności w prawdziwej aplikacji. To zadanie wymaga z naszej strony nieco wysiłku: formularz `AddressForm` składa się z około 180 linijek kodu, więc jest to najbardziej złożony przykład, z jakim mieliśmy dotychczas do czynienia. Nam to jednak nie przeszkadza — lepiej zakończyć ostatnie samodzielne sesje wielką eksplozją niż chlipaniem. Poza tym, chociaż omawiany kod może wydawać się nudny, dla wielu programistów aplikacji w środowisku Flex może okazać się prawdziwym asem w rękawie podczas poszukiwania pracy. Nie wspominając o tym, że istnieje wiele możliwości strzelenia sobie w stopę na etapie wiązania danych podczas integracji formatowania z analizą poprawności, przez co ten rozdział okazuje się bardzo na miejscu.

W następnym rozdziale wprowadzimy jeszcze dłuższy przykład, w którym przez około 50 stron będziemy tworzyć połączenie serwisów Twitter i Yahoo! Maps, zatem możemy uznać przeprawę przez kilka stron poświęconych kodowi formularza za przystawkę przed głównym daniem. Wreszcie, będziemy korzystać w formularzu `AddressForm` ze składników `Form` i `FormItem` typu Halo — nie powiedzieliśmy więc jeszcze wszystkiego o tej architekturze.

Na początek jednak, zgodnie z obietnicą, napiszmy niewielki przykład wykorzystujący wbudowane formatery i analizatory poprawności.

SESJA 25. Formatery i analizatory poprawności


Sid, w rzeczywistości powiedzenie „w takiej gospodarce" jest moim stałym kawałem. Jest on idealną odpowiedzią na każde pytanie. Jeżeli znajduję się w barze dla zmotoryzowanych i zostaję zapytany: „A może frytki do tego?", moja odpowiedź zawsze brzmi: „Frytki? W *takiej* gospodarce?"

Obecną sesję rozpoczniemy od sprawdzenia możliwości wbudowanych formaterów i analizatorów poprawności w środowisku Flex. Mógłbym teraz zaprezentować olbrzymi przykład wykorzystujący wszystkie standardowe formatery i analizatory poprawności, jednak nie ma to sensu, gdyż działają one na tej samej zasadzie (z wyjątkiem analizatora `CreditCardValidator`; szczegóły znajdziemy w dokumentacji interfejsów API).

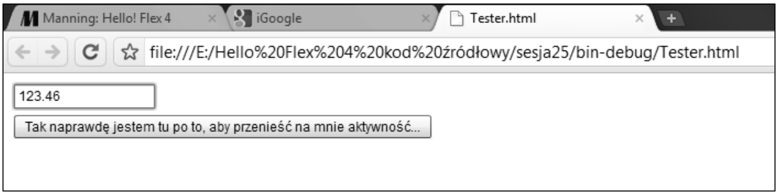
Wszystkie formatery są podklasami węzła `mx.formatters.Formatter`. W środowisku Flex 4 znajdziemy w tej klasie następujące podklasy: `CurrencyFormatter`, `DateFormatter`, `NumberFormatter`, `PhoneFormatter` i `ZipCodeFormatter`. Klasa `Formatter` definiuje metodę `format()`, która musi zostać przesłonięta przez jej używane podklasy.

W analogiczny sposób wszystkie analizatory poprawności stanowią podklasy węzła `mx.validators.Validator`, który pozwala na implementację analizy poprawności wobec danego pola poprzez ustanowienie wartości `true` we właściwości `required` obiektu `Validator`. W środowisku Flex 4 mamy do czynienia z następującymi podklasami klasy `Validator`: `CreditCardValidator`, `CurrencyValidator`, `DateValidator`, `EmailValidator`, `NumberValidator`, `PhoneNumberValidator`, `RegExpValidator`, `SocialSecurityValidator`, `StringValidator`, `StyleValidator` oraz `ZipCodeValidator`.

Skoro więc mechanizm ich działania jest taki sam, w tej sesji przyjrzymy się jednemu formaterowi i jednemu analizatorowi poprawności: odpowiednio `CurrencyFormatter` i `CurrencyValidator`. Wybrałem je, ponieważ przedstawienie ich interakcji nie stanowi problemu, a formatowanie walut może być przez Was bardzo pożądane. Zbudujemy niewielki programik, w którym obydwa obiekty będą wykorzystywane wobec tego samego pola `TextInput` typu Spark. Podczas tworzenia formularza adresowego w następnej sesji poznamy wiele innych rodzajów formaterów i analizatorów poprawności.

Wygląd tworzonej przez nas aplikacji podczas wprowadzania tekstu w pole `TextInput` został zaprezentowany na rysunku 6.1.

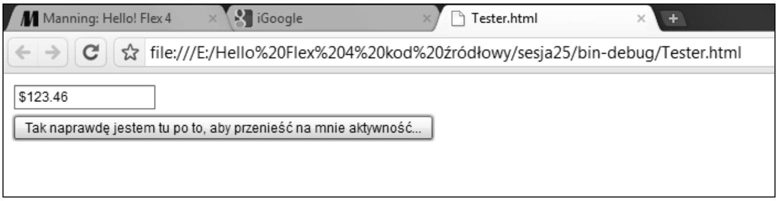
Rysunek 6.1. Wprowadzanie tekstu w formularzu walutowym


Manning: Hello! Flex 4
iGoogle
Tester.html
file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja25/bin-debug/Tester.html
Tak naprawdę jestem tu po to, aby przenieść na mnie aktywność...

123.46

Po przeniesieniu aktywności (za pomocą klawisza *Tab*) obiekt `CurrencyFormatter` sformatuje tekst, co widać na rysunku 6.2.

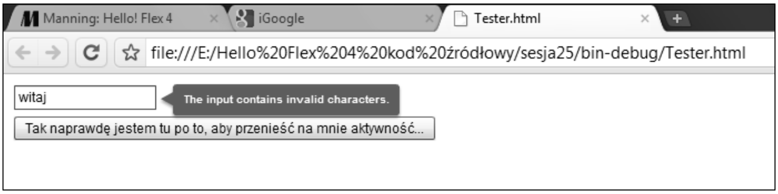
Rysunek 6.2. Sformatowanie wprowadzonego tekstu


Manning: Hello! Flex 4
iGoogle
Tester.html
file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja25/bin-debug/Tester.html
Tak naprawdę jestem tu po to, aby przenieść na mnie aktywność...

$123.46

Jeżeli wprowadzimy niepoprawne dane i przeniesiemy aktywność, ich wartość merytoryczna zostanie zakwestionowana i analizator poprawności wyświetli komunikat widoczny na rysunku 6.3.

Rysunek 6.3. Odpowiedź na wprowadzenie nieprawidłowych danych


Manning: Hello! Flex 4
iGoogle
Tester.html
file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja25/bin-debug/Tester.html
Tak naprawdę jestem tu po to, aby przenieść na mnie aktywność...

witaj

The input contains invalid characters.

Widzimy tu implementację wielu funkcji, które w niektórych środowiskach programistycznych wymagałyby olbrzymiej ilości kodu. Spójrzmy, jak sobie z tym radzi środowisko Flex (listing 6.1).

Listing 6.1. sesja25/src/Tester.mxml

```
   <?xml version="1.0" encoding="utf-8"?>
   <s:Application
     xmlns:fx="http://ns.adobe.com/mxml/2009"
     xmlns:s="library://ns.adobe.com/flex/spark"
     xmlns:mx="library://ns.adobe.com/flex/mx"
     width="100%" height="100%">
   <fx:Script><![CDATA[
❶    protected function moneyTIFocusOutHandler(event:FocusEvent):void {
       var formattedText:String =
         currencyFormatter.format(moneyTI.text);
❷      if (formattedText != "") {
         moneyTI.text = formattedText;
       }
     }
   ]]></fx:Script>
   <fx:Declarations>
❸    <mx:CurrencyFormatter id="currencyFormatter"
       precision="2" rounding="nearest"/>
❹    <mx:CurrencyValidator id="currencyValidator" precision="2"
       source="{moneyTI}" property="text" triggerEvent="focusOut"/>
   </fx:Declarations>
     <s:layout>
       <s:VerticalLayout paddingTop="10" paddingLeft="10"/>
     </s:layout>
❺    <s:TextInput id="moneyTI"
       focusOut="moneyTIFocusOutHandler(event)"/>
     <s:Button label="Tak naprawdę jestem tu po to, aby przenieść na mnie
     ➥aktywność..."/>
   </s:Application>
```

❶ Ta funkcja formatuje tekst w reakcji na zdarzenie `focusOut`.

❷ Przydzielamy tekst do obiektu `moneyTI.text` jedynie po jego pomyślnym sformatowaniu.

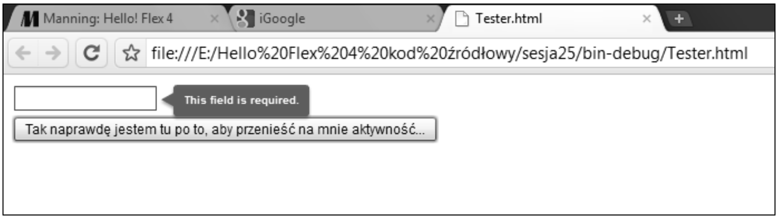
❸ Formater `CurrencyFormatter` będzie zaokrąglał kwotę do pełnego centa.

❹ Analizator poprawności `CurrencyValidator` analizuje wartość właściwości `text` obiektu `moneyTI`.

❺ Tutaj obsługujemy zdarzenie `focusOut` pola `textInput` obiektu `moneyTI`.

Zwróćmy uwagę, że gdybyśmy nie wprowadzili instrukcji dozoru ❷ wewnątrz procedury `moneyTIFocusOutHandler` ❶, formater `CurrencyFormatter` ❸ przydzielałby tekst do obiektu `moneyTI` ❺ w każdym przypadku. Wynik nie byłby satysfakcjonujący, ponieważ nawet w przypadku wyświetlenia przez analizator `CurrencyValidator` ❹ błędu byłby to jedynie błąd polegający na braku wpisanych danych (ponieważ formater `CurrencyFormatter` zostałby uruchomiony przed analizatorem). Widać to na rysunku 6.4.

Rysunek 6.4. Komunikat o braku wpisanych danych


Manning: Hello! Flex 4
iGoogle
Tester.html
file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja25/bin-debug/Tester.html
Tak naprawdę jestem tu po to, aby przenieść na mnie aktywność...

This field is required.

Tak jest! Dowiedzieliśmy się, jak sprawić, aby formatery współpracowały z analizatorami poprawności, i mamy już pewność, że przy odrobinie ostrożności w sprawdzaniu danych wyjściowych formatera możemy utworzyć bardzo praktyczny interfejs użytkownika.

➤➝ Punkty do zapamiętania

Moim ulubionym stałym kawałem jest powiedzenie „Tak właśnie powiedziała”. Jest ono bardziej irytujące
To ci wyszło
Naprawdę stosujesz w takiej gospodarce seksistowskie stałe kawały?
Tak właśnie... Nieważne

- Formatery służą do eleganckiego formatowania danych.
- Analizatory poprawności są stosowane do sprawdzania, czy wartość odpowiada zdefiniowanym w nich kryteriom. Zazwyczaj są one używane wraz z polami `TextInput`, jednak mogą być również implementowane w takich obiektach jak `DropDownList` (przekonamy się o tym w następnej sesji).

- W razie niepowodzenia formatery zwracają pusty ciąg znaków, powinniśmy więc sprawdzać je pod tym kątem, zanim prześlemy ich wartość gdzieś dalej. W przeciwnym razie będzie analizowana zła wartość.
- Jak dowiemy się z następnej sesji, możemy pisać własne formatery i analizatory poprawności.

SESJA 26. Praktyczne formularze, formatery i analizatory poprawności

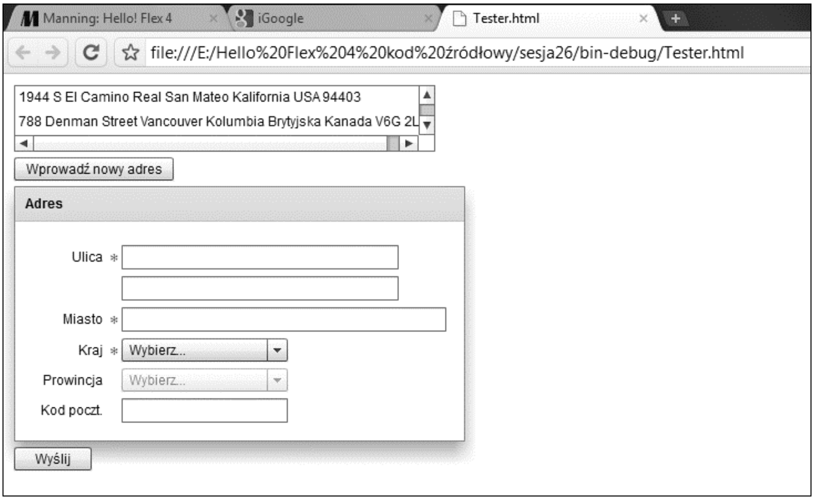
W tej sesji pogłębimy naszą znajomość formaterów i analizatorów poprawności poprzez zbudowanie formularza `AddressForm`, który będzie tak podobny do normalnego formularza używanego w rzeczywistości, jak to tylko możliwe. Oczywiście w prawdziwym świecie nic nie jest tak proste jak w przypadku książkowego przykładu, więc ta sesja również nie będzie łatwa. Wykorzystamy tu wiele rodzajów analizatorów poprawności, zwłaszcza `NumberValidator`, `RegExpValidator`, `String`↪`Validator` i `ZipCodeValidator`. Wprowadzimy również klasę `ZipCodeFormatter`. W tabeli 6.1 zostało wyjaśnione zastosowanie każdej z wymienionych klas w naszym przykładzie.

Tabela 6.1. Klasy zastosowane w sesji 26.

Klasa	**Zastosowanie w sesji**
`NumberValidator`	Sprawdzanie, czy obiekty `DropDownList` posiadają wybrane wartości.
`RegExpValidator`	Przeprowadzanie niestandardowej analizy kanadyjskich kodów pocztowych.
`StringValidator`	Sprawdzanie, czy w polach `Ulica` i `Miasto` została wprowadzona minimalna ilość tekstu.
`ZipCodeValidator`	Sprawdzanie amerykańskich kodów pocztowych (ang. *zip code*).
`ZipCodeFormatter`	Formatowanie amerykańskich i kanadyjskich kodów pocztowych.

Rysunek 6.5 przedstawia wygląd aplikacji budowanej w tej sesji.

Tworzymy składnik `AddressForm` wielokrotnego użytku, element `Address`, który będzie w nim wykorzystywany, oraz aplikację `Tester` demonstrującą przełączanie się pomiędzy elementami `Address` i prawidłową odpowiedź formularza `Address`↪`Form` (podobnie jak pozostałe fragmenty kodu umieszczone w tej książce i ten jest objęty licencją MIT, możemy więc wprowadzić go do własnej aplikacji).

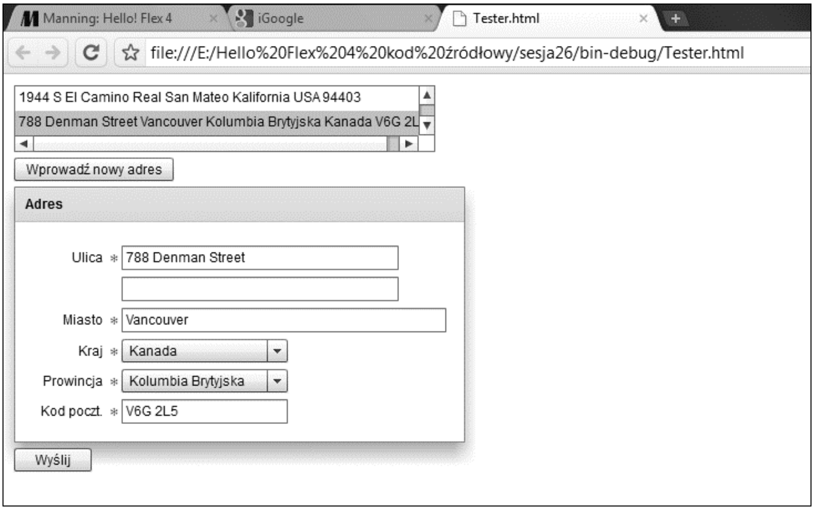


file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja26/bin-debug/Tester.html

1944 S El Camino Real San Mateo Kalifornia USA 94403
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

Rysunek 6.5. Przykładowy formularz adresowy

Jak zostało to pokazane na rysunku 6.6, po zaznaczeniu elementu `Address` umieszczonego w obiekcie `List` formularz `AddressForm` zostaje wypełniony danymi wybranego adresu.



file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja26/bin-debug/Tester.html

1944 S El Camino Real San Mateo Kalifornia USA 94403
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

Ulica * 788 Denman Street
Miasto * Vancouver
Kraj * Kanada
Prowincja * Kolumbia Brytyjska
Kod poczt. * V6G 2L5

Rysunek 6.6. Wprowadzenie do formularza gotowych danych adresowych

Zachowanie naszej aplikacji będzie przedstawiane na kolejnych rysunkach podczas omawiania kodu. Tymczasem przyjrzyjmy się na listingu 6.2 aplikacji `Tester`.

Listing 6.2. sesja26/src/Tester.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
```

```
      xmlns:mx="library://ns.adobe.com/flex/mx"
      xmlns:comp="components.*"
      width="100%" height="100%">
    <fx:Script><![CDATA[
      import mx.controls.Alert;
      import mx.collections.ArrayCollection;
      import model.Address;

      [Bindable]
❶     private var _addresses:ArrayCollection = new ArrayCollection([
        new Address("1944 S El Camino Real", "", "San Mateo",
          "Kalifornia", "USA", "94403"),
        new Address("788 Denman Street", "", "Vancouver",
          "Kolumbia Brytyjska", "Kanada", "V6G 2L5"),
        new Address("25 Oxford Street", "", "Londyn",
          "", "Wielka Brytania", "W1D 2DW"),
        new Address("21 Water Street", "#400", "Vancouver",
          "Kolumbia Brytyjska", "Kanada", "V6B 1A1")]);

❷     protected function submitClickHandler(event:MouseEvent):void {
        if (addressForm.validateAndSave()) {
          Alert.show("Dostanę teraz numer karty kredytowej?",
          ↪"Adres prawidłowy!");
        } else {
          Alert.show("Dostrzegam błędy", "O nie!");
        }
      }

❸     private function enterNewAddress():void {
        addressList.selectedItem = null;
      }
    ]]></fx:Script>
      <s:layout>
        <s:VerticalLayout paddingLeft="10" paddingTop="10" gap="5"/>
      </s:layout>
❹     <s:List id="addressList" dataProvider="{_addresses}"
        width="380" height="60"/>
      <s:Button label="Wprowadź nowy adres" click="enterNewAddress()"/>
      <s:Panel title="Adres">
        <s:layout>
          <s:VerticalLayout paddingLeft="5" paddingTop="5" gap="10"/>
        </s:layout>
        <comp:AddressForm id="addressForm"
❺         address="{addressList.selectedItem}"/>
      </s:Panel>
❻     <s:Button label="Wyślij" click="submitClickHandler(event)"/>
    </s:Application>
```

❶ Zmienna `_addresses` typu `ArrayCollection` przechowuje adresy, których będziemy używać (dwa pierwsze są adresami uwielbianych przeze mnie restauracji, odpowiednio Santa Ramen i Kintaro Ramen).

❷ Funkcja `submitClickHandler` sprawdza i zapisuje wprowadzony adres, a także wyświetla odpowiedni alert w oparciu o wynik.

❸ Funkcja `enterNewAddress` po prostu ustanawia wartość `null` dla elementu `selectedItem` i jednocześnie powoduje utworzenie wiązania z właściwością `address`.

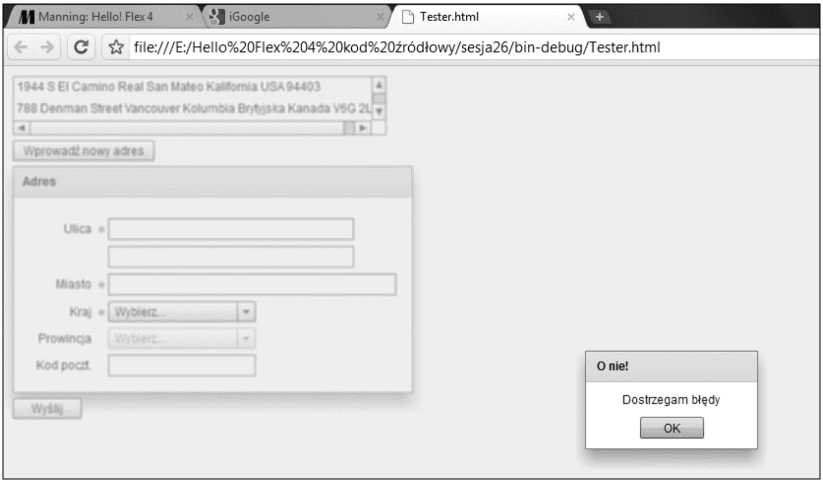
❹ Obiekt `addressList` posiada dostawcę `dataProvider` zmiennej `_addresses`.

❺ Formularz `addressForm` zawiera właściwość `address` powiązaną z właściwością `selectedItem` obiektu `addressList`.

❻ Zdarzenie typu `click` wobec przycisku `Wyślij` powoduje uruchomienie procedury `submitClickHandler`.

Rozpoczynamy od utworzenia tablicy `ArrayCollection` ❶ dla zmiennej `_addresses`, która pełni funkcję dostawcy danych `dataProvider` z listy `addressList` ❹. Wartość właściwości `selectedItem` z tej listy zostaje przekazana do tworzonego przez nas formularza `AddressForm` ❺. Posiadamy także przycisk `Wyślij` ❻, którego zdarzenie typu `click` jest obsługiwane przez funkcję wywołującą metodę `validateAndSave` ❷ formularza `AddressForm` oraz który powoduje wyświetlenie jednego z dwóch alertów w zależności od wyniku. Możemy wybrać również przycisk `Wprowadź nowy adres` wywołujący funkcję `enterNewAddress` ❸, która powoduje wyczyszczenie pól z wprowadzonymi danymi.

Rysunek 6.7 ilustruje zachowanie programu po wykryciu błędnych danych.



file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja26/bin-debug/Tester.html

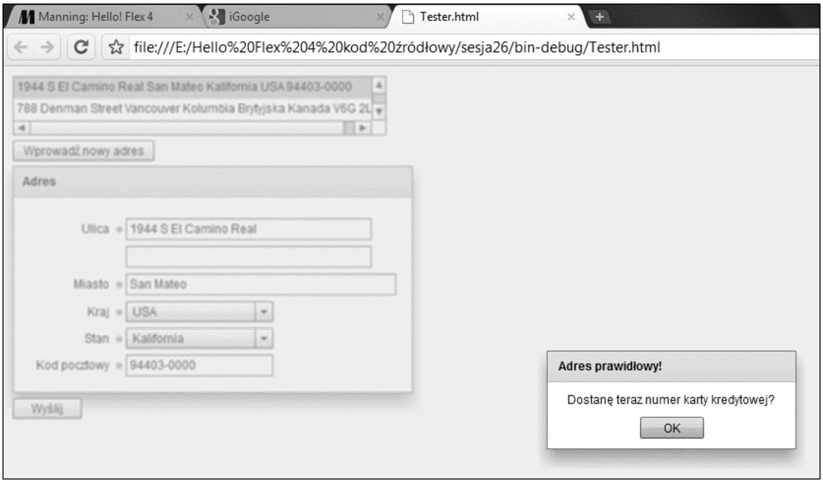
1944 S El Camino Real San Mateo Kalifornia USA 94403
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

O nie!

Dostrzegam błędy

Rysunek 6.7. Alert informujący o niewłaściwych danych

W podobny sposób przedstawiamy na rysunku 6.8 informacje o wprowadzeniu danych zakończonym sukcesem.



file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja26/bin-debug/Tester.html

1944 S El Camino Real San Mateo Kalifornia USA 94403-0000
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

Ulica * 1944 S El Camino Real
Miasto * San Mateo
Kraj * USA
Stan * Kalifornia
Kod pocztowy * 94403-0000

Adres prawidłowy!

Dostanę teraz numer karty kredytowej?

Rysunek 6.8. Informacja o wprowadzeniu poprawnego adresu

Stworzymy teraz model klasy `Address` (listing 6.3).

Listing 6.3. sesja26/src/model/Address.as

```
    package model {
❶     [Bindable]
      public class Address {
        public var lineOne:String;
        public var lineTwo:String;
        public var city:String;
        public var zipCode:String;
        public var state:String;
        public var country:String;

❷       public function Address(
          lineOne:String = "",
          lineTwo:String = "",
          city:String = "",
          state:String = "",
          country:String = "",
          zipCode:String = "") {
          this.lineOne = lineOne;
          this.lineTwo = lineTwo;
          this.city = city;
          this.state = state;
          this.country = country;
          this.zipCode = zipCode;
        }
        private function getAddrStr(str:String):String {
          return (str == null || str == "") ? "" : str + " ";
        }

    ❸   public function toString():String {
          return getAddrStr(lineOne) + getAddrStr(lineTwo) +
            getAddrStr(city) + getAddrStr(state) +
            getAddrStr(country) + getAddrStr(zipCode);
        }
      }
    }
```

Model `Address` nie jest skomplikowany: chcemy, aby można było powiązać dane z każdą zmienną, zatem wprowadzamy adnotację `[Bindable]` ❶ na początku klasy (tworzenie powiązanych danych generuje większą ilość kodu, zatem nie powinniśmy go nadużywać). Po drugie, tworzymy konstruktor ❷ posiadający domyślne wartości (puste ciągi znaków) dla wszystkich swoich parametrów. Możemy w ten sposób określić niektóre z tych parametrów (lub nie zdefiniować żadnego z nich) podczas tworzenia nowego obiektu `Address`. Wreszcie tworzymy funkcję `toString` ❸, która wykorzystuje metodę `getAddrStr` służącą do pomijania dodatkowych spacji w przypadku obiektów `Address` posiadających wypełnionych tylko kilka pól (tak, na samym końcu znajdzie się jedna niepotrzebna spacja; ćwiczeniem dla czytelników będzie pozbycie się jej).

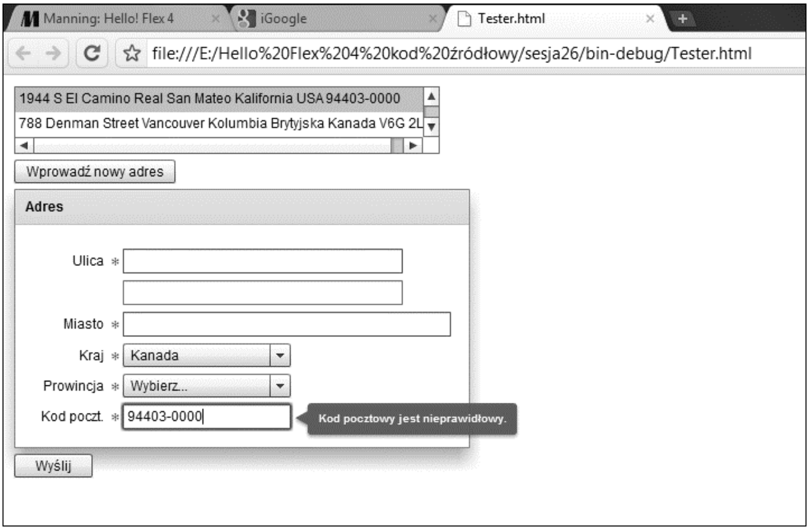
Podobnie jak w języku Java, metoda `toString` zostaje wywołana zawsze wtedy, gdy obiekt ma zostać zaprezentowany jako obiekt typu `String`, często natomiast jest przesłaniana w sposób umożliwiający bardziej użyteczne przedstawienie danych. Zwróćmy uwagę, że nie robię w tej metodzie `toString` nic złego, generalnie jednak kod modelu nie powinien definiować informacji dotyczących poziomu widoku.

Zanim utworzymy formularz `AddressForm`, zaprezentuję kilka zrzutów ekranu przedstawiających różne jego funkcje oraz wyświetlanie błędów walidacji.

Po pierwsze, rysunek 6.9 przedstawia sytuację, w której wpisywany jest kod pocztowy Stanów Zjednoczonych zamiast kanadyjskiego, co powoduje wyświetlenie odpowiedniej informacji (jest to niekonwencjonalne zachowanie analizatora `ZipCodeValidator`, które naprawiamy w tym przykładzie).

Zwróćmy także uwagę, że dla Kanady etykiety formularza mają nazwy `Prowincja` i `Kod poczt.`, a nie `Stan` i `Kod pocztowy` (zarezerwowane dla Stanów Zjednoczonych).

Rysunek 6.9. Efekt wpisania niewłaściwego kodu pocztowego



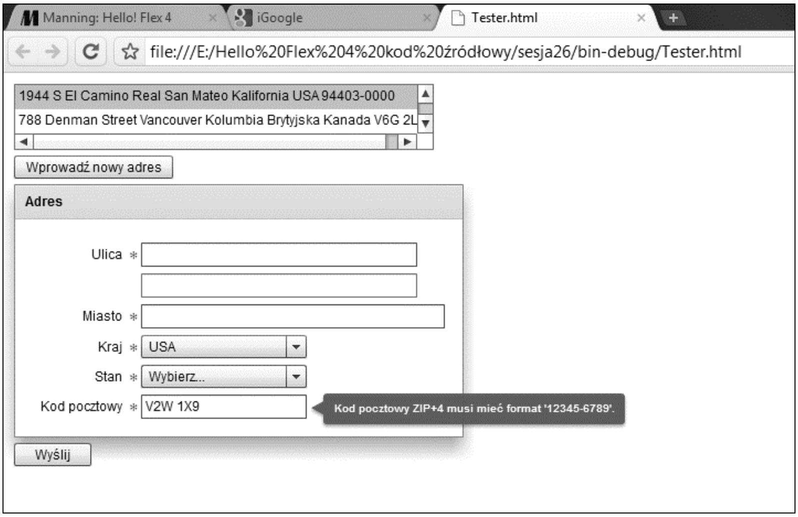
1944 S El Camino Real San Mateo Kalifornia USA 94403-0000
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

Kraj: Kanada
Prowincja: Wybierz...
Kod poczt.: 94403-0000

Kod pocztowy jest nieprawidłowy.

Odnotujmy teraz fakt, że jeżeli wpiszemy kanadyjski kod pocztowy w miejscu przeznaczonym na kod Stanów Zjednoczonych, zostanie wyświetlony błąd, a nieprawidłowe dane nie zostaną wyczyszczone przez formater (rysunek 6.10).

Rysunek 6.10. Kanadyjski kod pocztowy wpisany w miejsce amerykańskiego



1944 S El Camino Real San Mateo Kalifornia USA 94403-0000
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

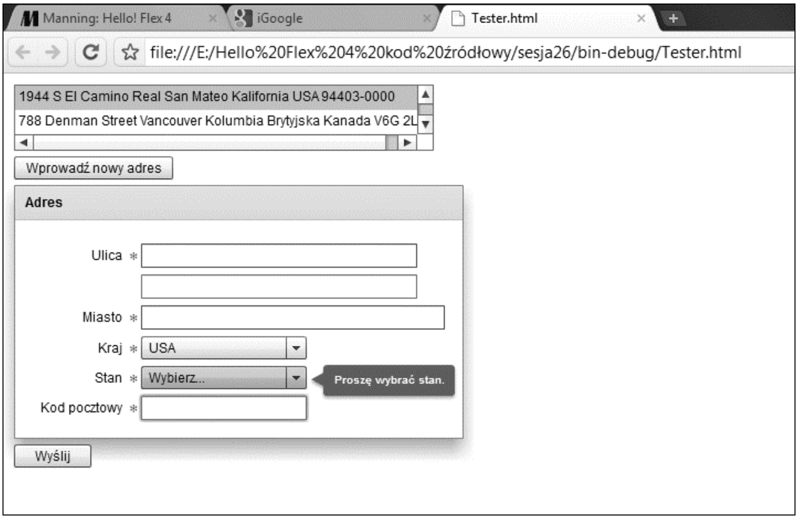
Kraj: USA
Stan: Wybierz...
Kod pocztowy: V2W 1X9

Kod pocztowy ZIP+4 musi mieć format '12345-6789'.

Po napisaniu tego przykładu ujrzycie formatery w akcji: będziecie mogli aktualizować kody pocztowe Stanów Zjednoczonych do nowego stylu 5+4, pisać kody pocztowe Kanady od dużej litery i dodawać spacje pomiędzy ich segmentami.

Tak, naprawdę jest to tak pasjonujące, jak się wydaje.

Wreszcie na rysunku 6.11 widzimy, że wykorzystujemy analizator poprawności również do wprowadzenia wymogu, aby użytkownik wybrał jeden ze stanów USA (lub prowincję w Kanadzie), a także wpisał odpowiedni kod pocztowy.



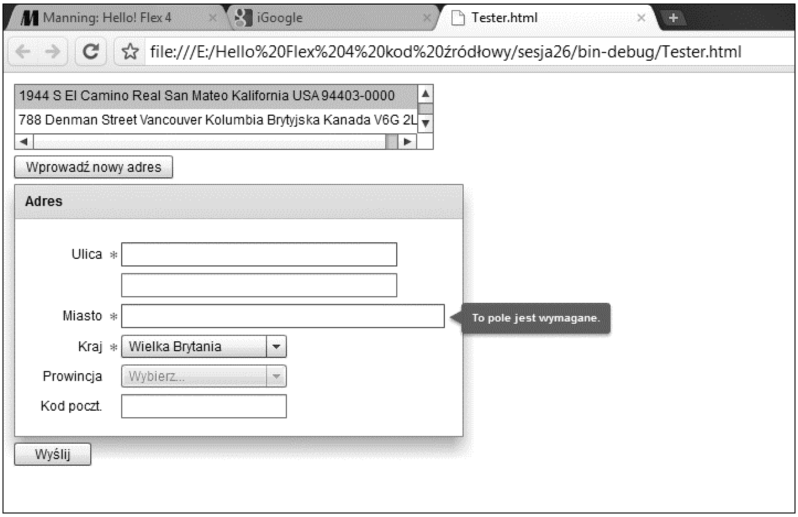
file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja26/bin-debug/Tester.html

1944 S El Camino Real San Mateo Kalifornia USA 94403-0000
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

Kraj: USA
Stan: Wybierz...
Proszę wybrać stan.

Rysunek 6.11. Prośba o wybór stanu

Dla krajów innych niż Stany Zjednoczone lub Kanada nie wymagamy wyboru prowincji czy wpisania kodu pocztowego, co widać na rysunku 6.12.



file:///E:/Hello%20Flex%204%20kod%20źródłowy/sesja26/bin-debug/Tester.html

1944 S El Camino Real San Mateo Kalifornia USA 94403-0000
788 Denman Street Vancouver Kolumbia Brytyjska Kanada V6G 2L

To pole jest wymagane.
Kraj: Wielka Brytania
Prowincja: Wybierz...

Rysunek 6.12. Obsługa państwa innego niż USA lub Kanada

Nie tylko zostają pominięte błędy, lecz także znikają czerwone gwiazdki oznaczające, że uzupełnienie danego pola jest wymagane.

Bez niepotrzebnego przedłużania przejdźmy do tworzenia formularza `AddressForm` (nareszcie!). Mamy tu do czynienia ze sporą ilością kodu, więc rozbijemy go na kilka części, z których każda zostanie dokładnie objaśniona. Zaczniemy od początku (listing 6.4).

Listing 6.4. sesja26/src/components/AddressForm.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Form
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  width="400">
<fx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  import mx.events.ValidationResultEvent;
  import mx.validators.Validator;
  import model.Address;

❶   private var _address:Address = new Address();

❷   public function validateAndSave():Boolean {
    if (isFormValid()) {
      address.lineOne = addressOneTI.text;
      address.lineTwo = addressTwoTI.text;
      address.city = cityTI.text;
      address.country = countryDDL.selectedItem;
      if (stateDDL.dataProvider.length == 0) {
        address.state = "";
      } else {
❸         address.state = stateDDL.selectedItem;
      }
      address.zipCode = zipTI.text;
      return true;
    } else {
      return false;
    }
  }
❹   private function isFormValid():Boolean {
    var validators:Array = [addressValidator, cityValidator,
      countryValidator, stateValidator];
    var zipCodeValid:Boolean = validateAndFormatZipCode();
    var results:Array = Validator.validateAll(validators);
    return results.length == 0 && zipCodeValid;
  }
❺   private function setFormFromAddress():void {
    addressOneTI.text = address.lineOne;
    addressTwoTI.text = address.lineTwo;
    cityTI.text = address.city;
    countryDDL.selectedItem = address.country;
    var states:ArrayCollection = getStates(address.country);
    stateDDL.dataProvider = states;
    stateDDL.selectedIndex = states.source.indexOf(address.state);
    zipTI.text = address.zipCode;
  }
...
```

❶ Zmienna `_address` przechowuje obiekt `Address` edytowany w formularzu `AddressForm`.

❷ Metoda `validateAndSave` jest wywoływana przez aplikację `Tester`. Wywołuje ona z kolei funkcję `isFormValid`, która sprawdza poprawność wszystkich składników formularza. Jeżeli składniki te są prawidłowe, adres zostaje zaktualizowany o umieszczone w nich wartości. Takie rozwiązanie zapobiega wprowadzeniu do obiektu `Address` nieprawidłowych lub tylko częściowo poprawnych danych.

❸ Przydzielamy dla danego stanu wartość `""`, jeśli wartość właściwości `selectedItem` jest równa `null` (nie wybraliśmy żadnego składnika), na przykład gdy lista stanów (prowincji) jest pusta dla danego kraju.

❹ Metoda `isFormValid` uruchamia analizatory poprawności poprzez utworzenie ich tablicy `Array` i wywołanie wobec niej funkcji `Validator.validateAll`. Aktywuje ona również oddzielnie analizator kodu pocztowego poprzez wywołanie funkcji o nazwie `validateAndFormatZipCode`, którą w dalszej części opisu przeanalizujemy. Jeżeli w wywołaniu funkcji `Validator.validateAll` pojawią się jakiekolwiek błędy walidacji, wartość obiektu `results.length` będzie niezerowa. Zwróćmy uwagę, że stosujemy tymczasową zmienną dla obiektu `zipCodeValid`, ponieważ nie chcemy, aby w trakcie analizy poprawności nastąpiło uproszczenie skutkujące pominięciem metody `validateAndFormatZipCode` (mamy zamiar wywołać naraz wszystkie analizatory poprawności i wyświetlić wszystkie błędy walidacji).

❺ Metoda `setFormFromAddress` aktualizuje stan formularza w oparciu o stan adresu. Ponieważ jest to przeprowadzane wewnątrz jednej metody, mamy pewność, że wybraliśmy właściwe państwo, zanim zaktualizujemy stany. Po przejrzeniu kodu formularza widocznego na listingu 6.5 znaczenie tego rozwiązania stanie się bardziej zrozumiałe.

Listing 6.5. sesja26/src/components/AddressForm.mxml (kontynuacja)

```
  …
❻ public function set address(value:Address):void {
      if (value == null) {
        _address = new Address();
        setFormFromAddress();
      } else {
        _address = value;
        setFormFromAddress();
        callLater(isFormValid);
      }
    }
    [Bindable]
    public function get address():Address {
      return _address;
    }

❼   private static const EMPTY:ArrayCollection =
      new ArrayCollection([]);
    private static const COUNTRIES:ArrayCollection =
      new ArrayCollection(["USA", "Kanada", "Wielka Brytania", "Francja"]);
    private static const STATES:ArrayCollection =
      new ArrayCollection(["Kalifornia", "Oregon", "Waszyngton"]);
    private static const PROVINCES:ArrayCollection =
      new ArrayCollection(["Kolumbia Brytyjska", "Alberta",
      ↪"Saskatchewan"]);

❽   private function getStates(country:String):ArrayCollection {
      if (isUSA(country)) {
        return STATES;
      } else if (isCanada(country)) {
        return PROVINCES;
      } else {
        return EMPTY;
      }
    }
❾   private function usaOrCanada(country:String):Boolean {
      return isUSA(country) || isCanada(country);
    }
    private function isUSA(country:String):Boolean {
      return country == "USA";
    }
    private function isCanada(country:String):Boolean {
      return country == "Kanada";
    }
    private function getStateMsg(country:String):String {
      return isUSA(country) ? "Proszę wybrać stan." :
        "Proszę wybrać prowincję.";
    }
  …
```

---

❻ Setter adresu tworzy nowy obiekt `Address`, jeżeli przekazana wartość wynosi `null`. W każdym przypadku po ustanowieniu tego adresu wywoływana jest funkcja `setFormFromAddress`, jednak tam, gdzie zostaje przekazana niezerowa wartość, wywołujemy także metodę `isFormValid` (została wcześniej zaprezentowana) poprzez wywołanie metody `callLater`. Za jej pomocą dajemy kontrolkom formularza czas na odzwierciedlenie nowych wartości, które zostały dla nich ustanowione (nie omawiałem w tym podręczniku metody `callLater`, gdyż stanowi ona zaawansowaną technologię. W ogólnym zarysie opóźnia ona uruchomienie funkcji — zostaje ona uruchomiona w następnym cyklu odświeżania ekranu — dzięki czemu wartości mogą zostać odzwierciedlone). Powodem przeprowadzania przez nas analizy poprawności jedynie w przypadku niezerowych wartości obiektu `Address` jest chęć uniknięcia wielu błędów walidacji w przypadku pustego formularza — wyglądałoby to po prostu brzydko (poza tym mogłoby być mylące, ponieważ użytkownik nie popełnił — jeszcze — żadnego błędu w pustym formularzu!). Tworzymy również pod metodą `isFormValid` getter adresu, który jest o wiele prostszy.

❼ Te stałe stanowią oczywiście „podręcznikowy przykład”. Na świecie istnieje o wiele więcej państw, stanów i prowincji, niż przedstawiłem to w aplikacji. Właśnie uratowałem drzewo przed przerobieniem na papier.

❽ Funkcja `getStates` zwraca wartość `STATES` dla Stanów Zjednoczonych, `PROVINCES` dla Kanady i `EMPTY` dla reszty świata. Jest to tak lubiane przez nas amerykocentryczne zachowanie; „międzynarodowi” czytelnicy mogą w razie potrzeby dowolnie modyfikować ten kod!

❾ Te cztery wygodne funkcje są wykorzystywane w formularzu do ukazywania (chowania) gwiazdek przy polach wymagających uzupełnienia, obok potomków obiektów `FormItem`. Wprowadziłem je do kodu, ponieważ wykorzystuję je w wiązaniu danych oraz dlatego, że są w prosty sposób odczytywane.

Czas na dalszą część kodu (listing 6.6).

Listing 6.6. sesja26/src/components/AddressForm.mxml (kontynuacja)

```
    …
❿      private function validateAndFormatZipCode():Boolean {
        var unformattedText:String =
          zipTI.text.toUpperCase().replace(/\W/g, "");
        var country:String = countryDDL.selectedItem;
        var result:ValidationResultEvent;
        var usa:Boolean = isUSA(country);
        var canada:Boolean = isCanada(country);
        zipCodeValidator.required = usa;
        postalCodeValidator.required = canada;
        if (usa) {
          postalCodeValidator.validate("");
⓫         result = zipCodeValidator.validate(unformattedText);
        } else if (canada) {
          zipCodeValidator.validate("");
⓬         result = postalCodeValidator.validate(unformattedText);
        } else {
          postalCodeValidator.validate("");
          zipCodeValidator.validate("");
          return true;
        }
⓭       if (result.type == ValidationResultEvent.VALID) {
          if (usa) {
            zipTI.text = zipCodeFormatter.format(unformattedText);
          } else {
            zipTI.text = postalCodeFormatter.format(unformattedText);
          }
          return true;
        } else {
          return false;
        }
      }
    ]]></fx:Script>
      <fx:Declarations>
⓮       <mx:StringValidator id="addressValidator" minLength="5"
          source="{addressOneTI}" property="text" required="true"
        requiredFieldError="To pole jest wymagane."/>
        <mx:StringValidator id="cityValidator" minLength="2"
          source="{cityTI}" property="text" required="true"
        requiredFieldError="To pole jest wymagane."/>
⓯       <mx:NumberValidator id="countryValidator"
          lowerThanMinError="Proszę wybrać kraj."
          source="{countryDDL}" property="selectedIndex" minValue="0"/>
        <mx:NumberValidator id="stateValidator"
⓰         lowerThanMinError="{getStateMsg(countryDDL.selectedItem)}"
```

```
      source="{stateDDL}" property="selectedIndex"
      enabled="{usaOrCanada(countryDDL.selectedItem)}"
      minValue="0"/>
⑰    <mx:ZipCodeFormatter id="zipCodeFormatter"
      formatString="#####-####"/>
⑱    <mx:ZipCodeFormatter id="postalCodeFormatter"
      formatString="### ###"/>
⑲    <mx:ZipCodeValidator id="zipCodeValidator"
      listener="{zipTI}"
    wrongUSFormatError="Kod pocztowy ZIP+4 musi mieć format '12345-6789'."
    requiredFieldError="To pole jest wymagane."/>
⑳    <mx:RegExpValidator id="postalCodeValidator"
      listener="{zipTI}"
      expression="^[A-Z]\d[A-Z]\d[A-Z]\d$"
      noMatchError="Kod pocztowy jest nieprawidłowy."/>
  </fx:Declarations>
…
```

⑩ Funkcja `validateAndFormatZipCode` zwraca wynik walidacji i analizy formatowania kodu pocztowego.

⑪ Dla Stanów Zjednoczonych zostaje uruchomiona funkcja `zipCodeValidator`.

⑫ Dla Kanady zostaje uruchomiona funkcja `postalCodeValidator`.

⑬ Jeżeli otrzymamy wartość `VALID`, zostaną uruchomione formatery. Dla Stanów Zjednoczonych zostanie uruchomiony formater `zipCodeFormatter`; dla Kanady będzie to `postalCodeFormatter` (ponieważ kody pocztowe są sprawdzane tylko dla Stanów Zjednoczonych i Kanady, uzasadnione jest wprowadzenie przypadku `else`).

⑭ Tworzymy wystąpienia analizatora `StringValidator` zapewniające zachowanie określonej minimalnej długości danych wejściowych. Obiekt `source` jest składnikiem przechowującym analizowaną właściwość.

⑮ Analizatory `countryValidator` i `stateValidator` są wystąpieniami analizatora `NumberValidator` uruchamianymi wobec właściwości `selectedIndex` list `DropDownList` zawierających dany kraj i stan. Tak, jest to powszechnie uważane za najlepsze rozwiązanie.

⑯ Analizator `stateValidator` jest dostępny wyłącznie dla Stanów Zjednoczonych lub Kanady.

⑰ Formater `zipCodeFormatter` wykorzystuje amerykański format kodów pocztowych 5+4.

⑱ Jakby poziom zagmatwania był zbyt mały, `postalCodeFormatter` jest formaterem `ZipCodeFormatter`.

⑲ Analizator `zipCodeValidator` wykorzystuje wbudowany analizator `ZipCodeValidator`.

⑳ Analizator `postalCodeValidator` wykorzystuje wbudowany analizator `RegExpValidator` do sprawdzania kanadyjskich kodów pocztowych za pomocą wyrażeń regularnych.

Przechodzimy do ostatniej części kodu (listing 6.7).

Listing 6.7. sesja26/src/components/AddressForm.mxml (kontynuacja)

```
   …
㉑   <mx:FormItem label="Ulica" required="true" width="100%">
       <s:TextInput id="addressOneTI" width="250"/>
       <s:TextInput id="addressTwoTI" width="250"/>
     </mx:FormItem>
     <mx:FormItem label="Miasto" required="true" width="100%">
       <s:TextInput id="cityTI" width="100%"/>
     </mx:FormItem>
     <mx:FormItem label="Kraj" required="true">
㉒     <s:DropDownList id="countryDDL" width="150"
         dataProvider="{COUNTRIES}" prompt="Wybierz..."
         change="validateAndFormatZipCode();"/>
     </mx:FormItem>
     <mx:FormItem
㉓     label="{isUSA(countryDDL.selectedItem) ? 'Stan' : 'Prowincja'}"
       required="{usaOrCanada(countryDDL.selectedItem)}">
       <s:DropDownList id="stateDDL" width="150"
㉔       dataProvider="{getStates(countryDDL.selectedItem)}"
         prompt="Wybierz..."
         enabled="{stateDDL.dataProvider.length > 0}"/>
     </mx:FormItem>
     <mx:FormItem
㉕     label="Kod {isUSA(countryDDL.selectedItem) ? 'pocztowy' : 'poczt.'}"
       width="100%" required="{usaOrCanada(countryDDL.selectedItem)}">
       <s:TextInput id="zipTI" width="150"
㉖       focusOut="validateAndFormatZipCode()"/>
     </mx:FormItem>
   </mx:Form>
```

㉑ Składniki `FormItem` stanowią jedynie elementy układu graficznego, podobnie jak sam pojemnik `Form`. Pojemnik ten nie posiada żadnych dodatkowych funkcji — w przeciwieństwie do formularzy języka HTML obecne w środowisku Flex obiekty `Form` typu Halo *są wyłącznie narzędziami układu graficznego*.

㉒ Nazwy państw znajdują się w obiekcie `countryDDL` listy `DropDownList`. Podczas zmiany państwa wywołujemy metodę `validateAndFormatZipCode()` w celu sprawdzenia kodu pocztowego.

㉓ Właściwość `selectedItem` obiektu `countryDDL` jest używana do sprawdzania, czy obiekty `FormItem` są wymagane. Jedynym skutkiem wprowadzenia tej właściwości jest wyświetlenie niewielkiej czerwonej gwiazdki. Nie ma ona *żadnego* wpływu na kontrolki znajdujące się we wnętrzu obiektu `FormItem`, chyba że jest wykorzystywana również przez te elementy.

㉔ Właściwość `dataProvider` obiektu `stateDDL` jest określana przez właściwość `selectedItem` obiektu `countryDDL`.

㉕ Właściwość `selectedItem` obiektu `countryDDL` jest używana do określenia etykiety obiektu `FormItem` dla pól `Kod pocztowy` (`Kod poczt.`) oraz `Stan` (`Prowincja`).

㉖ Zdarzenie `focusOut` obiektu `zipTI` uruchamia funkcję `validateAndFormat` ➥`ZipCode`, dzięki czemu błędy w kodzie pocztowym są natychmiast wychwytywane.

Ufffff!

To był bardzo długi przykład, przez który przewinęły się nawet wyrażenia regularne. Przepraszam za to — gdybym na samym początku powiedział, co Was tu czeka, moglibyście ominąć tę sesję! Całe szczęście w środowisku Flex znajduje się klasa `RegExpValidator` pozwalająca nam w bardzo łatwy sposób konstruować analizatory poprawności wykorzystujące wyrażenia regularne.

Kto by się spodziewał, że kody pocztowe mogą być tak skomplikowane (co więcej, całkowicie zignorowaliśmy całą resztę świata, więc właściwy przebieg tego ćwiczenia mógłby wyglądać jeszcze gorzej!)?

Pisałem niegdyś wyrażenia regularne pod górkę, w obydwie strony. W języku Perl

Punkty do zapamiętania

- Środowisko Flex zawiera pewną liczbę przydatnych formaterów i analizatorów poprawności.
- We Fleksie można (względnie) łatwo tworzyć niestandardowe składniki, które pozwalają na jednoczesne wykorzystywanie wbudowanych formaterów i analizatorów poprawności, dzięki czemu zwiększa się łatwość obsługi aplikacji.
- Należy być ostrożnym podczas stosowania wiązania danych wobec współzależnych składników formularza, na przykład wobec list `DropDownList` zawierających nazwy państw i prowincji. Możemy utworzyć formularz właściwie działający podczas modyfikowania nowego modelu obiektowego, ale nie rozwiąże to problemu obsługi modelu obiektowego ustanowionego w zewnętrznym świecie.
- W pewnych sytuacjach funkcja `callLater` może zostać wykorzystana do rozwiązania problemów z synchronizacją czasową interfejsu użytkownika. Rozwiązanie takie należy jednak stosować ostrożnie, ponieważ istnieje pokusa jego nadużywania przy jednoczesnym ignorowaniu leżących u podstaw problemów.
- Obiekty `Form` i `FormItem` stanowią *jedynie narzędzia układu graficznego*. Nie ma potrzeby, aby stosować je do wysyłania formularzy — inaczej niż ma to miejsce w języku HTML. Uznajmy je za pojemniki `VGroup` pozwalające na ładne rozmieszczenie pól formularza.

Miałeś luksus! Ja tworzyłem wyrażenia regularne w kodzie binarnym. Za pomocą kart perforowanych

Co dalej?

Bardzo ładnie, Sid. Dlaczego jednak zawsze, gdy zaczynacie rozmawiać na temat wyrażeń regularnych lub innych technicznych spraw, wasza dyskusja musi przekształcić się w... rywalizację?

Na przestrzeni całej książki staram się pokazać, że w środowisku Flex można w bardzo prosty sposób tworzyć niestandardowe składniki. W tej sesji zgłębiliśmy świat formaterów i analizatorów poprawności, tworząc między innymi własny analizator poprawności i złożony, niestandardowy formularz.

Nie wiem jak Wy, ale ja mam już zupełnie dość formularzy — pomimo ich przydatności. W następnym rozdziale zrobimy coś z zupełnie innej beczki i będziemy się wreszcie bawić podczas budowania naszego połączenia serwisów Twitter i Yahoo! Maps. Dowiemy się, jak należy budować prawdziwą aplikację, w jaki sposób można projektować większe programy w środowisku Cairngorm, a także jak można porozumiewać się z serwerami za pomocą klasy `HTTPService`.