 **INSTYTUT PODSTAW INFORMATYKI POLSKIEJ AKADEMII NAUK**

***PROGRAMOWANIE NA PLATFORMIE .NET***

**Programowanie aplikacji samodzielnych (WPF) - Warsztat 3**

**MVVM Tutorial from Start to Finish**

<http://www.software-architects.com/devblog/2010/09/10/MVVM-Tutorial-from-Start-to-Finish>

### MSDN Magazine

**Patterns - WPF Apps With The Model-View-ViewModel Design Pattern**

<https://web.archive.org/web/20150521014055/https://msdn.microsoft.com/en-us/magazine/dd419663.aspx> (tekst)

<https://web.archive.org/web/20090303100155/http://msdn.microsoft.com/en-us/magazine/dd419663.aspx> (kod)

<https://msdn.microsoft.com/en-us/magazine/dd419663.aspx> (2016)

**Data and WPF: Customize Data Display with Data Binding and WPF**

<http://web.archive.org/web/20130911065600/http://msdn.microsoft.com/en-us/magazine/cc700358.aspx> (tekst)

<http://web.archive.org/web/20090717162741/http://msdn.microsoft.com/en-us/magazine/cc700358.aspx> (kod)

<https://msdn.microsoft.com/pl-pl/magazine/cc700358.aspx> (2016 / PL)

**Advanced WPF: Understanding Routed Events and Commands In WPF**

<https://web.archive.org/web/20130911064828/http://msdn.microsoft.com/en-us/magazine/cc785480.aspx> (tekst)

<https://web.archive.org/web/20090331212758/http://msdn.microsoft.com/en-us/magazine/cc785480.aspx> (kod)

### Do przejrzenia

**Getting Started With MVVM in WPF**

<http://web.archive.org/web/20120705230636/http://codingtales.com/2010/01/12/getting-started-with-mvvm-in-wpf>

**MVVM Pattern in WPF**

<http://www.mindscapehq.com/products/wpfelements/mvvm-pattern-in-wpf>

### Gotowe części składowe - kod źródłowy

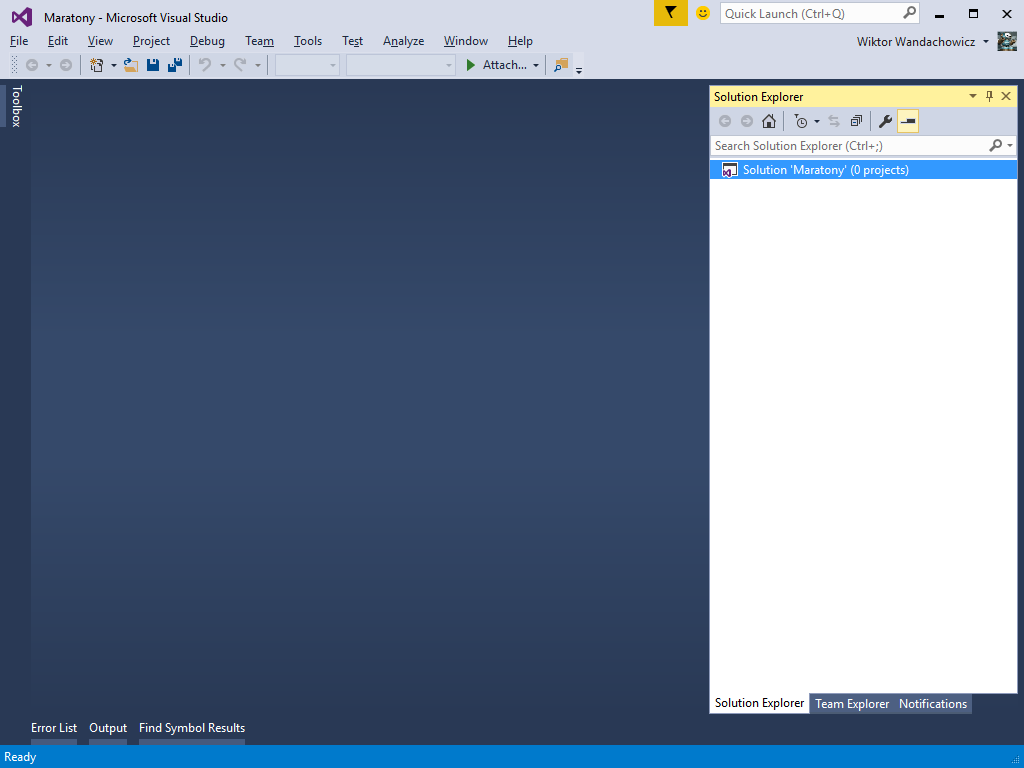
**MVVM Foundation**

<https://mvvmfoundation.codeplex.com/>

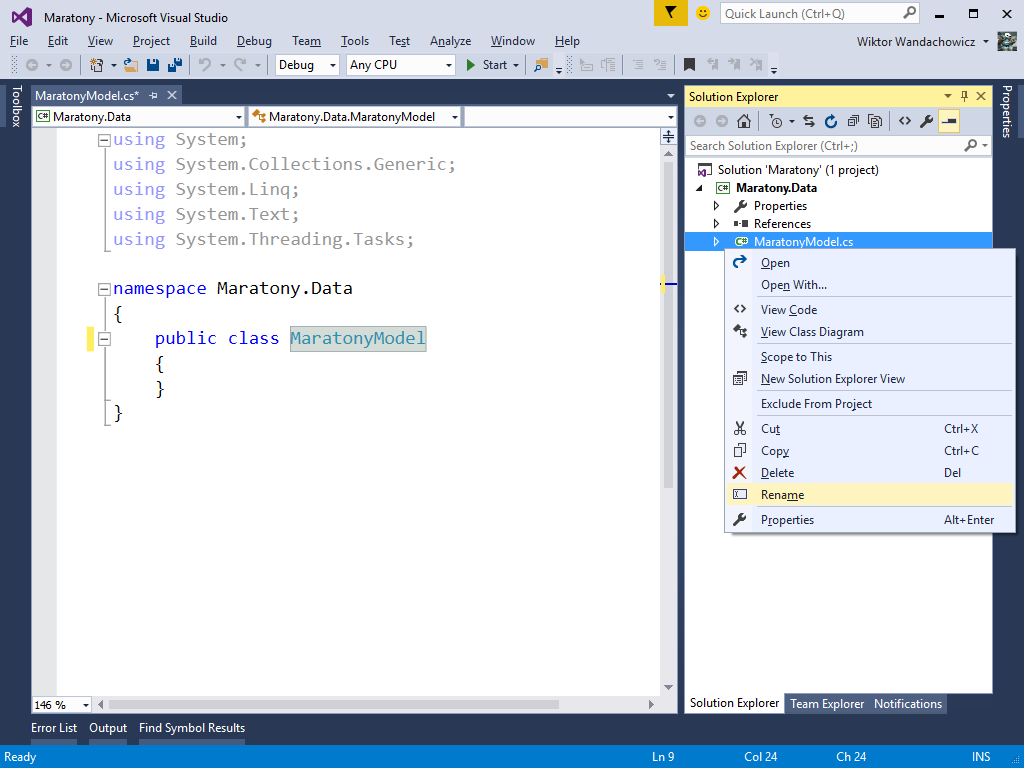
<https://mvvmfoundation.codeplex.com/SourceControl/latest>

#### Część 1 - Budowa modelu

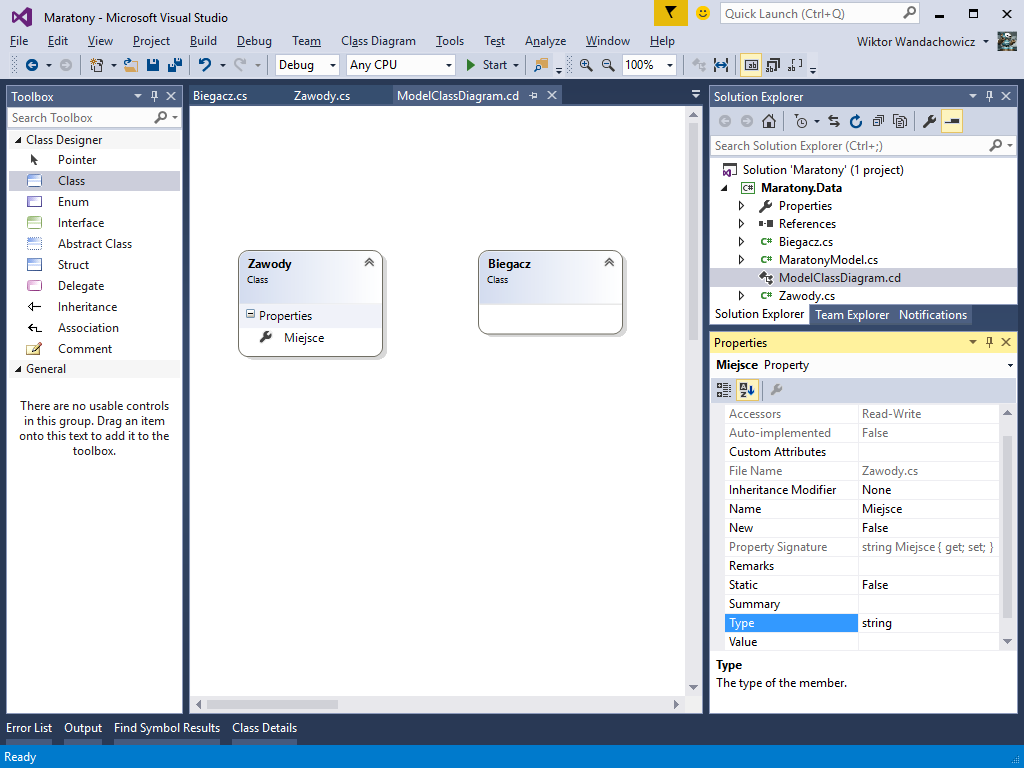
1. Utworzyć nowe, puste rozwiązanie:  
   File > New > Project…  
   Visual C# > Other Project Types > Visual Studio Solutions > **Blank Solution**  
   **Name: Maratony**  
   Location: *wybrany katalog*



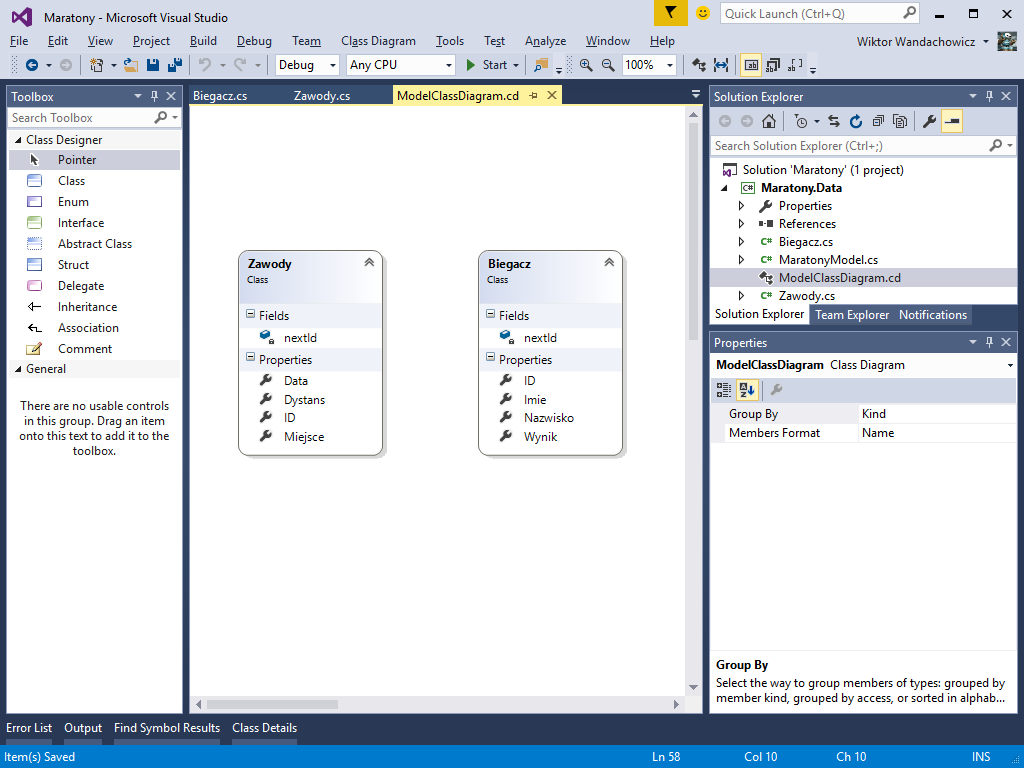
1. Dodać do rozwiązania projekt - bibliotekę z modelem danych:  
   File > Add > New Project…  
   Visual C# > **Class Library**  
   **Name: Maratony.Data**
2. Zmienić nazwę automatycznie dodanej klasy Class1 na MaratonyModel:  
   Solution Explorer > Maratony.Data > Class1 > [PPM] > Rename > **MaratonyModel**  
   potwierdzić zmianę nazwy klasy równocześnie przy zmianie nazwy pliku.



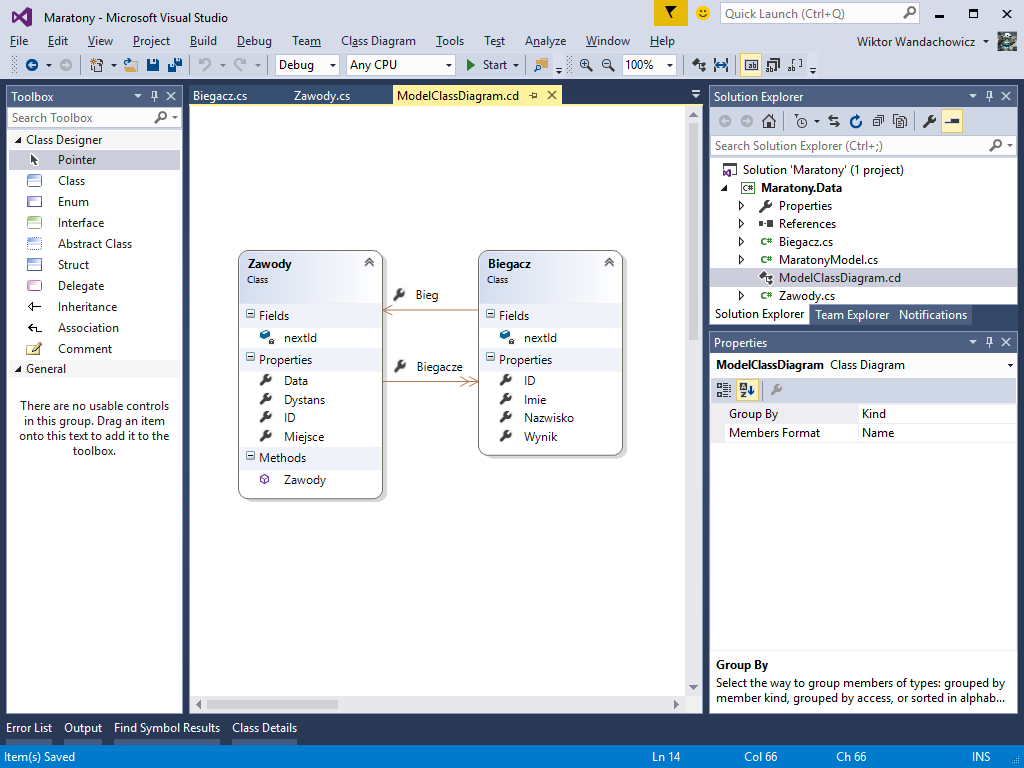
1. Do projektu dodać diagram klas:  
   Solution Explorer > Maratony.Data > [PPM] > Add > New Item…  
   Visual C# Items > Class Diagram  
   **Name: ModelClassDiagram**
2. Otworzyć Toolbox i przeciągnąć nowy element **Class** na diagram, podając:  
   **Class name: Zawody**  
   Przeciągnąć drugi element Class na diagram podając:  
   **Class name: Biegacz**
3. W klasie Zawody dodać *właściwość*:  
   Zawody > [PPM] > Add > Property > **Miejsce**  
   Otworzyć okno właściwości Properties, kliknąć na Miejsce i ustawić:  
   Type: **string**



1. W klasie Zawody dodać następne właściwości:
   1. Name: **Data**  
      Type: **System.DateTime**
   2. Name: **Dystans**  
      Type: **double**
   3. Name: **ID**  
      Type: **long**
2. W klasie Zawody dodać *pole*:  
   Zawody > [PPM] > Add > Field > **nextId**  
   Otworzyć okno właściwości Properties, kliknąć na **nextId** i ustawić:
   1. Type: **long**  
      Static: **True**  
      Inital Value: **0**
3. W klasie Biegacz dodać właściwości:  
   Biegacz > [PPM] > Add > Property > *nazwa*
   1. Name: **Imie**  
      Type: **string**
   2. Name: **Nazwisko**  
      Type: **string**
   3. Name: **Czas**  
      Type: **System.TimeSpan**
   4. Name: **ID**  
      Type: **long**
   5. Name: **Bieg**  
      Type: **Zawody**
4. W klasie Biegacz dodać *pole*:  
   Biegacz > [PPM] > Add > Field > **nextId**
   1. Type: **long**  
      Static: **True**  
      Inital Value: **0**



1. Dodać relacje między klasami Biegacz i Zawody:
   1. Klasa Zawody - dodać Property:  
      Name: **Biegacze**  
      Type: **List<Biegacz>**
   2. Wyświetlić właściwość jako relację kolekcji:  
      Zawody > Biegacze > [PPM] > **Show As Collection Association**
   3. Klasa Biegacz - wybrać Property:  
      Name: **Bieg**  
      Type: **Zawody**
   4. Wyświetlić właściwość jako relację:  
      Biegacz > Bieg > [PPM] > **Show As Association**



1. Dodać konstruktory:
   1. Klasa Zawody > [PPM] > Add > Constructor
   2. Klasa Biegacz > [PPM] > Add > Constructor
2. Zmodyfikować kod źródłowy w klasach, aby pola były inicjalizowane w konstruktorze:

public class Zawody

{

private static long nextId = 0;

public Zawody()

{

**this.ID = System.Threading.Interlocked.Increment(ref nextId);**

**Biegacze = new List<Biegacz>();**

}

// ... właściwości ...

public class Biegacz

{

private static long nextId = 0;

public Biegacz()

{

**this.ID = System.Threading.Interlocked.Increment(ref nextId);**

}

// ... właściwości ...

1. Zmodyfikować kod źródłowy w klasach, aby były używane właściwości automatyczne, np:

public class Zawody

{

// ...

public string Miejsce **{ get; set; }**

public DateTime Data **{ get; set; }**

public double Dystans **{ get; set; }**

public long ID **{ get; set; }**

public List<Biegacz> **Biegacze { get; set; }**

}

public class Biegacz

{

// ...

public string Imie **{ get; set; }**

public string Nazwisko **{ get; set; }**

public System.TimeSpan Wynik **{ get; set; }**

public long ID **{ get; set; }**

public Zawody Bieg **{ get; set; }**

}

1. W klasie MaratonyModel dodać pola przechowujące dane:

public class MaratonyModel

{

**private ObservableCollection<Zawody> listaZawodow;**

**private ObservableCollection<Biegacz> listaBiegaczy;**

}

1. Rozwiązać błąd nieznanego symbolu ObservableCollection:  
   postawić kursor na podkreślonym na czerwono symbolu i wcisnąć skrót [Ctrl]+[.] (kropka)  
   lub:  
   najechać myszą na podkreślony na czerwono symbol i kliknąć na ikonie żółtej żarówki,  
   wybrać propozycję: using System.Collections.ObjectModel;
2. W klasie MaratonyModel dodać konstruktor oraz właściwości read-only:

public class MaratonyModel

{

// ...

**public MaratonyModel()**

**{**

**this.listaZawodow = new ObservableCollection<Zawody>();**

**this.listaBiegaczy = new ObservableCollection<Biegacz>();**

**}**

**public ObservableCollection<Zawody> ListaZawodow**

**{**

**get { return listaZawodow; }**

**}**

**public ObservableCollection<Biegacz> ListaBiegaczy**

**{**

**get { return listaBiegaczy; }**

**}**

}

1. W klasie MaratonyModel dodać metody wykonujące operacje na zbiorach danych.
   1. Metoda DodajZawody() ma utworzyć nowy obiekt Zawody na podstawie podanych parametrów oraz dodać ten obiekt do kolekcji listaZawodow.
   2. Metoda DodajBiegacza() ma odnaleźć zawody na podstawie podanego identyfikatora, a następnie utworzyć nowy obiekt Biegacz z podanymi parametrami oraz dodać ten obiekt do kolekcji listaBiegaczy. Metoda ma jeszcze zapewnić poprawność dwustronnej relacji pomiędzy obiektem biegacza i zawodami (pole: Bieg), oraz pomiędzy obiektem zawodów i biegaczem (kolekcja: Biegacze).

public class MaratonyModel

{

// ...

public Zawody DodajZawody(string miejsce, DateTime data, double dystans)

{

Zawody z = new Zawody() { Miejsce = miejsce, Data = data, Dystans = dystans };

this.listaZawodow.Add(z);

return z;

}

public Biegacz DodajBiegacza(long zawodyId, string imie, string nazwisko)

{

Zawody zawody = listaZawodow.Where(z => (z.ID == zawodyId)).FirstOrDefault();

Biegacz b = new Biegacz() { Imie = imie, Nazwisko = nazwisko };

this.listaBiegaczy.Add(b);

b.Bieg = zawody;

zawody.Biegacze.Add(b);

return b;

}

}

#### Część 2 - Testy jednostkowe dla modelu danych

1. Dodać do rozwiązania projekt - testy jednostkowe:  
   File > Add > New Project…  
   Visual C# > Test > **Unit Test Project**  
   **Name: Maratony.Data.Test**
2. Dodać referencję do projektu zawierającego model danych:  
   Solution Explorer > Maratony.Data.Test > References > [PPM] > Add Reference…  
   Solution > ⌧ **Maratony.Data** -- zaznaczyć opcję i zatwierdzić [OK]
3. Zmienić nazwę automatycznie dodanej klasy UnitTest1 na MaratonyModelTest:  
   Solution Explorer > Maratony.Data.Test > UnitTest1 > [PPM] > Rename > **MaratonyModelTest**  
   potwierdzić zmianę nazwy klasy równocześnie przy zmianie nazwy pliku.
4. Zamienić pustą metodę TestMethod1 na TestNowyMaratonyModel\_PusteKolekcje:

[TestMethod]

public void TestNowyMaratonyModel\_PusteKolekcje()

{

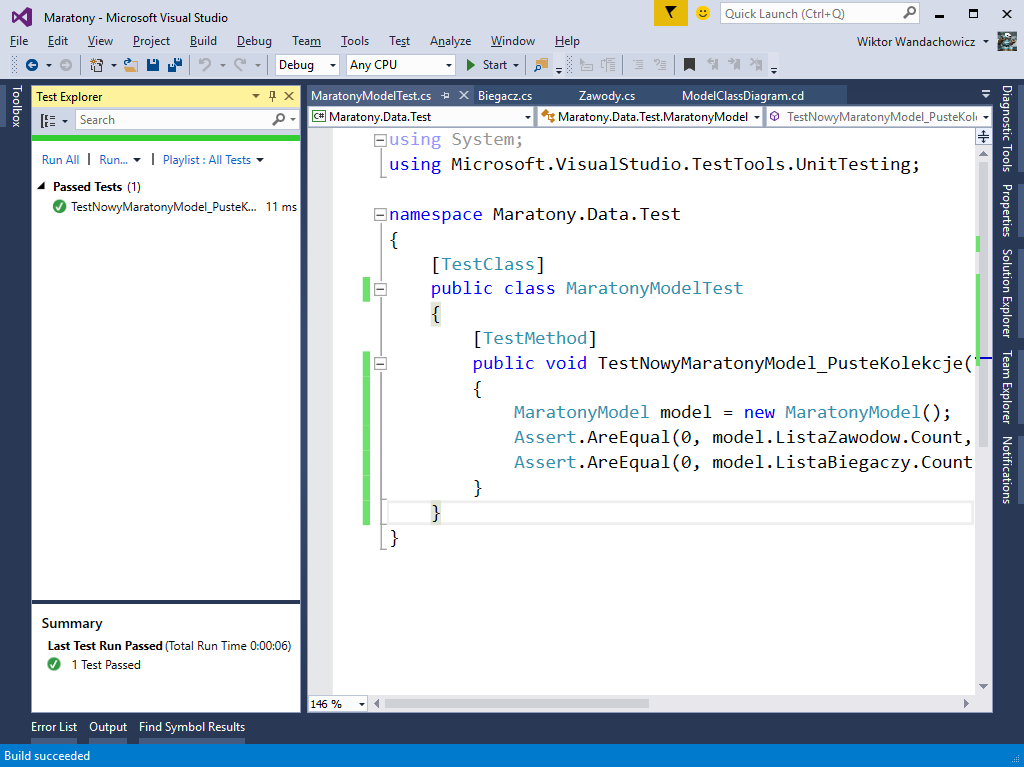
MaratonyModel model = new MaratonyModel();

Assert.AreEqual(0, model.ListaZawodow.Count, "ListaZawodow powinna byc pusta");

Assert.AreEqual(0, model.ListaBiegaczy.Count, "ListaBiegaczy powinna byc pusta");

}

1. Uruchomić testy jednostkowe:  
   Test > Run > All tests
2. Wyświetlić okno wyników testów i sprawdzić, czy testy są zielone:  
   Test > Windows > Test Explorer



1. Dodać testy dodawania zawodów do modelu, oraz biegacza do wybranych zawodów:

[TestMethod]

public void TestDodajZawody\_ZawodyDodaneBiegaczeBezZmian()

{

MaratonyModel model = new MaratonyModel();

Zawody zawody = model.DodajZawody("Kraków", new DateTime(2016, 1, 10), 11.6);

Assert.AreEqual(1, model.ListaZawodow.Count, "ListaZawodow nie powinna byc pusta");

Assert.AreEqual(0, zawody.Biegacze.Count, "ListaBiegaczy powinna byc pusta");

Assert.AreSame(zawody, model.ListaZawodow[0]);

}

[TestMethod]

public void TestDodajBiegacza\_ZawodyZnalezioneBiegaczDodany()

{

MaratonyModel model = new MaratonyModel();

Zawody zawody1 = model.DodajZawody("Kraków", new DateTime(2016, 1, 10), 11.6);

Zawody zawody = model.DodajZawody("Warszawa", new DateTime(2016, 1, 23), 6);

Biegacz biegacz = model.DodajBiegacza(zawody.ID, "Młody", "Bóg");

Assert.AreEqual(0, zawody1.Biegacze.Count);

Assert.AreEqual(1, zawody.Biegacze.Count);

Assert.AreSame(biegacz, zawody.Biegacze[0]);

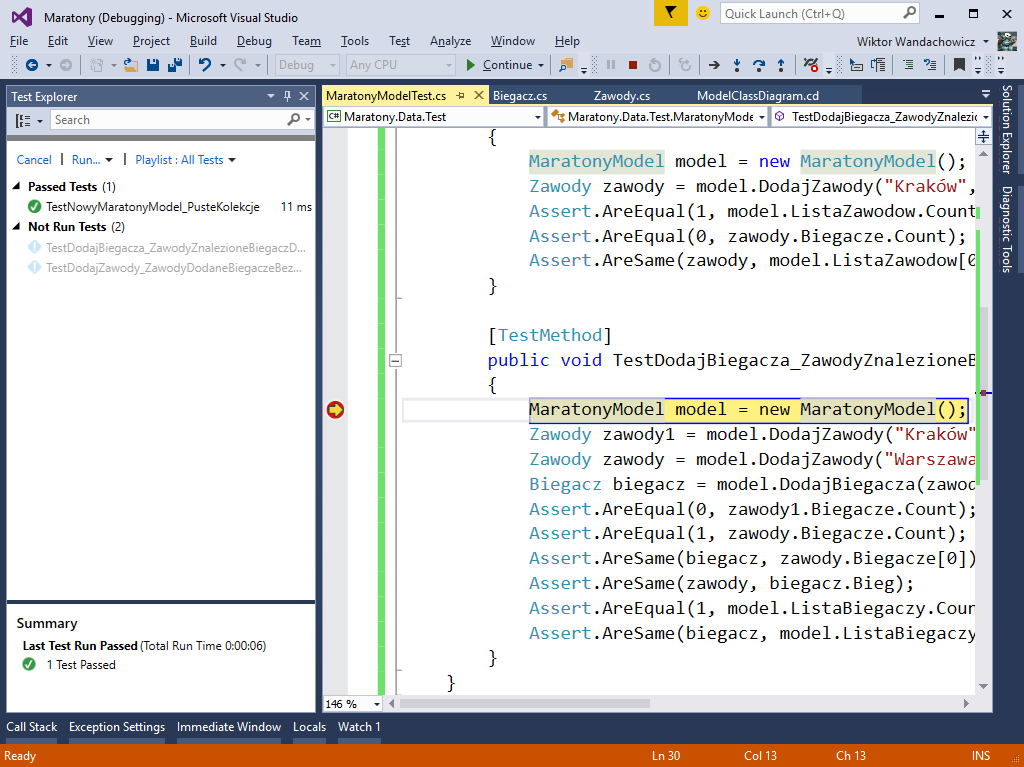
Assert.AreSame(zawody, biegacz.Bieg);

Assert.AreEqual(1, model.ListaBiegaczy.Count);

Assert.AreSame(biegacz, model.ListaBiegaczy[0]);

}

1. W oknie wyników testów uruchomić ponownie wszystkie testy i sprawdzić czy są zielone:  
   Test Explorer > Run All
2. Testy można także debugować. W tym celu trzeba ustawić pułapkę bądź pułapki, a następnie uruchomić testy przez:  
   Test > Debug > All tests



#### 

#### 

#### Część 3 - Budowa podstawowego interfejsu użytkownika

1. Dodać do rozwiązania projekt - aplikację WPF:  
   File > Add > New Project…  
   Visual C# > **WPF Application**  
   **Name: Maratony.FullClient**
2. Zmienić domyślną przestrzeń nazw dla nowych elementów projektu:  
   Solution Explorer > Maratony.FullClient > [PPM] > Properties  
   Application > Default namespace: **Maratony.UI**  
   Zapisać zmieniony plik i zamknąć go.
3. Dodać referencję do projektu zawierającego model danych:  
   Solution Explorer > Maratony.FullClient > References > [PPM] > Add Reference…  
   Solution > ⌧ **Maratony.Data** -- zaznaczyć opcję i zatwierdzić [OK]
4. Ustawić projekt jako startowy, aby móc go wygodnie uruchamiać:  
   Solution Explorer > Maratony.FullClient > [PPM] > Set as StartUp Project
5. Zmienić tytuł okna aplikacji w pliku MainWindow.xaml:

**Title="MainWindow"** Height="350" Width="525">

na:

Title="**Maratony**" Height="350" Width="525">

1. Dodać kontrolkę ComboBox na górze ekranu z marginesem:

<Grid>

**<ComboBox Margin="5" />**

</Grid>

1. Dodać definicje wierszy i kolumn głównej siatki okna:

<Grid>

**<Grid.RowDefinitions>**

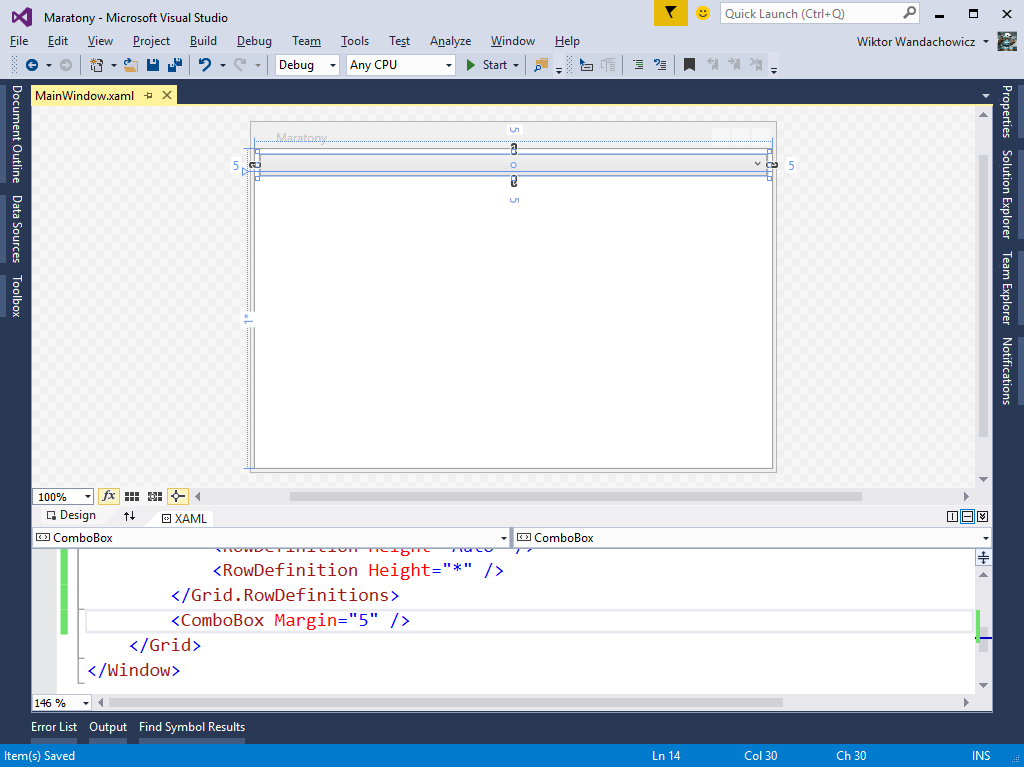
**<RowDefinition Height="Auto" />**

**<RowDefinition Height="\*" />**

**</Grid.RowDefinitions>**

<ComboBox Margin="5" />

</Grid>



Naszym celem jest teraz wykorzystanie wzorca Model-View-ViewModel tak, aby widok odnosił się do danych poprzez wiązania, oraz aby warstwa widoku nie zawierała kodu logiki, ani nie odwoływała się bezpośrednio do modelu.  
Operacje takie będą dodawane w obiekcie pośrednim ViewModel, pomiędzy modelem danych a warstwą prezentacji.

1. Dodać nowy folder pozwalający uporządkować kod:  
   Solution Explorer > Maratony.FullClient > [PPM] > Add > New Folder  
   Zmienić jego nazwę (skrót: [F2]) na: **ViewModel**
2. Dodać nową klasę w folderze ViewModel:  
   Solution Explorer > Maratony.FullClient > ViewModel > [PPM] > Add > Class…  
   **Name: MaintenanceFormViewModel**
3. Dodać specyfikator dostępu public do klasy, pole reprezentujące model danych obiektu ViewModel oraz domyślny konstruktor klasy:

**public** class MaintenanceFormViewModel

{

**private MaratonyModel model = new MaratonyModel();**

**public MaintenanceFormViewModel()**

**{**

**}**

}

1. Rozwiązać błąd nieznanego symbolu MaratonyModel:  
   postawić kursor na podkreślonym na czerwono symbolu i wcisnąć skrót [Ctrl]+[.] (kropka)  
   lub:  
   najechać myszą na podkreślony na czerwono symbol i kliknąć na ikonie żółtej żarówki,  
   wybrać propozycję: using Maratony.Data;
2. Zgodnie z wzorcem MVVM klasa ViewModel udostępnia właściwości, do których może odwoływać się warstwa widoku.  
   Dodać publiczną właściwość reprezentującą zarejestrowane zawody wraz z typowym kodem jej obsługi (get / set):

public class MaintenanceFormViewModel

{

// ...

**private IEnumerable<Zawody> zawody;**

**public IEnumerable<Zawody> Zawody**

**{**

**get**

**{**

**return this.zawody;**

**}**

**set**

**{**

**this.zawody = value;**

**}**

**}**

}

1. Mimo, iż powyższa właściwość mogłaby być zaimplementowana automatycznie przez podanie { get; set; } - nie byłby to dobry sposób.  
   Po każdej zmianie właściwości (klauzula set) ViewModel powinien informować interfejs użytkownika o takiej zmianie. Kiedy kontrolki mają wskazane wiązania danych, muszą być one powiadamiane gdy ViewModel odświeża zawartość właściwości.  
   W tym celu trzeba dodać obsługę interfejsu powiadamiania o zmianach:

public class MaintenanceFormViewModel **: INotifyPropertyChanged**

1. Rozwiązać błąd nieznanego symbolu INotifyPropertyChanged:  
   wybierając propozycję: using System.ComponentModel;
2. Zaimplementować metody interfejsu INotifyPropertyChanged:  
   postawić kursor na podkreślonym na czerwono symbolu i wcisnąć skrót [Ctrl]+[.] (kropka)  
   lub:  
   najechać myszą na podkreślony na czerwono symbol i kliknąć na ikonie żółtej żarówki,  
   wybrać pozycję: **Implement interface**
3. Dodać metodę OnPropertyChanged() powiadamiającą o zmianach i wywoływać ją w klauzuli set właściwości Zawody:

public IEnumerable<Zawody> Zawody

{

get

{

return this.zawody;

}

set

{

this.zawody = value;

**this.OnPropertyChanged("Zawody");**

}

}

**public event PropertyChangedEventHandler PropertyChanged;**

**private void OnPropertyChanged(string propertyName)**

**{**

**/\***

**var handler = this.PropertyChanged;**

**if (handler != null)**

**{**

**handler(this, new PropertyChangedEventArgs(propertyName));**

**}**

**\*/**

**// C# 6.0:**

**this.PropertyChanged?.Invoke(this,**

**new PropertyChangedEventArgs(propertyName));**

**}**

}

1. Trzeba zauważyć, że w klauzuli set właściwości Zawody nazwa właściwości podana jest jawnie jako napis. Jest to kłopot, kiedy zajdzie potrzeba zmiany nazwy właściwości - nazwa będąca napisem nie ulegnie zmianie.

public IEnumerable<Zawody> **Zawody**

{

...

set

{

this.zawody = value;

this.OnPropertyChanged(nameof(**Zawody**));

}

}

1. Można wykorzystać dodatkową usługę kompilatora, aby napis odpowiadający nazwie właściwości był wstawiany automatycznie przez kompilator, poprzez podanie domyślnej wartości pustej oraz użycie atrybutu CallerMemberName dla parametru metody:

public IEnumerable<Zawody> Zawody

{

...

set

{

this.zawody = value;

**this.OnPropertyChanged();**

}

}

private void OnPropertyChanged(

**[System.Runtime.CompilerServices.CallerMemberName]**

**string propertyName = ""**)

1. Pozostało jeszcze dodać kod odświeżający właściwość Zawody w odpowiednich momentach, tutaj - wywoływany w konstruktorze ViewModel:

public class MaintenanceFormViewModel

{

public MaintenanceFormViewModel()

{

Task.Run(**() => Init()**);

}

public MaintenanceFormViewModel(bool init)

{

// Konstruktor uzywany w testach jednostkowych

if (init)

**Init();**

}

internal void Init()

{

**this.PrzykladoweDane();**

**this.OdswiezZawody();**

}

**private void OdswiezZawody()**

**{**

**// this.Zawody -> wywoluje Zawody.set{...} oraz OnPropertyChanged**

**this.Zawody = model.ListaZawodow;**

**}**

1. Potrzebna będzie jeszcze pomocnicza metoda w ViewModel tworząca przykładowe dane:

public class MaintenanceFormViewModel

{

// ...

**internal void PrzykladoweDane()**

**{**

**model.DodajZawody("Kraków", new DateTime(2016, 1, 10), 11.6);**

**model.DodajZawody("Warszawa", new DateTime(2016, 1, 23), 6);**

**model.DodajBiegacza(model.ListaZawodow[0].ID, "Młody", "Bóg");**

**model.DodajBiegacza(model.ListaZawodow[1].ID, "Jan", "Kowalski");**

**model.DodajBiegacza(model.ListaZawodow[1].ID, "Adam", "Nowak");**

**}**

}

1. Aby móc nareszcie wyświetlić wszystkie zawody w kontrolce ComboBox trzeba utworzyć obiekt ViewModel, oraz ustawić utworzony ViewModel jako DataContext dla okna - w pliku code-behind MainWindow.xaml.cs.  
   Tworzenie obiektu będzie odbywać się po załadowaniu okna oraz jego zasobów, w obsłudze zdarzenia Loaded - napisanej jako anonimowa metoda lambda:

public partial class MainWindow : Window

{

**private MaintenanceFormViewModel viewModel =**

**new MaintenanceFormViewModel();**

public MainWindow()

{

InitializeComponent();

**this.Loaded += (s, e) => { this.DataContext = this.viewModel; };**

}

1. Teraz trzeba dodać wiązanie danych w pliku MainWindow.xaml - wiązanie to odwołuje się poprzez DataContext do obiektu ViewModel, oraz przez Path do jego właściwości Zawody:

<ComboBox Margin="5"

**ItemsSource="{Binding Path=Zawody}"** />

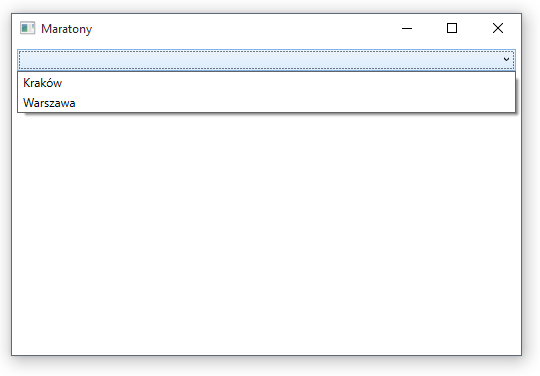
1. Po uruchomieniu widać w kontrolce ComboBox dwie pozycje “Maratony.Data.Zawody”. Dzieje się tak, ponieważ została wywołana domyślna metoda ToString() dla obiektów Zawody. Można jednak łatwo wskazać, aby kontrolka odwoływała się do wybranego pola każdego z obiektów, które ma pokazać:

<ComboBox Margin="5"

ItemsSource="{Binding Path=Zawody}"

**DisplayMemberPath="Miejsce"** />

1. Teraz po uruchomieniu programu w kontrolce ComboBox wyświetlają się nazwy miejscowości, gdzie odbywają się zawody.



Część 4 - Rozbudowa wiązań danych w interfejsie użytkownika

1. Dodać kontrolkę DataGrid, która będzie wyświetlać wszystkich biegaczy dla wybranych przez użytkownika zawodów - w pliku MainWindow.xaml:

<ComboBox Margin="5"

ItemsSource="{Binding Path=Zawody}"

DisplayMemberPath="Miejsce" />

**<DataGrid Grid.Row="1"**

**Margin="5"**

**AutoGenerateColumns="True"**

**CanUserAddRows="False" />**

1. W tym celu trzeba ViewModel wyposażyć w wiedzę, jakie zawody są aktualnie wybrane. Można to zrobić przez nową właściwość w ViewModel, oraz dwukierunkowe wiązanie danych dla kontrolki ComboBox:

public class MaintenanceFormViewModel : INotifyPropertyChanged

{

// ...

**public Zawody WybraneZawody { get; set; } // do zmiany (!!!)**

1. Kontrolkę ComboBoxnależy powiązać z nową właściwością, w trybie TwoWay:

<ComboBox Margin="5"

ItemsSource="{Binding Path=Zawody}"

DisplayMemberPath="Miejsce"

**SelectedItem="{Binding Path=WybraneZawody, Mode=TwoWay}"** />

1. Niestety, nie można użyć tu automatycznych właściwości - zmiana wybranego zawodu musi wysyłać powiadomienie do interfejsu użytkownika:

public class MaintenanceFormViewModel : INotifyPropertyChanged

{

// ...

**private Zawody wybraneZawody;**

**public Zawody WybraneZawody**

**{**

**get**

**{**

**return this.wybraneZawody;**

**}**

**set**

**{**

**this.wybraneZawody = value;**

**this.OnPropertyChanged();**

**this.OdswiezBiegaczy(); // dodane ponizej**

**}**

**}**

1. Źródło danych dla DataGrid także będzie udostępniane jako właściwość ViewModel, a więc podobnie trzeba je udostępniać i odświeżać we właściwych momentach:

**private IEnumerable<Biegacz> biegacze;**

**public IEnumerable<Biegacz> Biegacze**

**{**

**get**

**{**

**return this.biegacze;**

**}**

**set**

**{**

**this.biegacze = value;**

**OnPropertyChanged();**

**}**

**}**

**private void OdswiezBiegaczy()**

**{**

**this.Biegacze = null;**

**this.Biegacze = this.WybraneZawody?.Biegacze;**

**}**

1. Dodać wiązanie dla DataGrid do właściwości Biegacze:

<DataGrid Grid.Row="1"

Margin="5"

AutoGenerateColumns="True"

CanUserAddRows="False"

**ItemsSource="{Binding Path=Biegacze}"** />

1. Uruchomić aplikację i sprawdzić jej działanie.

#### Część 5 - Testy jednostkowe dla ViewModel

1. Dodać do rozwiązania projekt - testy jednostkowe:  
   File > Add > New Project…  
   Visual C# > Test > **Unit Test Project**  
   **Name: Maratony.UI.Test**
2. Dodać referencje do projektu zawierającego model danych oraz ViewModel:

Solution Explorer > Maratony.UI.Test > References > [PPM] > Add Reference…  
Solution > ⌧ **Maratony.Data** -- zaznaczyć opcję  
 ⌧ **Maratony.FullClient** -- zaznaczyć opcję  
zatwierdzić [OK]

1. Zmienić nazwę automatycznie dodanej klasy UnitTest1 na ViewModelTest:

Solution Explorer > Maratony.UI.Test > UnitTest1 > [PPM] > Rename > **ViewModelTest**  
potwierdzić zmianę nazwy klasy równocześnie przy zmianie nazwy pliku.

1. Zamienić pustą metodę TestMethod1 na TestNowyViewModel\_OdswiezZawody:

[TestMethod]

public void TestNowyViewModel\_OdswiezZawody()

{

var vm = new MaintenanceFormViewModel(**true**); // konstruktor dla UT

Assert.IsTrue(vm.Zawody.Count() > 0);

}

1. Rozwiązać błąd nieznanego symbolu MaintenanceFormViewModel:

wybrać propozycję: using Maratony.UI.ViewModel;  
Rozwiązać błąd nieznanej metody rozszerzającej Count():  
wybrać propozycję: using System.Linq;

1. Uruchomić testy jednostkowe:  
   Test > Run > All tests
2. Wyświetlić okno wyników testów i sprawdzić, czy testy są zielone:  
   Test > Windows > Test Explorer
3. Dodać metodę TestNowyViewModel\_OdswiezBiegaczy:

[TestMethod]

public void TestNowyViewModel\_OdswiezBiegaczy()

{

var vm = new MaintenanceFormViewModel(**true**); // konstruktor dla UT

var wywolaneOdswiezBiegaczy = false;

vm.PropertyChanged += (s, e) =>

{

if (e.PropertyName == "Biegacze")

{

wywolaneOdswiezBiegaczy = true;

}

};

vm.WybraneZawody = vm.Zawody.Skip(1).First();

Assert.IsTrue(wywolaneOdswiezBiegaczy);

Assert.IsTrue(vm.Biegacze.Count() > 0);

}

1. W oknie wyników testów uruchomić ponownie wszystkie testy i sprawdzić czy są zielone:  
   Test Explorer > Run All

#### 

#### 

#### Część 6 - Dodanie polecenia (Command) do ViewModel, użycie RelayCommand

1. Dodać do projektu nową klasę: RelayCommand  
   Solution Explorer > Maratony.FullClient > [PPM] > Add > Class…  
   Name: **RelayCommand.cs**
2. Wstawić poniższą implementację - z jednego ze źródeł:

* <https://msdn.microsoft.com/en-us/magazine/dd419663.aspx#code-snippet-1>
* <https://mvvmfoundation.codeplex.com/SourceControl/latest#MvvmFoundation/MvvmFoundation.Wpf/RelayCommand.cs>

public class RelayCommand : ICommand

{

#region Fields

readonly Action<object> \_execute;

readonly Predicate<object> \_canExecute;

#endregion // Fields

#region Constructors

public RelayCommand(Action<object> execute) : this(execute, null) { }

public RelayCommand(Action<object> execute,

Predicate<object> canExecute)

{

if (execute == null)

throw new ArgumentNullException(nameof(execute));

\_execute = execute;

\_canExecute = canExecute;

}

#endregion // Constructors

#region ICommand Members

[DebuggerStepThrough]

public bool CanExecute(object parameter)

{

return \_canExecute == null ? true : \_canExecute(parameter);

}

public event EventHandler CanExecuteChanged

{

add {

if (\_canExecute != null)

CommandManager.RequerySuggested += value;

}

remove {

if (\_canExecute != null)

CommandManager.RequerySuggested -= value;

}

}

public void Execute(object parameter) { \_execute(parameter); }

#endregion // ICommand Members

}

1. Dodać w oknie głównym definicję trzeciego wiersza siatki oraz dwóch przycisków zagnieżdżonych w dodatkowej siatce - w pliku MainWindow.xaml:

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

**<RowDefinition Height="Auto" />**

</Grid.RowDefinitions>

<ComboBox Margin="5"

... />

<DataGrid Grid.Row="1"

... />

<Grid Grid.Row="2">

**<Grid.ColumnDefinitions>**

**<ColumnDefinition Width="50\*"/>**

**<ColumnDefinition Width="50\*"/>**

**</Grid.ColumnDefinitions>**

**<Button Margin="5"**

**Grid.Column="0"**

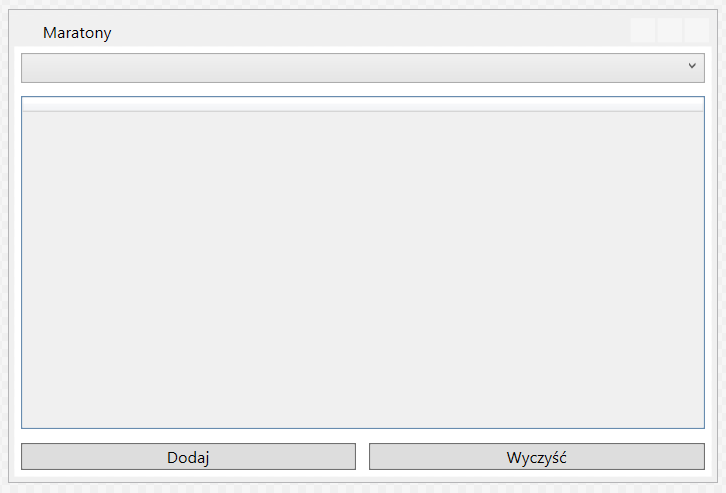
**Content="Dodaj" />**

**<Button Margin="5"**

**Grid.Column="1"**

**Content="Wyczyść" />**

</Grid>



1. Dodać w **MaintenanceFormViewModel**:

* metodę generującą przykładowego biegacza, oraz
* metodę sprawdzającą czy można dodać biegacza - warunkiem jest, aby były wybrane jakiekolwiek zawody:

public class MaintenanceFormViewModel : INotifyPropertyChanged

{

// ...

**public void DodajBiegacza()**

**{**

**if (WybraneZawody != null)**

**{**

**model.DodajBiegacza(this.WybraneZawody.ID, "Nowy", "Biegacz");**

**this.OdswiezBiegaczy();**

**}**

**}**

**public bool CzyMoznaDodacBiegacza()**

**{**

**return (this.WybraneZawody != null);**

**}**

1. Nie można niestety powiązać bezpośrednio przycisku z metodą ViewModel. Trzeba w tym celu dodać pomocniczą właściwość:

public class MaintenanceFormViewModel : INotifyPropertyChanged

{

// ...

**public ICommand SaveCommand { get; private set; }**

1. Rozwiązać błąd nieznanego symbolu ICommand:  
   wybierając propozycję: using System.Windows.Input;
2. Dodać w funkcji inicjalizującej kod tworzący polecenie SaveCommand jako obiekt klasy RelayCommand, ze wskazaniem dwóch metod:

* DodajBiegacza - wykonującą polecenie, oraz
* CzyMoznaDodacBiegacza - sprawdzającą możliwość wykonania polecenia

internal void Init()

{

this.PrzykladoweDane();

this.OdswiezZawody();

**this.SaveCommand = new RelayCommand(**

**action => this.DodajBiegacza(),**

**enable => this.CzyMoznaDodacBiegacza());**

}

1. Dodać wiązanie pomiędzy przyciskiem “Dodaj” a poleceniem SaveCommand - w pliku MainWindow.xaml:

<Button Margin="5"

Grid.Column="0"

Content="Dodaj"

Command="**{Binding SaveCommand}**" />

1. Przetestować działanie aplikacji.
2. Dodać w ViewModel metodę WyczyscWybraneZawody:

**private void WyczyscWybraneZawody()**

**{**

**this.WybraneZawody = null;**

**}**

1. Na podobnej zasadzie co w punkcie 5. dodać właściwość ClearCommand.
2. Podobnie jak w punkcie 7. dodać kod tworzący polecenie ClearCommand jako obiekt klasy RelayCommand, ze wskazaniem dwóch metod:

* WyczyscWybraneZawody - wykonującą polecenie, oraz
* CzyMoznaDodacBiegacza - sprawdzającą możliwość wykonania polecenia

1. Powiązać w MainWindow.xaml przycisk “Wyczyść” z poleceniem ClearCommand.
2. Przetestować działanie aplikacji.

### Polecenie wielokrotnego użytku - DelegateCommand / RelayCommand

**WPF Tutorial:  
How to implement a reusable ICommand**

<https://wpftutorial.net/DelegateCommand.html>

### Użycie RelayCommand zamiast ICommand

W tym celu należy dodać nową klasę w projekcie: **RelayCommand**  
Następnie wkleić poniższy kod:

using System;

using System.Diagnostics;

using System.Windows.Input;

namespace Maratony.UI

{

public class RelayCommand : ICommand

{

#region Fields

readonly Action<object> \_execute;

readonly Predicate<object> \_canExecute;

#endregion // Fields

#region Constructors

public RelayCommand(Action<object> execute)

: this(execute, null)

{ }

public RelayCommand(Action<object> execute,

Predicate<object> canExecute)

{

if (execute == null)

throw new ArgumentNullException(nameof(execute));

\_execute = execute;

\_canExecute = canExecute;

}

#endregion // Constructors

#region ICommand Members

[DebuggerStepThrough]

public bool CanExecute(object parameter)

{

return \_canExecute == null ? true : \_canExecute(parameter);

}

public event EventHandler CanExecuteChanged

{

add { CommandManager.RequerySuggested += value; }

remove { CommandManager.RequerySuggested -= value; }

}

public void Execute(object parameter) { \_execute(parameter); }

#endregion // ICommand Members

}

}

Kod **MaintenanceFormViewModel.cs**

using Maratony.Data;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Input;

namespace Maratony.UI.ViewModel

{

public class MaintenanceFormViewModel : INotifyPropertyChanged

{

private MaratonyModel model = new MaratonyModel();

public MaintenanceFormViewModel()

{

Task.Run(() => Init());

}

public MaintenanceFormViewModel(bool init)

{

if (init) // Konstruktor używany w testach jednostkowych

Init();

}

internal void Init()

{

this.PrzykladoweDane();

this.OdswiezZawody();

this.SaveCommand = new RelayCommand(

action => this.DodajBiegacza(),

enable => this.CzyMoznaDodacBiegacza());

this.ClearCommand = new RelayCommand(

action => this.WyczyscWybraneZawody,

enable => this.CzyMoznaDodacBiegacza);

}

internal void PrzykladoweDane()

{

model.DodajZawody("Kraków", new DateTime(2016, 1, 10), 11.6);

model.DodajZawody("Warszawa", new DateTime(2016, 1, 23), 6);

model.DodajBiegacza(model.ListaZawodow[0].ID, "Młody", "Bóg");

model.DodajBiegacza(model.ListaZawodow[1].ID, "Jan", "Kowalski");

model.DodajBiegacza(model.ListaZawodow[1].ID, "Adam", "Nowak");

}

private void OdswiezZawody()

{

// this.Zawody -> wywoluje Zawody.set{...} oraz OnPropertyChanged

this.Zawody = model.ListaZawodow;

}

private void OdswiezBiegaczy()

{

this.Biegacze = null;

this.Biegacze = this.WybraneZawody.Biegacze;

}

#region Properties: Zawody, WybraneZawody

private IEnumerable<Zawody> zawody;

public IEnumerable<Zawody> Zawody

{

get

{

return this.zawody;

}

set

{

this.zawody = value;

this.OnPropertyChanged(nameof(Zawody));

}

}

private IEnumerable<Biegacz> biegacze;

public IEnumerable<Biegacz> Biegacze

{

get

{

return this.biegacze;

}

set

{

this.biegacze = value;

OnPropertyChanged();

}

}

private Zawody wybraneZawody;

public Zawody WybraneZawody

{

get

{

return this.wybraneZawody;

}

set

{

this.wybraneZawody = value;

this.OnPropertyChanged();

this.OdswiezBiegaczy();

}

}

#endregion

#region Commands: SaveCommand, ClearCommand

public ICommand SaveCommand { get; private set; }

public void DodajBiegacza()

{

if (WybraneZawody != null)

{

model.DodajBiegacza(this.WybraneZawody.ID, "Nowy", "Biegacz");

this.OdswiezBiegaczy();

}

}

public bool CzyMoznaDodacBiegacza()

{

return (this.WybraneZawody != null);

}

public ICommand ClearCommand { get; private set; }

private void WyczyscWybraneZawody()

{

this.WybraneZawody = null;

}

#endregion

/\*

// Polecenie jako tzw. Singleton - bez tworzenia w konstruktorze

private RelayCommand saveCommand;

public ICommand SaveCommand

{

get

{

// Opoznione tworzenie polecenia przy pierwszym uzyciu

if (saveCommand == null)

{

saveCommand = new RelayCommand(

param => this.DodajBiegacza(),

param => this.CzyMoznaDodacBiegacza());

}

return saveCommand;

}

}

\*/

public event PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged(

[System.Runtime.CompilerServices.CallerMemberName]

string propertyName = "")

{

/\*

var handler = this.PropertyChanged;

if (handler != null)

{

handler(this, new PropertyChangedEventArgs(propertyName));

}

\*/

// C# 6.0

this.PropertyChanged?.Invoke(this,

new PropertyChangedEventArgs(propertyName));

}

}

}