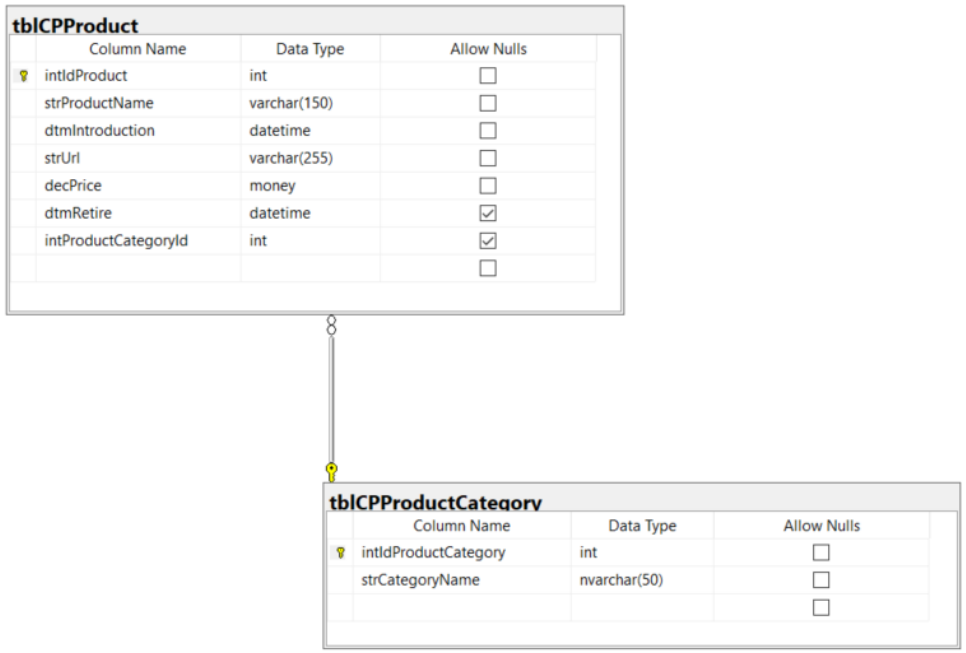

Figure 1: Database Diagram

Notes

.....

.....

.....

.....

.....

.....

.....

COPY PASTE)

Creating a WPF MVVM Project

Basic info

At the end, we will use the SQL Server data from database cpADONET.

The script to create this database can be found on GitHub and on Moodle.

https://github.com/DontDestroy/C-Sharp.NET-2019

- Look for Database Scripts.
- Look for cpADONET.sql.

This contains:

- 2 tables.
  - tblCPProduct.
  - tblCPProductCategory.
- 1 relation between the 2 tables.
- And a bit of data in both tables.
- The Database diagram can be created, but is also shown in picture beside. (*SEE FIGURE 1*)

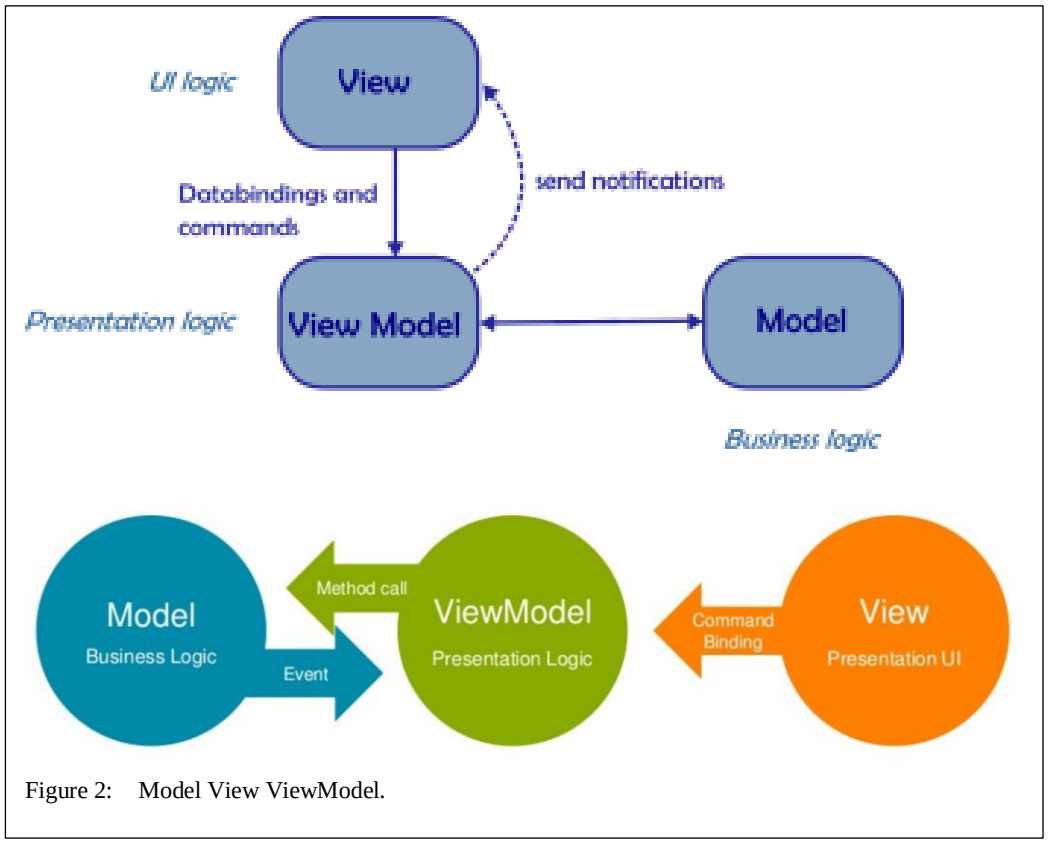

Figure 2: Model View ViewModel.

Notes

.....

.....

.....

.....

.....

.....

.....

COPY PASTE)

What is Model-View-ViewModel (MVVM)?

MVVM is a design pattern. A way of working for modern User Interface applications.

It is based on MVC (Model View Controller) and MVP (Model View Presenter).

The main goal of this design pattern is to separate the state and behaviour in the software application.

Model (Business Logic)

- Manage the application data.
- Manage the state of information.
- Can be a Data Access Layer as ADO.NET or Entity Framework).

View (User Interface Logic)

- How is it shown on the screen?
- User Interface elements.
  - Windows, Forms, Controls, Fields, Buttons, …

ViewModel (Presentation Logic)

- The interaction between the View and the Model.
- Data binding and data conversion are triggered in the View.
- Commands are used to bind to the Views.
- The Model handles events.
- The ViewModel calls methods.

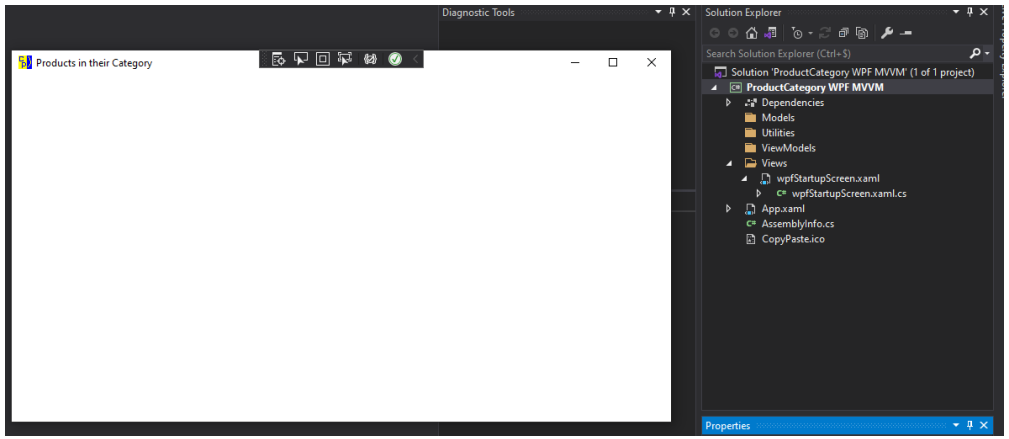

Figure 3: The structure and the application.

Notes

.....

.....

.....

.....

.....

.....

.....

COPY PASTE)

Exercise 1: Create a WPF MVVM project from scratch

The end goal is a working application, with a connection to the database.

We build up slowly.

The goal is to explain the process to getting there.

Task 1: Making the structure

1. Create a new WPF Application (.Net Framework or .Net Core). Give your solution and your project a good name. We will show product categories and the products inside it.
2. Create folder "Models" in the project.
3. Create folder "Views" in the project.
4. Create folder "ViewModels" in the project.
5. Create folder "Utilities" in the project.
6. Rename you start-up window "MainWindow", into a more meaningful name.
7. Move that file towards the folder "Views".
8. Change the StartupUri in the file "App.xaml" to the correct path and window. "Views/YourStartupScreen.xaml".
9. Rename your classes, constructors so that automatic code is generated in a correct way.
10. Give your screen a good title.
11. You will have something like beside (*SEE FIGURE 3*).
12. Run the application.

By running the application, you recreate automatic generated code and you will see if you haven't forgot something to rename.

Check App.xaml and YourStartupScreen.xaml.

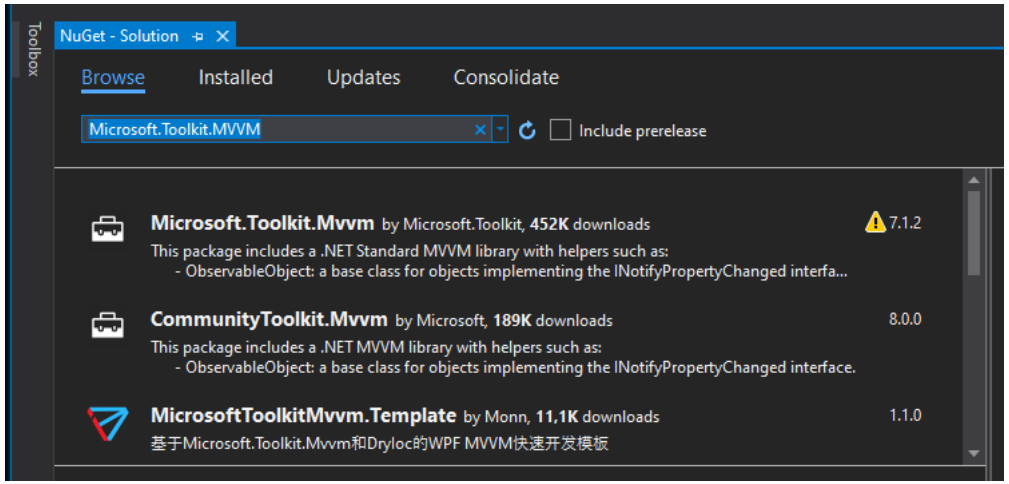

Figure 4: Microsoft.Toolkit.MVVM.

Task 2: Create the class ObservableObject

For this exercise we create the class by ourselves.

There are existing NuGet packages where all this is already done.

- *Tools > NuGet Package Manager > Manage NuGet Packages for Solution.*
- *Search for "Microsoft.Toolkit.MVVM"*
- *See Figure 4.*

In real life project, I use one of the toolkits, or my own toolkit, but for training purposes it is good that you understand what must be done for getting this running.

The end result of this exercise is a working project that can be a template for all your future work.

1. Create a class "ObservableObject" in the folder "Utilities".
2. Copy this code to it.

What does this code do?

- The main purpose of this code is to make sure the application can detect if something is changed on a property.
- When this is the case, the change must be visible on the screen.
- INotifyPropertyChanged is part of System.ComponentModel.
- CallerMemberName is part of System.Runtime.CompilerServices.

Begin code

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace YourNamespaceName
{

  public class ObservableObject : INotifyPropertyChanged
```

Notes

```
  {
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string
propertyName = "")
    {
      PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }

    protected virtual bool OnPropertyChanged<T>(ref T backingField, T
value, [CallerMemberName] string propertyName = "")
    {
      if (EqualityComparer<T>.Default.Equals(backingField, value))
      {
        return false;
      }

      backingField = value;
      OnPropertyChanged(propertyName);
      return true;
    }

  }

}
```

End code

Try to understand what is happening here.

Can you read the code? Do you understand the complete syntax? Do you recognize the different items?

- *Event.*
- *Overloading.*
- *Inheritance.*

Notes

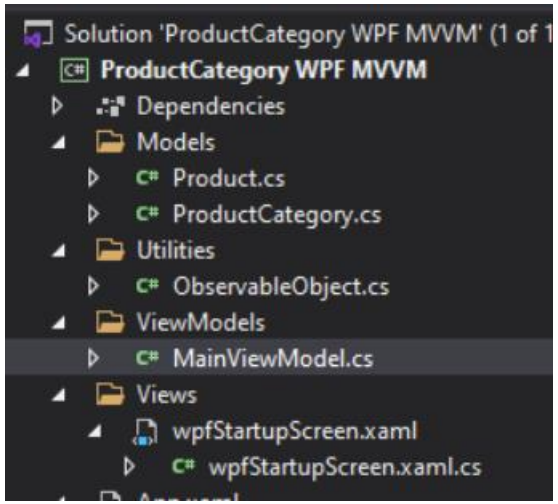

Figure 5: Views and ViewModels.

Task 3: Create 2 classes in the folder Models

1. Create a class "ProductCategory" in the folder "Models" that inherits from the class "ObservableObject".

Depending on how your namespaces are called, you will have to add a reference to this class.

2. Create a class "Product" in the folder "Models" that inherits from the class "ObservableObject".

We will make a 2-way binding between these classes and the user interface.

A change in the data, after reading it, will change the form content, and a change in the form will change the data.

3. Create a class "MainViewModel" in the folder "ViewModels" that inherits from the class "ObservableObject".
4. You will have something like beside (*SEE FIGURE 5*).
5. Check if the namespaces are correct of the added code, meaning, code in Models must have the correct namespace, code in ViewModels must have the correct namespace.

Notes

COPY PASTE)

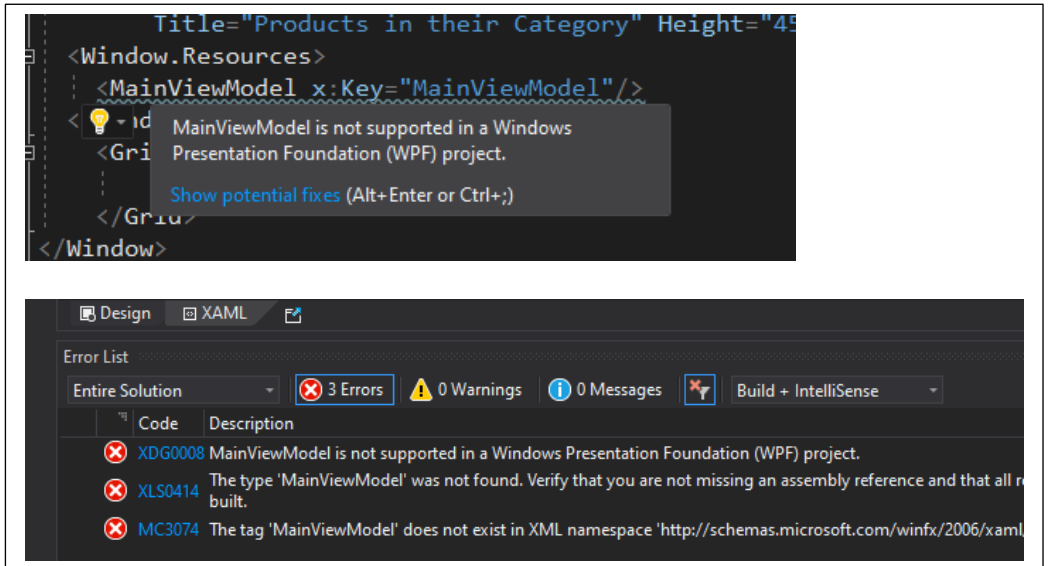

Figure 6: xaml file of the start-up screen.

Notes

.....

.....

.....

.....

.....

.....

.....

COPY PASTE)

Task 4: Prepare the start-up screen to use MainViewModel as resource

We want to connect the ViewModel "MainViewModel" to the start-up screen.

1. Create in "MainWindow.xaml", the tags <Window.Resources> </Window.Resources> before the <Grid> tag.
2. Type <MainViewModel x:Key="MainViewModel"/> between the tags your created in step 1. This adds a Window.Resource for MainViewModel.

Doing this will go wrong, because in the xaml file we need to add the namespaces where the MainViewModel can be found.

You can hover over the added tag, and see what is wrong. You can also see the issues in the Error List Window.

See Figure 6.

3. Click on the arrow of the yellow bulb (*SEE FIGURE 6*) and click "Add xmlsn …" to add the namespace on top of the xaml file.
4. The namespace got a prefix "viewmodels".
5. Rebuild your application.
6. Check if your application still runs.

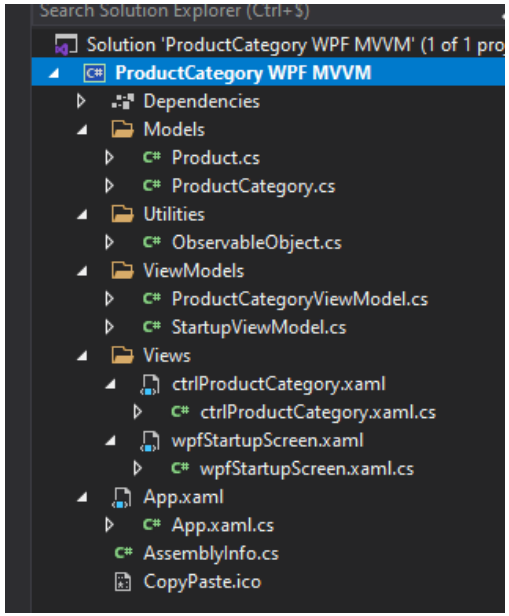

Figure 7: Add a User Control and View Model

Notes

Task 5: Create a User Control to show a Product Category

This control will be added to the start-up screen.

1. In the folder "Views" add a User Control and give that control a good name.
2. Add a control of the type TextBlock to it.
3. Give that control a good name.
4. Add some text to the TextBlock, that later you will see here the product categories.
5. Create a new class under the folder view models that will represent the viewmodel of ProductCategory.
6. Give that class a good name.
7. You will have something like beside (*SEE FIGURE 7*).
8. Add that control to the grid in the start-up screen.

Doing this will go wrong, because in the xaml file we need to add the namespaces where the MainViewModel can be found.

9. Do the same steps for adding the correct namespace as in task 4.
10. Rebuild your application.
11. Check if your application still runs.

End result

The end result can be found in 00061-a ProductCategory WPF MVVM.zip, but not all step are followed to the letter. The example is just for checking, it is not to copy paste the solution.