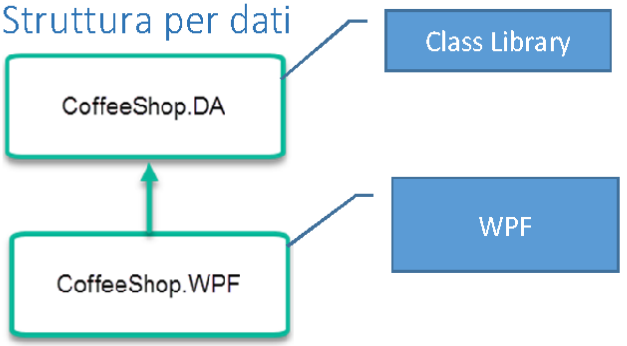
Manuale MVVM CoffeeShop

Struttura per dati

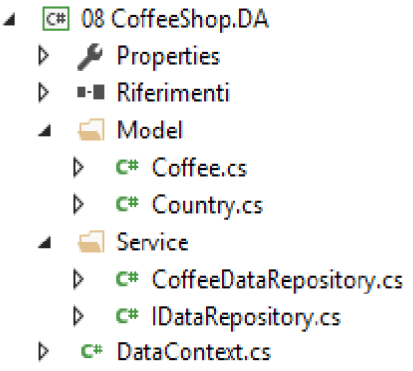


Lavorare con dati (parte 1)

Creare le Classi

Nel progetto CoffeeShop.DA creare una cartella Model, dove vengono inserite le classi dei dati.

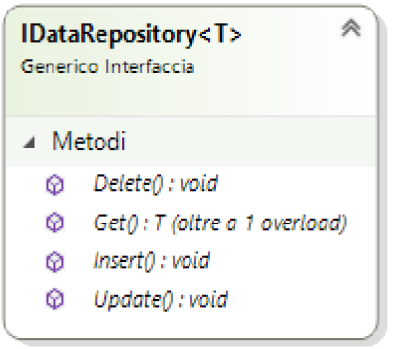
```
public class Coffee
{
    public int CoffeeId { get; set; }
    public string Name { get; set; }
    public int Price { get; set; }
    public string Description { get; set; }
    public Country Origin { get; set; }
    public bool InStock { get; set; }
    public int Stock { get; set; }
    public DateTime FirstAdded { get; set; }
    public int ImageId { get; set; }
}
public enum Country
{
    Brasile, Ecuador, Etiopia, Vietnam, Colombia, Peru,
    Messico, Honduras, India, Indonesia, Guatemala
}
```



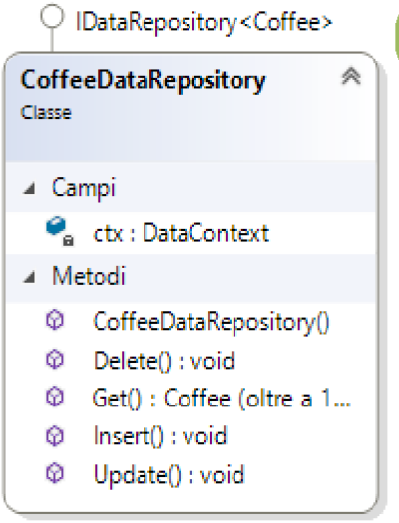
Nella root mettere la classe della banca dati:

```
public class DataContext
{
    public List<Coffee> Coffees { get; private set; }
    public DataContext()
    {
        Coffees = new List<Coffee>()
        {
            new Coffee ()
            {
                CoffeeId = 1,
                Name = "Latte alla moda dello chef Gill",
                Description = "Semplicemente il miglior latte al mondo, con un po' di sciroppo alle nocciole!",
                ImageId = 1,
                Stock=10,
                InStock = true,
                FirstAdded = new DateTime(2014,1,3),
                Origin = Country.Etiopia,
                Price = 12
            },
            new Coffee ()
            {
                CoffeeId = 2,
                Name = "Espresso",
                Description = "Espresso è un caffè forte dove il vapore ad alta pressione ..." +
                              "in una macchina per l'espresso. Un espresso perfetto avrà ..." +
...
    }
```

Per accedere ai dati, si usa un Pattern (DataRepository, più brevemente repository) dove definisco nell'interfaccia i metodi di accesso CRUD (da inserire nella cartella Service). La classe implementa le varie operazioni (qui solo parzialmente).



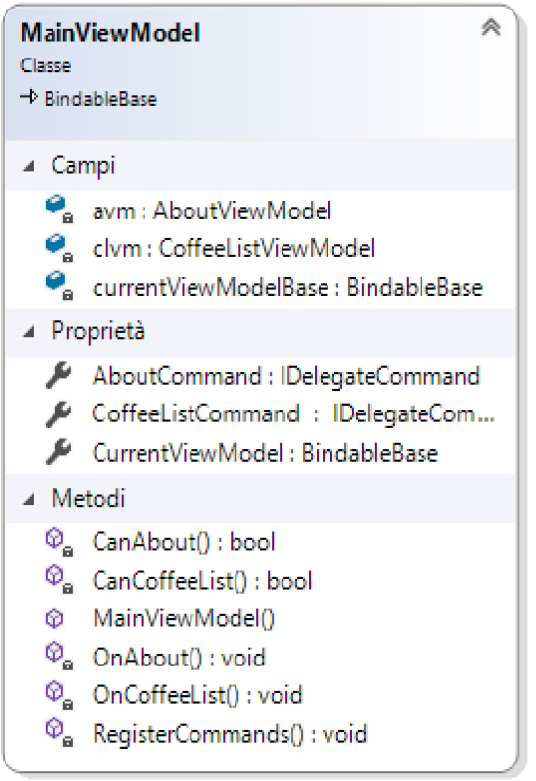

```
public class CoffeeDataRepository : IDataRepository<Coffee>
{
    private static DataContext ctx;
    public CoffeeDataRepository()
    {
        ctx = new DataContext();
    }

    public void Delete(Coffee c)
    {
        throw new NotImplementedException();
    }

    public IEnumerable<Coffee> Get()
    {
        return ctx.Coffees;
    }

    public Coffee Get(int id)
    {
        return ctx.Coffees.Where(c => c.CoffeeId == id).FirstOrDefault();
    }

    public void Insert(Coffee c)
    {
        throw new NotImplementedException();
    }

    public void Update(Coffee c)
    {
        throw new NotImplementedException();
    }
}
```



Preparazione della MainView

Per poter gestire e rappresentare le specifiche View nella MainView, pure questa deve avere un ViewModel che implementi INPC (inclusa in BindableBase).

All'interno del ViewModel sono definiti tutti i ViewModel del programma che sono utilizzati, esponendo una proprietà di tipo BindableBase (il nome scelto è CurrentViewModelBase) che indica alla MainView quale dettaglio deve mostrare.




```
private BindableBase currentViewModel;
public BindableBase CurrentViewModel
{
    get { return currentViewModel; }
    set { SetProperty(ref currentViewModel, value); }
}
```

In questo caso sono inoltre definiti i Command (in RegiterCommands richiamato nel costruttore) per la modifica della rappresentazione con i relativi metodi.

```
public MainViewModel()
{
    avm = new AboutViewModel();
    clvm = new CoffeeListViewModel();
    RegisterCommands();
    // per default mettiamo il coffeelist
    CurrentViewModel = clvm;
}
```

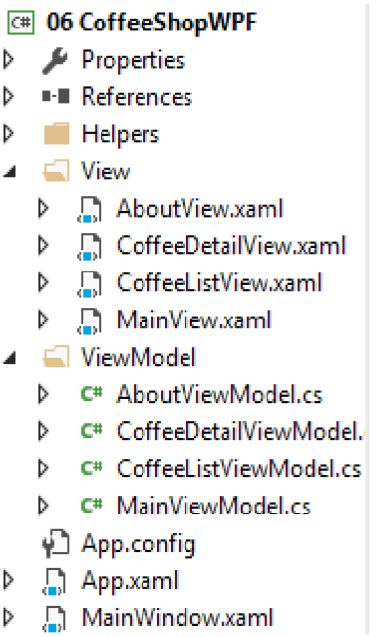
Nella MainView si utilizza un ContentControl che permette di rappresentare la View dinamicamente. Qualora definito nelle risorse la "conversione" da effettuare, ottenuto un ViewModel, mi rappresenta la corrispondente View (una specie di ToString).

```
...
    <UserControl.Resources>
        <DataTemplate DataType="{x:Type viewmodel:CoffeeListViewModel}">
            <local:CoffeeListView/>
        </DataTemplate>

        <DataTemplate DataType="{x:Type viewmodel:AboutViewModel}">
            <local:AboutView/>
        </DataTemplate>
    </UserControl.Resources>
    <Grid>
        <ContentControl Content="{Binding Path=CurrentViewModel}"/>
    </Grid>
```


Nel caso specifico, quando CurrentViewModel del MainViewModel è CoffeeListViewModel, viene mostrata CoffeeLisView, se è AboutViewModel, AboutView.

Rappresentare i dati

Preparare la struttura delle cartelle View + ViewModel e preparare la referenza al progetto CoffeeShop.DA come pure gli xmlns nelle view.



CoffeeList

ViewModel

Visto che voglio rappresentare una serie di dati, definisco una proprietà ObservableCollection dove tutte le caratteristiche per la gestione sono già state implementate. Il "visualizzatore" gestisce normalmente una proprietà (SelectedItem) dove viene memorizzato il dato selezionato. Nel costruttore andiamo a prendere i dati dal repository.

```
public class CoffeesListViewModel
{
    #region ================== membri & proprietà =========
    public ObservableCollection<Coffee> Coffees { get; set; }
    private Coffee selectedCoffee;
    public Coffee SelectedCoffee
    {
        get { return selectedCoffee; }
        set { SetProperty(ref selectedCoffee, value); }
    }
    #endregion

    #region ================== costruttori ================
    public CoffeeListViewModel()
    {
        CoffeeDataRepository repo = new CoffeeDataRepository();
        //Coffees = repo.Get().ToObservableCollection();
        Coffees = new ObservableCollection<Coffee>(repo.Get());
    }
    #endregion
}
```

View

Per mostrare i dati usiamo prima un DataGrid che rappresenta automaticamente i dati

CoffeeId	Name	Price	Descriptio
1	Latte alla moda dello chef Gill	12	Semplicem
2	Espresso	12	Espresso è
3	Cappuccino	12	Questo po
4	Americano	14	Un caffè A
5	Caffè Latte	9	Una parte
6	Caffè au Lait	11	Questa be
7	Caffè marocchino	10	si prepara
8	Caramel Macchiato	11	Questa è u

```
<DataGrid ItemsSource="{Binding Path=Coffees}"/>
```

Per poter usare una rappresentazione personalizzata si deve usate un ListBox e personalizzare la rappresentazione con un DataTemplate che definisco nelle Resources dell'UserControl e sul ListBox indico l'ItemTemplate da utilizzare

```
        <DataTemplate x:Key="CoffeeTemplate">
            <Grid ShowGridLines="True">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="100"/>
                    <ColumnDefinition Width="100"/>
                    <ColumnDefinition Width="300"/>
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition Height="30"/>
                    <RowDefinition Height="30"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>
                <Label Grid.Column="1" Grid.ColumnSpan="2"
                       Content="{Binding Path=Name}"
                       FontWeight="Bold" FontSize="14"/>
                <StackPanel Grid.Row="1" Grid.Column="1" Orientation="Horizontal">
                    <Label Content="Stock: "/>
                    <Label Content="{Binding Path=Stock}"/>
                </StackPanel>
                <Label Grid.Row="1" Grid.Column="2"
                       Content="{Binding Path=Price}" HorizontalAlignment="Right" />
                <Button Grid.Column="2" Grid.Row="2"
                        Content="Dettaglio"/>
            </Grid>
        </DataTemplate>
...
        <ListBox ItemsSource="{Binding Path=Coffees}"
                 ItemTemplate="{StaticResource CoffeeTemplate}"
                 SelectedItem="{Binding Path=SelectedCoffee}"/>
```

_08_CoffeeShop.DA.Model.Coffee
_08_CoffeeShop.DA.Model.Coffee
_08_CoffeeShop.DA.Model.Coffee
_08_CoffeeShop.DA.Model.Coffee
08_CoffeeShop.DA.Model.Coffee

Per poter eseguire un Command nell'esempio indicato è necessario indicare dove si trova il DataContext: deve risalire l'albero fino a quando non trova il suo AncestorType (in questo caso il ListBox).

```
                <Button Grid.Column="2" Grid.Row="2"
                        Content="Dettaglio"
                        Command="{Binding Path=DataContext.CoffeeDetailCommand,
                                          RelativeSource={RelativeSource AncestorType={x:Type ListBox}}}"
                        CommandParameter="{Binding}"/>
```

ViewModelLocator (senza IoC)

ViewModelLocator é dichiarato come un oggetto in App.xaml e soddisfa il pattern Singleton (un unico oggetto in tutta l'applicazione). Il seguente codice lo definisce in App.xaml:

```
            xmlns:viewmodel="clr-namespace:_08_CoffeeShop.ViewModel"
…
    <Application.Resources>
        <viewmodel:ViewModelLocator x:Key="Locator"/>
    </Application.Resources>
```

Esso è la sorgente di tutti i nostri ViewModel, per ogni ViewModel si ha una proprietà nel ViewModelLocator che ci permette di ottenere un ViewModel per una View.

Questo ci permette di eliminare la definizione e creazione dei ViewModel nel MainViewModel.

```
    public class ViewModelLocator
    {
        public MainViewModel Main { get; private set; }
        public AboutViewModel About { get; private set; }
        public CoffeeListViewModel CoffeeList { get; private set; }
        public ViewModelLocator()
        {
            Main = new MainViewModel();
            About = new AboutViewModel();
            CoffeeList = new CoffeeListViewModel();
        }
    }
```

Inoltre permette di semplificare il DataContext della View usando

```
        DataContext="{Binding Source={StaticResource Locator}, Path=<nome>}"
```

Inversion of Control (IoC)

Gli scopi di un IoC sono molteplici:

- Rendere la classe facilmente testabile
- Rendere il codice interface-driven: vengono referenziate le interfacce piuttosto che le classi concrete
- Rendere il codice lievemente legato: qualcuno può modificare l'implementazione di un'interfaccia e le classi che consumano quella interfaccia non devono essere riprogrammate
- Risolvono le dipendenze delle classi in modo automatico

Uso di SimpleIoC da MVVM-Light.

```
public MainViewModel Main { get { return ServiceLocator.Current.GetInstance<MainViewModel>(); } }
public CoffeeListViewModel CoffeesList { get { return ServiceLocator.Current.GetInstance<CoffeeListViewModel>(); } }
public CoffeeDetailViewModel CoffeeDetail { get { return ServiceLocator.Current.GetInstance<CoffeeDetailViewModel>();
} }
public AboutViewModel About { get { return ServiceLocator.Current.GetInstance<AboutViewModel>(); } }
```

```
        /// <summary>
        /// Initializes a new instance of the ViewModelLocator class.
        /// </summary>
        public ViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

            if (ViewModelBase.IsInDesignModeStatic)
            {
                // Create design time view services and models
                SimpleIoc.Default.Register<IDataRepository<Coffee>, DesignDataRepository>();
            }
            else
            {
                // Create run time view services and models
                SimpleIoc.Default.Register< IDataRepository<Coffee>, CoffeeRepository>();
            }

            SimpleIoc.Default.Register<MainViewModel>();
            SimpleIoc.Default.Register<AboutViewModel>();
            SimpleIoc.Default.Register<CoffeeListViewModel>();
            SimpleIoc.Default.Register<CoffeeDetailViewModel>(true);

        }
```