|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Application Development, Part II |

|  |
| --- |
| By Per Laursen  01-08-2018 |

[Introduction 2](#_Toc522011851)

[The MVVM application architecture - Overview 3](#_Toc522011852)

[MVVM – single domain object 3](#_Toc522011853)

[MVVM – collection of domain objects 8](#_Toc522011854)

[A Data view model class 9](#_Toc522011855)

[Creating a collection of data view model objects 11](#_Toc522011856)

[A Page view model class 12](#_Toc522011857)

[Adding a Delete command to an MVVM-based view 15](#_Toc522011858)

[Generalising the classes 18](#_Toc522011859)

[Exercises 21](#_Toc522011860)

[AD.2.0 21](#_Toc522011861)

[AD.2.1 22](#_Toc522011862)

[AD.2.2 23](#_Toc522011863)

[AD.2.3 24](#_Toc522011864)

# Introduction

The previous chapter has covered a lot of ground, going from a simple data bind­ing between two GUI elements, to a collection-based data binding to a Master/Details view, using command objects for invoking functionality. Still, our approach is not very robust with regards to separation of visual presen­tation and domain logic. We have not really stated anything about the roles of the **Car** and **CarCatalog** classes in the example, but it seems a fair assumption that the **Car** class is a domain class, and that other domain-oriented classes will use **Car** objects to perform some sort of business logic. The **CarCatalog** class could be perceived as such a domain class as well. In other words: the **Car** and **CarCatalog** classes should not be involved in how data relating to cars are presen­ted to the user.

In our example, we had fairly modest requirements with regards to visual presenta­tion of cars, so we could get by with using the simple properties available on the **Car** class. We will now pursue an architecture for our application that clearly separates the domain classes from the presentation-oriented part of the application. One such architecture is the **MVVM** architecture.

# The MVVM application architecture - Overview

The acronym **MVVM** is short for **Model-View-ViewModel**, and is as stated above found­ed on the principle of clear separation between presentation and domain logic. We did have some degree of separation in the previous example, in the sense that the presentation was defined in one class, while the domain logic – tiny as it was – was confined to the **Car** class. This can be described as a Model-View architecture:

Model

View

In this setup, the View classes will have explicit knowledge about the Model classes, and will thus be vunerable to changes in the Model classes. Likewise, the Model class­es may have to contain properties that are only present for making life easier for the View classes, and may even need to be up­dated if the view-oriented requirements for the application change.

The remedy for this situation is to insert a layer between the View and the Model, aptly named the ViewModel. This layer is meant to “mediate” between the two original layers, such that each can change independently of the other:

View-Model

View

Model

# MVVM – single domain object

How would this idea manifest itself in the previous example? Let’s unwind all the way back to the example where we had a simple binding between a **Car** (domain) class and a simple view in place. At that point, the **Car** class looked like this:

**public class Car : INotifyPropertyChanged**

**{**

**private string \_brand;**

**public string Brand**

**{**

**get { return \_brand; }**

**set**

**{**

**\_brand = value;**

**OnPropertyChanged();**

**}**

**}**

**public Car()**

**{**

**\_brand = "Toyota";**

**}**

**// … and code for OnPropertyChanged**

**}**

The highlighted code is the code we added in order to make the binding to the view work properly. Essentially, this implies implementing the **INotifyPropertyChanged** interface, plus calling **OnPropertyChanged** in the **set**-part of properties. Now that we are used to this sort of code, it may look quite innocent. Still, this code is only added in order to service views wishing to present car data. This means that the **Car** class now has (at least) two main responsibilities:

* Acting as a domain class
* Acting as a “service provider” for views wishing to present car data

This is not an ideal situation, since classes should in general only have a single main re­spon­sibility. Following the MVVM architecture outlined above, we should therefore insert a new class between the **Car** class and the views. We could name this class **CarViewModel**. What responsibilities should this new class have? It should act as the service provider for views interested in presenting car data, and service the specific needs of those views. Also, it should be in contact with a **Car** object, since it will need to access car data in order to relay these data – perhaps in a modified form – to the views. More specifically, we can then conclude that the class **CarViewModel** should

* Implement **INotifyPropertyChanged**, in order to provide views with up-to-date car data (we can imagine that more than one view binds to **CarViewModel**)
* Refer to a **Car** object, and keep this object up-to-date in accordance with any actions initiated from the views (via data bindings and commands).

With this, we can begin to outline an example of a **CarViewModel** class:

**public class CarViewModel : INotifyPropertyChanged**

**{**

**private Car \_domainObject;**

**public CarViewModel()**

**{**

**\_domainObject = new Car();**

**}**

**public string Brand**

**{**

**get { return \_domainObject.Brand; }**

**set**

**{**

**\_domainObject.Brand = value;**

**OnPropertyChanged();**

**}**

**}**

**public string BrandText**

**{**

**get { return "The car is a " + \_domainObject.Brand; }**

**}**

**// ... plus OnPropertyChanged code**

**}**

As we will see later, this class does have some issues, but the general concepts are in place. We refer to a domain object (the **Car** object), and manage requests regarding data for this object. That is, we return relevant data from the object when requested, either “unprocessed” as in the **get**-part of the **Brand** property, or “processed” as in the **get**-part of the **BrandText** property, where a bit of additional text is prepended.

The term “relevant” here means that we should not just make up properties that we imagine could be of use; the properties added to the **CarViewModel** class should be driven by requirements for the views presenting car data. These requirements have ideally nothing to do with the domain-driven requirements for a **Car** class, and they should therefore be completely separated. A **Car** class should not have to change because a view-oriented requirement changes, and a view should not have to change because a detail in the **Car** implementation changes. However, the **CarViewModel** is indeed allowed to change if either the views or the domain class changes, since this is exactly its main responsibility; to mediate between a domain class and the views pre­sen­ting it.

The introduction of the **CarViewModel** class enables us to do two things:

* We can “clean up” the **Car** class, so it no longer needs to contain any elements relating to presentation
* We can create bindings using the **CarViewModel** class instead of the **Car** class

The cleaned-up – but still rather primitive – **Car** class now looks like a “pure” domain class:

**public class Car**

**{**

**private string \_brand;**

**public string Brand**

**{**

**get { return \_brand; }**

**set { \_brand = value; }**

**}**

**public Car()**

**{**

**\_brand = "Toyota";**

**}**

**}**

Bindings to **CarViewModel** are now done like this (the **Button** control has no func­tionality yet):

**<Page.DataContext>**

**<local:CarViewModel/>**

**</Page.DataContext>**

**<StackPanel>**

**<TextBox Text="{Binding Brand, Mode=TwoWay}"/>**

**<TextBlock Text="{Binding Brand}"/>**

**<TextBlock Text="{Binding BrandText}"/>**

**<Button Content="OK"/>**

**</StackPanel>**

If we run this example, we will indeed see that the first **TextBlock** control (bound to the **Brand** property in **CarViewModel**) is updated whenever we type a new value into the **TextBox** control. Placing a breakpoint in the **set**-part of the **Brand** pro­perty on the **Car** class will also reveal that the value does indeed get set in the encapsulated **Car** object. All is thus in order, and we have obtained the desired separation while main­taining the functionality. Except for one problem…

Note that we added an additional **TextBlock** control to the example, binding to the **BrandText** property – which has been moved out of the **Car** class – on the **CarView­Model** class. This text did not get updated when we typed in a new brand value. Why not…? In order for that text to get updated, someone must call **OnPropertyChanged** with **BrandText** as parameter! So far, we have always called **OnPropertyChanged** without any parameter. Let’s have a second look at the **OnPropertyChanged** code:

**protected virtual void OnPropertyChanged**

**([CallerMemberName] string propertyName = null)**

**{**

**PropertyChanged?.Invoke(this,**

**new PropertyChangedEventArgs(propertyName));**

**}**

We said earlier that we need not understand the details of this method, and this is still true. However, we note that the implementation is such that if no para­meter is given, the implementation sees – through a bit of C# magic –the name of the calling property as the actual parameter. The consequence is that if we call **OnProperty­Changed** from a property named **Brand**, the implementation sees that name as the parameter, and does what it should; it notifes all properties bound to **Brand** that the value of the property has changed.

With regards to the **BrandText** property, there are two problems:

* The property doesn’t have a **set**-part.
* Nobody calls **OnPropertyChanged** for this property

It is pretty obvious the **BrandText** property cannot have a **set**-part, since its value is a so-called **aggregated value**, where the value does not have a direct counter­part in the domain object. **BrandText** is of course a simple case, where the value is compo­sed of a single property value plus some fixed text, but we could easily imagine e.g. a longer text aggregated from several properties. The main point is in any case valid: **Brand­Text** cannot have a **set**-part, so we cannot call **OnProperty­Changed** from there.

Instead, we must look for places in the code where something happens that might affect the value of **BrandText**. Again, this case is pretty obvious, since it is only in the **set**-part of the **Brand** property such a change happens. We must therefore add an extra call to **OnPropertyChanged** here:

**public string Brand**

**{**

**get { return \_domainObject.Brand; }**

**set**

**{**

**\_domainObject.Brand = value;**

**OnPropertyChanged();**

**OnPropertyChanged(nameof(BrandText));**

**}**

**}**

The parameter to **OnPropertyChanged** is of type **string**, so we could also have writ­ten **“BrandText”** directly. However, the **nameof(BrandText)** style – which returns the name of the property as a string – is more robust, since it will be affected at compile-time, if we decide to rename **BrandText** to something else. If we had just written a string directly, we would not see any errors before running the application. With this addition, the application behaves as expected. For a simple setup like this, it is rela­tively easy to manage such dependencies, and add extra calls to **OnProperty­Changed** where needed. However, when you have a more complex setup with many depend­en­cies, you may need some sort of framework to manage the dependencies.

# MVVM – collection of domain objects

The next natural step is to extend this principle to a scenario where a collection of domain objects needs to be handled. The guiding principle is again that the domain classes should not contain presentation-oriented elements. Thererfore, a collection class **CarCatalog** for **Car** objects will be pretty simple (the **Car** class has been exten­d­ed with a few properties):

**public class CarCatalog**

**{**

**private List<Car> \_cars;**

**public List<Car> All { get { return \_cars; } }**

**public CarCatalog()**

**{**

**\_cars = new List<Car>();**

**\_cars.Add(new Car("AX 32 501", "BMW", "318i","Assets\\BMW.jpg"));**

**\_cars.Add(new Car("CP 73 001", "Volvo", "X40", "Assets\\Volvo.jpg"));**

**\_cars.Add(new Car("BK 55 734", "Opel", "Astra", "Assets\\Opel.jpg"));**

**\_cars.Add(new Car("AZ 60 922", "Toyota", "Auris", "Assets\\Toyota.jpg"));**

**}**

**}**

Our goal is – just as in the View-Model setup – to work our way towards a working Master/Details view.

# A Data view model class

A starting point is to consider how car data is presented in the “Master” part of the view. The Master part will in practice be one of the collection-oriented GUI controls, e.g. a **ListView** control. We deliberately didn’t say “how a **Car** object is presented”, since the **ListView** will not bind to a collection of **Car** objects, but rather to a collect­ion of view model objects relating to cars. The reason for this indirection is again to iso­late the domain classes from the presentation logic.

We will therefore need to create such a view model class. The purpose of this view model class is to provide properties for data binding... but who will actually bind to these properties? Since we are considering the presentation of data in the Master part of the view, it must be the Data Template defined in the ListView which will bind to these properties. The specific properties to include in this view model class will thus depend on the definition of the Data Template. If we name the class **CarData­ViewModel**, a first version of the class could look like this:

**public class CarDataViewModel**

**{**

**private Car \_obj; // Encapsulated domain object**

**public CarDataViewModel(Car obj)**

**{**

**\_obj = obj;**

**}**

**public Car DomainObject { get { return \_obj; } }**

**public string ImageSource { get { return \_obj.ImageSource; } }**

**public string Description**

**{**

**get { return $"{\_obj.Brand} ({\_obj.LicensePlate})"; }**

**}**

**}**

The **Car** class will probably contain many more properties, but for the sole purpose of presenting data in a Master view, this could be sufficient. The purpose of the Master view is to provide the user with enough data to make a well-defined selection in the view – such that further details on the selected item can be presented in the Details view – so we only need to provide a small subset of the available data. Since we will not be able to change data through the items in the Master view, we don’t need any **set**-parts of any properties here, and thus no management of depen­dencies. Note that we have also included a property **DomainObject** which returns a reference to the encapsulated domain object; this will come in handy later ☺

Moving on to the Details part of the view, the requirements are somewhat different. In the Details part, many more details about the selected item are presented, and it may even be possible to modify the data. The view model class for this purpose will thus need to contain a lot more properties than the **CarDataViewModel** class shown above, and the properties may also need to have both a **get**- and **set**-part. These addi­tional properties can either be defined in a separate class, or be added to the **Car­Data­ViewModel** class. This is mostly a matter of taste: creating two separate classes is probably most in tune with the principle of high cohesion, but they may also have significant overlap, and will inevitably increase the total number of classes in a full application. Going forward, we have chosen to expand the existing class, but the alter­native solution is equally good.

The extended **CarDataViewModel** class could then look like this:

**public class CarDataViewModel: INotifyPropertyChanged**

**{**

**private Car \_obj;**

**public CarDataViewModel(Car obj)**

**{**

**\_obj = obj;**

**}**

**// Properties for Master view part omitted for brevity**

**public Car DomainObject { get { return \_obj; } }**

**public string LicensePlate { get { return \_obj.LicensePlate; } }**

**public string Brand { get { return \_obj.Brand; } }**

**public string Model { get { return \_obj.Model; } }**

**// This is an aggregated property.**

**public string Heading**

**{**

**get { return $"A {Brand} {Model}, costs {Price} kr."; }**

**}**

**// The price of a Car can be changed.**

**public int Price**

**{**

**get { return \_obj.Price; }**

**set**

**{**

**\_obj.Price = value;**

**OnPropertyChanged();**

**OnPropertyChanged(nameof(Heading))**

**}**

**}**

**}**

Here we have assumed that it only makes sense to change the **Price** property of the car, once it has been created. It is therefore only the **Price** property that has a **set**-part. The **Heading** property is an aggregated property, and it should be “refreshed” (by a call to **OnPropertyChanged**) whenever the **Price** property changes.

We can now also use this class for establishing bindings in the Details part of the view. In the below example, we assume that the selection made in the Master part of the view will be reflected in a property called **DetailsViewModel**, which has the type **Car­DataView­Model**. Exactly where this property is defined – and how the binding to the property is established – is discussed later.

**<StackPanel>**

**<TextBlock Style="{...}"**

**Text="{Binding DetailsViewModel.Heading}"/>**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Style="{...}" Text="License plate"/>**

**<TextBox Style="{...}" IsReadOnly="True"**

**Text="{Binding DetailsViewModel.LicensePlate}"/>**

**</StackPanel>**

**...**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Style="{...}" Text="Price"/>**

**<TextBox Style="{...}"**

**Text="{Binding DetailsViewModel.Price, Mode=TwoWay}"/>**

**</StackPanel>**

**</StackPanel>**

With these bindings, we can display the relevant properties of a car in the Details view. We can even edit the properties, if it makes sense. We assumed earlier that it only makes sense to change the price of a car, and it is therefore only the bind­ing to **Price** which is a two-way binding. Also, the **TextBox** controls showing read-only data have been disabled for user input, by setting the **IsReadOnly** property to true.

# Creating a collection of data view model objects

The **ListView** control in the Master part of the view will as mentioned need to bind to a collection of data view model objects. Somebody must thus pro­duce such a collec­tion. The production process is in itself fairly simple; for each domain object in the cata­log, produce a corresponding data view model object. Code for this process will look something like this:

**public List<CarDataViewModel> CreateDataViewModelCollection(CarCatalog catalog)**

**{**

**List<CarDataViewModel> items = new List<CarDataViewModel>();**

**foreach (var item in catalog.All)**

**{**

**items.Add(new CarDataViewModel(item));**

**}**

**return items;**

**}**

Where should this functionality be placed, i.e. in which class? We will postpone that decision a bit, but for now just note that we need such a method.

# A Page view model class

We are now pretty close to having all the pieces we need in order to build a working Master/Details view. Let us recap what we have in place now:

* **DomainModel:** The **CarCatalog** class plays the role of the domain model, since it contains and maintains a collection of domain objects.
* **DataViewModel:** Exposes relevant properties relating to a single **Car** object. Used for Data Binding in a Data Template for e.g. a **ListView**, and for specific properties in the Details part.
* **CreateDataView­ModelCollection:** This method handles the transformation of domain objects into a collection of data view model objects, to be displayed in e.g. a **ListView** control.

The only piece missing is a class that ties these elements together. This class will then become the Data Context for the entire view. Since a view is typically defined in terms of a **Page** control; we will therefore call such a class a **PageViewModel**. In order to see what this class should contain, we briefly recap how the Master/Details view is sup­posed to work:

* In the Master part, a collection of **DataViewModel** objects are displayed, and the user can select one of these objects. The items should correspond to the domain objects in the domain model.
* In the Details part, details corresponding to the item selected in the Master part should be displayed. The controls displaying these details are also bound to a **DataViewModel** object.

The class must then be able to do (at least) the following:

* Keep a reference to the domain model, to enable the creation of items to dis­play in the Master view.
* Implement a method which can create a collection of data view model objects from a collection of domain objects.
* Provide a property to which the selected item in the Master view can bind. This must be a two-way binding, since changes in the selection should update the Details part of the view.

These requirements imply that the class should contain (at least) these elements:

**public class CarPageViewModel : INotifyPropertyChanged**

**{**

**private CarCatalog \_catalog;**

**private CarDataViewModel \_selectedViewModel;**

**private CarDataViewModel \_detailsViewModel;**

**public List<CarDataViewModel> DataViewModelCollection {...}**

**public CarDataViewModel SelectedViewModel {...}**

**public CarDataViewModel DetailsViewModel {...}**

**public CarPageViewModel()**

**{**

**\_catalog = new CarCatalog();**

**\_selectedViewModel = null;**

**\_detailsViewModel = null;**

**}**

**private List<CarDataViewModel> CreateDataViewModelCollection()**

**{**

**// Implementation of CreateDataViewModelCollection**

**}**

**}**

These three properties are central for the class:

* **DataViewModelCollection**: Will be bound to the **ItemsSource** property in the Master view
* **SelectedViewModel**: Will be bound to the **SelectedItem** property in the Master view with a two-way binding.
* **DetailsViewModel**: Will refer to a **CarDataViewModel** object, speci­fically that object which corresponds to the selected item in the Master view. The Details view will then bind to this property.

We can then establish the bindings for the View:

**<Page.DataContext>**

**<local:CarPageViewModel/>**

**</Page.DataContext>**

**<!--Master view-->**

**<ListView**

**ItemsSource="{Binding DataViewModelCollection }"**

**SelectedItem="{Binding SelectedViewModel, Mode=TwoWay}">**

**<ListView.ItemTemplate> ...**

**</ListView>**

**<!--Details view-->**

**<StackPanel>**

**<TextBlock Style="{...}"**

**Text="{Binding DetailsViewModel.Heading}"/>**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Style="{...}"**

**Text="License plate"/>**

**<TextBox Style="{...}"**

**Text="{Binding DetailsViewModel.LicensePlate}"/>**

**</StackPanel>**

**...**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Style="{...}"**

**Text="Price"/>**

**<TextBox Style="{...}"**

**Text="{Binding DetailsViewModel.Price,**

**Mode=TwoWay}"/>**

**</StackPanel>**

**</StackPanel>**

As long as we are only considering how to display existing data – i.e. not worrying about how to change or delete data – it seems logical that the value of **SelectedView­Model** and **DetailsViewModel** should always be identical. If we change the selection in the Master view, we expect the Details part to show detailed data corresponding to the newly selected item. Could we then just omit one of those two properties? If we stick with the display-only scenario, the answer is yes. We do however choose to keep both properties, since more complex scenarios may render it necessary to assign tru­ly different values to the properties.

The only remaining issue is the implementation of the three properties. In this dis­play-only scenario, all three are fairly straightforward to implement:

**public List<CarDataViewModel> DataViewModelCollection**

**{**

**get { return CreateDataViewModelCollection(); }**

**}**

**public CarDataViewModel SelectedViewModel**

**{**

**get { return \_selectedViewModel; }**

**set**

**{**

**\_selectedViewModel = value;**

**DetailsViewModel = \_selectedViewModel;**

**OnPropertyChanged();**

**}**

**}**

**public CarDataViewModel DetailsViewModel**

**{**

**get { return \_detailsViewModel; }**

**private set**

**{**

**\_detailsViewModel = value;**

**OnPropertyChanged();**

**}**

**}**

In the **set**-part of **SelectedViewModel**, we update the value of **DetailsView­Model**, since the two properties should – in this simplified scenario – always have the same value. The set-part of **DetailsView­Model** has been made private, since it should only be changed when the value of **SelectedViewModel** changes.

With this in place, the pieces have come together. We now have a functional, MVVM-based Master/Details view for car data, responding to user selections as expect­ed. One significant omission so far is that we have not added functionality for e.g. dele­ting an item. We will now add this functionality by using Commands, just as we did for the View-Model setup.

# Adding a Delete command to an MVVM-based view

Since the view binds to **CarPageViewModel**, all properties for binding to commands should be implemented in this class. This is not particularly difficult, and follows the same pattern as for the Model-View setup. A couple of complications do arise when we need to implement a new version of **DeleteCommand**. Let’s review the imple­men­t­ation in the Model-View setup:

**public class DeleteCommand : ICommand**

**{**

**private CarCatalog \_carCatalog;**

**public DeleteCommand(CarCatalog carCatalog)**

**{**

**\_carCatalog = carCatalog;**

**}**

**public bool CanExecute(object parameter)**

**{**

**return \_carCatalog.SelectedCar != null;**

**}**

**public void Execute(object parameter)**

**{**

**\_carCatalog.Delete(\_carCatalog.SelectedCar.LicensePlate);**

**}**

**public void RaiseCanExecuteChanged()**

**{**

**CanExecuteChanged?.Invoke(this, EventArgs.Empty);**

**}**

**public event EventHandler CanExecuteChanged;**

**}**

The problem is that even though we can still perform a deletion in the **CarCatalog** object, it is not enough. Since the domain model is not involved in the presen­tation anymore, it does not implement **INotifyPropertyChanged** or use an **Observable­Collection**. In other words; just deleting an object in the domain model will not cause the view to update accordingly.

The solution is to provide the deletion command object with more information, speci­fically a reference to the Page view model object as well. With this reference avail­able, we can extend the implementation of **Execute** (weassume a reference to the Page view model object is kept in an instance field called **\_pageViewModel**):

**public void Execute(object parameter)**

**{**

**\_carCatalog.Delete(\_pageViewModel.SelectedViewModel.LicensePlate);**

**\_pageViewModel.SelectedViewModel = null;**

**\_pageViewModel.Refresh();**

**}**

Let’s detail what happens here:

1. The domain object identified by the license plate (which acts as a key here) of the selected car is deleted from the catalog – this is just as before.
2. The selection in the Master part of the view is set to **null**. This is “by-design”; we have chosen the interaction logic to work this way.
3. Since we have deleted a domain object, we somehow need to tell the Page view model object that something has happened, which should cause it to “refresh” itself. Exactly what this “refresh” action entails it not of any con­cern to the command object.

The last point deserves a bit of elaboration. Our main concern is to ensure that the data presented in the view is consistent with the data currently stored in the data model, i.e. the catalog. Who should be responsible for this? In the above setup, it becomes the responsibility of the command object, which then becomes a kind of “controller” (in terms of GRASP patterns[[1]](#footnote-1)). Whether this is the correct choice is certainly debatable. A more elaborate solution could be to implement some sort of event-driven scheme: A catalog class could contain so-called **event** properties, to which other parts of the code can subscribe. The catalog could then raise a certain event when it is modfied; a Page view model object could then subscribe to such events, thereby making it capable of keeping itself consistent with the catalog. Such a solu­tion is however a bit beyond the scope of this chapter.

We therefore stick with the somewhat simplistic scheme, where the Page view model class implements a public **Refresh** method. The **Refresh** method itself is quite simple:

**public void Refresh()**

**{**

**OnPropertyChanged(nameof(DataViewModelCollection));**

**}**

The revised implementation of the **DeleteCommand** then becomes:

**public class DeleteCommand : ICommand**

**{**

**private CarCatalog \_catalog;**

**private CarPageViewModel \_pageViewModel;**

**public DeleteCommand(CarCatalog catalog, CarPageViewModel pageViewModel)**

**{**

**\_catalog = catalog;**

**\_pageViewModel = pageViewModel;**

**}**

**public bool CanExecute(object parameter)**

**{**

**return \_pageViewModel.SelectedViewModel != null;**

**}**

**public void Execute(object parameter)**

**{**

**// Delete from catalog**

**\_catalog.Delete(\_pageViewModel.SelectedViewModel.LicensePlate);**

**// Set selection to null**

**\_pageViewModel.SelectedViewModel = null;**

**// Refresh the item list**

**\_pageViewModel.Refresh();**

**}**

**public event EventHandler CanExecuteChanged;**

**public void RaiseCanExecuteChanged()**

**{**

**CanExecuteChanged?.Invoke(this, EventArgs.Empty);**

**}**

**}**

We note that the command is only allowed to execute if a selection has been made in the view (the **CanExecute** method), and that the **Execute** metod retrives the license plate for the currently selected item, and uses it when calling **Delete** on the catalog.

# Generalising the classes

These final additions bring us to a major milestone – we now have a Master/Details view including command-based functionality, playing entirely by the MVVM rule­book. The obvious question is now: what will it take to implement the same functionality for another domain class? On an architectural level, the entire class structure should be highly reusable, since very few parts of it relies on specific details for a domain class. The only assumption has been that the domain class is “simple”, i.e. that all properties have a simple type.

If this assumption is met, it should almost be a matter of copy-paste plus search-and-replace for adapting the structure for a different domain class. Some details will of course be domain-specific, like the properties in the **CarDataViewModel**, but quite a lot of code has a very generic nature. Consider for instance the below outline of a **CarCatalog** class:

**public class CarCatalog**

**{**

**private Dictionary<string, Car> \_objects;**

**public CarCatalog()**

**{**

**\_objects = new Dictionary<string, Car>();**

**}**

**public List<Car> All**

**{**

**get { return \_objects.Values.ToList(); }**

**}**

**public void Create(Car obj)**

**{**

**\_objects.Add(s.LicensePlate, s);**

**}**

**public void Delete(string licensePlate)**

**{**

**\_objects.Remove(licensePlate);**

**}**

**}**

This is rather simplistic, but does provide basic catalog functionality. How much diffe­rent would a catalog class for a different domain class be? **Car** should of course be replaced with the name of the new domain class, and the property acting a key will probably not be called **LicensePlate** either. Also, it might not have the type **string**. Is it possible to turn these characteristics of the **CarCatalog** into parameters for a more general **Catalog** class? Indeed it is, at least to some extent.

We have not discussed the topic of **Generics** in C# yet, but we have actually used it a lot. We have often written e.g. **List<Car>**, meaning “a **List** object that can hold a col­lec­tion of **Car** objects”. **Car** is here used as a specifc kind of parameter; a **type** para­meter. List-based functionality is very generic, so it would be nice if you could write that functionality once, and then use it no matter the type of objects you put into the list. Generics enable you to do just that. An outline of a generalised **Catalog** class using Generics is given below, where we have introduced a **type parameter** **T**.

**public class Catalog<T>**

**{**

**private static int \_keyCount = 1;**

**private Dictionary<int, T> \_objects;**

**public Catalog()**

**{**

**\_objects = new Dictionary<int, T>();**

**}**

**public List<T> All**

**{**

**get { return \_objects.Values.ToList(); }**

**}**

**public void Add(T obj)**

**{**

**obj.Key = \_keyCount++;**

**\_objects.Add(obj.Key, obj);**

**}**

**public bool Delete(int key)**

**{**

**return \_objects.Remove(key);**

**}**

**}**

We are not yet fully equipped to understand such code, but the point is just to illu­strate that we can parameterise the code to such an extent that it can be reused for all domain classes we wish to include. With a class like the above available, we could change e.g. the type of an instance field in the **CarPageViewModel** class:

**// private CarCatalog \_catalog; (Original)**

**private Catalog<Car> \_catalog;**

Instead of having to create a new catalog class for each domain class, we could then simply reuse the generic version. All we need to do is specify the actual domain type for which we need a catalog. We will not go into further details with this topic here, but revisit it when we have learned more about **Generics**, since Generics is a crucial tool to use for this purpose.

# Exercises

|  |  |
| --- | --- |
| **Exercise** | AD.2.0 |
| **Project** | ExamAdmV20 |
| **Purpose** | Change the given application from a Model-View (MV) architecture to a Model-View-ViewModel (MVVM) architecture. |
| **Description** | The project initially contains a class **Student**, which acts both as a domain class and a “service provider” to the main view (via data bindings in MainPage.xaml). |
| **Steps** | Add a new class **StudentDataViewModel** to the project, which will acts as the ViewModel in an MVVM architecture. This involves:   1. Create the class **StudentDataViewModel**. 2. Add an instance field **\_domainObject** of type to **StudentData­View­Model**, and initialise it to refer to a new **Student** object in the constructor. 3. Let **StudentDataViewModel** inherit from **INotifyPropertyChanged**, and generate the code needed (Tip: click the lightbulb ). If the includes are not generated automatically, add to the top of the file:   using System.ComponentModel;  using System.Runtime.CompilerServices;   1. Add properties **Name**, **Subject** and **Score** to **StudentDataViewModel**, in the style described in the notes. 2. Clean up the **Student** class, such that it no longer inherits from **INotify­PropertyChanged** 3. Change the data context in **MainPage.xaml**, and check that the new bindings work as expected. 4. Now create a new property **TopLineText** in **StudentDataViewModel**, of type **string**. The intention is that this property should provide enough information to enable you to delete the six **TextBlocks** in the top line of the GUI, and replace them with a single **TextBlock**, that binds to **TopLineText**. 5. Delete the six **TextBlocks**, replace them with a single **TextBlock**, and bind the new **TextBlock** to **TopLineText**. Are changes to the data reflected in the top text line? 6. Add extra calls of **OnPropertyChanged** to the **Name**, **Subject** and **Score** properties, in the style described in the notes. Check that changes are now reflected in the top text line |

|  |  |
| --- | --- |
| **Exercise** | AD.2.1 |
| **Project** | ExamAdmV21 |
| **Purpose** | Add and use view model classes in an application |
| **Description** | The given application provides simple read-only functionality for a collection of students (only the Master part of a Master/Details view). However, the view uses the **StudentCatalog** class as Data Context. For now, the class **StudentDataViewModel** is not used. |
| **Steps** | 1. Add a new class **StudentPageViewModel** to the project. The intention is that MainPage.xaml should use this class as its new data context. The class must contain a constructor that does not take any parameters, 2. Add an instance field **\_studentCatalog** of type **StudentCatalog** to the class, and initialise it in the constructor, to refer to a new **StudentCatalog** object. 3. Take a look at the **StudentDataViewModel** class. Note that the construc-tor takes a parameter of type **Student**. 4. Add a private method **CreateStudentDataViewModelCollection()** to the **StudentPageViewModel** class (it’s okay to use a shorter name ). The return type of the method should be **List<StudentDataView­Model>**. The method should build the list by looping through the list of **Student** objects in the catalog (Hint: use the **Students** property in the **StudentCatalog** class), and create a new **Student­DataViewModel** object for each **Student** object. 5. Add a property**StudentDataViewModelCollection** to **Student­Page­View­Model**. Only the **get**-part of the property is needed; it should look like this: **{ return CreateStudentDataViewModelCollection(); }**. 6. Change the data context in MainPage.xaml to **Student­Page­View­Model** instead of **StudentCatalog**, and change the binding of the **ListView** proper­ty **ItemsSource** from **Students** to **StudentDataViewModel­Collection**. 7. Rebuild the application, and check that the data is still shown properly when running the application. 8. Do these changes enable you to clean out any properties from **Student** and **StudentCatalog**, that were only there to supply GUI-specific data? |

|  |  |
| --- | --- |
| **Exercise** | AD.2.2 |
| **Project** | ExamAdmV22 |
| **Purpose** | Add deletion functionality to a working read-only Master/Details view. |
| **Description** | The application contains a functional read-only Master/Details view, where students can be viewed. The application uses the MVVM archi­tecture. We now want to add deletion functionality to the application |
| **Steps** | In the **StudentPageViewModel** class (in the **ViewModel** folder):   1. Initialise the \_**deletionCommand** instance field, by creating a new **Delete­Command** object. Note that the **DeleteCommand** constructor requires two parameters; one of type **StudentCatalog**, one of type **StudentPage­View­Model**. 2. Add a new property **DeletionCommand** (only the **get**-part is needed) of type **ICommand**, that returns **\_deletionCommand**.   In MainPage.xaml (in the **View** folder)   1. Find the **Delete** button (just after the **ListView** control), and bind its **Command** property to **DeletionCommand** 2. Build and run the application. Does the **Delete** button work as it should (probably not…)?   Back in the **StudentPageViewModel** class:   1. At the end of the **set**-part of the **SelectedStudent­** proper­ty, add a call of **RaiseCanExecuteChanged** on \_**deletionCommand**. 2. Build and run the application again. Does the **Delete** button now work as it should (Hopefully it does )?   Consider the below questions:   1. Which class in the project has knowledge about all the steps involved in deleting a student (deleting a **Student** object from the catalog, and upda­ting the GUI accordingly)? What do we usually call classes of this kind? 2. Imagine we had to add a new domain class **Teacher** to the application, with the same functionality (Master/Details view, Delete functionality). How many new classes would we have to create for that purpose? How similar (or different) will these classes be compared to their **Student**… counterpart? |

|  |  |
| --- | --- |
| **Exercise** | AD.2.3 |
| **Project** | ExamAdmV23 |
| **Purpose** | Rewrite domain-specific classes to use provided base classes |
| **Description** | The project contains a working Master view with delete function­a­li­ty (the Details part has been omitted). All domain-specific classes are located in the folder **DomainClasses**. A number of base classes are available in the folder **BaseClasses**, but are not used yet. |
| **Steps** | 1. Let the **Student** class inherit from **DomainClassBase**. This will require that you override the property **Key** (you just add the keyword **override** to the pro­perty). 2. Let the **StudentCatalog** class inherit from **CatalogBase<Student>**. You can then delete everything else from the **StudentCatalog** class, except the con­structor. In the constructor, change the calls of **Create** to use the base class method **Add**. 3. Let the **StudentDataViewModel** class inherit from **DataViewModel­Base<Student>**. Remove the existing lines of code from the constructor body. The constructor must then call the base class constructor with **obj** as parameter. Also delete the instance field **\_domainObject** and the property **DomainObject**, and replace the use of **\_domainObject** with **DomainObject** in the other properties (this will then refer to **DomainObject** from the base class). 4. Let the **StudentPageViewModel** class inherit from **Page­ViewModelBase<Student>** (it no longer needs to inherit from **INotifyPropertyChanged**). Delete everything else from the class definition, except the constructor and the method **CreateData­ViewModel**. 5. Implement the constructor for **StudentPageViewModel** like this**:**   public StudentPageViewModel() : base(new StudentCatalog())  {}   1. Implement the method **CreateData­ViewModel** like this**:**   public override DataViewModelBase<Student> CreateDataViewModel( Student obj)  {  return new StudentDataViewModel(obj);  }   1. Let the **DeleteCommand** class inherit from **DeleteCommandBase<Student, StudentPageViewModel>**. Delete everything else from the class definition, except the constructor. Remove the existing lines of code from the constructor body. The constructor must then call the base class constructor with **catalog** and **viewModel** as parameters. 2. In **MainPage.xaml**, change the binding of **ItemsSource** to **DataView­Model­Collection**, and the binding of **SelectedItem** to **Item­Selected** 3. Make sure all files are saved, then build and run the application. Check that the original functionality is retained. |

1. https://en.wikipedia.org/wiki/GRASP\_(object-oriented\_design) [↑](#footnote-ref-1)