MVVMStarter Guide

This document contains a general description of the **MVVMStarter** project, and specific guidelines for how to add a new domain class to the project.

Table of Content

[Introduction 2](#_Toc481952012)

[Purpose 2](#_Toc481952013)

[Status (May 07 2017) 3](#_Toc481952014)

[GUI Functionality 4](#_Toc481952015)

[Framework structure 7](#_Toc481952016)

[Adding filters to a Catalog class 10](#_Toc481952017)

[Adding and using hard-coded domain objects 12](#_Toc481952018)

[Adding and using hard-coded Image objects 13](#_Toc481952019)

[Defining state-specific view appearances 17](#_Toc481952020)

[Performing property validation 19](#_Toc481952021)

[Setting up property dependencies 21](#_Toc481952022)

[Accessing domain objects 22](#_Toc481952023)

[Adding a domain class - details 25](#_Toc481952024)

# Introduction

The **MVVMStarter** project is a framework coded in C# (and XAML), and built on the principles of the **Model-View-ViewModel** (**MVVM**) application architecture. The domain objects are intended to be completely separated from the presentation, and the presentation can be customised in various ways without any direct involve­ment of the domain objects.

# Purpose

The intention of the **MVVMStarter** framework is to make it relatively easy for a user of the framework to get basic CRUD (Create, Read, Update, Delete) functionality up-and-running for simple (see later) domain classes. Once the functionality is in place, the framework user can spend time on creating more complex business logic involving the added domain classes.

The project rests on the assumption that once a simple domain class is defined, it is essentially trivial to create the code needed for showing objects of the class in a GUI, as well as saving the objects to persistent storage. By defining a set of feature-rich base classes in C#, a framework user can then add his own domain classes by inheriting from these base classes. Only the domain-specific parts of the domain class need to be explicitly specified.

In order to make the process of adding a new domain class as easy as possible, the project itself con­tains templates for all new files that need to be created, plus specific examples of code that needs to be added in specific places in the existing files.

# Status (May 07 2017)

The **MVVMStarter** project currently supports:

* Handling CRUD operations for **domain classes**. Domain classes can contain properties of simple type, but may also refer to other domain classes, using a unique **int** key as reference (see below).
* Showing domain objects in a **Master/Details view**, one view per domain classes. All CRUD operations are done in the same view, but the view can be configured to appear differently for each type of operation.
* **Filtering** of domain objects on multiple criteria. You can add filters to a catalog, such that only those domain objects meeting the filter criteria are shown in the view.
* Saving domain objects to **files in JSON format**. This functionality comes out-of-the-box, without the need for explicit code. You can also choose to add “hard-coded” domain objects – domain objects created directly in the source code – in a manner that makes it transparent whether the objects originate from a file or are hand-coded.
* Add and use hard-coded **image objects**. The image objects can be tagged with multiple tags, and the collection of image objects can then be filtered according to tags.
* Performing **simple single-property validation**, like checking that numeric values are within a certain range, strings contain certain sub-strings, etc..
* Management of **cross-property update dependencies**. An aggregated property shown in the GUI – without a direct underlying domain-property – may need to be refreshed when the properties on which the aggregated property relies are updated.
* Management of **unique key assignment** on a per-class basis. Domain objects of a specific class will be assigned a unique numerical key upon creation, which can later be accessed program­matically. Keys are retained when saving/loading objects. It is therefore possible to let domain objects refer to other domain objects, using said key as reference.

Within these limitations, the system can still be extended by users, since most key methods are declared as **virtual**. This will of course require that the user has deeper knowledge of how the base classes work and cooperate.

NB: The **MVVMStarter** project is now only maintained on **GitHub**:

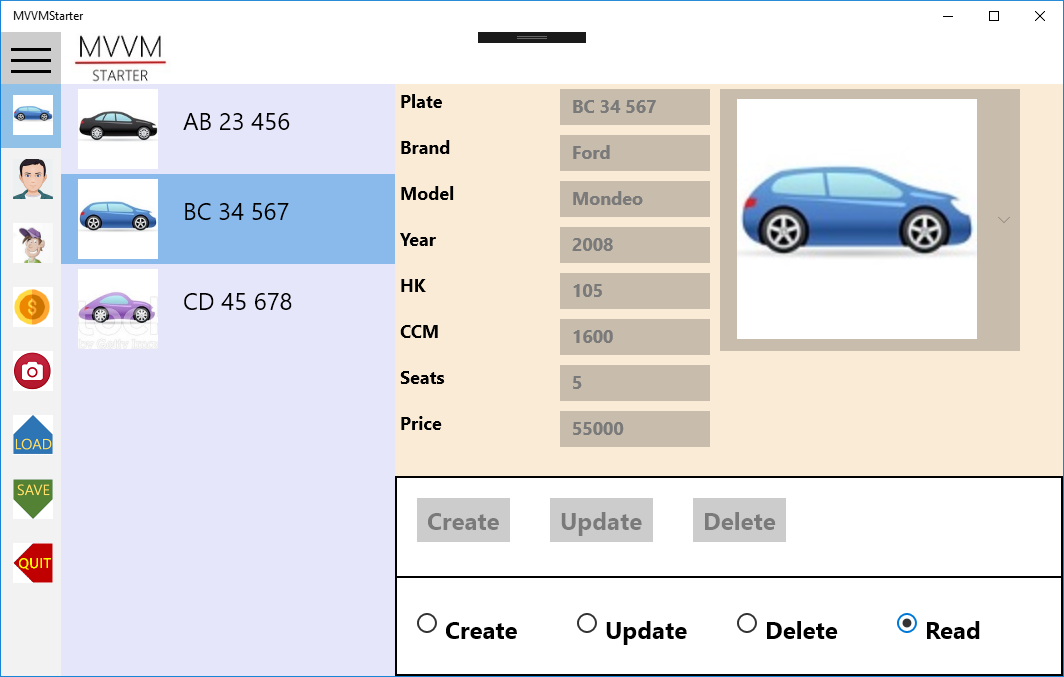
<https://github.com/perl-easj/MVVMStarter>

# GUI Functionality

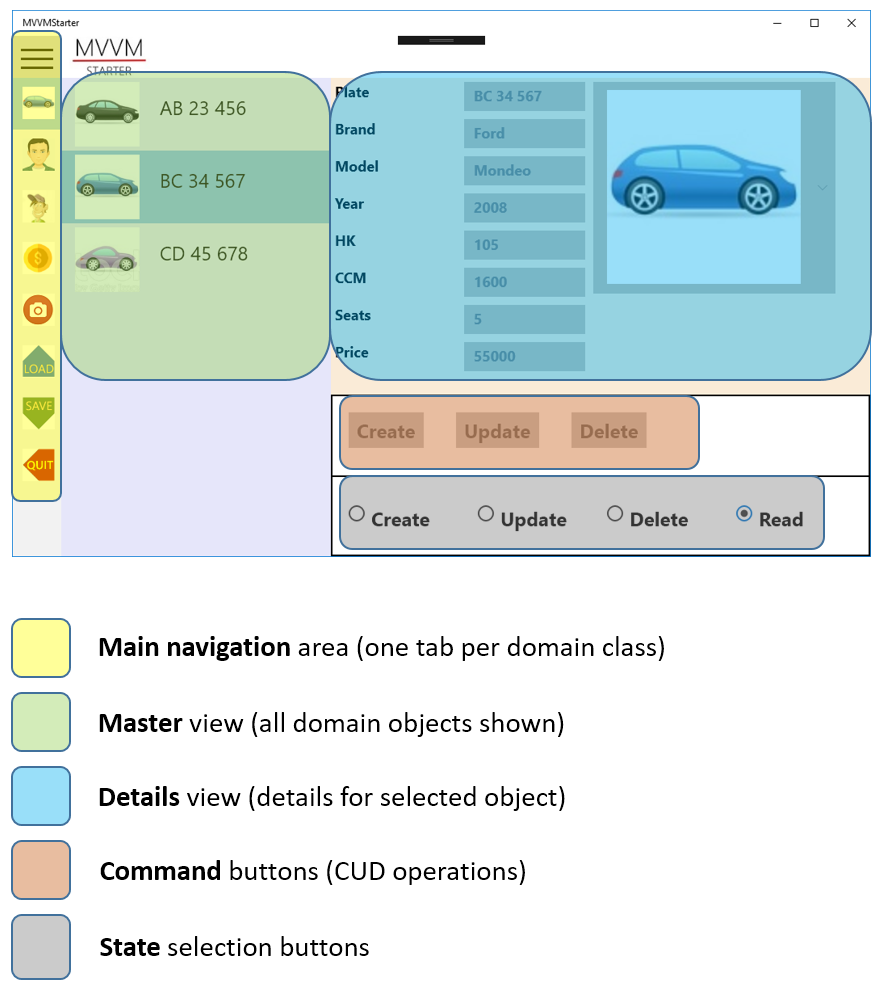
The primary functionality offered by **MVVMStarter** is the ability to show domain objects of a particu­lar class in a so-called **Master/Details** view. The main characteristics of a Master/Details view are:

* In the **Master** part of the view, all domain objects are presented in a compact form. The compact form should contain just enough information to allow the user to confidently (i.e. be certain of the object’s identity) select one of the objects. A typical GUI control for this purpose is the **ListView** or **GridView** control.
* In the **Details** part of the view, detailed information about the selected object is shown. This will often be most (if not all) of the properties in the domain object, and possibly some aggre­gated properties as well. This information is usually shown as a combination of **TextBlock** and **TextBox** controls, and perhaps **CheckBox** controls or similar controls for information of a true/false nature.
* The view will usually contain action-oriented control (like e.g. a **Button** control), from which various actions can be invoked, like deletion of a selected object, insertion of a new object, etc..

In the **MVVMStarter** project, the “default” view looks like the below example:



The view shows data for one specific domain (**Car** in this example). The view is composed of five main parts:



**Main navigation**: This is a so-called Hamburger menu, to which a number of tabs can be added. The **Image**, **Load**, **Save** and **Quit** tabs are fixed. A new tab should be added for each domain class. Clicking on a tab brings you to the Master/Details view for the selected domain class.

**Master** View: A **ListView** control, which show a collection of so-called **ItemViewModel** objects to the user. Each **ItemViewModel** object corresponds to an underlying domain object. The mapping of domain object data to the data presented by the **ItemViewModel** object is defined by the user, in a domain-speci­fic **ItemViewModel** class. A default implementation is available, which only requires the user to override a few virtual methods from the **ItemViewModel** base class.

**Details** view: This part of the view will be domain-specific, depending highly on the nature of the data in the domain object. Text data is usually presented using **TextBlock**/**TextBox** controls, but other con­trol types may also be appropriate, like **CheckBox**, **ToggleSwitch**, etc.. The user must implement a **DetailsViewModel** class, specifying the details of how to map data from the underlying domain object to the corresponding part of the Details view. Dependencies between simple and aggregated proper­ties are also specified here.

**Command** buttons: These are **Button** controls to which base-class versions of CUD commands are bound. The state of each button (visibility, enabled/disabled) can be specified to be dependent on the state of the view (see below). The user can freely override the control bindings in the domain-specific view, to use domain-specific commands instead. Alternatively, the user can keep the existing controls, and add extra domain-specific controls next to them.

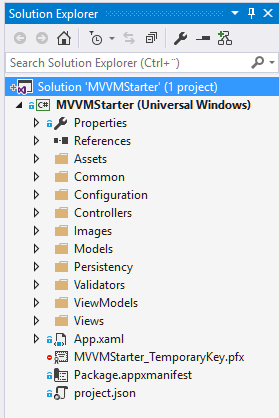
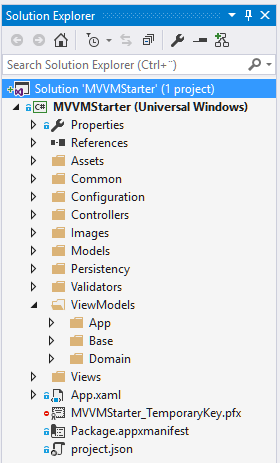
**State selection** buttons: The view is defined to always be in one of four specific states, corresponding to the four CRUD operations. Since all controls can be configured to be visible/collapsed and enabled/ disabled depending on the selected state, the visual appearance of the view can change from state to state. It is up to the user of the framework to define the specific configuration of the view for each view state.

Entire **Master/Details** view: Corresponding to the view as a whole, the framework user must imple­ment a domain-specific class **MasterDetailsViewModel**, which inherits from **MasterDetails­View­ModelBase**. All functionality as such is located in the base class, so the domain-specific class just offers the user the opportunity to specifiy configuration details, as mentioned above.

# Framework structure

The **MVVMStarter** framework relies heavily on active use of namespaces and type parameteri­sa­tion. The active use of namespaces implies a rather strict hierarchical file structure, where the specific loca­tion of a file bears meaning in inself.

At the top level, the framework project consist of folders corresponding to the main layers or areas in the MVVM architecture, plus some additional folders:

Within each folder, there may be up to three additional folders called **App**, **Base** and **Domain,** as seen in the **ViewModels** folder. Note that it is not all folders that contain these three folders.

**App**: Folders named **App** will contain classes that are not tied to a specific domain, but rather supply some general-purpose (but still related to a specific area of the framework) functionality like file persistency, presenting dialogs to the application user, etc..

**Base**: Will contain base classes that domain-specific classes can inherit from, within a specific area of the framework

**Domain**: Will contain a sub-folder called **Template**, and one sub-folder for each domain class, as seen in the **ViewModels/Domain** folder. The **Template** folder contains files that show examples of how to create a domain-specific file of the same kind. This is often just a matter of copy-paste and replacing a few text occurrences. The domain-specific folders (like **Photo**) will contain the domain-specific classes.

The top-level folders serve these purposes (detailed descriptions of how to add domain-specific classes in each area follow later):

**Assets**: Is only used for storing generic and domain-specific image files. If the user wants to create an application-specific splash-screen, Store logo, etc., he is free to do so.

**Common**: Contains a few utility classes. General-purpose classes should be added here.

**Configuration**: Only contains the file **AppConfig.cs**, which contains a few application-wide constants, and management of **Load**/**Save** functionality.

**Controllers**: Business logic is generally encapsulated into **Controller** classes. The framework offers basic CUD functionality in the form of three controller-like base classes, that also implement the **ICommand** interface. A controller object can thus be tied to the **Command** property for a GUI control. A user can add domain-specific controller classes here as well, and/or override the existing classes.

**Images**: The framework contains functionality for creating and using images. The classes created for this purpose are located in this folder.

**Models**: The **Base** folder contains base classes for domain-specific classes. For each domain class, the user must create a corresponding sub-folder in the **Domain** folder, and there create the domain class itself, plus a class called **Catalog**.

**Persistency**: The framework offers out-of-the-box file persistency. The user does not need to create any domain-specific classes here.

**Validators**: The framework offers simple validation of individual property values. The user only needs to create domain-specific classes here, if validation is required.

**ViewModels**: The **Base** folder contains base classes for domain-specific view model classes. The base classes follow the class structure driven by the Master/Details view: **ItemViewModelBase**, **Master­ViewModelBase**, **DetailsViewModelBase**, **MasterDetailsViewModelBase** and **ViewModelFactory­Base**. The user must create corresponding domain-specific view model classes

**Views**: Contains XAML definitions of a couple of application-level views, plus a template **View.xaml** (in **Domain**/**Template**) for domain-specific views. The user must create domain-specific view defini­tions here. It is not as such required that the user follows the provided template.

# Adding filters to a Catalog class

A **Catalog** is the class that holds an in-memory collection of domain objects. This collection can be retrieved from a **Catalog** object in the form of a **List** of domain objects, through the property **All**. The property **FilteredAll** also retrieves a **List** of domain objects, but now filtered through the set of cur­rently active filters.

A **Filter** is essentially just a method, that takes a single domain object as a parameter, and returns a **bool** value. If the method returns **true**, the object is said to have passed the filter.

A filter is declared in the domain-specific **Catalog** class, like this:

public Filter<Car> PriceFilter;

private bool PriceFilterCondition(Car obj) { return obj.Price < 100000; }

The filter is then created – specifying an ID and the filter condition itself – and added to the catalog in the **Catalog** constructor:

private Catalog() : base(...)

{

PriceFilter = new Filter<Car>("PriceFilter", PriceFilterCondition);

AddFilter(PriceFilter);

}

In the domain-specific **MasterDetailsViewModel** class, you can now add a suitable property used for toggling the filter on and off, by referring to the **On** property on the **Filter** object.:

public bool PriceFilterOn

{

get { return CarCatalog.Instance.PriceFilter.On; }

set

{

CarCatalog.Instance.PriceFilter.On = value;

OnPropertyChanged(nameof(ItemViewModelCollection));

}

}

The call of **OnPropertyChanged** ensures that the collection shown in the **Master** part of the view gets refreshed after changing the filter state. You must also set up the view state for the control which will bind to this property, in the **MasterDetailsViewModel** class constructor:

StateManager.AddViewControlState(new ViewControlState("PriceFilter", "Price below 100k"));

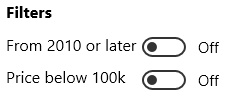
The first parameter is the identifier used for control appearance bindings in the view, and the second parameter is the lead text you can retrieve for use in the view.

You can now create a suitable GUI control in the **View.xaml** file – e.g. a **ToggleSwitch** – and bind to it.

<TextBlock Text="{Binding ViewControlStates[PriceFilter].Description}"/>

<ToggleSwitch IsOn="{Binding PriceFilterOn, Mode=TwoWay}"/>

You can place these controls anywhere in the view you find appropriate:



In summary, the steps needed to create and use a filter are:

1. Define the filter itself in the domain-specific **Catalog** class, and add it to the catalog
2. Add a property of type **bool** in the domain-specific **MasterDetailsViewModel** class, and use it to toggle the filter on and off
3. Bind to the property from the view, e.g. using a ToggleSwitch control. The binding must be a two.way binding.

# Adding and using hard-coded domain objects

The standard functionality for domain classes w.r.t. storage is that domain objects are written to a file when **AppConfig.Save** is invoked, and subsequently read again when **AppConfig.Load** is invoked. It is however possible to create domain objects directly in the source code. This can be relevant in a test scenario, or if it for some reason is difficult (or irrelevant) to define a proper GUI for creating these domain objects.

Even so, we would still like to use the standard framework for managing these domain objects, once they have been created. Therefore, the (small) class **HardCodedObjectsBase** is available. The class is essentially just a wrapper around a list of domain objects. If you wish to create domain objects for e.g. a domain class called **Contact**, you should then create a class in the **Models**/**Domain**/**Contact** folder called **HardCodedObjects**, inheriting from **HardCodedObjectsBase**:

public class HardCodedObjects : HardCodedObjectsBase<Contact>

{

public HardCodedObjects()

{

ObjectList.Add(new Contact(...)); // Add domain objects to list…

ObjectList.Add(new Contact(...));

...

}

}

You can then create a normal **Catalog** class from the available template, with a single modifi­cation:

public class Catalog : CatalogBase<Contact>

{

... (default implementation)

private Catalog()

: base(new CollectionBase<Contact>(),

HardCodedSourceBase<Contact>(new HardCodedObjects()))

{

}

}

Creating the **Catalog** class in this way makes it transparent to the rest of the application where the **Contact** objects originate from. The objects will not be available in the **Catalog** before **Load** is invoked, just as for the domain objects using a file source. Saving will – of course – not have any effect.

# Adding and using hard-coded Image objects

It will in many situations be useful to use images as part of a domain object. If you are creating a system for e.g. car sales, adding an image of the car would be natural. The framework contains facilities for handling images. The classes for this purpose are found in the **Images** folder.

The **Image** class represents a single image. The class inherits from **DomainClassBase**, and therefore contains a key of type **int**. Furthermore, the **Image** class contains:

* A **description** of type **string**
* A **source** (a path to a file, or a URL) of type **string**
* A list of **tags** (of type **string**) associated with the image

The idea is thus that you can associate a number of tags to a given image; these tags can later be used to filter out specific images.

Specific **Image** instances are now defined in a **HardCodedObjects** class, which simply contains code for creating the **Image** objects needed, like this (the string **imageFilePrefix** points to a “default” location for actual image files, but you can use any location you prefer):

public HardCodedObjects()

{

string imageFilePrefix = "..\\..\\..\\Assets\\Images\\";

Image c1 = new Image("Red Sedan", imageFilePrefix + "CarRedSedan.jpg");

c1.AddTag("Car");

c1.AddTag("Red");

c1.AddTag("Sedan");

c1.AddTag("Individual");

Image c2 = new Image("Blue Combi", imageFilePrefix + "CarBlueCombi.jpg");

c2.AddTag("Car");

c2.AddTag("Blue");

c2.AddTag("Combi");

c2.AddTag("Family");

Image s1 = new Image("Ann", imageFilePrefix + "Ann.jpg");

s1.AddTag("Student");

s1.AddTag("Female");

ObjectList.Add(c1);

ObjectList.Add(c2);

ObjectList.Add(s1);

}

With the images defined in this way, the usual **Catalog** class is then used for storing and accessing the **Image** objects at run-time.

public class Catalog : CatalogBase<Image>

{

...

private Catalog()

: base(new CollectionBase<Image>(),

new HardCodedSourceBase<Image>(new HardCodedObjects()))

{

}

// Retrieves all Image objects tagged with the given tag

public List<Image> AllWithTag(string tag) {...}

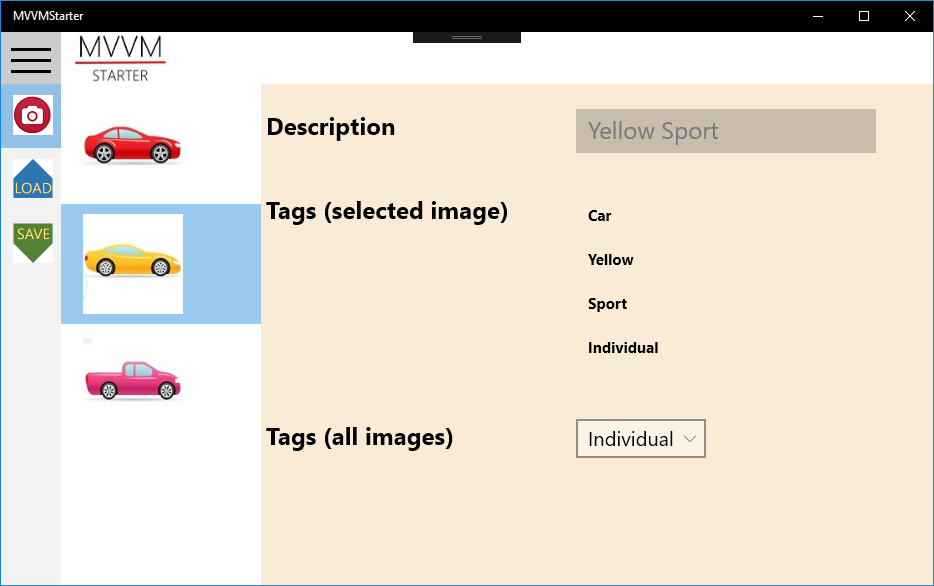
// Retrieves all tags used in the collection of Image objects

public List<string> AllTags {...}

}

The use of **HardCodedSourceBase** makes it transparent to the rest of the application that the **Image** objects are in fact hard-coded. **AllWithTag** and **AllTags** are particular for this **Catalog** class, and are used for filtering **Image** objects according to tags.

The collection of **Image** objects can be viewed in the ***Images*** view in the application.



This view allows you to browse through the created **Image** objects, filter according to a chosen tag, and see the tags associated with a specific image. You can, however, not change any data in the view. The view only displays the data defined in the **HardCodedObjects** class.

The **Image** objects can now be used as an image source for domain objects. Suppose you have defined a **Car** class, which should refer to an image:

public class Car : DomainClassBase

{

private int \_imageKey;

private string \_licensePlate;

private int \_price;

...

}

The view for creating **Car** objects could then contain a GUI control (e.g. a **ComboBox**) for choosing an image to associate with the car. In the corresponding **DetailsViewModel** class, you could define pro­per­ties for providing images to choose from – specifically, those images tagged with ***“Car”*** – and to keep track of the selected image, like:

public ObservableCollection<Image> CarImageCollection

{

get

{

var collection = new ObservableCollection<Image>();

ObjectProvider.ImageCatalog.AllWithTag("Car").ForEach(collection.Add);

return collection;

}

}

public Image CarImageSelected

{

get { return ObjectProvider.ImageCatalog.Read(DomainObject.ImageKey); }

set

{

if (value != null)

{

DomainObject.ImageKey = value.Key;

}

OnPropertyChanged();

}

}

Finally, bindings to these properties can be defined in the View:

<ComboBox

ItemsSource="{Binding DetailsViewModel.CarImageCollection}"

SelectedItem="{Binding DetailsViewModel.CarImageSelected, Mode=TwoWay}">

<ComboBox.ItemTemplate>

...

</ComboBox.ItemTemplate>

</ComboBox>

A couple of closing remarks about using images in the **MVVMStarter** Framework

* It is optional to use the described built-in image handling facilities. If you prefer to define and use your own way to handle images, you are free to do so.
* It is optional to store images in the suggested location (**Assets**/**Images**). You can store image files anywhere you prefer, even online (the source string will then be a URL).
* Even though the image management system uses the “standard” **Catalog** class and **Hard­Coded­Objects** class for storage, you should not use the structure of the classes and views for image management as a “template” for adding domain classes to the framework. The **Image** class is not considered to be a domain class, but rather a support class for true domain classes. Please follow the guidelines for adding actual domain classes instead.

# Defining state-specific view appearances

It is possible to configure the appearance of the Master/Details view according to the selected state. It may for instance be required that only certain properties of a domain object can be changed after creation, meaning that only certain controls in the view should be enabled in the Update state. Also, it may be required to hide certain elements of the view, e.g. buttons not available in the given state.

A GUI control inside a view – referred to as a **ViewControl** – is considered to have four properties:

|  |  |  |
| --- | --- | --- |
| **Property Name** | **string** | Actual name of the property, as defined in the **DetailsViewModel** class |
| **Description** | **string** | Textual description of the property, to be displayed as the lead text for the property in the view |
| **Enabled** | **bool** | Is the control enabled? |
| **Visible** | **bool** | Is the control visible? |

The class **ViewControlState** holds these four properties. The class contains definitions of a default control state for list view, properties and buttons. It also contains a number of constructors, to make it easy to create the appropriate **ViewControlState** object.

The idea is that a **ViewControlState** object can be associated with a specific view state, such that separate **ViewControlState** objects define the behavior of the control for each state. Specific state definitions are then defined in the constructor of the domain-specific **MasterDetailsViewModel** class, by calling methods on the **StateManager** property (of type **ViewControlStateManager**)

public MasterDetailsViewModel()

: base(...)

{

StateManager.AddViewControlState(

ViewControlState.ViewState.Update,

new ViewControlState("LicensePlate", "License plate", false, true));

}

This example should be read as *“In the* ***Update*** *state, the* ***LicensePlate*** *property should be* ***visible****, but* ***not enabled****”*. In many cases, many controls will share the same behavior, and many controls may even have the same behavior in all view states. The **ViewControlStateManager** therefore contains methods for setting many control state definitions in one method call; see the class for more details.

It is important to note that the view will not function, if view control states have not been defined for all controls in the view. If it is acceptable to go with the default behavior, which is as follows:

|  |  |
| --- | --- |
| **All states** | All lead texts identical to property names.  All buttons visible.  All properties visible. |
| **Create** | Only **Create** button enabled.  All properties enabled. |
| **Read** | All buttons disabled.  All properties disabled. |
| **Update** | Only **Update** button enabled.  Only editable properties enabled. |
| **Delete** | Only **Delete** button enabled.  All properties disabled. |

you can define a complete setup as below:

public MasterDetailsViewModel()

: base(...)

{

List<string> fixedProperties = new List<string>();

fixedProperties.Add("FixedPropertyA");

fixedProperties.Add("FixedPropertyB");

List<string> nonFixedProperties = new List<string>();

nonFixedKeyProperties.Add("NonFixedPropertyC");

nonFixedKeyProperties.Add("NonFixedPropertyD");

nonFixedKeyProperties.Add("NonFixedPropertyE");

StateManager.AddFixedPropertiesDefaultStates(fixedProperties);

StateManager.AddNonFixedPropertiesDefaultStates(nonFixedProperties);

StateManager.AddButtonDefaultStates();

}

This is essentially just a division of the view properties into editable and non-editable properties, and then applying the default behavior as listed above. The property names used here must match the names used in the **view.xaml** file for binding to the *VisibilityState* and *Enabled* settings.

# Performing property validation

The framework offers some lightweight validation, in the form of validation of updates to property values. The general execution pattern for validation is as follows:

The validation is performed in the domain classes, more specifically in the **set**-part of the relevant properties in the domain class. If a proper **Validator** class has been defined (see later), validation is added by adding one line of code:

public int Year

{

get { return \_year; }

set

{

ValidationHandler.ThrowOnInvalid<int>(Validator.ValidateYear, value);

\_year = value;

}

}

Note the **ValidateYear** parameter – it is a method defined in a domain-specific **Validator** class:

public static ValidationOutcome ValidateYear(int value)

{

return ValidationHandler.ValidateIntInInterval(value, 1900, 2020);

}

The type parameter to **ThrowOnInvalid** must match that of the property being validated (**int** in the example). The **ValidateYear** method uses a method from the **ValidationHandler** class, which contains a number of general-purpose validation methods. It is not mandatory to create a domain-specific **Validator** class, since you can call the methods from **ValidationHandler** directly in a domain class.

If the validation fails, an exception object of type **ValidationException** is thrown. This object should be catched in the domain-specific **DetailsViewModel** class. A typical property will therefore need to be updated as follows:

public int Year // Before

{

get { return DomainObject.Year; }

set

{

DomainObject.Year = value;

OnPropertyChanged();

}

}

public int Year // After

{

get { return DomainObject.Year; }

set

{

var orgValue = DomainObject.Year;

try

{

DomainObject.Year = value;

}

catch (ValidationException e)

{

PresentValidationError(e.Message, () => { Year = orgValue; });

}

OnPropertyChanged();

}

}

The **PresentValidationError** method is defined in the base class, and presents the problem for the application user in a simple dialog. The change is then reverted by the small snippet of code given as parameter. For a different property, simply replace **Year** with the name of the property in question. Again, if validation is not required, no code needs to be added.

# Setting up property dependencies

If the Details view contains so-called **aggregated properties** (properties calculated from the value of other properties), it becomes necessary to keep track of dependencies between such properties.

More specifically: if aggregated property A is calculated from the values of properties P1 and P2, we need to notify all properties binding to A to re-read the value, if P1 or P2 is changed. This would normally be done simply by calling **OnPropertyChanged()**, but since aggregated properties do not have a **set**-part, the properties on which A depends must call **OnPropertyChanged(A)** on behalf of A.

This is handled by setting up such dependencies in the **DetailsViewModel** constructor, by calling **AddDependency** on the **PropertyDependencies** property from the base class.

public DetailsViewModel(Photo obj) : base(obj)

{

PropertyDependencies.AddDependency(nameof(P1),nameof(A));

PropertyDependencies.AddDependency(nameof(P2), nameof(A));

}

The base class also contains an extended version of the well-known **OnPropertyChanged** method, called **OnProperty­Changed­WithDependencies**. To be sure that **OnPropertyChanged** gets called on all dependent properties, simply replace all occurrences of **OnPropertyChanged** with **OnProperty­Changed­WithDependencies**:

public int P1

{

get { return DomainObject.P1; }

set

{

DomainObject.P1= value;

OnPropertyChangedWithDependencies();

}

}

# Accessing domain objects

If the framework is only used for simple CRUD functionality for independent domain objects, there is not as such any need for explicitly accessing the domain objects in the code. However, if a domain object refers to “foreign” domain objects (domain objects of a different class), you may need to know how to access such objects. Also, the framework needs to access the domain-specific **Catalog** classes, in order to set up loading and saving functionality. For this purpose, a bit of code must be added to the class **ObjectProvider** (In **Models**/**App**) for each new domain class:

public static Domain.Car.Catalog CarCatalog

{

get { return Domain.Car.Catalog.Instance; }

}

For each domain class, a class called **Catalog** must also be created, in the folder **Models**/**Domain**/ (*name of your domain class*). The **Catalog** object will contain all the domain objects of that particular class. **Catalog** is a so-called **singleton** class, meaning that only one **Catalog** object (for each domain class) will ever be created. With the above “shorthand” defined in the **ObjectProvider** class, the **Catalog** objet can be accessed like this:

ObjectProvider.CarCatalog;

You can then use various collection-oriented methods on the catalog, like **Insert**, **Delete**, **Read**, etc.. Since all domain objects contain a **Key** property of type **int** (due to inheritance from **DomainClass­Base**), you can choose to refer to a foreign domain object simply by using its key:

public class Sale : DomainClassBase

{

private int \_carKey;

...

public int CarKey

{

get { return \_carKey; }

set { \_carKey = value; }

}

}

Setting up a GUI for CRUD operations on such a class may be a bit more tricky than setting up the standard GUI. A typical setup will probably allow the user to select which foreign object to refer to, by presenting the objects in a collection-type control like a **ListBox** or **ComboBox**. These controls are of course more complex to handle than e.g. **TextBox** controls, since they must be populated with foreign objects (or more specifically, item view model objects corresponding to the foreign domain objects), and you may also need to keep track of the current selection. Code for these purposes should be located in the **DetailsViewModel** class, and could look like the below:

private ObservableCollection<Car.ItemViewModel> \_observableCollectionCars;

...

public ObservableCollection<Car.ItemViewModel> CollectionCars

{

get { return \_observableCollectionCars; }

}

...

public DetailsViewModel(SaleClass obj) : base(obj)

{

\_observableCollectionCars = new ObservableCollection<Car.ItemViewModel>();

...

// Lists are set up in the constructor

foreach (var car in ObjectProvider.CarCatalog.All)

{

\_observableCollectionCars.Add(new Car.ItemViewModel(car));

}

...

}

...

public Car.ItemViewModel SelectedCar

{

get { return GetCarItemViewModel(DomainObject.CarKey); }

set

{

if (value != null)

{

DomainObject.CarKey = value.DomainObject.Key;

}

OnPropertyChanged();

}

}

...

// Helper method to find the selected item in the list

private Car.ItemViewModel GetCarItemViewModel(int key)

{

foreach (var carItemViewModel in \_observableCollectionCars)

{

if (carItemViewModel.DomainObject.Key == key)

{

return carItemViewModel;

}

}

return null;

}

...

With these properties available, you can bind to e.g. a **ComboBox** in the view, like:

<ComboBox

ItemsSource="{Binding DetailsViewModel.CollectionCars}"

SelectedItem="{Binding DetailsViewModel.SelectedCar, Mode=TwoWay}">

<ComboBox.ItemTemplate>

...

</ComboBox.ItemTemplate>

</ComboBox>

Once you let objects refer to foreign objects, you must also be aware of potential problems relating to deletion of objects. If a **Sale** object refers to a **Car** object, you need to consider how to handle the situation where the referred-to **Car** object has been deleted. Keeping track of such dependencies is not a trivial matter.

# Adding a domain class - details

On the next pages follow more specific instructions about what to do, when adding a new domain class to the framework. All files that need to be created or modified are mentioned. The creation/ modification should be done in the same order as specified in the document.

The instructions will typically refer to a template file, which contains more specific instructions for how to create that specific kind of file. In many cases, it is just a matter of copy/paste and some textual substitution.

In the descriptions and the file templates, the placeholder name **\_REPLACEME\_** is used to denote the name of the domain class being added to the project. It is essential that the entire placeholder text – including the underscores at the start and end – are replaced with the exact name of the domain class, when creating or modifying the files in question.

Below follows a list of files you need to create/modify. Details follow on the subsequent pages. Note that some files are optional. In addition to these files, you can add domain-specific **Controller** classes as well.

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Location** | **Action** | **Required** |
| **\_REPLACEME\_**.cs | Models/Domain/**\_REPLACEME\_** | Create | Mandatory |
| Catalog.cs | Models/Domain/**\_REPLACEME\_** | Create | Mandatory |
| HardCodedObjects.cs | Models/Domain/**\_REPLACEME\_** | Create | Optional |
| ObjectProvider.cs | Models/App | Modify | Mandatory |
| Validator.cs | Validators/Domain/**\_REPLACEME\_** | Create | Optional |
| ItemViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Mandatory |
| MasterViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Mandatory |
| DetailsViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Mandatory |
| ViewModelFactory.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Mandatory |
| MasterDetailsViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Mandatory |
| View.xaml | Views/Domain/**\_REPLACEME\_** | Create | Mandatory |
| **\_REPLACEME\_**.jpg | Assets/Domain/**\_REPLACEME\_** | Create | Mandatory |
| MainPage.xaml | Views/App | Modify | Mandatory |
| MainPage.xaml.cs | Views/App | Modify | Mandatory |

|  |  |
| --- | --- |
| **Filename** | **\_REPLACEME\_**.cs |
| **Location** | Models/Domain/**\_REPLACEME\_** |
| **Purpose** | The domain class itself, with all relevant properties |
| **Action** | Create |
| **Template (if create)** | Models/Domain/Template/Template.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class 2. Add properties, methods etc. to your domain class 3. Implement **SetDefaultValues** |
| **Remarks** | * The domain class must inhert from **DomainClassBase** * All non-aggregated properties must contain a **get**-part and a **set**-part. Aggregated properties only need a **get**-part * If you want to validate a value before assigning it to a property; see the above section on property validation |

|  |  |
| --- | --- |
| **Filename** | Catalog.cs |
| **Location** | Models/Domain/**\_REPLACEME\_** |
| **Purpose** | Defines a catalog for the domain. More specifically, the class ties an in-memory collection to an object source. |
| **Action** | Create |
| **Template (if create)** | Models/Domain/Template/Catalog.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | HardCodedObjects.cs |
| **Location** | Models/Domain/**\_REPLACEME\_** |
| **Purpose** | If you want to create domain objects directly in the source code, you should do it in this file. The objects will be available in the application after **Load** has been invoked. |
| **Action** | Create |
| **Template (if create)** | Models/Domain/Template/HardCodedObjects.cs |
| **Required** | Optional |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | ObjectProvider.cs |
| **Location** | Models/Domain/**\_REPLACEME\_** |
| **Purpose** | The static methods in this class provide a shorthand notation for accessing the catalog instances. Also, the application setup uses this class to setup all catalogs for load and save functionality |
| **Action** | Modify |
| **Template (if create)** | Models/App/ObjectProvider.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | Validator.cs |
| **Location** | Validators/Domain/**\_REPLACEME\_** |
| **Purpose** | If you need to define domain-specific validator methods, they should be defined in this class. |
| **Action** | Create |
| **Template (if create)** | Models/App/ObjectProvider.cs |
| **Required** | Optional |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | ItemViewModel.cs |
| **Location** | ViewModels/Domain/**\_REPLACEME\_** |
| **Purpose** | Defines a view-model for showing an item in a collection-oriented GUI control (e.g. a **ListView**) |
| **Action** | Create |
| **Template (if create)** | ViewModels/Domain/Template/ItemViewModel.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class 2. Overide the properties from **ItemViewModelBase** |
| **Remarks** | * Default values (used if properties are not overrided):   + **Description**: Calls **ToString** on enclosed domain object   + **FontSize**: 18   + **ImageSource**: Empty string   + **ImageIsVisible**: true   + **ImageSize**: 80 pixels * You can refer to a property on the enclosed domain object by writing e.g.: **DomainObject.MyProperty** |

|  |  |
| --- | --- |
| **Filename** | MasterViewModel.cs |
| **Location** | ViewModels/Domain/**\_REPLACEME\_** |
| **Purpose** | Defines the Master part of a Master/Details view model |
| **Action** | Create |
| **Template (if create)** | ViewModels/Domain/Template/MasterViewModel.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | DetailsViewModel.cs |
| **Location** | ViewModels/Domain/**\_REPLACEME\_** |
| **Purpose** | Defines the Details part of a Master/Details view model |
| **Action** | Create |
| **Template (if create)** | ViewModels/Domain/Template/DetailsViewModel.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class 2. Implement properties needed in the Details view, following the example in the template |
| **Remarks** | * All view properties corresponding to non-aggregated proper­ties in the domain class must contain a **get**-part and a **set**-part. View properties corresponding to aggregated properties only need a **get**-part * The template file contains an example template for both validated and non-validated properties * If you need to set of dependencies between non-aggregated and aggre­gated properties; see the above section about setting up property dependencies |

|  |  |
| --- | --- |
| **Filename** | ViewModelFactory.cs |
| **Location** | ViewModels/Domain/**\_REPLACEME\_** |
| **Purpose** | Defines factory methods for domain-specific view models |
| **Action** | Create |
| **Template (if create)** | ViewModels/Domain/Template/ViewModelFactory.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | MasterDetailsViewModel.cs |
| **Location** | ViewModels/Domain/**\_REPLACEME\_** |
| **Purpose** | Defines the top level of a Master/Details view model |
| **Action** | Create |
| **Template (if create)** | ViewModels/Domain/Template/MasterDetailsViewModel.cs |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class 2. Set up GUI control states, using the **StateManager** property |
| **Remarks** | * See the above section on setting up state-specific control appearance, or use the example in the template |

|  |  |
| --- | --- |
| **Filename** | View.xaml |
| **Location** | Views/Domain/**\_REPLACEME\_** |
| **Purpose** | Defines a physical view for the domain, based on the corresponding view models. |
| **Action** | Create |
| **Template (if create)** | Views/Domain/Template/View.xaml |
| **Required** | Mandatory |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class 2. For each of your **DetailsViewModel** properties: Implement a TextBlock/TextBox combination as outlined in the template, or implement a different GUI control if needed. |
| **Remarks** | * The view will not function before control states have been defined in **MasterDetailsViewModel** * You are not limited to using TextBlock/TextBox controls in the Details view, but using other controls may require a bit more coding effort in the **DetailsViewModel** class |

|  |  |
| --- | --- |
| **Filename** | **\_REPLACEME\_**.jpg |
| **Location** | Assets/Domain/**\_REPLACEME\_** |
| **Purpose** | Image file for top-level navigation |
| **Action** | Create |
| **Template (if create)** | (none) |
| **Required** | Mandatory |
| **Guide** | The image file named **\_REPLACEME\_**.jpg will be shown on the tab for the domain in the top-level navigation.   1. Copy an image file named **\_REPLACEME\_**.jpg to the specified location |
| **Remarks** | * If you wish to use instance-specific images for a domain class; see the section on adding and using Image objects. |

|  |  |
| --- | --- |
| **Filename** | MainPage.xaml |
| **Location** | Views/App |
| **Purpose** | Defines the application-level main view |
| **Action** | Modify |
| **Template (if create)** | (none) |
| **Required** | Mandatory |
| **Guide** | The file contains a short commented-out code block, corresponding to an entry in the main view splitview pane. In order add an entry for your own domain:   1. Copy-paste the code 2. Replace **\_REPLACEME\_** with the name of your domain class 3. Uncomment the code |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | MainPage.xaml.cs |
| **Location** | Views/App(under mainPage.xaml) |
| **Purpose** | Code-behind for the application-level main view |
| **Action** | Modify |
| **Template (if create)** | (none) |
| **Required** | Mandatory |
| **Guide** | The file contains a short commented-out code block, that handles navigation to the domain-specific view. In order add an entry for your own domain:   1. Copy-paste the code 2. Replace **\_REPLACEME\_** with the name of your domain class 3. Uncomment the code |
| **Remarks** |  |