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](#_Toc481612662)

[Purpose 2](#_Toc481612663)

[Status (May 03 2017) 3](#_Toc481612664)

[GUI Functionality 4](#_Toc481612665)

[Framework structure 7](#_Toc481612666)

[Adding filters to a Catalog class 9](#_Toc481612667)

[Defining state-specific view appearances 11](#_Toc481612668)

[Performing property validation 13](#_Toc481612669)

[Setting up property dependencies 15](#_Toc481612670)

[Accessing domain objects 16](#_Toc481612671)

[Adding a domain class - details 18](#_Toc481612672)

# 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 03 2017)

The **MVVMStarter** project currently supports:

* Handling CRUD operations for **simple domain classes**, defined as domain classes where the type of each property is of a simple type, like **string**, **int**, etc..
* 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.
* 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 still possible to let domain objects refer to other domain objects, using the 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.

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

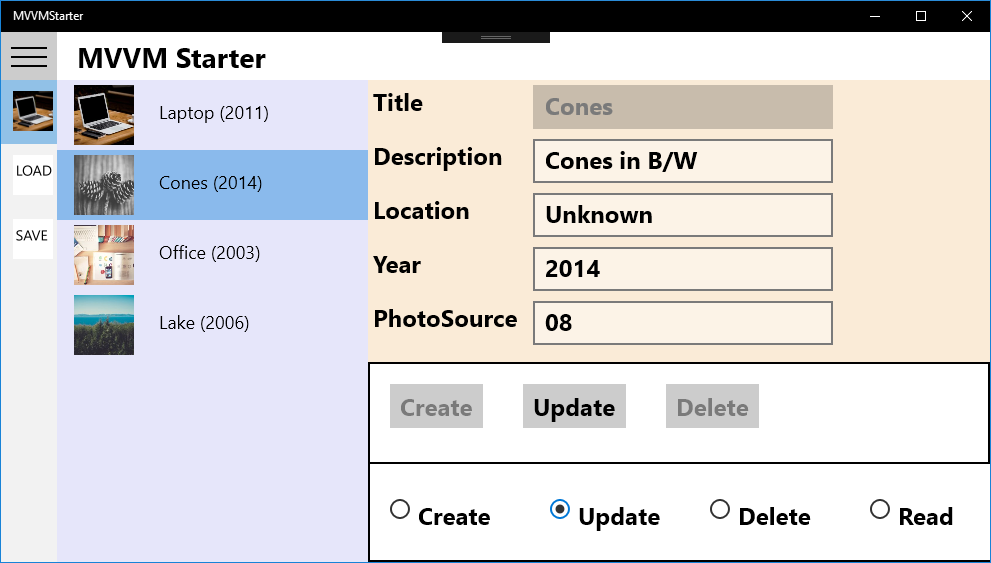
<https://github.com/perl-easj/MVVMStarter>

# GUI Functionality

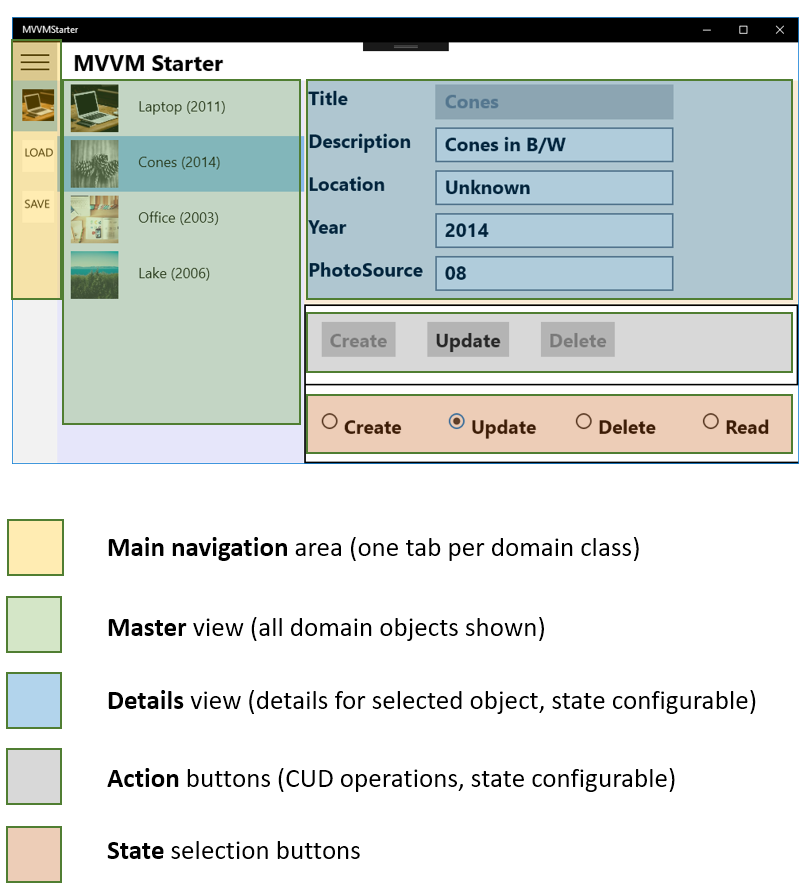
The primary functionality offered by **MVVMStarter** is the ability to show domain objects of a particu­lar class in a so-called **Master/Details** view. The main characteristics of a Master/Details view are:

* 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 (**Photo** 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 **Load** and **Save** 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.

**Action** 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.

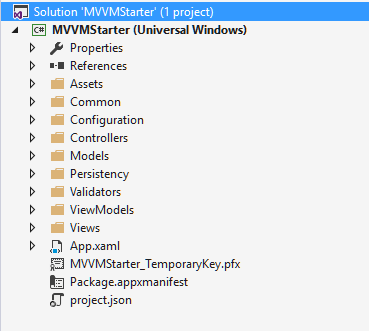
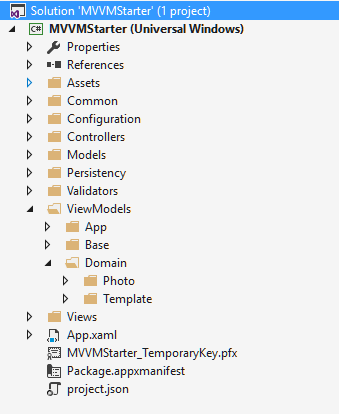
**State selection** buttons: The view is defined to always be in one of four specific states, corresponding to the four CRUD operations. Since all controls can be configured to be visible/collapsed and enabled/ disabled depending on the selected state, the visual appearance of the view can change from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

**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"));

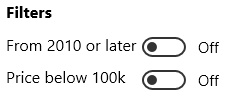
The first parameter is the identifier used for control appearance bindings in the view, and the second parameter is the lead text you can retrieve for use in the view.

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.

# 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 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(nameof(CarClass.LicensePlate));

fixedProperties.Add(nameof(CarClass.Year));

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

nonFixedProperties.Add(nameof(CarClass.Description));

nonFixedProperties.Add(nameof(CarClass.Price));

nonFixedProperties.Add(nameof(CarClass.PhotoSource));

StateManager.AddFixedPropertiesDefaultStates(fixedProperties); StateManager.AddNonFixedPropertiesDefaultStates(nonFixedProperties);

StateManager.AddButtonDefaultStates();

}

This is essentially just a division of the properties into editable and non-editable properties, and then applying the default behavior as listed above.

# 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.

For each domain class, a class called **Catalog** must 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. If the domain class is e.g. called **Car**, the corresponding **Catalog** object can be accessed in this way:

var carCatalog = Models.Domain.Car.Catalog.Instance;

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 Car : DomainClassBase

{

private int \_photoKey;

...

public Models.Domain.Photo.Photo Picture

{

get { return Models.Domain.Photo.Catalog.Instance.Read(\_photoKey); }

}

public int PhotoKey

{

get { return \_photoKey; }

set { \_photoKey = 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, 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 (we assume **\_photoCatalog** to refer to the catalog of **Photo** objects):

public ObservableCollection<PhotoClass> PictureCollection

{

get

{

ObservableCollection<PhotoClass> collection =

new ObservableCollection<PhotoClass>();

\_photoCatalog.All.ForEach(collection.Add);

return collection;

}

}

public PhotoClass PictureSelected

{

get { return \_photoCatalog.Read(DomainObject.PhotoKey); }

set

{

if (value != null)

{

DomainObject.PhotoKey = value.Key;

}

OnPropertyChanged();

}

}

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

<ComboBox

ItemsSource="{Binding DetailsViewModel.PictureCollection}"

SelectedItem="{Binding DetailsViewModel.PictureSelected, 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 **Car** object refers to a **Photo** object, you need to consider how to handle the situation where the referred-to **Photo** 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 all mandatory files to create and modify. Details follow on the subsequent pages. In addition to these files, you can optionally choose to add domain-specific **Validator** classes, domain-specific **Controller** classes, etc..

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Location** | **Action** | **Complexity** |
| **\_REPLACEME\_**.cs | Models/Domain/**\_REPLACEME\_** | Create | Moderate |
| Catalog.cs | Models/Domain/**\_REPLACEME\_** | Create | Low |
| ItemViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Low |
| MasterViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Low |
| DetailsViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Moderate |
| ViewModelFactory.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Low |
| MasterDetailsViewModel.cs | ViewModels/Domain/**\_REPLACEME\_** | Create | Moderate |
| View.xaml | Views/Domain/**\_REPLACEME\_** | Create | Moderate |
| (image files) | Assets/Domain/**\_REPLACEME\_** | Create | Low |
| AppConfig.cs | Configuration/App | Modify | Low |
| MainPage.xaml | Views/App | Modify | Low |
| MainPage.xaml.cs | Views/App | Modify | Low |

|  |  |
| --- | --- |
| **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 |
| **Complexity** | Moderate (depends on domain class complexity) |
| **Guide** | Follow the instructions in the template file.  Summary:   1. Replace **\_REPLACEME\_** with the name of your domain class 2. 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 * You should not declare any constructors in the domain class * 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, it ties an in-memory collection to a persistent source. |
| **Action** | Create |
| **Template (if create)** | Models/Domain/Template/Catalog.cs |
| **Complexity** | Low (a few textual replacements) |
| **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 |
| **Action** | Create |
| **Template (if create)** | ViewModels/Domain/Template/ItemViewModel.cs |
| **Complexity** | Low (optionally override a few virtual methods) |
| **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.ImageSource** |

|  |  |
| --- | --- |
| **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 |
| **Complexity** | Low (a few textual replacements) |
| **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 |
| **Complexity** | Moderate (depends on domain class complexity) |
| **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 |
| **Complexity** | Low (a few textual replacements) |
| **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 |
| **Complexity** | Moderate: a few textual replacements, plus setting up control states |
| **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 |
| **Complexity** | Moderate (depends on domain class complexity) |
| **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 |
| **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 (+ instance-specific image files) |
| **Location** | Assets/Domain/**\_REPLACEME\_** |
| **Purpose** | Image files for top-level navigation and instance-specific images |
| **Action** | Create |
| **Template (if create)** | (none) |
| **Complexity** | Low (copy-paste images from source + rename a single image file) |
| **Guide** | If you need instance-specific images for the domain data, you should copy these images to the specified location. Also, one image file must be called **\_REPLACEME\_**.jpg. This image will be shown on the tab for the domain in the top-level navigation.   1. Copy image files to the specified location 2. Include an image file named **\_REPLACEME\_**.jpg |
| **Remarks** | * If you wish to use instance-specific images from another file location or from the web, you are free to do so. * The string constants **ImageFilePrefix** and **ImageFilePostfix** from **AppConfig.cs** can be used for easier management of image source strings. |

|  |  |
| --- | --- |
| **Filename** | AppConfig.cs |
| **Location** | Configuration/App |
| **Purpose** | Ensures all catalogs get loaded and saved |
| **Action** | Modify |
| **Template (if create)** | (none) |
| **Complexity** | Low (add two lines by following copy-paste examples in code) |
| **Guide** | The file contains two lines of code that have been commented out. Both lines of code contain the **\_REPLACEME\_** placeholder  For each line of code:   1. Copy-paste the code 2. Replace **\_REPLACEME\_** with the name of your domain class 3. Uncomment the code |
| **Remarks** |  |

|  |  |
| --- | --- |
| **Filename** | MainPage.xaml |
| **Location** | Views/App |
| **Purpose** | Defines the application-level main view |
| **Action** | Modify |
| **Template (if create)** | (none) |
| **Complexity** | Low (copy-paste and textual replacement in a short code block) |
| **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) |
| **Complexity** | Low (a few textual replacements) |
| **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** |  |