

Application Programmers Guide

By Per Laursen

# Preface

This document is intended for students (or others) wishing to use the MVVMStarter class libraries as a foundation for a software application. The libraries support vari­ous aspects of implementing an application according to the **MVVM** (**M**odel **V**iew **V**iew**M**odel) architecture. The libraries are written in C#. The source code for all libraries is available on GitHub: <https://github.com/perl-easj/MVVMStarterLibrary>

The libraries are primarily aimed at being a toolkit for students developing an appli­cation on the **U**niversal **W**indows **P**latform (**UWP**). Currently, the libraries are only compatible with UWP applications. The libraries can be used as-is, or as inspiration for your own implementation.

The libraries are – and will probably always be – a work in progress, and changes to the libraries may thus occur without notice. Also, the libraries are not subjected to very rigorous testing, and may thus contain errors.

In the subsequent parts of the document, we will assume that the reader has some familiarity with these topics:

* Object-Oriented programming
* C# programming
* MVVM architecture
* UWP
* XAML and Data Binding

The reader does not need to be an expert in any of these topics, but we will not make any effort to explain the topics further here. As always, you can find ample material on any of these topics online.

Table of Content

[Preface 1](#_Toc500176792)

[Introduction 3](#_Toc500176793)

[Scope 3](#_Toc500176794)

[Principles 4](#_Toc500176795)

[Overall structure 5](#_Toc500176796)

[Main library packages 8](#_Toc500176797)

[InMemoryData 9](#_Toc500176798)

[TransformedData 9](#_Toc500176799)

[Model 10](#_Toc500176800)

[PersistentData 11](#_Toc500176801)

[DataSources 11](#_Toc500176802)

[ViewModel 12](#_Toc500176803)

[CommandControl 13](#_Toc500176804)

[Services 14](#_Toc500176805)

[Extensions 15](#_Toc500176806)

[ExtensionsCommands 15](#_Toc500176807)

[ExtensionsModel 16](#_Toc500176808)

[ExtensionsServices 16](#_Toc500176809)

[ExtensionsViewModel 17](#_Toc500176810)

[Using the libraries in a client project 18](#_Toc500176811)

# Introduction

The motivation for creating these libraries originally came from a predicament relating to teaching in Software Development. Just a few months into the study, students were expected to create a (fairly simple) software application as part of their final project. The application should be written in C# using Visual Studio, and should adhere to the MVVM architecture. I found this to be a quite ambitious goal for a first-semester project, and therefore wanted to create an MVVM “toolbox” for the students, which could aid in developing the parts for the application relating to basic **CRUD** (**C**reate, **R**ead, **U**pdate and **D**elete) operations. The intention was that if CRUD functionality could be implemented fairly quickly by using the library, the stu­dents could spend more time on implementing domain-specific business logic.

## Scope

As specified above, the libraries are targeted at applications which are

* Written in C# (and XAML)
* Targeting the Universal Windows Platform (UWP)
* Adhering to the MVVM architecture

The libraries are mainly targeted at the ViewModel and Model layers, so there is very little support in the libraries with regards to GUI development as such (on the UWP platform, GUI is typically specified in XAML).

Even though the libraries do not make any detailed assumptions about the structure of the GUI, it is to some extent assumed that CRUD operations are done in views using the Master/Details view layout. This does not mean that it is impossible to use the libraries with different layouts, but you may need to do a bit more work yourself to make ends meet.

## Principles

The MVVMStarter libraries are currently available as a single Visual Studio solution, which contains all libraries as individual projects. The intention is to deploy the libra­ries as one or more NuGet packages, but that point has not been reached yet. For now, each library is built as an individual .dll (Dynamic Link Library), which a client project can then reference.

The overall principle for designing the libraries has been that “smaller is better”. Classes are kept small and focused (preferably with just a single responsibility), and each library is also kept small and focused. Interfaces and implementations are usu­ally kept in separate libraries, with the consequence that a significant part of the libraries only contain interface definitions, and are therefore quite small. The speci­fic number of individual libraries will probably change over time, but currently (late 2017) hovers around 30 libraries.

The motivation for this design strategy is simply to follow some well-known and pro­ven design strategies for Object-Oriented programming. Some of these are:

* Program towards interfaces instead of implementations
* Keep classes and interfaces small and focused (single responsibility)
* Make classes open for extension, but closed for modification
* Isolate elements that vary
* Use Design Patterns where appropriate

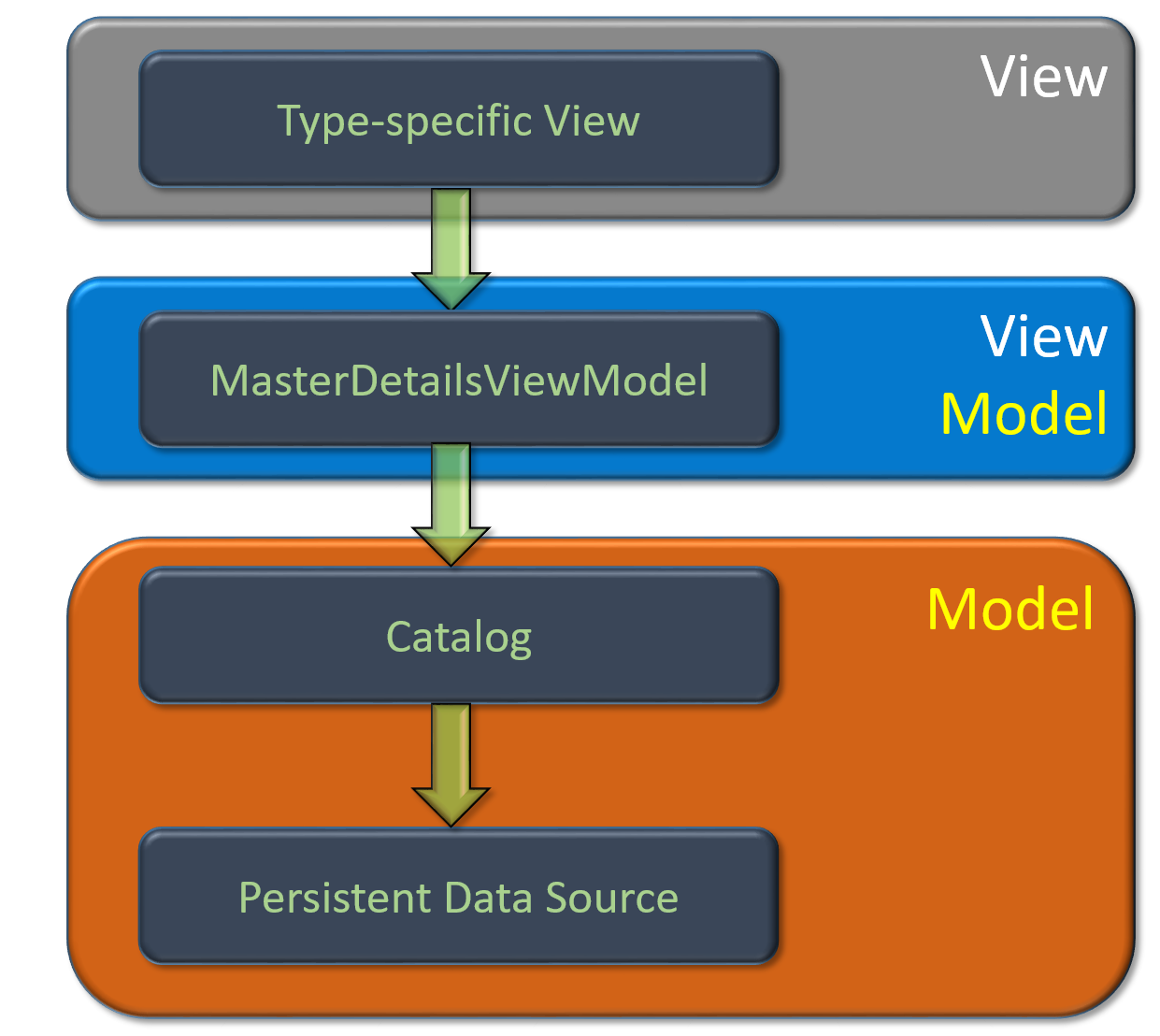
Many of these principles are also encompassed by the term SOLID, even though we do not claim that the libraries follow these principles completely.

A significant dilemma when designing such a system of class libraries – in particular when the aim is to provide relatively novice students with a toolbox – is the object-oriented language elements used in the source code. Generics (the ability to use a type as a parameter) is the primary example of this dilemma. Since a lot of the code needed for CRUD operations is type-independent, it is almost impossible not to use Generics when writing such code. Still, code relying on Generics is inevitably more “abstract” that non-generic code, and does make the learning curve for the students more steep. Generics is used quite heavily in the libraries, so some level of introduc­tion to Generics is mandatory for being able to use the library.

## Overall structure

The main focus area for the libraries are the Model and ViewModel layers in the MVVM architecture. In each of these layers, one kind of class is at the center stage:

* **MasterDetailsViewModel**: In the ViewModel layer, the MasterDetailsView­Model will act as a data context for a (type-specific) View, and thus contain properties to which the View can bind properties of GUI controls. The Master­DetailsViewModel will “harvest” the data entered into the View by the end user, and relay the data to the Model layer
* **Catalog**: In the Model layer, the Catalog maintains a collection of domain objects (typically objects of the same type). The Catalog provides a boundary between the ViewModel and Model layer, and will also be in contact with persistent data sources, where domain data can be stored permanently.



A large part of the class libraries is thus concerned with handling the flow of data from the MasterDetailsViewModel to the Catalog, and from the Catalog to a persis­tent data source. In general terms, we can say that domain data will be represented in three distinct parts of the application:

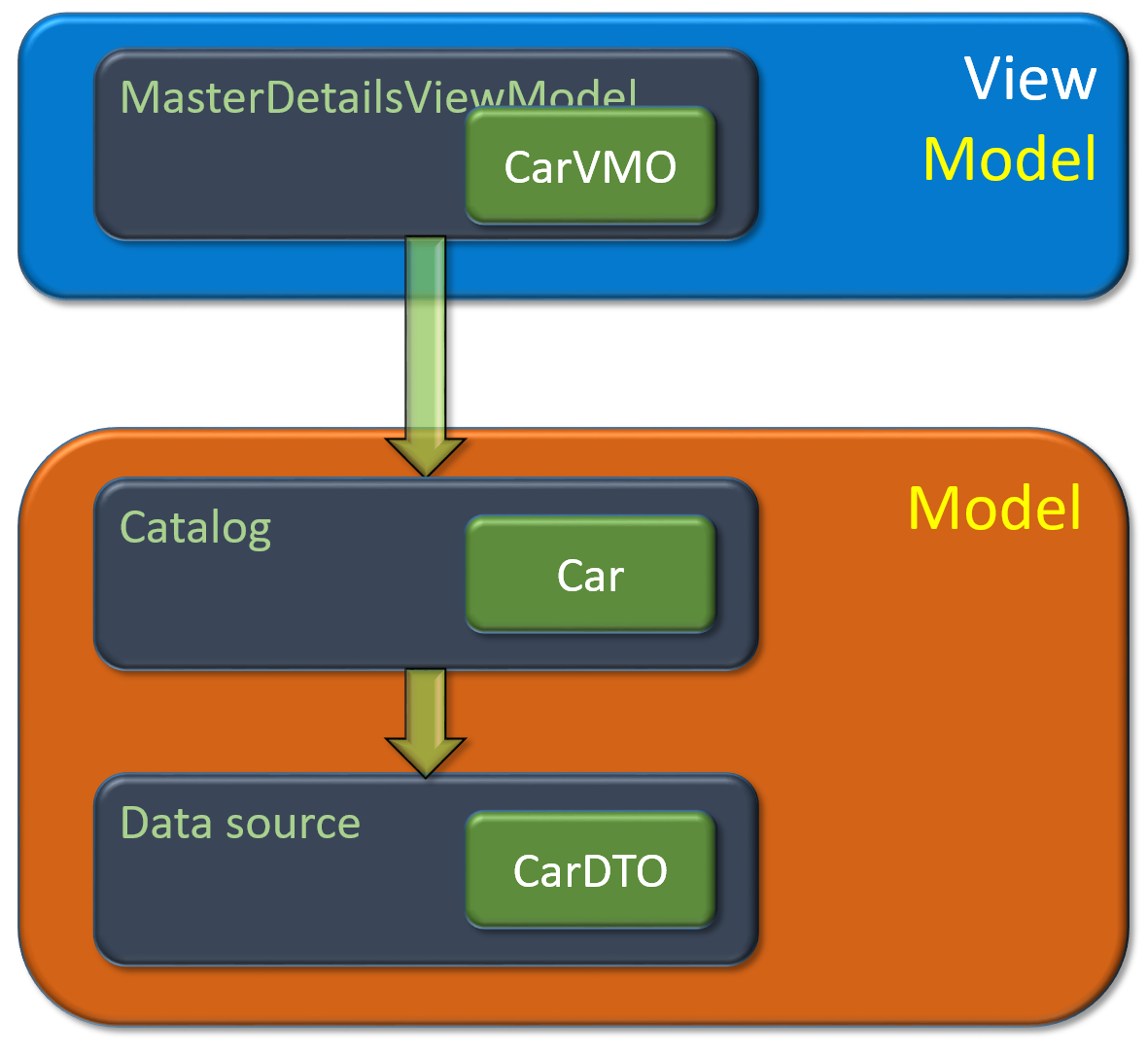
* In the ViewModel layer, managed by the MasterDetailsViewModel
* In the Catalog (being part of the Model layer)
* In the persistent data source (being part of the Model layer)

We stated earlier that the Catalog will maintain a collection of domain data, mea­ning that if e.g. a domain class **Car** had been defined, the Catalog will maintain a collection of **Car** objects. A quite important issue is then: should domain objects (e.g. of type **Car**) also be allowed to exist outside the Catalog? This is actually a somewhat deep question, since the answer depends on what “school” of Object-Oriented de­sign you subscribe to. The libraries do not make any assumptions about what school of design you subscribe to, but it does allow you to define different representations of domain data, according to the three parts of the application mentioned above.

What does that imply more specifically? Suppose that you do insist on confining domain objects to the internals of the Catalog. This implies that it is only the Catalog which may create actual **Car** objects. You then need some other means of trans­porting the data entered by the user – e.g. when the user wants to create a new car entry through a car-specific view – from the MasterDetailsViewModel to the Cata­log. This data can be considered the “raw material” for creation of a car object. The solution adopted here is to allow the application programmer to create a new class – it could e.g. be called **CarViewModel** – which will act as a device for carrying the “raw material” data from the ViewModel layer to the Catalog, which will then con­struct the actual **Car** object from the data. Note that it is not mandatory to cre­ate such an extra class, but the libraries allow you to do it.

Similar considerations apply to the transportation of data from the Catalog to the persistent data source. Again, you may – but are not required to – create a new class for this specific purpose; such classes are often calkled “data transport classes”, and we will use that terminology as well.

Still using the **Car** example, the three different representations will then fit into the architecture like this:

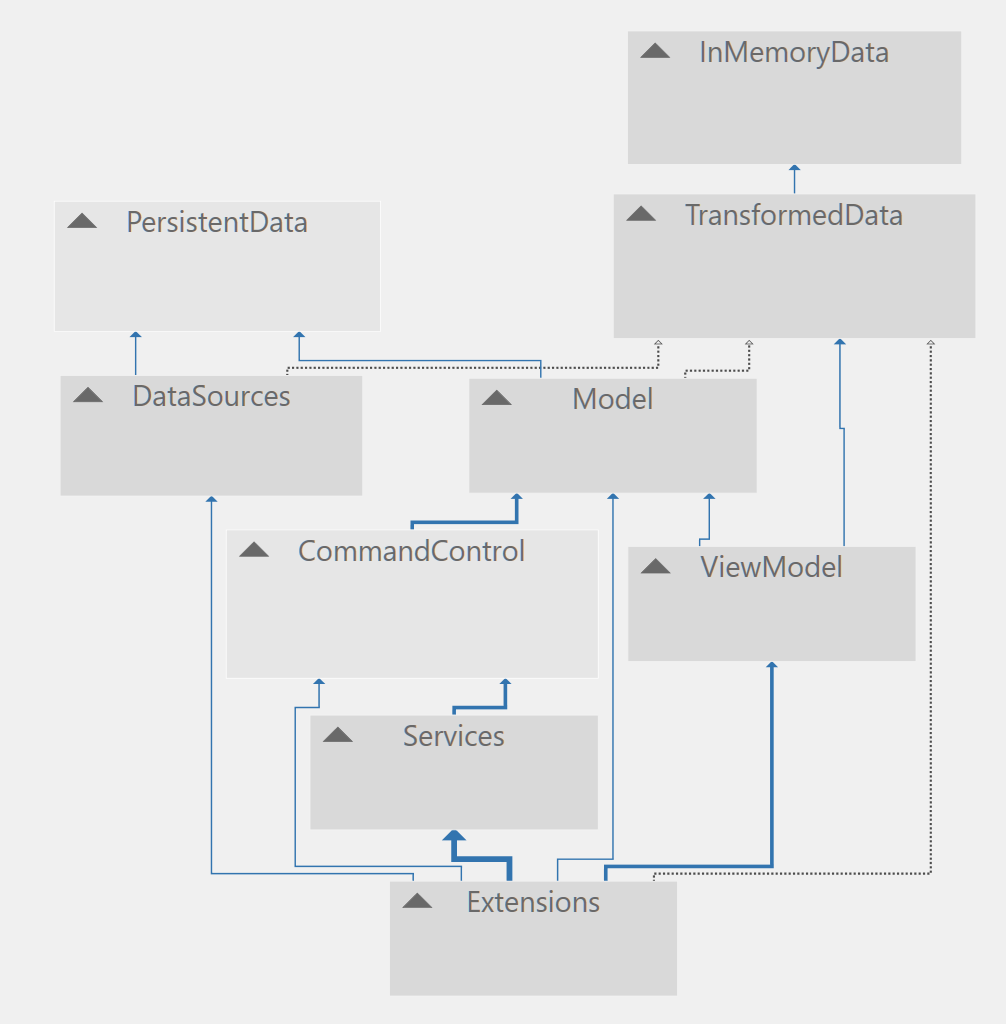


We have introduced a “shorthand” for the two additional classes: **CarVMO** for “Car View Model object”, and **CarDTO** for “Car Data Transfer object”. Again, the libraries do not require you to actually define these two additional classes, but it allows you to do so.

If we have a scenario where domain data may be available in three different forms – as per the classes above – we also need to consider how data can be “transformed” from one representation to another. We need to consider two pairs of trans­for­­ma­tions which might be needed: **Car** to **CarVMO** (and vice versa), and **Car** to **CarDTO** (and vice versa). If the Catalog is the only entity which has knowledge about domain classes, it follows that such transformations should also be performed inside the Catalog. This is also the solution adopted in the library; specific details are described later in this document.

# Main library packages

The entire collection of libraries does as mentioned contain about 30 libraries. The libraries are organised into a number of “packages” (which are essentially just fold­ers in the Visual Studio solution containing the libraries), which are descibed briefly below. We have strived to achieve a package organisation where there are no mutu­al depen­dencies between packages. The overall package dependency is as illustrated below:



Some of the packages contain classes that are mainly used inside the library itself, while others contain classes that are intended to become base classes for classes in a client project (i.e. a project which uses the MVVMStarter class libraries). In the detailed descriptions below, any passages describing how a client project should use the classes in the libraries are written in red text.

## InMemoryData

This package contains two very small – but important – interfaces **IStorable** and **ICopyable**. In order to be able to manage a collection of domain objects, it is required that such domain objects have a unique key property; this is exactly what the **IStorable** interface defines: a property called **Key** of type **int**. Furthermore, it may also be required that an object is “copyable”, which is defined by the **ICopyable** interface with the single method **Copy**.

The package also contains two straightforward implementations of the two inter­faces, named **StorableBase** and **CopyableBase**. **StorableBase** and **CopyableBase** should be used as base classes for domain classes in client projects.

Finally, the package contains the interface **IInMemoryCollection**, with the corre­sponding implementation **InMemoryCollection**. This defines a collection class which is intended for storing a collection of domain objects of a single type. A client project will however usually not need to inherit from this class, since it is primarily used in­ter­nally by the Catalog classes in the libraries (see later).

## TransformedData

We will often need to be able to “transform” a domain object to another represen­tation, e.g. the VMO and DTO classes described above. This package contains inter­faces related to such transformations. The interfaces **IDTOTransform** and **IVMO­Trans­form** defines methods for such transformations; these methods will usually be implemented by Catalog classes. A VMO class may also need to define default values for its property, which is suported by the **IDefaultValues** interfaces, with the corre­sponding implementation **CopyableWithDefaultValuesBase**. If a client project con­tains classes for transformed data types, **CopyableWithDefaultValuesBase** should be be used as a base classes for these classes.

Some of the classes used in the ViewModel layer can be considered to be “data wrap­pers”, i.e. the will contain a reference to an underlying data object, which is usually of a transform type. The interface **IDataWrapper** and implementation **DataWrapper** support this concept.

## Model

The model package is one of the central packages, since it contains classes that define the Catalog concept. The interface **ICatalog** defines catalog functionality in terms of the fundamental CRUD operations, plus a property for obtaining all objects stored in the catalog. Note that the interface methods are intended to work with transformed objects rather than domain objects, since creation of domain objects is performed inside the catalog itself.

The class **Catalog** contains an implementation of the **ICatalog** interface. A couple of features are worth taking note of. First, the **Create** method contains a parameter called **keyManagement**, of type **KeyManagementStrategyType**. We mentioned before that a prerequisite for being a “storable” object is that the object implements the **IStorable** interface, which contains a **Key** property. Each stored object should then have a unique key. But who sets the value of this key, and when? The **InMemo­ry­Collection** class – which is typically used internally in the catalog classes – contains functionality for finding a unique key for a new domain object. If this strategy is to be used, the caller does not need to specify a value for **keyManagement**, since it defaults to **CollectionDecides**. However, two more strategies are available:

* **CallerDecides**: This simply leaves the choice of key to the caller, which also implies that the catalog does not check if the provided key in already in use.
* **DataSourceDecides**: Some data sources – typically relational databases – can choose or generate a new unique key for the new domain object. By choosing this option, the catalog will first try to insert the new object into the data source. This operation – if successful – will return a new key value, which then becomes the key for the object in the catalog as well.

Second, the Catalog class contains four abstract methods, which are originally defined in the interfaces **IDTOTransform** and **IVMOTransform**:

* **CreateDomainObjectFromDTO**
* **CreateDomainObjectFromVMO**
* **CreateDTO**
* **CreateVMO**

These four methods are used when there is a need to transform back and forth between a domain objects and its transformed representations. Since the specific transformations of course depends on the individual domain class, the methods must be implemented by the domain-specific catalog classes (see later).

The package also contains the interface **IPersistableCatalog** and corresponding implementation **PersistableCatalog**. These add Load- and Save-functionality to the **Catalog** implementation, defined as:

* **Load**: Delete all objects currently stored in the catalog, and replace them with objects read from the data source.
* **Save**: Replace all objects in the data source with the objects currently stored in the catalog.

Load and Save are thus not incremental.

Even though this package is one of the central packages of the libraries, a client project will usually not inherit from one of the classes in this package. One of the packages described later contains more specialised catalog classes, which are inten­ded to act as base classes in client projects.

## PersistentData

This package contains various interfaces and classes relating to general management of persistent data sources. The package does not contain implementations for any specific data source types. The content of the package is mainly for internal use in the libraries, and a client project will usually not inherit from any class or interface in the package.

## DataSources

This package contains implementations of access to specific data sources, currently a file-based data source and web service-based data source. These implementations will typically be used as parameters to catalog classes. A client project will usually not need to use these classes directly, or inherit from them.

## ViewModel

This is also one of the central packages, since it contains the base class implementa­tion of the Master/Details view model concept. The base implementation is found in the class **MasterDetailsViewModelBase**. At this level, very little is assumed about the Master/Details view model, essentially only that the view model contains three properties central for data binding:

* **ItemCollection**: Returns a collection of objects – typically objects which are “data wrappers” around a transformed data object – to which a collection-oriented view control can bind the **ItemsSource** property.
* **ItemSelected**: Keeps track of the item currently “selected” in the collection-oriented view control. The control typically bind its **SelectedItem** property to this property.
* **ItemDetails**: Returns the item for which details should be shown. This will usually be the same item as the selected item. Controls in the Details part of the view can bind its controls to properties on this property.

The specific strategy for interaction between the elements is however isolated in a “mediator” class, which defines the details about what should happen when e.g. the select­ion changes. The base implementation of the mediator concept is found in **MasterDetailsViewModelMediatorBase**.

The package also contains interfaces and classes acting as base class definitions for item view model classes and details view model classes. It is not a requirement that a client project uses these interfaces and classes, since the only requirement for item and details view model classes is that they implement the **IDataWrapper** inter­face. However, a client project must implement an item view model class and details view model class for each domain class. Both implementation must inherit from IDataWrapper, or from a class which itself inherits from IDataWrapper.

The package contains the interface **IViewModelFactory** as well, along with the imple­mentation **ViewModelFactoryBase**. This class is responsible for creating domain-specific instan­ces of item view model and details view model classes, often based on a given domain view model object (VMO). The base class implementation contains two abstract methods **CreateDetailsViewModel** and **CreateItemView­Model**, which must be implemented in domain-specific derived classes, since the details of how to create objects of these two classes are of course domain-specific. A client project will need to implement such derived factory classes, one for each do­ma­in class.

## CommandControl

The libraries follow this convention concerning controllers and commands:

* **Controller**: A sequence of operations corresponding to a single, well-defined piece of business logic. The sequence can involve elements from both the view model layer and the model layer.
* **Command**: A class which implements the **ICommand** interface. A Command object will typically be linked to a view, through a property of type **ICommand** on the class which is the data context for view. A GUI control with a **Com­mand** property (e.g. a **Button**) can then bind to this property on the data context. When the control is activated, the **Execute** method on the bound-to Command object will then be invoked. A Command object will typically invoke a Controller, and can thus be considered a kind of “wrapper” around a Con­trol­ler object.

This package contains some high-level interfaces and classes for both controllers and commands, in the sub-packages **Commands** and **Controllers**. It also contains classes specifically targeted at CRUD operations, in the sub-packages **DataCom­mands** and **DataControllers**. For the controller part, three classes **…ControllerBase** implement the Create, Update and Delete operations, which are then “wrapped” by three corresponding Controller classes.

The package also defines a command “manager” concept. The idea is that a com­mand manager stores a set of references to Command objects, identified by a string identifier. A command manager has a property **Commands**, which is a Dictionary between identifiers and Command object references. A command manager can then be associated with a MasterDetailsViewModel object, which can then expose a simi­lar Dictionary property. Bindings between view controls and the dictionary can now be performed by indexing into the dictionary directly in the XAML code. In this way, the MasterDetailsViewModel object does not have explicit knowledge about the commands it exposes through the associated command manager.

If a client project only needs to perform CRUD operations on domain data, it does not need to use the classes in this package explicitly, since some of the more specia­lised MasterDetailsViewModel classes come with CRUD commands available (see later). However, if a client project needs to make additional functionality invokable from the GUI, it should do this by adding Command objects to an existing command manager in one of the derived MasterDetailsViewModel classes, or by adding a new command manager – containing the new Command objects – to the relevant, do­main-specific MasterDetailsViewModel classes. This is also feasible, since you can include any number of command managers with a MasterDetailsViewModel class.

## Services

The Services package has a different nature that the other packages, since it consists of various functionalities that are not as such mandatory for an application, but can be added if such functionality is needed. The package currently consists of:

**ControlState**: Management of state of GUI controls. The state of certain GUI controls w.r.t. e.g. visibility may depend on certain conditions. Some aspects of this can be managed by the classes in this library.

**Filtering**: Enables specification of sets of “filters”, which essentially are methods taking a domain object as a parameter, and returning true/false. A set of filters can then e.g. be applied to a collection of domain objects.

**Images**: Can handle a set of “in-memory” images, including e.g. images used as icons in top-level application navigation.

**PropertyDependency**: If a view model contains aggregated properties, it can easily beco­me complex to mange dependencies between properties. Classes in this library enables specification of property dependencies as “sources” and “sinks”.

**Security**: Can be used for setting up user groups, and specifying access right for users in specific user groups. This information can then be utilised to make access to certain functionality dependent on access rights.

**UI**: Contains a simple set of methods for displaying information to a user through a pop-up message box, where certain user actions can be defined.

**Validation**: Contains a simplistic framework for performing input value validation, and reporting validation errors by use of exceptions. The library does not assume any details about where in an application the validation is performed.

**ViewState**: If you wish to be able to perform all CRUD operations from a single view, it may be useful to be able to set the view in a certain “state”, when a specific ope­ra­tion is to be performed. This could e.g. be to enable a certain set of control, which corresponds to a specific state. Classes in this library support this idea.

At the time of writing, there are significant differences in how well the libraries in this package have been tested, so proceed with some caution. The **ControlState** and **ViewState** libraries are used by some of the classes in the **Extensions** library (see below), and are therefore fairly well tested.

## Extensions

The intention with the **Extensions** package is to provide a number of classes, which are more specialised versions of classes from the previous libraries. These classes can then serve as base classes for classes in a client application. In many cases, the specialisation is either about making a class CRUD-specific, or making the class de­pend on a specific type of data source. The extensions package is divded into four sub-packages:

* **ExtensionsCommands**
* **ExtensionsModel**
* **ExtensionsServices**
* **ExtensionsViewModel**

### ExtensionsCommands

This is a fairly small package, which defines a Command manager specialisation containing commands for switching between four predefined view states, corre­sponding to the four CRUD operations. It also contains a CRUD command manager specialisation, where the criterion for being able to execute a command now also depends on the state of the view model object.

### ExtensionsModel

This package contains three classes derived from the **Catalog**/**PersistableCatalog** classes from the **Model** package:

**InMemoryCatalog**: This is a very simplistic catalog class, which does not have any connection to a data source, and does not use any data transformations. It is mainly included to provide a simple catalog implementation for test purposes, which can then later be replaced with a more useful implementation.

**FilePersistableCatalog**: This class implements a catalog which uses a text file as data source. This data source only supports Load and Save operations, meaning that if e.g. the **Create** method is called, the created object is not saved to the data source be­fore an explicit call of **Save** is made.

**WebAPIPersistableCatalog**: This class implements a catalog which uses a RESTful web service – usually with an underlying database – as data source. This data source sup­ports all operations except Save, meaning that if e.g. the **Create** method is called, the created object is immediately saved to the data source as well.

A client project will typically implement a catalog class for each domain class; such catalog classes can inherit from **FilePersistableCatalog** or **WebAPIPersistable­Cata­log**, depending on the type of data persistency the client project wishes to use. Each catalog implementation will need to implement four methods for data transforma­tion. Depending on the specific project setup for data transformation, some of these methods may be trivial to implement.

### ExtensionsServices

This is also a small package, which just contains sample view model classes for an Image service and a Security service. It also contains a singleton class **ServiceProvi­der**, which provides a single point-of-access to application services.

### ExtensionsViewModel

The main feature of this package is two specialised classes **MasterDetailsView­Model­WithState** and **MasterDetailsViewModelCRUD** (the latter inherits from the former), which turn the general **MasterDetailsViewModel** class into a class which has functionality specific for CRUD operations. More specifically, the view model is extended with “view states” (one state for each CRUD operation), and with two commands managers: one for commands for switching between view states, and one for invoking commands for CRUD operations.

A client project will typically implement (at least) one **MasterDetailsView­Model** class for each domain class, the specific number will depend on the number of views needed for each domain class. A number of options for how to use the classes in this package exist:

* If a view with CRUD functionality only is needed, the MasterDetailsViewModel class should inherit from **MasterDetailsViewModelCRUD**. Commands for CRUD operations are then directly available.
* If a view with CRUD functionality and additional functionality is needed, the MasterDetailsViewModel class should inherit from **MasterDetailsViewModelCRUD**. Commands for CRUD operations are then directly available, and commands for additional functionality can then be added either as an extra command manager, or added to the existing CRUD command manager.
* If a view with functionality different from CRUD is needed, the MasterDetailsViewModel class should inherit from **MasterDetailsViewModelBase** or **MasterDetailsViewModelWithState**. Commands for invoking the functionality can then be added as an extra command manager, or added individually to the class.

# Using the libraries in a client project

---not ready yet.