

Simulation software for safe, sustainable and future deltas

DELFT3D FM SUITE

Deltares systems

Delta Shell

Delta Shell

Technical Reference Manual

Released for:
Delft3D FM Suite 2023
D-HYDRO Suite 2023
SOBEK Suite 3.7

Version: 0.2
Revision: 78359

25 April 2024

Delta Shell, Technical Reference Manual

Published and printed by:

Deltares
Boussinesqweg 1
2629 HV Delft
P.O. 177
2600 MH Delft
The Netherlands

telephone: +31 88 335 82 73

e-mail: [Information](#)

www: [Deltares](#)

For sales contact:

telephone: +31 88 335 81 88

e-mail: [Sales](#)

www: [Sales & Support](#)

For support contact:

telephone: +31 88 335 81 00

e-mail: [Support](#)

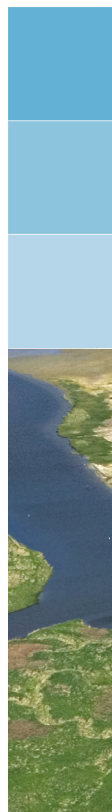
www: [Sales & Support](#)

Copyright © 2024 Deltares

All rights reserved. No part of this document may be reproduced in any form by print, photo print, photo copy, microfilm or any other means, without written permission from the publisher: Deltares.

Contents

List of Figures	vii
List of To Do's	ix
1 Introduction	1
1.1 Overview of the directory structure	1
1.2 Plugins	1
2 The Delta Shell application	5
2.1 Introduction	5
2.2 Project and project items	6
2.3 Activities and models	6
2.4 Hydro models and regions	8
2.5 Data items and model coupling	9
2.6 Functions and spatial data	9
2.6.1 Functions	11
2.6.2 Coverages	11
2.6.3 Function filters	11
2.6.4 Evaluating a function	12
2.6.5 Function store	12
2.7 Model validation	12
3 The Delta Shell GUI	15
3.1 Overview	15
3.2 The ribbon	16
3.3 Docking	16
3.4 Windows and views	17
3.5 The GUI Context Manager	19
3.6 Treeviews	19
3.7 Project explorer	20
3.8 Property grid	20
3.8.1 Static declaration of properties	21
3.8.2 Dynamic declaration of properties	22
3.8.3 Property categories and read-only properties	22
3.9 Time navigator	23
4 Eventing	25
4.1 PostSharp in Delta Shell	25
4.2 Undo/redo	27
4.3 Accessing the GUI thread from a worker thread	29
5 Persistence	33
5.1 History of persistence in Delta Shell	33
5.2 Persistence with nHibernate	34
5.2.1 Overall set-up	34
5.2.2 Mapping files	36
5.2.3 How to enable or disable lazy loading	36
5.2.4 How to solve: sqlite - max 64 tables in join	36
5.2.5 How to use cascades	37
5.2.6 How to create a legacy mapping for a class where one property has changed	39
5.2.7 How to create a legacy mapping for a class that has been split up	40
5.2.8 How to create a legacy mapping for a class that has been merged	42



5.2.9	How to create a legacy mapping for a class that has been changed to a hierarchy of classes	43
5.2.10	How to deal with class renames or changes of namespace	45
5.3	File-based persistence	45
6	Geospatial data and visualisation	47
6.1	GeoAPI	47
6.2	NetTopologySuite	47
6.3	SharpMap	47
6.4	The central map and its panel	47
6.5	Spatial operations	47
6.5.1	Introduction	47
6.5.2	Object model	48
6.5.3	Framework Embedding	49
6.5.4	Spatial Operations	50
6.5.5	GUI	51
6.5.6	Tests	52
7	Other topics	55
7.1	Tables and graphs	55
7.2	Basic Modeling Interface (BMI)	55
7.3	Remote Instance Container	55
7.4	NetCDF	55
7.5	Scripting using Python	55
8	Programming help	57
8.1	NuGet packages	57
8.2	32-bit vs 64-bit	57
8.3	Build process	57
8.4	Creating an installer	57
8.5	Testing	57
8.6	Code conventions	57
8.7	Localisation	57
8.8	Licensing	57
9	Plugins	59
9.1	Flexible Mesh	59
9.1.1	Introduction	59
9.1.1.1	Overview	59
9.1.2	The <code>FlexibleMeshModelApi</code> Class	60
9.1.3	The File System	61
9.1.4	The Unstructured Grid	62
9.1.5	Boundary conditions	62
9.1.6	GUI	63
9.1.7	Tests	63
A	Tutorial: start a new plugin	65
A.1	Add new plugin	65
A.1.1	Edit <code>AssemblyInfo.cs</code>	65
A.1.2	Edit project's <code>csproj</code> file	65
A.1.3	Add code	66
A.1.4	Result so far	67
A.2	Import time series from WaterML2 files	67
A.2.1	Create a new importer class	67
A.2.2	Register the importer in the application plugin class	70

A.2.3	Create test data	70
A.2.4	Test importer	70
A.3	Create a simple hydrological model	70
A.3.1	Create a new model class	70
A.3.2	Register the model in the application plugin class	73
A.3.3	Test model functionality	73
A.4	Customize model properties	73
A.4.1	Create a new object properties class	73
A.4.2	Register the object properties in the gui plugin class	73
A.4.3	Test properties	73
A.5	Show model data on a map	73
A.5.1	Create a new map layer provider	73
A.5.2	Register the map layer provider in the gui plugin class	73
A.5.3	Test map layer provider	73
A.6	Set up a custom model view	73
A.6.1	Create a new view	73
A.6.2	Register the view in the gui plugin class	73
A.6.3	Test view	73
A.7	Create a ribbon button	73
A.7.1	Create a new gui command	73
A.7.2	Create a new ribbon control	73
A.7.3	Register the ribbon control in the gui plugin class	73
A.7.4	Test ribbon button	74
A.8	WaterML2 test data	74

DRAFT

List of Figures

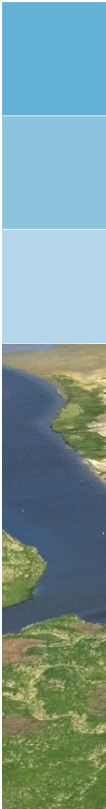
1.1	Class diagrams for plugins.	3
2.1	Class diagram for the Delta Shell application and its services.	5
2.2	Class diagram for <i>Project</i> and <i>ProjectItem</i>	6
2.3	Depicted is a sample script that gives an overview of the domain model of the current project, including some sample output.	7
2.4	Class diagram for activities and models.	8
2.5	Class diagram around <i>IHydroModel</i> and the integrated model: <i>HydroModel</i> . Notice that these classes are having somewhat confusing names: an <i>IHydroModel</i> is a model that contains an hydro region, whereas the <i>HydroModel</i> is a class that represents an <i>integrated model</i> , which is in turn a set of models. In other words, all <i>HydroModels</i> implement <i>IHydroModel</i> , but other (non-composite) models might implement this interface as well.	9
2.6	Class diagram for the Hydro region.	10
2.7	Class diagram for <i>IFunction</i> and related interfaces and classes.	10
2.8	Class diagram for model validation.	13
3.1	Class diagram for the Delta Shell GUI.	15
3.2	Class diagram for the Delta Shell main window.	16
3.3	Class diagram for the docking mechanism.	16
3.4	Class diagram for the <i>ViewResolver</i>	17
3.5	Class diagram for different types of views. <i>ProjectItemMapView</i> has been included in this diagram because it plays a central role in the Delta Shell GUI.	18
3.6	Class diagram for <i>TreeViews</i> . For readability, not all entries have been specified.	19
3.7	Class diagram for the Property Grid.	21
3.8	Class diagram for <i>ITimeNavigatable</i> . <i>ProjectItemMapView</i> and <i>NetworkSideView</i> are shown as examples of classes that implement <i>ITimeNavigatable</i>	23
4.1	Simple example of three classes with composition and aggregation relations.	25
4.2	Eventing using the Delta Shell Postsharp solution, based on the class diagram in Figure 4.1.	26
4.3	This code snippet shows how to subscribe to Postsharp-generated events and how to distinguish 'normal' effects from side effects.	30
4.4	Code snippet to illustrate the usage of <i>InvokeRequired</i>	31
5.1	Class diagram illustrating how a Delta Shell project contains both ORM and file-based persistence.	33
5.2	Class diagram illustrating how the nHibernate infrastructure is encapsulated in the <i>DeltaShellApplication</i> object. In order to keep this picture understandable, some interface classes have been omitted.	34
5.3	Class diagram of <i>NHibernateRepository</i> , one of the central classes in Delta Shell.	35
5.4	Example of how a class can be split up.	40
5.5	Example of how two classes can be merged.	42
5.6	Example of how a class can be split up into one base class and two derived classes. Some of the properties in the original class need to be divided over the new derived classes.	43
5.7	Class diagram describing file-based persistence. The given classes on the right of the diagram are examples: more classes implement the <i>IFileBased</i> interface.	46
6.1	UML diagram of spatial operation and operation sets	48

6.2	Three-operation chain	48
6.3	Three-operation set with subset topology	49
6.4	Spatial operation inheritance tree.	51
6.5	Class diagram on spatial operations.	52
6.6	Class diagram on how spatial operations are integrated into the GUI.	53
9.1	Global data model of the Flexible Mesh model in DeltaShell.	60
9.2	Remote and actual implementations of the <code>IFlexibleMeshModelApi</code>	60
9.3	Control structure of the <code>ReloadGrid</code> implementation	62
9.4	Boundary condition object model in <code>FMSuite</code>	63

List of To Do's

A.1 see section TODO 66

DRAFT



DRAFT

1 Introduction

In this chapter, a superficial overview of Delta Shell will be given. First, the structure of the entire solution will be discussed. Because the plugins are a central part of the system architecture, this aspect will be introduced in this chapter as well. The next two chapters will focus on the Delta Shell Application and GUI, respectively.

1.1 Overview of the directory structure

The entire directory is divided into a number of folders:

- Common** This solution folder is for all components that can be re-used by different applications. For instance, Delta Shell uses a number of libraries for geographical representations (GeoAPI, SharpMap, NetTopologySuite; see [chapter 6](#)). Other **Common** assemblies are: the Remote Instance Server (see [section 7.3](#)), DelftTools.Controls and DelftTools.Functions (see [section 2.6](#)).
- Delta Shell** This is the folder that is meant for the Delta Shell application itself. It includes the 'Delta Shell plugins': plugins that are distributed with all products that rely on the Delta Shell framework. In principle, the **Common** and **Delta Shell** folders together constitute what is commonly referred to as the 'Delta Shell Framework'. In this folder, there is one Visual Studio solution for the framework, which does not include any plugins. There is no single solution that includes *all* source code.
- Products** This folder contains the plugins for the individual products. A product can be seen as the combination of the framework with a certain set of plugins. Each product has a separate installer. At the time of writing, the largest product is **NGHS**¹. Other products include: MorphAn, Habitat and WFD Explorer. In the individual product folder, several solutions reside that include the Delta Shell Framework using NuGet packages (without its source code).

There are two start-up projects available: Delta Shell GUI and the Delta Shell Console. Delta Shell Console only starts the Application, whereas Delta Shell GUI starts the entire GUI, which in turn contains the Application. The Delta Shell Console will also only load the application plugins, whereas the GUI will load *all* plugins (both GUI and application plugins).

1.2 Plugins

A Delta Shell installation consists of two directories: *bin* and *plugins*. When Delta Shell is started, one of the first things it does is to search the *plugins* folder for assemblies that are recognised as Delta Shell plugins. This plugin architecture has the advantage that it is straightforward to make different distributions (e.g. SOBEK), with different sets of plugins. The plugins are mostly considered to be independent of each other, meaning that normally you would not want one plugin to reference another plugin. There are a few exceptions to this rule:

- ◇ You can reference 'Delta Shell' plugins: plugins that are always installed. Examples of such plugins are the Project Explorer and the NetCDF plugin.
- ◇ A GUI plugin can reference an application plugin. For instance, the Flexible Mesh GUI plugin references the Flexible Mesh application plugin.
- ◇ If there are common functionalities between two plugins, you can create a 'Common' plugin. For instance, both Waves and Flexible Mesh reference the FMSuite Common plugin.

¹ The Next Generation Hydro Software project (2010-2015) was the project during which the Delta Shell Framework created.



The plugins are recognised by means of the *AddinManager* from `Mono.Addins.dll`², which is wrapped in the static *PluginManager* class in *DeltaShell.Core*. For a class to be recognised as a Delta Shell plugin, it needs to be decorated as follows:

```
namespace DeltaShell.Plugins.ProjectExplorer
{
    [Extension(typeof(IPlugin))]
    public class ProjectExplorerGuiPlugin : GuiPlugin
    {
        ...
    }
}
```

The extension point is found as follows:

```
[TypeExtensionPoint]
public interface IPlugin
{
    ...
}
```

The *AssemblyInfo.cs* file needs the following declarations:

```
using Mono.Addins;

[assembly: Addin]
[assembly: AddinDependency("DeltaShellApplication", "1.0")]
```

The class that is responsible for keeping a reference to all plugins is *PluginManager*. A plugin developer normally does not need this class: registering it as either an application or GUI plugin is sufficient. The class diagram around the plugin architecture is depicted in figure 1.1. Here, a distinction is being made between GUI plugins and application plugins. The application plugins contain the domain objects of models, whereas the GUI plugins should only contain graphical objects, such as windows and panels. When the Console version of Delta Shell is started, only application plugins will be loaded.

²More information on Mono Addins: <http://monoaddins.codeplex.com/>

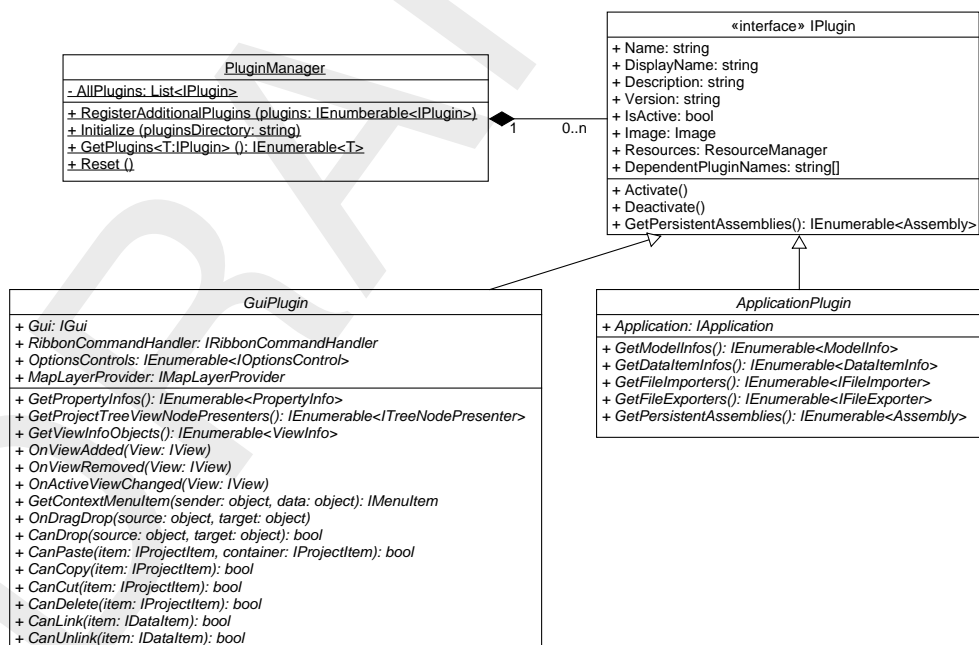


Figure 1.1: Class diagrams for plugins.

DRAFT

2 The Delta Shell application

As discussed in the previous chapter, Delta Shell makes a clear distinction between code that is related to the object model and code that has to do with the GUI. This chapter will focus on the application part of the codebase. We shall explain what the general shape of a Delta Shell project is, and how it can be composed of models and other activities. Data items form an integral part of projects, because it facilitates sharing of data between models, and will be discussed in [section 2.5](#). An introduction to a few available mathematical building blocks (functions, coverages, variables, multi-dimensional arrays) will be given in [section 2.6](#). Last, validation of models will be touched upon briefly.

2.1 Introduction

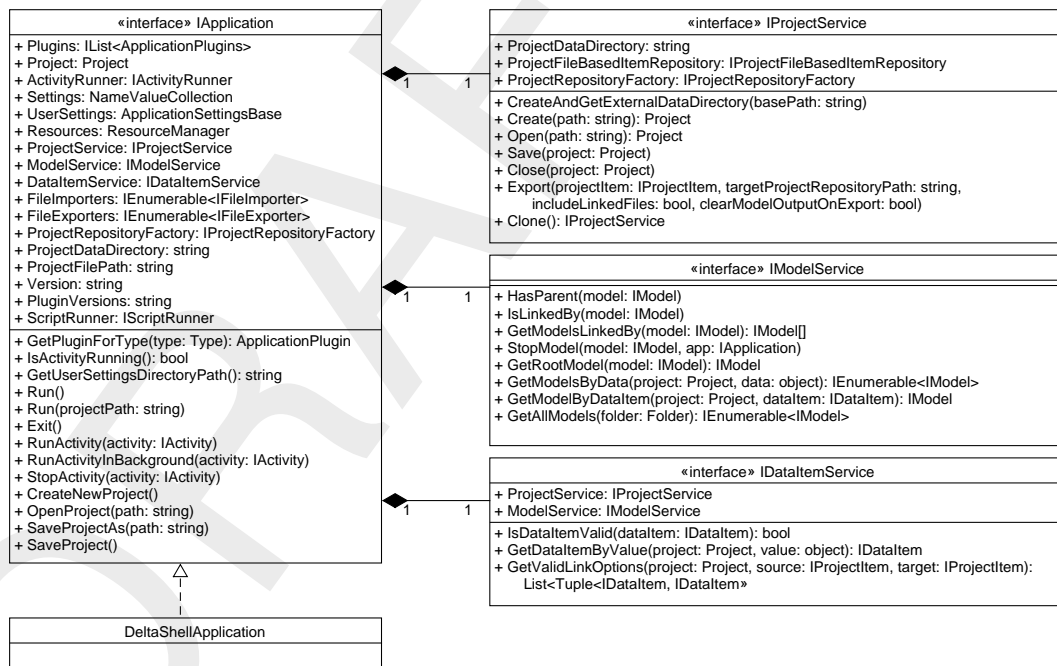


Figure 2.1: Class diagram for the Delta Shell application and its services.

The class diagram for *IApplication* is depicted in [Figure 2.1](#). It contains a list of application plugins, which is filled once the application starts. The *Project* is the object that is persisted eventually in a *.dsproj* file, and most operations on the project are handled via the *ProjectService*. The project will contain one or more models, which can be run in the *ActivityRunner*. All models are not the only type of activities: importers and exporters are other types of activities. The *FileImporters* and *FileExporters* are collected via all plugins that expose such importers and exporters. Python scripts, which interact directly with the Delta Shell Framework API, can be run through the *ScriptRunner* (see [section 7.5](#)). *IModelService* and *IDataItemService* are services that are used for coupling models.

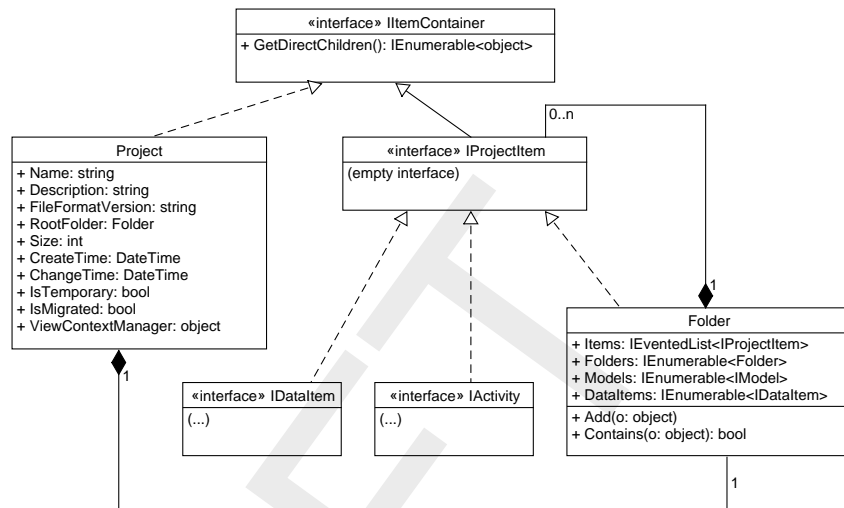


Figure 2.2: Class diagram for Project and ProjectItem.

2.2 Project and project items

A class diagram of projects and its subparts can be seen in [Figure 2.2](#). Every *Project* contains exactly one (root) *Folder*, which in turn contains a number of other folders (recursive!), data items and activities (grouped under a category ‘project items’). Activities can contain other activities as well (‘composite activities’), and activities can contain data items. Activities/models and data items will be discussed in more detail in [section 2.3](#) and [section 2.5](#), respectively.

Because the *Project* object is the root for the entire composition of domain objects, it is important how all domain objects relate to *Project*. In [Figure 2.3](#), a simple script is presented, that recursively walks through all folders, data items and activities. Also, sample output for this script is shown. This script can help to get a good understanding of the object model at any time, while running Delta Shell.

2.3 Activities and models

In Delta Shell, there is a separate service to run activities in the background: the *ActivityRunner* (see [Figure 2.4](#)). These activities will run in a separate thread normally. In some cases, even a new process will be started, but that is up to the activity itself. When the activity runner is active, a modal pop-up will appear to indicate that one or more activities are running. The main application can not be operated by the user when the activity runs. Internally, the activity runner uses a FIFO queue to determine the order of the activities.

Activities are divided into several stages. An activity reaches the following stages if it proceeds normally (there is also a Failed stage in case of errors):

- ◇ None: before the initialisation
- ◇ Initializing: during the initialisation
- ◇ Initialized: after the initialisation
- ◇ Executing: during the execution (of one step)
- ◇ Executed: after the execution (of one step)
- ◇ Done: after the execution (of all steps)
- ◇ Finishing: during the finalisation
- ◇ Finished: after the finalisation
- ◇ Cleaning: during the clean-up

```
def write_tree(f, node, indent) :
    f.write(" " * indent + str(node.ToString()) + " : " + str(type(node)) + '\n')
    if "GetDirectChildren" in dir(node) :
        for childNode in node.GetDirectChildren() :
            write_tree(f, childNode, indent+4)

f = open("d:/tree.txt", 'w')
write_tree(f, CurrentProject, 0)
f.close()
```

```
Project1 : <type 'Project'>
  <root> : <type 'Folder'>
    Integrated Model : <type 'HydroModel'>
      start time : <type 'DataItem'>
        start time : <type 'Parameter[DateTime]'>
      stop time : <type 'DataItem'>
        stop time : <type 'Parameter[DateTime]'>
      Region : <type 'DataItem'>
        DelftTools.Hydro.HydroRegion : <type 'HydroRegion'>
          Network : <type 'HydroNetwork'>
            Channell : <type 'Channel'>
              LateralSource1 : <type 'LateralSource'>
              ObservationPoint1 : <type 'ObservationPoint'>
              Weir1 : <type 'Weir'>
              StructureFeature : <type 'CompositeBranchStructure'>
            Node001 : <type 'HydroNode'>
            Node002 : <type 'HydroNode'>
            Main : <type 'CrossSectionSectionType'>
            DelftTools.Hydro.DrainageBasin : <type 'DrainageBasin'>
              Catchment1 : <type 'Catchment'>
              HydroLink1 (Catchment1 -> LateralSource1) : <type 'HydroLink'>
            Network : <type 'DataItem'>
              (convertor) reference to Network :
                <type 'AggregationValueConverter'>
            Basin : <type 'DataItem'>
              (convertor) reference to DelftTools.Hydro.DrainageBasin :
                <type 'AggregationValueConverter'>
            Flow1D : <type 'WaterFlowModel1D'>
              start time : <type 'DataItem'>
                start time : <type 'Parameter[DateTime]'>
              stop time : <type 'DataItem'>
                stop time : <type 'Parameter[DateTime]'>
```

Figure 2.3: Depicted is a sample script that gives an overview of the domain model of the current project, including some sample output.

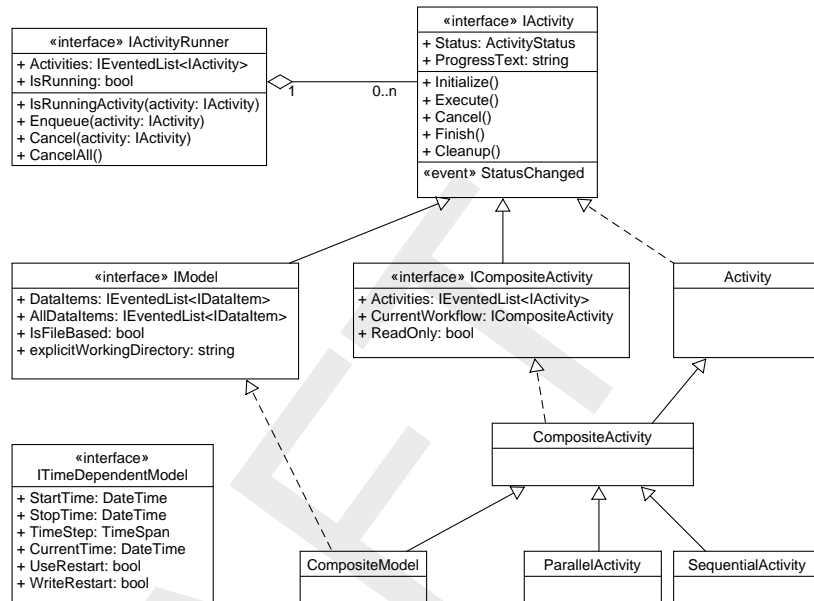


Figure 2.4: Class diagram for activities and models.

◇ Cleaned: after the clean-up

Most time-dependent models use a scheme of using the ‘execution’ step for every time step. This part is called repeatedly, until the ‘current time’ has reached the ‘stop time’, after which the model will finish. There is a distinction between ‘finishing up’ and ‘cleaning up’: when an exception occurs during initialisation, execution or finalisation of the activity, the clean-up procedure will be called. This stage can therefore safely be used for implementing the *IDisposable* interface, to avoid memory leaks.

Typical workflows in Delta Shell that are exposed as an *Activity* include the running of a model and importing/exporting data from/to files. Models are activities appended with properties that accommodate running models individually and coupling models together. If an activity is composed of several sub-activities, a composite activity can be used, which, if applied recursively, can form a tree of activities. Even though the name suggests it, a *ParallelActivity* is not entirely parallel (as in: parallel threads), but means that the ‘execute’ steps of the activities are performed alternately. This gives the possibility for the pseudo-parallel models to exchange information after each time step. The main instrument to share data between such coupled models are *data items*, to be discussed in [section 2.5](#).

2.4 Hydro models and regions

[Figure 2.5](#) shows a number of objects that are frequently used in the line of NGHS products¹. The central concept that was intended to be implemented was *integrated modelling*: a standardised integration of different models, which would be able to seamlessly share information while running the models. The implementation of the integrated model is captured in *Hydro-Model*. This composite, time-dependent model can hold several types of models. Because the different plugin assemblies are not allowed to reference each other (see [section 1.2](#)), an omniscient Python script is used to bind the different types of models together: *HydroModel-Builder.py*. It can be informative to take a look at this script to understand which models can

¹ It is actually unclear whether these objects are actually part of the Delta Shell Framework. However, the NGHS product line form a large part of the Delta Shell plugins, currently, so this part will be discussed superficially.

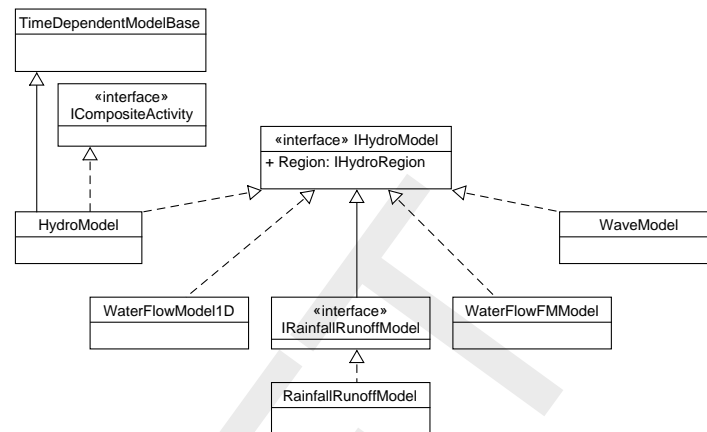


Figure 2.5: Class diagram around `IHydroModel` and the integrated model: `HydroModel`. Notice that these classes are having somewhat confusing names: an `IHydroModel` is a model that contains an hydro region, whereas the `HydroModel` is a class that represents an integrated model, which is in turn a set of models. In other words, all `HydroModels` implement `IHydroModel`, but other (non-composite) models might implement this interface as well.

be linked together and which can not. In this setting, a *workflow* is a set-up of parallel and sequential composite models, which are pre-defined in the `HydroModel` builder script as well.

According to the Delta Shell philosophy, model data is separate from GIS² data. The idea behind this is that GIS data is likely to be acquired from external sources (e.g. ArcGIS), and remains mostly constant across models or scenarios. The GIS data of NGHS models are represented in the ‘hydro region’. Although different models do generally not have access to each other’s model data, all models do have access to all GIS data.

The class diagram for the hydro region is depicted in Figure 2.6. Depending on the model, the hydro region can contain three types of subregions: *Network* (for Flow1D models), *Drainage-Basin* (for Rainfall-Runoff models) and *HydroArea* (for Flexible Mesh models). A *HydroLink* object is used to link two *IHydroObjects* from (usually) different *IHydroRegions*. An example of the use of *HydroLinks* is the discharge that is transferred from a Rainfall-Runoff catchment to a Flow1D channel.

2.5 Data items and model coupling

TODO: Obsolete when all model couplings are performed using `d_hydro.exe`?

2.6 Functions and spatial data

Delta Shell offers facilities for defining a number of mathematical concepts³. These mathematical concepts are often used as the underlying data structures that are sent to the kernels, and to save results from the kernels in. Also, these structures are used as data sources for the visualisation. It is therefore recommended to understand the basic functionalities that Delta Shell has to offer in this respect.

²GIS: Geographic Information System.

³The description of this functionality in this document will be quite practical. A more theoretical background on this topic can be found in [the following conference paper](#).

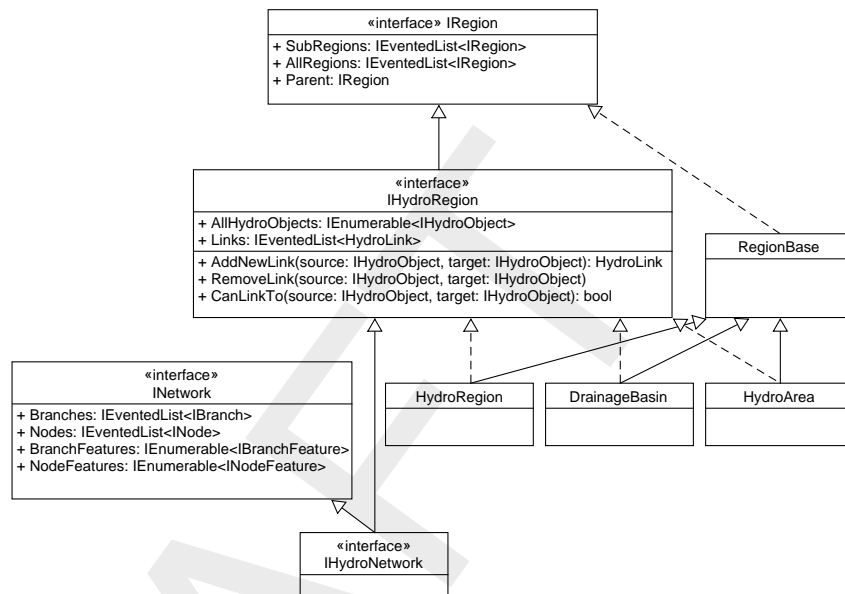


Figure 2.6: Class diagram for the Hydro region.

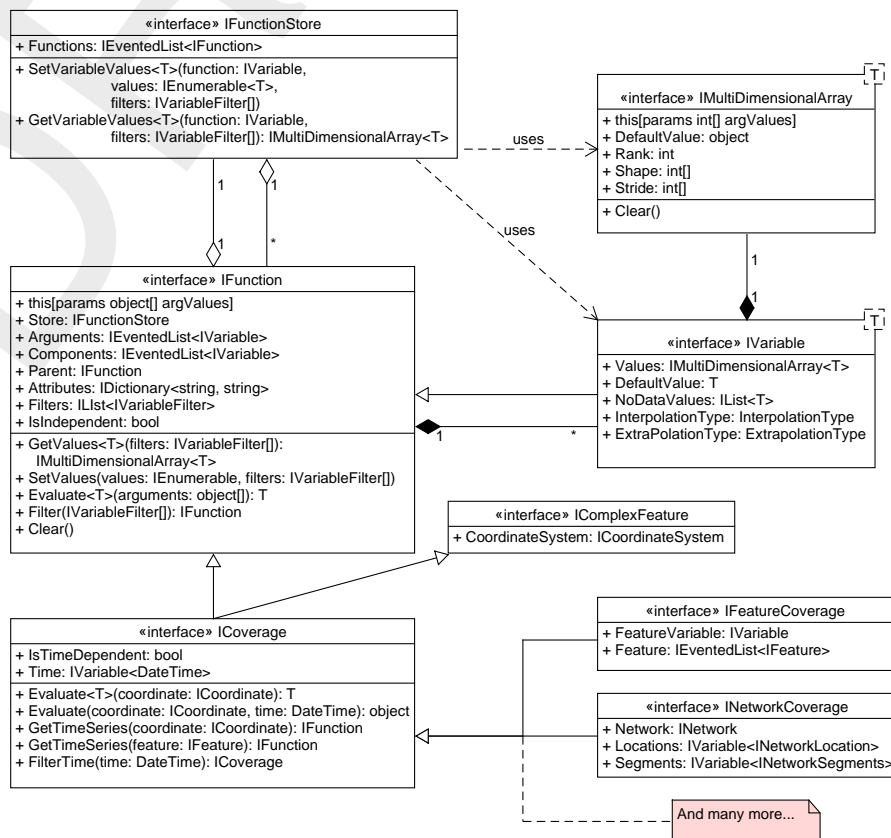


Figure 2.7: Class diagram for IFunction and related interfaces and classes.

2.6.1 Functions

One of the central concepts is the *IFunction* (see [Figure 2.7](#) for a class diagram): It defines a function, which can have an arbitrary number of arguments and components (which are sometimes called independent and dependent variables as well, respectively). In other words, it can map from an m -dimensional space to an n -dimensional space:

$$F = (f_1, f_2, \dots, f_n)(x_1, x_2, \dots, x_m) \quad (2.1)$$

Under this nomenclature, F is the function, f_1, f_2, \dots, f_n are the components (or: dependent variables) and x_1, x_2, \dots, x_m are the arguments (or: independent variables). In theory, arguments can be functions by themselves (which in fact they are, according to the class hierarchy), but that hardly happens. Let's consider a few examples to illustrate the use of functions:

- ◇ When a time series for water level is requested in the central map, Delta Shell will create a simple function, with one argument (time) and one component (water level).
- ◇ A constant wind field in Flexible Mesh is defined as a vector with two components (an x-component and a y-component) and two arguments (the x-location and the y-location).

A function can have arguments and components with different units (e.g. time, water level, concentrations, etc), but a single variable is always expressed in one unit. Therefore, *IVariable* is a templated class. Variables also have one multi-dimensional array, which contains the actual data. This needs to be generic, because the dimensionality of the variable is determined by the arguments of the variable/function it depends on. For instance, in the constant wind field example just discussed, the shape of the components' arrays is 2-dimensional: the number of x-locations on one axis and the number of y-locations on the second axis. Therefore, if an argument is added to a function, the dimensionality of the components is updated as well.

2.6.2 Coverages

A coverage (as depicted in [Figure 2.7](#), also often called 'spatial data') is a special kind of function: it defines values with a location as one of its arguments. Therefore, it also inherits from *IFeature* (see [chapter 6](#) for more information on any kind of spatial data). Additionally, it might be time-dependent. For instance, a coverage defined for *initial* water level is defined only once: therefore, it is not time-dependent and has only one argument (location, in the form of a feature) and one component (water level). On the other hand, the water level coverage that is the result of a computation is a time-dependent coverage, so it has two arguments instead of one: location and time.

It is sensible to have custom-made coverages for different types of features. In the current implementation, custom coverages have been crafted for different kinds of features: *NetworkCoverage*, *UnstructuredGridCoverage*, *CurvilinearGridCoverage*, *FeatureCoverage*...

2.6.3 Function filters

Occasionally, the user does not need all data that is available in a function, but a subset or a reduced version. To this end, filters have been defined in Delta Shell. The following filters are frequently used (more are available):

- ◇ *VariableValueFilter*: creates a slice from the data based on values, without reducing dimensionality.
- ◇ *VariableIndexFilter*: idem, but uses indices to filter.
- ◇ *VariableReduceFilter*: reduces the dimensionality of the data. For instance, create a time series from a coverage that is both time and location-dependent. The output function will

only depend on time, and not on location. In order to use this filter, a *VariableValueFilter* needs to make the remaining arguments unique. See below for an example.

```
var function2D = new Function { Arguments = { x1, x2 },  
                               Components = { y } };  
  
IFunction filteredFunction = function2D.Filter(  
    new VariableValueFilter<int>(x1, 333),  
    new VariableReduceFilter(x1));  
  
// filteredFunction now has one argument (x2).  
// All y values contained in filteredFunction are now evaluated with x1=333
```

Filters only represent a view on the original data, they do not perform a copy or execute the operation. The implementation of filters is defined in every function store (see below). So, the memory store contains an implementation of the variable value filter, but the netCDF function store has it as well. It is possible to manipulate the original values of a function through a filtered version.

2.6.4 Evaluating a function

One of the features of an *IFunction* is that it is not necessary to provide an input value at a specific data point to retrieve a value for a component for that data point. For each argument and component, an interpolation and extrapolation strategy is defined (e.g. linear or constant), which is respected when the best possible approximation is computed. The actual implementation of the intrapolations and extrapolations can be found in the *Function* class.

2.6.5 Function store

The function store is the place where the data inside the function are actually stored. This can be memory (*MemoryFunctionStore*) or a netCDF file (*NetCdfFunctionStore* and *ReadOnlyNetCdfFunctionStoreBase*), for instance. In the memory function store, all data is always in memory, but for external sources, such as a netCDF file, this needs not be the case. This is also not always desirable: if a netCDF file is very large, it might be worthwhile to only load and cache the data that is requested. Therefore, if an owner requests or sets data from *Function* via *GetValues()* or *SetValues*, *Function* will not directly access its variables and arrays, but will dispatch this call to the function store. The function store will then either get the values from memory or from the external source.

2.7 Model validation

The user can create a model freely, and customise its settings as he wishes. However, this may lead to a model configuration of which it can be recognised by the plugin that it is invalid. For instance, a Flow1D model that contains different types of cross sections on one branch is considered to be invalid. Before the model is run, it is checked that the model can be validated. If this is not the case, Delta Shell will stop during the initialisation of the model. A user can also validate the model manually. The resulting view is the *ValidationView*, which will show a report of the validations that have been performed.

The resulting object model (see [Figure 2.8](#)) is fairly simple, with a tree structure of validation reports and validation issues. Validation issues have a severity (info, warning or error). When the severity of a *report* is calculated, the level of the most severe issue that is under that tree is taken. A validation issue also contains information about the object that is incorrect. For

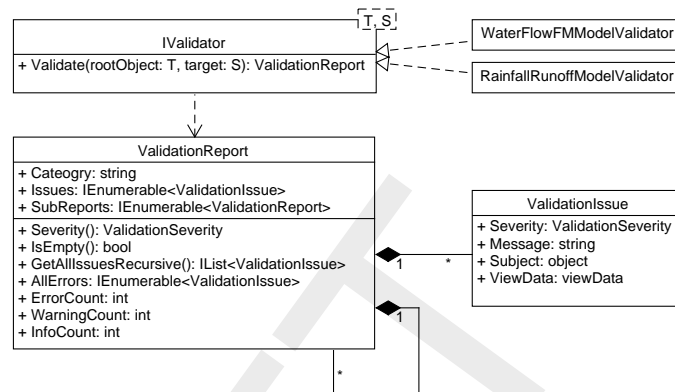


Figure 2.8: Class diagram for model validation.

instance, if one control group in RTC is incorrect, a reference to that control group is put in the *Subject* property. This can be used to guide the user to an applicable view when he clicks on a validation issue in the validation view.

DRAFT

3 The Delta Shell GUI

3.1 Overview

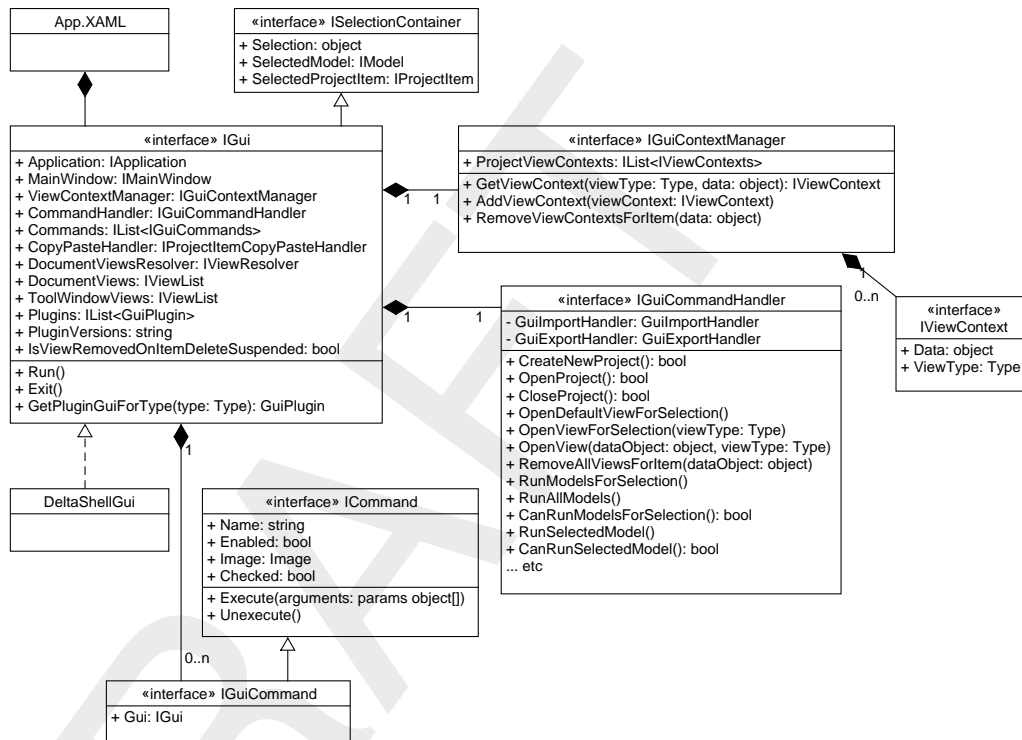


Figure 3.1: Class diagram for the Delta Shell GUI.

Delta Shell is started as a Windows Presentation Framework (WPF) application. The corresponding startup XAML file (*App.xaml*) can be found in the *DeltaShell.Gui* project. WPF has been chosen as the start platform, because it supports the external component *FluentRibbon*, which manages and visualises the ribbon in Delta Shell. Most other user controls (including windows and panels) use standard components from Windows Forms instead, although it is still possible to use WPF components in your plugin.

When *App.xaml* is started, it will create a *DeltaShellGui* object (which inherits from *IGui*; see Figure 3.1.). *IGui* contains an *Application* (discussed previously), as well as the *MainWindow* (the actual window; discussed below). While *IApplication* administers a list of application plugins, *IGui* has a list of GUI plugins. The *Commands* are used to standardise certain frequent operations that can be issued to the GUI. Eventually, the *CommandHandler* will dispatch the given commands. Views and the *ViewResolver* are discussed in [section 3.4](#).

As a ribbon is supposed to do, it changes its behaviour based on its context. Depending on the selected object, menu and buttons are activated and deactivated, or disappear entirely. Menus and buttons can be introduced by plugins, and are found in the *Ribbon.xaml* files in the respective GUI plugins.

The main window is captured in the *MainWindow* object, which realises the *IMainWindow* interface (see Figure 3.2). It contains direct references to three important parts of the graphical interface: the project explorer, the property grid and the log message window. Each of these will be discussed in separate parts below.

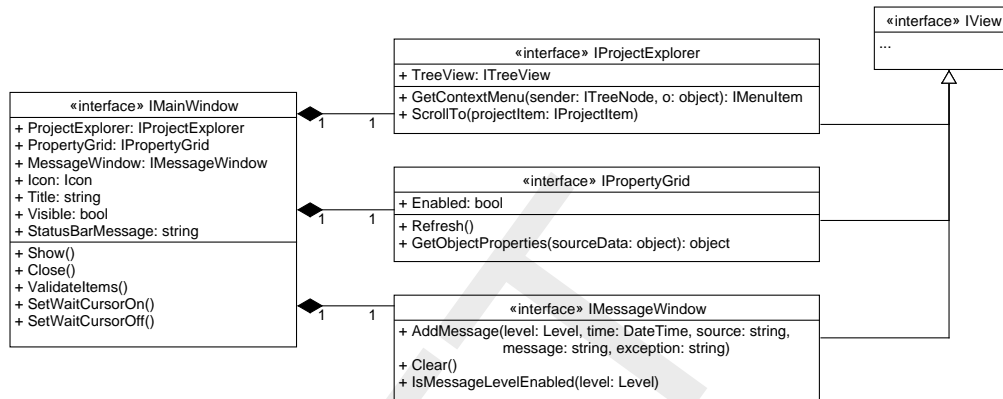


Figure 3.2: Class diagram for the Delta Shell main window.

3.2 The ribbon

Delta Shell uses *FluentRibbon* for its ribbon functionality. The definition of the (WPF) ribbons is done in files with the extension *.xaml* and *.xaml.cs*. Because of the plugin structure of Delta Shell, the ribbon is dynamic, depending on which plugins are loaded. Therefore, ribbon fragments are defined in the GUI plugins, and they need to be registered in the GUI plugin's *RibbonCommandHandler* property. This interface exposes the *Commands* list, which is queried by the Delta Shell GUI, in order to maintain an extensive list of commands from *all* GUI plugins.

A common type of command is the *MapToolCommand*. A map tool command uses a *MapTool* which interacts with the central map. Such commands can add, remove, move, edit (etc) map features, or zoom in or pan.

3.3 Docking

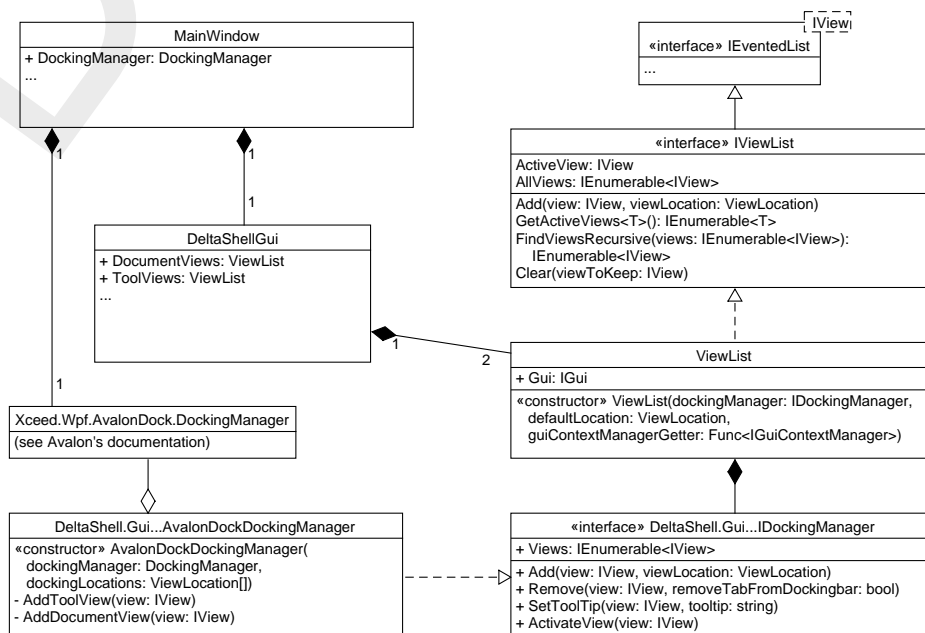


Figure 3.3: Class diagram for the docking mechanism.

Delta Shell uses a system of dockable windows to give the user a good control of how she organises her application. Internally, Delta Shell employs the AvalonDock component to achieve the docking behaviour. *MainWindow's DockingManager* is the Avalon docking manager that is instantiated only once, which is passed through to two *ViewLists*: one for documents (which can only appear in the center top or as a floating view) and one for tools (which can appear on the sides (left, right or bottom) or as a floating view). The *ViewList* is eventually a list of *IViews*. Each *ViewList* also has its own (Delta Shell) docking manager: the *AvalonDockDockingManager*.

3.4 Windows and views

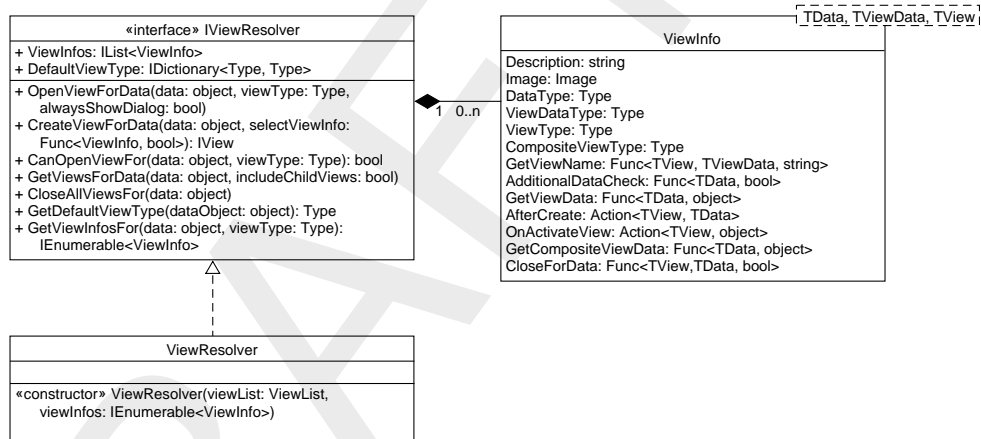


Figure 3.4: Class diagram for the ViewResolver.

A *ViewInfo* object is defined in a GUI plugin, which exposes its *ViewInfo* objects in the *GetViewInfoObjects()* method. The purpose of this class is to create a relation between data objects and views. This way, a plugin can universally send a message to the *IGuiCommandHandler* to open the view that is associated with that object type. For instance, if the user clicks on the 'General' project tree item in the Flexible Mesh plugin, the following *ViewInfo* definition is used:

```

yield return new
    ViewInfo<FlowFMTreeShortcut, WaterFlowFMMModel, WaterFlowFMMModelView>
    {
        Description = "FM Model",
        GetViewName = (v, o) => o.Name + " (FM model)",
        AdditionalDataCheck = o => o.CanSwitchToTab,
        GetViewData = o => o.Model,
        CompositeViewType = typeof(ProjectItemMapView),
        GetCompositeViewData = o => o.Model,
        OnActivateView = (v, o) => ((FlowFMTreeShortcut)o).NavigateToInView(v),
        AfterCreate = (v, o) =>
        {
            v.Gui = Gui;
            o.NavigateToInView(v);
        }
    };

```

In the constructor of this *ViewInfo* object, the three template parameters (*FlowFMTreeShortcut* etc) are *DataType*, *ViewDataType* and *ViewType* respectively. In this case, when the user double-clicks on the 'General' item in the project tree (a *FlowFMTreeShortcut*), it needs to

open the view for a *WaterFlowFMModel*, the *ViewDataType*. *GetViewData* is the delegate that indicates how the *ViewResolver* can determine what the data is that needs to be visualised (in this case: the FM model itself). Furthermore, you can indicate whether an extra check needs to be done before considering opening this view (*AdditionalDataCheck*). The *IView* that will be opened (*WaterFlowFMModelView* in this case), is part of a composite view. The type of that composite view also needs to be given (*ProjectItemMapView* in this case, generally known as the central map). In many cases, *TData* equals *TViewData*.

The *ViewResolver* is responsible for the behaviour described above. An often used entry point is *OpenViewForData*. This method will search for all *ViewInfos* that have a matching *TData*. If there is more than one matching *ViewInfo*, a number of heuristics have been implemented to determine which *ViewInfo* has priority. If there are still multiple options afterwards, the user will be presented a dialog in which the user can pick his desired view.

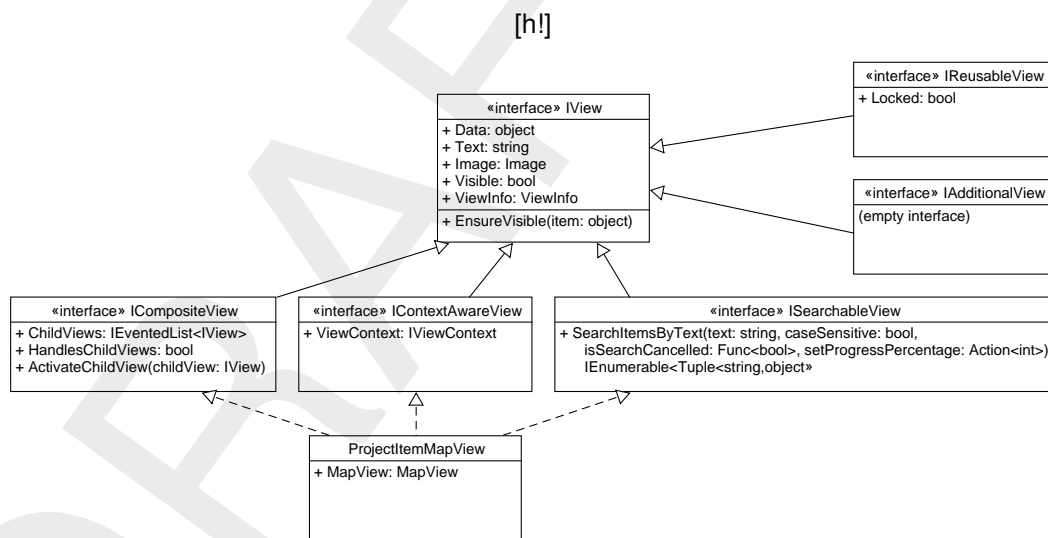


Figure 3.5: Class diagram for different types of views. *ProjectItemMapView* has been included in this diagram because it plays a central role in the Delta Shell GUI.

All views that can be shown using the docking mechanism need to comply to the constraints that are specified by the *IView* interface (see [Figure 3.5](#)). Whenever a view is opened using the *ViewResolver*, the *ViewInfo* object that is used for the creation of the view, is copied into the *ViewInfo* property of the view itself. The view data will be copied into the *Data* property of the *IView*.

There are a number of interfaces that inherit from *IView* that need to be mentioned here:

ICompositeView A composite view consists of several views. A good example of this type of view is the *BoundaryConditionEditor* in the Flexible Mesh plugin.

IContextAwareView A view that is context-aware, has a so-called 'view context'. This means that the view will be configured differently, depending on the data object that is visualised or edited. This topic is covered in some more depth in [section 3.5](#).

ISearchableView Certain views can be searched in the GUI using the keyboard shortcut Ctrl-F. This functionality is available to all views that implement *ISearchableView*.

IReusableView A view is called reusable if the *Data* object can be changed in the view. For instance, when a user inspects a cross section in SOBEK (in the *CrossSectionView*) and double-clicks a new cross section in the central map, the new cross section will appear in the reused view, instead of opening a new view.

3.5 The GUI Context Manager

The *GUIContextManager* has the role to administer all view contexts. A view context here defines how a view should behave. This behaviour can also be persisted into the project database. A good example of a view context can be found in the Real-Time Control (RTC) plugin. The Control Group Editor has a Boolean setting ('auto-resize') that can have distinct values for different data objects: for instance, the user wants to have this setting turned on for Control Group 1, and turned off for Control Group 2. Instead of registering this setting in the data object itself (which would mean that GUI information is written into a non-GUI object), two view contexts are registered in the *GUIContextManager*: one for Control Group 1 and one for Control Group 2.

3.6 Treeviews

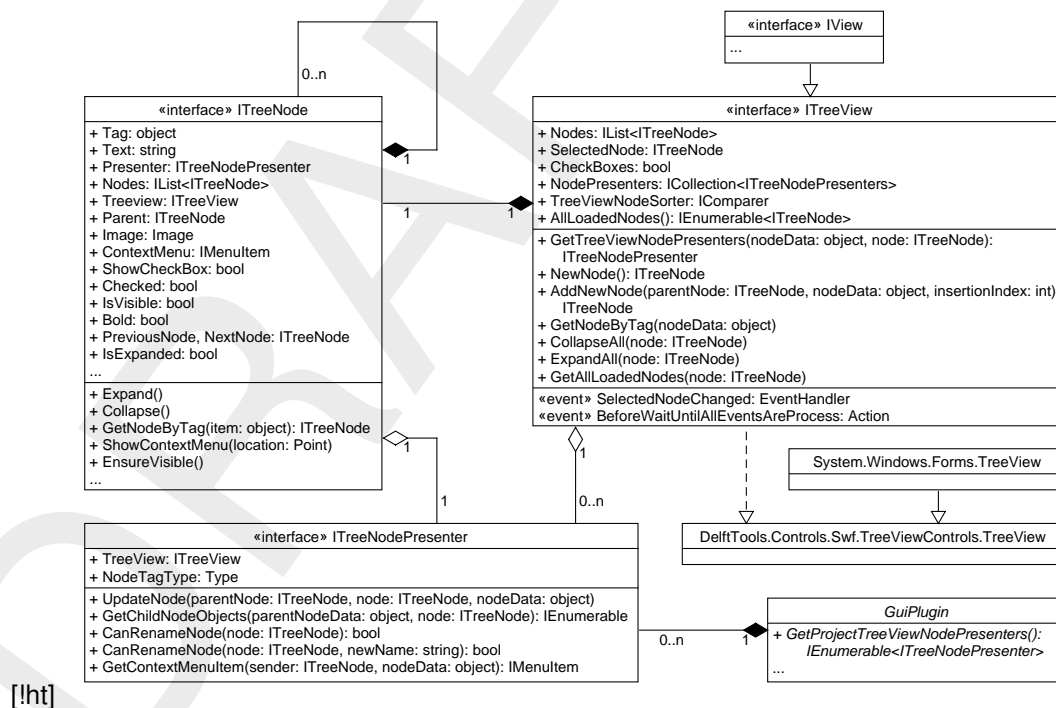


Figure 3.6: Class diagram for TreeViews. For readability, not all entries have been specified.

Treeviews are a commonly used user control in Delta Shell. Among others, it is used in the following graphical components:

- ◇ The project explorer: the panel that is usually at the left top. It contains the entire project structure (to be discussed in [section 3.7](#))>
- ◇ The map legend: the panel that by default is located at the left bottom of the window. This view will be discussed in more detail in [section 6.4](#).
- ◇ The region explorer. This component is mostly used in SOBEK, to navigate through the objects (channels, cross sections etc.) that have been created by the user.
- ◇ The spatial operations window (see [section 6.5](#)).

The class diagram for this component is depicted in [Figure 3.6](#). Although the standard component from the Windows Forms library is used, a new interface has been defined to standardise the possibilities that are used in Delta Shell: *ITreeView*.

The interface that programmers will use most is *ITreeNodePresenter*. This defines in the

GUI plugin how the project tree will look like in the GUI. In practice, most node presenters inherit directly from *TreeViewNodePresenterBase<T>*, where *NodeTagType=T*. Node presenters indicate how the node itself should look like (text and image) using the *UpdateNode* method. Node presenters also declare what its immediate children should be, using the *GetChildNodeObjects*. *TreeFolders* are used to create a folder structure (to separate input and output in the Project Explorer for instance). *TreeFolder* has its own *TreeFolderNodePresenter*. Otherwise, the objects that need to be represented in the tree structure are returned by *GetChildNodeObjects*. The *TreeView* will then search for appropriate node presenters that can represent that object. For instance, a wind object is returned by the model node presenter's *GetChildNodePresenter* method. For this wind object, a separate wind node presenter is available:

```
public class SomeModelNodePresenter : ModelNodePresenterBase<SomeModel>
{
    public override void UpdateNode(ITreeNode parentNode,
                                   ITreeNode node,
                                   SomeModel nodeData)
    {
        node.Text = nodeData.Name;
        node.Image = Properties.Resources.ModelIcon;
    }

    public override IEnumerable GetChildNodeObjects(SomeModel parentNodeData,
                                                    ITreeNode node)
    {
        yield return parentNodeData.Wind; // NOT an ITreeNode, but the data itself.
        yield return parentNodeData.InitialConditions;
    }
}

public class SomeModelWindNodePresenter :
    TreeViewNodePresenterBaseForPluginGui<WindDefinition>
{
    public override void UpdateNode(ITreeNode parentNode,
                                   ITreeNode node,
                                   WindDefinition nodeData)
    {
        node.Text = "Wind";
        node.Image = Properties.Resources.Wind;
    }

    public override IEnumerable GetChildNodeObjects(WindDefinition parentNodeData,
                                                    ITreeNode node)
    {
        yield break; // No children
    }
}
```

3.7 Project explorer

TODO.

3.8 Property grid

The panel in the right bottom of the window is called the property grid. It uses the standard Windows Forms *PropertyGrid* component¹. The main input to this *PropertyGrid* is the

¹For a nice introduction, please visit: <http://www.codeproject.com/Articles/22717/Using-PropertyGrid>.

SelectedObject, of which the properties are derived by reflection. The *PropertyGrid* implementation is therefore agnostic to the type of objects it receives via *SelectedObject*.

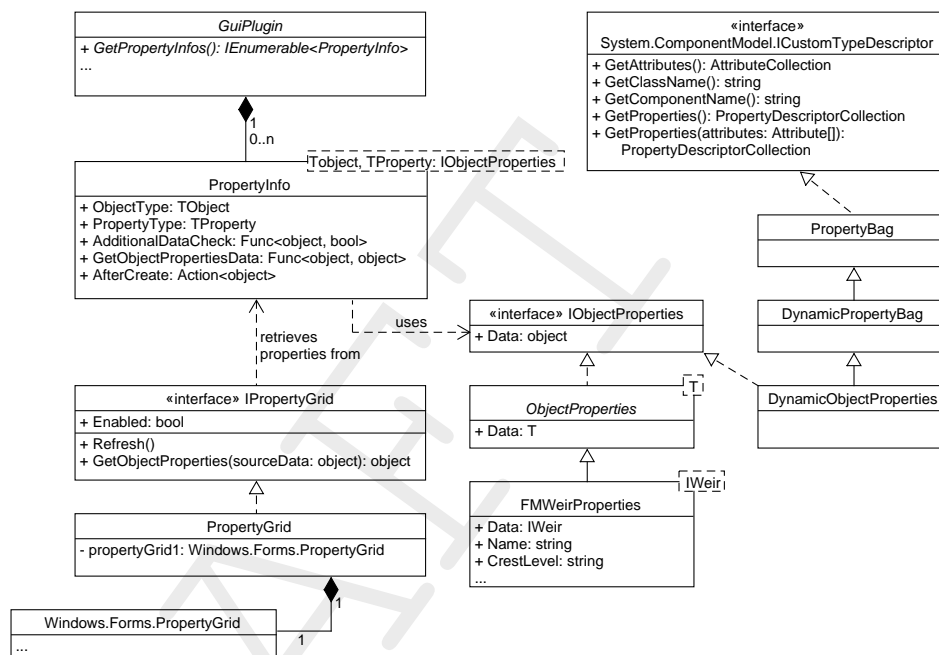


Figure 3.7: Class diagram for the Property Grid.

In Delta Shell, a distinction is made between the object that is selected (e.g. a weir, represented by an *IWeir* object) and its properties (e.g. *FMWeirProperties*). The link between these two concepts is made in the GUI plugin, where *PropertyInfo* objects are registered (see Figure 3.7 for a class diagram). When an object is selected in the GUI (expressed in *IGui.Selection*), an event is fired to update the property grid. The type of the selected object is then looked up in the *PropertyInfos* of all GUI plugins, and the corresponding property data is retrieved using *GetObjectPropertiesData*.

The properties that are shown in the property grid can be created in two ways: statically (using *ObjectProperties<T>*) or dynamically (using *DynamicObjectProperties*). They will be discussed separately below.

3.8.1 Static declaration of properties

A static declaration of properties looks as follows²:

```
[ResourcesDisplayName(typeof(Resources), "WindFunctionProperties_DisplayName")]
public class WindFunctionProperties : ObjectProperties<WindFunction>
{
    [PropertyOrder(1)]
    [ResourcesCategory(typeof(Resources), "Categories_General")]
    [ResourcesDisplayName(typeof(Resources), "Common_Name_DisplayName")]
    [ResourcesDescription(typeof(Resources), "WindFunctionProperties_Name")]
    public string Name
    {
        get { return data.Name; }
    }
}
```

²Please consult [the WinForms documentation](#) for details on the attributes with which the properties are decorated.

```
}  
}
```

In this case, the *WindFunctionProperties* class has a reference to the original data (the *WindFunction*, via *data*), so that the values in the grid can be retrieved. The following *PropertyInfo* object needs to be registered in the GUI plugin for this object to show up in the property grid:

```
public override IEnumerable<PropertyInfo> GetPropertyInfos()  
{  
    yield return new PropertyInfo<WindFunction, WindFunctionProperties>();  
    ...  
}
```

3.8.2 Dynamic declaration of properties

Dynamic properties can just be added to the object that you want to show the properties of. See for instance the following code from *WaterFlowFMModel.cs*:

```
[PropertyGrid]  
[DisplayName("Validate before run")]  
[Category("Run mode")]  
public bool ValidateBeforeRun { get; set; }
```

In order for dynamic properties to show up in the property grid, the following *PropertyInfo* object needs to be registered in the GUI plugin:

```
public override IEnumerable<PropertyInfo> GetPropertyInfos()  
{  
    yield return new PropertyInfo<WaterFlowFMModel, DynamicObjectProperties>();  
    ...  
}
```

3.8.3 Property categories and read-only properties

You can specify categories using the *[Category]* attribute. If you want to change the order in which the categories appear, you need to use the 'tab trick' (the more tabs, the lower the category will appear in the property grid):

```
[Category("\t\t\t\t\tGeneral")]
```

The relative order of the properties is determined by the *[PropertyOrder]* attribute. Additionally, you can set certain properties to be read-only, by using the *[ReadOnly(true)]* attribute. In some cases, you might want a property to be read-only depending on certain conditions. In that case, you will need the *[DynamicReadOnly]* attribute on the property that you want

to make dynamically read-only, and a Boolean method that is decorated with the *[DynamicReadOnlyValidationMethod]*. In case there are multiple dynamic read-only properties, only one validation method is necessary.

```
[PropertyOrder(0)]
[DisplayName("Enable salt")]
public bool EnableSalt { get; set; }

[PropertyOrder(1)]
[DynamicReadOnly]
[DisplayName("Salt parameter 1")]
public int SaltParameter1 { get; set; }

[DynamicReadOnlyValidationMethod]
public bool ValidateDynamicAttributes(string propertyName)
{
    if (propertyName.Equals("SaltParameter1") && !EnableSalt)
    {
        return true; // This property is made read-only.
    }
    return false; // Otherwise: property is editable.
}
```

3.9 Time navigator

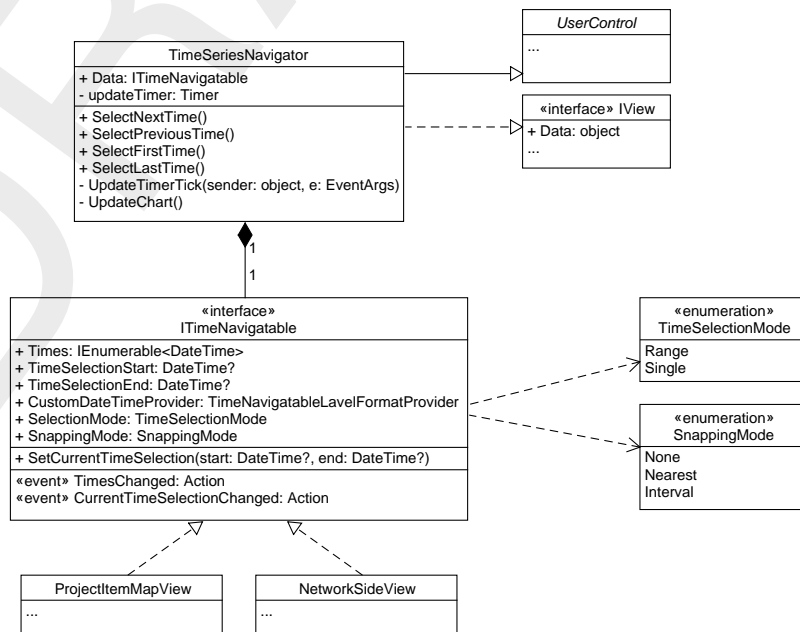


Figure 3.8: Class diagram for *ITimeNavigatable*. *ProjectItemMapView* and *NetworkSideView* are shown as examples of classes that implement *ITimeNavigatable*.

Many of the models in Delta Shell are time-dependent, and Delta Shell caters for this functionality with a time navigator. By default, it is located at the center bottom of the window. When the user has a time-dependent view or map layer, the user can browse through the time with this control, or let it play automatically. The view or map layer that is active will be updated along with time in the time navigator, yielding an animation-like behaviour.

In order for a view or map layer to show this behaviour, it needs to implement the *ITime-*

Navigatable interface (see [Figure 3.8](#)). The *TimeSeriesNavigator* control will use the *SetCurrentTimeSelection* method to notify the *ITimeNavigatable* that it needs to update its view. Because the time navigator can only notify one time-navigatable view, there is no generic way to update several views at the same time. However, the *CurrentTimeSelectionChanged* event can be used to which subviews can subscribe. This is used in the central map (*ProjectItemMapView*), for instance: this way, all layers will be notified that they need to update after the time has changed in the time navigator.

The time-navigatable object also has to give information to the time series navigator: which times should be available in the time navigator control? This is achieved using the *TimesChanged* event, which the *TimeSeriesNavigator* control will monitor.

4 Eventing

In large graphical applications, it can be painful to manage all types of events in such a way that all changes in the domain objects are correctly reflected in the views, without too much code repetition and performance degradation. Also, it is required that an undo/redo mechanism is present. In Delta Shell, the chosen solution is Aspect-Oriented Programming (AOP)¹.

4.1 PostSharp in Delta Shell

The implementation of AOP is accomplished by Postsharp². Although Postsharp 4.1 has been released at the time of writing of this document, Delta Shell uses version 2.1, that last version that was released under an open source license. That entails that not much documentation is available online³.

The basic idea behind Postsharp is to change the Microsoft Intermediate Language (MSIL) code *after* compilation. For instance, an extra method call or spawning of an event is triggered before or after a method that is decorated with a certain attribute. The theory behind this is that certain cross-cutting concerns (e.g. security, logging or eventing) is often intertwined in the code, but is highly repetitive, and can be automated.

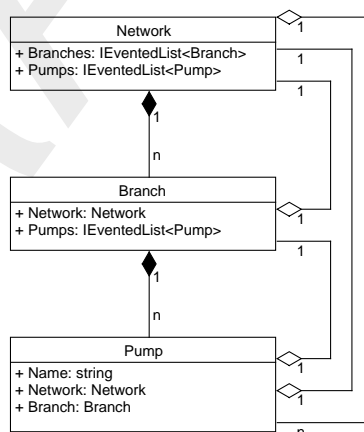


Figure 4.1: Simple example of three classes with composition and aggregation relations.

Delta Shell primarily uses Postsharp aspects to deal with eventing. For instance: how can a view be notified that something deep down in the object model has changed? In order to explain this, consider the example in Figure 4.1, where a part of the SOBEK object model is depicted. A network consists of several branches, and pumps are placed on these branches somewhere. Although the pumps are explicitly placed on one branch, the network also maintains a list of pumps. Both branches and pumps contain references to their respective parents. A solution without AOP would probably resort to creating custom events in the domain objects and maintaining many event subscriptions. The Delta Shell solution (which diverges a bit from the original Postsharp solution that was mentioned in the blog post) is shown in Figure 4.2.

The first thing that needs attention is the use of `[Entity]`⁴. It refers to the `EntityAttribute` class in

¹ A small introduction to AOP: https://en.wikipedia.org/wiki/Aspect-oriented_programming.

² See <http://www.postsharp.net> for more information.

³ Basically, the current implementation mostly hinges on [this blog post](#) from 2009. Documentation of PostSharp 2.x can be found at <https://www.postsharp.net/downloads/documentation/PostSharp-2.0.chm>.

⁴ Although the term 'Entity' is being used, it has nothing to do with Microsoft's Entity Framework, which is an

```

[Entity]
public class Network
{
    public IList<Branch> branches {get; set; }

    [Aggregation]
    public IList<Pump> Pumps {get; set; }
}

[Entity]
public class Branch
{
    [Aggregation]
    public Network Network { get; set; }

    public IList<Pump> Pumps { get; set; }
}

[Entity]
public class Pump
{
    public string Name { get; set; }

    [Aggregation]
    public Network Network { get; set; }

    [Aggregation]
    public Branch Branch { get; set; }
}

```

Figure 4.2: Eventing using the Delta Shell Postsharp solution, based on the class diagram in Figure 4.1.

the Delta Shell Framework and indicates that by default *all* properties in the decorated class should be emitting *PropertyChanged* and *CollectionChanged* events⁵. Consider the following two examples:

- ◇ Postsharp will have changed the setter of *Pump.Name*, resulting in a *PropertyChanged* event being emitted when the name of a pump has been changed.
- ◇ If a new *Pump* is added to a *Branch*, a *CollectionChanged* event will be sent. In this case, it is likely that the *Network* object will be listening to this event, in order to keep its own list of pumps synchronised. Whereas *PropertyChanged* is created automatically by Postsharp, *CollectionChanged* has been defined in Delta Shell, using *EventedList*.

To reduce the total number of events, the programmer is able to indicate whether the events will be sent or not. This is achieved by maintaining a difference between composition and aggregation. Details can be found [here](#), but the gist is: in the case of composition, object A *owns* object B, whereas in the case of aggregation, object A *uses* object B. Aggregated properties need to have the *[Aggregation]* attribute. In the Delta Shell Framework, it is expected that all domain objects that have been decorated with the *Entity* attribute, occur exactly once in the entire composition tree. In this case, it is clear that the *Network* owns the *Branches*, which in turn own the *Pumps*. Although every *Pump* knows the *Network* and *Branch* they belong to, the *Pump* does not own them.

object-relational mapping solution.

⁵Besides 'changed' events which are fired *after* the change, there are also 'changing' events, which fire *before* the change. These are however used a lot less frequently.

Although *[Aggregation]* is the preferred way to prevent events to be fired, there are some other ways, too, that may be necessary for some reason. You can disable *CollectionChanged* or *PropertyChanged* for an entire class by adding *[Entity(FireOnPropertyChange = false)]* or *[Entity(FireOnCollectionChange = false)]* to the class definition. Also, you can disable the events for individual properties by adding *[NoNotifyPropertyChange]* or *[NoNotifyCollectionChange]* to the property.

In order for the events to have effect, objects need to subscribe to the *PropertyChanged* and *CollectionChanged* events. This is shown in [Figure 4.3](#). The example also makes clear that proper unsubscribing is necessary when the value of a collection property is changed, in order to prevent event leaks. It is important to realize that when you assign a property that is a list itself, this will trigger a *PropertyChanged* event instead of a *CollectionChanged* event.

Emitting events is automatic. Also, the passing upwards ('bubbling') of these events is also automatic. In this example, it means that if *Network* is subscribed to *Branches*' *PropertyChanged* event, this event handler will be triggered whenever a property of *Pump* has been changed. *CollectionChanged* events will be bubbled in the same way. Eventually, this is how views in Delta Shell will be updated:

- 1 The user clicks in the central map (or changes a property in the property grid), which will trigger custom events.
- 2 The custom event will somehow change the object data.
- 3 The changes in the object data will trigger *PropertyChanged* and *CollectionChanged* events.
- 4 These events will bubble upwards (if necessary), whose changes will be reflected in the views that have subscribed to the *PropertyChanged* and *CollectionChanged* events.

Although most eventing in Delta Shell is done with the use of Postscript, this does not of course inhibit the use of custom events.

4.2 Undo/redo

Delta Shell uses the eventing mechanism to implement an undo/redo mechanism⁶. The user control that takes care of this mechanism is the *UndoRedoHistoryControl*. The main data structure for the undo/redo mechanism is the *UndoRedoManager*. The internal working of this instrument will not be discussed here, but it is important how to enable undo/redo in your code:

- ◇ Each change must start with a 'changing' and (unless cancelled) end with a 'changed' event. This is done automatically when a class is decorated with the *[Entity]* attribute.
- ◇ Each event must be received at the root node. In the case of Delta Shell, this is the *Project*.
- ◇ Restoring one property (undo), must not cascade and change other properties ('side-effects'). This is what the *[EditAction]* attribute signifies in [Figure 4.3](#).
- ◇ Each event must be received only once.

Sometimes, a user action has the consequence that several properties or collections are changed. This will result in a large list of actions that can be undone using the undo/redo mechanism. This can be quite clumsy to the user, and therefore a feature has been created to group a number of actions. For instance, when the user adds a new model, this will result

⁶Currently, this feature is only supported in the plugins D-Flow1D, D-RainfallRunoff and D-RealTimeControl. This feature is therefore only available if the project solely contains models from these three; otherwise is turned off. In the future, this feature might be dropped and instead, a macro recorder will be used, so that after a number of edits by the user, we can see what equivalent Python code is inserted.

in a cascade of events, but when the following code pattern is followed, the user will only see one action in the undo history.

```
public class GuiCommandHandler : IGuiCommandHandler
{
    public IModel AddNewModel()
    {
        gui.Application.Project.BeginEdit("Add new model: " + newModel.ToString());

        var newModel = CreateNewModelUsingDialog(gui.SelectedProjectItem);
        Models.Add(newModel);

        if(!ReferenceEquals(newModel, items.LastOrDefault()))
        {
            cancel = true; // Apparently, the operation has failed...
        }

        if(cancel)
        {
            gui.Application.Project.CancelEdit(); // So cancel the operation.
        }
        else
        {
            gui.Application.Project.EndEdit(); // Round off the operation.
        }
    }
}
```

The use of *BeginEdit* and *EndEdit* can also be a very good tool to improve the performance of operations that incur many events. When *BeginEdit* is called, the *IsEditing* property is set to *true*. *EndEdit* sets it to *false* again. The object that is listening to the changes, can therefore decide to postpone updating itself until *EndEdit* has been called. Therefore, you can often see code snippets as follows in the listeners:

```
private void MeteoDataPropertyChanged(object sender,
                                     PropertyChangedEventArgs e)
{
    if (ReferenceEquals(sender, meteoData))
    {
        if (e.PropertyName == "DataDistributionType" ||
            (e.PropertyName=="IsEditing" && !meteoData.IsEditing))
        {
            SetMeteoDataView();
            SetMeteoDataTypeComboBox();
        }
    }
}
```

In this example, the views will update immediately when either *DataDistributionType* is changed or when the *IsEditing* is set to false (during the *EndEdit* call). Otherwise, no updates will be performed.

4.3 Accessing the GUI thread from a worker thread

In some cases, Delta Shell uses a worker thread to do something in the background. For instance, the `ActivityRunner` will pop up a progress screen when an importer is doing its work. Normally, when the worker thread would attempt to update the progress bar text, a `CrossThreadException` would be thrown. Windows Forms has an elegant way to overcome this barrier: `ISynchronizeInvoke`. All Windows Forms controls implement this interface, and the worker thread can reach the main UI thread using this interface. More on this topic can be found [in this article](#).

In Delta Shell, this is automated using AOP as well. The `InvokeRequiredAttribute` can be used to indicate that a certain method will attempt to operate on data that is owned by a different thread. An example usage of `[InvokeRequired]` is documented in [Figure 4.4](#). In this example, the `GuiImportHandler` is owned by the GUI thread, and the `ConfigureImporterAndRun` method will also be executed in the main thread. However, the importer that it starts is running asynchronously (in a worker thread). When the import ends, the worker thread will trigger the event handler. The event handler will run in the worker thread, but will change data in the main thread. The `[InvokeRequired]` will make sure that the execution of the event handler will be passed on to the GUI thread before it is executed, preventing a `CrossThreadException`. However, using `InvokeRequired` incurs a large performance penalty.

```
[Entity]
public class Network : EditableObjectUnique<long>, INetwork
{
    public virtual IEventedList<IBranch> Branches
    {
        get { return branches; }
        set
        {
            if (Branches != null)
            {
                Branches.CollectionChanging -= BranchesCollectionChanging;
            }

            branches = value;

            if (Branches != null)
            {
                Branches.CollectionChanging += BranchesCollectionChanging;
            }
        }
    }

    private void BranchesCollectionChanging
        (object sender, NotifyCollectionChangingEventArgs e)
    {
        if (Equals(sender, Branches) // Bubbled CollectionChanged from Pumps
            // are now discarded.
        {
            privateField = 3 // Some non side-effect logic here.
            OnBranchesCollectionChanging(sender, e); // All side-effect logic
                                                    // in an EditAction method.
        }
    }
}

[EditAction]
private void OnBranchesCollectionChanging
    (object sender, NotifyCollectionChangingEventArgs e)
{
    switch (e.Action)
    {
        case NotifyCollectionChangeAction.Add:
            var branch = (IBranch) e.Item;
            branch.Network = this; // side-effect
            break;
        case NotifyCollectionChangeAction.Remove:
            // more side-effect logic here.
    }
}
```

Figure 4.3: This code snippet shows how to subscribe to Postsharp-generated events and how to distinguish 'normal' effects from side effects.

```
public class GuiImportHandler
{
    private void ConfigureImporterAndRun(IFileImporter importer,
                                        IProjectItem importedItemOwner,
                                        object target)
    {
        var importActivity = new FileImportActivity(importer, target)
        {
            ImportedItemOwner = importedItemOwner
        };
        importActivity.OnImportFinished += ImportActivityOnImportFinished;
        ActivityRunner.Enqueue(importActivity);
    }

    [InvokeRequired]
    private void ImportActivityOnImportFinished(FileImportActivity fileImportActivity,
                                                object importedObject,
                                                IFileImporter importer)
    {
        if (importer.OpenViewAfterImport)
        {
            gui.Selection = importedObject;
            gui.CommandHandler.OpenViewForSelection();
        }
    }
}
```

Figure 4.4: Code snippet to illustrate the usage of `InvokeRequired`.

DRAFT

5 Persistence

An important part of Delta Shell's functionality is its ability to store and load projects. We call this mechanism *persistence*: it gives a project a lifetime beyond that of the application. Persistence within Delta Shell is not easy, unfortunately, and needs an entire chapter to be discussed in enough detail. Still, this chapter will not be sufficient for those who want to make changes to the persistence mechanism, but hopefully, it will prove helpful for those who want to create a plugin with persistent data.

5.1 History of persistence in Delta Shell

Historically¹, the models that are now incorporated under the Delta Shell plugin umbrella, were persisted using a set of files. Besides a few binary formats, such as the .HIS and .MAP files, these files were human-readable, and modellers have been able to modify these files in text editors. With the introduction of Delta Shell, the original intention was to create an all-encompassing object-relational mapping (ORM) solution for persistence. At first, Microsoft's Entity Framework was used, but this turned out to be insufficient². In the end, nHibernate (the C# port of Hibernate, a Java ORM framework) was chosen as persistence mechanism. All data of a Delta Shell project is now stored in an SQLite database³, a file with the extension *.dsproj*.

For some, this solution was insufficient. Some users are accustomed to the fact that they could edit the model definition files themselves, which became nearly impossible in this setup. From a developer's perspective, getting the persistence right, including backwards compatibility, turned out to be fairly complicated.

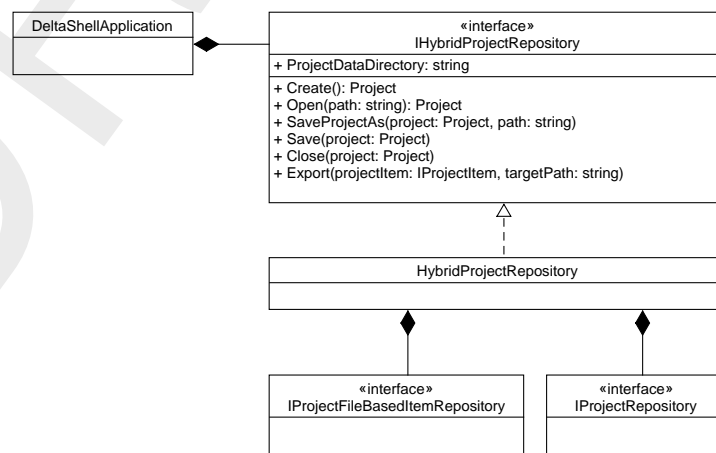


Figure 5.1: Class diagram illustrating how a Delta Shell project contains both ORM and file-based persistence.

Therefore, a file-based persistence mechanism was created in parallel. The basis of a Delta Shell project is still the nHibernate database, but the project might refer to files that are stored in the data directory of the project. This has the result that persistence in Delta Shell is actually a hybrid of database storage and file storage.

Both types of storage will be discussed in this chapter, in separate sections. In Figure 5.1, a

¹For the record: I do not want to impose an opinion on this topic in this section, but rather give an overview that is as factual as possible.

²Not to be confused with the *Entity* attribute that is used for eventing.

³<https://www.sqlite.org/>

very rough insight is given into where in Delta Shell both types of persistence are located. In *IApplication*, a *HybridProjectRepository* is defined. This contains both an *IProjectRepository* (NHibernate) and an *IProjectFileBasedItemRepository* (file-based).

5.2 Persistence with nHibernate

In order to understand this section, it is necessary to have a basic understanding of what Object-Relational Mapping (ORM) is, and to have read a tutorial on nHibernate⁴.

5.2.1 Overall set-up

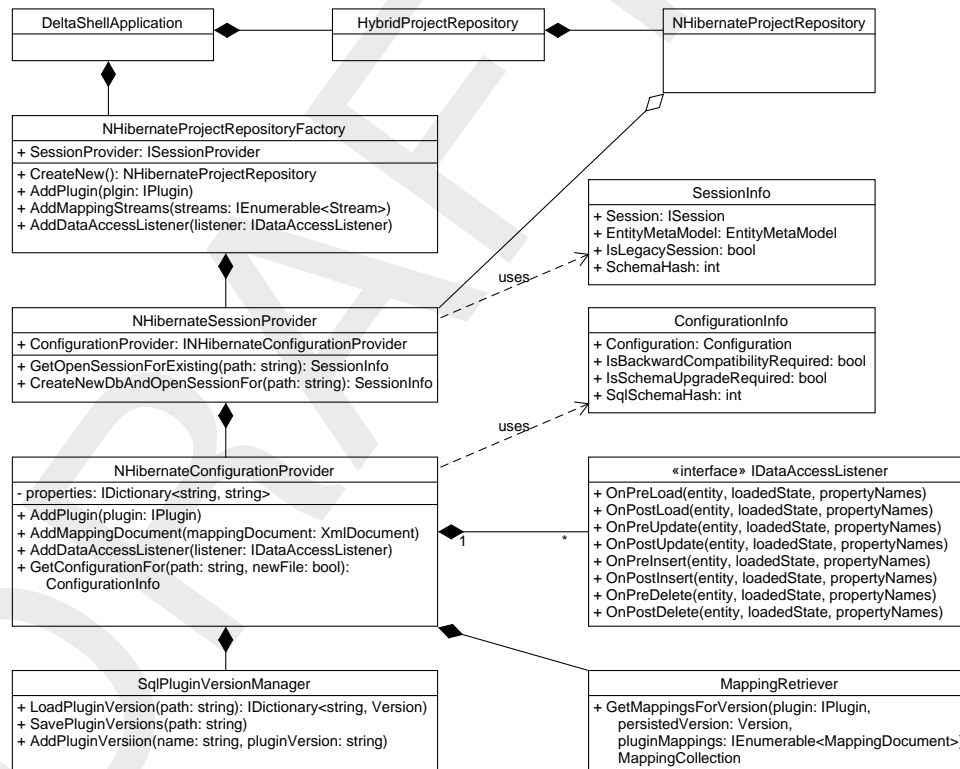


Figure 5.2: Class diagram illustrating how the nHibernate infrastructure is encapsulated in the *DeltaShellApplication* object. In order to keep this picture understandable, some interface classes have been omitted.

In order to use nHibernate in a C# project, there is still some code necessary in the project. In Delta Shell, this code is concentrated in an assembly called *DeltaShell.Plugins.Data.NHibernate*. This is implemented as an *ApplicationPlugin*.

The encapsulation of the nHibernate infrastructure into the Delta Shell Application is depicted in Figure 5.2. The application always has one project, and this project is represented on disk by a project repository. In order to create a project repository, the application owns a project repository factory. A project repository maintains a path to the current project (the .dsproj) but does *not* renew itself when a new project is created or opened. Therefore, the project repository factory will create only one project repository during the lifetime of the application⁵.

⁴A good candidate for starting is: <http://nhibernate.info/doc/nhibernate-reference/>. Another good tutorial is <http://www.codeproject.com/Articles/21122/NHibernate-Made-Simple>. It might be good, however, to realise that Delta Shell currently uses version 2.0 of nHibernate, whereas the latest version is 4.1, at the time of writing.

⁵This appears to be a design flaw. This could be simplified by not using a factory at all.

Roughly, the project repository, its factory and its subcomponents as shown in Figure 5.2 have the following functionalities:

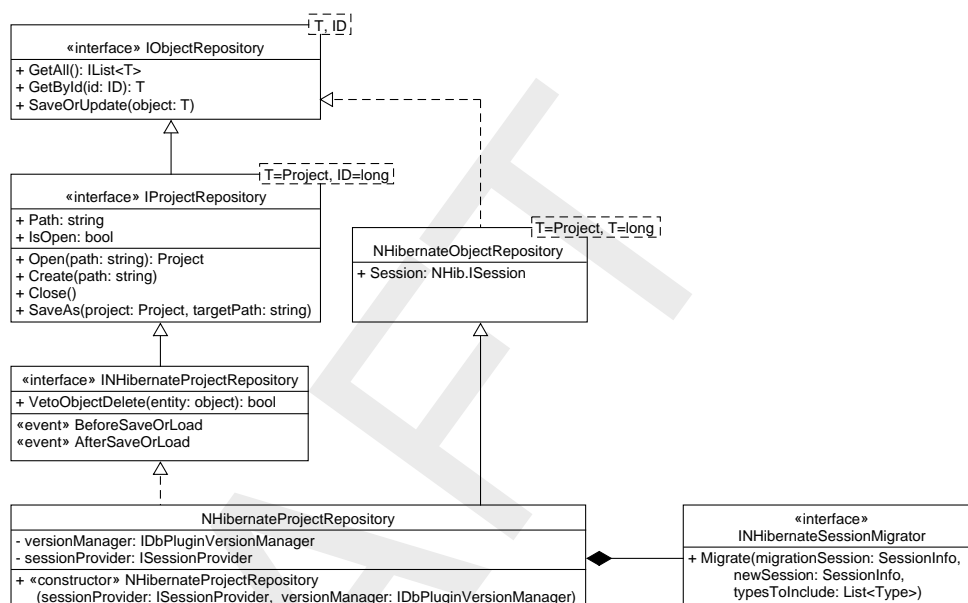


Figure 5.3: Class diagram of *NHibernateRepository*, one of the central classes in *Delta Shell*.

- ◇ Creating and loading, saving and closing projects. *NHibernateProjectRepository* contains methods for these functionalities.
- ◇ Lazy loading of objects. When a project is loaded, not all objects are immediately read from the database and converted into data objects. Instead, *proxy objects* are created, and when the code requests the contents of that object, it is unproxied. This can greatly enhance the efficiency of loading a large project.
- ◇ Creating and disposing NHibernate sessions. A *session* is a fully configured connection of NHibernate to a database, including its configuration (e.g. mappings). It is represented by the *NHibernate.ISession* interface, implemented by *NHibernate.Impl.SessionImpl*. In Delta Shell, sessions are wrapped in *SessionInfo* objects, which indicate whether the session is a legacy session, for instance.
- ◇ Creating and disposing NHibernate configurations. An NHibernate configuration has knowledge of all mappings (the .hbm.xml files), and also configures the way NHibernate connects to the database. The latter can include batch sizes, driver type (in our case SQLite) etc. See also the *NHibernateSqlLiteConfigurationProperties* class for some set properties.
- ◇ Keeping track of plugin versions that are loaded into the running version of Delta Shell. When a project is saved, a separate table is written to the database that indicates with which version of the plugins the database has been written (see the *PluginVersion* table in the database⁶).
- ◇ Migrating an old project to the current format. Every time a project is loaded, Delta Shell will check, for every plugin that is contained in the database, whether its format is the same as the plugin version that is loaded in the application. Delta Shell determines this on the basis of the *FileFormatVersion* property in *IPlugin*. If the file format versions are different between the loaded plugin version and the persisted plugin version, there are three possibilities:

- 1 The saved plugin version is newer. In this case, the project will not be loaded (Delta Shell does not support forward compatibility).

⁶There are a number of free SQLite database viewers/editors available, such as SQLiteBrowser.

- 2 The saved plugin version is older. In this scenario, the session migrator (see [Figure 5.3](#)) will make sure that the database will be properly translated into the new format.

- ◇ Data access listeners provide the possibility to subscribe to certain persistence events. For instance, it is possible for an application plugin to implement the *IDataAccessListenerProvider* interface. If so, when the application loads the plugin, this will be detected, and the plugin's data access listeners will be added to the application's collection of data access listeners. The effect is that whenever an object is persisted or loaded (and some more events), the data access listener will be notified. This can have quite an effect on the performance, because this will be done for every single write and read action to the database.

5.2.2 Mapping files

The mapping files are a crucial part in the persistence mechanism of Delta Shell. In principle, NHibernate's own documentation should cover this topic sufficiently. Some 'how-to' question do come up regularly, however. These are therefore treated in the sections below.

5.2.3 How to enable or disable lazy loading

In some cases, projects can be so large that it hampers the usability of the software if the entire project is loaded into memory. Therefore, a mechanism called *lazy loading* is set up. This is a standard feature in NHibernate, and it is enabled by default. You can find more documentation on this feature at, for instance, <http://nhibernate.info/doc/howto/various/lazy-loading-eager-loading.html>. In other words, if you would like to eagerly load parts of the project, you have to explicitly turn this on in a mapping file. <http://ayende.com/blog/4573/nhibernate-is-lazy-just-live-with-it> shows how you can do it, and also why it can be dangerous.

5.2.4 How to solve: sqlite - max 64 tables in join

From time to time the following error occurs in DeltaShell: 'SQLiteException occurred. SQL logic error or missing database. At most 64 tables in a join.'. This indicates that NHibernate is attempting to fetch data from too many different tables at the same time. Note that it doesn't matter at all if there is any data in those tables, all that matters is how big the biggest class hierarchy is.

That hierarchy usually grows if you have many plugins loaded, for example many plugins have *ProjectItems*, so the *ProjectItem* hierarchy can get pretty big! NHibernate by default attempts to load all objects at once, but as the data is spread over many tables it uses SQL JOINS to do so. This is fine, up to the point where it gets above (or equal) to 64 joins, as that is the limit of the database driver: *SQLite*.

To prevent NHibernate from joining so many tables together, you have to add instructions in the mapping files to prevent this. Unfortunately there doesn't seem to be a simple permanent fix, and instead as the relationships change and grow, this problem resurfaces every few months, sometimes hidden and sometimes as a big exception whenever you try to load any *dsproj*. Do note however that in production environments this issue is much less likely to occur, as usually only a few plugins are shipped in a single install.

The best solutions appears to be to set (non-lazy) relationships to `fetch=select`, for example:

```
<many-to-one name="CalculationSelection"
```



```
lazy="false"  
fetch="select"  
cascade="all-delete-orphan"  
class="DeltaShell.Plugins.MorphAn.Models.CalculationSelection"  
column="calculation_selection_id"/>
```

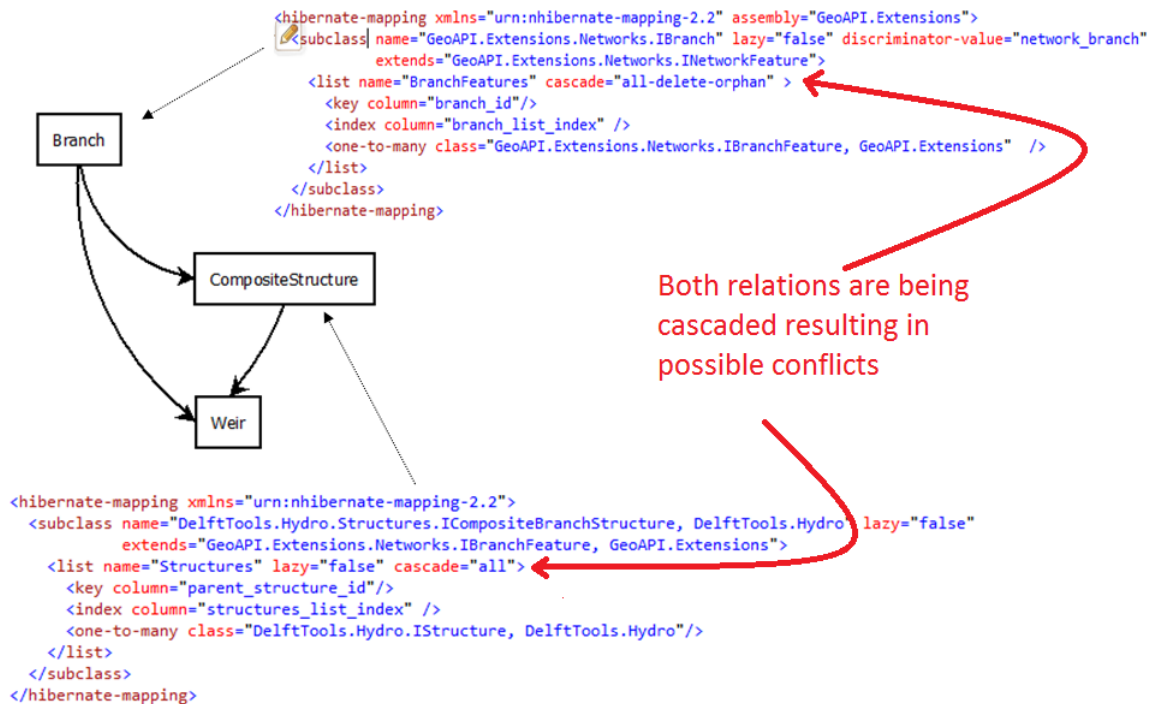
Recommended workflow:

- 1 Turn 'just my code' off just prior to opening an offending dsproj.
- 2 Determine the extend of the problem: copy the sql query causing the exception (part of command object) into a text editor
- 3 Count the number of occurrences of the string 'JOIN' in the file (as a baseline) -> typically between 64 and 100
- 4 Now the hard part: scan through that query and identify plugins / classes that seem to be occurring a lot, or combinations you wouldn't expect
- 5 Apply fetch=select on one or more (non-lazy!) relationships where you think it would reduce the number of joins (see revision 21942 as an example)
- 6 Reload the project; if the exception still occurs check if the number of JOIN has actually lessened (using text editor)
- 7 If not, revert, if it has, find another relationship to change.
- 8 Continue until no exception (preferably a bit longer: eg <40 joins!)

5.2.5 How to use cascades

Sometimes, the following error occurs: *deleted object would be re-saved by cascade (remove deleted object from associations)*. A very common cause is the cascade rules in the mapping as they require close attention. An example of a 'wrong' mapping was the mapping of composite branch structures (CS) (the mapping is shown below). The CS has a list of child structures and these were persisted using a cascade='all'. The same structures were also in the list of branchfeatures on the branch and these were mapped using cascade=all-delete-orphan. So the same weir was saved via branch and via the composite structure. This is fine as long as the cascade don't conflict.

Sometimes a conflict occurs when one cascade results in a delete and another in a save. For example a composite structure is deleted and the cascade deletes all child-structures. But one of these structures is still in the branchfeatures collection of a branch. The branch cascade insists on the object being saved (or at least not deleted) and the CS cascade wants to delete. Hence a conflict. This can be fixed by removing the structure from both lists or downgrading/removing one of the cascades. In this case the branch should be responsible for saving the features and the CS should have a list of child structures without cascades.



A quick recap of the different cascades :

- ◇ **none**: do not do any cascades, let the users handles them by themselves.
- ◇ **save-update**: when the object is saved/updated, check the associations and save/update any object that require it (including save/update the associations in many-to-many scenario).
- ◇ **delete**: when the object is deleted, delete all the objects in the association.
- ◇ **delete-orphan**: when the object is deleted, delete all the objects in the association. In addition to that, when an object is removed from the association and not associated with another object (orphaned), also delete it. Orphan only checks one relation.
- ◇ **all** when an object is save/update/delete, check the associations and save/update/delete all the objects found.
- ◇ **all-delete-orphan** when an object is save/update/delete, check the associations and save/update/delete all the objects found. In addition to that, when an object is removed from the association. (orphaned) delete it. So 're-parenting' (changing parent) is a problem.

A note about orphan cascade: Orphan only looks at one instance at a time. So a composite structure might have a list of child structures. When a child structure is removed from the CS it is considered an orphan. NHibernate does not care about other relation the structure might have (for example with branch or a new CS). More information about reparenting can be found at <http://fabioaulo.blogspot.com/2009/09/nhibernate-tree-re-parenting.html>.

5.2.6 How to create a legacy mapping for a class where one property has changed

Let's assume we have a *Product* class, with the following fields: Category, Id, Name, Available. However, it was decided that it was more logical to use a 'Discontinued' flag. This is not just a simple rename! We need to do something equivalent to 'Discontinued = !Available' when loading products from the old database. Specifically, we need to write a hibernate HBM mapping file which reads the old database format into the new object model.

The original HBM looks something like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2" assembly="..." namespace="...">
  <class name="Product">
    <id name="Id"> <generator class="guid" /> </id>
    <property name="Name" />
    <property name="Category" />
    <property name="Available"/>
  </class>
</hibernate-mapping>
```

The new HBM looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2" assembly="..." namespace="...">
  <class name="Product">
    <id name="Id"> <generator class="guid" /> </id>
    <property name="Name" />
    <property name="Category" />
    <property name="Discontinued"/>
  </class>
</hibernate-mapping>
```

To migrate from the old database to the new object model, we must create another, special mapping. This mapping will only be used for loading, not for saving, so we can use some special features like embedding SQL queries into the HBM. Let's start by writing the basics. We can map most properties 1-to-1, as you can see here:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2" assembly="..." namespace="...">
  <class name="Product">
    <id name="Id"> <generator class="guid" /> </id>
    <property name="Name" />
    <property name="Category" />
    <property name="Discontinued" formula="( Available = 0 )"/>
  </class>
</hibernate-mapping>
```

We must keep in mind that we can only map to new properties, so we cannot have a property with name="Available" here. The 'Available' value is however still there in the database, so in this case we can solve it by using the 'formula' field inside a property tag, to insert SQL queries and logic to get values from the database. To understand why this works, it is important to

understand how NHibernate processes these formulas. NHibernate inserts the formula as a subquery into the larger SQL. This is the cleaned-up version of what NHibernate generates:

```
SELECT product.Id, product.Name, product.Category, ( product.Available = 0 )
FROM Product product WHERE product.Id=...
```

Since 'Available' is a boolean value stored as integer, we do a '=0' compare to inverse the boolean value. Also notice that NHibernate will insert 'product.' in front of 'Available', since it recognizes it as a column in the database.

5.2.7 How to create a legacy mapping for a class that has been split up

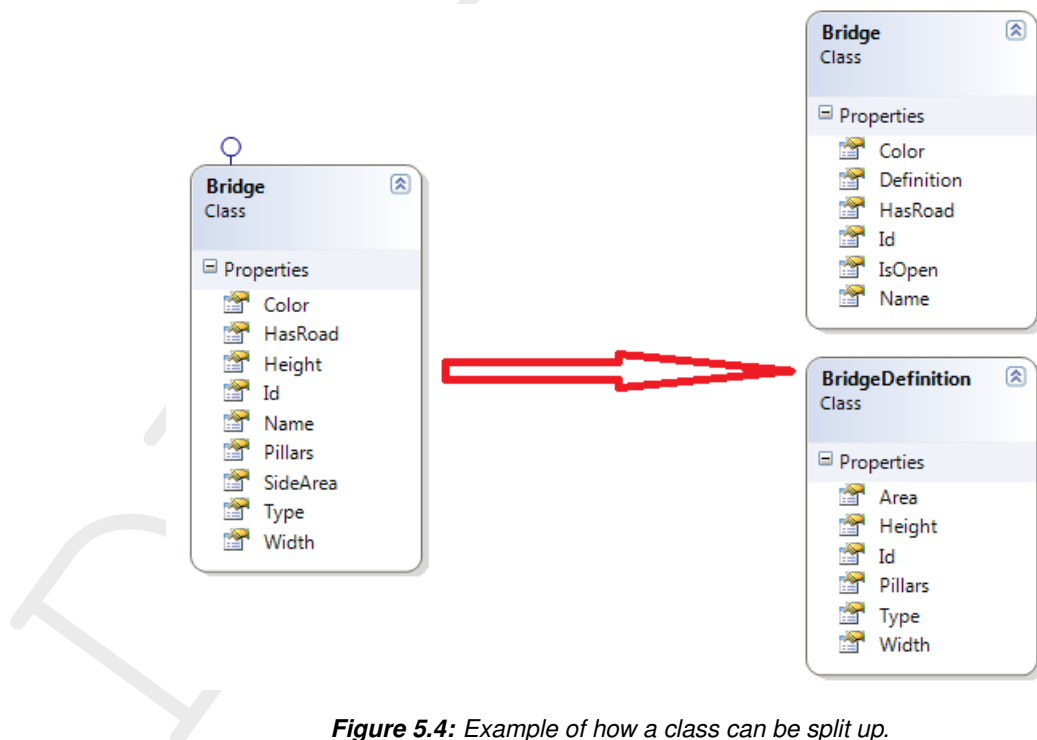


Figure 5.4: Example of how a class can be split up.

Consider the situation in [Figure 5.4](#). Originally there was one big Bridge class, but later it was decided to split it into a Bridge and a BridgeDefinition. Also the properties don't exactly match:

- ◇ SideArea: no longer exists
- ◇ Pillars -> renamed to NumPillars
- ◇ Color: should always be 'RED'
- ◇ IsOpen: new

The old HBM is straightforward:

```
<class name="Bridge">
  <id name="Id"> <generator class="guid" /> </id>
  <property name="Name" />
  <property name="Color" />
  <property name="HasRoad"/>
  <property name="Width"/>
  <property name="Height"/>
```

```

    <property name="SideArea"/>
    <property name="Type"/>
    <property name="Pillars"/>
  </class>

```

The new HBM has now the following definition (there might be several bridges that point to the same definition):

```

<class name="Bridge">
  <id name="Id"> <generator class="guid" /> </id>
  <property name="Name" />
  <property name="Color" />
  <property name="HasRoad"/>
  <property name="IsOpen"/>
  <many-to-one name="Definition" cascade="all-delete-orphan"/>
</class>

<class name="BridgeDefinition">
  <id name="Id"> <generator class="guid" /> </id>
  <property name="Type"/>
  <property name="Width"/>
  <property name="Height"/>
  <property name="NumPillars"/>
</class>

```

We need to tackle four property changes and the class split itself. Let's start with the property changes:

Property	Change	Solution
SideArea	No longer exists	Do nothing!
Pillars	Renamed to NumPillars	<pre> <property name="NumPillars" formula="Pillars"/> </pre>
Color	Should always be 'RED' when loading old data	<pre> <property name="Color" formula="'RED'"/> </pre>
IsOpen	New property, should always be 'true' when loading old data	<pre> <property name="IsOpen" formula="1"/> </pre>

Finally we need to tackle how to split the class. Fortunately NHibernate has something called components, where it persists two classes into one table. We just tell NHibernate to treat the table as such a table, with BridgeDefinition being a 'component' of Bridge and NHibernate will load the single table into two entities, just like we want. The resulting HBM looks like this:

```

<class name="Bridge">
  <id name="Id"> <generator class="guid" /> </id>
  <property name="Name" />
  <property name="Color" formula="'RED'"/>
  <property name="HasRoad"/>
  <property name="IsOpen" formula="1"/>

  <component name="Definition" class="BridgeDefinition">
    <property name="Type"/>

```

```

    <property name="Width"/>
    <property name="Height"/>
    <property name="NumPillars" formula="Pillars"/>
  </component>
</class>

```

Note that the Id property of the BridgeDefinition component is not mapped. However when saving this class in the new session (with the new hbm's), it will receive an Id anyway.

The resulting SQL:

```

SELECT bridge.Id, bridge.Name, bridge.HasRoad, bridge.Type, bridge.Width,
       bridge.Height, 'RED', 1, bridge.Pillars
FROM Bridge bridge
WHERE bridge.Id=...

```

Note that the splitting into a component is handled by NHibernate in code and has no effect on the SQL.

5.2.8 How to create a legacy mapping for a class that has been merged

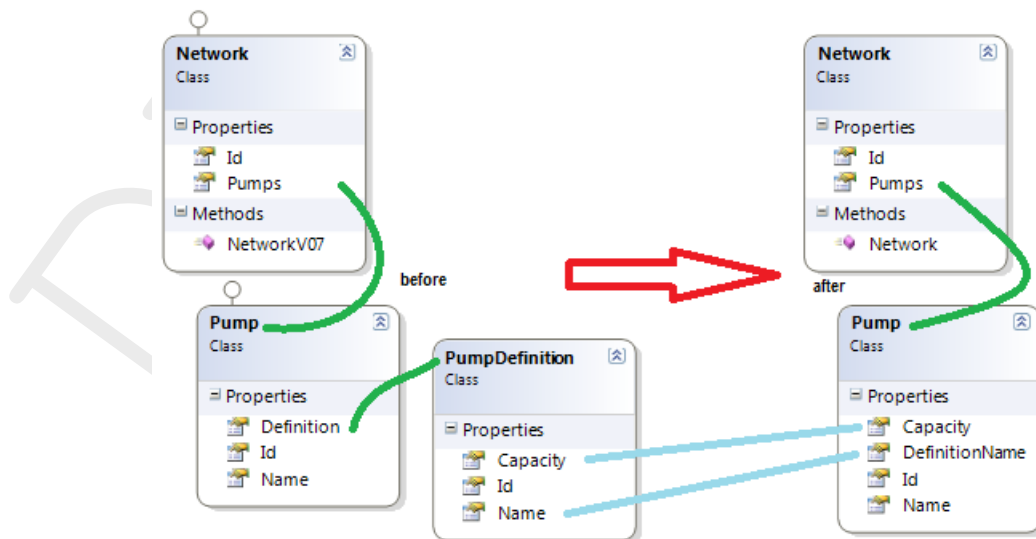


Figure 5.5: Example of how two classes can be merged.

The following situation we are going to look at is what should happen when two classes are merged. Let's consider the example in [Figure 5.5](#). We should have no trouble maintaining the Pumps list in Network; nothing changed there. So we just need to retrieve the Pump properties DefinitionName and Capacity from another table: pump_definition.

Let's start with the migrating HBM, which is almost the same as the new HBM:

```

<class name="Network">
  <id name="Id"> <generator class="guid" /> </id>
  <bag name="Pumps" cascade="all-delete-orphan">
    <key column="PumpId"/>

```

```

    <one-to-many class="Pump"/>
  </bag>
</class>

<class name="Pump">
  <id name="Id">
    <generator class="guid" />
  </id>
  <property name="Name" />
  <property name="DefinitionName" formula=
    "( SELECT def.Name FROM pump_definition def WHERE def.Id = Definition)"/>
  <property name="Capacity" formula=
    "( SELECT def.Capacity FROM pump_definition def WHERE def.Id = Definition)"/>
</class>

```

The two most interesting parts are how DefinitionName and Capacity have been defined. Using the formula field and simple SQL we can retrieve these fields from the pump_definition table, so we just need to match the Ids. The resulting cleaned-up SQL looks like this:

```

SELECT pump.NetworkId, pump.Id, pump.Id, pump.Name,
      ( SELECT def.Name FROM pump_definition def WHERE def.Id = pump.Definition),
      ( SELECT def.Capacity FROM pump_definition def WHERE def.Id = pump.Definition)
FROM Pump pump
WHERE pump.NetworkId=...

```

5.2.9 How to create a legacy mapping for a class that has been changed to a hierarchy of classes

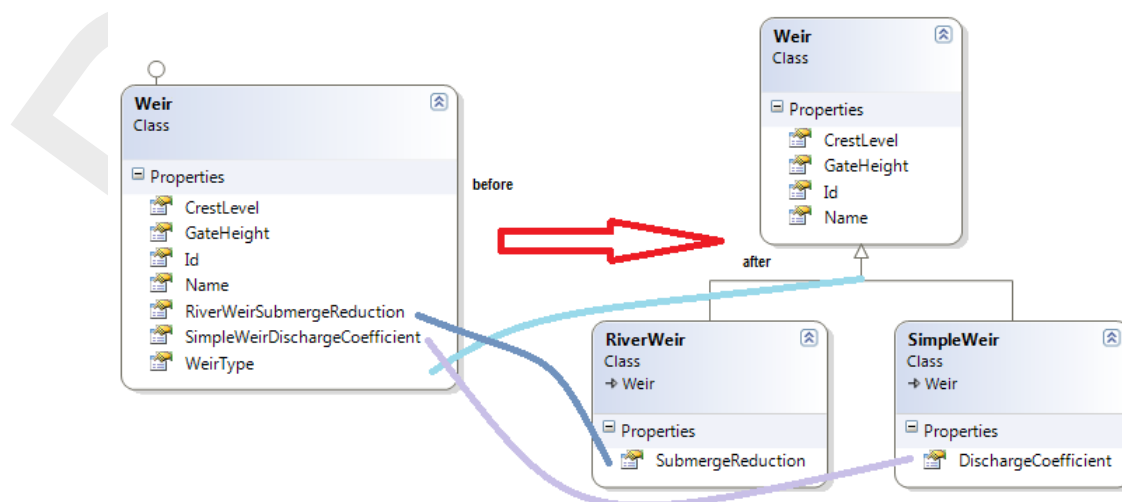


Figure 5.6: Example of how a class can be split up into one base class and two derived classes. Some of the properties in the original class need to be divided over the new derived classes.

Consider the migration in ???. We need to map specific properties for specific subclasses, but the biggest challenge is how to tell NHibernate which subclass to instantiate on load. In the old situation to differentiate between a RiverWeir and a SimpleWeir, one had to check the WeirType (string) property.

In the past, WeirType could contain the following strings:

- ◇ simple_weir
- ◇ river_weir
- ◇ advanced_river_weir

We want NHibernate to use this property to decide which subclass to create, and we want to map both 'river_weir' and 'advanced_river_weir' to the RiverWeir class. Let's first assume we don't have this 'advanced_river_weir'. In that case we can use the normal discriminator system of NHibernate (note: discriminator must come after Id!). We also map the specific properties while we're at it:

```
<class name="Weir">
  <id name="Id">
    <generator class="guid" />
  </id>

  <discriminator column="WeirType"/>

  <property name="Name" />
  <property name="CrestLevel" />
  <property name="GateHeight" />

  <subclass name="SimpleWeir" discriminator-value="simple_weir" >
    <property name="DischargeCoefficient" formula="SimpleWeirDischargeCoefficient" />
  </subclass>

  <subclass name="RiverWeir" discriminator-value="river_weir" >
    <property name="SubmergeReduction" formula="RiverWeirSubmergeReduction" />
  </subclass>
</class>
```

Now let's look at the more complex situation with 'advanced_river_weir' included. Again, formula comes to the rescue, this time in 'discriminator'. This way you can influence the value of what NHibernate sees as the discriminator value. The possible discriminator values you return must match with the discriminator-values you supply for each subclass:

```
<class name="Weir">
  <id name="Id">
    <generator class="guid" />
  </id>

  <discriminator formula="case when WeirType in ('river_weir', 'advanced_river_weir')
                                then 'RiverWeir' else 'SimpleWeir' end"/>

  <property name="Name" />
  <property name="CrestLevel" />
  <property name="GateHeight" />

  <subclass name="SimpleWeir" discriminator-value="SimpleWeir" >
    <property name="DischargeCoefficient" formula="SimpleWeirDischargeCoefficient" />
  </subclass>

  <subclass name="RiverWeir" discriminator-value="RiverWeir" >
    <property name="SubmergeReduction" formula="RiverWeirSubmergeReduction" />
  </subclass>
</class>
```


Note that I have changed the discriminator-value of both subclasses and also in the discriminator formula, to indicate you can choose any.

5.2.10 How to deal with class renames or changes of namespace

Whenever you put an object in a DataItem, NHibernate figures out how to save this 'Value' entity using what is called an Any mapping. As part of this Any mapping mechanism, NHibernate stores the (CLR, .NET) type of the object in a column in the database as fully qualified string. Also sometimes we map types directly, for example ValueType of a dataitem. This is all fine, until you rename a class...

When NHibernate tries to convert the string back into a .NET type, it will fail and crash. Legacy mappings won't help you there. To solve this issue, we have added a custom HBM (meta)attribute in DeltaShell: `oldClassName`. The value of this attribute is used as an alias when resolving types. This happens automatically for the any mapping of dataitem, but if you have mapped a Type column/property directly, make sure to load it using `TypeUserType` in your legacy mapping.

Example: Your old class 'Branch' has been renamed to 'Channel'. Now in your legacy mapping you do the following:

```
<class name="Channel" table="branch" lazy="false">
  <meta attribute="oldClassName">Domain.Branch, Domain</meta>
  <id name="Id" column="Id" type="Int64" unsaved-value="0">
    <generator class="native" />
  </id>
  ...
</class>
```

The value of the 'oldClassName' attribute is the string as it appears in the old database, typically 'namespace.class, assembly'.

To summarize: if you rename a persistent class, and there is any chance the object has ever been used as DataItem value, parameter, variable, filter, or had its type mapped directly for any other reason: make sure to add a meta attribute with the old class alias.

5.3 File-based persistence

Although most persistence is done with NHibernate, some items are represented by separate files. A good example of this are restart files, which are usually written as ZIP files. Also separate GIS files (shapefiles) or some databases are kept in separate files. Besides these, some models are for the most part written as files, such as Flexible Mesh models and Waves models. All object data are represented by the *IFileBased* interface (see [Figure 5.7](#)). It is vital that these objects are in a composition relation to the *Project*. The repository listens to the *Project*, and synchronises the list of all *IFileBased* items in the project. The main goal of this synchronisation is that the file-based project repository will make sure that the changes to the file-based items in the project are reflected on disk. The most important functionalities of this class are therefore:

- ◇ When a new file-based item (i.e. it implements *IFileBased*) is added to the project, the file-based project repository takes care that this file is indeed added to the correct directory

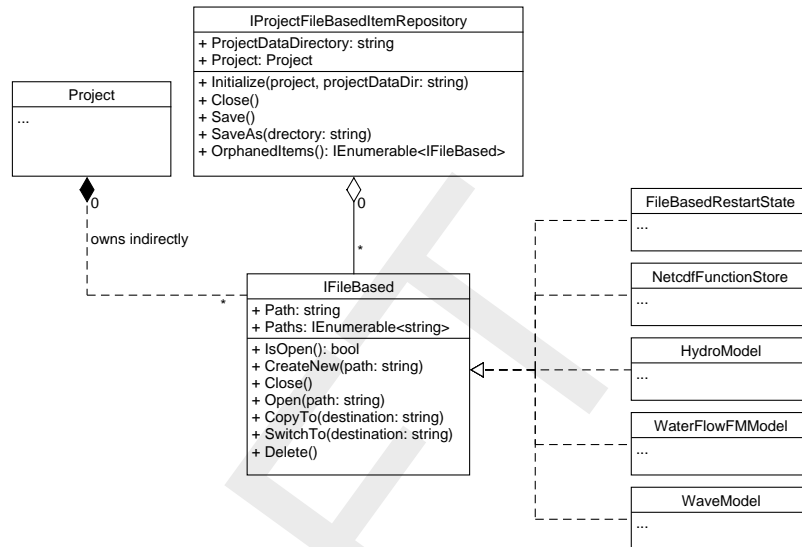


Figure 5.7: Class diagram describing file-based persistence. The given classes on the right of the diagram are examples: more classes implement the **IFileBased** interface.

inside the data directory of the project. Similarly, when a file-based item is removed, the file will also be deleted from the data directory of the project. This behaviour is triggered by the *CollectionChanged* event. In case the filename of a file is changed, this is also picked up using the *PropertyChanged* event.

- ◇ When changes are made to a file-based object and the project is saved, the changes will indeed percolate to the file that is persisted on disk.
- ◇ When the project is saved into a different location, the file-based project repository makes sure that the file-based items of the project are located inside the data directory

6 Geospatial data and visualisation

One of Delta Shell's main functionalities is its ability to work with geospatial data. Therefore, the central map is a key component in the user interface. For instance, the user can draw model schematisations, import data in different coordinate systems and visualise real maps downloaded from external sources. In the Delta Shell implementation, there are a number of open source libraries that are used to achieve these functionalities¹:

- ◇ GeoAPI: a set of interfaces that defines geometries and geometric operations, but contains no implementations of these interfaces. This library is discussed in [section 6.1](#). The current version of this C# library can be downloaded from <https://github.com/NetTopologySuite/GeoAPI>. This library is not to be confused with the Java GeoAPI interface, which bears no relation with the C# variant.
- ◇ NetTopologySuite: this library provides an implementation of the GeoAPI interface. It can be found at <https://github.com/NetTopologySuite/NetTopologySuite>.
- ◇ SharpMap: a set of that provide the mapping and drawing facilities.

6.1 GeoAPI

6.2 NetTopologySuite

6.3 SharpMap

GeometryDll directly to FM implementation.

6.4 The central map and its panel

6.5 Spatial operations

6.5.1 Introduction

One of the key features of DeltaShell is –or at least should be– extensive support for manipulating and preprocessing geospatial data. The spatial operation functionality in the framework attempts to achieve this goal for data defined on spatial or network discretizations (grids)² or point clouds. The former data is usually captured within a (time-independent) *coverage*³ in DeltaShell. Spatial operations in DeltaShell fulfill the following extra requirements:

- ◇ Reproducible manipulation of input data: all operations that a user performs are recorded as objects and added to a stack such that all steps that lead to the final result can be reproduced.
- ◇ Grid independence: no single operation explicitly depends upon grid indices, all geometric input data is projected onto the grid at each operation execution.
- ◇ Transparency: within an operation stack it is possible to visualize all intermediate results.
- ◇ Flexibility: within an operation stack it is possible to change properties of any operation and re-evaluate with the new settings. It is generally possible to remove or disable operations from an existing stack.
- ◇ When working with large data sets, performance and responsiveness is more important than consistency at all times, which is why a 'refresh' button has been introduced.

Although the request for the spatial data editing functionality is strongest among FM users, it was decided to adopt spatial operations in the DeltaShell framework, such that all plugins that

¹ Of these packages, out-of-date versions are used.

² some info about the type of grids??

³ some info about coverage and spatial ordered data??



expose coverage data to the central map could in theory make use of them. In practice, spatial operations work generally only on unstructured grid coverages or curvilinear⁴ grid coverages, which basically rules out most SOBEK input data for instance. However the design does not keep us from building proper support for network coverages, it is simply a matter of priorities.

6.5.2 Object model

The spatial operation stack is represented by the class `SpatialOperationSet`, and has the structure of a single-linked graph. The vertices of this graph are the spatial operations and the links are instances of `SpatialOperationData`, and act as a wrappers around the fundamental data object, `FeatureProvider`. Spatial operations process input feature providers to produce an output that is connected to other operations. When the whole chain is executed, the result is that all intermediate feature providers are up to date, and the final output is evaluated by the last operation in the stack.

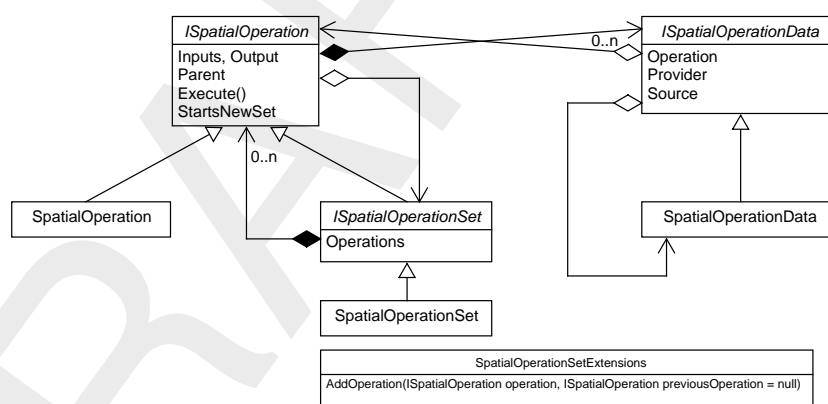


Figure 6.1: UML diagram of spatial operation and operation sets

Note that both input and output `SpatialOperationData` instances are owned by the corresponding spatial operation; the sharing of the output is obtained by a proxy pattern⁵: a connected input `SpatialOperationData` has a `Source` operation data that is invoked upon calling its provider data. The spatial operations have been designed with the idea that

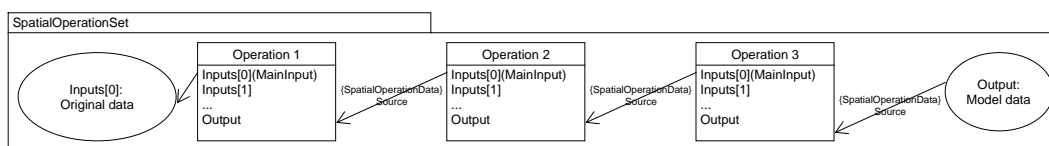


Figure 6.2: Three-operation chain

there is one chain of operations within a spatial operation set. It is however possible that there is a subset in the spatial operation set that has its own chain, which executes independently from some operations in the parent set. The data that is passed through the chain is always

⁴some info about this?

⁵In distributed object communication, a local object represents a remote object (one that belongs to a different address space). The local object is a proxy for the remote object, and method invocation on the local object results in remote method invocation on the remote object. Think of an ATM implementation, it will hold proxy objects for bank information that exists in the remote server.

coupled to the input with label "MainInput", and it is implicitly assumed that the feature type of that propagating provider remains the same throughout the set⁶.

The creation of subsets is determined by the the property `StartsNewSet` on the spatial operation to be added. If it is true, the extension method `AddOperation` will automatically create a subset. Operations that impose this are:

- ◇ `SamplesOperation`: base class for operations that have point cloud output. If such an operation is applied after a coverage operation, it will automatically create a new subset.
- ◇ `CopyToSpatialDataOperation`: clones the coverage to a new subset. Starts a new subset with a coverage main input that comes from the current last operation output in the main set.

In network topology terms, the first operation starts a new (sub-)branch, whereas the second and third correspond to a branch split. Conversely, there are also branch-merging operations:

- ◇ `InterpolateOperation`: interpolates a point cloud from a subset onto the main input coverage.
- ◇ `MergeSpatialDataOperation`: performs an arithmetic operation of a subset output coverage onto the main coverage.

Note that a subset without any link to the main set via one of these merging operations will not have any impact on the final result. Whereas the main input is reserved for the data propagating through the chain, the other inputs of spatial operations are usually reserved for input geometries, e.g. bounding polygons or contours for sample generation.

Finally, we mention the `Dirty` flag, which keeps track of operations that are no longer up-to-date. Upon a property change, this flag is set in the dependent operations, and upon execution only the operations that are dirty are processed.

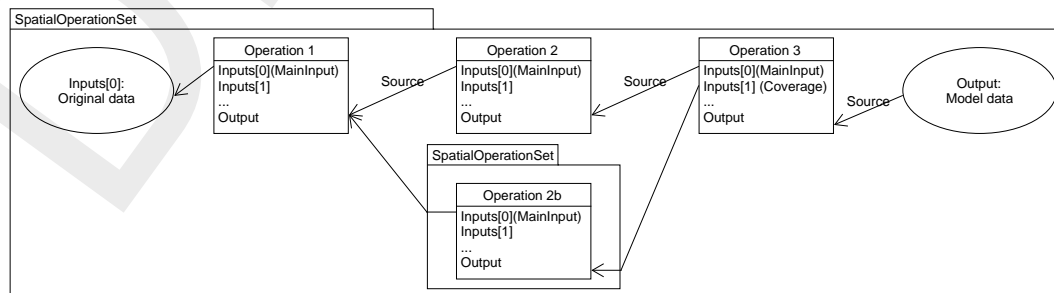


Figure 6.3: Three-operation set with subset topology

6.5.3 Framework Embedding

The embedding of spatial operations in the DeltaShell framework was aimed to be non-intrusive and backwards compatible, but at the same time could be widely used among plug-ins. We have chosen the following strategies:

Inside a dedicated map layer `SpatialOperationSetLayer`.

This group layer is created whenever a data object is the result of a spatial operation stack.

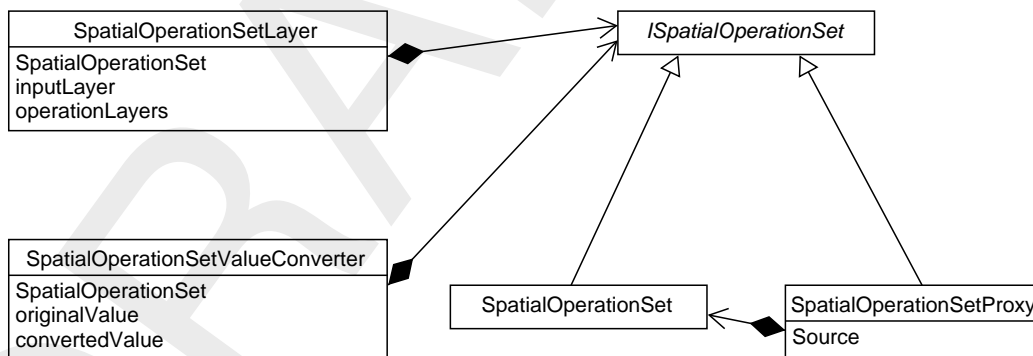
⁶In practice, these types can be either coverages or point clouds

This layer however may or may not be the owner of the spatial operation set itself. In this case, the operation set in the layer will be of the type `SpatialOperationSetProxy`, and its remote source is part of a value converter (see below). The central maps determines the ownership of the spatial operation.

Inside a (dedicated) value converter of input data items.

The responsibility of the value converter is to own the spatial operation set, provide with an input coverage at construction (the *original input*) and copy the values after each execution to the actual coverage (the *converted value*) without replacing the reference to the data item value, because this would break existing functionality in client code.

The creation mechanism for this set of classes is provided by the central map. From the visible, writable layers, the ones whose data source qualifies for spatial operations appear in the combo-box in the spatial operation ribbon section. If a spatial operation command is executed, it checks whether the underlying data is exposed to DeltaShell as a data item, in which case the value converter will be created containing the new spatial operation set. The layer itself is replaced with a `SpatialOperationLayer` containing the operation set proxy.



6.5.4 Spatial Operations

The spatial operations can be divided into two categories: coverage operations and sample set operations. The first ones directly implement the abstract base class `SpatialOperation`, while the second type have a common base `SampleSpatialOperation`, and they are meant to produce point clouds. We already explained the chain splitting and merging operations before; the other operations are

- ◇ **EraseOperation**: Sets the values of a coverage to the `NoDataValue` within a given set of polygons (the `Mask` input).
- ◇ **CropOperation**: Spatial complement of the erase operation: sets values of a coverage to `NoDataValue` outside the given polygons.
- ◇ **SetValueOperation**: Performs an arithmetic operation by a single value to all sample point values or coverage values within a given set of polygons.
- ◇ **GradientOperation**: Performs an arithmetic operation by a gradient field to all sample point values or coverage values within a given set of polygons.
- ◇ **SmoothingOperation**: Performs an iterative averaging of coverage values within a given set of polygons.
- ◇ **OverwriteValueOperation**: Overwrites a coverage value of the point nearest to the given point.
- ◇ **AddSamplesOperation**: Adds a group of samples with a given value to the input point cloud.

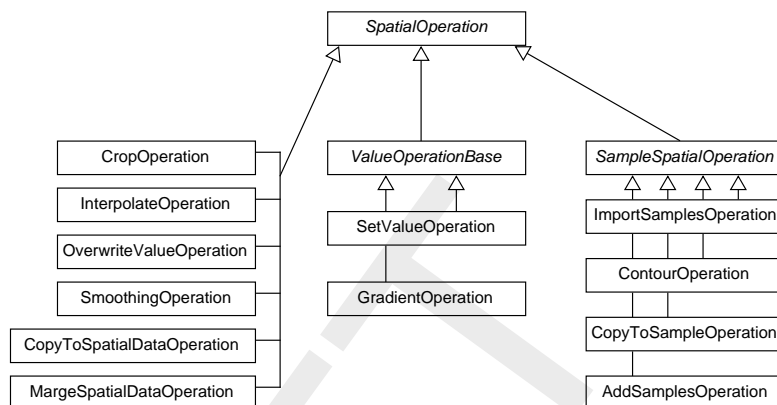


Figure 6.4: Spatial operation inheritance tree.

- ◇ `ContourOperation`: Adds a group of samples along a given polyline with a single value to the input point cloud.
- ◇ `CopyToSamplesOperations`: copies coverage values to a point cloud, skipping no-data valued grid points.

Note that many operations have the option to be confined within a set of input polygons; to avoid duplicate code the utility class `FeatureProviderMask` has been introduced, which contains algorithms for testing whether data points are within a polygon. Operations that perform an arithmetic operation within a given polygon have the common base `ValueOperationBase`. We stress however that the inheritance tree is not definitive and may need restructuring as more operations are included in future work.

6.5.5 GUI

In the DeltaShell user interface, the user creates spatial operations by clicking a the buttons in the spatial editor ribbon section. These buttons are activated by the SharpMapGis GUI plugin and upon clicking a corresponding command is executed. The commands all inherit from the `SpatialOperationCommandBase` class which contains logic to determine whether the command is enabled for the current target layer data source, and handles the insertion of a newly created operation into an existing operation set.

The visualization of a given spatial operation stack is the responsibility of the `SpatialOperationSetLayerVi` which is essentially a tree view of the graph of operations. It contains buttons to disable and delete operations, a refresh button to execute all operations that are marked dirty, and a button to export intermediate results to sample point files.

6.5.6 Tests

To validate the spatial operations we have set up three test projects :

- ◇ SharpMap.Tests: Tests to validate the workings of spatial data implementations
- ◇ SharpMap.UI.Tests: Tests to validate the workings of the GUI elements of the central map with the spatial operation objects
- ◇ SharpMap.Extensions.Tests: Tests to validate the abstract data model of the spatial editor.

All three can make use of the test helper project SharpMapTestUtils. Especially the extensions tests are important because they ensure the correct linking of the spatial data wrappers in various configurations. Note that spatial operations are also tested on the plugin level in e.g. DeltaShell.Plugins.FMSuite.FlowFM.Tests.ImportSpatialOperationsTest.

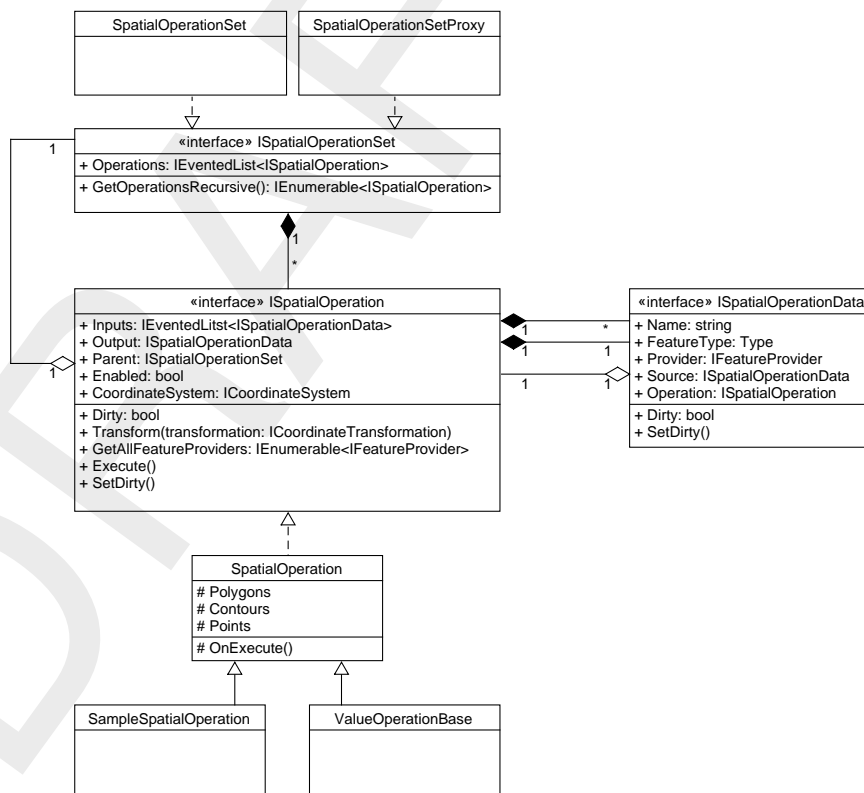


Figure 6.5: Class diagram on spatial operations.

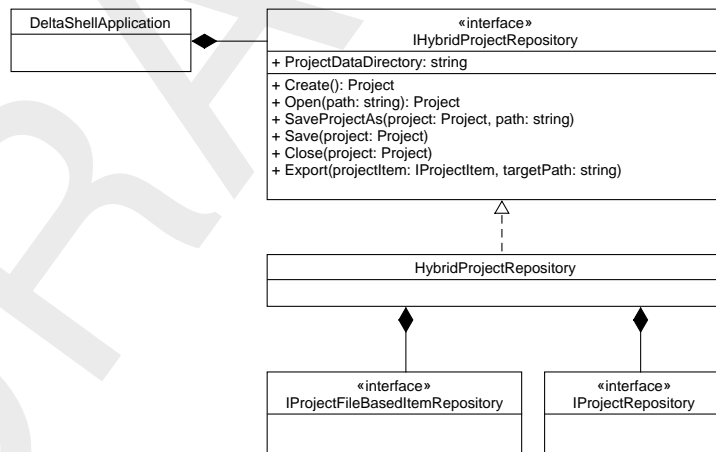


Figure 6.6: Class diagram on how spatial operations are integrated into the GUI.

DRAFT

7 Other topics

7.1 Tables and graphs

7.2 Basic Modeling Interface (BMI)

7.3 Remote Instance Container

7.4 NetCDF

Two netCDF DLLs:

- ◇ one by the framework (netcdf-4.3.dll), which is downloaded from netCDF website
- ◇ one by FM (netcdf.dll), which has been compiled manually by us in order to have access to the internal methods by the FM kernel.

7.5 Scripting using Python



DRAFT

8 Programming help

8.1 NuGet packages

What happens if MorphAn is branched: branch framework along? Or reference one specific Framework NuGet package in plugin/product?

8.2 32-bit vs 64-bit

Chosen for AnyCPU, meaning that by default, it is impossible to reference 32-bit native DLLs.

8.3 Build process

8.4 Creating an installer

build_an_msi.bat Wix

8.5 Testing

RhinoMocks!

8.6 Code conventions

8.7 Localisation

Most important: F6 in ReSharper.

8.8 Licensing

DRAFT

9 Plugins

9.1 Flexible Mesh

9.1.1 Introduction

The purpose of the Flexible Mesh (FM) plugin for DeltaShell is to provide a new user interface to D-Flow FM, a Fortran code which solves the shallow water equations on two-dimensional unstructured grids. In this system, the computational core is compiled as a dynamic link library and wrapped by the `FlexibleMeshModelApi` class for usage in the plugin. The interface consists of the standard BMI, augmented with utility¹ algorithms for spatial interpolation, grid snapping and cell searching. Transmitting data between the FM kernel and the DeltaShell plugin is not the exclusive responsibility of the `FlexibleMeshModelApi`; the bulk of the model data is serialized to ASCII or NetCDF files and read by the kernel at model initialization. In fact, the variable getters and setters within the BMI are only used for real-time coupling of the model and online visualization of data.

The serialization to files provides the actual storage for Flexible Mesh models in DeltaShell. The database mapping of the models is extremely sparse, with only the path to the master definition file and output data as its members. This has the important advantage that all existing FM models can be imported in DeltaShell and conversely FM models in the plugin can be exported to files and run for example on a Linux machine. Things become more complicated for models that are coupled to D-RTC and models that contain spatial operations on their initial data or bed levels, because they depend on framework objects that are mapped in the database.

In the same source folder, `FMSuite`, we have created the Wave plugin which has many shared components with Flexible Mesh. It uses a similar file-based master definition file and corresponding dynamic model settings, it uses comparable boundary data and its bed level data and initial conditions are defined on a 2d grid too, albeit a curvilinear regular grid. The common components of wave and FM are grouped into a separate plugin `FMSuite.Common`.

9.1.1.1 Overview

The basic object model of the Flexible Mesh plugin in DeltaShell is as follows: we have the `WaterFlowFMModel` as the object containing all relevant data.

- ◇ Its parameters are contained by the so-called `WaterFlowFMModelDefinition` in a dynamically typed fashion.
- ◇ The grid- and time-independent schematization is contained by the `HydroArea` class
- ◇ The `IHydroRegion` implementation that provides spatial features such as dams, weirs, structures and observation stations.
- ◇ The model contains an unstructured grid and spatial fields that are defined on this grid, such as bed levels, initial water levels and other initial conditions.
- ◇ Finally the model contains boundaries with corresponding boundary conditions, and other time-dependent data such as wind fields, temperature series, source and sink discharges.

The model output can be divided in three categories:

- 1 Spatial time-dependent data from the `MapFileFunctionStore` defined on the grid
- 2 Time series on observation points which are read by the `HisFileFunctionStore`
- 3 Possibly restart files.

¹We stress here that these additional methods do not change the state of the underlying model in the native library, such functions are exclusively within the BMI.



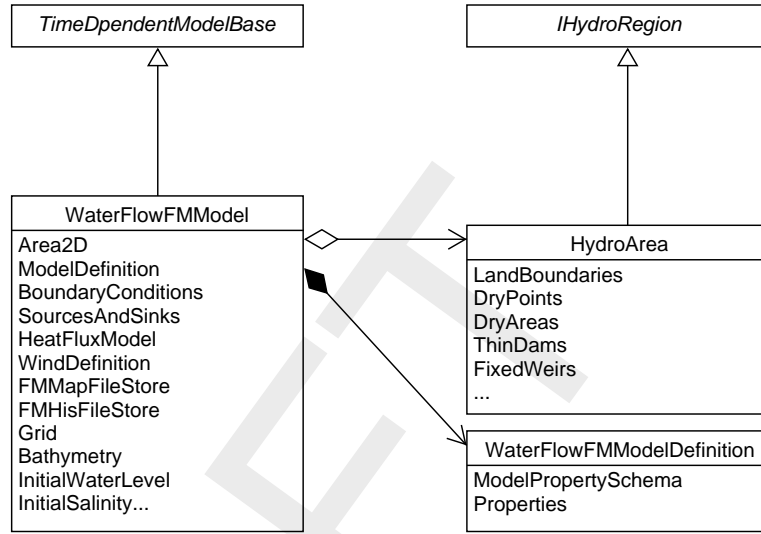


Figure 9.1: Global data model of the Flexible Mesh model in DeltaShell.

The output function stores above are both NetCDF function stores with optimizations, with the assumption of read-only function values.

9.1.2 The FlexibleMeshModelApi Class

As mentioned above, the wrapper of the computational core in the FM plugin implements the `IFlexibleMeshModelApi` interface. There are in fact two implementations: the actual `FlexibleMeshModelApi` class calling the native library and `RemoteFlexibleMeshModelApi` that creates a separate process in which a `FlexibleMeshModelApi` instance exists, and which serializes all calls and routes them to this instance. This construction has been chosen to protect the user interface from memory corruption and process killing statements in the Fortran code. Moreover it allows us to deploy a 64-bit kernel library within a 32-bit DeltaShell instance and vice versa. Which implementation is used in the flow model instance is controlled with the `UseLocalApi` property. The API consists of a BMI, extended with the following

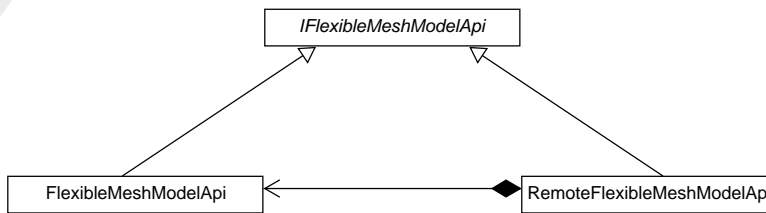


Figure 9.2: Remote and actual implementations of the `IFlexibleMeshModelApi`.

methods:

- ◇ `InitializeUserTimeStep`, `FinalizeUserTimeStep`, `InitializeComputationalTimeStep`, `RunComputationalTimeStep`, `FinalizeComputationalTimeStep` and `Compute1d2dCoefficients` functions necessary for the coupling of FM with a 1D SOBEK model,
- ◇ `WriteNetGeometry`, `WritePartitioning`: utility functions to export the grid geometry (or partition the grid),
- ◇ `GetSnappedFeature`: utility function to project a given geometry onto the grid, depending upon the type of the object, e.g. observation points to cell-centers, thin dams to flow links, etc.

The latter function is purely exposed through the interface `IGridOperationApi` with an adapter² implementation.

Note that there exist additional subroutines in the interface of `flowfm.dll` which are not exposed by the API above. These methods are used for interpolation and cell-finding algorithms in the DeltaShell framework and exposed through the `GeometryApi` class. In the installation, the native library is actually included twice: once as kernel for the FM plugin and once as geometry utility library, under the name `geometry.dll`.

9.1.3 The File System

The bulk of the data is passed to the `dflowfm` kernel by means of files; it is actually impossible to initialize a model in the kernel without providing a *master definition* (`mdu`) file. This file, with all its referenced sub-files, constitutes the storage of the model in DeltaShell as well. Hence, loading and saving a model uses the exact same code as importing and exporting to `mdu`, where the latter is also called when initializing a model run. The `mdu`-file consists of a grouped set of keywords with corresponding values, and optional comment. The set of keywords with corresponding value types, bounds, defaults and descriptions are listed in a csv-file named *dflowfm-properties*, which is distributed with the application and parsed at runtime by the plugin. This creates a dynamic list of model properties in `WaterFlowFMModelDefinition`, which will be matched with each `mdu`-file that is being read or written. Moreover, the file provides essential UI information such as conditions under which certain controls are visible/enabled. These conditions are parsed and converted to predicates used by the `WaterFlowFMModelView` class.

The `mdu`-file references various (sub-)files containing spatial data or time series. The reading/writing of the master definition file, the `MduFile` class, takes care of these sub-files by cascading read/write operations. For complete list of the sub-files we refer to the FM user manual. The most important file is the *external forcings file* that holds spatial information on initial conditions and wind fields, but possible also boundary conditions in the deprecated format (`pli`-file with `tim`-files or `cmp`-files). Newly created boundary conditions are always serialized to `bc`-files, which are referenced by the *boundary external forcings file*.

The file-based nature of FM makes saving and loading of the model slightly cumbersome; it is therefore essential for developers to keep in mind the different states that the `WaterFlowFMModel` can adopt:

- ◇ Pure memory state: when calling the default constructor from C# or from a script, the resulting instance will have its property `MduFilePath` equal to the `null`-string. Hence there are no files corresponding to the instance. The model name will be the default "FlowFM", which will also be the name of the `mdu` file upon serialization at this time.
- ◇ Temporary state: when calling the constructor with the `mdu-path` argument, this will be the value of `MduFilePath`, and all data will be read from the referenced sub-file. Possible output map- or his-files will be attached to function stores for lazy readout.
- ◇ Saved state: Upon saving, DeltaShell creates a data folder next to the `dsproj`-file in which two folders are created: the first which has the same name as the `mdu`-file and all input data files are written to it, and an empty directory `<mdufile>_output`, which is called the explicit working directory. The property `MduFilePath` takes the name of the master definition file within the local data directory.
- ◇ Running state: Upon execution, the kernel will run inside the `WorkingDirectory`. If the model was never saved, this directory will be randomly named and created within the user Temp directory, otherwise the working directory is the explicit working directory mentioned above.

²The Adapter design pattern allows otherwise incompatible classes / interfaces to work together by converting the interface of one class into an interface expected by the clients.

Note that since the underlying kernel needs an mdu file to initialize, the plugin frequently uses a 'stripped' copy of the definition file in a temporary directory to call certain API functions, for example when fetching snapped feature geometries.

9.1.4 The Unstructured Grid

One of the more intricate and error-prone features of the FM plugin is the way in which the grid is handled. The main difficulty lies in the fact that some of the data in the unstructured grid, i.e. the list of *flow links* is actually dependent upon the geometries of model features like thin dams, dry points and boundary locations. The computing algorithm for these flow links is within the computational core, but it requires initialization of d-flow FM with the master definition file of the model. After this, a map-file is written by the kernel that contains the flow link data, which is read to 'dress' the unstructured grid. The essential method in this workflow is `ReloadGrid`, which ensures the file-based state of the grid (net-file) and the memory state are synchronized. The various moments for reloading the grid are

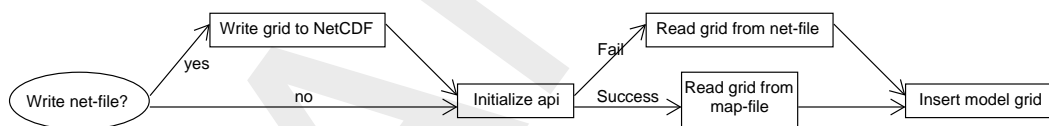


Figure 9.3: Control structure of the *ReloadGrid* implementation

- ◇ Default constructor: the grid is empty, no action is taken.
- ◇ Constructor with mdu-file path and net-file: we check whether there is a corresponding map-file. If so, the grid including flow links is read from this output, otherwise `ReloadGrid` is called with the first argument set to false.
- ◇ Import of a net-file: the net-file is copied to the mdu-directory and `ReloadGrid` is called with the first argument set to false.
- ◇ Exit of `RgfGrid` after grid edits: same actions as importing a net-file.
- ◇ Setting the grid from a script or test: it is recommended to call `ReloadGrid` with first element set to true if you wish to run the model with the new grid or get the correct flow links. Be aware that the grid instance in the model will change after this.

The last step of the `ReloadGrid` method is inserting the resulting grid into the model instance. The grid setter will determine whether the new instance differs from the current instance, and if so it will re-evaluate all the operations on the spatial data, and substitute the new grid.

9.1.5 Boundary conditions

The main philosophy in D-Flow FM regarding spatial features is that they exist independently from the grid; it is sufficient to provide their geometries in .pli, .xyz or .pol format, and the kernel performs the necessary preprocessing to project them onto the mesh. Hence, in our object model all these features are represented by classes that contain no more data than the standard `Feature` class, plus a name identifier. These features are usually contained in the hydro area of the model, with exception of the boundaries and sources and sink geometries ('pipes').

These features are accompanied with time series data (or some other parameterization of the flow dynamics), we have chosen a design which makes a clear distinction between the feature geometry and model data³ that is defined on it.

³In contrast with e.g. structures in FM, whose time series are defined within the class.

For instance, the FM model contains a list of pipes –line features– and a separate list of `SourceSink` objects that define a discharge time series associated with a feature. The model itself is responsible for synchronizing both lists. The design of boundary conditions resembles this, with the distinction that the correspondence is not one-to-one: multiple boundary conditions can be associated with a single boundary feature. Furthermore, multiple functions may be contained inside condition, as they are generally associated with the coordinates that define the polyline feature geometry, although there are exceptions (e.g. discharge boundaries).

Boundary conditions in the FM suite have a common interface defined by `IBoundaryCondition`.

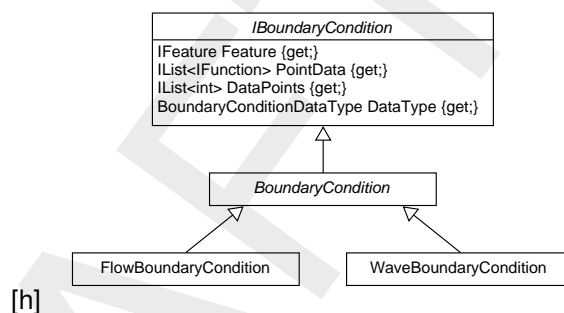


Figure 9.4: Boundary condition object model in FMSuite

It contains the feature, the functions defining the actual positions, a list of indices defining the location of this data⁴ and a `DataType` enumerable which determines the parameterization of the data. This interface has a generic abstract implementation which allows both the FM plugin as well as the Wave plugin to use it for their boundary conditions. Multiple conditions (with various `FlowBoundaryQuantityTypes`) can be defined on the same boundary, but there is a limited set of valid combinations defined in 'dictionaries / lists' in the child class. Such flow boundary conditions are grouped into a `BoundaryConditionSet` in the FM model. The single interface has allowed for both FM and Wave to also use a common view on their boundary conditions, the `BoundaryConditionEditor`. However there are sub-controls within this editor that are injected by the plugins to customize the interface. Historically, boundary conditions were listed in the external forcings file, and every single condition meant a single pli-file with a set of corresponding data files⁵. Nowadays we have bc-files containing all boundary data that belongs to a certain `FlowBoundaryQuantityType`. These files shall soon be used for SOBEK (flow 1d) boundary conditions as well.

9.1.6 GUI

todo...

9.1.7 Tests

To validate the spatial operations we have set up two test projects :

- 1 DeltaShell.Plugins.FMSuite.FlowFM.Tests : Tests to validate the workings of the FM plugin and GUI.
- 2 DeltaShell.Plugins.FMSuite.Common.Tests : Tests to validate the workings of common

Deltares, 2016. "BIBTEX key with no entry, needed if no citations are made in the document."

⁴There should be no concerns about geometry points with no data defined on them; the kernel performs automatic spatial interpolation along the polyline.

⁵If such a set of files is read and no new boundaries are added, this is still the output format

DRAFT

A Tutorial: start a new plugin

The goal of this tutorial is to show how a new plugin for the Delta Shell framework can be developed.

A.1 Add new plugin

Open the solution in visual studio and in the solution explorer navigate to (Src | Plugins). Right click it and add a new solution folder named 'VolumeModel'. Right click that folder and choose (Add | New Project...). Choose Template C#, .NET Framework 4.5, Class Library. For the name enter 'DeltaShell.Plugins.VolumeModel'. The location should match the solution-folder that was just created (i.e. browse to it using the 'Browse...' button). In the solution folder also add a project named 'DeltaShell.Plugins.VolumeModel.Gui', using the same steps.

For both projects the following actions have to be performed:

A.1.1 Edit AssemblyInfo.cs

Add the bottom of the file add the following code

```
[assembly: Addin]
[assembly: AddinDependency("DeltaShellApplication", "1.0")]
```

A.1.2 Edit project's csproj file

The project's .csproj file needs to be adapted. This can be done either directly on the file system, by opening the file in a project editor. The second option is to unload the project in Visual Studio (one of the options in the right click context menu) and then choose to edit the .csproj (again an option in the right click context menu when the project is unloaded). After editing be sure to reload the project.

```
<Project>
...
...
<Import Project="$ (MSBuildToolsPath) \Microsoft.CSharp.targets" />
<PropertyGroup>
  <IsPluginComponent>true</IsPluginComponent>
  <PluginName>$(ProjectName)</PluginName>
</PropertyGroup>
<Import Project="..\..\..\..\build\DeltaShell.targets" />
<Import Project="..\..\..\..\packages\PostSharp.2.1.7.28\tools\PostSharp.
  targets" Condition="Exists('..\..\..\..\packages\PostSharp.2.1.7.28\
  tools\PostSharp.targets') " />
<PropertyGroup>
  <PreBuildEvent>cmd /c $(SolutionDir)\svn_insert_version.cmd $(ProjectDir)
    \Properties
  </PreBuildEvent>
</PropertyGroup>
<!-- To modify your build process, add your task inside one of the targets
  below and uncomment it.
  Other similar extension points exist, see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
-->
</Project>
```

Notice especially the first PropertyGroup that defines that this project represents a plugin. By importing project *build/DeltaShell.targets* it is ensured that any resource specific to this plugin

(

TODO *TODO A.1: see section TODO*

) gets copied to the correct location.

A.1.3 Add code

Remove file *Class1.cs*. Right click the project and select (Add | Class...). Choose the empty class definition simply labelled as *Class* and give it the name *VolumeModelApplicationPlugin*.

Edit the code to make it look like this:

```
using DelftTools.Shell.Core;
using Mono.Addins;

namespace DeltaShell.Plugins.VolumeModel
{
    [Extension(typeof(IPlugin))]
    public class VolumeModelApplicationPlugin : ApplicationPlugin
    {
        public override string Name
        {
            get { return "Volume model application"; }
        }

        public override string Description
        {
            get { return "Application plugin of the volume model application"; }
        }

        public override string Version
        {
            get { return "1.0"; }
        }
    }
}
```

Add another class using the same steps and name it *VolumeModelGuiPlugin*. Make it look like this:

```
using DelftTools.Shell.Core;
using DelftTools.Shell.Gui;
using Mono.Addins;

namespace DeltaShell.Plugins.VolumeModel
{
    [Extension(typeof(IPlugin))]
    public class VolumeModelGuiPlugin : GuiPlugin
    {
        public override string Name
        {
            get { return "Volume model application (UI)"; }
        }

        public override string Description
        {
            get { return "Gui plugin of the volume model application"; }
        }

        public override string Version
        {
            get { return "1.0"; }
        }
    }
}
```

```

    }
}
}

```

A.1.4 Result so far

To check if all went well so far, run the application and open the *About* window (File | Help | About). Ensure both the application plugin and the gui plugin present themselves in the list of loaded plugin that appears in the listbox.

A.2 Import time series from WaterML2 files

We will now create and register a WaterML2 importer class, making it possible to import WaterML2 files (containing time series data) into a Delta Shell project.

A.2.1 Create a new importer class

To the plugin project add a new folder named *Importers*. In this folder, create a new classed named *WaterML2TimeSeriesImporter* and make it look like this¹:

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Xml.Linq;
using DelftTools.Functions;
using DelftTools.Functions.Generic;
using DelftTools.Shell.Core;
using log4net;

namespace DeltaShell.Plugins.VolumeModel.Importers
{
    /// <summary>
    /// Importer for importing WaterML2 data to time series objects
    /// </summary>
    public class WaterML2TimeSeriesImporter : IFileImporter
    {
        private static readonly ILog Log = LogManager.GetLogger(typeof(
            WaterML2TimeSeriesImporter)); // Handle for writing log messages

        /// <summary>
        /// The name of the importer
        /// </summary>
        /// <remarks>Used in importer selection dialogs</remarks>
        public string Name
        {
            get { return "WaterML2 time series importer"; }
        }

        /// <summary>
        /// The category of the importer
        /// </summary>
        /// <remarks>Used in importer selection dialogs</remarks>
        public string Category
        {
            get { return "Volume model importers"; }
        }
    }
}

```

¹Class *WaterML2TimeSeriesImporter* is derived from the *IFileImporter* interface so that it can be registered in the application plugin (see the next step). The comments in the code explain the different parts of the importer implementation.

```
/// <summary>
/// The image of the importer
/// </summary>
/// <remarks>Used in importer selection dialogs</remarks>
public Bitmap Image
{
    get { return new Bitmap(16, 16); }
}

/// <summary>
/// The data types supported by the importer
/// </summary>
public IEnumerable<Type> SupportedItemTypes
{
    get { yield return typeof(TimeSeries); }
}

/// <summary>
/// Indicates that the importer can import at root level (folder/
    project). In other
/// words, indicates that the <see cref="ImportItem"/> method can be
    called without
/// specifying a time series target...
/// </summary>
public bool CanImportOnRootLevel
{
    get { return true; }
}

/// <summary>
/// The file filter of the importer
/// </summary>
/// <remarks>Used in file selection dialogs</remarks>
public string FileFilter
{
    get { return "WaterML2 files|*.XML"; }
}

/// <summary>
/// Path where external data files can be copied into
/// </summary>
/// <remarks>Not relevant in this tutorial</remarks>
public string TargetDataDirectory { get; set; }

/// <summary>
/// Whether or not an import task should be cancelled
/// </summary>
/// <remarks>Not part of this tutorial</remarks>
public bool ShouldCancel { get; set; }

/// <summary>
/// Fired when progress has been changed
/// </summary>
/// <remarks>Not a part of this tutorial</remarks>
public ImportProgressChangedDelegate ProgressChanged { get; set; }

/// <summary>
/// Imports WaterML2 data from the file with path <paramref name="path
    "> to the
/// time series <paramref name="target"/>
/// </summary>
/// <remarks>
/// The target parameter is optional. If a target time series is
    specified, the
/// importer should import the WaterML2 data to this existing time
    series. When
```



```

// no target is set, the importer should create a new time series.
/// </remarks>
public object ImportItem(string path, object target = null)
{
    // Check the file path
    if (!File.Exists(path))
    {
        Log.Error("File does not exist");

        return null;
    }

    // Obtain a new time series or check the provided target for being
    // a time series
    var timeSeries = target == null
        ? new TimeSeries { Name = Path.GetFileNameWithoutExtension(path),
            Components = { new Variable<double>() } }
        : target as TimeSeries;

    if (timeSeries == null)
    {
        Log.Error("Target is of the wrong type (should be time series)");
        ;

        return null;
    }

    // Load the WaterML2 XML document
    var doc = XDocument.Load(path);

    // Obtain the document elements
    var xElements = doc.Descendants();

    // Obtain the measurement TVP tags
    var measurements = xElements.Where(element => element.Name.
        LocalName == "MeasurementTVP");

    // Get the corresponding time and value for each measurement tag
    foreach (var measurement in measurements)
    {
        var time = DateTime.Parse(measurement.Elements().First(e => e.
            Name.LocalName == "time").Value);
        var value = double.Parse(measurement.Elements().First(e => e.
            Name.LocalName == "value").Value);

        timeSeries[time] = value;
    }

    // Return the time series
    return timeSeries;
}
}

```

In order to successfully build this code some references need to be added to the project. Right click on the project's <References> folder and select *Add Reference...* Then choose the *Browse...* button to select file <log4net.dll> from the <lib> folder from the Delta Shell code installation location. In a similar fashion add a reference to <System.Drawing> which resides in the Framework Assemblies (i.e. do not browse there but select it from the header (Assemblies | Framework)).

A.2.2 Register the importer in the application plugin class

Register the importer in the application plugin by adding the following code to *VolumeModelApplicationPlugin.cs*:

```
using System.Collections.Generic;
using DeltaShell.Plugins.VolumeModel.Importers;

and

public override IEnumerable<IFileImporter> GetFileImporters()
{
    yield return new WaterML2TimeSeriesImporter();
}
```

Delta Shell is now able to use this importer when importing data on new or existing time series objects.

A.2.3 Create test data

Select all text from [section A.8](#) and copy it into a text editor so you can save it to a file named `<WaterML2_precipitation_data.XML>`.

A.2.4 Test importer

Run the application and start an import by right clicking on the project and selecting *import...*. In the dialog that is presented the newly added WaterML2 time series importer appears. Select it and press *OK*. A file dialog appears to select the file to import. Navigate to the file you created in [section A.2.3](#), select it and finish the wizard. Upon conclusion a new time series object containing the data from the imported file is added to the project tree. Double click it to investigate it using Delta Shell's built in table and graph viewer.

In the steps above a project level import has been executed, thereby creating an entirely new time series item. The importer is also capable of importing into existing time series items by right clicking such an object and selecting *Import...*

For example, test this by:

- ◇ clearing (a part) of the imported time series data via the table view.
- ◇ importing the downloaded WaterML2 XML file directly onto this modified time series item (right click on the time series item | import ...)

A.3 Create a simple hydrological model

We will add a simple hydrological volume model to the plugin, allowing a user to run the volume model and inspect some simple spatio-temporal output results.

A.3.1 Create a new model class

To the plugin project add a new folder named *Models*. In this folder, create a new class named *VolumeModel* and make it look like this²:

```
using System;
using System.Linq;
using DelftTools.Functions;
```

²The model class is derived from the *ModelBase* class in order to automatically implement some basic time dependent modeling logic. The comments in the code explain the different parts of the model implementation.

```

using DelftTools.Functions.Generic;
using DelftTools.Hydro;
using DelftTools.Shell.Core.Workflow;
using DelftTools.Shell.Core.Workflow.DataItems;
using NetTopologySuite.Extensions.Coverages;

namespace DeltaShell.Plugins.VolumeModel.Models
{
    public class VolumeModel : ModelBase
    {
        private readonly DrainageBasin basin;
        private readonly TimeSeries precipitation;
        private readonly FeatureCoverage volume;

        /// <summary>
        /// Creates a volume model
        /// </summary>
        public VolumeModel()
        {
            // Create the input items of the volume model
            basin = new DrainageBasin();
            precipitation = new TimeSeries { Components = { new Variable<double>
                >("Precipitation") } } };

            // Create the output item of the volume model
            volume = new FeatureCoverage("Output data")
            {
                IsTimeDependent = true,
                Arguments = { new Variable<Catchment>("Catchment") {
                    FixedSize = 0 } },
                Components = { new Variable<double>("Volume") },
            };

            // Wrap fields as input/output data items
            DataItems.Add(new DataItem(precipitation, "Precipitation", typeof(
                TimeSeries), DataItemRole.Input, "PrecipitationTag"));
            DataItems.Add(new DataItem(basin, "Basin", typeof(DrainageBasin),
                DataItemRole.Input, "BasinTag"));
            DataItems.Add(new DataItem(volume, "Volume", typeof(FeatureCoverage
                ), DataItemRole.Output, "VolumeTag"));
        }

        /// <summary>
        /// The precipitation time series:  $P = P(t)$  [L/T]. Input of the model.
        /// </summary>
        public TimeSeries Precipitation
        {
            get { return precipitation; }
        }

        /// <summary>
        /// The drainage basin (set of catchments). Input of the model.
        /// </summary>
        public DrainageBasin Basin
        {
            get { return basin; }
        }

        /// <summary>
        /// Time-dependent feature coverage containing the volume of water per
        /// catchment:  $V = V(t, c)$  [L3/T]. Output of the model.
        /// </summary>
        public FeatureCoverage Volume
        {
            get { return volume; }
        }
    }
}

```

```
/// <summary>
/// The initialization of model runs
/// </summary>
protected override void OnInitialize()
{
    // Clear any previous output
    volume.Clear();

    // Ensure the coordinate system of the volume output is the same as
    // the catchments input (basin)
    volume.CoordinateSystem = basin.CoordinateSystem;

    // Ensure at least one catchment and one precipitation value is
    // present
    ValidateInputData();

    // Initialize the output feature coverage
    volume.Features.AddRange(basin.Catchments);
    volume.FeatureVariable.FixedSize = basin.Catchments.Count;
    volume.FeatureVariable.AddValues(basin.Catchments);
}

/// <summary>
/// The actual calculation during model run
/// </summary>
protected override bool OnExecute()
{
    // Loop all times
    foreach (var time in precipitation.Time.Values)
    {
        // Obtain the precipitation value for the current time
        var p = (double) precipitation[time];

        // Calculate a volume value for every catchment based on
        // catchment area and precipitation value
        var volumes = basin.Catchments.Select(c => c.AreaSize * p);

        // Add the calculated volume values to the output feature
        // coverage
        volume[time] = volumes;
    }

    return true;
}

private void ValidateInputData()
{
    var hasCatchments = basin.Catchments.Any();
    var hasPrecipitationData = precipitation.Time.Values.Any();

    if (!hasCatchments && !hasPrecipitationData)
    {
        throw new InvalidOperationException("At least one catchment and
            one precipitation value should be present");
    }

    if (!hasCatchments)
    {
        throw new InvalidOperationException("At least one catchment
            should be present");
    }

    if (!hasPrecipitationData)
    {
        throw new InvalidOperationException("At least one precipitation
```

```
        value should be present");  
    }  
}  
}
```

A.3.2 Register the model in the application plugin class

A.3.3 Test model functionality

A.4 Customize model properties

A.4.1 Create a new object properties class

A.4.2 Register the object properties in the gui plugin class

A.4.3 Test properties

A.5 Show model data on a map

A.5.1 Create a new map layer provider

A.5.2 Register the map layer provider in the gui plugin class

A.5.3 Test map layer provider

A.6 Set up a custom model view

A.6.1 Create a new view

A.6.2 Register the view in the gui plugin class

A.6.3 Test view

A.7 Create a ribbon button

A.7.1 Create a new gui command

A.7.2 Create a new ribbon control

A.7.3 Register the ribbon control in the gui plugin class

A.7.4 Test ribbon button

A.8 WaterML2 test data

```
<?xml version="1.0" encoding="UTF-8"?>
<wml2:Collection xmlns:wml2="http://www.opengis.net/waterml/2.0" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance" xmlns:gml="http://www.opengis
  .net/gml/3.2" xmlns:om="http://www.opengis.net/om/2.0" xmlns:sa="http://
  www.opengis.net/sampling/2.0" xmlns:sams="http://www.opengis.net/
  samplingSpatial/2.0" xmlns:xlink="http://www.w3.org/1999/xlink" xsi:
  schemaLocation="http://www.opengis.net/waterml/2.0 http://www.opengis.
  net/waterml/2.0/waterml2.xsd" gml:id="Ki.Col.1">
  <gml:description>KISTERS KiWIS WaterML2.0</gml:description>
  <wml2:metadata>
    <wml2:DocumentMetadata gml:id="Ki.DocMD.1">
      <wml2:generationDate>2012-06-12T12:10:12.670+00:00</wml2:
        generationDate>
      <wml2:generationSystem>KISTERS KiWIS</wml2:
        generationSystem>
    </wml2:DocumentMetadata>
  </wml2:metadata>
  <wml2:temporalExtent>
    <gml:TimePeriod gml:id="Ki.TempExt.1">
      <gml:beginPosition>1990-09-01T00:00:00.000+01:00</gml:
        beginPosition>
      <gml:endPosition>1990-09-30T00:00:00.000+01:00</gml:
        endPosition>
    </gml:TimePeriod>
  </wml2:temporalExtent>
  <wml2:observationMember>
    <om:OM_Observation gml:id="Ki.OM_Obs.1">
      <om:phenomenonTime>
        <gml:TimePeriod gml:id="Ki.ObsTime.1">
          <gml:beginPosition>1990-09-01T00
            :00:00.000+01:00</gml:beginPosition>
          <gml:endPosition>1990-09-30T00
            :00:00.000+01:00</gml:endPosition>
        </gml:TimePeriod>
      </om:phenomenonTime>
      <om:resultTime>
        <gml:TimeInstant gml:id="Ki.resTime.1">
          <gml:timePosition>1990-09-30T00
            :00:00.000+01:00</gml:timePosition>
        </gml:TimeInstant>
      </om:resultTime>
      <om:procedure xlink:href="http://kiwis.kisters.de/ts/Day.
        Cmd" xlink:title="10 - DailyMean"/>
      <om:observedProperty xlink:href="http://kiwis.kisters.de/
        parameters/557" xlink:title="Q"/>
      <om:featureOfInterest xlink:href="http://kiwis.kisters.de/
        stations/1732100" xlink:title="ATHIEME"/>
      <om:result>
        <wml2:MeasurementTimeseries gml:id="Ki.Ts.132042">
          <wml2:temporalExtent>
            <gml:TimePeriod gml:id="Ki.TsTime.1">
              <gml:beginPosition>1990-09-01T00
                :00:00.000+01:00</gml:
                  beginPosition>
              <gml:endPosition>1990-09-30T00
                :00:00.000+01:00</gml:
                  endPosition>
            </gml:TimePeriod>
          </wml2:temporalExtent>
          <wml2:defaultPointMetadata>
            <wml2:DefaultTVPMeasurementMetadata>
              <wml2:interpolationType xlink:href
                ="http://www.opengis.net/def/
```

```

        waterml/2.0/interpolationType/
        ConstPrec" xlink:title="
        Constant in preceding interval
        "/>
        <wml2:qualifier xlink:href="http
        ://kiwis.kisters.de/
        statusCodes/40" xlink:title="
        40"/>
        <wml2:uom uom="cumeC"/>
    </wml2:DefaultTVPMeasurementMetadata>
</wml2:defaultPointMetadata>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-01T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1930</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-02T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1820</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-03T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1710</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-04T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1190</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-05T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1290</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-06T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1250</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-07T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1300</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-08T00
        :00:00.000+01:00</wml2:time>
        <wml2:value>1080</wml2:value>
    </wml2:MeasurementTVP>

```

```
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-09T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>1000</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-10T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>890</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-11T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>890</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-12T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>1060</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-13T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>1320</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-14T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>1450</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-15T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>1500</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-16T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>1520</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>1990-09-17T00
      :00:00.000+01:00</wml2:time>
    <wml2:value>1180</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
  <wml2:MeasurementTVP>
```



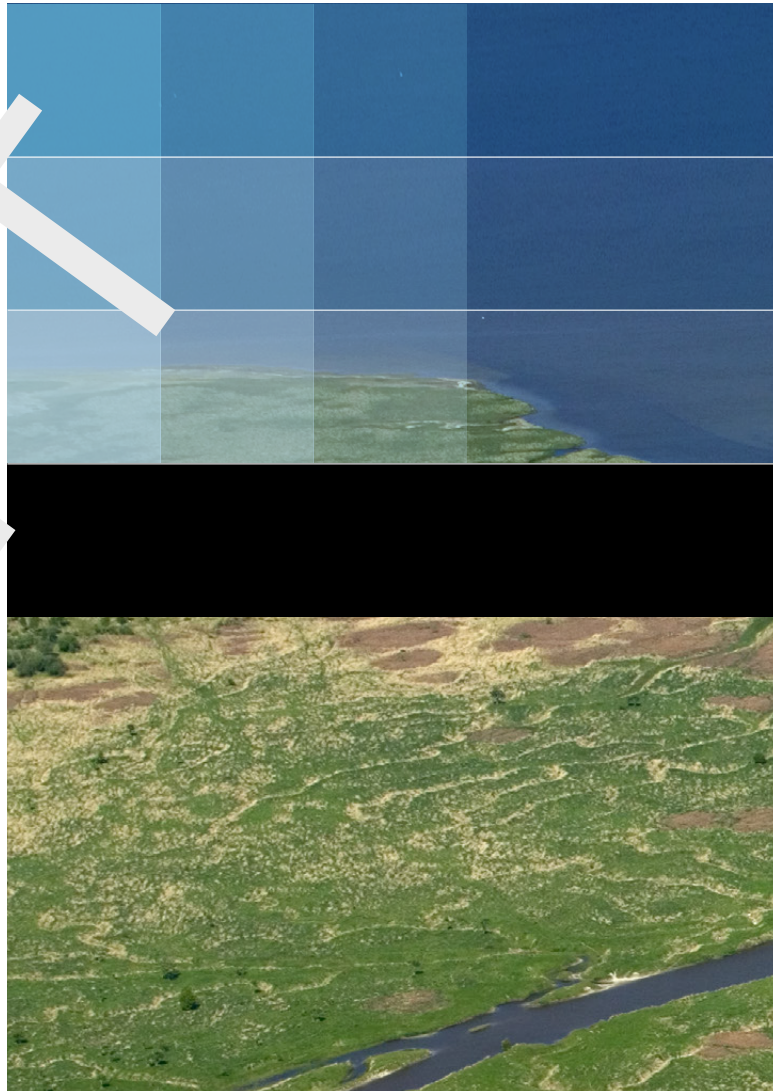
```

        <wml2:time>1990-09-18T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>920</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-19T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1060</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-20T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1290</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-21T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1250</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-22T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1520</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-23T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1520</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-24T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1200</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-25T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>960</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-26T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1230</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>
<wml2:point>
    <wml2:MeasurementTVP>
        <wml2:time>1990-09-27T00
            :00:00.000+01:00</wml2:time>
        <wml2:value>1270</wml2:value>
    </wml2:MeasurementTVP>
</wml2:point>

```

```
        </wml2:MeasurementTVP>
      </wml2:point>
      <wml2:point>
        <wml2:MeasurementTVP>
          <wml2:time>1990-09-28T00
            :00:00.000+01:00</wml2:time>
          <wml2:value>1300</wml2:value>
        </wml2:MeasurementTVP>
      </wml2:point>
      <wml2:point>
        <wml2:MeasurementTVP>
          <wml2:time>1990-09-29T00
            :00:00.000+01:00</wml2:time>
          <wml2:value>1410</wml2:value>
        </wml2:MeasurementTVP>
      </wml2:point>
      <wml2:point>
        <wml2:MeasurementTVP>
          <wml2:time>1990-09-30T00
            :00:00.000+01:00</wml2:time>
          <wml2:value>1390</wml2:value>
        </wml2:MeasurementTVP>
      </wml2:point>
    </wml2:MeasurementTimeseries>
  </om:result>
</om:OM_Observation>
</wml2:observationMember>
</wml2:Collection>
```


DRAFT



Deltares systems

PO Box 177
2600 MH Delft
Boussinesqweg 1
2629 HV Delft
The Netherlands

+31 (0)88 335 81 88
software@deltares.nl
www.deltares.nl/software

