



AVEVA (12 Series)

.NET Customisation

TRAINING GUIDE

TM-2268

Revision Log

Date	Revision	Description of Revision	Author	Reviewed	Approved
20/11/2009	0.1	Issued for Review	MD & PJ		
14/12/2009	0.2	Marine example and updates from the reviews	MD		
08/01/2010	0.3	Reviewed	MD&MP	LS	
08/01/2010	0.4	Issued for proof training	MD	LS	SH
04/02/2010	0.5	Final modifications	MP		
04/02/2010	1.0	Issued for internal training	MP	LS&JF	SH
23/02/2010	1.1	Small modifications	MD		
26/02/2010	1.2	Reviewed	MD	LS&HJ	
26/02/2010	2.0	Issued for Training on 12.0SP5	MD	LS&HJ	SH
18/03/2010	2.1	RegEx exercise simplified	MP		
23/03/2010	3.0	Issued for Training on 12.0SP5	MP	LS	SH
11/09/2010	3.1	Chapter 5.4 added and guide is updated for 12.0SP6	MP		
20/09/2010	4.0	Issued for Training on 12.0SP6	MP	LS	SH

Updates

All headings containing updated or new material will be highlighted.

Suggestion / Problems

If you have a suggestion about this manual or the system to which it refers please report it to the AVEVA Group Solutions Centre at gsc@aveva.com

This manual provides documentation relating to products to which you may not have access or which may not be licensed to you. For further information on which products are licensed to you please refer to your licence conditions.

Visit our website at <http://www.aveva.com>

Disclaimer

Information of a technical nature, and particulars of the product and its use, is given by AVEVA Solutions Ltd and its subsidiaries without warranty. AVEVA Solutions Ltd. and its subsidiaries disclaim any and all warranties and conditions, expressed or implied, to the fullest extent permitted by law.

Neither the author nor AVEVA Solutions Ltd or any of its subsidiaries shall be liable to any person or entity for any actions, claims, loss or damage arising from the use or possession of any information, particulars or errors in this publication, or any incorrect use of the product, whatsoever.

Trademarks

AVEVA and Tribon are registered trademarks of AVEVA Solutions Ltd or its subsidiaries. Unauthorised use of the AVEVA or Tribon trademarks is strictly forbidden.

AVEVA product names are trademarks or registered trademarks of AVEVA Solutions Ltd or its subsidiaries, registered in the UK, Europe and other countries (worldwide).

The copyright, trademark rights or other intellectual property rights in any other product, its name or logo belongs to its respective owner.

Copyright

Copyright and all other intellectual property rights in this manual and the associated software, and every part of it (including source code, object code, any data contained in it, the manual and any other documentation supplied with it) belongs to AVEVA Solutions Ltd. or its subsidiaries.

All other rights are reserved to AVEVA Solutions Ltd and its subsidiaries. The information contained in this document is commercially sensitive, and shall not be copied, reproduced, stored in a retrieval system, or transmitted without the prior written permission of AVEVA Solutions Limited. Where such permission is granted, it expressly requires that this Disclaimer and Copyright notice is prominently displayed at the beginning of every copy that is made.

The manual and associated documentation may not be adapted, reproduced, or copied in any material or electronic form without the prior written permission of AVEVA Solutions Ltd. The user may also not reverse engineer, decompile, copy or adapt the associated software. Neither the whole nor part of the product described in this publication may be incorporated into any third-party software, product, machine or system without the prior written permission of AVEVA Solutions Limited or save as permitted by law. Any such unauthorised action is strictly prohibited and may give rise to civil liabilities and criminal prosecution.

The AVEVA products described in this guide are to be installed and operated strictly in accordance with the terms and conditions of the respective licence agreements, and in accordance with the relevant User Documentation. Unauthorised or unlicensed use of the product is strictly prohibited.

Printed by AVEVA Solutions on 25 April 2012

© AVEVA Solutions and its subsidiaries 2001 – 2009

AVEVA Solutions Ltd, High Cross, Madingley Road, Cambridge, CB3 0HB, United Kingdom.

1	Introduction	7
1.1	Aim.....	7
1.2	Objectives	7
1.3	Prerequisites	7
1.4	Course Structure.....	7
1.5	Using this Guide.....	7
2	.NET Customisation Overview.....	9
2.1	PMLNetCallable Classes	9
2.2	Common Application Framework (CAF).....	9
2.2.1	.NET Addins	10
2.2.2	Customising Application UI	11
3	Calling PMLNetCallable Classes	13
3.1	Purpose.....	13
3.2	Preparations	13
3.3	PML Form – Hosting .NET Controls	13
3.4	Using PMLNet Objects from PML.....	14
3.5	Using PMLNet Controls from PML	14
3.6	Handling PMLNet Control's Events.....	15
3.7	Example of Using NetGridControl and PMLFileBrowser	16
3.7.1	Form Preparation	16
3.7.2	NetDataSource – Binding Grid to Data	17
3.7.3	Collecting Elements	18
3.7.4	Exporting Report to Excel	18
3.7.5	Modifying Grid Elements	20
4	Defining PMLNetCallable Classes.....	21
4.1	Creating C# project.....	21
4.2	Defining Namespace.....	21
4.3	Defining NetMessageBox Class	22
4.4	Exposing Assemblies, Classes and Methods to PML.....	22
4.5	Default Constructor	23
4.6	Assign() Method	23
4.7	Class Members and Properties	23
4.8	Overloading Methods	24
4.9	Data Types	24
4.10	Building and Installing PMLNetCallable Assembly	24
4.11	Debugging C# Projects	25
4.12	Testing PMLNetCallable Objects.....	25
4.13	Creating PMLNetCallable Control	26
4.14	Raising Events	27
4.15	Raising Exceptions.....	28
	Exercise: Extending PML with RegularExpressions Engine	29
5	.NET Addin Structure.....	31
5.1	Creating .NET Addin Visual C# Project.....	31
5.2	IAddin Interface.....	32
5.3	Registering an Addin	32
5.4	Deploying an addin on network location.....	33
5.5	Creating Addin Commands.....	34
5.6	Registering Addin Commands	35
5.7	Enabling and Disabling Addin Commands	35
6	UIC Files.....	36
6.1	Creating New UIC File.....	36
6.2	Registering UIC Files	36
6.3	Modifying UIC Files.....	37
6.4	Tools with Predefined Values	38
6.5	Project and User UIC File	39
7	Extending Addin Functionality.....	40
7.1	Using Forms	40
7.2	Managing Command Checked State.....	41

7.3	Using PMLNetCallable Controls	42
8	Database Interface	44
8.1	DbElement	44
8.1.1	Getting instance of DbElement	44
8.1.2	Current Element	44
8.1.3	Getting Element Type	45
8.1.4	Navigating	45
8.2	Element Attributes	46
8.2.1	Getting an Attribute	46
8.2.2	Attribute Qualifier.....	46
8.2.3	Setting an Attribute.....	47
8.3	Creating Elements.....	47
8.4	Deleting Elements	47
8.5	Moving Elements.....	47
	Exercise: Creating Panel Element	48
9	Collections and Filters	50
9.1	Collection.....	50
9.2	Filters	50
9.2.1	TypeFilter	50
9.2.2	AttributeFalse/TrueFilter.....	50
9.2.3	AttributeRefFilter	51
9.2.4	And/OrFilter.....	51
9.2.5	BelowFilter.....	51
9.2.6	CustomFilter - using BaseFilter.....	51
	Exercise: Filtering panel elements	52
10	Miscellaneous Tools.....	54
10.1	Database Expressions.....	54
10.2	MDB/DB Operations.....	54
10.2.1	Accessing Current Project and MDB.....	54
10.2.2	Opening Project.....	54
10.2.3	Opening MDB.....	55
10.2.4	Accessing DBs	55
10.2.5	Simple Transaction Mechanism	55
10.3	Invoking PDMS Commands	55
10.4	Getting Information about Current Project and Session	56
11	PdmsStandalone Interface.....	58
11.1	Standalone Customization Structure.....	58
11.2	Preparing Runtime Environment.....	59
12	Hull API	60
	Appendix A – Source code	62
	Appendix A1 – Example of using GridControl and PMLFileBrowser.....	62
	myForm.pmlfrm.....	62
	Appendix A2 – Example of enhancing PML with PMLNetCallable components.....	64
	NetMessageBox.cs	64
	NetCalendar.cs	64
	myCalendarForm.pmlfrm	65
	Appendix A3 – Regular expressions support in PML.....	67
	NetRegEx.cs	67
	Appendix A4 – MyFirstAddin source code	68
	MyFirstAddin.cs	68
	CreatePanelCmd.cs.....	68
	PanelSurfacFilter.cs.....	70
	FilterPanelsCmd.cs.....	71
	PanelsCtrl.cs.....	72
	Appendix A5 – MyFirstStandalone code	73
	Program.cs.....	74

1 Introduction

This manual is designed to give an introduction to the AVEVA .NET customisation. There is no intention to teach software programming but provide instructions on how to customise the AVEVA application using the .NET platform and the C# language.



This training guide is supported by the reference manuals available within the products installation folder. References will be made to these manuals throughout the guide.

1.1 Aim

To provide the participants with enough knowledge to efficiently understand the following:

- How the .NET platform can be used to customise the AVEVA application
- How to create AVEVA addins
- How to customise the applications UI
- Use of Addins to customise the environment
- How to extend PML functionality with C# assemblies

1.2 Objectives

At the end of this training the participants will have:

- A broad overview of customising the AVEVA software using the .NET platform.
- Knowledge of creating .NET addins.
- Knowledge of creating PML callable .NET objects.
- Ability to work with databases.
- An overview of creating custom Toolbars and Menu bars.

1.3 Prerequisites

The participants must have completed an AVEVA Basic Foundations Course and be familiar with the AVEVA applications. AVEVA Marine or Plant and Visual Studio 2005 or 2008 must be installed. Moreover, have knowledge of PML and object oriented programming with some experience with .NET coding using MS Visual Studio 2008. Some .NET framework knowledge is a plus, but not necessary.

1.4 Course Structure

The training will consist of oral and visual presentations, demonstrations, worked examples and practical exercises. Each trainee will be provided with some example files to support this guide. Each workstation will have a training project, populated with model objects. This will be used by the trainees to practice their methods, and complete the set exercises.

1.5 Using this Guide

Certain text styles are used to indicate special situations throughout this document. Below is a summary;

Menu pull downs and button press actions are indicated by **bold dark turquoise text**.

Information the user has to Key-in '**will be red and BOLD**'



Additional information will be highlighted



Reference to other documentation will be separate

System prompts should be bold and italic in inverted commas i.e. '**Choose function**'

Example files or inputs will be in the `courier new font` with colours and styles used as before.

2 .NET Customisation Overview

This chapter gives a general overview of the mechanisms that can be used to customise an application via PML, addins, commands and user interface customisation (UIC) files

2.1 PMLNetCallable Classes

PMLNET allows the user to write classes in .NET, and to use them directly from PML.

This is done by decorating the .NET classes and methods with the PMLNetCallable attribute, which will be described later on in this guide.

Using the .NET framework from PML also allows the user to build their own controls and use them on PML forms and debug code at runtime.

2.2 Common Application Framework (CAF)

The CAF provides a number of interfaces for .NET programmers to access various services which support both application development and customisation. There are two subsystems implemented in namespaces:

- **Aveva.ApplicationFramework**
- **Aveva.ApplicationFramework.Presentation**

The CAF manages and provides access to application services using the manager objects listed below:

- **ServiceManager** – manages all application services.
- **AddinManager** – manages application addins (plugins).
- **SettingsManager** – manages application settings.
- **CommandBarManager** – manages application Toolbars and Menus.
- **CommandManager** – manages application commands used to interact with GUI.
- **ResourceManager** – provides Addins with a simplified mechanism to access its private localizable resources as well as providing access to a set of standard icons.
- **WindowManager** – provides access to the main application window, the StatusBar, the SplashScreen and a collection of MDI and docked windows.

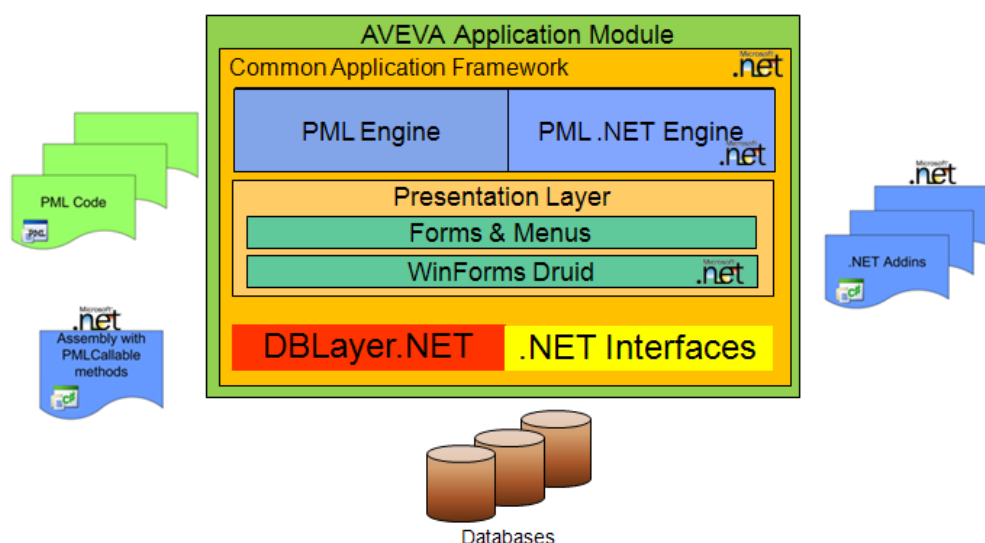


Figure 1

The figure above shows the AVEVA Customisation Architecture using PML, PML.NET and Addins which are managed by the CAF. PML.NET and Addins may use .NET interfaces to access the database and other functionality.

2.2.1 .NET Addins

The CAF is used to create application .NET addins (plugins) for customisation. A .NET addin is an assembly with specific implementation added to an application. The assembly must contain one (and only one) class which implements the IAddin interface, which allows an AVEVA application to register it using the AddinManager. The Addin then uses .NET Interfaces to communicate with the application database and user interactions (selection, CE changing, dealing with geometry).

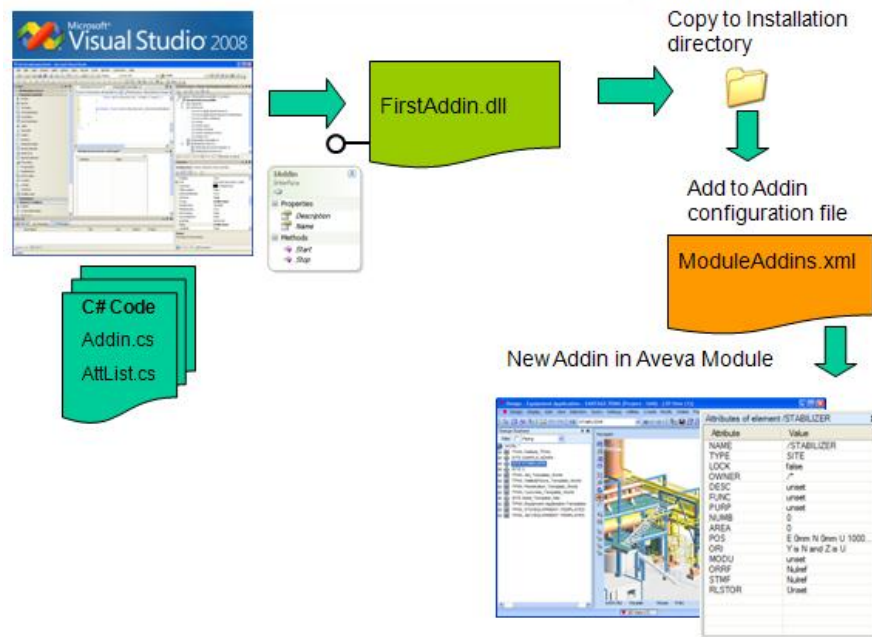


Figure 2

The above figure shows how to create an AVEVA application addin. It describes each step of the creation - starting from implementation - building and configuring an application to load the new addin.

2.2.2 Customising Application UI

The UI is customised using reusable tools like menu items and buttons and separate commands. The tools are associated with commands which are executed when the tool is clicked.

The below figure 3 shows relations between command implementation and GUI items like tools, menu items and toolbar buttons.

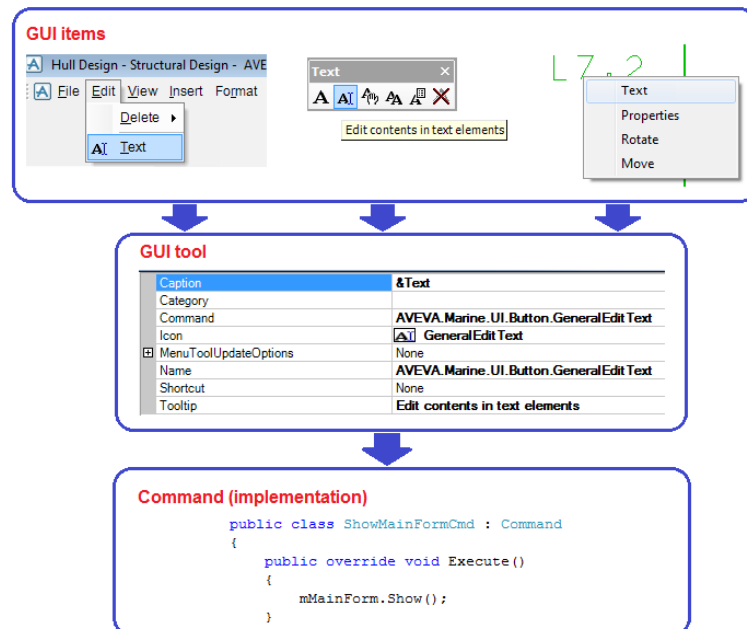


Figure 3: Relationship between commands and user interface items.

Commands are usually added by an addin.

All new commands provided by an addin (as well as other existing commands) can then be associated with a number of different tools

- ButtonTool
- ComboBoxTool
- ContainerTool
- FontListTool
- LabelTool
- ListTool
- MdiWindowListTool
- MenuTool
- PopupColorPickerTool
- PopupContainerTool
- StateButtonTool
- TaskPaneTool
- TextBoxTool

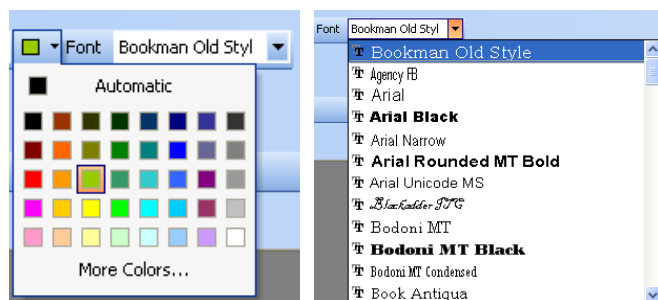


Figure 4: Examples of PopupColorPickerTool and FontListTool

Association of a given command with a given tool can be done programmatically via the CAF or by using the Customisation tool shown in figure 5 below. This will then modify the currently active customisation .uic file.

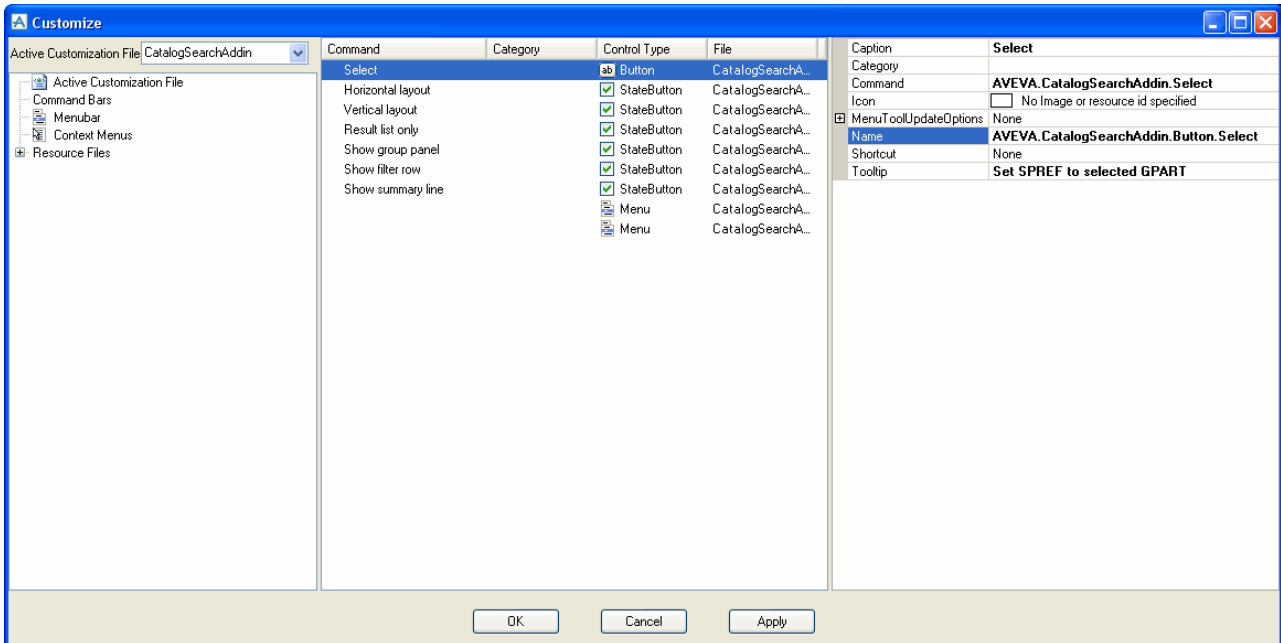


Figure 5

3 Calling PMLNetCallable Classes

3.1 Purpose

The purpose of this chapter is to understand how PML can call .NET assemblies, host .NET controls and handle .NET events. At the end of the chapter the trainees will be asked to prepare a .NET assembly that provides a PML form with a calendar control and a regular expression parser.

3.2 Preparations

In this chapter we are going to use a mixture of PML and C# code. It is recommended to organize the files depending on their purpose. Create a directory structure like the one below. The training directory can be created anywhere that is suitable for the user:

Training	
pmllib	- pml files
functions	- pml functions (.pmlfnc files)
forms	- pml forms (.pmlfrm files)
objects	- pml objects (.pmlobj files)

After creating this directory structure, add the path to the PMLLIB environment variable. This variable is used by the system to locate PML files and can contain multiple paths. For training purposes it can be set in marine.bat (pdms.bat for plant users) directly before the monitor module is executed.

Example:

```
set PMLLIB=c:\Training\pmllib;%PMLLIB%

echo running: %monexe%\mon %args%
cmd/c "%monexe%\mon" %args%
```

i Please note that whenever a new pml file is stored under one of the PMLLIB paths it is necessary to reindex the files. It is automatically done during system startup and on demand by using the **pml rehash all** command.

It is also possible to check the value of the environment variable from the AVEVA command line:

```
q evar PMLLIB
```

3.3 PML Form – Hosting .NET Controls

For the examples in this chapter create a PML form which will be the base for every exercise. First create a new file and name it myForm.pmlfrm (extension is important). Then place it in the training pmllib\forms folder. Make sure that the file is indexed by retyping

```
pml rehash all
```

Add the following code to the file. This is the simplest form definition

```
setup form !!myForm resize
--form initialization method
exit

define method .myForm()
--default constructor here
endmethod
```

To show the form type the following on the command line:

```
show !!myForm
```

3.4 Using PMLNet Objects from PML

In order to instantiate and call methods on a PML.NET object the assembly must be loaded and class definitions created. This is done by importing the assembly using the IMPORT syntax.

The default location of .NET assemblies is the AVEVA executables directory. The PML syntax below is used to import the assembly. It is recommended to place import statements at the beginning of each PML source code file to ensure that the PMLNet class definitions have been created

```
import 'GridControl'  
handle any  
endhandle
```

i This should be done only once as it remains in the memory to the end of the session. The next import attempt will result in an error message. As the example form can be shown several times it is necessary to catch the 'Assembly already loaded' exception via **handle any** block.

i If the assembly is located in a directory other than PDMSEXE then the full path can be specified

PML.NET objects defined in a given assembly will usually be organized into namespaces. The namespace can be treated like a container where the given class is defined and is useful for avoiding name conflicts. Before instantiating an instance of the class it is necessary to tell the system which namespace to use. The code below shows how this is done in PML.

```
using namespace 'Aveva.Pdms.Presentation'
```

This statement must appear in all PML methods before an object is going to be used.

3.5 Using PMLNet Controls from PML

A PMLNet control is a special kind of PMLNet object which provides some graphical user interface and can be hosted on a PML form.

i Note that using namespace statement must exist in all methods where the control is going to be used. Form **setup block** can be treated as one of the form methods so it must also contain using namespace statements.

A PMLNet control is hosted by a PML form inside a **container** element. The container is a special control that works like a place holder for future PMLNet control. The container is created using the PML code below:

```
container .GridFrame PMLNetCONTROL anchor ALL at 0 0 width 50 height 10
```

where:

- **.GridFrame** is the PML name of the container member
- **PMLNetCONTROL** is a type of container (key word).

The PMLNet control that will be hosted within the container is normally declared as a member of the form as follows

```
member .GridCtrl is NetGridControl
```

where:

- **.GridCtrl** is the name of the member
- **NetGridControl** is the name of PMLNet control.

It is then necessary to create an instance of the control and associate it with the control container. The most suitable place for this is the form's constructor:

```
!this.GridCtrl = object NetGridControl()  
!this.GridFrame.Control = !this.GridCtrl.handle()
```

i *Note that an association between container and control is done via the handle provided by the control object.*

3.6 Handling PMLNet Control's Events

PMLNet controls can expose events that are raised in particular circumstances. Please refer to the AVEVA C# Grid Control documentation for a list of events supported by the NetGridControl.

In PML it is then possible to define event handlers for the control's events.

Each of the PMLNet controls provide two methods for handling events:

- **AddEventHandler**(EventName, ObjectInstance, ObjectMethodName)
- **RemoveEventHandler**(EventName, HandlerId)

The first method should be used for registering a new event handler. The first argument of this method provides the event name that is going to be handled. The next two arguments provide information about the instance of the object that provides the handler method and the handler method name.

The below shows an example of how to register a handler for the **AfterSelectChange** event, provided by NetGridControl.

```
!handlerId = !this.GridCtrl.AddEventHandler('AfterSelectChange', !this,  
                                           'OnAfterSelectChange')
```

This is normally added in the form's constructor following the statement instantiating the NetGridControl member.

The method AddEventHandler returns a handle that can later be used to remove the handler.

i *Notice that method name is specified without parenthesis like it is done in case of pml control's callbacks.*

The event handler method follows the same format in all events provided by PMLNet controls. It accepts only one argument of type array. The array may contain a number of different arguments relevant to the given event.

Example of grid AfterSelectChange event handler:

```
define method .OnAfterSelectChange(!args is array)  
  q var !args  
endmethod
```

In most cases control events handlers are methods of a PML form that hosts the control. In this case there is no need to remove the handler as the form (and its handler method) is deleted from the memory together with the control that raises events. However, it could happen when the handler object is removed from memory and the control object still remains or there is no more need to listen to the event. In this case it is necessary to remove the registered handler by the RemoveEventHandler method.

Example:

```
!this.GridCtrl.RemoveEventHandler('AfterSelectChange', !handlerId)
```

3.7 Example of Using NetGridControl and PMLFileBrowser

AVEVA provides PML users with two PMLNet controls; a grid control called **NetGridControl** and file browser dialogue called **PMLFileBrowser**. These controls are implemented in the GridControl and PMLFileBrowser assemblies respectively. Both of the controls are defined in the same namespace: **Aveva.Pdms.Presentation**.

The purpose of this example is to provide the user with a PML form that displays a collection of PDMS PANEL objects found under the current element, which allows the user to change some attributes and also provides export to Excel functionality.

It is possible to change the type of collected elements depending on trainee preferences.

3.7.1 Form Preparation

The example form should look like below:

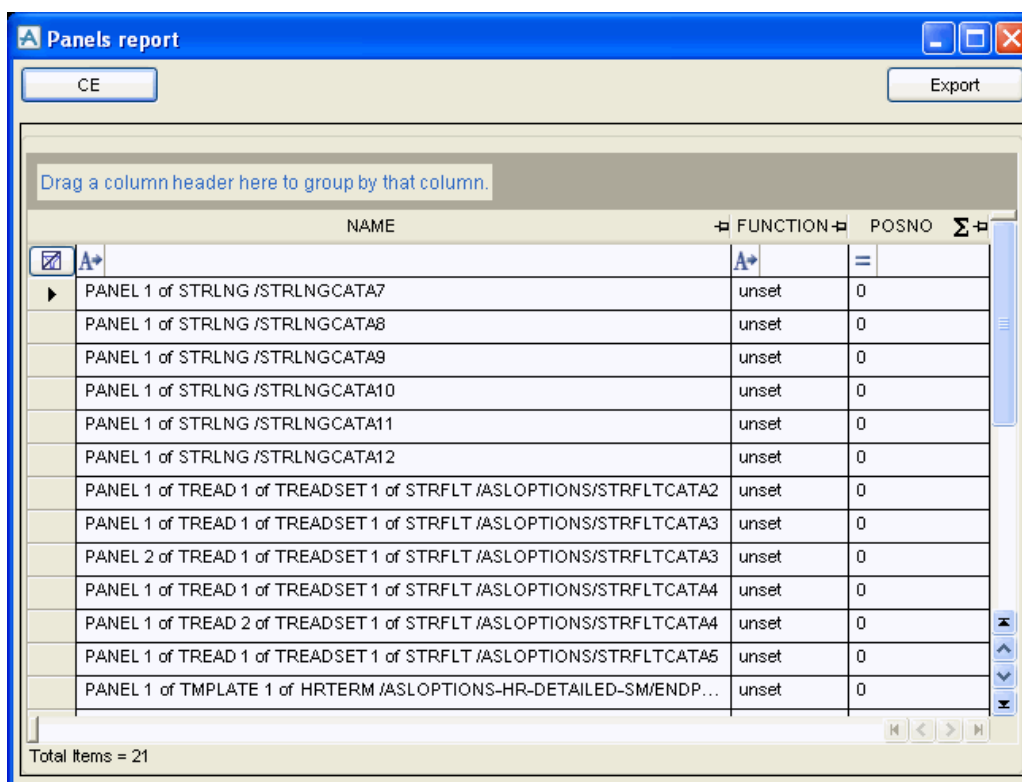


Figure 6

First of all it is necessary to import both assemblies to define the controls that will be hosted on the form. Add the following lines to the top of your myForm.pmlfrm file:

```
import 'GridControl'
handle any
endhandle

import 'PMLFileBrowser'
handle any
endhandle
```

We can then create and position all the controls and declare the member for the GridControl object. Add these lines into your form setup block:

```
using namespace 'Aveva.Pdms.Presentation'

title 'Panels report'
```



```
button .CE 'CE' at 0 0 width 10 height 1 callback '!this.OnCE()'
button .Export 'Export' at 39 0 anchor R+T width 10 height 1
                                callback '!this.OnExport()'
container .GridFrame PMLNetCONTROL anchor ALL at xmin.CE ymax.CE+0.5
                                width 50 height 10

member .GridCtrl is NetGridControl
```

i Be aware that PML syntax does not allow breaking one statement into multiple lines. All breaks given in this training are because of page limits and for better visualization.

When controls are declared create a **GridControl** instance and associate it with the container in your form constructor method:

```
using namespace 'Aveva.Pdms.Presentation'

!this.GridCtrl = object NetGridControl()
!this.GridFrame.Control = !this.GridCtrl.handle()
```

At the end of file define callback methods **OnCE()** and **OnExport()** and leave them empty:

```
define method .OnCE()
endmethod

define method .OnExport()
endmethod
```

Then, on the command line type:

```
show !!myForm
```

to check that the form is defined correctly and can be shown

i Note that PML forms are global objects so any changes done to the form layout and interface require killing the form before showing it again: kill !!myForm

3.7.2 NetDataSource – Binding Grid to Data

AVEVA provides the user with a **NetDataSource** object that provides a data source that can be bound to a grid. The **NetDataSource** object is designed to work with dabacon elements. It is very easy to initialize the data set object with elements and list of attributes names that are going to be displayed as grid columns. The **NetDataSource** object is implemented in the **GridControl** assembly and is defined in the **Aveva.Pdms.Presentation** namespace.

One of its constructor methods takes 3 arguments:

- Name of data set
- Array of attributes given as strings
- Array of elements names given as strings

When an instance of **NetDataSource** is ready we can bind our grid control to this instance by calling grid method **BindToDataSource**.

Add the following lines to your **OnCE()** method:

```
!source = object NetDataSource('Elements', !columns, !elements)
!this.GridCtrl.BindToDataSource(!source)
```


3.7.3 Collecting Elements

Before showing our form we need to provide our data source with an array of attributes (columns) and an array of collected elements names.

In our case the array of attributes can be fixed and we can use any attribute name valid for the PANEL object.

An example of such an array:

```
!columns = array()
!columns.append('NAME')
!columns.append('FUNCTION')
!columns.append('POSNO')
```

 *Note that this set of attributes also can be used for other types of design elements as they are common. In the case that the given attribute is not valid for certain types of elements the grid control will display an exclamation icon as a value of such a column.*

To collect elements we can use **!!CollectAllFor(...)** global function. Its arguments are:

- Type of element (string)
- Filter expression (string)
- Scope element (dbref)

The function returns an array of dbref's of all elements found under the given scope (parent element).

To convert this array of dbref's to an array of strings representing names we can use Evaluate method of array object. Our **OnCE()** method should look like this:

```
define method .OnCE()
    using namespace 'Aveva.Pdms.Presentation'

    !columns = array()
    !columns.append('NAME')
    !columns.append('FUNCTION')
    !columns.append('POSNO')

    !elements = !!CollectAllFor('PANEL', '', !!ce)
    !elements = !elements.Evaluate(object BLOCK('!elements[!evalIndex].name'))
    !source = object NetDataSource('Elements', !columns, !elements)
    !this.GridCtrl.BindToDataSource(!source)
endmethod
```

3.7.4 Exporting Report to Excel

The NetGridControl provides a method that exports the current grid content to an Excel file given as an argument:

```
!this.gridCtrl.SaveGridToExcel(!fileName)
```

Before calling this method it is necessary to prompt the user for a file name. Here we can use the PMLFileBrowser object.

To create an instance of PMLFileBrowser we need to specify whether we want an 'OPEN' or 'SAVE' dialogue:

```
!fileBrowser = object PMLFileBrowser('SAVE')
```

The object's method **Show(...)** displays the file open/save dialogue box and allows the user to select the file name and location where to store the file.

```
!fileBrowser.Show('C:\\',
```

```
'report.xls',
'Open File',
false,
'Excel files (*.xls)|*.xls|All files (*.*)|*.*',
1)
```

The arguments are:

- Initial directory
- Proposed file name
- Dialogue title
- Check file exists flag
- Extension filter
- Index of initial extension filter

The **File()** method of **PMLFileBrowser** object returns a full path of the selected file. If the user cancels the file selection the method returns an empty string.

Now we are ready to implement our **OnExport()** method:

```
define method .OnExport()

    using namespace 'Aveva.Pdms.Presentation'

    !fileBrowser = object PMLFileBrowser('SAVE')
    !fileBrowser.Show('C:\\', 'report.xls', 'Save as', false, 'Excel files
                        (*.xls)|*.xls|All files (*.*)|*.*', 1)

    !fileName = !fileBrowser.File()

    if (!fileName neq '') then
        !this.GridCtrl.SaveGridToExcel(!fileName)
    endif

endmethod
```

3.7.5 Modifying Grid Elements

The **NetGridControl** can be used for modifying an element's attributes. Whenever an attribute is modified the grid control raises the **BeforeCellUpdate** event. Handling of this event is useful for validating the cell value and updating the element.

Before adding the event handler we need to set the grid control into 'editable' and 'allow updates' mode. This can be achieved by calling the **EditableGrid(...)** and **Updates(...)** methods in the form constructor just after the grid object is instantiated:

```
!this.GridCtrl.EditableGrid(true)  
!this.GridCtrl.Updates(true)
```

Now we can add the new event handler:

```
!this.GridCtrl.AddEventHandler('BeforeCellUpdate', !this, 'OnBeforeCellUpdate')
```

To perform an element update we need to define the **OnBeforeCellUpdate** handler method where validation should be performed. To make an update of the element's attribute we can use the **DoDabaconCellUpdate** method of the grid control. It takes exactly the same argument as provided to the handler:

```
define method .OnBeforeCellUpdate(!args is array)  
    -- validate new value  
  
    -- and perform update  
    !this.GridCtrl.DoDabaconCellUpdate(!args)  
endmethod
```

 *The source code of the provided example can be found in Appendix A.*

4 Defining PMLNetCallable Classes

In this chapter you will learn how to create **PMLNetCallable** C# classes and use them in PML exposing classes and methods using the PMLNetCallable attribute. The training material is based on Visual Studio 2008, however, any other VS version that supports .NET is allowed. Most of the chapter material will be explained using an example project.

The purpose of the example is to provide a MessageBox object that can be used from PML to display message boxes and a Calendar control that can be used on a PML form.

4.1 Creating C# project

Before starting Visual Studio create the directory **projects** under your **Training** directory. The **Training** directory structure should look like this:

Training	
pmllib	- pml files
functions	- pml functions (.pmlfnc files)
forms	- pml forms (.pmlfrm files)
objects	- pml objects (.pmlobj files)
projects	- C# projects

PMLNetCallable classes are implemented within .NET assemblies. Follow the steps below to create a C# assembly:

1. Start Visual Studio session
2. Choose **File->New->Project** from the VS main menu
3. Select **Visual C#** as the project type
4. Select **Class Library** as the project template
5. Setup NetTools as the project name
6. Select the Training\projects directory as the project location
7. Make sure that **.NET Framework 2.0** is selected. Visual C# Express users may set it up later in the project settings as it is not displayed in the new project window.
8. Click the **OK** button to create the project
9. In the Solution Explorer window rename the Class1.cs project file to NetMessageBox.cs. Answer **Yes** to update the class name.

4.2 Defining Namespace

It is recommended to use namespaces following the format below:

```
CompanyName.ModuleName.Functionality
```

Using such namespaces will prevent name clashes between different modules from different providers.

Examples:

```
Aveva.ContourEditor.Geometry
Aveva.ContourEditor.Commands
```

The default namespace can be changed in the project settings window on the Application tab. Change the default namespace of our example to **Training.NetTools.UI**

Do not forget to also change it in the NetMessageBox.cs file as VS will not update it automatically.

4.3 Defining NetMessageBox Class

Define the default constructor in your **NetMessageBox** class and public **Show(...)** method that takes two strings as arguments: message and message box title. Use the **System.Windows.Forms.MessageBox.Show(...)** method to display the message box within your **Show(...)** method. To be able to use the .NET MessageBox class add a reference to the **System.Windows.Forms** .NET component.

```
using System;
using System.Collections.Generic;
using System.Text;

using System.Windows.Forms;

namespace Training.NetTools.UI
{
    public class NetMessageBox
    {
        public NetMessageBox()
        {
        }

        public void Show(string message, string title)
        {
            MessageBox.Show(message, title);
        }
    }
}
```

4.4 Exposing Assemblies, Classes and Methods to PML

To expose a class and its methods to PML it is necessary to decorate all the classes and methods with the **[PMLNetCallable()]** attribute. The attribute is implemented in **PMLNet.dll** assembly in the **Aveva.PDMS.PMLNet** namespace. The PMLNet.dll can be found in the AVEVA executables directory. Exposing an assembly to PML must be done on 3 levels: assembly, class and method.

To mark the assembly with **[PMLNetCallable()]** attribute add the following lines to **AssemblyInfo.cs** file in your project.

```
using Aveva.PDMS.PMLNet;
[assembly: PMLNetCallable()]
```

Exposing classes and methods is done by adding the **[PMLNetCallable()]** attribute just before the class or method definition:

```
[PMLNetCallable()]
public class NetMessageBox
{
    [PMLNetCallable()]
    public NetMessageBox()
    {
    }
}
```

 *Do not forget to use the Aveva.PDMS.PMLNet namespace whenever PMLNetCallable attribute is used.*

4.5 Default Constructor

The C# class that is going to be exposed to PML must have a default constructor declared. It must be public and marked with the [PMLNetCallable()] attribute.

```
[PMLNetCallable()]
public NetMessageBox()
{ }
```

4.6 Assign() Method

Each C# class that is going to be exposed to PML must have the **Assign(...)** method declared. The method takes only one argument and it is an instance of the same class:

```
[PMLNetCallable()]
public class NetMessageBox
{
    [PMLNetCallable()]
    public void Assign(NetMessageBox that)
    {
    }
}
```

Both methods, default constructor and Assign(...) are used internally by the PML engine whenever a PML assignment takes place:

```
!a = object NetMessageBox()
!b = !a
```

In the code above the PML engine will create an !b instance using the default constructor of NetMessageBox class and then call its Assign(...) method passing instance to !a as an argument.

The purpose of the Assign(...) method is therefore to copy any state from one instance to another.

4.7 Class Members and Properties

There is no way to expose class members directly to PML. These may be exposed as properties. Each property is translated into two methods of the same name as the property. One of the methods takes no arguments and returns a property value. This is an equivalent of the property **get()** method. The other one takes one argument of the same type as the property and returns no value. This is an equivalent of the property **set(...)** method.

As an exercise, add a new member to your NetMessageBox class:

```
private string mDefaultTitle = "Error";
```

Then wrap it with property DefaultTitle and expose it to PML:

```
[PMLNetCallable()]
public string DefaultTitle
{
    get { return mDefaultTitle; }
    set { mDefaultTitle = value; }
}
```

 *Do not forget to update your Assign method to copy the value of mDefaultTitle variable.*

4.8 Overloading Methods

The PMLNet interface fully supports method overloading.

Provide the NetMessageBox class with another Show(...) method that takes only the message string as an argument and uses mDefaultTitle as the window title.

```
[PMLNetCallable()]
public void Show(string message)
{
    MessageBox.Show(message, mDefaultTitle);
}
```

Create another constructor that takes the default title as an argument:

```
[PMLNetCallable()]
public NetMessageBox(string title)
{
    mDefaultTitle = title;
}
```

4.9 Data Types

All methods and properties that are exposed to PML can pass only PML basic data types. Table below shows the PML basic data types and their corresponding C# equivalents.

<i>PML data type</i>	<i>C# data type</i>
String	string
Real	double
Array	System.Collections.Hashtable
Boolean	bool

For Hashtables the key is of type double and represents the PML Array index.

In addition, any PMLNet objects can be used as data types of methods arguments and return values.

 *Note that type integer is not supported by the PML. Use double instead.*

4.10 Building and Installing PMLNetCallable Assembly

To build the project select the appropriate configuration (Debug by default) and choose **Build->Build Solution** from the VS main menu or press **Ctrl+Shift+B** keys.

An assembly file is created in the project bin\Debug or bin\Release directory depending on the current configuration.

As the PML **import** statement accepts the full path library name it can be loaded directly from the project target directory:

```
import 'C:\Training\Projects\NetTools\NetTools\bin\Debug\NetTools'
```

As the path is rather long it is better to copy the target file to the AVEVA executables directory. This can be done automatically as a Post-Build event (Project settings -> Build events).

Example:

```
copy $(TargetPath) C:\AVEVA\Marine\OH12.0.SP5\$(TargetFileName)
```


When this is done the import statement can be reduced to:

```
import 'NetTools'
```

4.11 Debugging C# Projects

Visual Studio allows debugging by attaching a debug session to the running process. To do this start AVEVA product and from the VS main menu choose **Debug->Attach to process**. The following window will be displayed:

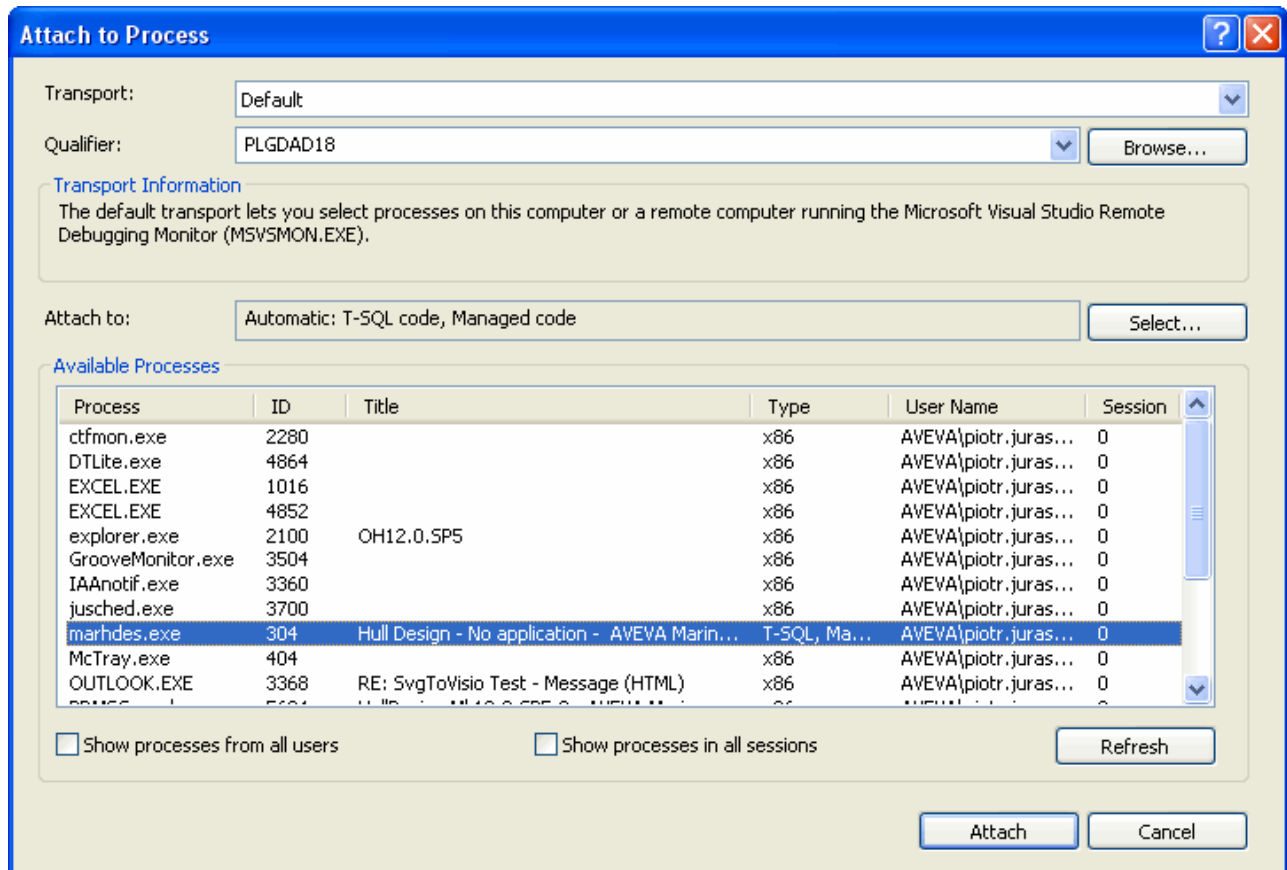


Figure 7

From the **Available Processes** list select the appropriate AVEVA product process and click the **Attach** button.

From now on it is possible to break the execution at any breakpoint in your project.

4.12 Testing PMLNetCallable Objects

It is not necessary to create a PML macro that uses PMLNetCallable objects exposed by your assembly. This can be tested directly from the Command Line window.

Type the following at the command line:

```
import 'NetTools'
using namespace 'Training.NetTools.UI'
!m = object NetMessageBox('Warning')
!m.Show('Settings file not found!')
```

As an exercise set a breakpoint in the Assign and default constructor methods and observe how the PML engine makes calls to these methods when an assign is made:

```
!m1 = !m
```

4.13 Creating PMLNetCallable Control

It is possible to extend the PML form gadgets set with user custom controls provided by a PMLNetCallable assembly. The C# class defining the control must be derived from UserControl and exposed to PML using the [PMLNetCallable()] attribute.

As an example we are going to create a calendar control that is provided by the .NET environment and not supported in the PML standard controls set.

Add a new class called NetCalendar derived from the UserControl. To do this choose **Project->Add User Control** from the VS main menu. When the new user control is created VS displays it in design mode. Next drag the MonthCalendar control from the VS Toolbox window and drop it onto your control's design area. Adjust the control's size to fit the calendar control. Change the name of the calendar control to **NetCalendar** in the Properties window.

Switch VS to source code mode by choosing **View Code** from the context menu displayed on a right click in the control's design area.

Mark class definition and constructor with the [PMLNetCallable()] attribute.

```
namespace Training.NetTools.UI
{
    [PMLNetCallable()]
    public partial class NetCalendar : UserControl
    {
        [PMLNetCallable()]
        public NetCalendar()
        {
            InitializeComponent();
        }
    }
}
```

Provide class with the Assign(...) method and expose it to PML :

```
[PMLNetCallable()]
public void Assign(NetCalendar that)
{
}
```

Create a Date read only property of type string and return the selected date in the preferred format:

```
[PMLNetCallable()]
public string Date
{
    get { return CalendarCtrl.SelectionStart.ToLongDateString(); }
}
```

After building your project the control is ready to use.

Prepare a PML form called myCalendarForm and host the NetCalendar control in your form. Add the button **Get date** and text gadget to your form. When the button is pressed retrieve the currently selected date and display it in the text gadget.

The form should look like the one below.

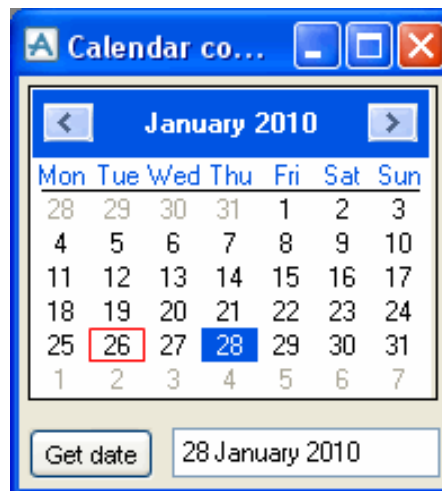


Figure 8

4.14 Raising Events

PMLNetCallable components can expose events to PML. There is one type of event declared in PMLNet assembly under Aveva.PDMS.PMLNet namespace.

Event declaration:

```
[PMLNetCallable()]
public event PMLNetDelegate.PMLNetEventHandler EventName;
```

As PMLNetEventHandler type is defined by the PMLNet component there is no need to expose it to PML with [PMLNetCallable()] attribute.

To raise an event it is necessary to declare an ArrayList containing the (System.Collections namespace) data that is going to be sent to the event handler.

```
if (EventName != null)
{
    ArrayList args = new ArrayList();
    args.Add("value 1");
    args.Add("value 2");

    EventName(args);
}
```

We can easily enhance our NetCalendar control with an **DateSelected** event. To do this - declare the event in NetCalendar class:

```
public event PMLNetDelegate.PMLNetEventHandler DateSelected;
```

Add the C# event handler to the DateSelected event provided by .NET MonthCalendar control. To do this, switch to design mode, select our calendar control and double click on the DateSelected event in the Properties windows. Visual Studio will automatically generate an event handler method. The method shall be used to raise our PMLNet event:

```
private void CalendarCtrl_DateSelected(object sender, DateRangeEventArgs e)
{
    if (DateSelected != null)
    {
        ArrayList args = new ArrayList();
        args.Add(CalendarCtrl.SelectionStart.ToLongDateString());
        DateSelected(args);
    }
}
```

```
    }  
}
```

Build the project and then add an event handler to your myCalendarForm definition.

Add this line to form constructor:

```
!this.CalendarCtrl.AddEventHandler('DateSelected', !this, 'OnDateSelected')
```

And define handler:

```
define method .OnDateSelected(!args is array)  
    !this.CurDate.val = !args[0]  
endmethod
```

4.15 Raising Exceptions

The PMLNet interface allows throwing exceptions.

Use its PMLNetException class for all exceptions that are going to be handled by PML.

```
throw new PMLNetException(1000, 1, "Error message");
```

 *All the source code provided as an example to this chapter can be found in Appendix A2*

Exercise: Extending PML with RegularExpressions Engine

There is little support for regular expressions in PML compared to .NET which provides the RegEx class under **System.Text.RegularExpressions**.

Very often model names hold a lot of information that can be easily used for filtering and locating.

Example of hull stiffener name: GA393-224010-S10P

Where:	GA393	- block name
	224010	- panel number
	S	- part type (stiffener)
	10	- stiffener number
	P	- side (S-Starboard, P-Portside, SP-Both sides)

Imagine that the user wants to filter all stiffeners valid for portside or both sides for specific block and panel number in range 220000-235000. This kind of search could be rather difficult to implement with PML.

The regular expression:

GA393-22[0-5][0-9]{3}-S[0-9]+(P|SP)

defines all the mentioned above criterias in one expression where:

[0-5]	- any character between 0-5
[0-9]{3}	- any character between 0-9 three times
(P SP)	- P or SP

Below C# code can be used to check part name against defined criteria:

```
Regex r = new Regex("GA393-22[0-5][0-9]{3}-S[0-9]+(P|SP)");
if (r.IsMatch("GA393-224010-S10P"))
{
    // name matches criteria
}
```

As a first step in our exercise extend PML with a NetRegEx object that provides PML with a .NET regular expressions engine.

 Notice that the namespace should now be *Training.NetTools.RegularExpressions*

Define a NetRegEx object constructor that accepts regular expression as an argument and an IsMatch method that takes a string as input and returns True if string matches the criteria given to the constructor method.

One of the advantages of RegEx is the use of back references. The regular expression can contain a group definition that is evaluated when a match occurs and can then be accessed to provide the user with an evaluated value.

Imagine that the user wants to get a stiffener number for each stiffener that matches the criteria defined above.

UsingRegEx it is easy to name the part of regular expression responsible for stiffener name. The name of this part of the expression (group) can be then used to get the value that matches this part of the criteria.

GA393-22[0-5][0-9]{3}-S(?<partno>[0-9]+)(P|SP)

See C# code below for better understanding:

```
Regex r = new Regex("GA393-22[0-5][0-9]{3}-S(?<partno>[0-9]+)(P|SP)");
Match m = r.Match("GA393-224010-S10P");
if (m.Success)
{
    Group g = m.Groups["partno"];
    if (g.Success)
        return g.Value; // in this case it will be "10"
}
```

Extend our example with GetGroup method that takes group name as input and returns group value or raises exception in case if group is not found.

Test your NetRegEx object directly from the AVEVA command line window:

```
import 'NetTools'
using namespace 'Training.NetTools.RegularExpressions'

!a = object NetRegEx('GA393-22[0-5][0-9]{3}-S(?<partno>[0-9]+)(P|SP)')
q var !a.IsMatch('GA393')
<BOOLEAN> FALSE

q var !a.IsMatch('GA393-224010-S10P')
<BOOLEAN> TRUE

q var !a.GetGroup('partno')
<STRING> '10'

q var !a.GetGroup('partno1')
(1000,0)Group not found!
```

 Completed exercise can be found in Appendix A3

5 .NET Addin Structure

This chapter describes how to create .NET Addins and is an extension of Chapter 2 of this guide. Here you will learn how to create a complete C# addin using the AVEVA .NET interface.

This chapter contains a number of exercises which build an addin which manages PANEL objects.

5.1 Creating .NET Addin Visual C# Project

An addin project is a .NET class library that provides at least one class defining the IAddin interface provided by Aveva.ApplicationFramework assembly and defined under namespace with the same name as assembly.

Follow the same steps as in creating the PMLNetCallable project done in chapter 4.1. Use **MyFirstAddin** as the project name.

When your project is generated rename **Class1.cs** module to **MyFirstAddin.cs** and change the default namespace to **Training.MyFirstAddin**

Add a reference to **Aveva.ApplicationFramework** to your project and then inherit your MyFirstAddin class from the **IAddin** interface declared under **Aveva.ApplicationFramework**.

```
using Aveva.ApplicationFramework;

namespace Training.MyFirstAddin
{
    public class MyFirstAddin : IAddin
    {
        #region IAddin Members

        public string Description
        {
            get { throw new NotImplementedException(); }
        }

        public string Name
        {
            get { throw new NotImplementedException(); }
        }

        public void Start(ServiceManager serviceManager)
        {
            throw new NotImplementedException();
        }

        public void Stop()
        {
            throw new NotImplementedException();
        }

        #endregion
    }
}
```

5.2 IAddin Interface

An IAddin interface defines two read only members, Name and Description which must be implemented. The first one provides the name of addin and the second the description string.

The interface also defines two other methods which must be implemented. The start method is called when addin is loaded by the CAF and Stop is called when an addin is about to be unloaded. Use these methods to initialize and release all the resources used by addin.

Provide Name and Description strings and remove throwing exceptions statements from your **Start(...)** and **Stop()** methods. Write the following message to the console window in your Start method:

```
using Aveva.ApplicationFramework;

namespace Training.MyFirstAddin
{
    public class MyFirstAddin : IAddin
    {
        #region IAddin Members

        public string Description
        {
            get { return "My first addin"; }
        }

        public string Name
        {
            get { return "MyFirstAddin"; }
        }

        public void Start(ServiceManager serviceManager)
        {
            Console.WriteLine("MyFirstAddin loaded");
        }

        public void Stop()
        {
        }

        #endregion
    }
}
```

The Start method is passed the **ServiceManager** instance. This instance can be used to get different services controlling application behaviour. Some of the services will be discussed later in this chapter.

Build the project and fix any errors.

5.3 Registering an Addin

Each AVEVA module that loads addins has a corresponding *ModuleNameAddins.xml* file. All these files are located by default in the AVEVA distribution path. The addins definition file contains a list of all addins loaded by a given module. To install an addin add its name to the list of addins:

```
<string>MyFirstAddin</string>
```

It is also possible to specify an addin file with full path:

```
<string>
    C:\Training\Projects\MyFirstAddin\MyFirstAddin\bin\Debug\MyFirstAddin
</string>
```


If the path is not given then the system will search for an addin file in the AVEVA executables directory.

 *Notice that the addin file name is given without a file extension which is always .dll by default.*

It is recommended by AVEVA to avoid changes in files provided with the AVEVA installation. The environment variable **CAF_ADDINS_PATH** may be used to give an alternative location for the **ModuleNameAddins.xml** files.

Now create a new directory called addins under your Training directory:

Training	
pmllib	- pml files
functions	- pml functions (.pmlfnc files)
forms	- pml forms (.pmlfrm files)
objects	- pml objects (.pmlobj files)
projects	- C# projects
addins	- customized *Addins.xml

Copy all *Addins.xml files from the AVEVA distribution path to the newly created addins directory. Setup the CAF_ADDINS_PATH environment variable in an appropriate .bat file used for initializing the AVEVA product. By default it is the marine.bat for marine users and pdms.bat for plant users (both files are located in the AVEVA distribution path by default).

Example:

CAF_ADDINS_PATH=c:\Training\addins

Modify your project settings to copy the target file to your Training\addins directory in a post-build event.

 *Setting up post-build event was described in chapter 4.10*

Add your addin entry to the appropriate *Addins.xml file already copied to Training\addins directory.

Example:

<string>c:\Training\addins\MyFirstAddin</string>

 *Notice that the file is given with full path as it is not stored under the AVEVA distribution path.*

Rebuild your project so that the post-build event copies our addin to Training\addins directory. Start the AVEVA application and check the console window. During application startup your addin will be loaded and a message string will appear in the console.

5.4 Deploying an Addin on Network Location

The .NET platform prevents .NET addins running if deployed on a network. This will not usually cause an issue for AVEVA products, for which AVEVA recommends a local installation on each machine but might cause problems for customers running their own add-ins.

.NET security can cause issues when running AVEVA products across the network where the add-in assemblies reside on a different machine to the .NET runtime. The default security level is not set to Full Trust, which means that programs may not be able to access resources on the network machine. To overcome this, the intranet security may be set to Full Trust, though this means that any .NET assembly may run. Alternatively, Full Trust may be given to a specified group of strongly named assemblies.

Full Trust is configured using the Code Access Security Policy tool Caspol. First of all the assemblies must be strongly named. Then Caspol is run on each client machine to add all the

assemblies on a given server directory to a group and give Full Trust to this group as follows:

To trust all assemblies in a given folder:

The **Caspol** need to be used from .NET 2.0, using the one from .NET 1.1 does not work.

```
caspol -m -ag LocalIntranet_Zone -url
    \\<ServerName>\<FolderName>\* FullTrust -n "<Name>" -d "<Description>"
```

OR to trust all assemblies with the same strong name:

```
caspol -m -ag LocalIntranet_Zone -strong -file
    \\<ServerName>\<FolderName>\<assemblyName> -noname -noversion FullTrust -n
    "Aveva" -d "Full trust for Aveva products"
```

where <ServerName> is the UNC (Uniform Naming Convention).

The format of a UNC path is: \\<servername>\<sharename>\<directory>

where:

<servername> The Network name,
 <sharename> The name of the share,
 <directory> Any additional directories below the shared directory.

Caspol can be found in c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ or is part of the .NET Framework 2.0 SDK.

5.5 Creating Addin Commands

The purpose of an addin is to provide an application with new functions which may be exposed as commands.

To provide an addin with a new command, create new class and derive it from the **Command** class provided by the **Aveva.ApplicationFramework.Presentation** namespace and defined in the assembly of the same name.

Add a new reference to the **Aveva.ApplicationFramework.Presentation** assembly to your project and create the new class **CreatePanelCmd** inherited from the **Command** class:

```
using Aveva.ApplicationFramework;
using Aveva.ApplicationFramework.Presentation;

namespace Training.MyFirstAddin
{
    public class CreatePanelCmd : Command
    { }
}
```

Each of the addin commands must provide a unique string key. As the key must be unique across all commands registered in the system it is recommend to follow this pattern:

CompanyName.AddinName.CommandName

Use your command constructor to set a **Key** property inherited from **Command** class:

```
public CreatePanelCmd()
{
    base.Key = "Training.MyFirstAddin.CreatePanelCmd";
}
```

The Command class provides the **Execute()** method that should be overridden to perform the desired action. Override the Execute() method and use it to display the message in the application console window.

```
public override void Execute()
{
    Console.WriteLine("Command executed!");
    base.Execute();
}
```

5.6 Registering Addin Commands

The addin command must be registered in the system. The CAF provides the **CommandManager** class that is responsible for managing commands. The class is defined under **Aveva.CommonFramework.-Presentation** namespace. An instance of CommandManager can be retrieved from the **ServiceManager** instance passed as an argument to addin Start(...) method.

The Start(...) method is suitable for registering new commands as it is provided with an instance of **ServiceManager**. To register the new command retrieve the **CommandManager** service:

```
CommandManager sCommandManager =
    (CommandManager) serviceManager.GetService (typeof (CommandManager)) ;
```

create instance of addin command:

```
CreatePanelCmd createPanel = new CreatePanelCmd();
```

and add it to the Commands collection provided by the CommandManager service:

```
sCommandManager.Commands.Add(createPanel);
```

5.7 Enabling and Disabling Addin Commands

The CAF provides functionality to enable and disable commands. The command state can be controlled via the **Command** property **Enabled** that can be overridden by the user command inherited from CAF Command class.

```
public override bool Enabled
{
    get { return true; }
    set { base.Enabled = value; }
}
```

All UI items associated with the command reflect the command state. This means that when command is disabled then all menu items and command bar buttons associated with this command will be disabled.

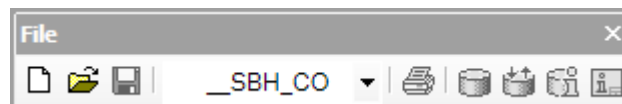


Figure 9: Example of command bar with dimmed controls

Our command is going to create a panel under the current element object (CE). The Command state will depend on the type of CE as panels can be created only under certain types of elements. Override the Enabled property on your CreatePanelCmd. For the moment return a true value in its get method. We will take care of this function later on when accessing dabacon elements are discussed.

6 UIC Files

The user interface (toolbars, commandbars, menus, ...) may be defined in UIC files that are built using the Customisation utility



Refer to chapter 2.2.2 for an overview of the UIC mechanism

Each AVEVA module that supports UIC files has its corresponding **ModuleNameCustomization.xml** file. The file contains a list of all .uic files considered when building toolbars, menu bars and context menus. The order of files is significant as it is possible to layer UIC files on top of each other.

In this chapter we will continue with an exercise started in the previous chapter.

6.1 Creating New UIC File

A uic file is an xml file that contains a set of ui definitions. The name of the file is not important but the file extension must be .uic and must contain at least the xml header defined:

```
<? xml version="1.0" encoding="utf-8"?>
<UserInterfaceCustomization xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="www.aveva.com">
  <Version>1.0</Version>
</UserInterfaceCustomization>
```

Create the .uic file **MyFirstAddin.uic** and add the above header. It can be copied from one of the existing .uic files provided by AVEVA in the distribution directory. Place the file in your Training\addins folder.

6.2 Registering UIC Files

Registering uic files is similar to registering addins but in this case the modules <module>Customization.xml file should be used.

As it is not recommended to change files provided in the AVEVA distribution path you can use the CAF_UIC_PATH environment variable. This is used to inform the CAF where to look for the appropriate <module>Customization.xml file.

Create CAF_UIC_PATH variable in the same way as it was done for CAF_ADDINS_PATH and point it to the same directory; Training\addins\.

Then copy all the customisation files <module>Customization.xml from AVEVA executables directory to your Training\addins directory.

Depending on the module you are using for your addin open the appropriate <module>Customization.xml file and add a new entry at the end of the file.

Example:

```
<CustomizationFile Name="MyFirstAddin"
  Path="c:\Training\addins\MyFirstAddin.uic" />
```



Notice that uic files can be provided with full path name or with just a file name. In the second case the CAF will look for the file in the AVEVA distribution directory.

6.3 Modifying UIC Files

It is strongly recommended to use the interactive user interface customisation tool provided by AVEVA instead of editing the file in a text editor. The interactive tool can be started from any module that supports UI customisation.

To start the tool right click in the toolbars area and choose **Customize** from the context menu.

The module will display the interactive tool for UI customisation:

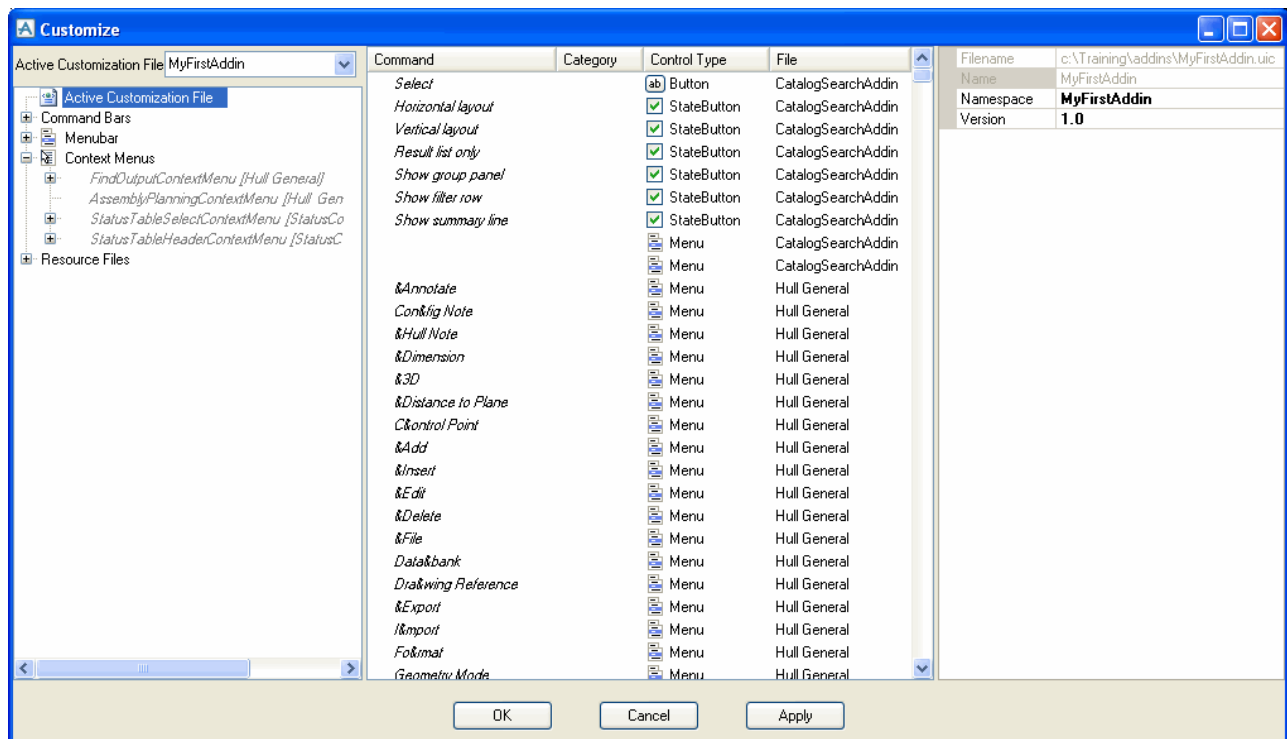


Figure 10

There is a combo box with all loaded uic files in the top left corner. Use this combo box to select the active configuration file. All the new tools, toolbars and menubars will be defined in this file.

The left pane contains a list of all menu bars, toolbars, context menus and resource files that exist in the current module. Each toolbar, menu bar and context menu displays all its tools.

The middle pane lists all available tools. This pane has a context menu defined that can be used to create new tools.

The right pane is a property grid that displays the property of the currently selected item. It is used for modifying properties of a given item.

Tools can be associated with GUI items with a simple drag and drop of a tool into the GUI item.

To connect the command provided by our addin – go through the following steps:

1. Select MyFirstAddin as the active uic file
2. Create a new tool of type Button
3. Set its properties in the property grid:

3.1 Caption to 'Create panel'

3.2 Category to 'Training'

3.3 Command – select CreatePanelCmd from Core commands

3.4 Name to 'Training.MyFirstAddin.CreatePanel'

3.5 Tooltip to 'Create rectangular panel'

Having created the tool you can create the new command bar. Right click on the **Command Bars** node in the left pane and select **New CommandBar** from the context menu.

Select the newly created command bar item and set its properties:

- Caption to 'Training'
- Name to 'Training.MyFirstAddin.Toolbar1'

Now you can drag and drop your tool from the tools pane into your new command bar.

In a similar way create a new menu bar (set Caption to 'Training' and Name to 'Training.MyFirstAddin.Menu1') and again drag and drop your tool from the tools pane into your new menu bar.

Finally click OK.



Refer to the .NET Customization User Guide for a more comprehensive description of the interactive tool described above.

Execute your command by selecting the items defined in the Training command and menu bars.

6.4 Tools with Predefined Values

AVEVA provides a set of tools that display a list of predefined values:

- ComboBox
- List
- FontList
- PopupColorPicker

Each of these kind of tools executes a command whenever the selected item is changed. The FontList and PopupColorPicker list of items is populated by the CAF. For the ComboBox and List the user can define their own set of items. This can be done from the property grid pane.

To provide a command with information about the selected item there are 3 properties on the Command class:

- Value – currently selected value
- List – list of all possible values
- SelectedIndex – index of item in list collection

As an exercise, play a bit with different types of tools and observe the values of the properties mentioned above. Finally, replace our Button tool connected to CreatePanelCmd command with ComboBox and provide it with a predefined set of values describing width, height and thickness of the PANEL that we are going to create later in this training.

Example:

- 500 x 500 x 12
- 500 x 800 x 10
- 700 x 900 x 12

6.5 Project and User UIC File

There are two special uic files that are used to define customisation at a project and user level.

The project customisation file has exactly the same name as the project and should be stored in the AVEVA executables directory.

The user customisation file is automatically created and stored under windows current user directory. Note that even in this case AVEVA still provides a separate customisation file per each module.

7 Extending Addin Functionality

7.1 Using Forms

The CAF exposes an API for creating and managing the application's forms. There are two types of forms supported by the CAF interface **DockedWindow** and **MdiWindow**.

The **Design Explorer** window is an example of a **DockedWindow** and the **3D View** is an example of **MdiWindow**

The CAF **WindowManager** class declared under **Aveva.ApplicationFramework.Presentation** namespace is used for creating and managing forms. The **WindowManager** is a service that can be obtained from the **ServiceManager** instance passed to the addin **Start(...)** method.

The **WindowManager** class provides two methods for creating new forms:

- **CreateDockedWindow(string key, string title, Control control, DockedPosition position)**
- **CreateMdiWindow(string key, string title, Control control)**

Where:

- **key** – String key used as form ID. Must be unique.
- **title** – Window title.
- **control** – Instance of user control that will be displayed inside form client area.
- **position** – Initial docking position

As an exercise provide your addin with another command **FilterPanelsCmd** class. Provide its key and override the **Execute()** method. We are going to use this command to display our own **DockedWindow**.

As the command is needed to register the new window we need to have access to the system **WindowManager** instance passed to the addin **Start(...)** method. As our command is registered in the **Start(...)** method we can use its constructor to pass an instance of **WindowManager** to the command object.

```
public class FilterPanelsCmd : Command
{
    public FilterPanelsCmd(WindowManager wndManager)
    {
        base.Key = "Training.MyFirstAddin.PanelFilterCmd";
    }

    public override void Execute()
    {
        base.Execute();
    }
}
```

Add to your project a new user control called **PanelsCtrl**. Go back to our new command constructor and create a **DockedWindow** instance providing it with the instance of **PanelsCtrl** class. Keep the reference to the created form in a class member for future use.

Set the form property **SaveLayout** to true. This enables the CAF to save the form layout on closing the application.

When the form is ready, show or hide the form in the command **Execute** member depending on its current state. The form's **Visible** property can be used to determine whether to show or hide the form.

Compare your solution with code below:

```
public class FilterPanelsCmd : Command
{
    public FilterPanelsCmd(WindowManager wndManager)
```



```
{
    base.Key = "Training.MyFirstAddin.PanelFilterCmd";

    mForm = wndManager.CreateDockedWindow(
        "Training.MyFirstAddin.PanelsFilterForm",
        "Panels filter",
        new PanelsCtrl(),
        DockedPosition.Right);

    mForm.SaveLayout = true;
}

public override void Execute()
{
    if (mForm.Visible)
        mForm.Hide();
    else
        mForm.Show();

    base.Execute();
}

private DockedWindow mForm;
}
```

Having created the command you now need to register it with the CommandManager. This can be done in your addin Start(...) method:

```
public void Start(ServiceManager serviceManager)
{
    Console.WriteLine("MyFirstAddin loaded");

    CommandManager sCommandManager =
        (CommandManager)serviceManager.GetService(typeof(CommandManager));
    WindowManager sWindowManager =
        (WindowManager)serviceManager.GetService(typeof(WindowManager));

    CreatePanelCmd createPanel = new CreatePanelCmd();
    sCommandManager.Commands.Add(createPanel);

    FilterPanelsCmd filterPanels = new FilterPanelsCmd(sWindowManager);
    sCommandManager.Commands.Add(filterPanels);
}
```

7.2 Managing Command Checked State

The CAF interface provides a StateButton tool type. Such a tool supports two states *Checked* and *Unchecked*.



Figure 11: Example of Checked StateButton tool

The StateButton tool manages its state based on the state provided by the command associated with it. The Command class supports a **Checked** property that can be overridden for customization purposes.

Override the Checked property in your PanelsFilterCmd class and provide it with the appropriate value depending on the form visibility.

```
public override bool Checked
{
    get { return mForm.Visible; }
    set { base.Checked = value; }
}
```

As the form's initial state could depending on the stored layout it is necessary to update the command Checked state after the form layout is loaded. To do this subscribe to the **WindowLayoutLoaded** event provided by the **WindowManager** class.

Add this line to your command constructor:

```
wndManager.WindowLayoutLoaded += new EventHandler(OnWindowLayoutLoaded);
```

and define the new handler:

```
void OnWindowLayoutLoaded(object sender, EventArgs e)
{
    this.Checked = mForm.Visible;
}
```

It is also necessary to update the command Checked state when the form is closed by the standard **Red Cross** button. To handle this update subscribe to the **Closed** event exposed by form class.

Add this line to your command constructor:

```
mForm.Closed += new EventHandler(OnFormClosed);
```

and define the new handler:

```
void OnFormClosed(object sender, EventArgs e)
{
    this.Checked = false;
}
```

Build your project and fix any errors.

Start the AVEVA application and expose the new command to the user interface with the StateButton tool. Observe the state of the button when showing and hiding the form.

7.3 Using PMLNetCallable Controls

As mentioned before, PMLNetCallable objects can be shared between PML and .NET. It is therefore possible to use controls like the AVEVA NetGridControl or PMLFileBrowser from .NET.

In this chapter we are going to provide our addin form with the NetGridControl that will be used for collecting PANEL elements filtered by certain criteria.

To be able to use NetGridControl add a reference to AVEVA **GridControl.dll** assembly provided in the distribution path.

As it is not possible to use NetGridControl directly from the VS design mode we can use standard the .NET Panel control to reserve space for our grid. Open our PanelsCtrl user control in design mode. Add a panel control, name it **mFiltersPanel** and dock it to the top. Add one more panel, name it **mResultsPanel** and set its Dock property to Fill. This is the panel that is going to host our NetGridCtrl.

Drag the Button component into your top docked panel. Name it **CollectBtn** and set its title to Collect. Double click on it to generate the Click event handler and switch to source code mode.

Create a new class member **mGrid** of type NetGridControl. Instantiate it in your PanelsCtrl constructor and add it to mResultsPanel controls collection:

```
public partial class PanelsCtrl : UserControl
{
    public PanelsCtrl()
    {
        InitializeComponent();

        mGrid = new NetGridControl();
        mResultsPanel.Controls.Add(mGrid);
    }

    private void CollectBtn_Click(object sender, EventArgs e)
    {
    }

    private NetGridControl mGrid;
}
```

Run the AVEVA application and display your form. It should look similar to the one below:

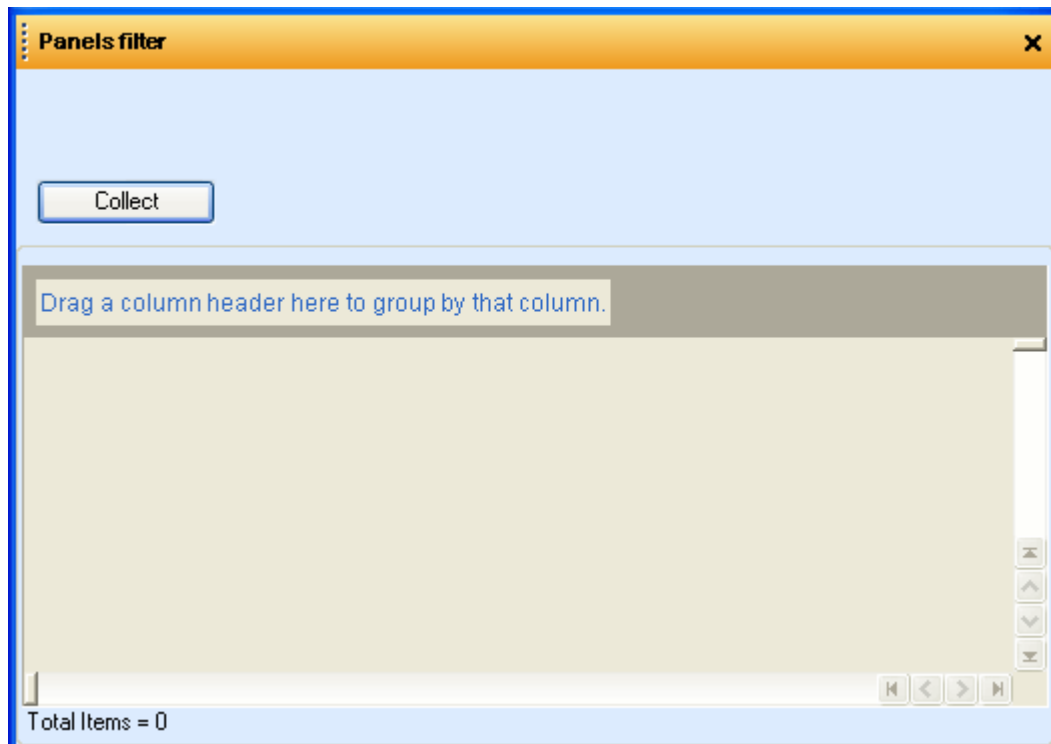


Figure 12

8 Database Interface

The database interface provides one class which represents all types of database elements. The class name is **DbElement** and it is implemented in **Aveva.Pdms.Database** assembly under namespace of the same name.

i *Most of AVEVA assemblies depending on Aveva.Pdms.Utilities assembly. Visual Studio will prompt for adding also this assembly to your project references if not yet added.*

8.1 DbElement

The DbElement class is the most widely used class and it covers a large proportion of the database functionality that will be used in practice.

The methods fall into the following groups:

- Navigation
- Querying of attributes
- Database modifications
- Storage of rules and expressions
- Comparison across sessions

DbElement is a generic object that represents all database elements regardless of their type.

8.1.1 Getting instance of DbElement

To get an instance of any database element use the static method **GetElement(...)** on the DbElement. This method provides two overrides to get an element from its name or reference number and type. The second override is used in the following example where we know the name of the element

Example:

```
DbElement element = DbElement.GetElement("/ACC-STEEL");
```

If the element cannot be found then the method returns a 'null' DbElement. To check if the element exists it can be tested using the **IsNull** property.

An element may exist but may not be valid (for example deleted element). There is an **IsValid** property to test if an element is valid.

Getting the element "/" will return the world element of the current default database

8.1.2 Current Element

To get an instance of the current element (CE) the Element property of the **CurrentElement** class can be used. The CurrentElement class is implemented in **Aveva.Pdms.Shared** assembly under namespace of the same name.

Example:

```
DbElement element = Aveva.Pdms.Shared.CurrentElement.Element
```

The CurrentElement class also provides a CurrentElementChanged event which can be used to handle CE changed events.

8.1.3 Getting Element Type

As all elements are represented by the same class it is necessary to query the element for its type. There is a method **GetElementType()** on the DbElement class that provides the type of element as an instance of **DbElementType**.

Specific DbElementType can be obtained in several ways:

- Use the globally defined instances in **DbElementTypeInstance**. This is the recommended and easiest way to obtain a DbElementType. Example: **DbElementTypeInstance.EQUIPMENT**.
- Use DbElementType.GetElementType(...) method to get type via name. Useful in case of UDETs – must be given with colon.
- Use DbElementType.GetElementType(...) method to get type via hash value. Useful when stored outside PDMS.

Example:

```
if (elem.GetElementType() == DbElementTypeInstance.EQUIPMENT)
```

8.1.4 Navigating

There are basic methods to navigate the primary hierarchy, e.g. consider the following hierarchy:

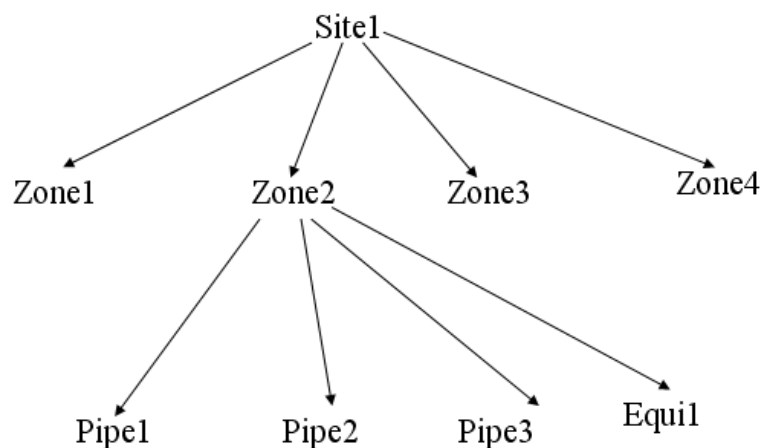


Figure 13

If we are sitting at Zone2, we can navigate as follows:

```
DbElement zone2 = DbElement.GetElement("/Zone2")

DbElement temp;

temp = zone2.Next(); // temp is now Zone3
temp = zone2.Previous; // temp is now Zone1
temp = zone2.Owner; // temp is now Site1
temp = zone2.FirstMember(); // temp is now Pipe1
DbElement pipe1=temp;

temp = zone2.FirstMember(DbElementTypeInstance.EQUIPMENT); // temp is Equi
temp = zone2.LastMember(); // temp is now Equil

temp = pipe1.Next(DbElementTypeInstance.EQUIPMENT); // temp is Equil
temp = pipe1.Previous; // temp is 'null'
```

8.2 Element Attributes

The attributes available on a given DbElement will depend on its type. E.g. a site will have different attributes to a branch. For this reason attributes are accessed through generic methods rather than specific methods. These generic methods pass in the identity of the attribute being queried (a DbAttribute object). There are separate methods for each attribute type (int, double etc), plus separate methods for single values or arrays.

8.2.1 Getting an Attribute

There is a general method **GetAttribute(...)** on the DbAttribute class that returns a **DbAttribute** instance. This class provides information about an attribute: Name, Category, Description, possible values, etc.

To get an attribute value the DbElement class provides a set of methods.

Example:

```
elem.GetString(DbAttributeInstance.NAMN);
elem.GetBool(DbAttributeInstance.LOCK);
elem.GetPosition(DbAttributeInstance.POS);
```

Supported types of attributes:

- int
- double
- bool
- string
- DbElement
- DbElementType
- DbAttribute
- Position
- Direction
- Orientation
- Expresion

If an attribute is not valid for a given object the system will raise an exception.

To avoid exceptions there are a set of methods that return false if an attribute cannot be accessed:

```
elem.GetValidString(DbAttributeInstance.NAMN, ref name);
elem.GetValidBool(DbAttributeInstance.LOCK, ref locked);
```

Also a general method for validating attributes can be used:

```
elem.IsAttributeValid(DbAttributeInstance.TDIR)
```

8.2.2 Attribute Qualifier

Many attributes take a qualifier. The qualifier is the extra information to make the query. Examples of where a qualifier is used are:

- Querying a ppoint position (**PPOS**) requires the ppoint number.
- The **ATTMOD** attribute can be used to query when an attribute was modified but it needs to be given the identity of the attribute.
- A direction/position may be queried with respect to another element.

Example:

```
DbQualifier q = new DbQualifier();
q.wrtQualifier = DbElement.GetElement("/");
```

```
Position pos = CE.GetPosition(DbAttributeInstance.POS, q);
```

The definition of pseudo attributes and their qualifiers is described in the Data Model Reference Manual.

The **DbQualifier** class represents the qualifier. This can hold any type of qualifier, i.e. int, double, string, DbElementType, Attribute, position, direction, orientation. It can hold multiple qualifier values, although few current attributes require multiple valued qualifiers. There is a separate method to set the WRT element.

There is a set of query routines that take a qualifier as an extra argument.

8.2.3 Setting an Attribute

As for getting attributes, there is a family of overloaded methods for setting attributes depending on the attribute type.

Examples:

```
ele.SetAttribute(DbAttributeInstance.DESC, "Example description");
ele.SetAttribute(DbAttributeInstance.HEIGHT, 12.0);
```

There is also a boolean method IsAttributeSettable to test if a given attribute may be set.

8.3 Creating Elements

Creation of new elements is straightforward. It is possible to create elements:

- Below given element
- After given element
- Before given element

When creating element below, the position must be given. If the position is beyond the end of the current members list, it will create it at the end.

Examples:

```
elem.Create(1, DbElementTypeInstance.PLOOP);
elem.CreateAfter(DbElementTypeInstance.PLOOP);
elem.CreateBefore(DbElementTypeInstance.PLOOP);
```

If a required element type cannot be created at the required point an exception will be raised. Method IsCreatable can be used to test if an element of given type can be created at a given location.

Example:

```
if (elem.IsCreatable(DbElementTypeInstance.PLOOP))
    elem.Create(1, DbElementTypeInstance.PLOOP)
```

8.4 Deleting Elements

There is a **Delete()** method provided by the DbElement class that deletes an element. All descendants in the primary hierarchy will be deleted. The **IsHierarchyDeleteable** property can be used to determine if an element and all its descendants can be deleted.

8.5 Moving Elements

An element can be moved to a different location in the primary hierarchy. There are methods to:

- Insert before a given element
- Insert after a given element
- Insert into members list at the last position

Currently an element may only be moved within the same database.

Exercise: Creating Panel Element

Provide your CreatePanelCmd Execute method with code that creates a rectangular PANEL element with all necessary descendants. To make it simple leave panel position and orientation with default values. The size is provided by ComboBox tool connected to this command.

Completed example can be found in Appendix A4

9 Collections and Filters

9.1 Collection

DBElementCollection is an object implemented in the **PDMSFilters** assembly under the **Aveva.PDMS.Database.Filters** namespace. It is specially designed for efficient searching and working with many DbElements.

DBElementCollection constructor will take:

- No parameters
- Root element (type DbElement)
- Root element and Filter (type BaseFilter)

Example:

```
DBElementCollection collect =
    DBElementCollection(DbElement.GetElement("/ACC_STEEL")) ;
```

The resulting collection will contain all the elements under /ACC_STEEL element.

9.2 Filters

Database filter is an object used to filtering elements in a database hierarchy. The filter object is used together with collections.

Example:

```
DBElementCollection collect = DBElementCollection(
    DbElement.GetElement("/"),
    filter) ;
```

This collection will collect all the elements that match the filters criteria.

There is a set of predefined filters in **Aveva.PDMS.Database.Filters**. Some essential ones are presented in this training.

9.2.1 TypeFilter

This is used to collect elements by type. The constructor takes DbElementype or DbElementype[] of types that the collection should contain.

Example matching all pipes:

```
TypeFilter pipeFilter = new TypeFilter(DbElementypeInstance.PIPE) ;
```

9.2.2 AttributeFalse/TrueFilter

AttributeFalseFilter and **AttributeTrueFilter** are used to querying elements by attribute logical value. The constructor takes a DbAttribute object as a parameter. The collection will contain all DbElements where the value of the given attribute is false or true depending of filter type.

Example matching all locked elements:

```
AttributeTrueFilter lockFilter =
    new AttributeTrueFilter(DbAttributeInstance.LOCK) ;
```

9.2.3 AttributeRefFilter

AttributeRefFilter can be used to test if a reference attribute of DbElement matches the given value. The constructor takes a DbAttribute and a DbElement test value. The collection will contain all elements where the reference attribute refers to given DbElement.

Example matching all components that have owner named "/80-B-1-A3B":

```
AttributeRefFilter ownerfilter =
    new AttributeRefFilter(DbAttributeInstance.OWNER,
        DbElement.GetElement("/80-B-1-A3B"));
```

9.2.4 And/OrFilter

These filters are used to build a more complex query by logical constructions of defined filters. Call **Add(BaseFilter)** method to add a new filter.

Example matching all locked pipes:

```
AndFilter lockedPipes = AndFilter();
lockedPipes.Add(pipeFilter);
lockedPipes.Add(lockFilter);
```

9.2.5 BelowFilter

Filter to test if element is below an element for which the given filter is true. Constructor takes BaseFilter parameter.

Example matching all DBElements that are below all locked pipes:

```
BelowFilter belowLockedPipes = new BelowFilter(lockedPipes);
```

9.2.6 CustomFilter - using BaseFilter

Sometimes the user may want to define other search criteria to collect elements from the database. For this AVEVA provides a **BaseFilter** class that is an interface for all database filters. To create a custom database filter define a public class and derive it from the BaseFilter. Override methods: **Clone()**, **Valid(DbElement element)** and **ScanBelow(DbElement element)**. This class should look like:

```
public class CustomFilter : BaseFilter
{
    public override object Clone()
    {
        return new CustomFilter();
    }

    public override bool ScanBelow(DbElement element)
    {
        return false;
    }

    public override bool Valid(DbElement element)
    {
        // check element against criteria and
        // return true or false
    }
}
```

Clone method returns an exact copy of a filter object. ScanBelow() decides if search under this element should be performed (improves search performance). Valid() decides which elements will match the filter criteria.

Exercise: Filtering panel elements

Write the Collect button handler to collect all panels under CE that matches the given criteria:

- Type of element is PANEL (TypeFilter)
- Element is not locked (Lock attribute – AttributeFalseFilter)
- Surface area is higher then value given by user (in square meters). Add text box control to your form for user input. (CustomFilter)

i All of the filters should be grouped together in an AndFilter

i Surface area is not given explicitly so it must be calculated on demand based on volume size divided by panel thickness. The volume can be obtained from panel NVOL attribute and thickness from first PLOOP Height attribute or panel LOHE pseudo attribute. Provide PanelSurfaceFilter class inherited from BaseFilter.

Fill the grid control with collected elements. Use **NetDataSource** for this purpose.

Notice that as NetDataSource is a class that is shared between PML and C# it uses Hashtables as arguments to its constructor. The hashtable keys are just indexes but as PML doesn't support integer types the key must be given as double. The indexes start at 1.

Result window:

Panels filter

All unlocked panels with surface area higher than: m2

Drag a column header here to group by that column.

	NAMN	LOCK	LOHE	FUNCTION	POSNO	PURPOSE
<input checked="" type="checkbox"/>	=24124/11456	False	50mm	unset	0	unset
<input type="checkbox"/>	=24124/10196	False	30mm	unset	0	unset
<input type="checkbox"/>	=24124/10328	False	30mm	unset	0	unset
<input type="checkbox"/>	=24124/10406	False	30mm	unset	0	unset

Total Items = 4

Figure 14

Completed example can be found in appendix A4.

10 Miscellaneous Tools

10.1 Database Expressions

Database expressions are PML1 expressions. E.g. (XLEN * 1000). Expressions are of the following type:

- Double
- DbElement
- Bool
- String
- Position
- Direction
- Orientation

There is a **DbExpression** class to hold an expression. DbExpression provides static method **Parse(...)** to create an instance of DbExpression based on a given expression string:

```
DbExpression dbexp = DbExpression.Parse("( SUBSTR( NAMN OF SITE, 1,3 ))");
```

Once an expression has been created it can be evaluated against a DbElement instance. The DbElement instance provides a set of **Evaluate** methods depending on the expression's result type:

```
elem.EvaluateBool(dbexp);
elem.EvaluateString(dbexp);
elem.EvaluateElement(dbexp);
```

If the expression cannot be evaluated an exception is raised.

10.2 MDB/DB Operations

MDB and DB operations are provided by **DatabaseService**, **MDB**, **Project** and **Db** objects implemented in **Aveva.Pdms.Database** assembly in namespace with the same name. They provide a number of administration API's.

10.2.1 Accessing Current Project and MDB

If you want to access the current opened project or opened MDB you can simply use static properties that return the current instance.

Example – printing to console current MDB name:

```
Console.WriteLine(MDB.CurrentMDB.Name);
```

10.2.2 Opening Project

If you want to start working with databases the project has to be open. To open a project use the **OpenProject(...)** static method from the DatabaseService class.

Example – opening SAM project:

```
Project proj = DatabaseService.OpenProject("SAM", "SYSTEM", "/XXXXXX");
```

OpenProject(...) takes arguments:

1. Project name
2. User name
3. User password (starting with "/")

Returns null if a project could not be opened.

i Note that before exiting the application Project should be closed **proj.Close()**

10.2.3 Opening MDB

After a project is accessed you can open an available MDB. Use **OpenMDB(...)** static method from Project class.

Example – opening /SAMPLE MDB:

```
MDB mdb = Project.OpenMDB(MDBSetup.CreateMDBSetup("/SAMPLE")) ;
```

OpenMDB(...) takes the **MDBSetup** argument that stores options for opening the MDB. MDBSetup can be created passing the name of the MDB as a string starting "/" to static **CreateMDBSetup(...)** method.

i Note that before exiting the application MDB should be closed **mdb.Close()**

10.2.4 Accessing DBs

Often the user wants to work with all project data stored in many databases. To access all DbElements you might want to access all the databases in a given MDB. This can be done using the **GetDBArray()** method.

Example – accessing all databases:

```
Db[] databases = mdb.GetDBArray();
```

To access a particular DB use the **GetDB(...)** method that uses the DB number as parameter:

10.2.5 Simple Transaction Mechanism

To see other user's changes in multiwrite databases or save work the following methods on the MDB class can be used:

1. **GetWork(...)** – gets saved changes since last got work or MDB open
2. **SaveWork(...)** - saves all changes
3. **QuitWork(...)** – drops all changes
4. **Refresh()** – refreshes all changes (leaves user changes but doesn't save them)

Example – saving work:

```
if(mdb.SaveWork("Training changes"))
    MessageBox.Show("Work has been saved successfully!");
```

SaveWork(...) method takes string parameter to indicate action history.

10.3 Invoking PDMS Commands

It may be necessary in some instances to invoke PML commands from .NET using the **Command** class. This is implemented in **Aveva.Pdms.Utilities** and stored under **Aveva.Pdms.Utilities.CommandLine** namespace.

The Command class can be instantiated with its **CreateCommand(...)** static method. Having created an instance it can be executed with its **RunInPdms()** method.

```
if (!Command.CreateCommand("FINISH").RunInPdms())
    return false;
```

10.4 Getting Information about Current Project and Session

To get information about the current session, you need to get a reference to the session object, which is stored in the **SystemDB** in the project properties:

```
Db sysDb = Project.CurrentProject.SystemDB;  
DbSession dbSession = sysDb.CurrentSession;
```

Now you can refer to the object's properties to get the desired information

```
dbSession.Date  
dbSession.Desctiption  
dbSession.SessionNumber  
dbSession.User
```

To retrieve the project information, this is a bit more complicated, when you want to get properties like Name, User or Type from the project, you'll only need to call **CurrentProject**'s properties:

```
Project.CurrentProject.Name;  
Project.CurrentProject.Type;  
Project.CurrentProject.User;
```

The remaining properties like project number, message and description must be retrieved from the Database manually.

First you will have to get a reference to the STATUS element which can be found in the DB.

```
Db sys = Project.CurrentProject.SystemDB;  
DbElement world = sys.World;  
DbElement stat = world.FirstMember(DbElementTypeInstance.STATUS);
```

Now that you have a Status element, you can get it's attributes from the Database.

Example – getting project properties:

```
//Project Description  
stat.GetString(DbAttributeInstance.PRJD);  
//Project Message  
stat.GetAsString(DbAttributeInstance.INFB);  
//Project Number  
stat.GetString(DbAttributeInstance.PRJN);
```


11 PdmsStandalone Interface

This interface, implemented in **Aveva.Pdms.Standalone**, allows access to the database from a new user defined application rather than from an existing AVEVA module.

11.1 Standalone Customization Structure

The application may be a .NET windows or console application. For a better understanding of this chapter a worked example is presented.

Start a Console Application project named **MyFirstStandalone**.

When your project is generated change the default namespace to **Training.MyFirstStandalone**.

Add a reference to Aveva.Pdms.Standalone, Aveva.Pdms.Database and Aveva.Pdms.Utilities and add the appropriate namespaces (for Aveva.Pdms.Utilities use **Aveva.Pdms.Utilities.Messaging**)

Edit static Main() method of the Program class by adding:

```
try
{
    // Initialise Plant/Marine application
    if (!PdmsStandalone.Start())
    {
        PdmsStandalone.ExitError("Failed to initialise Standalone");
        return;
    }

    // Opening project and mdb
    if (!PdmsStandalone.Open("MAR", "SYSTEM", "XXXXXX", "ALL_NO_MDS"))
    {
        PdmsStandalone.ExitError("Failed to open SAMPLE project or Mdb");
        return;
    }

    // Normal work with DB or other AVEVA API
    string name =
        MDB.CurrentMDB.GetFirstWorld(DbType.Design).FirstMember().GetAsString(
            DbAttributeInstance.NAME);

    Console.WriteLine(
        string.Format("First member of design world = {0}", name));
}

catch (PdmsException ex)
{
    PdmsStandalone.ExitError(ex);
}

Console.Read();
// Finalising application
PdmsStandalone.Finish();
```

To initialise the AVEVA application Standalone interface the static method **Start()** was used. This returns false if the initialization failed. To open the project with MDB static **Open(...)** method was used, that returns false when opening failed and takes four string arguments:

1. Project name
2. User name

3. User password
4. MDB

These two methods should return true if the project and environment have been set up correctly.

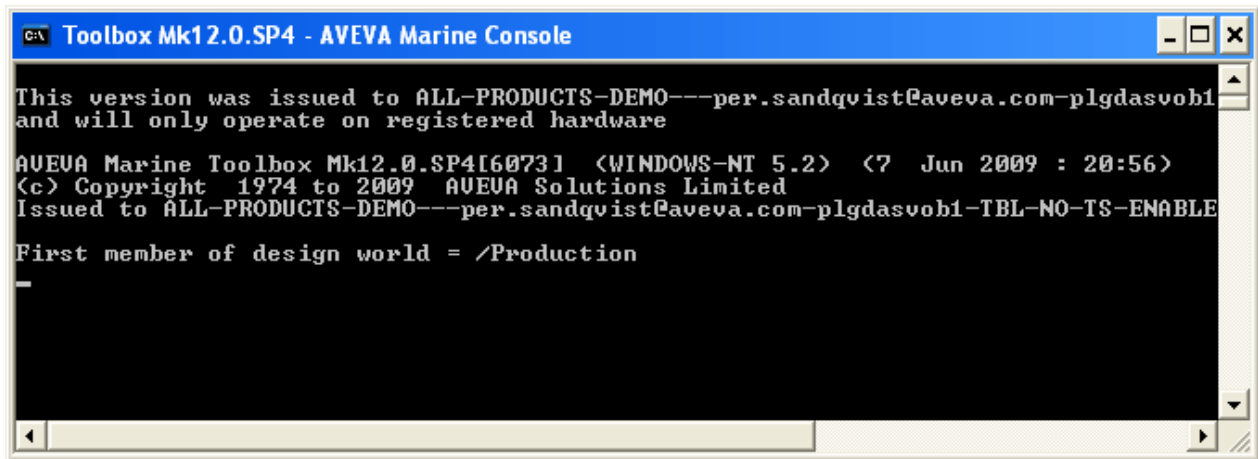


Figure 15: Standalone running

The code in enclosed in try catch statement to catch any PdmsException's on start up. Any exceptions will cause PDMS to exit. The application then waits for user input before calling Finish() to end the PDMS session.

 *Source code for this example is provided in Appendix A5*

11.2 Preparing Runtime Environment

All standalone applications need to have some environment variables like project and PDMSEXE set. This is necessary for initializing the standalone API. Copy %pdmsexex%\marine.bat or pdms.bat file and edit the copied file by replacing the following three lines at the end:

```
echo running: %monexe%\mon %args%
cmd/c "%monexe%\mon" %args%
goto end
```

by

```
echo running: MyFirststandalone
cmd/c "MyFirstStandalone.exe"
goto end
```

After that copy this file to the output location (where MyFirstStandalone.exe is built). Now use this .bat file for running MyFirststandalone application.

12 Hull API

The Vitesse interface is now exposed also as a standard .NET assembly called **MarAPI**. This means that it can be used from the .NET environment in exactly the same way as any other .NET assemblies. The Hull API is located in the **Aveva.Marine.*** namespace family in **marAPI.dll** assembly.

It contains:

- Data extraction tools
- Design tools
- Drafting tools
- Marine geometry objects
- Welding tools



For more details use .NET Interface Reference files (%pdmsexex%\manuals\Docs) and NETmarAPI.chm file (%pdmsexex%\Documentation)

Appendix A – Source code

Appendix A1 – Example of using GridControl and PMLFileBrowser

myForm.pmlfrm

```

import 'GridControl'
handle any
endhandle

import 'PMLFileBrowser'
handle any
endhandle

setup form !!myForm dialog resizable
    using namespace 'Aveva.Pdms.Presentation'

    title 'Panels report'

    button    .CE      'CE' at 0 0 width 10 height 1 callback '!this.OnCE()'
    button    .Export 'Export' at 39 0 anchor R+T width 10 height 1 callback
    '!this.OnExport()'

    container .GridFrame PMLNetCONTROL anchor ALL at xmin.CE ymax.CE+0.5 width
    50 height 10

    member .GridCtrl is NetGridControl
exit

-- constructor method
define method .myForm()
    using namespace 'Aveva.Pdms.Presentation'

    !this.GridCtrl = object NetGridControl()

    !this.GridCtrl.EditableGrid(true)
    !this.GridCtrl.Updates(true)

    !this.GridCtrl.AddEventHandler('AfterSelectChange', !this,
    'OnAfterSelectChange')
    !this.GridCtrl.AddEventHandler('BeforeCellUpdate', !this,
    'OnBeforeCellUpdate')

    !this.GridFrame.Control = !this.GridCtrl.handle()
endmethod

-- CE button callback
define method .OnCE()
    using namespace 'Aveva.Pdms.Presentation'

    -- prepare columns array (list of attributes)
    !columns = array()
    !columns.append('NAME')
    !columns.append('FUNCTION')
    !columns.append('POSNO')

```

```

-- collect elements and get array of their names
!elements = !!CollectAllFor('PANEL', '', !!ce)
!elements = !elements.Evaluate(object BLOCK('!elements[!evalIndex].name'))

-- bind grid to datasource
!source = object NetDataSource('Elements', !columns, !elements)
!this.GridCtrl.BindToDataSource(!source)
endmethod

define method .OnExport()
    using namespace 'Aveva.Pdms.Presentation'

    !fileBrowser = object PMLFileBrowser('SAVE')
    !fileBrowser.Show('C:\\', 'report.xls', 'Save as', false, 'Excel files
(*.xls)|*.xls|All files (*.*)|*.*', 1)
    !fileName = !fileBrowser.File()

    if (!fileName neq '') then
        !this.GridCtrl.SaveGridToExcel(!fileName)
    endif
endmethod

define method .OnAfterSelectChange(!args is array)
    q var !args
endmethod

define method .OnBeforeCellUpdate(!args is array)
    -- validate new value

    -- and perform update
    !this.GridCtrl.DoDabaconCellUpdate(!args)
endmethod

```

Appendix A2 – Example of enhancing PML with PMLNetCallable components

NetMessageBox.cs

```
using System;
using System.Collections.Generic;
using System.Text;

using System.Windows.Forms;
using Aveva.PDMS.PMLNet;
using System.Collections;

namespace Training.NetTools.UI
{
    [PMLNetCallable()]
    public class NetMessageBox
    {
        [PMLNetCallable()]
        public NetMessageBox()
        {
        }

        [PMLNetCallable()]
        public NetMessageBox(string title)
        {
            mDefaultTitle = title;
        }

        [PMLNetCallable()]
        public void Assign(NetMessageBox that)
        {
            DefaultTitle = that.DefaultTitle;
        }

        [PMLNetCallable()]
        public void Show(string message, string title)
        {
            MessageBox.Show(message, title);
        }

        [PMLNetCallable()]
        public void Show(string message)
        {
            MessageBox.Show(message, mDefaultTitle);
        }

        [PMLNetCallable()]
        public string DefaultTitle
        {
            get { return mDefaultTitle; }
            set { mDefaultTitle = value; }
        }

        private string mDefaultTitle = "Error";
    }
}
```

NetCalendar.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;
using Aveva.PDMS.PMLNet;
using System.Collections;

namespace Training.NetTools.UI
{
    [PMLNetCallable()]
    public partial class NetCalendar : UserControl
    {
        public event PMLNetDelegate.PMLNetEventHandler OnDateSelected;

        [PMLNetCallable()]
        public NetCalendar()
        {
            InitializeComponent();
        }

        [PMLNetCallable()]
        public void Assign(NetCalendar that)
        {
        }

        [PMLNetCallable()]
        public string Date
        {
            get { return CalendarCtrl.SelectionStart.ToLongDateString(); }
        }

        private void CalendarCtrl_DateSelected(object sender,
                                                DateRangeEventArgs e)
        {
            if (OnDateSelected != null)
            {
                ArrayList args = new ArrayList();
                args.Add(CalendarCtrl.SelectionStart.ToLongDateString());
                OnDateSelected(args);
            }
        }
    }
}
```

myCalendarForm.pmlfrm

```
import 'NetTools'
handle any
endhandle

setup form !!myCalendarForm dialog resizable
    using namespace 'Training.NetTools.UI'

    title 'Calendar control'

    container .CalendarFrame PMLNetCONTROL at 0 0 width 29 height 6.9
    button    .GetDate 'Get date' at xmin.CalendarFrame ymax.CalendarFrame+0.5
                width 10 callback '!this.OnGetDate()'

```

```

        text          .CurDate ' ' at xmax.GetDate+1 ymax.CalendarFrame+0.5 width 16 is
                                STRING

    member .CalendarCtrl is NetCalendar
exit

define method .myCalendarForm()
    using namespace 'Training.NetTools.UI'

    !this.CalendarCtrl = object NetCalendar()
    !this.CalendarCtrl.AddEventHandler('OnDateSelected', !this,
                                        'OnDateSelected')

    !this.CalendarFrame.Control = !this.CalendarCtrl.handle()
endmethod

define method .OnGetDate()
    using namespace 'Training.NetTools.UI'

    !this.CurDate.val = !this.CalendarCtrl.Date()
endmethod

define method .OnDateSelected(!args is array)
    !this.CurDate.val = !args[0]
endmethod

```

Appendix A3 – Regular expressions support in PML

NetRegex.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Text.RegularExpressions;

using Aveva.PDMS.PMLNet;

namespace Training.NetTools.RegularExpressions
{
    [PMLNetCallable()]
    public class NetRegex
    {
        [PMLNetCallable()]
        public NetRegex()
        {
        }

        [PMLNetCallable()]
        public NetRegex(string pattern)
        {
            mPattern = pattern;
        }

        [PMLNetCallable()]
        public void Assign(NetRegex that)
        {
            mPattern = that.mPattern;
            mResult = that.mResult;
        }

        [PMLNetCallable()]
        public bool IsMatch(string text)
        {
            Regex r = new Regex(mPattern);
            mResult = r.Match(text);
            return mResult.Success;
        }

        [PMLNetCallable()]
        public string GetGroup(string group)
        {
            if (mResult == null)
                throw new PMLNetException(1000, 0, "Perform match first!");

            Group g = mResult.Groups[group];
            if (g.Success)
                return g.Value;
            else
                throw new PMLNetException(1000, 0, "Group not found!");
        }

        private string mPattern = "";
        private Match mResult;
    }
}
```

Appendix A4 – MyFirstAddin source code

MyFirstAddin.cs

```
using System;
using System.Collections.Generic;
using System.Text;

using Aveva.ApplicationFramework;
using Aveva.ApplicationFramework.Presentation;

namespace Training.MyFirstAddin
{
    public class MyFirstAddin : IAddin
    {
        #region IAddin Members

        public string Description
        {
            get { return "My first addin"; }
        }

        public string Name
        {
            get { return "MyFirstAddin"; }
        }

        public void Start(ServiceManager serviceManager)
        {
            Console.WriteLine("MyFirstAddin loaded");

            CommandManager sCommandManager =
                (CommandManager)serviceManager.GetService(typeof(CommandManager));
            WindowManager sWindowManager =
                (WindowManager)serviceManager.GetService(typeof(WindowManager));

            CreatePanelCmd createPanel = new CreatePanelCmd();
            sCommandManager.Commands.Add(createPanel);

            FilterPanelsCmd filterPanels = new FilterPanelsCmd(sWindowManager);
            sCommandManager.Commands.Add(filterPanels);

        }

        public void Stop()
        {
        }

        #endregion
    }
}
```

CreatePanelCmd.cs

```
using System;
```

```

using System.Collections.Generic;
using System.Text;

using Aveva.ApplicationFramework;
using Aveva.ApplicationFramework.Presentation;
using Aveva.Pdms.Database;
using Aveva.Pdms.Shared;
using Aveva.Pdms.Geometry;

namespace Training.MyFirstAddin
{
    public class CreatePanelCmd : Command
    {
        public CreatePanelCmd()
        {
            base.Key = "Training.MyFirstAddin.CreatePanelCmd";
        }

        public override void Execute()
        {
            // Decompose selected string to double values

            String value = base.Value.ToString();

            string[] separator = { "x" };
            string[] values = value.Split(separator,
                StringSplitOptions.RemoveEmptyEntries);
            double[] results = { 0, 0, 0 };
            bool badArgument = values.Length != 3;
            if (!badArgument)
            {
                try
                {
                    for (int idx = 0; idx < values.Length; idx++)
                    {
                        results[idx] = Double.Parse(values[idx].Trim());
                        if (results[idx] <= 0.0)
                            badArgument = true;
                    }
                }
                catch (Exception)
                {
                    badArgument = true;
                }
            }

            if (!badArgument)
            {
                double width = results[0];
                double height = results[1];
                double thickness = results[2];

                // create panel element

                DbElement CE = CurrentElement.Element;
                if (CE.IsCreatable(DbElementTypeInstance.PANEL))
                {
                    DbElement panel = CE.Create(1, DbElementTypeInstance.PANEL);
                }
            }
        }
    }
}

```

```

        if (panel.IsValid)
        {
            DbElement ploop = panel.Create(1,
                                           DbElementTypeInstance.PLOOP);
            if (ploop.IsValid)
            {
                ploop.SetAttribute(DbAttributeInstance.HEIG,
                                  thickness);

                DbElement v1 = ploop.Create(1,
                                           DbElementTypeInstance.PAVERT);
                v1.SetAttribute(DbAttributeInstance.POS,
                               Position.Create(0, 0, 0));
                DbElement v2 =
                    v1.CreateAfter(DbElementTypeInstance.PAVERT);
                v2.SetAttribute(DbAttributeInstance.POS,
                               Position.Create(width, 0, 0));

                DbElement v3 =
                    v2.CreateAfter(DbElementTypeInstance.PAVERT);
                v3.SetAttribute(DbAttributeInstance.POS,
                               Position.Create(width, height, 0));

                DbElement v4 =
                    v3.CreateAfter(DbElementTypeInstance.PAVERT);
                v4.SetAttribute(DbAttributeInstance.POS,
                               Position.Create(0, height, 0));

                CurrentElement.Element = panel;
            }
        }
    }

    base.Execute();
}

public override bool Enabled
{
    get
    {
        return true;
    }
}
}
}

```

PanelSurfacFilter.cs

```

using System;
using System.Collections.Generic;
using System.Text;
using Aveva.PDMS.Database.Filters;
using Aveva.Pdms.Database;

namespace Training.MyFirstAddin
{
    public class PanelSurfaceFilter : BaseFilter
    {
        public PanelSurfaceFilter(double area)
        {

```

```

        mArea = area;
    }

    public override object Clone()
    {
        return new PanelSurfaceFilter(mArea);
    }

    public override bool ScanBelow(DbElement element)
    {
        if (element.GetElementType() == DbElementTypeInstance.PANEL)
            return false;
        else
            return true;
    }

    public override bool Valid(DbElement element)
    {
        // calculate area in square meters
        double thickness = element.GetDouble(DbAttributeInstance.LOHE);
        double volume = element.GetDouble(DbAttributeInstance.NVOL);
        double area = volume / thickness / 1000000;

        return mArea < area;
    }

    private double mArea = 0.0;
}

```

FilterPanelsCmd.cs

```

using System;
using System.Collections.Generic;
using System.Text;

using Aveva.ApplicationFramework.Presentation;

namespace Training.MyFirstAddin
{
    public class FilterPanelsCmd : Command
    {
        public FilterPanelsCmd(WindowManager wndManager)
        {
            base.Key = "Training.MyFirstAddin.PanelFilterCmd";

            wndManager.WindowLayoutLoaded +=
                new EventHandler(OnWindowLayoutLoaded);

            mForm = wndManager.CreateDockedWindow(
                "Training.MyFirstAddin.PanelsFilterForm",
                "Panels filter",
                new PanelsCtrl(),
                DockedPosition.Right);

            mForm.SaveLayout = true;

            mForm.Closed += new EventHandler(OnFormClosed);
        }

        void OnWindowLayoutLoaded(object sender, EventArgs e)
    }
}

```

```

    {
        this.Checked = mForm.Visible;
    }

    bool bSkip = false;

    void OnFormClosed(object sender, EventArgs e)
    {
        // bug in 12.0 SP5: setting up
        // Checked raises command execution
        this.bSkip = true;
        this.Checked = false;
        this.bSkip = false;
    }

    public override void Execute()
    {
        if (bSkip)
            return;

        if (mForm.Visible)
            mForm.Hide();
        else
            mForm.Show();

        base.Execute();
    }

    public override bool Checked
    {
        get
        {
            return mForm.Visible;
        }
        set
        {
            base.Checked = value;
        }
    }

    private DockedWindow mForm;
}

```

PanelsCtrl.cs

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

using Aveva.Pdms.Presentation;
using Aveva.Pdms.Database;
using Aveva.PDMS.Database.Filters;
using System.Collections;

namespace Training.MyFirstAddin

```



```

{
    public partial class PanelsCtrl : UserControl
    {
        public PanelsCtrl()
        {
            InitializeComponent();

            mGrid = new NetGridControl();
            mResultsPanel.Controls.Add(mGrid);
        }

        private void CollectBtn_Click(object sender, EventArgs e)
        {
            double area = 0.0;
            if (!Double.TryParse(AreaTxt.Text, out area))
                return;

            DbElement CE = Aveva.Pdms.Shared.CurrentElement.Element;

            TypeFilter typeF = new TypeFilter(DbElementTypeInstance.PANEL);

            AttributeFalseFilter lockF = new
                AttributeFalseFilter(DbAttributeInstance.LOCK);

            PanelSurfaceFilter surfaceF = new PanelSurfaceFilter(area);

            AndFilter filter = new AndFilter(typeF, lockF);
            filter.Add(surfaceF);

            DBElementCollection col = new DBElementCollection(CE, filter);

            PopulateGrid(col);
        }

        private void PopulateGrid(DBElementCollection col)
        {
            Hashtable headings = new Hashtable();
            Hashtable elements = new Hashtable();

            headings.Add(1.0, "NAMN");
            headings.Add(2.0, "LOCK");
            headings.Add(3.0, "LOHE");
            headings.Add(4.0, "FUNCTION");
            headings.Add(5.0, "POSNO");
            headings.Add(6.0, "PURPOSE");

            double idx = 1.0;
            foreach(DbElement elem in col)
                elements.Add(idx++, elem.GetAsString(DbAttributeInstance.REFNO));

            NetDataSource ds = new NetDataSource("Panels", headings, elements);
            mGrid.BindToDataSource(ds);
        }

        private NetGridControl mGrid;
    }
}

```

Appendix A5 – MyFirstStandalone code

Program.cs

```
using System;

using Aveva.Pdms.Standalone;
using Aveva.Pdms.Database;
using Aveva.Pdms.Utilities.Messaging;

namespace Training.MyFirstStandAlone
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            try
            {
                // Initialise Plant/Marine application
                if (!PdmsStandalone.Start())
                {
                    PdmsStandalone.ExitError("Failed to initialise Standalone");
                    return;
                }

                // Opening project and mdb
                if (!PdmsStandalone.Open("MAR", "SYSTEM", "XXXXXX",
                                         "ALL_NO_MDS"))
                {
                    PdmsStandalone.ExitError(
                        "Failed to open SAMPLE project or Mdb");
                    return;
                }

                // Normal work with DB or other AVEVA API
                string name =
                    MDB.CurrentMDB.GetFirstWorld(DbType.Design).FirstMember().GetAsString(
                        DbAttributeInstance.NAME);

                Console.WriteLine(
                    string.Format("First member of design world = {0}", name));
            }

            catch (PdmsException ex)
            {
                PdmsStandalone.ExitError(ex);
            }

            Console.Read();
            // Finalizing application
            PdmsStandalone.Finish();
        }
    }
}
```