



## **AVEVA™ E3D Design (3.1) PML Form Design**

TG1031-CC1-C01

Customer: Technip Energies  
Date: 19-06-23 - 27-06-23

# **Training Guide**

This page is intentionally left blank.

Customer: Technip Energies  
Date: 19-06-23 - 27-06-23

## Revision Log

DATE	REVISION	DESCRIPTION OF REVISION	AUTHOR	REVIEWED	APPROVED
05/12/2022	0.1	Issued for Review	PW	SF	
03/05/2023	0.2	Reviewed	PW	SF	
04/05/2023	0.3	Issued for Review	PW	SF	
24/05/2023	0.4	Reviewed	PW	SF	
01/06/2023	1.0	Approved for Training AVEVA™ E3D Design 3.1	PW	SF	ST

Template version: V5.1.5

## Updates

Change highlighting will be employed for all revisions. Where new or changed information is presented, section headings will be highlighted in **Yellow**.

## Suggestion / Problems

If you have a suggestion about this manual or the system to which it refers please report it to AVEVA Training & Product Support at [CSE@aveva.com](mailto:CSE@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 <https://www.aveva.com>

## Disclaimer

---

- 1.1 AVEVA does not warrant that the use of the AVEVA software will be uninterrupted, error-free or free from viruses.
- 1.2 AVEVA shall not be liable for: loss of profits; loss of business; depletion of goodwill and/or similar losses; loss of anticipated savings; loss of goods; loss of contract; loss of use; loss or corruption of data or information; any special, indirect, consequential or pure economic loss, costs, damages, charges or expenses which may be suffered by the user, including any loss suffered by the user resulting from the inaccuracy or invalidity of any data created by the AVEVA software, irrespective of whether such losses are suffered directly or indirectly, or arise in contract, tort (including negligence) or otherwise.
- 1.3 AVEVA's total liability in contract, tort (including negligence), or otherwise, arising in connection with the performance of the AVEVA software shall be limited to 100% of the licence fees paid in the year in which the user's claim is brought.
- 1.4 Clauses 1.1 to 1.3 shall apply to the fullest extent permissible at law.
- 1.5 In the event of any conflict between the above clauses and the analogous clauses in the software licence under which the AVEVA software was purchased, the clauses in the software licence shall take precedence.

## Copyright Notice

---

All intellectual property rights, including but not limited to, copyright in this Training Guide and the associated documentation belongs to or is licensed to AVEVA Solutions Limited or its affiliates.

All rights are reserved to AVEVA Solutions Limited and its affiliates companies. The information contained in this Training Guide and associated documentation is commercially sensitive, and shall not be adapted, copied, reproduced, stored in a retrieval system, or transmitted in any form or medium by any means (including photocopying or electronic means) without the prior written permission of AVEVA Solutions Limited. Where such permission is granted, AVEVA Solutions Limited expressly requires that the Disclaimer included in this Training Guide and this Copyright notice is prominently displayed at the beginning of every copy that is made.

Licenses issued by the Copyright Licensing Agency or any other reproduction rights organisation do not apply. If any unauthorised acts are carried out in relation to this copyright work, a civil claim for damages may be made and or criminal prosecution may result.

AVEVA Solutions Limited and its affiliate companies shall not be liable for any breach or infringement of a third party's intellectual property rights arising from the use of this Training Guide and associated documentation.

Incorporates Qt Commercial, © 2011 Nokia Corporation or its subsidiaries.

@AVEVA Solutions Limited 2015

## Trademark Notice

---

AVEVA™, [AVEVA Tags], Tribon and all AVEVA product and service names are trademarks of AVEVA Group plc or its subsidiaries.

Use of these trademarks, product and service names belonging to AVEVA Group plc or its subsidiaries is strictly forbidden, without the prior written permission of AVEVA Group plc or AVEVA Solutions Limited. Any unauthorised use may result in a legal claim being made against you.

All other trademarks belong to their respective owners and cannot be used without the permission of the owner.

Customer: Technip Energies  
Date: 19-06-23 - 27-06-23

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>9</b>
1.1	Target Audience / Course Type .....	9
1.2	Aim .....	9
1.3	Objectives .....	9
1.4	Prerequisites .....	10
1.5	Course Structure .....	10
1.6	Using this Guide .....	10
<b>2</b>	<b>Forms .....</b>	<b>11</b>
2.1	Forms are Global Object .....	11
2.2	Dynamic Loading of Objects, Forms and Functions .....	12
2.3	Defining a Form .....	13
2.3.1	Using the .NET Framework .....	13
2.3.2	Built-in Methods for Forms .....	14
2.4	Callbacks .....	14
2.5	Form Gadgets .....	16
2.5.1	Built-in Members and Methods for Gadgets .....	16
2.5.2	Gadget Positioning .....	17
2.5.3	Docking and Anchoring Gadgets .....	19
2.6	Paragraph Gadgets .....	20
2.7	Button Gadgets .....	20
2.8	Text Entry Gadgets .....	21
2.9	Format Object .....	22
2.10	List Gadgets .....	22
2.11	Frame Gadgets .....	24
2.12	Text pane Gadgets .....	26
2.13	Option Gadgets .....	27
2.14	Toggle Gadgets .....	28
2.15	Radio Gadgets .....	29
2.16	Container Gadgets .....	30
2.17	Tooltips .....	30
<b>Exercise 1</b>	<b>Creation of PML form .....</b>	<b>31</b>
2.18	Progress Bars and Interrupting Methods .....	33
2.19	Open Callbacks .....	34
2.20	Menus .....	35
2.20.1	Defining a Menu Object on a Form .....	35
2.20.2	Defining a Bar Menu on a Form .....	36
2.20.3	Defining a Popup Menu on a Form .....	36
2.20.4	Adding Menus Objects to a Form .....	37
2.21	User-Defined Form Members .....	38
<b>Exercise 2</b>	<b>Extending the form .....</b>	<b>39</b>
<b>3</b>	<b>PML Objects .....</b>	<b>41</b>
3.1	Built in PML Object types .....	41
3.2	Methods Available to All PML Objects .....	42
3.3	The FILE Object .....	42
3.3.1	Using FILE Objects .....	43
3.3.2	Opening a FILE Object in Notepad .....	43
3.3.3	Using the Standard File Browser .....	44
<b>Exercise 3</b>	<b>Use of File object .....</b>	<b>46</b>
<b>4</b>	<b>Collections .....</b>	<b>47</b>
4.1	COLLECT Command Syntax (PML 1 Style) .....	47

4.2	EVALUATE Command Syntax (PML 1 Style).....	47
4.3	COLLECTION Object (PML 2 Style).....	48
4.4	Evaluating the Results from a COLLECTION Object .....	49
<b>Exercise 4 Equipments Collection .....</b>		<b>50</b>
<b>5</b>	<b>View Gadgets .....</b>	<b>53</b>
5.1	Alpha Views.....	53
5.2	Plot View Example.....	54
5.3	Volume View Example.....	54
<b>Exercise 5 Adding a Volume View to a form .....</b>		<b>57</b>
<b>6</b>	<b>Event Driven Graphics (EDG).....</b>	<b>61</b>
6.1	A Simple EDG Event .....	61
6.2	Using EDG.....	62
<b>Exercise 6 Adding EDG to Forms .....</b>		<b>64</b>
<b>7</b>	<b>PML.NET .....</b>	<b>65</b>
7.1	Import an Assembly into AVEVA™ E3D Design .....	65
7.2	Syntax .....	66
7.3	Create a File browser Object .....	66
7.4	Creating a PML form containing the .NET Control.....	67
<b>Exercise 7 Explorer Control form .....</b>		<b>68</b>
7.5	Grid Control.....	69
7.5.1	Applying Data to the Grid .....	69
7.5.2	Events and Callbacks .....	69
<b>Exercise 8 Grid Control Form .....</b>		<b>71</b>
<b>Exercise 9 Developing a PML .NET form .....</b>		<b>73</b>
<b>8</b>	<b>Implementation of Command in Model .....</b>	<b>75</b>
8.1	Defining a Command .....	75
8.2	Loading and Registering a Command.....	76
8.3	Attaching a Command to the UI .....	76
8.4	Killing a command .....	76
<b>Exercise 10 Creation of buttons using command &amp; macro.....</b>		<b>77</b>

This page left intentionally blank.

Customer: Technip Energies  
Date: 19-06-23 - 27-06-23




## 1 Introduction

---

AVEVA™ Programmable Macro Language (PML) is a domain specific language developed by AVEVA to make customizations in AVEVA dabacon based products and projects.

This training guide is about how to create and edit Forms and Menus in the user interface, handling input and output, and actions. The training is building on top of the **TG1031-CC0-C01 AVEVA™ E3D Design (3.1) PML Macros and Functions**.

 *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 Target Audience / Course Type

---

**Target personnel** – All Designer and Engineer

Course Type – Dual

### 1.2 Aim

---

The aim of this training guide is to provide the basic knowledge of the programmable macro language that provide the Macro customization for the **AVEVA™ E3D Design** modules.

The following points need to be understood by the trainees:

- Understand how PML can be used to customise **AVEVA™ E3D Design**.
- Understand the use of Add-ins to customise the environment.

### 1.3 Objectives

---

At the end of this training, the trainees will have a:

- Knowledge of how Forms are created and how Form Gadgets and Form Members are defined.
- Understanding of Menus and Toolbars are defined with PML.
- Understanding of Collections, Basic Event Driven Graphics, Error Tracing and PML Encryption.

## 1.4 Prerequisites

---

- Completed an AVEVA Basic Design Course and have a familiarity with **AVEVA™ E3D Design**.
- Completed the **TG1031-CC0-C01 AVEVA™ E3D Design (3.1) PML Macros and Functions**.
- This course requires the installation of **AVEVA™ Training Setup (EBU) 3.0.8.0** or later.

## 1.5 Course Structure

---

Training will consist of oral and visual presentations, demonstrations, worked examples and set exercises. Any training data or training projects to practice the methods and complete the set exercises outlined in the Training Guide will be provided.

## 1.6 Using this Guide

---

Certain text styles are used to indicate special situations throughout this document, here is a summary:

- Menu pull-downs and button click actions are indicated by bold blue text
- Information that needs to be entered into the software will be in bold red text
- System prompts will be bold italic black text
- Example files or inputs will be in the courier new font.
- Products, Applications, Modules, Toolbars, Explorers and other significant software elements will be in bold black text

Other areas in this Training Guide will be presented with italic blue text and an accompanying icon to classify the type of additional information. Other styles include:

 *Additional Information*

 *Refer to other documentation*

 *Warning*

 *Why*

The following icons will be used to identify industry or discipline specific content:

 *Plant - Content specific to the Plant industry*

 *Marine - Content specific to the Marine industry*

Other icons may be used if necessary.

## 2 Forms

### 2.1 Forms are Global Object

An AVEVA™ E3D Design form is a PML object, stored as a **GLOBAL VARIABLE** in the system. Form gadgets (i.e. button, lists, toggles etc) are PML objects too, but can also be considered as **MEMBERS** of the form object. As PML form is a global variable, once defined, the information held within a form is available to the rest of AVEVA™ E3D Design.

To find out information about a form, it can be queried as if it was an object. Before the form can be queried, it needs to be loaded. For example, show the Graphics Settings form (click the **View > Settings > Graphics Drawlist** button to display the **Graphics Settings: Drawlist** form).

As the form has been shown, it has been loaded into AVEVA™ E3D Design and registered with the Forms and Menus global object (**!!FMSYS**). There is a method on this global object that returns the names of any visible forms. Enter the following into the command window:

```
q var !!fmsys.shownforms()
```

```
<ARRAY>
```

```
[1] <FORM> GPHSETTINGS
```

```
[2] <FORM> GPH3DDESIGN1
```

This shows that the name of the form is **GPHSETTINGS** (or **!!gphsettings** as it is a global variable).



*If the name was already known then form could have been shown by entering **show !!gphsettings**. The **SHOW** syntax loads and displays a form. If you wish to load the form, but not see it, you could enter **pml load form !!gphsettings**.*

By entering **q var !!gphsettings** onto the **Command Window**, a list of form members is returned. As the form gadgets are members of the form, they will all be listed and can be investigated further.


The first gadget listed is called **APPLY** (representing the Apply button on the form). To find out information about it, enter **q var !!gphsettings.apply**.

Specific information about the gadget can be queried directly e.g.

**q var !!gphsettings.apply.tag**

**q var !!gphsettings.apply.val**

**q var !!gphsettings.apply.active**

 *For a form that helps to get this information, enter **show !!pmlforms**. The toggle allows shown forms to be investigated. Without it toggled, all loaded forms are available.*

## 2.2 Dynamic Loading of Objects, Forms and Functions

When a PML object is used for the first time, AVEVA™ E3D Design loads it automatically. This is possible because of the pml.index file found in the PMLLIB search paths. As forms are PML objects, this applies to forms too. For example, if the following was entered into the command window, the two objects will be loaded:

**!gphline = object GPHLINE()**

**show !!GPHMEASURE**

Once an object is loaded by AVEVA™ E3D Design, the definition is stored within the system. This means that if the object definition file is changed or updated whilst it is loaded inside AVEVA™ E3D Design, then the PML object will need to be reloaded. To reload PML object, it needs to be referred to by name. Enter the following:

**pml reload form !!GPHMEASURE**

**pml reload object GPHLINE**

If any instances of the previous object definition are still held within AVEVA™ E3D Design, then these variables will need to be destroyed and redefined to avoid any definition clashes. For example:

**pml reload object GPHLINE**

**!gphline.delete()**

**!gphline = object GPHLINE()**

If a new file is created whilst AVEVA™ E3D Design is open, the file will not be mapped and AVEVA™ E3D Design will not know its location. Even if it is saved in an appropriate file path, it will need to be mapped.

To update the pml.index file and remap files, enter **pml rehash**. This will remap all the files within the first file path in the PMLLIB variable. To remap all files in all of the PMLLIB file paths, enter **pml rehash all**.

 *This command will only work if there is write access to the pml.index file.*

## 2.3 Defining a Form

A form is defined within a **.pmlfrm** file and should be saved under the PMLLIB search path. The file will define the form, its members and any associated methods.

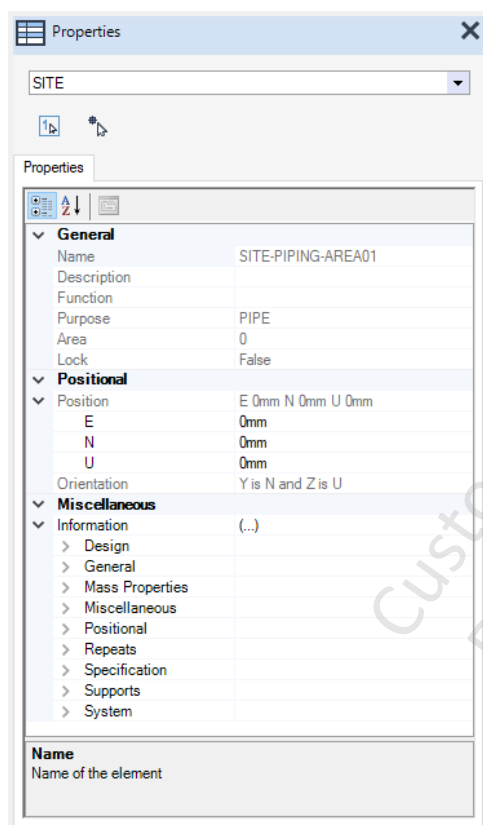
The file will first define the form gadgets and members within the keywords **LAYOUT FORM** and **EXIT**. The rest of the file will contain any form methods. These are defined within the keywords **DEFINE METHOD** and **ENDMETHOD**.

As all the form information can be contained within a single file, there is no longer the need to split a form over multiple files. This used to be a requirement of the PMLUI file structure.

Search for **gphsettings.pmlfrm** in the PMLLIB folder and open the file to investigate.

Enter **q var !! pml.getPathName('gphsettings.pmlfrm')** to find out what search paths are in use.

### 2.3.1 Using the .NET Framework



Form objects make use of the Windows .NET framework. This allows features such as the following to be used:

- Docking forms (example in adjacent screenshot)
- Anchoring gadgets (moving gadgets when resizing)
- Multicolumn lists
- Tabs on Form

Although it is now possible to dock a form into AVEVA™ E3D Design, it is not applicable for every form. Consider the following:

- Forms in heavy use or need to remain open can be docked.
- Any forms with an “OK” or “Cancel” button, should not dock.
- If a form has a menu bar, then these CANNOT be docked.

To declare a form as able to dock, this must be done on the top of the form definition. By including dialog dock, the ability is declared:

For example, the follow would dock the form to the left when it is loaded for the first time:

**setup form !!exampleForm dialog dock left**

To define a floating form that can dock, use the following line:

**setup form !!exampleForm dialog resizable**

To define a form of a certain size in the centre of the screen, use the following line:

**setup form !!exampleForm document at xr 0.5 yr 0.5 size 100 100**

If no additional details are included, the default will create Dialog, non-resizable, fixed size form. A form will always be as big as it needs to be to display the contents.

 *For more examples, refer to syntax graph for the FORM object in the Software Customisation Reference Manual.*

## 2.3.2 Built-in Methods for Forms

Although it is possible to define methods within user-defined forms (discussed later), all FORM objects have built-in methods available. Try on the following on the **Command Window**:

**!!gphsettings.show()**

**q var !!gphsettings.shown()**

**!!gphsettings.hide()**

**q var !!gphsettings.shown()**

In this example, the **.show()** method is used to show the form, the **.shown()** method returns whether the form is shown and the **.hide()** method hides the form from the user

## 2.4 Callbacks

A **FORM** object has several **CALLBACK** members. This means that if the user interacts with a form, a stored action can be performed. Stored as strings, these calls can execute commands, run methods or call PML functions.

These actions occur when the form completes certain actions (e.g. is shown or closed). The following example PML form definition demonstrates the various callbacks on a FORM object.

Display the example form by entering **show !!traExampleCallback** into the command window.



```
-- setup form !!traExampleCallback
layout form !!traExampleCallback
!this.formTitle = Callback Example|
!this.initCall = !this.init()|
!this.firstShownCall = !this.firstShown()|
!this.okCall = !this.okCall()|
!this.cancelCall = !this.cancelCall()|
!this.quitCall = !this.quitCall()|
!this.killingCall = !this.killCall()|
button .ok | OK | OK
button .cancel | Cancel | at x20 CANCEL
exit

define method .traExampleCallback()
    $p Constructor Method
endmethod

define method .init()
    $p Initialise Method
endmethod

define method .firstShown()
    $p First Shown Method
endmethod

define method .okCall()
    $p OK Method
endmethod

define method .cancelCall()
    $p Cancel Method
endmethod

define method .quitCall()
    $p Quit Method
endmethod

define method .killCall()
    $p Kill Method
endmethod
```

Test the form by interacting with it (i.e., press the buttons, close it). What is printed to the **Command Window**? The callbacks on a form object can run any command syntax, global function, local method.

In this example, the callbacks call local methods of similar names (to help recognize the methods). They could have called any methods, even the same methods.

The only limitation is that the callback must be valid, otherwise an error will occur.

**i** Notice how the form can be referred to as *!this* within its definition. The local variable *!this* refers to the owning object and can be used in any object definition.

There are seven main event callbacks built into the FORM object:

- The **CONSTRUCTOR** method was called when the form loaded, recognised as it has the same name as the owning form.
- The **INITCALL** method is called every time the form is shown (perfect for setting default values).
- The **FIRSTSHOWNCALL** method is called the when the form is activated (first shown).
- The **OKCALL** method is called by any button gadget with the OK in its definition.
- The **CANCELCALL** method is called by any button gadget with CANCEL in its definition.
- The **QUITCALL** method is called by clicking the close button on the form.
- The **KILLINGCALL** method is called when the form is unloaded (killed or reloaded).

 *If no callbacks are defined for OKCALL, CANCELCALL and QUITCALL, the default is to hide the form.*

## 2.5 Form Gadgets

There are many kinds of form gadgets, each is an object that will have its own **MEMBERS** and **METHODS**. When defining gadgets on a form, there are two common aims:

- To specify the area to be taken up on the form.
- To define the action to be taken by the gadget if interacted with.

It is the position and size of the gadget that determines the area taken up and its action is defined by its CALLBACK member.

### 2.5.1 Built-in Members and Methods for Gadgets

As gadgets are PML objects, there are a variety of useful members and built-in methods that can be used. Based on the previous example form, enter the following onto the **Command Window**:

To grey-out the **OK** button, enter

```
!!traExampleCallback.ok.active = FALSE
```

To reactive the **OK** button, enter

```
!!traExampleCallback.ok.active = TRUE
```

To hide the **CANCEL** button, enter

```
!!traExampleCallback.cancel.visible = FALSE
```

To show the **CANCEL** button again, enter

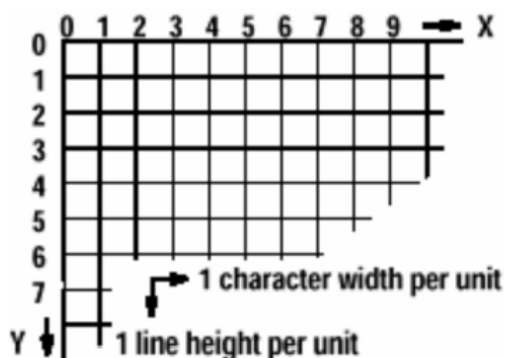
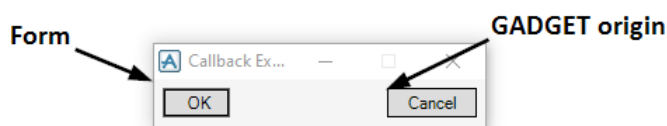
```
!!traExampleCallback.cancel.visible = TRUE
```

To update the tooltip on the **OK** button, enter

```
!!traExampleCallback.ok.setToolTip(|This is an OK button|)
```

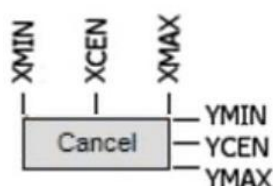


### 2.5.2 Gadget Positioning



- Gadgets are positioned on a form from top left using the AT syntax. The AT syntax defines the origin of the gadget in relation to the owning object (i.e. FORM or FRAME).
- The origin of the form is the top left corner, and positions are set from here. The X positions are across the screen to the right and the Y positions are down the screen.

Gadgets can be positioned explicitly (i.e. x 10) or in relation to other objects (i.e. ymin.ok).



When referring to other objects, there 6 known positions on a gadget: XMIN, XCEN, XMAX, YMIN, YCEN and YMAX. A gadget can be thought of as an enclosing box that will enclose the geometry of the gadget and these points will sit on this enclosing box.

For example, the positions on a button are shown in the adjacent picture.

The following are some examples of the AT syntax. To position a gadget at a known position use: **at x 2 y 1**

To position a CANCEL button using the XMAX and YMIN of OK button, use: **at xmax.ok + 10 ymin.ok**

To position a DISMISS button in the bottom corner of a form use: **at xmax form-size ymax form**

 Notice how the size of the gadget can be used by the keyword `SIZE`

If a gadget can be position using the AT syntax, then **<fgpos>** will appear in its syntax graph. This actually refers to another syntax graph. These common syntax graphs apply to multiple gadgets and are typically found at the beginning of the Software Customisation Reference Manual.

For example, the syntax graph for a button refer to **<fgpos>**

```

>-BUTTON gname +- LINKLabel +- tagtext -----|
|                                     |--<fgpos> -----|
|                                     |-- CALLBACK text -|

```

This means that the syntax found in can be added to the button definition.

```

>-- <fgpos> -- AT +-- val val -----
+- X val -----
+- XMIN -.
+- XCEN -|
+- XMAX +-- <fgprl> -|
\-----\-----+- Y val -----
+- YMIN -.
+- YCEN -|
+- YMAX -\-- <fgprl> -|
\-----\----->

```

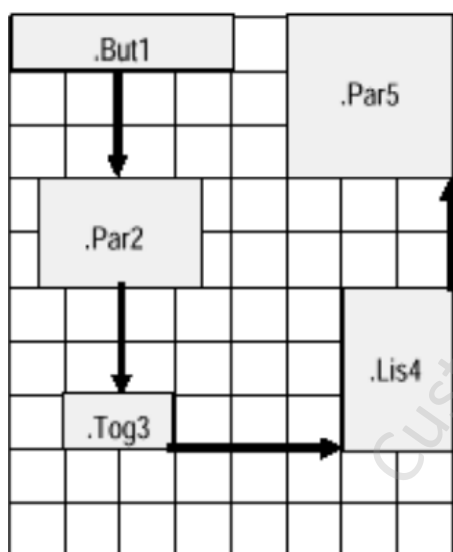
This syntax graph can also refer to another. The syntax graph **<fgprl>** helps define the relative position of the gadget. Try and find it in the reference manual.

**i** For further assistance reading syntax graphics and more examples, refer to the *Software Customisation Reference Manual*.

### 2.5.2.1 Position Gadgets Using the Path Command

The **PATH** command can be used to define logical positions for subsequent gadgets. This method has been superseded by the **AT** syntax but is still valid and has been included for information.

After a gadget has been defined, the next gadget is positioned based on a **PATH**, **HDIST** or **VDIST** and **HALIGN** or **VALIGN**. As an example, see the picture below:



**button .but1** \$\* default placement

PATH DOWN

HALIGN CENTRE

VDIST 2

**paragraph .par2** width 3 height 2 \$\* auto-placed

**toggle .tog3** \$\* auto-placed

PATH RIGHT

HDIST 3

VALIGN BOTTOM

**list .lis4** width 2 height 3 \$\* auto-placed

PATH UP

HALIGN RIGHT

**paragraph .par5** width 3 height 3 \$\* auto-placed

### 2.5.3 Docking and Anchoring Gadgets

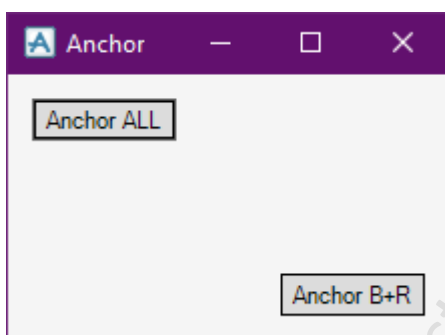
After a gadget has been loaded, its size and position are fixed. The position or size can only be manually changed in the form definition. If a form can be resized, it might be necessary for gadgets to move/resize to reflect the changes to the form.

There are two available syntax definitions that can help: **DOCK** or **ANCHOR**. This syntax is available if **<fgdock>** or **<fganch>** are included in the syntax graph of the gadget. Once a gadget is declared as Anchored or Docking it remains so for the life of the form.

**i** The **DOCK** and **ANCHOR** are mutually exclusive so only one is defined per gadget.

- **ANCHOR** - controls the position of an edge of the gadget relative to the corresponding edge of its container. For example, if a DISMISS button is anchored to the Right and Bottom, it will remain the same distance from the bottom right of the form if it is resized.
- **DOCK** - forces the gadget to fill the available space in a certain direction. For example, if a list is docked to the left, it will maintain its width, but its height will change it fill its container. **DOCK FILL** is very useful for ensuring a gadget is the full size of its container. The attribute is restricted to a small set of gadgets only and is principally aimed at the Frame gadget. View, Alpha, Paragraph, Text pane and Container gadgets support **DOCK FILL** only.

Show the example form by entering **show !!traExampleAnchor**.



```
layout form !!traExampleAnchor dialog resizable
!this.formTitle = |Anchor|
!buttpos = |xmax.f1 - size ymax.f1 - size|
frame .f1 panel anchor ALL width 26 height 5
  button .butt1 |Anchor ALL| anchor L+R+T+B
  button .butt2 |Anchor B+R| at $!buttpos anchor R+B
exit
exit

define method .traExampleDocking()
$p Constructor Method
endmethod

define method .quitCall()
$p Quit Method
endmethod

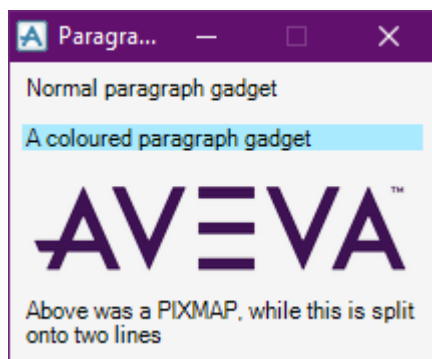
define method .killCall()
$p Kill Method
endmethod
```

Observe how the form behaves when it is resized. Try altering the directions which the gadgets are docked.

## 2.6 Paragraph Gadgets

Paragraph gadgets are simple named gadgets that allow a piece of TEXT or a PICTURE (PIXMAP) to be displayed on a form. It is a passive gadget that cannot be selected by the user so has no callback. Paragraph gadgets are typically used display information or images.

Show the example by typing **show !!traExampleParagraphs**.



```
layout form !!traExampleParagraphs
!this.formTitle = |Paragraphs|
para .para1 text |Normal paragraph gadget| width 25
!t = |A coloured paragraph gadget|
para .para2 at x0 ymax.para1 backg 157 text |$!t|
width 25
para .para3 at x0 ymax.para2 pixmap
/%PMLLIB%\AVEVA_E3D_Graphic.png width 30 height 75
para .para4 at x0 ymax.para3 + 1 text || wid 25 hei 2
exit

define method .traExampleParagraphs()
-- Update the displayed text using CONSTRUCTOR method
!this.para4.val = |Above was a PIXMAP, while this is
split onto two lines|
endmethod
```

Notice how the CONSTRUCTOR method was used to set value of the last gadget. It was only given a size to reserve the space during definition.

 [Refer to the Reference Manual and Guide for more information about paragraph gadgets](#)

## 2.7 Button Gadgets

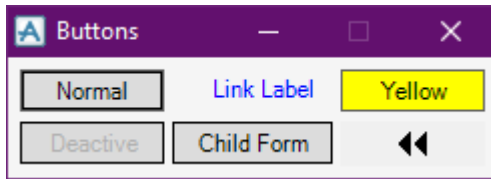
Button gadgets are typically used to invoke an action or to display a child form. Its CALLBACK can call any command, method or function. If a callback and a child form are both specified, the callback command will be run before the child form is displayed.

Show the example by entering **show !!traExampleButtons**.

```
layout form !!traExampleButtons
!this.formTitle = |Buttons|
button .but1 |Normal| width 8
button .but2 linklabel |Link Label| at xmax+2.7 width 6
button .but3 |Yellow| at xmax+1.3 background 4 width 8
button .but4 |Deactive| at xmin.but1 ymax+0.2 width 8
button .but5 |Child Form| form !!traExampleParagraphs width 9
button .but6 toggle call |!this.check()| pixmap width 68 height 10
exit

define method .traExampleButtons()
-- Deactivate but3 in CONSTRUCTOR method
!this.but4.active = FALSE
-- Use built in method to apply pictures
!off = !!pml.getPathName(|leftleftarrow16.png|)
!on = !!pml.getPathName(|rightrightarrow16.png|)
```

```
!this.but6.addPixmap(!off, !on)
endmethod
```



```
define method .check()
  -- Return the toggle button value to the Command
  Window
  !check = !this.but6.val
  $p Value: $!check
endmethod
```

Notice how the !!PML global object was used to get the file path of the picture file. This can be used to find the file path of any file in the pml.index file.

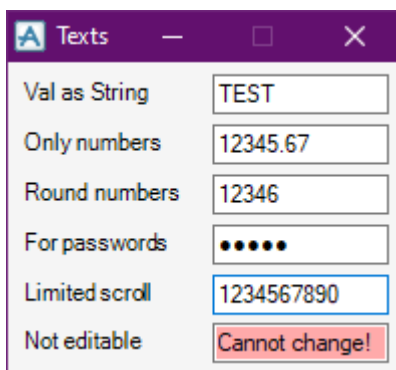
## 2.8 Text Entry Gadgets

A text input gadget provides the user a way of entering a single value into an AVEVA™ E3D Design form. A TEXT gadget needs two aspects covered in its definition:

- **WIDTH** – determines the displayed number of characters. An optional scroll width can also be specified.
- **TYPE** – determines the type of the variable created when inputting a value. This is important when PML uses the value. You may also supply a **FORMAT** object (explained below) to format the value entered (e.g. 2 d.p. number).

Show the example by entering **show !!traExampleTexts**.


```
layout form !!traExampleTexts
!this.formTitle = |Texts|
path down
text .txt1 |Val as String | width 10 is STRING
text .txt2 |Only numbers | width 10 is REAL format !!REALFMT
text .txt3 |Round numbers | width 10 is REAL format !!INTEGERFMT
text .txt4 |For passwords | width 10 NOECHO is STRING
text .txt5 |Limited scroll | width 10 scroll 1 is STRING
text .txt6 |Not editable | width 10 is STRING
exit
```



```
define method .traExampleTexts()
!this.txt6.val = |Cannot change!|
-- Make txt6 uneditable
!this.txt6.setEditable(FALSE)
!this.txt6.background = 28
endmethod
```

**i** Note that at the end of the TAG, a TAB is used instead of a space to align the position of the text field.

Try and fill in the form with various values to investigate the response.

 *Making a text gadget uneditable still allows the user to select its contents. A de-active text box will be full greyed out and unselectable.*

## 2.9 Format Object

A **FORMAT** object manages the information needed to convert a number (always in mm) to a STRING. It can also be used apply a format to a text gadget. Format objects are usually defined as global variables so that they are available across AVEVA™ E3D Design. For example, enter the following onto the **Command Window**:

**!!oneDP = object FORMAT()**

**!!oneDP.dp = 1**

**q var !!oneDp**

There are four standard **FORMAT** objects which are already defined in standard AVEVA™ E3D Design.

**!!distanceFmt** For distance units

**!!boreFmt** For Bore Units

**!!realFmt** To give a consistent level of decimal places on real numbers

**!!integerFmt** To force real numbers to be integers (0 dp Rounded)

To find out more information about these FORMAT object, query them as global variables on the **Command Window** **q var !!boreFmt**

For example, the number of decimal places displayed using **!!realFmt** could be changed by entering **!!realFmt.dp = 6** (the default value is 2).

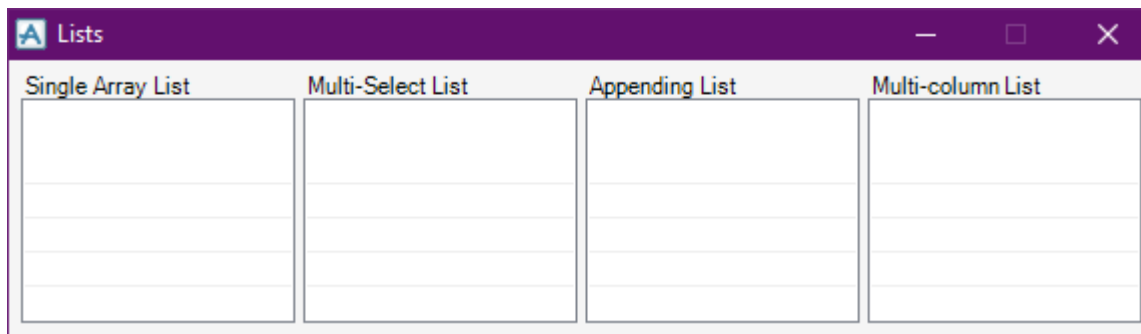
 *These standard format objects are used within the forms of AVEVA™ E3D Design. Changing the definition of these objects will change the way standard product behaves.*

## 2.10 List Gadgets

A **LIST** gadget presents an **ARRAY** of **STRING** values to the user. A single column, or multi-columns of information can be displayed. If set with a multidimensional array of values, the requirement for columns is implied and will be displayed.

Show the example by entering **show !!traExampleLists**. Test the different lists by trying to select row

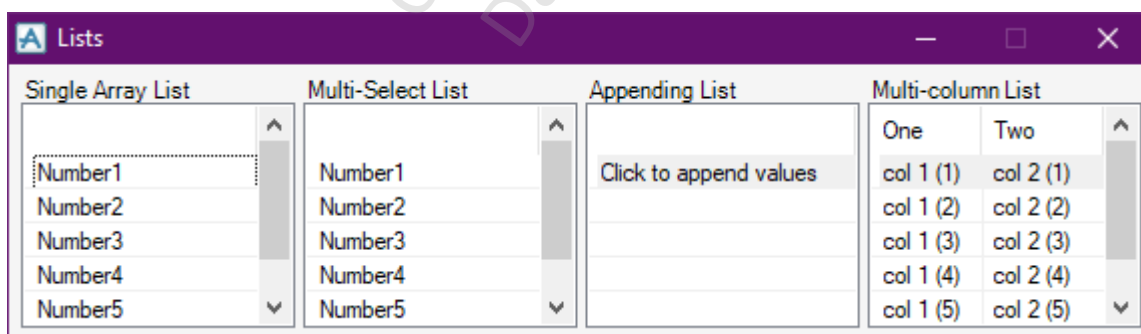
```
layout form !!traExampleLists
  !this.formTitle = |Lists|
  !vshap = |width 15 height 6|
  list .lst1 |Single Array List| call |!this.value()| SINGLE ZEROSEL $!vshap
  list .lst2 |Multi-Select List| MULTI $!vshap
  list .lst3 |Appending List| call |!this.append()| $!vshap
  list .lst4 |Multi-column List| $!vshap
exit
```



```
define method .traExampleLists()
do !n from 1 to 5
  !values[!n] = |Number| & !n
  !rtext[!n] = !n.string()
do !m from 1 to 2
  !multi[!n][!m] = |col | & !m & | (| & !n & |)|
enddo
enddo
!this.lst1.dtext = !values
!this.lst1.rtext = !rtext
!this.lst2.dtext = !values
!single[1] = |Click to append values|
!this.lst3.dtext = !single
!this.lst4.setRows(!multi)
!heading = |One Two|
!this.lst4.setHeadings(!heading.split())
endmethod

define method .value()
!dtext = !this.lst1.selection('Dtext')
!rtext = !this.lst1.selection('Rtext')
$p Selected Dtext = $!<dtext>; Rtext = $!<rtext> (hidden!)
endmethod

define method .append()
!nextLine = !this.lst3.dtext.size() + 1
!val = |Appended | & !nextLine
!this.lst3.add(!val)
endmethod
```



Investigate the lists and how the gadget definition affects how they work. What values are printed the command window when value is selected.

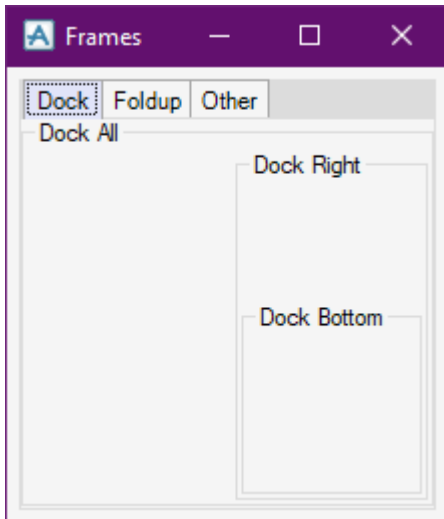
## 2.11 Frame Gadgets

A FRAME is a cosmetic gadget used to surround a group of similar gadgets. This helps with organisation, positioning and user experience. A frame can also be declared as a FOLDUP, PANEL, TABSET or even a TOOLBAR (when defined for the main AVEVA™ E3D Design window).

Show the example by entering **show !!traExampleFrames**.

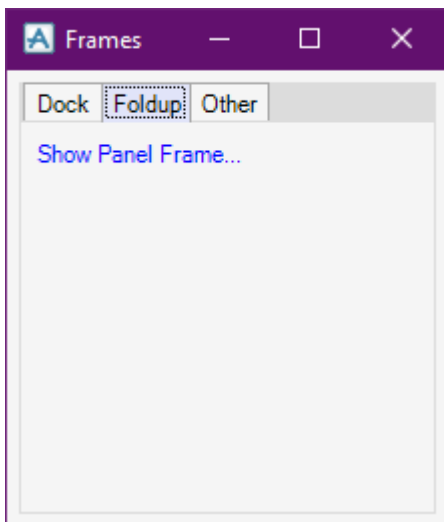
```
layout form !!traExampleFrames dialog resizable
!this.formTitle = |Frames|
frame .tabset TABSET anchor ALL wid 15 hei 5
frame .f1 |Dock| at 0 0 dock FILL
frame .fA |Dock All| dock FILL
frame .fB |Dock Right| dock RIGHT wid 10
frame .fC |Dock Bottom| dock B hei 4
exit
exit
exit
exit
frame .f2 |Foldup| at 0 0 dock FILL
button .but1 linklabel |Show Panel Frame...| at 0 0 call |!this.showD()|
frame .fD panel |Panel Frame| at x0 ymax.but1 dock FILL
frame .fE foldup |Foldup 1| at x1 ymin.fD + 1 anchor t+l+r wid 20 hei 3
button .but2 linklabel |Foldup Panel| anchor t call |!this.fold()|
exit
frame .fF foldup |Foldup 2| at xmin.fE ymax.fE anchor t+l+r wid 20 hei 3
para .para1 text |Inside a frame|
exit
exit
exit
frame .f3 |Other| at 0 0 dock FILL
frame .fG |Deactive| anchor t+l+r wid 21 hei 4
button .but3 |Try and click!| at x4.5 y1 anchor t width 10
exit
!pos = |at xcen.fG - 0.5 * size ymax+0.5|
button .but4 toggle |Show Frame| $!pos backg 8 anchor t call |!this.showF()|
frame .fH |Hidden Frame| at xmin.fG ymax.but4 anchor all width.fG hei.fG
exit
exit
exit
exit
```



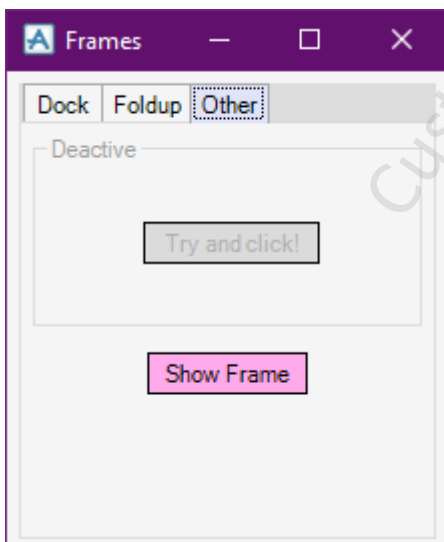


```
define method .traExampleFrames()
    !this.fD.visible = FALSE
    !this.fE.expanded = FALSE
    !this.fG.active = FALSE
    !this.fH.visible = FALSE
endmethod

define method .fold()
    !this.fE.expanded = FALSE
endmethod
```



```
define method .showD()
    if !this.but1.val then
        !this.fD.visible = TRUE
        !this.but1.tag = |Hide Panel Frame...|
    else
        !this.fD.visible = FALSE
        !this.but1.tag = |Show Panel Frame...|
    endif
endmethod
```



```
define method .showF()
    if !this.but4.val then
        !this.fH.visible = TRUE
        !this.but4.tag = |Hide Frame|
    else
        !this.fH.visible = FALSE
        !this.but4.tag = |Show Frame|
    endif
endmethod
```

The example shows the various types of frames and the members/methods available. Notice how the frames immediately below the TABSET are its tabs.

**i** When creating a FRAME gadget, for every FRAME there must be an associated EXIT.

If insufficient EXITS are provided, this may cause an error and the form MAY NOT LOAD. As the error occurred inside the FORM DEFINITION, the **Command Window** will still be in form definition mode and will not function as usual. To exit this mode, enter **EXIT** on the **Command Window** until an ERROR is received. This will mean that form definition mode has been exited and normal commands will work again.

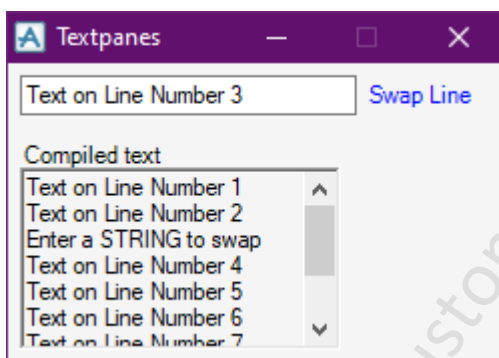
**i** It is good practise to inset code inside a frame block by 2 spaces. This provides an easy way to spot missing exits and makes the code easier to read.

## 2.12 Text pane Gadgets

A **TEXTPANE** gadget provides an area on a form into which a user may edit multiple lines of text and cut/paste from elsewhere on the PML screen. The inputted value is stored as an **ARRAY** of **STRINGS** and can be set and read from the gadget.

Show the example by entering **show !!traExampleTextpanes**.

```
layout form !!traExampleTextpanes
!this.formTitle = |Textpanes|
!this.initCall = |!this.init()|
text .txt1 width 20 is STRING
button .but1 linklabel |Swap Line| at xmax.txt1 + 0.5 y 0 call |!this.swap()| wid 7
textpane .txtp |Compiled text| at x 0 ymax+0.5 wid 20 hei 7
exit
```



```
define method .init()
!this.txt1.val = |Enter a STRING to swap|
do !n from 1 to 10
!val[!n] = |Text on Line Number | & !n
enddo
!this.txtp.val = !val
endmethod

define method .swap()
!lineNO = !this.txtp.curPos()
!line = !this.txtp.line( !lineNo[1] )
!this.txtp.setLine( !lineNo[1], !this.txt1.val )
!this.txt1.val = !line
endmethod
```

Choose different lines inside the textpane and press the button. The method reads the cursor position inside the gadget and it will swap that line. The textpane cannot be given a callback so any interaction has to be through something else.

## 2.13 Option Gadgets

An **OPTION** gadget offers a single choice from a list of items. It may contain either **PIXMAPS** or **STRINGS**, but not a mixture. The gadget displays the **CURRENT** choice in the list. When the user presses the option gadget, the entire set of items is shown as a drop-down list and the user can then select a new item by clicking on the option required. A **COMBO** gadget allows the user to enter in a value as well as choose one from the drop down. If used in conjunction with an open callback (explained in section 2.19), it can select from the list for the user. This would be very useful when there are a long number of options.

The width of a text option gadget must be specified. A tag name is optional and is displayed to the left of the gadget. The available options are stored as an **ARRAY** of **STRINGS** as the gadgets **DTEXT** or **RTEXT** and can be updated by altering this array.

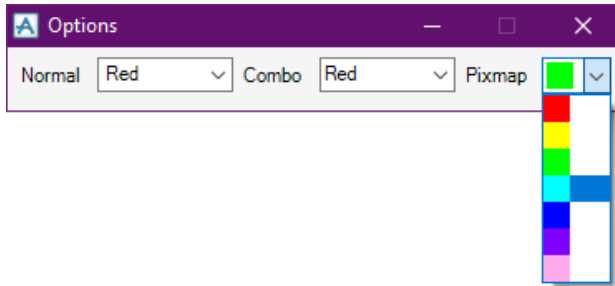
Show the example by entering **show !!traExampleOptions**.

```
layout form !!traExampleOptions
!this.formTitle = |Options|
path right
option .opt1 |Normal| tagwidth 6 width 7
combobox .opt2 |Combo| tagwidth 6 width 7
option .opt3 |Pixmap| tagwidth 6 pixmap width 16 height 16
exit
```

```
define method .traExampleOptions()
!dtext = |Red Yellow Green Cyan Blue Violet Pink|
!this.opt1.dtext = !dtext.split()
!this.opt2.dtext = !dtext.split()

!files[1] = !!pml.getPathName(|red16.png|)
!files[2] = !!pml.getPathName(|yellow16.png|)
!files[3] = !!pml.getPathName(|green16.png|)
!files[4] = !!pml.getPathName(|cyan16.png|)
!files[5] = !!pml.getPathName(|blue16.png|)
!files[6] = !!pml.getPathName(|violet16.png|)
!files[7] = !!pml.getPathName(|pink16.png|)
!this.opt3.dtext = !files
!this.opt3.rtext = !dtext.split()
!this.opt2.callback = | !this.setColour( |
endmethod
```

```
define method .setColour( !gad is gadget, !event is STRING )
if !event.eq('VALIDATE') then
!userInput = !this.opt2.displayText()
!chrs = !userInput.length()
do !n index !this.opt2.dtext
!test = !this.opt2.dtext[!n].upcase().substring(1, !chrs)
if ( !userInput.upcase() EQ !test ) then
!this.opt2.val = !n
break
endif
enddo
endif
endmethod
```



- A **COMBO** gadget allows values to be entered in as well as picked. In this example, an **OPEN CALLBACK** method is used to select the nearest available option in the list. Test the **COMBO** gadget by entering part of the required colour and press enter.

**i** With a **COMBO** box, a user entered value will only be processed if an **OPEN CALLBACK** is included to complete the action.

Enter the **Rtext** of the **pixmap** gadget by using a built-in gadget method:

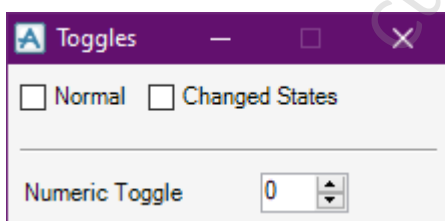
**q var !!traExampleOptions.opt3.selection()**

## 2.14 Toggle Gadgets

**TOGGLE** gadgets are used for independent on/off settings. This means they are used in situations where the user has two choices i.e. is it bold or not? Is it on or off? The state of the toggle is stored under its value member but can also store different descriptions for selected and unselected. **NUMERIC INPUT** gadgets can be used to allow users to enter real values within a range. The user can also use the supplied toggles to alter the value by the defined step.

Show the example by entering **show !!traExampleToggles**.

```
layout form !!traExampleToggles
!this.formTitle = |Toggles|
toggle .tog1 tagwid 5 |Normal| call !!this.state(!this.tog1|
toggle .tog2 tagwid 15 |Changed States| call !!this.state(!this.tog2| states |N| |Y|
line .line at xmin.tog1 ymax.tog1 horiz wid 26 hei 1
numeric .num |Numeric Toggle| at xmin.tog1 ymax.line range 0 10 step 1 NDP 0 wid 3
exit
```



```
define method .state(!gad is GADGET)
if !gad.val then
q var !gad.onvalue
else
q var !gad.offvalue
endif
endmethod
```

Click the toggles and observe the values on the **Command Window**. Notice how a **LINE** gadget has been used to divide the form.

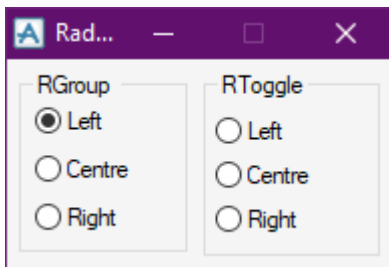
## 2.15 Radio Gadgets

A **RADIO GROUP** allows a user to make a single choice from a fixed number of choices. Two objects can be used to define a radio group: **RGROUP** or **RTOGGLE**. An **RGROUP** element defines the entire object and the tags contained; an **RTOGGLE** is contained within a normal **FRAME** object. **RGROUP** objects can be displayed vertically or horizontally. **RTOGGLE** objects can be arranged within a **FRAME** as required and can be placed alongside other gadgets.

**i** An **RGROUP** object has been deprecated and maybe withdrawn in the future. Use an **RTOGGLE** in preference for new and upgrading code.

Show the example by entering **show !!traExampleRGroups**.

```
setup form !!traExampleRGroups
!this.formTitle = |Radio Groups|
rgroup .vert |RGroup| FRAME vertical callback |!this.rg(!this.vert)|
add tag |Left| select |L|
add tag |Centre| select |C|
add tag |Right| select |R|
exit
frame .rtog |RToggle| at xmax + 1 y 0
path down
rToggle .left |Left| states || |L|
rToggle .cen |Centre| call |q var !this.cen.onvalue| at ymax - 0.1 states || |C|
rToggle .righ |Right| at ymax - 0.1 states || |R|
exit
exit
```



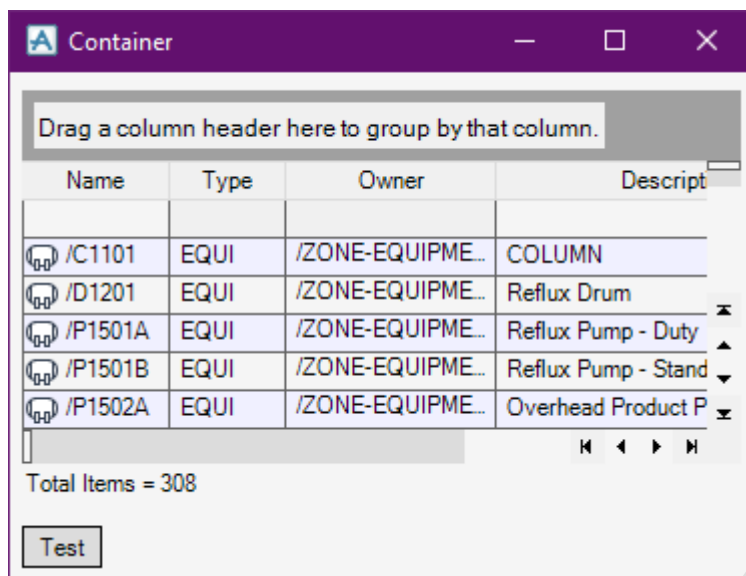
```
define method .traExampleRGroups()
!this.rtog.callback = | !this.rt(!this.rtog) |
endmethod

define method .rg( !gad is GADGET )
q var !gad.selection()
endmethod

define method .rt( !gad is GADGET )
!rtog = !gad.rtoggle(!gad.val)
q var !rtog.onvalue
endmethod
```

## 2.16 Container Gadgets

A CONTAINER gadget is a place holder for a PML .NET control. The PML .NET controls are objects developed in a .NET language, compiled into a .dll file and hosted by the PML form.



AVEVA™ E3D Design is supplied with some example controls that are used within standard product but may be used in any customisation. These include:

- A Grid Control
- A Database Explorer
- A Database Search
- A File browser

The PML .NET controls are explained further later in the guide with examples of their definition and use. Examples of container gadgets will be included then.

 It is possible to develop customised controls using C# Language.

## 2.17 Tooltips

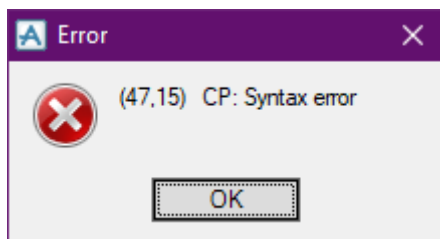
Tooltips are small help boxes which pop up when you move the mouse over an active gadget. Tooltips are typically used to provide more information to the user (for example, on a pixmap button). An example of the syntax is as follows:



`button .but1 |Normal| Tooltip |Run Script|`

## Exercise 1 Creation of PML form

Perform the following task:



- You have been provided with a form called **cc1ex1**. To show the form, enter **show !!cc1ex1**.
- The form has not been maintained and there are a number of bugs in the form. At least 3 are preventing it from being shown and at least 2 are causing the wrong information to be displayed.
- Debug the form so that it can be shown and displays the correct information.
- The form demonstrates poor gadget arrangement, a problem that impacts the user experience. Work through the gadgets and improve their arrangement and sizes to improve the form.
- An example of the aligned form is shown in the below screenshot.

- Write a method that will run when the form is shown. This method should be used to fill default values into the input frame.
- When the input values are set, the method should then run the available methods that fill in the output frame.

- Add a new button to the form that will fill the temperature conversion chart with Fahrenheit to Celsius values.
- Consider how best to position the button.
- Add a method to the new button that will perform the conversion.

- A method has already been written in the form. Find the method and set the button callback.

Temperature Range			Temperature Results	
Minimum	<input type="text" value="0"/>	Fill with °C to °F >>	No.	Celsius = Fahrenheit
Maximum	<input type="text" value="100"/>		1	0 = 32.00
Step Size	<input type="text" value="25"/>		2	25 = 77.00
		Fill with °F to °C >>	3	50 = 122.00

- There is currently no method available to split the temperature string the in lower frames.
- Write a method to do the following:
  - Split the input string based on the delimiter.
  - Read the individual temperatures and convert them (this will depend on °C or °F).
  - Compile the converted values as a space separated string.
  - Write the compiled string back to the form and display how many temperatures.
- This method should be run when the form is shown or the button is pressed.

Temperature Split			Temperature Split Result	
Input	<input type="text" value="10°C/30°C/20°C/5°C"/>	Split temperature >>	No. of Temp	<input type="text" value="4"/>
Delimiter	<input type="text" value="/"/>		Result	<input type="text" value="50°F 86°F 68°F 41°F"/>

- Test the form and check all features work.

Example Form - Exercise 1

Inputs

Temperature conversion (Input)

Temperature  ☒ °C ☐ °F

Temperature Range

Minimum

Maximum  Fill with °C to °F >>

Step Size  Fill with °F to °C >>

Temperature Split

Input

Delimiter  Split temperature >>

Results

Temperature conversion (Output)

Temperature  °F

Temperature Results

No.	Celsius	=	Fahrenheit
1	0	=	32.00
2	25	=	77.00
3	50	=	122.00

Temperature Split Result

No. of Temp

Result

 An example of the completed exercise “EX Creation of PML form” can be found in Training Assistant.

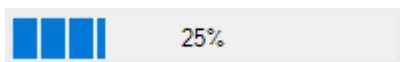


## 2.18 Progress Bars and Interrupting Methods

The standard AVEVA™ E3D Design global object **!!FMSYS** (Forms & Menus System) provides two pieces of functionality which can be applied when designing a form:

- A progress bar (appearing at the bottom right of the main program).
- The ability to interrupt a PML method.

The progress bar is activated by passing a real number to the **.setProgress()** method. For example:  
**!!FMSYS.setProgress(25)**



Passing an argument of zero will cause the progress bar to disappear.

To identify a gadget which can be used to interrupt a method, the gadget must be passed as an argument to the **.setInterrupt()** method. This method can then check the **!!FMSYS** object to see if the gadget has been pressed through its **.interrupt()** method.

Show the example by entering **show !!traExampleProgressBar**.

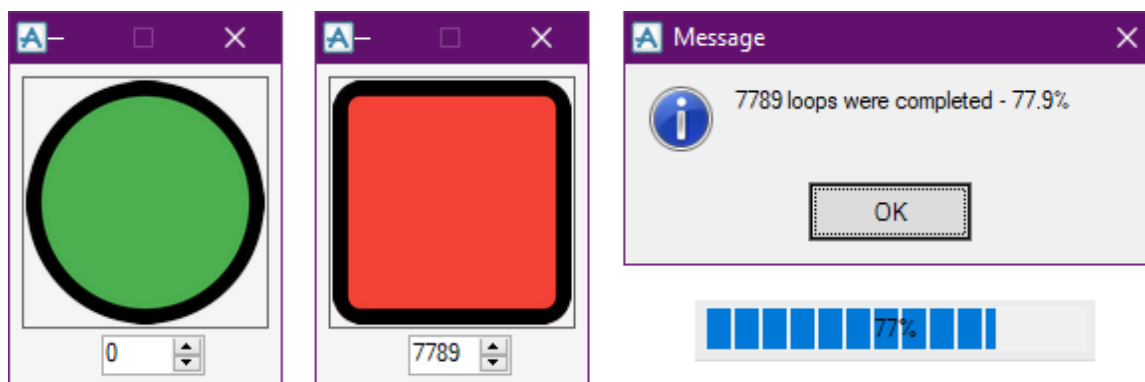
```
layout form !!traExampleProgressBar dialog NoAlign
  button .b1 at xmin ymax call |!this.counter()| pixmap wid 120 hei 120
  numeric .num at x5 ymax + 0.1 range 0 10000 step 1 ndp 0 val 0 wid 4
exit

define method .traExampleProgressBar()
  !this.b1.addPixmap( '/%PMLLIB%\Resources\CC2_START.png' )
endmethod

define method .counter()
  !!FMSYS.setInterrupt( !!traExampleProgressBar.b1 )
  !this.b1.addPixmap( '/%PMLLIB%\Resources\CC2_STOP.png' )
  !this.b1.refresh()
  do !n from 1 to 10000
    !this.num.val = !n
    !this.num.refresh()
    !percent = !n / 100

    -- Check if the method has been interrupted. Break if it has
    if ( !!FMSYS.interrupt() ) then
      !!alert.message(!n & | loops were completed - | & !percent.string(|D1|) & |%|)
      break
    endif
    !!FMSYS.setProgress( !percent )
  enddo
  !this.b1.addPixmap( '/%PMLLIB%\Resources\CC2_START.png' )
  !!FMSYS.setProgress(0)
endmethod
```

Press the start button and press it again to stop the method early.



## 2.19 Open Callbacks

An **OPEN CALLBACK** is a way of providing change information about the gadget the user is interacting with. This means that for every interaction event, the callback will be called. Two pieces of information are supplied during an open callback:

- the gadget being interacted.
- a keyword (as a STRING).

These keywords indicate the event which caused the callback. Different keywords will be generated by different gadgets under different circumstances e.g. a multi-selection gadget may have a 'SELECT' 'UNSELECT', as well as the more standard 'START' 'STOP'.

AVEVA™ E3D Design will recognise an open callback if a method only has one bracket e.g. `call [|this.opencall|]`.

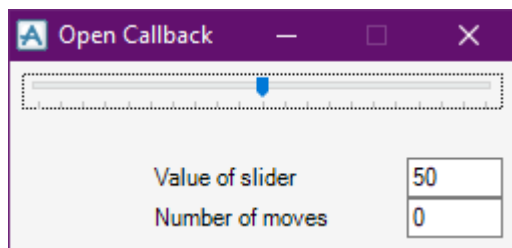
For an open callback to work, there must be an appropriate method defined. An open callback method must be able to receive two arguments (1) a gadget and (2) a keyword.

e.g. `define method .opencall ( !a is GADGET, !b is STRING )`.

As the method is called at every significant event, the method should be written to recognise keywords. The following example has been written around the slider gadget.

Show the example by entering `show !!traExampleOpenCallback`.

```
layout form !!traExampleOpenCallback
!this.formTitle = |Open Callback|
slider .slide anchor L+R horizontal range 0 100 step 5 val 50 width 30
text .text |Value of slider| at xmax - size ymax + 1 anchor B+R width 5 is REAL
text .moves |Number of moves| at xmax.text - size ymax anchor B+L width 5 is REAL
member .store is REAL
exit
define method .traExampleOpenCallback ()
!this.slide.callback = [|this.slideMove(|
!this.moves.val = 0
!this.store = 0
!this.text.val = !this.slide.val
endmethod
```



```
define method .slideMove(!gad is GADGET, !val is
STRING )
!this.text.val = !gad.val
!this.text.refresh()
if !val.eq(|MOVE|) then
!this.store = !this.store + 1
elseif !val.eq(|STOP|) then
!this.moves.val = !this.store
!this.store = 0
endif
endmethod
```

Investigate the form by sliding the slider. Review the methods to identify how it works. Also, revisit the **!!traExampleOptions** to review that open callback.

The **.refresh()** method has been applied to the **.text** gadget to ensure its value updates as the slider is moved. The number of moves is stored as a member of the form. This saves a new global variable from being defined and means the increasing value can be shared between the open callbacks.

**i** An open callback can be given to any gadget that accepts a callback. Different gadgets may generate different event keywords.

## 2.20 Menus

Menu objects are used to provide options to a user and can be applied to a form as either a:

- Menu bar across the top of the form
- Popup menu on a gadget

In both cases, a menu object must be defined to represent the options part of the menu bar or popup menu.

### 2.20.1 Defining a Menu Object on a Form

Within the form definition, the form method **.newMenu()** creates a named menu object. Once the object is defined, the **.add()** method on the menu object can be used to add named menu fields. A menu field can do one of three things:

- Execute a callback
- Display a form
- Display a sub-menu

You can also add a visual separator between fields.

```
!menu = !this.newmenu( |exampleMenu|, |MAIN| )

!menu1.add( |CALLBACK|, |Query|, |q atts| )

!menu1.add( |FORM|, |Progress Bar...|, |exampleProgressBar| )

!menu1.add( |SEPARATOR| )

!menu1.add( |MENU|, |Pull-right1|, |Pull1| )
```

This creates a menu object called exampleMenu with 3 fields (Query, Hello... and Pull-right1) and a separator between the last two fields.

- The Query field when picked will execute the CALLBACK command 'q att' (prints the CE attributes to the command window).
- The Hello... field when picked will load and display the FORM from the previous section !!exampleProgressBar. By convention, the text on a menu field leading to a form ends with three dots, which you must include with the text displayed for the field.
- The SEPARATOR, usually a line, will appear after the previous field.
- The Pull-right1 MENU field when picked will display the sub-menu !this.Pull1 to its right. A menu field leading to a sub-menu ends with a > symbol (this is added automatically).

## 2.20.2 Defining a Bar Menu on a Form

Forms may have a bar menu gadget which appears as a row of options across the top of the form. A bar menu is defined within form definition and specifies the options the user has to choose from. There are three types that can be added:

- Choose – displays a user-defined menu object (reference by its name).
- Window – displays a list of the open form.
- Help – displays the standard help options.

After the bar command, use the bar object method **.add()** to add extra options to the bar menu. As there can only be one bar menu, **!this.bar** refers to the bar menu. For example:

```
bar
!this.bar.add( |Choose|, |exampleMenu| )
!this.bar.add( |Window|, |Window| )
!this.bar.add( |Help|, |Help| )
```

 *Bar menus can only be added to forms which cannot dock (i.e. dialog docking)*

## 2.20.3 Defining a Popup Menu on a Form

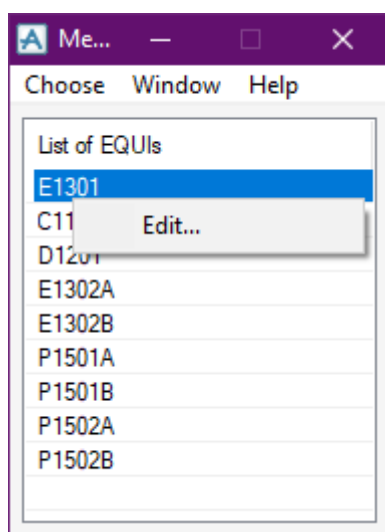
Pop-up Menu (or Context Menu) is a MENU object displayed from a gadget by right-clicking on it. This is a useful method of providing the user with relevant functionality relating to the associated gadget:

**!this.exampleList.setPopup( !this.exampleMenu )**

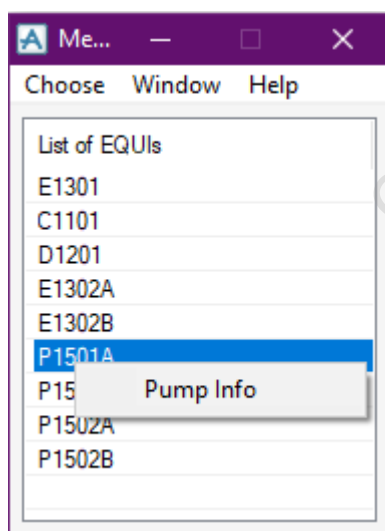
The MENU object is applied to a gadget through the **.setPopup()** method (available on a number of gadgets). The MENU object can be applied at any time, so popup menus can change to ensure the content is always relevant.

## 2.20.4 Adding Menu Objects to a Form

Show the example by entering **show !!traExampleMenus**.



```
layout form !!traExampleMenus
!this.formTitle = |Menus|
!this.initcall = |!this.init()|
bar
!this.bar.add(|Choose|, |exampleBarMenu|)
!this.bar.add(|Window|, |WINDOW|)
!this.bar.add(|Help|, |HELP|)
list .list callback |!this.pickList()| wid 20 hei 12
!menu = !this.newMenu(|exampleBarMenu|)
!menu.add(|CALLBACK|, |Query|, |q_atts|)
!menu.add(|FORM|, |Progress Bar...|, |exampleProgressBar|)
!menu.add(|SEPARATOR|)
!menu.add(|MENU|, |Pull-right1|, |Pull1|)
!menu = !this.newMenu(|exampleEquiMenu|)
!menu.add(|CALLBACK|, |Edit...|, |$p equipment specific|)
!menu = !this.newMenu(|examplePumpMenu|)
!menu.add(|CALLBACK|, |Pump Info|, |$p pipe specific|)
exit
```



```
define method .init()
!titles = |List of EQUIs|
!this.list.setHeadings( !titles.split(|/|) )
-- Collect the equipment from the current element
!ce = !!ce.name
var !equipment coll all EQUI for $!ce
var !equipmentNames eval FLNN for all from !equipment
!this.list.dtext = !equipmentNames
!this.list.rtext = !equipment
endmethod

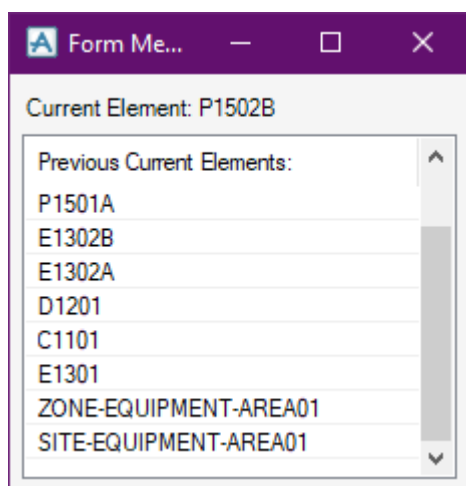
define method .pickList()
-- Set the popup menu based on the first letter of name
if !this.list.selection(|DTEXT|).substring(1,1).eq(|P|)
then
!this.list.setPopup(!this.examplePumpMenu)
else
!this.list.setPopup(!this.exampleEquiMenu)
endif
endmethod
```

Notice how the context menu changes depending on the choice in the list. This allows specific functionality to be available based on the choice of the user.

## 2.21 User-Defined Form Members

As a **FORM** is an **OBJECT**, it may be given user-defined **MEMBERS**. This is a useful way to store data, effectively creating global variables. The benefit of this method is that data is global without having large numbers of individual variables. Form members replaces the previous method of USERDATA (PML 1 style data storage) and are given an object-type. These variables have the same lifetime as the form and are deleted when the form itself is unloaded.

Show the example by entering **show !!traExampleFormMembers**.



```
layout form !!traExampleFormMembers resize
!this.formTitle = |Form Members|
!this.initcall = |!this.init()|
track |DESICE| call |!this.update()|
para .ceName anchor t+l+r wid 20 hei 1
list .stored || at x 0 ymax anchor all wid 25 hei 10
member .ceRef is DBREF
member .storage is ARRAY
exit

define method .traExampleFormMembers()
!this.stored.callback = |!this.return()|
!headings = |Previous Current Elements:|
!this.stored.setHeadings( !headings.split(|/|) )
endmethod

define method .init()
!this.ceRef = !!ce
!this.ceName.val = |Current Element: | &
!!ce.attribute(|FLNN|)
endmethod

define method .update()
!this.storage.insert(1, !this.ceRef)
!block = object BLOCK(
|!this.storage[!evalIndex].attribute('FLNN')| )
!dtext = !this.storage.evaluate(!block)
!this.stored.dtext = !dtext
!this.init()
endmethod

define method .return()
!!ce = !this.storage[!this.stored.val]
endmethod
```

It shows how a form can be assigned an element when shown and how this element can be updated. Every time the update button is pressed, the element is stored in the member storage.

Investigate the information stored on the form members by entering:

**q var !!traExampleFormMembers.ceRef**

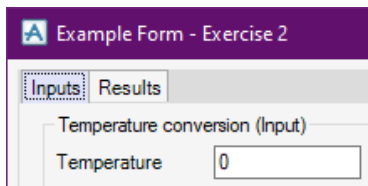
**q var !!traExampleFormMembers.storage**

**i** Notice how the form uses the **TRACK** syntax to follow the current element. This syntax works for other modules too. Just replace the **DESI** from **DESICE** with the module database type e.g. **PADDCE**, **CATACE** etc.

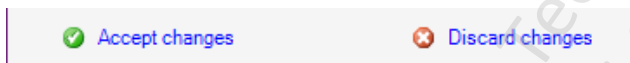
## Exercise 2 Extending the form

Perform the following task:

- Save the form from the previous exercise as **cc1ex2.pmlfrm**.
- Update form definition so the name, title and constructor method are correctly named.
- Enter **PML REHASH ALL** so AVEVA™ E3D Design finds the new file.
- Upgrade the “Inputs” and “Results” frames into a single tabset.



- Consider what code is needed to define a tabset and how the existing frames will need to be updated.
- Write a method that will allow data to be applied to and taken from this member.
- Consider how the code will get the values from the gadgets and how best to store them. What type of values will need to be stored?
- Add two buttons to the base of the form. One to update the stored values (Accept changes) and the other will apply the stored values to the gadgets (Discard changes)



- The buttons will be link labels and should call the method to store/reset data. To the left of the buttons there should be 16x16 pixmap paragraph.
- Consider how the gadgets will be positioned and how they will move if the form is resized.
- Using the `.addPixmap()` method, apply standard AVEVA™ E3D Design images to them.
  - `.addPixmap(!pml.getPathName('accept.png'))`
  - `.addPixmap(!pml.getPathName('discard.png'))`
- As the buttons have been removed from the form, only one type of conversion can be applied to the List gadget (depending on which method the open callback calls)
- Add a **POPUP MENU** to the list gadget which will allow the other type of conversion to be completed.

Temperature Results				
No.	Celsius	=	Fahrenheit	
1	0	=	32.00	
2	25	=	77.00	
3	50	=	122.00	

Temperature Results				
No.	Fahrenheit	=	Celsius	
1	0	=	-17.7	
2	25	=	-3.89	
3	50	=	10.00	

- There will need to be two new menu objects created on the form.
  - to run the method as Celsius.
  - to run the method as Fahrenheit.
- After a choice has been made, the popup menu needs to be swapped so the user always sees the correct one.
- Consider what methods will need to be called and how the popup menus can be managed.
- Test the form and check all features work.

The image shows two screenshots of a software form titled 'Example Form - Exercise 2'. The left screenshot shows the 'Inputs' tab, and the right screenshot shows the 'Results' tab.

**Inputs Tab:**

- Temperature conversion (Input):** A text box for 'Temperature' with the value '0'. Below it are radio buttons for '°C' (selected) and '°F'.
- Temperature Range:** Three text boxes for 'Minimum' (0), 'Maximum' (100), and 'Step Size' (25). To the right of the 'Maximum' and 'Step Size' boxes are links: 'Fill with °C to °F >>' and 'Fill with °F to °C >>'.
- Temperature Split:** A text box for 'Input' with the value '10°C/30°C/20°C/5°C'. Below it is a text box for 'Delimiter' with the value '/'. To the right is a link: 'Split temperature >>'.
- At the bottom are two buttons: 'Accept changes' (with a green checkmark icon) and 'Discard changes' (with a red X icon).

**Results Tab:**

- Temperature conversion (Output):** A text box for 'Temperature' with the value '32'. To its right is '°F'.
- Temperature Results:** A table with 4 columns: 'No.', 'Celsius', '=', and 'Fahrenheit'. It contains 3 rows of data.
 

No.	Celsius	=	Fahrenheit
1	0	=	32.00
2	25	=	77.00
3	50	=	122.00
- Temperature Split Result:** A text box for 'No. of Temp' with the value '4'. Below it is a text box for 'Result' with the value '50°F 86°F 68°F 41°F'.
- At the bottom are two buttons: 'Accept changes' (with a green checkmark icon) and 'Discard changes' (with a red X icon).

An example of the completed exercise “EX Extending the form” can be found in Training Assistant.



## 3 PML Objects

### 3.1 Built in PML Object types

PML objects allow values and methods to be bound together such that they can be considered as a single entity. The use of PML objects means that code can be standardised and reduced. There are a large number of standard objects which are supplied with AVEVA™ E3D Design.

These include STRING, REAL, BOOLEAN, ARRAY, BORE, DIRECTION, DBREF, FORMAT, MDB, ORIENTATION, POSITION, FILE, PROJECT, SESSION, TEAM, USER, ALERT, FORM, all form Gadgets and various graphical aid objects. Each object has a set of built in methods for setting or formatting the object contents. The following are the objects covered by the Software Customisation Reference Manual:

ARRAY	DATEFORMAT	FORMAT	REAL
BANNER	DATETIME	LOCATION	REPORT
BLOCK	DB	MACRO	SESSION
BOOLEAN	DBREF	MDB	STRING
BORE	DBSESS	OBJECT	TABLE
COLLECTION	DIRECTION	ORIENTATION	TEAM
COLUMN	EXPRESSION	POSITION	UNDOABLE
COLUMNFORMAT	FILE	PROJECT	USER

The following objects from the reference manual cover forms and menus:

ALERT	FORM	OPTION	TEXTPLANE
BAR	FRAME	PARAGRAPH	TOGGLE
BUTTON	LINE	RTOGGLE	VIEW ALPHA
COMBOBOX	LIST	SELECTOR	VIEW AREA
CONTAINER	MENU	SLIDER	VIEW PLOT
FMSYS	NUMERIC	TEXT	VIEW VOLUME

The following objects from the reference manual cover 3D geometry:

ARC	LOCATION	POINTVECTOR	PROFILE
LINE	PLANE	POSTEVENTS	RADIAL GRID
LINEARGRID	PLANTGRID	POSTUNDO	RADIAL GRID

## 3.2 Methods Available to All PML Objects

In addition to the object specific methods, there are methods that are common to all objects. The following table lists the methods available to all objects:

NAME	RESULT	PURPOSE
<b>.attribute(STRING)</b>	ANY	To set or get a member of an object, providing the member's name as a STRING.
<b>.attributes()</b>	ARRAY OF STRINGS	To get a list of the names of the members of an object as an array of STRING.
<b>.delete()</b>	NO RESULT	Destroy the object - make it undefined
<b>.eq(any)</b>	BOOLEAN	Type-dependent comparison
<b>.lt(any)</b>	BOOLEAN	Type-dependent comparison (converting first to STRING if all else fails)
<b>.max(any)</b>	ANY	Return the maximum of object and second object
<b>.min(any)</b>	ANY	Return the minimum of object and second object
<b>.neq(any)</b>	BOOLEAN	TRUE if objects do not have the same value(s)
<b>.objectType()</b>	STRING	Return the type of the object as a string
<b>.set()</b>	BOOLEAN	TRUE if the object has been given a value(s)
<b>.string()</b>	STRING	Convert the object to a STRING
<b>.unset()</b>	BOOLEAN	TRUE if the object does not have a value

## 3.3 The FILE Object

The FILE object is an example of how older functionality has been replaced with a PML 2 object. This object replaces the 'openfile' 'readfile' 'writefile' 'closefile' syntax and provides increased functionality (file path, if the file is open etc). It is now possible to read or write to a file in a single operation.

To create a file object:

```
!input = object file('C:\FileName')
```

```
!output = object file('C:\FileName.out')
```

To open a file so it's available to be written to

```
!output.open('WRITE')
```

These arguments are applicable for the **.open()** method: **READ, WRITE, OVERWRITE, APPEND**.

To close a file once finished

```
!output.close()
```

To check if the file is already open

```
q var !output.isOpen()
```

To find out when the file was last written to

```
q var !output.dtm()
```

### 3.3.1 Using FILE Objects

To read a line from file:

```
!line = !input.readRecord()
```

 (File must be open)

To write a line to file:

```
!output.writeRecord(!line)
```

 (File must be open)

To read all the input file:

```
!fileArray = !input.readFile()
```

 (Files are opened and Closed automatically)

To write all of the data to file

```
!output.writeFile('WRITE',!fileArray)
```

 (Files are opened and Closed automatically)

 The `.readFile()` method has a default maximum file size which it can read. This can be increased by passing the method a REAL argument (representing the number of lines in the file).

### 3.3.2 Opening a FILE Object in Notepad

The **SYSCOM** command can be used to open the file in an external program, such as Notepad. The syntax inside the string marks will be run as a windows command.

```
!file = object file(|C:\temp\pmlib\forms\c2ex1.pmlfrm|)
```

```
syscom|C:\Windows\system32\notepad.exe $!file &|
```

By using the command START, the file will open in the Windows default program

```
syscom |start $!file &|
```

 The “&” to the end of the line will open the file as a new process. Without it, the AVEVA™ E3D Design session will suspend until the external program is closed.

### 3.3.3 Using the Standard File Browser

AVEVA™ E3D Design is supplied with a standard file browser that can be used to load forms. The best way to load the form is to use the **!!fileBrowser** function it will initialise the browser (based on the arguments).

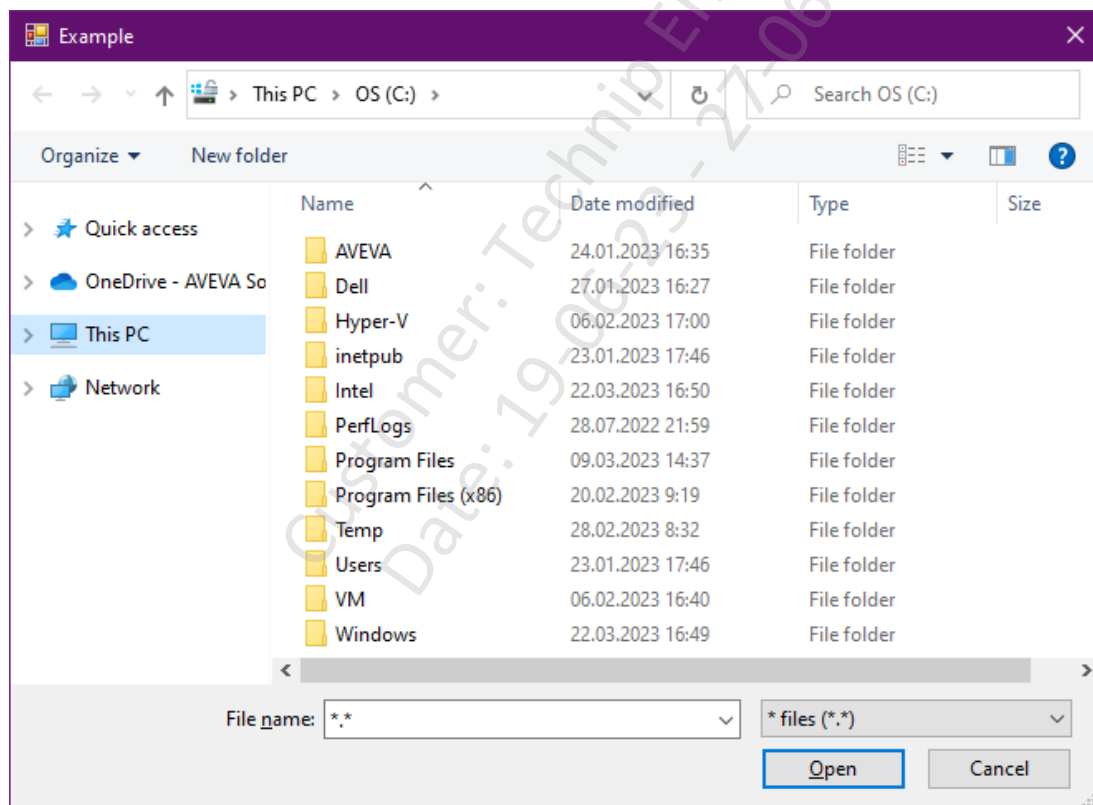
To show the browser form, enter the following:

```
!!fileBrowser('c:\','*.*','Example',true,'q var !!fileBrowser.file')
```

Where the arguments are:

- Initial file path for the browser form.
- File type filter.
- Title for the browser form.
- Does the file need to exist? Boolean - used to differentiate between Open and Save.
- The Callback on the action button of the browser form.

The standard windows file browser should be displayed, defaulting to the C:\



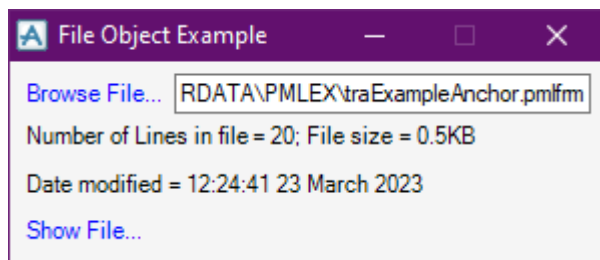
Navigate to a file and click **Open**, a file object should be printed to the command window:

```
<FILE> C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\cc0ex6.mac.
```

Once a file is chosen, it is held as the **.file** member of the **fileBrowser** form and is available for use.

Show the example by entering **show !!traExampleFile**. The example demonstrates how the object can be used to read a file, gain information about it and even show it.

```
layout form !!traExampleFile
!this.formTitle = |File Object Example|
button .browse linklabel |Browse File...| wid 8
text .fileName || call !!this.read(object file(!this.txt1.val), 1)| width 25 is string
para .par1 at x 0 ymax text || width 35
para .par2 at x 0 ymax text || width 35
button .load linklabel |Show File...| at x0 ymax call !!this.load() wid 8
member .file is FILE
exit
```



```
define method .traExampleFile()
!this.browse.callback =
!!!fileBrowser('C:\temp\pmlib\forms',
'*.pmlfrm', $
'Load File', TRUE, '!!traExampleFile.read(
!!fileBrowser.file, 2)')|
!this.load.visible = FALSE
endmethod
```

```
define method .read( !file is FILE, !flag is
REAL )
!this.file = !file
if !flag.eq(2) then
!this.fileName.val = !file.string()
endif
!file.open('READ')
!n = 0
do
!line = !file.readRecord()
break if !line.unset()
!n = !n + 1
enddo
!file.close()
!this.load.visible = TRUE
!date = !file.dtm()
!size = ( !file.size() / 1000 )
!fileSize = |File size = | &
!size.string(|d1|) & |KB|
!this.par1.val = |Number of Lines in file =
| & !n & |; | & !fileSize
!this.par2.val = |Date modified = | & !date
endmethod
```

```
define method .load()
!file = !this.file
syscom |start $!file&|
endmethod
```

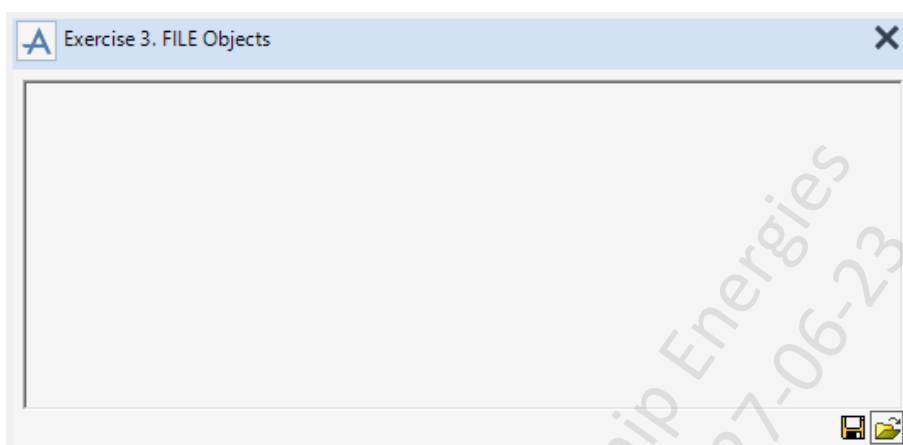
Try showing the file and see how Windows displays it. Try updating the example so that other file types can be opened. What programs are used to display this file.

 Refer to the Reference Manual and Guide for more information about radio toggle gadgets.

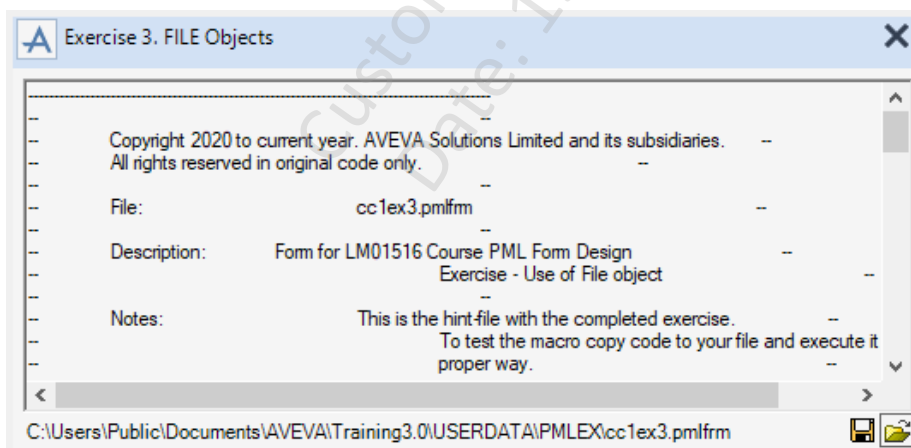
## Exercise 3 Use of File object

Perform the following task:

- Create a new form called **!!cc1ex3** and save it to the PMLLIB area.
- The purpose of this form will be to use the **FILE** object and a **standard file browser** to turn a **textpane** into a text editor.
- The form will require a textpane for the text to be displayed and two buttons to load and save with. Consider using a paragraph gadget to provide feedback about the file being used.



- What file types should be able to be loaded? What happens if a non-ascii file type is opened? Can the user save a blank file?
- Test the form with a range of files.
- What other features could be added to the form to enhance the user experience? Investigate the FILE object and TEXTPANE gadget further to see if anything more can be added.



 An example of the completed form “Use of File Object” can be found in the Training Assistant.

## 4 Collections

---

A very powerful feature of the AVEVA™ E3D Design database is the ability to collect and evaluate data according to rules. There are two available methods for collection that are valid in PML. The first is a command syntax PML 1 style and the other is with a PML **COLLECTION** object. Both are still valid, but when developing new PML, a **COLLECTION** object should be used in preference.

### 4.1 COLLECT Command Syntax (PML 1 Style)

---

The **COLLECT** syntax is based around three specific pieces of information:

- What element type is required?
- If specific elements are required, what aspect can be used to identify them.
- Which part of the hierarchy to look it?

If you wish to collect all the **EQUI** elements for the current **ZONE**, enter the following:

```
var !equipment collect all EQUI for ZONE
```

```
q var !equipment
```

If you wish to collect all the piping components owned by a specific **BRAN**, enter the following:

```
var !pipeComponents collect ALL with owner eq /200-B-4/B1 for SITE
```

```
q var !pipeComponents
```

if you wish to collect all the **BOX** primitives below the current element, enter the following:

```
var !boxes collect all BOX for ce
```

```
q var !boxes
```

 *You do not need to specify level of the hierarchy to search, if left out the entire MDB will be searched.*

 *For more examples, refer to Collections in Database Management Reference available at Aveva help.*

### 4.2 EVALUATE Command Syntax (PML 1 Style)

---

After elements have been collected through the **COLLECT** syntax, they are stored as an **ARRAY** of **STRING**s. This array (like any array) can be processed using the **EVALUATE** syntax to provide further information.

To get the names of all the elements held in **!equipment**, enter the following:

```
var !equipmentNames evaluate NAME for ALL from !equipment
```

```
q var !equipmentNames
```

To get the fullnames of the elbows held within **!pipeComponents**, enter the following:

```
var !elbowNames evaluate FLNN for all ELBO from !pipeComponents
```

```
q var !elbowNames
```

To get the names (without the leading slash) of the pumps in **!equipment**, enter the following:


```
var !pumpNames evaluate NAMN for ALL with MATCHWILD(NAMN, |P*|) from !equipment
```

```
q var !pumpNames
```

To get the volume of all the boxes in **!boxes**, enter the following:

```
var !volume eval (xlen * ylen * zlen) for ALL from !boxes
```

```
q var !volume
```

 Notice how the expressions are command syntax and must return a **BOOLEAN**. Refer to the *Software Customisation Reference Manual* for more examples.

### 4.3 COLLECTION Object (PML 2 Style)

A **COLLECTION** object is an example of a PML object that has directly replaced command syntax. It is recommended that all new PML uses **COLLECTION** objects as required. One advantage of using a **COLLECTION** object is that an **ARRAY OF DBREF** objects is returned.

A **COLLECTION** object is assigned to a variable in the same way as other objects:

```
!collection = object COLLECTION()
```

```
q var !collection
```

```
q var !collection.methods()
```

Once assigned, the methods on the object are used to set up the parameters of the collection. To set the element type for the collection to only **EQUI** elements, enter the following:

```
!collection.type(|EQUI|)
```

If more than one element type is required, the following could be entered:

```
!elementTypes = |EQUI BRAN SCTN|
```

```
!collection.type( !elementTypes )
```

The scope of the collection must be a **DBREF** object and should be passed as an argument to the **.scope()** method. For example, enter the following:

```
!collection.scope( !lce )
```



To filter the results, the **.filter()** method must be passed an **EXPRESSION** object. For example, to filter to members of a named branch, enter the following:

```
!expression = object EXPRESSION( | name of owner eq '/200-B-4/B1' | )
```

```
!collection.filter( !expression )
```

Once the collection object has been set up, the **.results()** is used to return the collected elements as an **ARRAY** of **DBREF** objects.

```
!results = !collection.results()
```

```
q var !results
```

#### 4.4 Evaluating the Results from a COLLECTION Object

After an **ARRAY** of **DBREF** objects has been generated from a **COLLECTION** object, the entire array can be evaluated using the **.evaluate()** method on an array object. The argument for the **.evaluate()** method must be a **BLOCK** object defined with the expression that needs evaluating.

To get an **ARRAY** of **STRINGS** holding the full names of the collected elements, enter the following:

```
!block = object BLOCK( | !results[!evalIndex].flnn | )
```

```
!resultNames = !results.evaluate(!block)
```

```
q var !resultNames
```

Notice how the **BLOCK** object uses the local variable **!evalIndex**. This variable effectively allows the **.evaluate()** method to loop through the **ARRAY**. To get the positions of the collected elements, enter the following:

```
!resultPos = !results.evaluate( object BLOCK( | !results[!evalIndex].pos | ) )
```

```
q var !resultPos
```

Instead of defining the block as a separate variable, this second example shows that the object can be defined within the argument to another object.

 Notice how the evaluated **ARRAY** contains the correct object types generated from the evaluation.

## Exercise 4 Equipments Collection

Nozzles	Connected?	Attached?
E1301/NS1	OK	OK
E1301/NS2	OK	OK
E1301/N1	OK	OK
E1301/N3	OK	OK

- Create a new form called **!cc1ex4** that will act as an **Equipment Checker**. The purpose of the form is to provide users information about **EQUI** elements from the hierarchy.
- Design the form with a **COMBO** gadget to display to the user the available **EQUI** elements, a **LIST** gadget to display the **NOZZ** elements owned by the chosen **EQUI**, a **BUTTON** to update the form and a **PARAGRAPH** as a title.
- Write a method that collects all the **EQUI** elements for the **ZONE** of the **CE**. Try and write the collection part of the method in a PML 1 style.
- What happens if no pieces of equipment are found in the **ZONE**? Call this method when the form is shown.
- Add an **OPEN CALLBACK** to the **COMBO** gadget to validate any user entered values.
- Add an **'Update'** button that will re-run this method if a different **ZONE** is chosen. What happens if the user is at a **SITE** when the update button is clicked?
- Write a method that runs after the piece of equipment has been chosen. This method should collect all the **NOZZ** elements for that **EQUI** element. Try writing the collection part of the method in a PML 2 style.
- After the method has run, set the full names of the collected **NOZZ** elements to the **LIST**. Make use of the **RTEXT** and **DTEXT** members of the gadgets (to store names and **DBREFs**)
- Add a popup menu to the **LIST** gadget to allow the users to navigate to the chosen element (set the current element).
- What method on a **LIST** gadget retrieves the chosen **RTEXT**?

- Write a method that looks at each of the collected NOZZ element and checks the following:
  - Is it connected? This can be decided by checking the NOZZ element's CREF attribute. If the attribute is unset or the reference is invalid, then the user will have to check that nozzle.
  - Is it attached? Check the positions of the NOZZ and PIPE elements and if they are different, then the NOZZ should be checked.
  - Is it aligned? Check the directions of the NOZZ and PIPE elements. If they are different, then they should be checked.
  - Is it sized correctly Compare the diameter of the PIPE to the size of NOZZ. If they are different, then they should be checked.
- Display the results of the check in the LIST gadget using columns.
- Provide a **Refresh** button to recheck the nozzles in case the users change anything in response to the check results. This button should rerun the check method and update the form correctly.

The screenshot shows a window titled "Exercise 4. Equipment Checker". It has a button "Update" and text "Available Equipments below ZONE". Below this is a dropdown menu "Select an Equipment" with "E1301" selected. Below the dropdown is a table with 5 columns: "Nozzles", "Connected?", "Attached?", "Aligned?", and "Size?". The table contains 4 rows of data, all with "OK" values.

Nozzles	Connected?	Attached?	Aligned?	Size?
E1301/NS1	OK	OK	OK	OK
E1301/NS2	OK	OK	OK	OK
E1301/N1	OK	OK	OK	OK
E1301/N3	OK	OK	OK	OK

Add a **TEXTPANE** to the form to allow users to enter attribute values for the chosen piece of equipment.

- The **TEXTPANE** should initially be filled with three attributes (**Description, Function and Purpose**) and the values for the chosen equipment. As the user chooses other pieces of equipment, the displayed attribute values should update.

The screenshot shows the same window as before, but with an additional section at the bottom. The "Select an Equipment" dropdown still shows "E1301". Below the table, there is a "Refresh Checks" button. Below that, there is a section titled "E1301 Attributes" which contains three text areas: "Description -", "Function -", and "Purpose -". At the bottom right of this section is an "Update Attributes" button.

- Write a method to read the attributes and values from the **TEXTPANE**. This will involve splitting the information entered by the user. If the user has entered valid values, the attributes should be updated.
- This method should cope with new attributes being entered by the user. Consider what should happen if an invalid attribute or value is entered.
- As the **TEXTPANE** has no callback, provide a button to call this method.

 *An example of the completed form “Equipment Collection” can be found in the Training Assistant.*

Customer: Technip Energies  
Date: 19-06-23 - 27-06-23

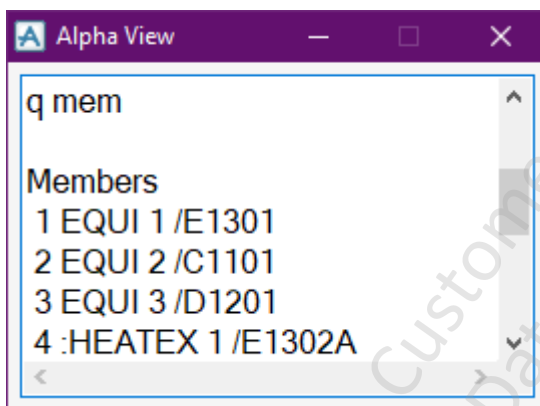
## 5 View Gadgets

VIEW gadgets are named gadgets which are used to display the information from available databases. The way the information is displayed depends on the VIEW gadget type.

- General View Types:
  - **ALPHA** views for displaying text output and / or allowing command input.
  - **PLOT** views for displaying non-interactive 2D plot files.
- Application-specific View Types
  - **AREA** views for displaying interactive 2D graphical views.
  - **VOLUME** views for displaying interactive 3D graphical views.

 Refer to the AVEVA™ E3D Design Customisation Reference Manual for more information.

### 5.1 Alpha Views

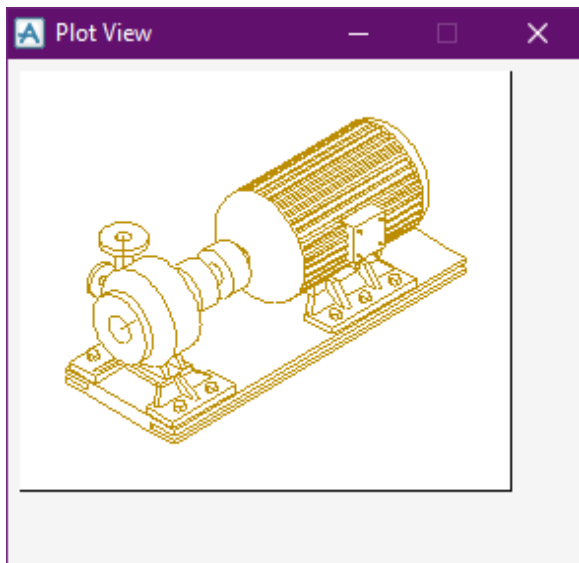


An **ALPHA** view gadget is the same as used for the standard command window.

For the example, enter **show !!traExampleAlphaView:**

```
layout form !!traExampleAlphaView
!this.formTitle = |Alpha View|
view .input AT X 0 Y 0 ALPHA
height 10 width 30
channel REQUESTS
channel COMMANDS
exit
exit
```

## 5.2 Plot View Example



A **PLOT** view can be used to provide the user with extra information. This extra information would be contained in a **.plt** file and is applied to the view. The following example applies the file during the constructor method.

For the example, enter **show !!traExamplePlotView:**

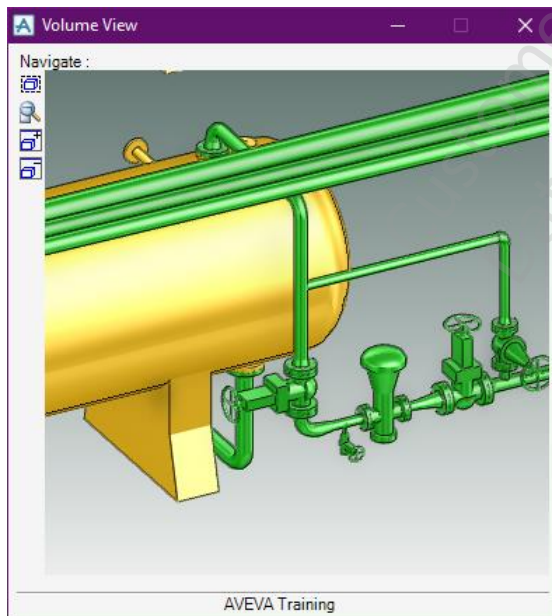
```
layout form !!traExamplePlotView
!this.formTitle = |Plot View|
view .plot plot width 35 height 10
CURS NOCURSOR
exit
exit

define method .traExamplePlotView()
!this.plot.borders = FALSE
!this.plot.background = 10
!this.plot.add(|%PMLLIB%\CC1_PVE.plt|)
endmethod
```

## 5.3 Volume View Example

A **VOLUME** view can be used to see the 3D data held within the AVEVA™ E3D Design database. The main AVEVA™ E3D Design window is an example of a volume view in use.

For the example, enter **show !!traExampleVolumeView:**



 The file *traExampleVolumeView.pmlfrm* is available within the *PMLLIB* directory.

With AVEVA™ E3D Design, it is possible to assign different drawlists to different volume views. This has been demonstrated in the example by giving the volume view a different drawlist to the main view. There are two global variables that make this possible: **!!gphviews** and **!!gphdrawlists**

To investigate the **!!gphdrawlists**, enter the following into the command window:

**q var !!gphDrawlists**

<GPHDRAWLISTS> GPHDRAWLISTS

DRAWLISTS <ARRAY> 1 Elements

**q var !!gphDrawlists.drawlists**

<ARRAY>

[1] <DRAWLIST> DRAWLIST

**q var !!gphDrawlists.methods()**

The following are some examples of the methods which are available on **!!gphDrawlists**. To find out information about all the drawlists in the global object, use:

**!!gphDrawlists.listall()**

To create a drawlist object in the global object, use:

**!drawlist = !!gphDrawlists.createDrawlist()**

**q var !drawlist**

To associate a drawlist from the global object with a view gadget, use:

**!!gphDrawlists.attachView(DRAWLIST, GADGET)**

Where **DRAWLIST** is a REAL represented the drawlist and **GADGET** is the volume view. Drawlists can only be attached to views that has been registered with the **!!gphViews** object. To investigate the **!!gphViews**, enter the following into the command window:

**q var !!gphViews**

<GPHVIEWS> GPHVIEWS

ACTIVEVIEW <GADGET> Unset

SELECTEDVIEWS <ARRAY> 0 Elements

VIEWS <ARRAY> 1 Elements

**q var !!gphViews.methods()**

To add a view to the global object, use:

**!!gphViews.add(GADGET)**





where GADGET is the volume view.

To set the limits of a view based on a DBREF object, use:

**!!gphViews.limits(GADGET, DBREF)**

where **GADGET** is the volume view and **DBREF** is the element to set the limits to.

The example form also mimics the functionality of the main 3D window by providing the following four buttons:

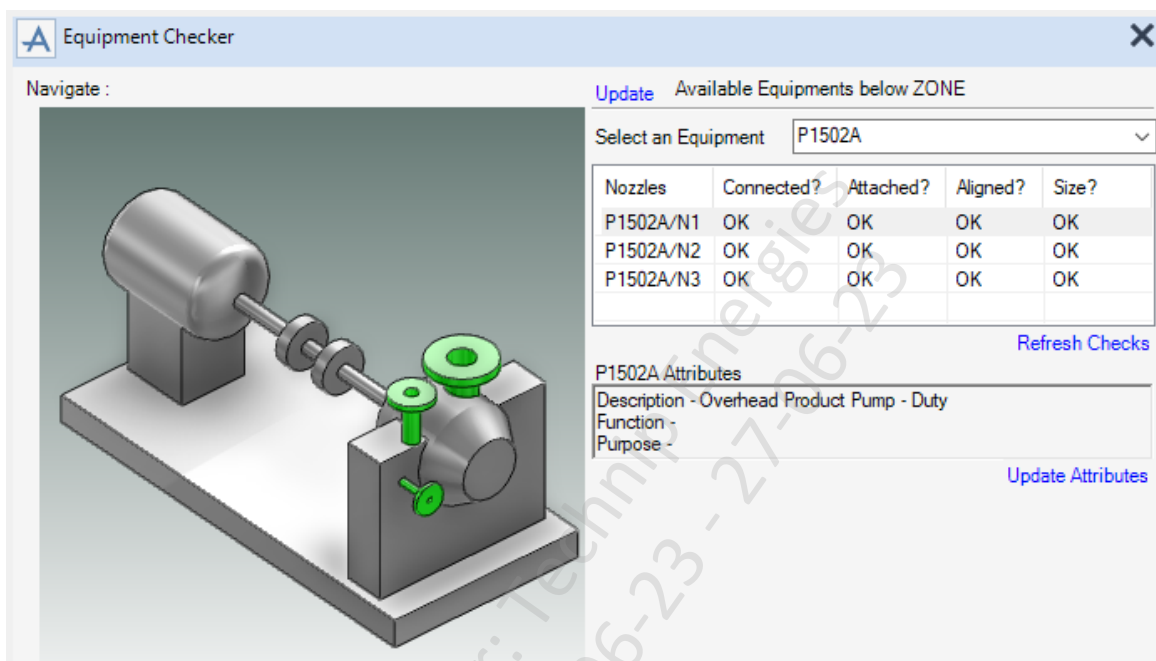
-  Set view limits to CE.
-  Walk to Drawlist (set limits of the view based on drawlist).
-  Add the current element to the drawlist.
-  Remove the current element from the drawlist.

 *These buttons are not always necessary when defining a volume view element but have been included in the example to demonstrate some further methods on the object.*

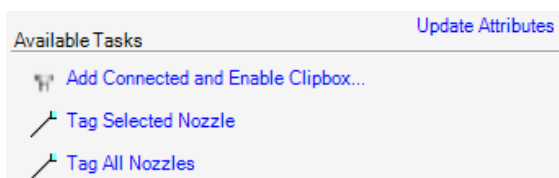


## Exercise 5 Adding a Volume View to a form

- Add a **VOLUME VIEW** element to the form created in Exercise 4. Save form to a new file called **cc1ex5.pmlfrm**. The Volume View should have its own drawlist and should correctly display the chosen piece of equipment only.
- Look at **!!exampleVolumeView** and see which parts of the PML can be re-used. How will the method know which element has been chosen and how will the drawlist/limits be updated?
- An example of the updated form is below (new view and given the ability to dock).

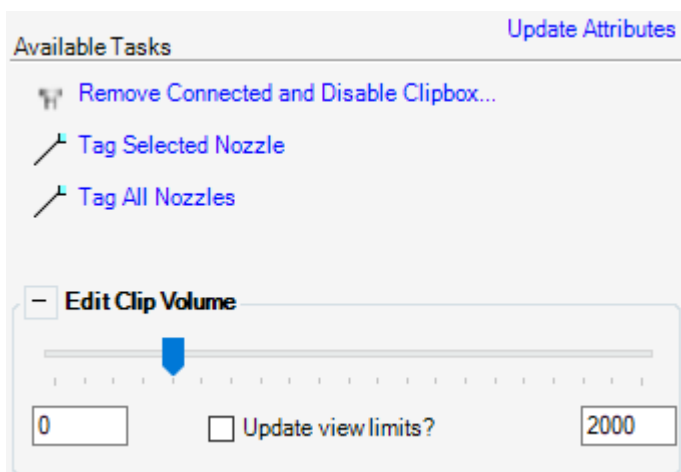


- Add an '**Available Tasks**' section to the form. Any additional functionality added to the form shall be added here.
- Add a method to the form to tag **NOZZ** elements on the chosen piece of equipment. This will allow to the user to identify which nozzle is which.
- Provide two extra tasks: **Tag Selected Nozzle** (chosen in the list) and **Tag All Nozzles**.

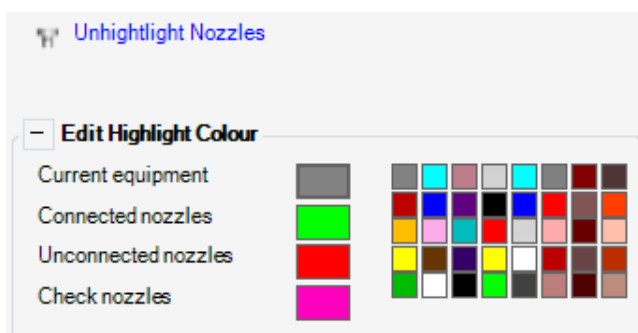


- When the user tags a nozzle, use the **MARK** or **AID TEXT** syntax to put the name of the nozzle into the volume view. The method should keep track of which nozzles are tagged so that they are only tagged once (use a form member?). This will also manage the state of the toggle so that it is always correct.
- Update the '**Add Connected**' method to include a clipboard. This will limit how much the user sees. A clipboard is added to a Volume View by using a **GPHCLIPBOX** object. The **VIEW** member of a **GPHCLIPBOX** object holds the name of the Volume View you wish to clip.

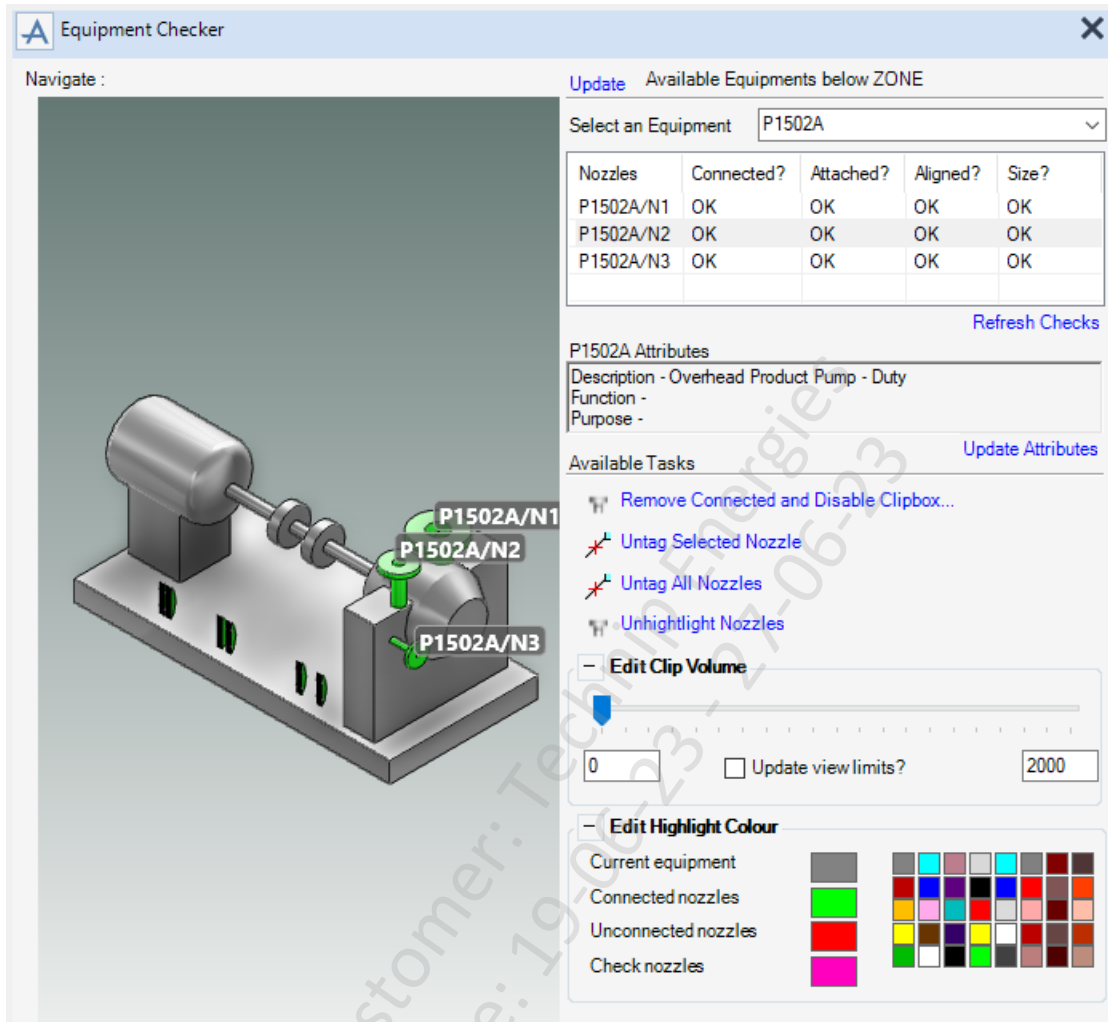
- The **GPHCLIPBOX** object should be stored as a member of the form. This is so it can be resized as different elements are chosen. To find out more about the **GPHCLIPBOX** object, open **gphclipbox.pmlobj**.
- When the user turns the clipbox on, they should then see a hidden fold-up panel which holds a slider. This slider will allow the user to dynamically change the size of the clipbox. Make use of the **.refresh()** method to update the volume view during the slide.



- Add a method to update the range of the slider based on the input into two text boxes. This will be in case the equipment is too large/small for the default value.
- Add a method that will update the limits of the view when the clipbox is resized. This is to stop the clipbox becoming larger than the limits of the view.
- Add a method to the form that will add colour to the elements of the drawlist to indicate the results of the nozzle check. Colour can be added through the **.highlight()** method on a **DRAWLIST** object.
- Three different colours should be added. A colour for unconnected nozzles, a colour for nozzles that need checking and a colour for connected nozzles which pass the checks. You may also wish to highlight the chosen piece of equipment.
- When the user turns on the highlighting, show a hidden frame which allows users to change the colours. Four buttons shall display the current colours. On pressing one of those buttons, a hidden panel frame shall be shown which will allow the users to update the colour. Think about how a DO loop could be used to create this grid of buttons.



- The methods will need to manage which elements are coloured, which colours should be used and the visibility of the gadgets on the form.
- To complete the form, test the functionality on a range of equipment elements. Address any errors that occur during this process.



 An example of the completed form “Adding a Volume View to a form” can be found in the Training Assistant.

This page left intentionally blank

Customer: Technip Energies  
Date: 19-06-23 - 27-06-23

## 6 Event Driven Graphics (EDG)

The EDG has been developed to allow a common interface for appware developers to use when setting up the graphic canvas for graphical selection. This method is relatively simple and easily extendible. This will allow the developer to concentrate on the development of their own application without the need to know the underlying mechanism (core implementation) of the EDG interaction handlers and system.

The system handles all the underlying maintenance of the current events stacked e.g. associated forms, initialisation sequences, close sequences, etc.

The current implementation of the system has mainly been developed for interaction with the 3D graphic views in the Design module. However, the interface can be used with any of the modules that use the same executable and the standard 3D graphic views. It is intended that EDG will supersede the old ID@ syntax. When setting up an EDG event, the following should be considered:

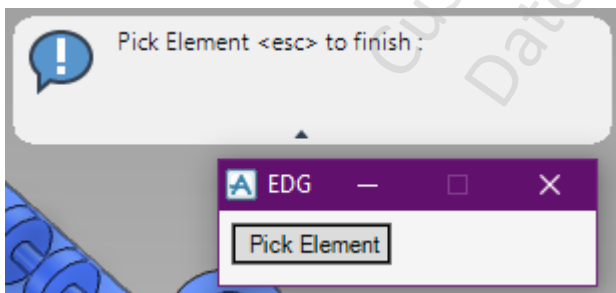
- What items will the user be picking?
- How many picks are required and in what order?
- What happens when the user makes a pick?
- What happens when the user has finished picking?

These aspects are controlled by methods on the objects associated with EDG. The main object is an **EDGPACKET** object. Once setup, this object is added to the **!!EDGCtrl** object that will activate the event.

 For more information about EDG, refer to the chapter Event Driven Graphics interface in Software Customisation Reference manual.

### 6.1 A Simple EDG Event

To demonstrate a simple EDG setup, show the example by entering **show !!traExampleSimpleEDG:**



```
layout form !!traExampleSimpleEDG
!this.formTitle = |EDG|
button .but1 |Pick Element| call
!!this.pick()
exit
```

```
define method .pick()
-- Define event packet object
!packet = object EDGPACKET()
-- Define object as a predefined standard element pick
!packet.elementPick(|Pick Element <esc> to finish|)
-- Query the item member of the returned information from the pick
!packet.action = | !!traExampleSimpleEDG.process(!this.return[1]) |
```

```
-- Set what happens when the user presses esc
!packet.close = |$p Finished|
-- Add the event packet to the global EDG control object
!!EDGCtrl.add(!packet)
Endmethod

define method .process( !item is ANY )
  q var !item
endmethod
```

The example sets up a predefined element pick event by running the **.elementPick()** method on a **EDGPACKET** object. The subsequent methods specify what happens when a pick is made and what happens when the event is finished.

**i** Notice how the action method of the **EDGPACKET** object references the form method explicitly. It displays the returned information (described in chapter Event Driven Graphics in software customisation reference manual).

## 6.2 Using EDG

The following example shows how EDG can be applied to a form to provide picking functionality for users. To show the example, enter **show !!traExampleEDG**:

```
layout form !!traExampleEDG
!this.formTitle = |Pick Equipment|
!this.initcall = |!this.init()|
!this.quitcall = |!this.clear()|
button .pick |Start Pick| call |!this.init()|
textpane .txt1 |Picked Equipments| at x 0 ymax.pick+0.2 width 25 height 5
member .storage is ARRAY
exit

define method .init()
!this.clear()
!packet = object EDGPACKET()
!packet.elementPick(|Pick Equipments <esc> to add to list|)
!packet.description = |Identify Equipment|
!packet.action = |!!traExampleEDG.identify(!this.return[1].item)|
!packet.escape = |!!traExampleEDG.setInfo()|
!!EDGCtrl.add(!packet)
endmethod

define method .identify( !pick is DBREF )
do
  if !pick.type.eq(|EQUI|) or !pick.type.eq(|WORL|) then
    break
  else
    !pick = !pick.owner
  endif
enddo
```

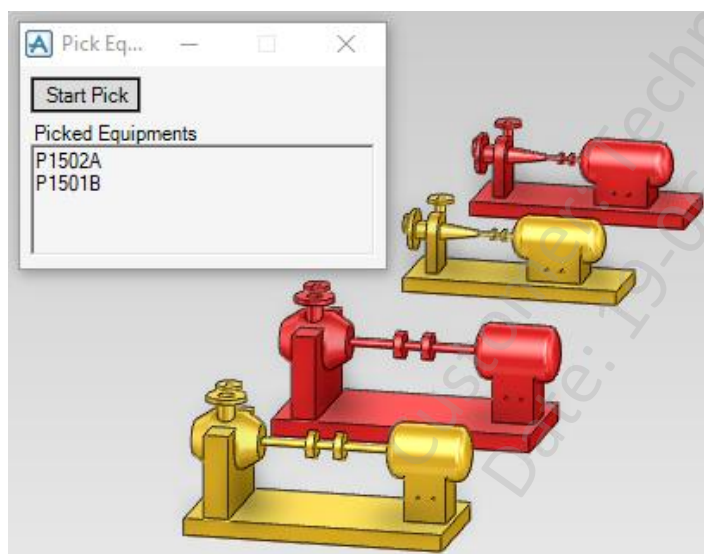
```

if !pick.type.eq(|EQUI|) then
  if !this.storage.findfirst(!pick).unset() then
    !!gphDrawlists.drawlists[1].highlight(!pick, 314)
    !this.storage.append(!pick)
  else
    !this.storage.remove(!this.storage.findfirst(!pick))
    !!gphDrawlists.drawlists[1].unhighlight(!pick)
  endif
endif
endmethod

define method .setInfo()
  !block = object BLOCK(|!this.storage[!evalIndex].flnn|)
  !names = !this.storage.evaluate(!block)
  !this.txt1.val = !names
endmethod

define method .clear()
  do !element values !this.storage
    !!gphDrawlists.drawlists[1].unhighlight(!element)
  Enddo
  !!edgCntrl.remove(|Identify Equipment|)
  !this.txt1.clear()
  !this.storage.clear()
endmethod

```



This example shows how a method can take the returned information and use it. In this case, the picked element type is checked whether it is a piece of equipment. If it is not, the method loops up the hierarchy until one is found (or the world is reached). The EQUI is then highlighted if it's new or unhighlighted if already chosen. There is a different method for the close action. For this reason, the picked elements are collected a form member. The escape method (press ESC keyboard button) applies the form member to the textpane gadget.

**i** Notice how the form uses the `.clear()` method to tidy up.

## Exercise 6 Adding EDG to Forms

- Add a pixmap button to the form created in Exercise 4 that will allow users to pick which equipment they want from the main 3D window. Save form to a new file called cc1ex6.pmlfrm. Once picked, that piece of equipment (if available) should be selected in the **OPTION** gadget.
- The button should initialise an EDG event. This EDG event should run a method on picking an element that does the following:
  - Turn the EDG event off after something has been chosen.
  - Identify the type of element chosen.
  - If an EQUI has been chosen, OK.
  - Otherwise, loop up the hierarchy to find an EQUI.
  - If an EQUI cannot be found, report to the user and re-initialise the EDG.
  - Find the chosen EQUI in the OPTION gadget and set it to the correct equipment.
  - Run the nozzle collection method.
- Extend the method to allow NOZZ elements to be chosen (as well as EQUIs). When a nozzle is chosen, highlight it in the LIST. Consider what happens if a nozzle is chosen which is not in the list but is owned by a piece of equipment in the OPTION gadget.

**Exercise 6. Equipment Checker**

Update Available Equipments below ZONE ZONE-EQUIPMENT-A

Select an Equipment

Nozzles	Connected?	Attached?	Aligned?	Size?
P1501A/N1	OK	OK	OK	OK
P1501A/N2	OK	OK	OK	OK
P1501A/N3	OK	OK	OK	OK

Refresh Checks

P1501A Attributes

Description - Reflux Pump - Duty  
Function -  
Purpose -

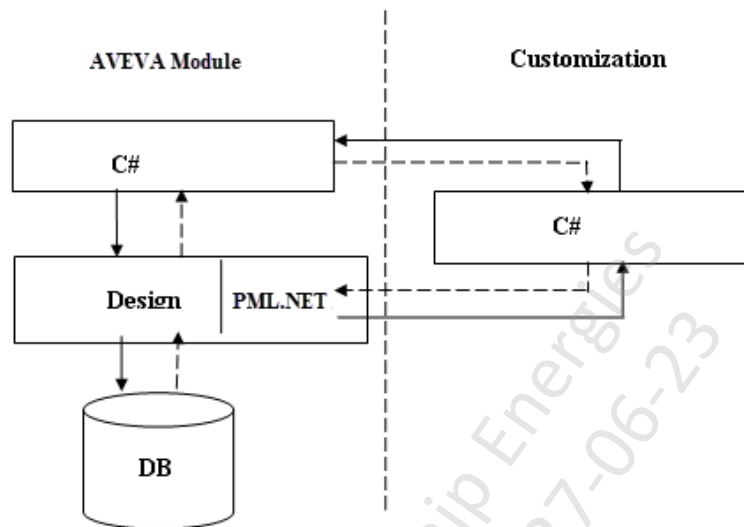
Update Attributes

 An example of the completed form “Adding EDG to Forms” can be found in the Training Assistant.



## 7 PML.NET

Since the AVEVA Plant 12 release, it is now possible to customise the product using .NET through .NET controls and objects. PML .NET allows you to instantiate and invokes methods on .NET objects from PML proxy objects.



- PML proxy class definitions are created from .NET class definitions at run time. These proxy classes present the same methods as the .NET class which are described using custom attributes. The PML proxy objects behave just like any other PML object.
- PML callable assemblies are loaded by AVEVA™ E3D Design using the IMPORT syntax. Assemblies may be defined in potentially any .NET language (for example managed C++, C# or VB.NET). The PML.Net Engine loads a given assembly and once loaded instances of PMLNetCallable classes may be created. No additional code to interface between PML and .NET is necessary. This is provided by the PMLNetEngine.

It is possible to add container objects to a conventional PML form creating a hybrid of PML & .NET. The container is treated as a normal form gadget (i.e. syntax graph) but can be filled with a .NET object

### 7.1 Import an Assembly into AVEVA™ E3D Design

Before a .NET control can be used in a form, it first has to be imported into AVEVA™ E3D Design. The required .dll file has to be loaded and the namespace specified. For example:

```
import 'GridControl'
```


```
using namespace 'Aveva.Core.Presentation'
```

**i** *Trying to import a .dll file which has already loaded or cannot be found will result in an error. Error handling should be considered to prevent the error messages being displayed to users.*

## 7.2 Syntax

To define your first .NET object, enter out the following into the Command Window:

```
import 'GridControl'  
  
using namespace 'Aveva.Core.Presentation'  
  
!netObj = object NETGRIDCONTROL()  
  
q var !netObj  
  
q var !netObj.methods()
```

 Notice the `.methods()` method. This is a new global method available on all objects. It returns all of the methods.

## 7.3 Create a File browser Object

- The following example shows **File browser** object. To Create a File Browser object, enter the following onto the **Command Window**.

```
import 'PMLFileBrowser'  
  
using namespace 'Aveva.Core.Presentation'  
  
!browser = object PMLFileBrowser('OPEN')
```

 The arguments are `OPEN` and `SAVE`. If no argument is given, then `OPEN` is used by default.

- To query the object and the methods of the **PMLFILEBROWSER** instance.

```
q var !browser  
  
q var !browser.methods()  
  
q var !browser.file()
```

- To display the browser form, enter the following:

```
!browser.show('%PMLLIB%', 'cc1ex1.pmlfrm', 'Example', true, ' Macro files (*.mac)|*.mac|PML  
Forms (*.pmlfrm)|*.pmlfrm| Text files (*.txt)|*.txt|All files (*.*)|*.*',2)
```

- The arguments to the `.show` method set up the browser, they are as follows:
  - Start directory (String)
  - Initial file name (String)
  - Title for the browser window (String)

- Should the file exist (Boolean)
- File filter (String) - type of files to open. A description, followed by “|” and then the file types – for example: Word Documents|\*.DOC|Text files (\*.txt)|\*.txt|All files (\*.\*)|\*.\*
- Index (integer) – for multiple strings, the index of the filter currently selected.

## 7.4 Creating a PML form containing the .NET Control

Once the **.dll** file has been loaded and the namespace specified the object is available for use. For example, a user may specify a **CONTAINER** within a form definition to hold the .NET control and a form member to define it:

```
container .netFrame PMLNETCONTROL 'NET' dock fill
```

```
member .netControl is NETGRIDCONTROL
```

After this, the **constructor method** would define the .NET object and apply it to the container gadget.

```
using namespace 'Aveva.Core.Presentation'
```

```
!this.netControl = object NETGRIDCONTROL()
```

```
!this.netFrame.control = !this.netControl.handle()
```

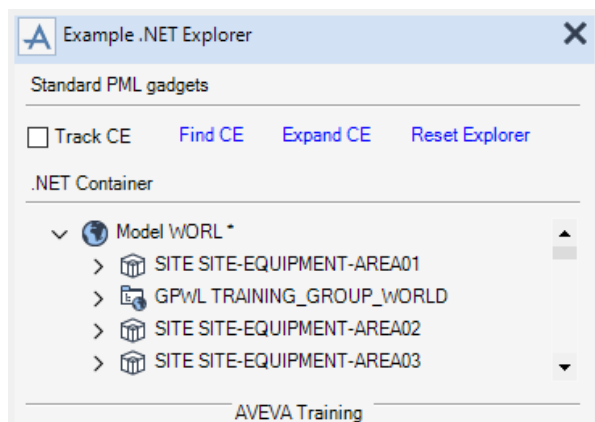
The user is now able to specify event methods to control the .NET gadget:

```
!this.netControl.addeventhandler('OnPopup', !this, 'rightClickExplorer')
```

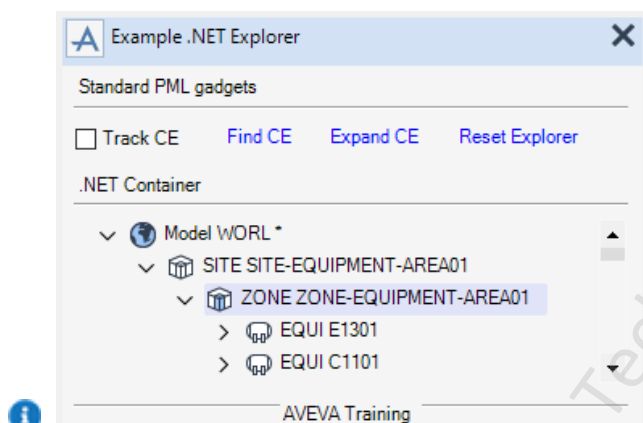
This allows the forms **.rightClickExplorer()** method to be run when the .NET object is right clicked.

## Exercise 7 Explorer Control form

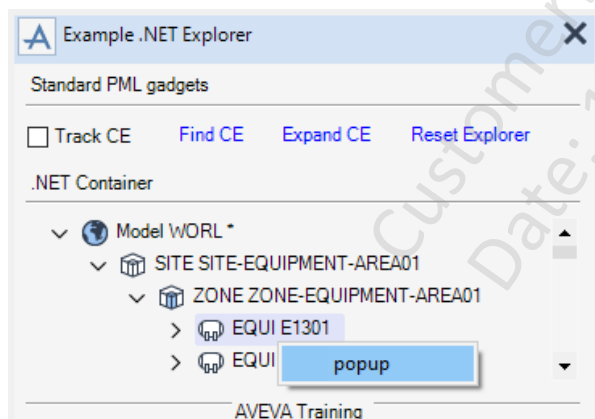
Perform the following tasks:



- Enter the following on the **Command Window**:  
**show !!traExampleExplorer**
- You will see a form which uses a container gadget to display a .NET explorer control.
- The example code for this form can be found in %PMLLIB%\traExampleExplorer.pmlfrm



- Experiment with the “Track CE” toggle and the “Find CE” functionality.
- If you “Expand CE”, what happens?
- Try to “Reset Explorer”.



- A popup menu has been programmed on the explorer control.
- The current method prints the name of the element to the command window.
- Create a copy of **!!traExampleExplorer** form and rename it to **!!cc1ex7**.
- Extend this method to increase the functionality of the new form **!!cc1ex7**. For example, add selected element to 3D View, show attributes and provide member information

 An example of the completed form “Explorer Control Form” can be found in the Training Assistant.

## 7.5 Grid Control

Another example of an available .NET object is the **Grid Control** gadget. Despite appearing similar to a **LIST** gadget, the Grid Control gadget has a large number of methods available to it allowing it to behave more like a spreadsheet. Some of the advantages are as follows:

- Data in the grid can be selected, sorted and filtered.
- Data in the grid can be exported to/imported from a Microsoft Excel file.
- Grid contents can be previewed and printed directly.
- The colour of rows, columns or cells can be set (including icons).
- The values of entire rows/columns can be extracted/set.
- The contents of the grid can be filled based on DB elements and their attributes.

### 7.5.1 Applying Data to the Grid

Data is applied to a Grid Control gadget through the use of a **NETDATASOURCE** object. While defining this object, the required data is collected and formatted for the grid. The data is applied to the grid by using its **.bindToDataSource()** method. For example:

```
var !coll collect all EQUI
```

```
!attribs = |NAME TYPE OWNER AREA|
```

```
!data = object NetDataSource('Example Grid ', !attribs.split(), !coll)
```

```
!this.exampleGrid.BindToDataSource(!data)
```

 *In this example, the `NetDataSource` object is collecting information from the database. There are other methods available. Refer to the *.NET Customisation User Guide* for more information.*

### 7.5.2 Events and Callbacks

The following events are available on the grid:

- OnPopup(Array)
  - Array[0] is the x coordinate of the current cursor position.
  - Array[1] is the y coordinate of the current cursor position.
  - Array[2] contains the ID of each of the selected rows

- BeforeCellUpdate(Array)
  - Array[0] is the new value the user entered in
  - Array[1] is the Row Tag of the cell
  - Array[2] is the Column key of the cell
  - Array[3] is a Boolean. If you set it to “false” then the new value will NOT be allowed
- AfterSelectChange(Array)
  - The array contains the ID of each of the selected rows
  - AfterCellUpdate
  - No arguments are passed.



*For more information, refer the .NET Customisation User Guide, available within the AVEVA™ E3D Design manuals.*

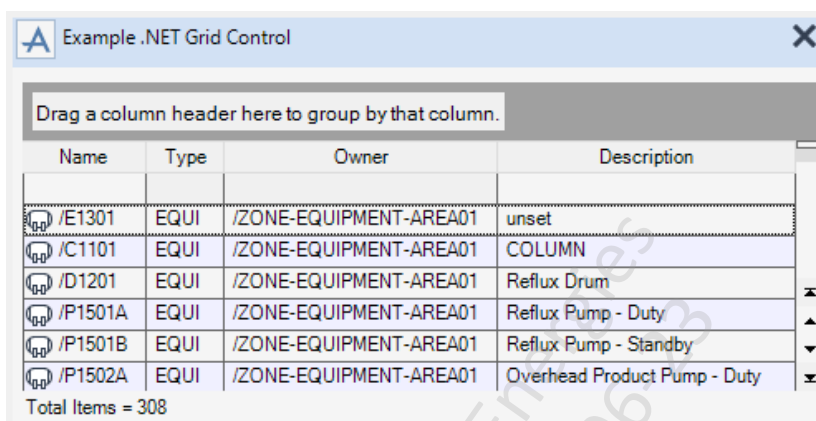
Customer: Technip Energies  
Date: 19-06-23 - 27-06-23

## Exercise 8 Grid Control Form

Perform the following tasks:

- Save a copy of %PMLLIB%\traExampleGridControl.pmlfrm form to a new file named !!cc1ex8.pmlfrm.
- Enter the following on the AVEVA™ E3D Design Command window:

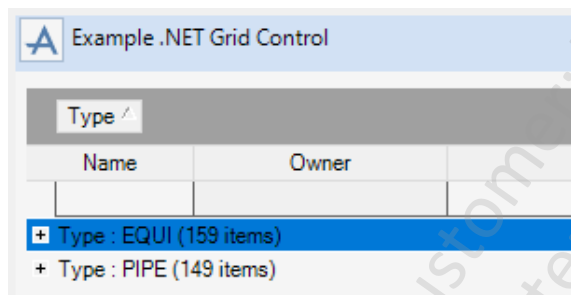
**show !!traExampleGridControl**



Name	Type	Owner	Description
/E1301	EQUI	/ZONE-EQUIPMENT-AREA01	unset
/C1101	EQUI	/ZONE-EQUIPMENT-AREA01	COLUMN
/D1201	EQUI	/ZONE-EQUIPMENT-AREA01	Reflux Drum
/P1501A	EQUI	/ZONE-EQUIPMENT-AREA01	Reflux Pump - Duty
/P1501B	EQUI	/ZONE-EQUIPMENT-AREA01	Reflux Pump - Standby
/P1502A	EQUI	/ZONE-EQUIPMENT-AREA01	Overhead Product Pump - Duty

Total Items = 308

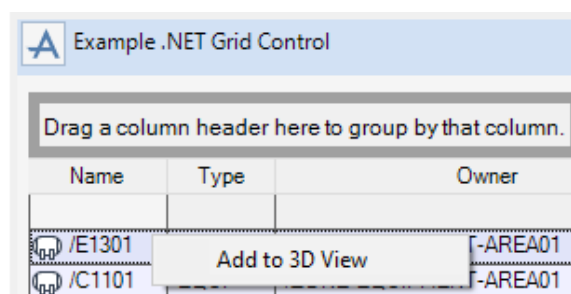
- i** The grid has been populated with a set of attributes (the headings), and a set of model items (Equipment and Pipes in this example). The grid control populates itself with the attribute data.



Type	Name	Owner
+ Type : EQUI (159 items)		
+ Type : PIPE (149 items)		

Grid Behaviour:

- Drag and drop the “TYPE” heading into the grouping area (see picture).
- Drag and drop selected items into the 3D view (or “My Data”).
- Investigate the filters on each column.

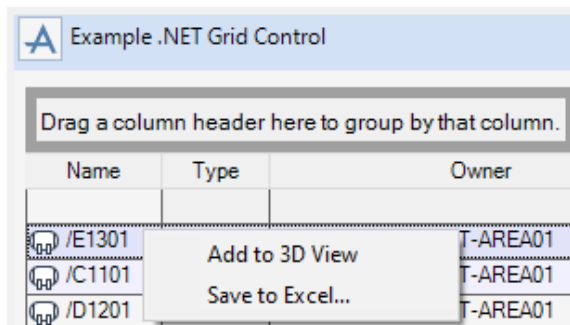


Name	Type	Owner
/E1301		-AREA01
/C1101		-AREA01

Add to 3D View

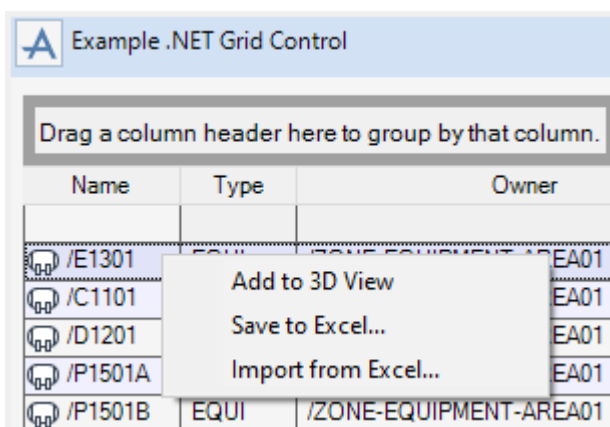
Pop-up Menu:

- Make a selection and open the pop-up menu. The menu doesn’t do anything yet.
- You need to edit the PML form to enable the Popup menu to run a callback function.



Save Data to an Excel file:

- Add to the popup menu the ability to save the grid data to an Excel spreadsheet.
- Make use of the File Browser object to enable the user to specify the Excel file on the file system.



Import Data from an Excel file:

- Add to the popup menu the ability to import data from an excel spreadsheet into the grid.
- The first row in the Excel file will represent the column headings. The other rows will be data.

Event - afterSelectChange:

- Add an “AfterSelectChange” event.
- Add a method to print out the name of the new selection whenever you change the selection in the grid.

Grid Control API:

- Try out some of the other events and methods available on the Grid Control.

**i** Note that the imported data will replace the data which is already there (not append to it) and the data is completely defined by the spreadsheet.

**i** An example of the completed form “Example .NET Grid Control” can be found in the Training Assistant.

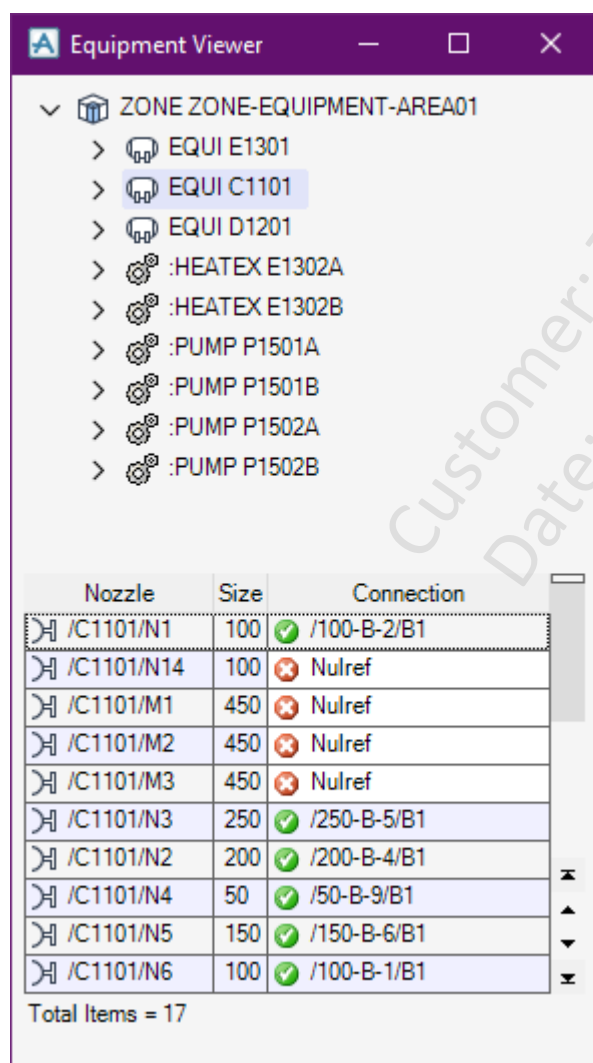


## Exercise 9 Developing a PML .NET form

Perform the following task:

- Create a new form called **!!cc1ex9** and save it to the PMLLIB area.
- The purpose of this form will be to use the PML .NET controls to create an Equipment Viewer form. This form will mimic the functionality of exercise 5 but will benefit from the increased user experience the .NET controls bring.
- Provide a **.NET Explorer** which displays the **ZONE /ZONE-EQUIPMENT-AREA01** and below from the example test model.
- Provide a **.NET Grid Control** gadget. This gadget will display the names of the nozzles owned by the chosen, as well as their size and connection information. The contents of the **Grid** should refresh every time a piece of equipment is chosen.
- When displaying the connections of the **NOZZ** elements, make it more obvious to the user which ones are connected. This could be done by changing colours, display text or displaying icons.

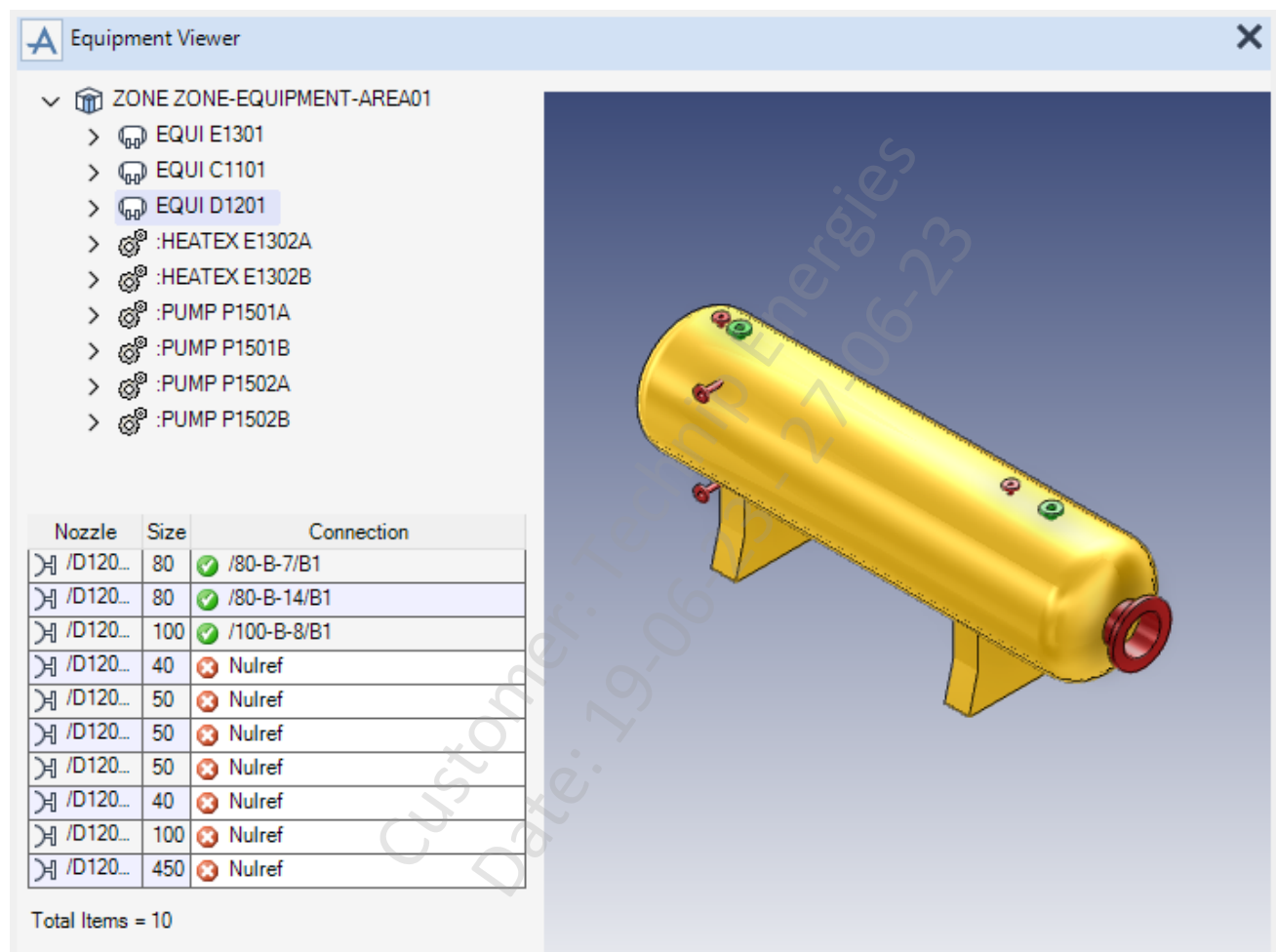
 Refer to the *.NET customisation Guide for suitable methods.*



Nozzle	Size	Connection
/C1101/N1	100	✓ /100-B-2/B1
/C1101/N14	100	✗ Nulref
/C1101/M1	450	✗ Nulref
/C1101/M2	450	✗ Nulref
/C1101/M3	450	✗ Nulref
/C1101/N3	250	✓ /250-B-5/B1
/C1101/N2	200	✓ /200-B-4/B1
/C1101/N4	50	✓ /50-B-9/B1
/C1101/N5	150	✓ /150-B-6/B1
/C1101/N6	100	✓ /100-B-1/B1

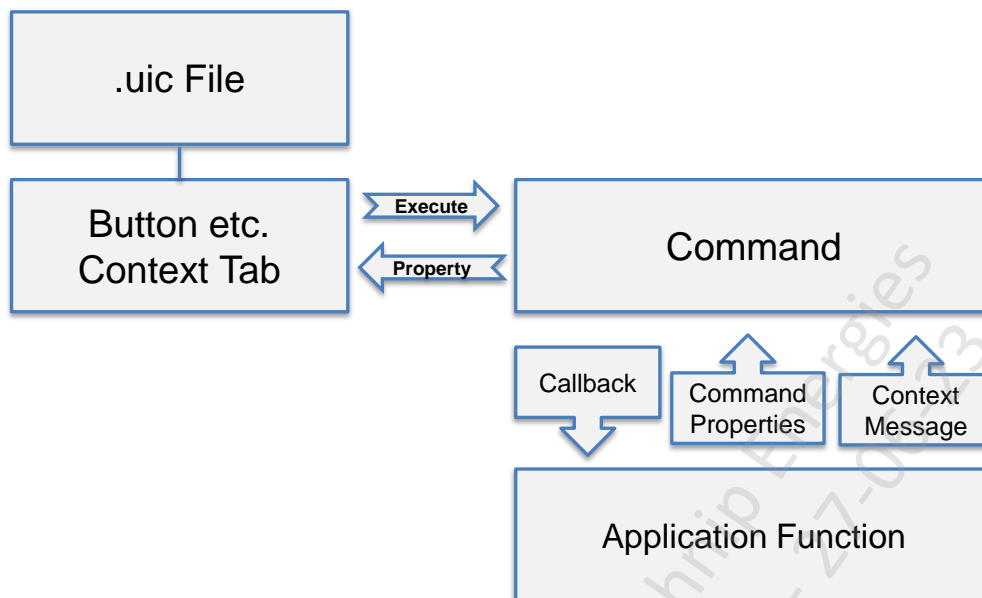
Total Items = 17

- Extend the form by adding a volume view gadget to the form.
- Give the volume view its own drawlist.
- Consider the management of this drawlist (when should it be created, modified, removed)
- What should be displayed in the drawlist?
- There are many methods available on a drawlist object. Provide a way for the user to see the results of the connection check within the view.
- Test the form on a range of elements.



## 8 Implementation of Command in Model

Commands can be implemented as PML command objects or core command objects. The diagram shows the Command Object linking tools defined in user interface configuration (uic) file with application functions.



PML commands are similar in many ways to PML forms. Forms and commands are a type of global variable. This means that a form or command cannot have the same name as any other global variable or any other form or command. A form or command definition is also the definition of an object, so a form or command cannot have the same name as any other object type.

### 8.1 Defining a Command

A PML COMMAND is a PML 2 object defined in a file with extension .pmlcmd located in a directory under %PMLLIB% and with the same name as the command. It is defined in a similar way to a FORM using the SETUP syntax. A command has a number of properties - **checked**, **enabled**, **value**, **visible** and callback – **execute**, **list**, **refresh**, **state**. Properties are used to get/set the command state and callback may be added to handle command events like the **execute** event which is raised when the attached tool is clicked.

A command is defined using the SETUP syntax as follows:

```

setup command !!avevaPmlTraining
exit

-- Constructor
define method .avevaPmlTraining()
  !this.key      = | AVEVA.DesignGeneral.CommandPMLTraining|
  !this.Execute = |execute|
Endmethod
  
```

The key and command members may be defined within the command setup. It is recommended that the command key and callbacks are defined in the constructor method.

The command key is used to uniquely identify the command when it is registered with the command manager. Similarly, any callbacks should be set before the command is registered

 *Once the command has been registered the key or callbacks cannot be changed).*

The **Execute** callback is called when the attached tool is clicked and optionally can be passed a single STRING argument **!args[0]** from the tool.

In its simplest form a button requires a simple callback to be executed.

```
define method .execute( !args is ARRAY )
show !!form
endmethod
```

## 8.2 Loading and Registering a Command

Once a command has been defined (and added to pml.index) it can be loaded using the PML LOAD syntax.

**PML LOAD COMMAND !!cmdbutton**

which creates the command global instance. The command can then be registered with the Command Manager as follows.

**!!cmdbutton.register()**

The key set in the constructor is **!this**. key and is used to uniquely identify the command. Once registered, it will be available in the application's Customize tool below this key in the Command window.

 *Refer to .NET Customisation Guide, available within the AVEVA™ E3D Design manuals.*

A function **!!loadUserCommands()** in %PMLLIB% is called when entering Model. User PML commands can be loaded and registered in this function. Add-in applications will typically load commands within the application itself in initialisation callbacks.

## 8.3 Attaching a Command to the UI

In the application's Customisation tool (see .NET Customisation Guide) the command will appear allowing it to be attached to one or more tools. Clicking on the attached tool will call the commands **execute()** callback passing any arguments in the **!args[]** array.

## 8.4 Killing a command

Commands can be destroyed in the same way as a form using the KILL command as follows.

**KILL !!cmdbutton**

This will delete the command global instance and, if the command has been registered, unregister the command from the command manager.

 *Commands like forms are not persisted across module switches.*

## Exercise 10 Creation of buttons using command & macro

Perform the following task:

- Create command file **traPMLcommand.pmlcmd** and store it in PMLLIB directory.

```
setup command !!traPMLcommand
exit

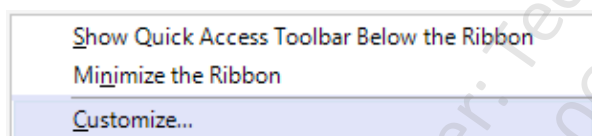
define method . traPMLcommand()
!this.key      = 'AVEVA.DesignGeneral.CommandPMLTraining'
!this.execute  = 'execute'
endmethod

define method .execute( !dummyArgs is ARRAY )
show !!traExampleButtons.pmlfrm
endmethod
```

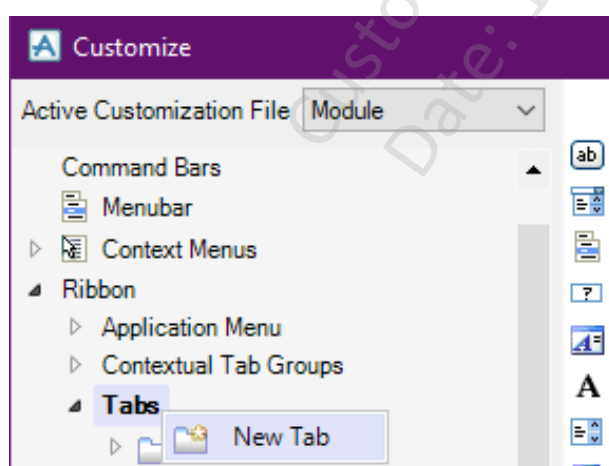
- Use command **PML REHASH ALL**. Make sure that **traExampleVolumeView.pmlfrm** is existing in PMLLIB.
- Load and register command:

**PML LOAD COMMAND !!traPMLcommand**

**!!traPMLcommand.register()**



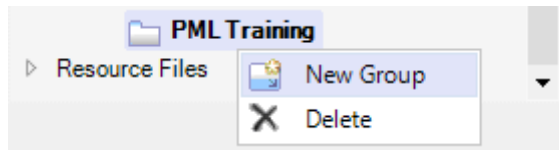
- Right click on the ribbon and select **Customize...** to open the **Customize** form.



- In **Customize** form, select **Module** from **Active Customization file** list.
- Select **Ribbon > Tabs**. From the right click menu select **New Tab** to create a new tab.

ApplicationContext	
Caption	PML Training
Command	
ContextualTabGroup	None
HelpContextID	
KeyTip	
Name	Aveva.Tab15

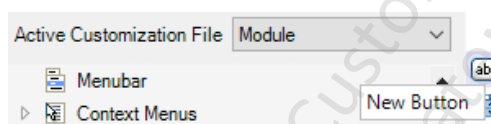
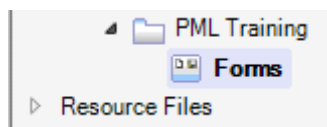
- Select the line titled **Caption** and enter **PML Training**.



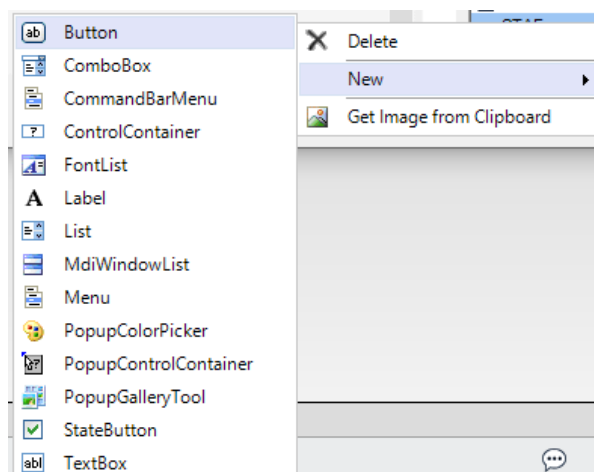
- To create a new group right click on tab **PML Training** and select **New group**.

Caption	Forms
DialogBoxLauncherKey	
HelpContextID	
HideWhenEmpty	False
KeyTip	
LayoutDirection	Vertical
Name	Aveva.PML TrainingGroup1

- Select the line titled **Caption** and enter **Forms**.

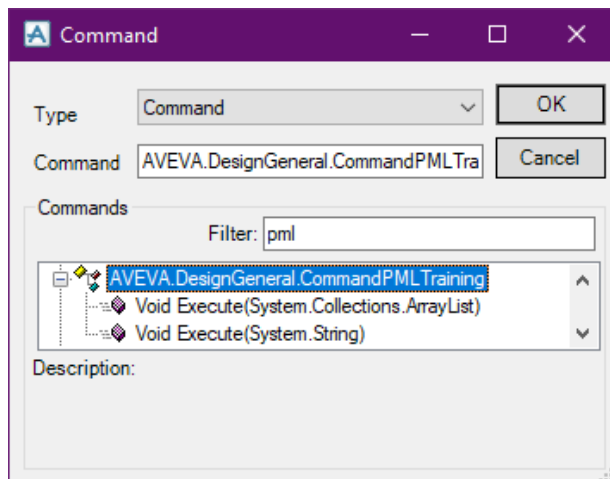


- Click the **New Button** button to create a new button. Or right click inside the middle part of the **Customize** form and select a **New > Button**. Name it as **PML Command**.



ApplicationContext	
Arguments	
Caption	<b>PML Command</b>
Category	
Command	...
DisplayStyle	<b>Default</b>

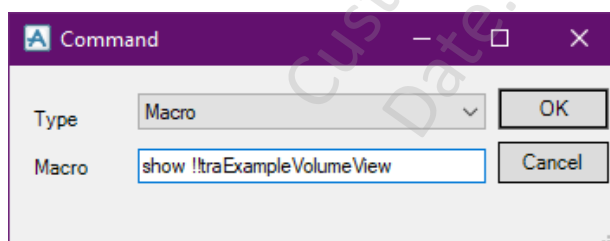
- Select the line titled **Command** and click the small “...” button which appears on the right. This will let us set the command.



- In **Filter** textbox enter **pml** and select **AVEVA.DesignGeneral.CommandPMLTraining**

ApplicationContext	
Arguments	<b>(Collection)</b>
Caption	<b>PML Command</b>
Category	
Command	<b>AVEVA.DesignGeneral.CommandPMLTraining</b>
DisplayStyle	<b>Default</b>
HelpContextID	
Icon	<b>AvevaSharedIcons:ID_BULLET_GREEN</b>
KeyTip	

- Select the line titled **Icon** and pick one of the built-in icons by entering : **AvevaSharedIcons:ID\_BULLET\_GREEN**

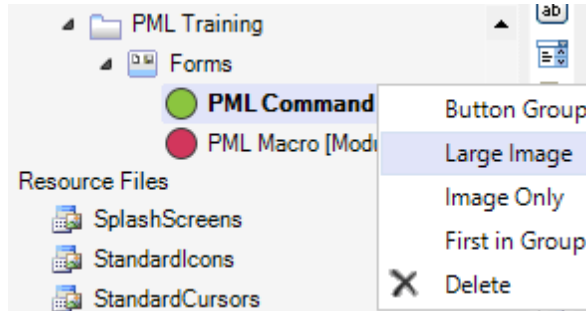


- In the similar way create another button in same group and name it as **PML Macro**.
- Setting the command, select the type **Macro** from **Type** list and enter

**show !!traExampleVolumeView**

ApplicationContext	
Arguments	<b>(Collection)</b>
Caption	<b>PML Macro</b>
Category	
Command	<b>Macro: show !!traExampleVolumeView</b>
DisplayStyle	<b>Default</b>
HelpContextID	
Icon	<b>AvevaSharedIcons:ID_BULLET_RED</b>
KeyTip	

- Select the line titled **Icon** and enter **AvevaSharedIcons:ID\_BULLET\_RED**



- Drag and Drop both new buttons beneath the new **Forms** group to make the association.
- Right click on **PML Command** button and select the **Large Image**.
- Repeat for **PML Macro** button.



- Click on **Apply** button to save the changes and click on **OK** button to close the form.

Customer: Technip Energies  
Date: 19-06-23 - 27-06-23



Customer: Technip Energies  
Date: 19-06-23 - 27-06-23