



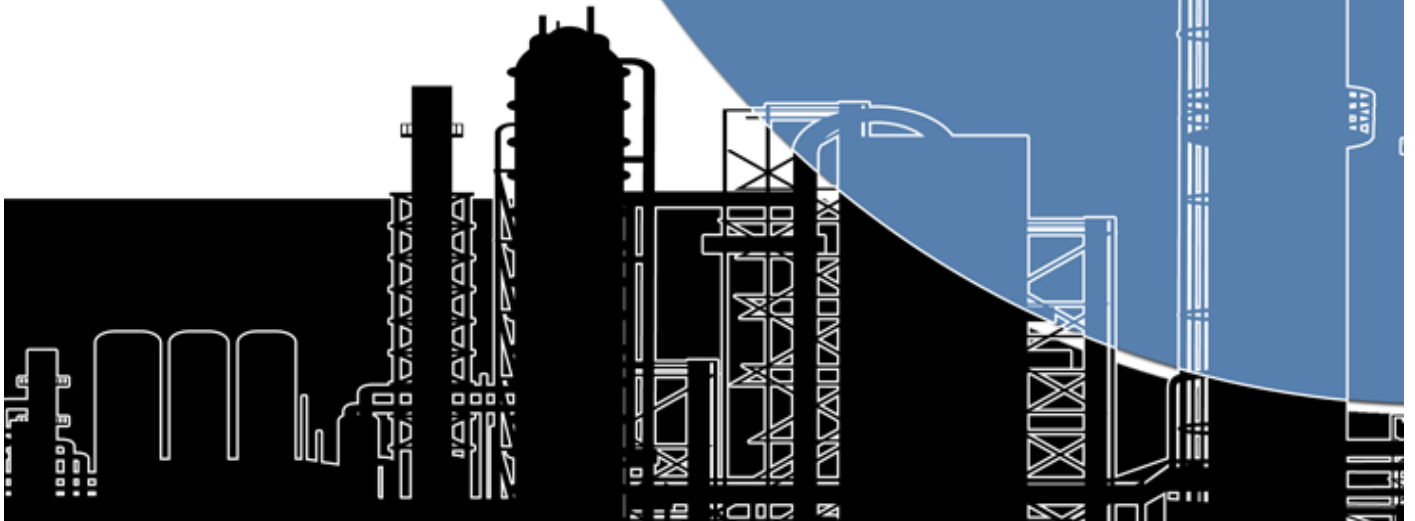
Training Guide

TM-1882

AVEVA Everything3D

PML Applied

Copia para EEA



Copia para EE AA

Revision Log

Date	Revision	Description	Author	Reviewed	Approved

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 tps@aveva.com

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

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

Disclaimer

- 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

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

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

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

The AVEVA software described in this guide is to be installed and operated strictly in accordance with the terms and conditions of the respective software licences, and in accordance with the relevant User Documentation.

Unauthorised or unlicensed use of the software is strictly prohibited.

Copyright 1974 to current year. AVEVA Solutions Limited and its subsidiaries. All rights reserved. AVEVA shall not be liable for any breach or infringement of a third party's intellectual property rights where such breach results from a user's modification of the AVEVA software or associated documentation.

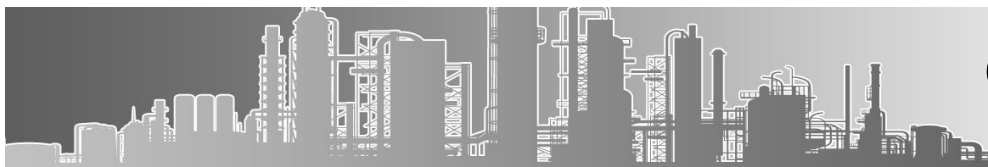
AVEVA Solutions Limited, High Cross, Madingley Road, Cambridge, CB3 0HB, United Kingdom

Trademarks

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

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

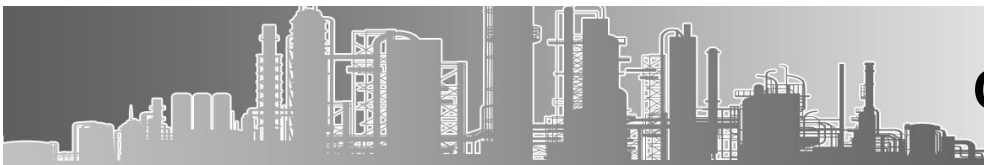
The copyright, trademark rights, or other intellectual property rights in any other product or software, its name or logo belongs to its respective owner.



1	Introduction	7
1.1	Aim	7
1.2	Objectives	7
1.3	Prerequisites	7
1.4	Course Structure	7
1.5	Using this Guide	7
2	PML Overview	9
2.1	PML 1 – String Based Command Syntax	9
2.1.1	Example of a simple command syntax macro	9
2.1.2	Examples of command syntax	9
2.1.3	Syntax graphs	11
2.2	PML 2 – Object-orientated programming	11
2.2.1	Features of PML 2	11
2.2.2	Examples of object-orientated PML	11
2.2.3	Software Customisation Reference Manual	12
2.3	PML Objects	12
2.3.1	Creating variables (instances of objects)	13
2.3.2	Naming conventions	13
2.3.3	Using the members of an object	14
2.3.4	Special objects used in E3D	14
2.4	PML Functions and Methods	14
2.4.1	Arguments of type ANY	15
2.5	PML Forms	15
2.6	PMLUI environment variable	15
2.7	PMLLIB environment variable	16
2.8	Modifications to the PMLUI and PMLLIB	16
	Exercise 1 – Updating the environment variables	17
3	Using PML Objects	19
3.1	What is a PMP object?	19
3.2	Example of PML objects available in E3D	19
3.3	Why are PML objects used?	20
3.4	Defining a data structure	20
3.5	The Model-View-Presenter concept	21
	Worked Example – Creating an object	22
	Exercise 2 – Developing another object	24
4	User Feedback	27
4.1	What is feedback?	27
4.2	Examples of feedback in within E3D	27
4.3	Aid Graphics	28
4.4	Adding colour to the model	29
4.5	Adding feedback to a form	30
	Worked Example – Adding feedback to the form	31
	Exercise 3 – Adding feedback to the model	32
5	Developing new PMP functionality	33
5.1	At the beginning of a development	33
5.2	Stages to the PML development	33
5.3	Using Wireline sketches	34
5.4	What do users always need to do?	35
5.5	What is good user interface design?	35
5.5.1	How easy is the form to use?	35
5.5.2	Have the correct gadgets been used?	36
5.5.3	Can the form be resized and have gadgets been anchored?	36
5.5.4	Have progressive or staged disclosure been considered?	37
5.5.5	How many mouse clicks are required to use the form?	38


5.5.6	Are pop-up alert boxes used?	38
5.5.7	Is there a flow through the form.....	39
5.5.8	How much feedback or assistance is offered?.....	40
5.5.9	How reliable is the form?.....	40
5.5.10	Is the form familiar?	40
5.6	Developing a style	40
5.7	Supplied Exercises.....	41
Exercise 4 – A development for the E3D Designer.....		42
Exercise 5 – A development for the E3D Checker.....		43
Exercise 6 – A development for the E3D Manager		44
Appendix A – Example PML.....		45
Appendix A.1 - Provided form !!c3ex2		45
Appendix A.2 - Example completed worked example for Chapter 3.....		47
Appendix A.3 - Example solution to Exercise 2.....		48
Appendix A.4 - Example completed worked example for Chapter 4.....		50
Appendix A.5 - Example solution to Exercise 3.....		51
Appendix A.6 - Example solution to Exercise 5.....		52
Appendix B – Supporting PML Toolbar		57
Appendix B.1 - The PML Toolbar.....		Error! Bookmark not defined.
Appendix B.2 - Available E3D Colours form		Error! Bookmark not defined.
Appendix B.3 - Available E3D Icons form		Error! Bookmark not defined.
Appendix B.4 – Training Examples form		Error! Bookmark not defined.

Cópia para EEA



1 Introduction

This training guide is designed to give an introduction to the AVEVA Everything3D (E3D) Programming Macro Language. There is no intention to teach software programming but only provide instruction on how to customise E3D using Programmable Macro Language (PML).

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

1.1 Aim

The following points need to be understood by the trainees:

- Understand how PML can be used to customise AVEVA E3D
- Understand how to create Forms
- Understand how to use the built-in objects of AVEVA E3D
- Understand the use of Addins to customise the environment.

1.2 Objectives

At the end of this training, you will have:

- 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

1.3 Prerequisites

The participants must have:

- Completed an AVEVA Basic Design Course and have a familiarity with E3D
- Completed the PML: Macros & Functions Course or have familiarity with PML.

1.4 Course Structure

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

1.5 Using this Guide

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

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

Information the user has to key-in will be in **bold red text**.

 *Additional information will be highlighted*

 *Reference to other documentation will be separate*

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

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

Copia para EE AA

2 PML Overview

Programmable Macro Language (PML) is the customisation language used by AVEVA E3D and it provides a mechanism for users to add their own functionality to the AVEVA E3D software family. This functionality could be as simple as a re-naming macro, or as complex a complete user-defined application (and everything in between).

PML is a coding language specific to AVEVA products based on the command syntax that is used to drive E3D. As the product develops, PML is also improved providing new functionality and bringing it closer to other object-orientated programming languages, while still retaining the powerful command syntax. Although it is one language, there are three distinct parts:

- PML 1 the first version of PML based on command syntax. String based and provides IF statement, loops, variables & error handling
- PML 2 object oriented language extending the ability of PML. Use of functions, objects and methods to process information (also covered in TM-1881 – Form Design)
- PML .NET provides the platform in PML to display and use objects created in other .NET languages (covered in TM-1881 – Form Design)

2.1 PML 1 – String Based Command Syntax

When PML is written as command syntax, E3D processes it as individual lines of command and runs them in sequence - as if they had been typed directly into the command window.

A simple macro is likely to be written completely in command syntax and allows users to re-run popular commands. Saved as an ASCII file, the macro can be run in E3D through the command window (by typing \$m/FILENAME or by dragging and dropping the file).

2.1.1 Example of a simple command syntax macro

The following is an example of a simple macro that can be used to create an Equipment element. As each line is run in order, the same piece of equipment will be created each time the macro is run.

```
NEW EQUIP /ABCD  
NEW BOX  
XLEN 300 YLEN 400 ZLEN 600  
NEW CYL DIA 400 HEI 600  
CONN P1 TO P2 OF PREV
```

Macros can be extended to include IF statements, DO loops, variables and error handling (explained further later in the course) if additional information is typed onto the same line as the call for the macro, then these become input parameters and are available for use within the macro.

i.e. **\$M/BUILDBOX 100 200 300**

In this example, the three numbers (100, 200 & 300) become the three parameters supplied to the macro.

2.1.2 Examples of command syntax

The following are examples of command syntax that can be typed directly into the command window (or used within a macro):

Working with elements and attributes:

- To find out attribute information about the current element, type **Q ATT**
- To create a new element (for example, BOX), type **NEW BOX**

- To find out a specific attribute (for example NAME), type **Q NAME**
- To set a value to an attribute on the current element (for example XLEN), type **XLEN 300**

Manipulating the drawlist in DESIGN:

- To add the current element to the drawlist, type **ADD CE**
- To add a specific element below a piece of equipment, type **ADD ONLY /E1301-S1**
- To remove all the elements from the drawlist, type **REM ALL**

Annotating elements in DESIGN:

- To label the current element with its name, type **MARK CE**
- To label the element /PIPE with its name, type **MARK /100-B-1-B1**
- To remove the label from the current element, type **UNMARK CE**
- To remove all labels, type **UNMARK ALL**
- To mark all branch elements with their names, type **MARK WITH (NAME) ALL BRAN**
- To mark all large valve elements with their specification reference, type **MARK WITH (NAME OF SPREF) ALL VALV WHERE CPAR[1] GT 100**

Using Aid graphics:

- To draw an unnumbered graphic aid line, type **AID LINE E0N0U0 TO E0N1000U1000**
- To draw a sphere aid (aid number 1000), type **AID SPHERE NUM 1000 E0N0U0 DIAM 200**
- To remove all unnumbered graphical aid lines, type **AID CLEAR LINE UNN**
- To remove all number 1000 sphere aids, type **AID CLEAR SPHERE 1000**
- To remove all graphical aids, type **AID CLEAR ALL**

Adding colour to the 3D model:

- To apply the standard enhance colour to the current element, type **ENHANCE CE**
- To apply colour 10 to element /PIPE, type **ENHANCE /PIPE COL 10**
- To remove all colour from the 3D model, type **UNENHANCE ALL**

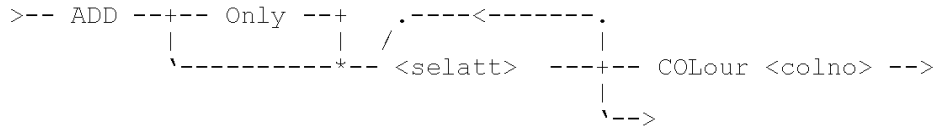
Position and orientation of elements:

- To move the current element 1000mm in a N45E direction, type **MOVE N45E DIST 1000**
- To move the current element E, until it is in line with /BOX, type **MOVE E THRU /BOX**
- To rotate the current element around its origin (pointing up) by 45, type **ROTATE BY 45 ABOUT U**
- To relatively move the current element east by 1000mm, type **BY E 1000**
- To explicitly position the current element, type **AT E0 N1000 U2000**
- To position & orientate the current element based on /EQUIP-N1, type **CONNECT P2 TO P0 OF /EQUIP-N1**

- PREV**

2.1.3 Syntax graphs

Many examples of command syntax are provided in the various reference manuals within the installation folder of AVEVA E3D. For each a syntax graph is provided to show the various combinations of syntax available:



The above syntax graph is for the ADD syntax (used to add elements to the drawlist). From the graph it can be seen that only two words are required (e.g. ADD /PIPE) but others can be included (e.g. ADD ONLY /PIPE1 /PIPE2 COL 3).

The **\$Q** syntax can also tell you the next allowable part of the syntax. For example **ADD \$Q-10** would print the 10 next available words that can follow the command **ADD**.

 More examples of command syntax and the supporting syntax graphs can be found in the relevant reference manuals provided with AVEVA E3D.

2.2 PML 2 – Object-orientated programming

PML 2 is almost an object-oriented language and it provides most features of other Object-Oriented Programming (OOP) languages. Operators and methods are polymorphic and overloading of methods is supported. However, there is no inheritance and no concept of public or private variables.

PML 2 provides for classes of built-in, system-defined and user-defined object types. Objects have members (their own objects) and methods (their own functions). All PML variables instances of (1) built-in, (2) system-defined or (3) user-defined objects.

Through the use of method concatenation, it is possible to achieve multi-operations in a single line of code. This means that PML 2 methods are typically shorter and easier to read than the PML 1 equivalent. While most PML 1 macros will still run within E3D, PML 2 brings many new features that were previously unavailable. If used together with command syntax it must be expressed as a string before use.

2.2.1 Features of PML 2

The main features of PML 2 are:

- Available built-in variable types - STRING, REAL, BOOLEAN, ARRAY
- Built in Methods for commonly used actions
- Global Functions supersede old style macros
- User Defined Object Types
- PML Search Path (PMLLIB)
- Dynamic Loading of Forms, Functions and Objects (no need for synonyms)

2.2.2 Examples of object-orientated PML

Declaration of objects (variables):

- To declare a local variable as a REAL 3, type **!realVariable = 3**

- To declare a global BOOLEAN variable as false, type **!!booleanVariable = FALSE**
- To set the third value in an ARRAY as Fred, type **!arrayVariable[3] = |Fred|**
- To declare a variable as an empty position object, type **!position = object position()**

Finding out information about objects (variables)

- To find out a variable, type **q var !exampleObject**
- To find out the object type of a variable, type **q var !exampleObject.objectType()**
- To find out what members an object has, type **q var !exampleObject.attributes()**

Methods available on objects:

- To find out what methods are available on an object, type **q var !exampleObject.methods()**
- To find out if an object has been given a value, type **q var !exampleObject.set()**
- To find out how large an ARRAY variable is, type **q var !arrayVariable.size()**
- To find out how long a STRING variable is, type **q var !stringVariable.length()**

Clearing an object

- To empty an object, type **!exampleObject.clear()**
- To delete an object that is no longer needed, type **!exampleObject.delete()**

2.2.3 Software Customisation Reference Manual

Many examples of the major objects available in AVEVA E3D are provided in the Software Customisation Reference Manual within the products installation folder. For each object, the members and methods are listed and explained.

- For each of the members the name, type and purpose is provided
- For each method the name, argument types, returned object type and purpose are provided.

 *Refer to the Software Customisation Reference Manual for more information about the major objects.*

2.3 PML Objects

Every variable defined with PML has an object type. This type is set when the variable is defined and is fixed for the life of the variable. When assigning a value to a variable, the object type needs to be considered, otherwise an error may occur. There are three groups of object types available:

- Built-in (e.g. String, Real, Boolean and Array)
- System-defined (e.g. Position, Orientation)
- User-defined
- Refer to the Software Customisation Reference Manual for more examples of System-defined objects

When a variable is declared as a specific object type, it is given all the members (attributes) and methods of the object definition. This means standard groupings can be setup to store data and that code repetition can be avoided.

A user-defined object provides an opportunity to group data together for a specific purpose. Once grouped as an object, it can be assigned to a variable and used as any other object. The following are examples of two user-defined objects:

```
define object FACTORY
  member .name is STRING
  member .workers is REAL
  member .output is REAL
endobject

define object PRODUCT
  member .productCode is STRING
  member .total is REAL
  member .site is FACTORY
endobject
```

These objects would be defined as two separate object files (.pmlobj) and loaded into E3D. You will notice that the object PRODUCT is able to have a member which is another user-defined object. This means that the PRODUCT object has access to all the members and methods of a FACTORY object.

2.3.1 Creating variables (instances of objects)

There are two types of variables in PML (1) Local and (2) Global

The difference between the two is that global variables last for the whole E3D session and can be referenced directly from other PML routines. Local variables are only available within the routine which defined them. The variables are declared with a single '!' for local and a double '!!' for global

- !localVariable – a local variable
- !!globalVariable – a global variable

A variable object-type can be implied from the assigned value:

!name = Fred 	To create a LOCAL, STRING variable
!!answer = 42	To create a GLOBAL, REAL variable
!!flag = TRUE	To create a GLOBAL, BOOLEAN variable

It is also possible to define a variable as an object-type without an initial value. The value is therefore UNSET

!name = STRING()	To create a LOCAL, UNSET STRING variable
!!answer = REAL()	To create a GLOBAL, UNSET REAL variable
!array = BOOLEAN()	To create a LOCAL, UNSET ARRAY variable

2.3.2 Naming conventions

It is common practice to follow a naming convention when defining objects and variables. Using upper case for the name of the object type and mixing upper and lower case for the variable is a good practice to follow:

For example, the type might be WORKERS while the name of the variable might be numberOfWorkers

- i** Notice that to make the variable name more meaningful, full words are used with a mixture of upper and lower case letters to make it readable.

Variable names should not start with a number or contain any spaces or full stops (full stops are used in PML2 to indicate methods and members – explained later)

- i** AVEVA uses a CD prefix on most of its global variables. Newer functionality does not use a prefix so all new PML must be checked for name clashes. Using your own prefix could help avoid this.

2.3.3 Using the members of an object

When a variable is declared as an object-type, it is also given all the objects members and methods. For example, to a local variable as a factory object:

!newPlant = object FACTORY()

After being declared as above (using the OBJECT keyword and ending with a double bracket, the local variable !newPlant is now a FACTORY object and has the same members as the FACTORY object. These members are available to store information and can be assigned values in the following way:

!newPlant.name = |ProcessA|

!newPlant.workers = 451

!newPlant.output = 2000

① Notice the use of a dot between the variable and its member. This works as long as the word after the dot is a valid member of the variable object-type.

Once assigned a value, this value is available for use and can be recalled. For example:

!numberOfWorkers = !newPlant.workers

This creates a new local real variable and assigns it the value 451

2.3.4 Special objects used in E3D

In a standard E3D DESIGN session, there are number of specialised objects which are loaded and used by standard product. These objects should not be deleted or overwritten, but are available for use. The particularly useful objects are:

- **!!ICE** - a global DBREF object which tracks and represents the current element
- **!!ERROR** - a global ERROR object which holds information about the last error
- **!!PML** - used to obtain file path strings through the .getPathName() method
- **!!ALERT** - used to provide popup feedback to users
- **!!AIDNUMBERS** - used to manage aid graphics
- **!!APPDESMAN** - the form which represents the main DESIGN interface

2.4 PML Functions and Methods

Functions and methods provide the actions of PML. When called, a function or method will run through the lines of PML it contains in order – just like a PML 1 style macro. Functions and methods may be passed arguments and even return values. A function which does not return a value is typically referred to as a PML Procedure.

A function and a method are written in the same style, the only difference is where the definition is stored and how it is called. A function is a global method (stored in its own file) and can be called directly on the command line (e.g. call !!exampleFunction()) while a method is local to the object it is defined within (e.g. !exampleObject.exampleMethod())

Arguments become local variables within the function/method and the object-types need to be declared within the definition. The returned object-type is also defined. For example:

```
define function !!area( !length is REAL, !width is REAL) is REAL
    !area = !length * !width
    return !area
endfunction
```

In this example, the function !!area is expecting two real arguments. The two arguments are expressed as local variables which are multiplied together to calculate the local variable !area. Using the **return** keyword,

the variable !area is then returned. If the function was called in following way, the variable !area will have a value of 2400:

!area = !!area(12, 200)

 *It is now common practice to write all macros as a functions or procedures*

2.4.1 Arguments of type ANY

When defining an argument within a method or function, you may declare it as an ANY object. This means that the argument can be of any object-type, allowing any variable to be supplied to the function. As no argument check is carried out, the function needs to be robust enough to cope with any argument. For this reason, ANY should be used with special consideration:

```
define function !!argumentType(!argument is ANY) is STRING
    !type = !argument.objecttype()
    return !type
endfunction
```

2.5 PML Forms

For users of E3D, the concept of a form should already be familiar. In terms of PML, a form is a global variable (for example, !exampleForm). A form is capable of owning members and methods (like any other object) but there are a set of predefined members which can be used to put gadgets on the form (buttons, lists, options etc). These gadgets are objects in their own right and can be given callbacks, which can activate a standard command or call a function or run a method defined within the form.

```
setup form !!nameCE
    !this.formTitle = |Name CE|
    button .button |Print Name Of CE| call ||!this.print()|
exit

define method .print()
    !name = !!ce.flnn
    $p Name of Current Element = $!name
endmethod
```

The above example is the definition of a form called **nameCE**. Saved within one file (.pmlfrm), it defines two form members (a predefined member for the title and a new button gadget) and a form method.

!this is a special local variable and using it replaces the need to reference the owning object directly. For example, to call the method .print() from within the form, the call is !this.print() and to call the method from anywhere else, it's !!nameCE.print()

2.6 PMLUI environment variable

All PML1 Macros are typically stored a directory structure pointed at by the E3D environment variable PMLUI. The PMLUI environment variable is set in the .bat file that is used to load E3D (typically within evvars.bat). To set the environment variable, the following line is needed in the .bat file:

set PMLUI= C:\AVEVA\Plant\E3D1.1.0\PMLUI

The purpose of an environment variable is to reduce the length of the command used to call a macro. This means that \$M/%PMLUI%/DES\PIPE\MPIPE can be typed instead of the full file path.

In standard product, this process is shortened further as all PML1 macros and forms are called using synonyms. For example, the macros associated with piping are called using the synonym CALLP:

\$\$ CALLP=\$M/%PMLUI%/DES/PIPE/\$s1
CALLP MPIPE

 *If all synonyms are killed then E3D will cease to function as normal*

2.7 PMLLIB environment variable

All PML 2 objects and functions are in a directory structure pointed at by the E3D environment variable PMLLIB. As with the PMLUI variable, this is set in the .bat file which runs E3D. To set the environment variable, the following line is needed in the .bat file:

set PMLLIB= C:\AVEVA\Plant\E3D1.1.0\pmllib

The PMLLIB environment variable differs because it can be searched dynamically. This means that the individual files do not need to be referenced directly and can be called by name. This is possible because E3D compiles a pml.index file which sits in the PMLLIB folder and provides the path to all suitable files within it. For example, to load and show a form the command is show !!exampleForm, there is no need to reference the file path at all.

If a new file is created or the PMLLIB variable is changed then there is a need to update the pml.index file. This can be done by typing PML REHASH onto the command window. If there are multiple paths that need updating, type PML REHASH ALL

If a PML object has already been loaded into E3D, but the file definition has changed then the object needs to be killed and reloaded before the changes can be seen. This can be done by typing either pml reload form !!exampleForm or pml reload object EXAMPLEOBJECT

Although not necessary, it is good practice to organise the files below the PMLLIB folder. A standard E3D installation organises the files based on application and then on forms, functions, objects. This is normally a good starting point for organising customisation.

2.8 Modifications to the PMLUI and PMLLIB

As PML is developed, it is normal practice to store it in a parallel file structure outside the standard installation directory. This parallel file structure can then be referenced by the .bat files. There are a couple of reasons why this is a good idea:

- It keeps the customisation separate from the standard install, so as subsequent versions are installed there will not be a need to move the customised files.
- If any standard files are modified then the originals are still available if required
- If the customisation fails, the standard installation is available to go back to
- Many local E3D installations can reference the same customisation from a network address

 *Any changes to AVEVA standard product may cause E3D to function inappropriately*


It is possible to get E3D to look in different places for PML and this is done by setting the environment variables to multiple paths. This allows the standard install to be kept separate from user and company customisation. This is done by updating the variable to include another path, for example:

set PMLUI=C:\temp\pmlui %pmlui%
set PMLLIB=C:\temp\pmlib %pmlib%

This will put the additional file path in front of the standard (which would have already been defined in the .bat file). This change can also be checked in a E3D session by typing q evar PMLUI or q evar PMLLIB onto the command window.

Exercise 1 – Updating the environment variables

- 1 Extract the provided files into the folder **C:\AVEVA\Plant\Training**
- 2 Open the file in a suitable text editor
Add the following lines to **custom_evars.bat**
set PMLUI= C:\AVEVA\Plant\Training\pmlui %pmlui%
set PMLLIB= C:\AVEVA\Plant\Training\pmllib %pmllib%
- 3 Load **E3D Model** module.

 *The choice of project and MDB will depend on what is available in the E3D setup.
This should be discussed with your trainer.*
- 4 Confirm that the search paths have been correctly updated by typing **q evar PMLUI**
or **q evar PMLLIB** into the command window.

Cópia para EEA

Copia para EE AA

3 Using PML Objects

3.1 What is a PML object?

A PML object is a grouping of common information and/or methods for easier use and reference. The two major aspects of a PML object are:

- Object Members – the properties of the object, used to hold its definition
- Object Methods – the actions available on the object

An object can be used to represent anything; consider the computer being used to complete this training guide:

Object Computer



Members:

- Processor speed
- Size of HDD
- Name of manufacturer
- List of peripherals

Methods:

- Turn the computer on
- Moving the mouse
- Typing on the keyboard
- Format disk

3.2 Example of PML objects available in E3D

PML objects are available across E3D to standardise code. These objects include:

GPHLINE – an object to represent a graphical line

- Members – tag, line, colour, style
- Methods - .definition(), draw(), edit()

GPHDRAWLISTS – an object to represent and control the E3D drawlists

- Members – drawlists, debug
- Methods – .attachView(), createDrawlist(), .list(), .saveall()

VOLUME – an object to represent the volume occupied by an element

- Members – to, from
- Methods - .box(), .cable(), .volume()

To investigate these objects, type the following into the **Command Window**:

```
!gphline = object GPHLINE()
q var !gphline.attributes()
q var !!gphline.methods()
```

```
q var !!gphdrawlists.attributes()
q var !!gphdrawlists.methods()
```

```
!volume = object VOLUME(!ce)
q var !volume.attributes()
q var !volume.methods()
```

These objects are defined by a .pmlobj file, stored within the %PMLLIB% environment variable. Browse to find the files and look through their definition.

① *PML files can be located by passing the name of the object to the PMLDEFINITION function e.g. `!!pmldefinition(jgphline)`*

3.3 Why are PML objects used?

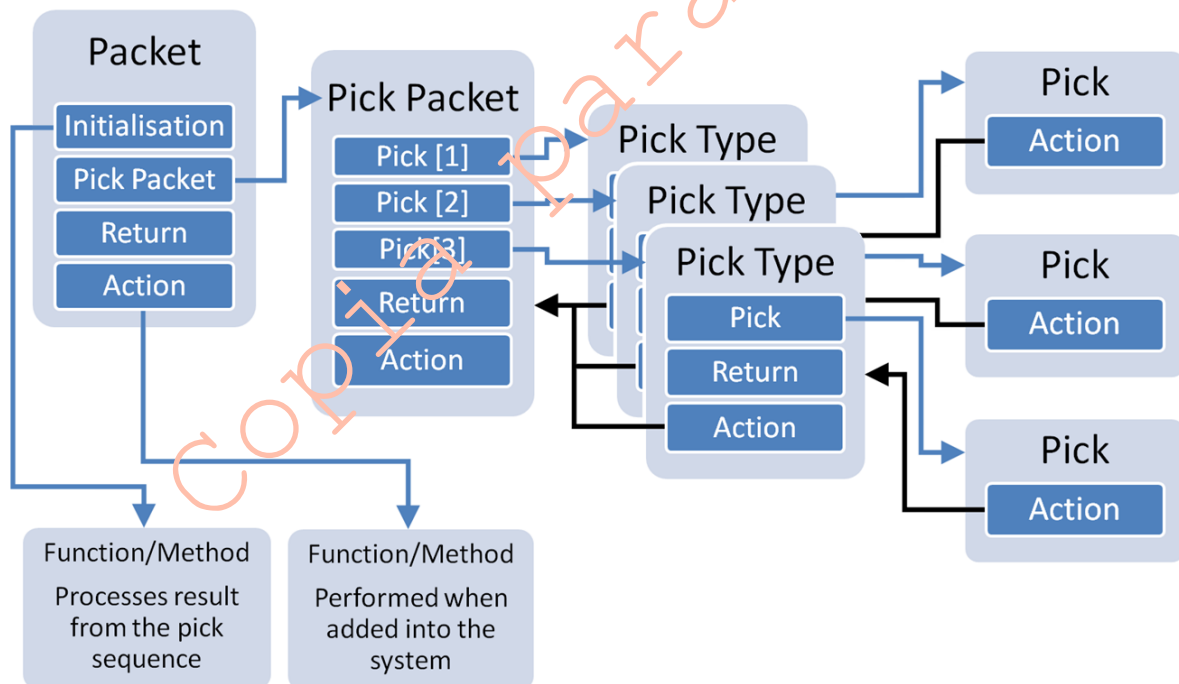
By using PML objects, common code can be isolated, allowing it to be shared and reused. This reuse means that task-specific code can be reduced, relying on the generic object code to perform the bulk of the effort. This will ultimately create code that is easier to manage and maintain.

PML objects can also be used to solve complex problems by representing actual objects in the real world. Consider how a complex problem, such as particle motion could be solved by modelling each particle as an object.

3.4 Defining a data structure

PML objects provide an opportunity to define a data structure that can be used to store information. Instead of using generic ARRAY objects to store information, bespoke PML objects can store the same information in a more defined and predictable way.

As an example, consider the packet of information which is used to configure a graphical pick within E3D. Event Driven Graphics (EDG) are controlled by an EDGPACKET object, defining the required pick. The detail of the EDGPACKET object is fully defined by a series of PML objects, each used to control and configure different parts of the pick.



To investigate the object structure, type the following into the **Command Window**:

```

!packet = object EDGPACKET()
q var !packet
q var !packet.methods()

q var !packet.pickpacket
q var !packet.pickpacket.methods()
    
```

```
!packet.pickpacket.picks[1] = object EDGPICK()  
q var !packet.pickpacket.picks[1].attributes()  
q var !packet.pickpacket.picks[1].methods()
```

3.5 The Model-View-Presenter concept

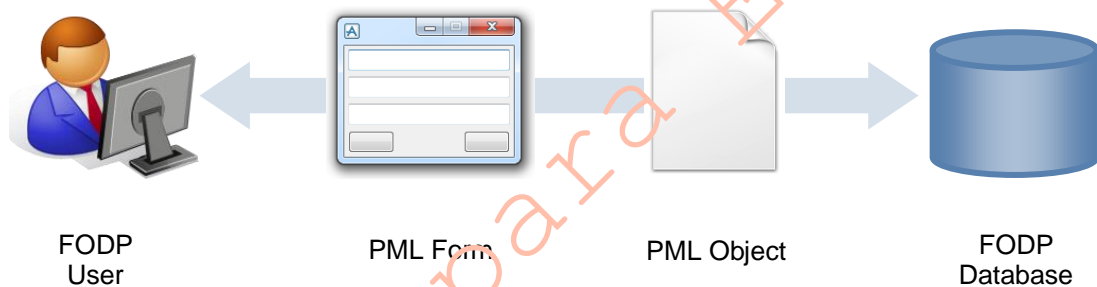
When developing a user interface, it can be considered a good idea to separate the supporting data model and methods from the main form. One such technique is Model-View-Presenter, the three main components of which are:

- Model – the stored data or data model that will be interacted with by the user interface
- View – provides the data stored in the model and passes it the interaction with the presenter
- Presenter – receives data and displays data in the model and displays it in the view.

This technique is widely used in .NET languages when designing a user interface, allowing code to more flexible and easier to maintain. It also means the model and view can be tested separately.

 *Microsoft use MVP in their .NET examples for user interfaces*

Consider the following arrangement. It is a PML application of MVP principle, allowing a user to modify a E3D element via a PML object:



In this arrangement, the E3D user interacts with the form to enter the required changes. The form methods supply the entered information to the PML object, which processes the information and updates the E3D element. Once updated, the new information is held by the object and sent back to the form to be displayed to the user.

It allows the PML object to be created and tested separately from the PML form. It also means that the PML object can be designed to work with multiple forms.

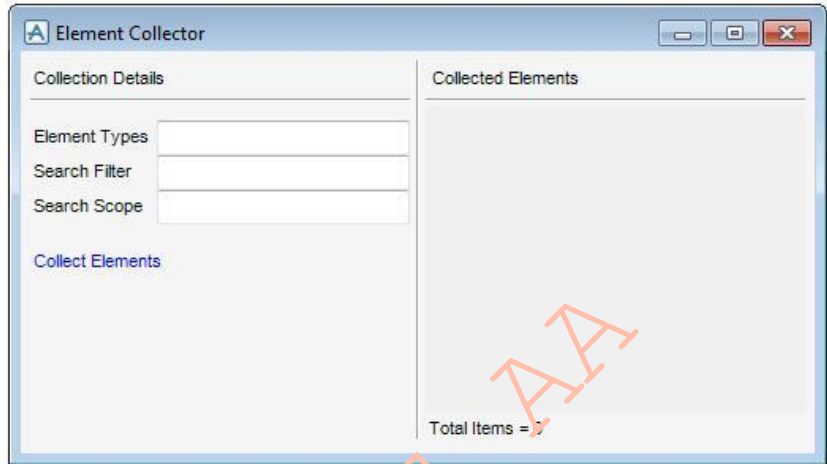
Worked Example – Creating an object

1. You have been given a PML form which collects and displays collected elements.

Show the form by typing

show !!c3ex2

Investigate the functionality of the form by typing in some search conditions and collecting some elements. Browse to find the PML form file and open it up.



2. The form performs the element collection within the form methods. An element collection is a generic operation that could be used by multiple forms. Lets move the collection code to a separate object.

Create a new PML object file called `elementcollection.pmlobj` and add the following definition:

```
define object elementcollection
  member .types      is ARRAY
  member .filter     is EXPRESSION
  member .scope      is DBREF
  member .elements  is ARRAY
endobject
```

Save the file and type `PML REHASH ALL` into the command window to make the object available

3. Move the `.collect()` method from the form to the object and update it to refer to the object members.

For example,

!this.elementTypes.val becomes **!this.elementTypes**

 Notice how the object member types are specific

Update the method so that it returns an array of the collected elements and also stores them as part of the object. Remove the `!this.fillGrid()` call.

Test the object on the command line by typing the following:

```
!elementCollection = object ELEMENTCOLLECTION()
!elementCollection.elementTypes[1] = !EQUI
!elementCollection.scope = !!ce
q var !elementCollection.collect()
```

4. Add an **ELEMENTCOLLECTION** object to the form as a form member and call it **.collection**

Add the following callbacks to the three textboxes on the PML form:

```
!this.types.callback = !this.collection.types = !this.types.val.split()  
!this.filter.callback = !this.collection.filter = object EXPRESSION(!this.filter.val)  
!this.scope.callback = !this.collection.scope = object DBREF(!this.scope.val)
```

Update the callback on the collect button to be the following:

```
!this.collect.callback = !this.fillGrid()
```

Update the .fillGrid() method to call the collect method on the ELEMENTCOLLECTION object by including the following:

```
!this.elements = !this.collection.collect()
```

As it is no longer need, the .collect() method can now be removed

5. Test the form with a range of collections and filters. How does the form react? How can the methods be made more robust to prevent any errors?
-

Cópia para EEA

Exercise 2 – Developing another object

If a user works with the same form regularly, they may wish for the form to remember the last values they entered. Within the same E3D session, this is already taken care of (as long as the form isn't reloaded), but not between sessions.

To give forms the ability to remember values, develop a PML object called **FORMVALUESTORE**

This object should have the following members:

```
.form is FORM  
.gadgets is ARRAY  
.values is ARRAY  
.file is FILE
```

The object should have the following methods:

```
.formValueStore(!form is FORM)  
.saveValues()  
.loadValues()
```

The constructor method receives an instance of the form so it knows which form to read from and load to.

The `.saveValues()` method needs to do the following:

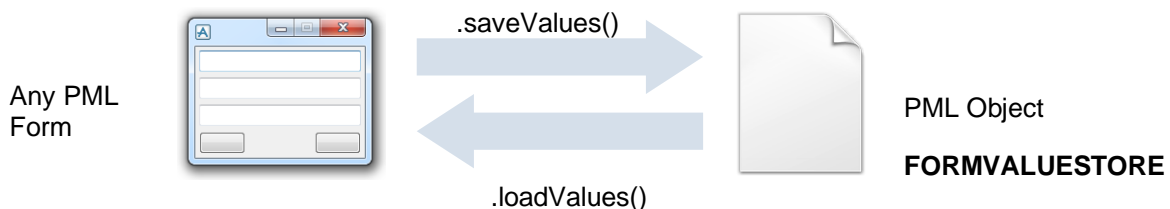
- Loop through the members of the form and look for gadgets.
- If it's a gadget, does it have a `.val` member
- If it does, store the value and gadget in the object members
- Save the form name, gadget names and values to a settings file (using `%E3DUSER%`)

The `.loadValues()` method needs to do the following:

- Look for the settings file associated with the form
- If found, load the gadget information to the object
- Loop through the stored information and place it into the form

Consider when the form should call the methods on the object? Would it be when it is loaded or unloaded? Shown or hidden?

Test the object to ensure it can work between sessions.



Copia para EE AA

Copia para EE AA

4 User Feedback

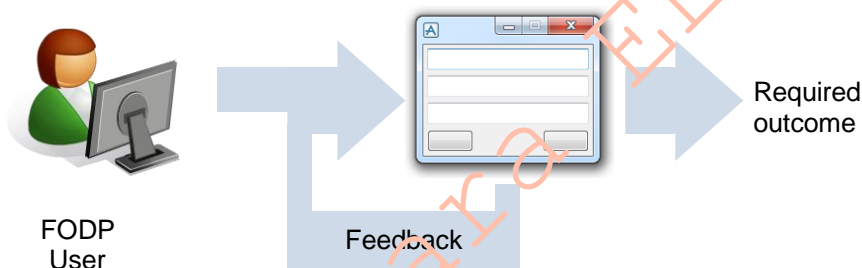
4.1 What is feedback?

Feedback is the mechanism of providing information about an event which is occurring (or has occurred) so to influence any future occurrences. Sometimes described as a “Feedback Loop”, it improve the user experience of any PML customisation.

Anything which provides the users with information about an action they have just completed can be classed as feedback. This might include:

- Changing the colours used (either on the form or in the model)
- Providing a preview with aid graphics
- Manipulation of the User Interface (visibility or state of a gadget)

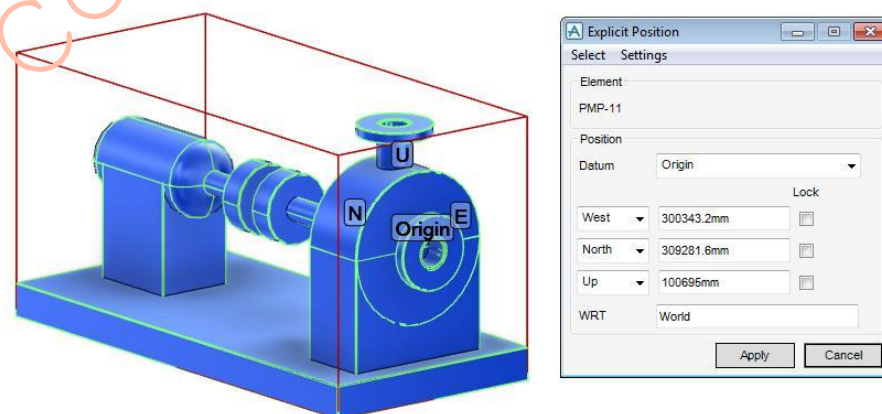
Consider the following situation; feedback reinforces and enhances the use of a PML form. The more feedback provided, the more likely the form will be used correctly and successfully.



4.2 Examples of feedback in within E3D

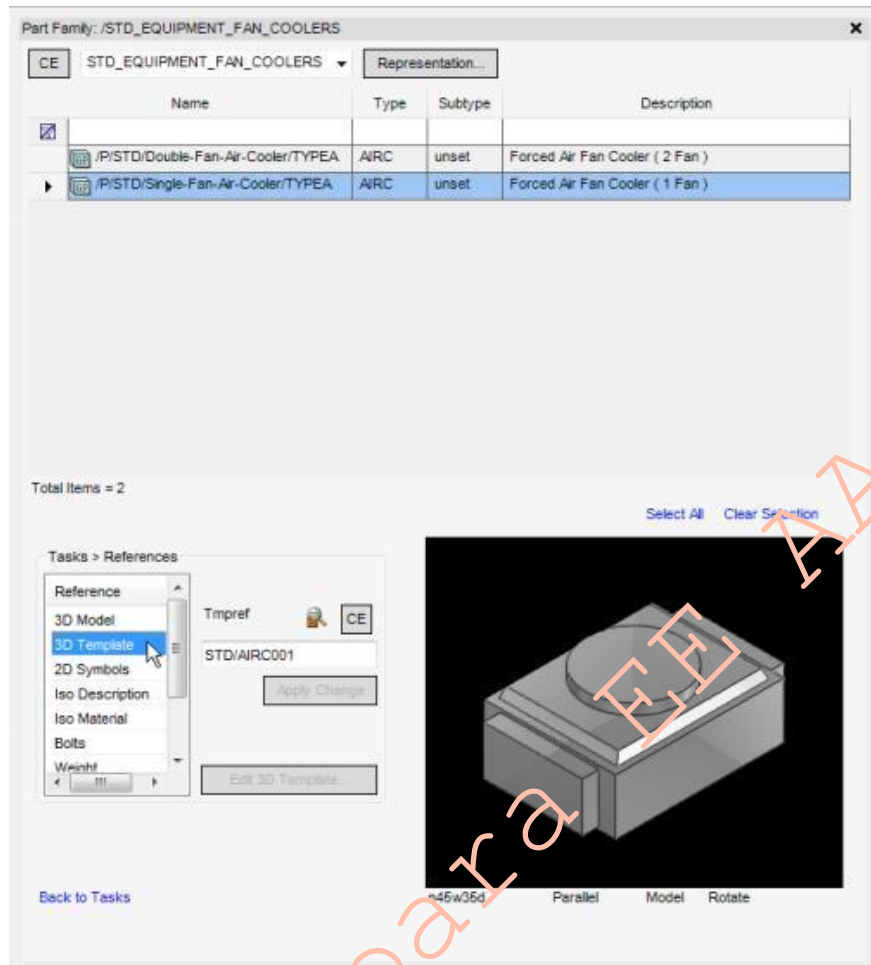
Working with E3D, you will encounter feedback across the product. The following are some typical examples taken from across the product set:

On the **Equipment** tab, in the **Common** group, click the **Position Explicit** button to display the **Explicit Position** form



The form locates the position using a set of axis and draws the volume of the current element. This aid graphic information updates to show a preview of where the element will move to if Apply is clicked. This feedback allows users to visually confirm the move, before committing to it.

In Paragon, Modify > Part Family...



A GPART element in the catalogue acts as a pointer element, linking the catalogue or template item to the specification. The form uses a local 3D view to preview of the item being referenced by the GPART. This means the user doesn't have to rely on the name of the item to understand what it is.

4.3 Aid Graphics

Aid graphics provide an opportunity to preview changes to a user before any are made. Aid graphics are temporary items that can be created and removed with PML commands or objects.

For example, to create an aid line between two points with a PML command:

AID LINE E0N0U0 TO E1000N0U0

When an aid graphic is created it can be assigned a number. This helps identify it, allowing it to be altered or removed. For example, to create an number dashed aid line with a PML command:

AID LINE NUM 100 E0N0U0 TO E1000N0U0 DASHED

To remove the numbered line aid graphics, the following command can be used:


AID CLEAR LINE 100

To remove any unnumbered line aid graphics, type the following:

AID CLEAR LINE UNN

To remove all aid graphics

AID CLEAR ALL

 *Using this command will remove all aid graphics from the screen. If any aid graphics are being used by another form, these will also be removed*

 *For more information about the available commands and associated syntax graphics, refer to the Design Reference Manual : General Commands*

To create a line aid graphic using a PML object, use the following

```
!poss = object POSITION(|E 0 N 0 U 0 WRT /*|)  
!pose = object POSITION(|E 1000 N 0 U 0 WRT /*|)  
!line = !poss.line(!pose)  
!line.draw(100, 1, 1)
```

The three REAL arguments to the .draw() method on a LINE object are:

1. The aid graphic number
2. The aid graphic style
 - 1 = solid
 - 2 = dashed
 - 3 = dotted
 - 4 = chained
3. The aid graphic colour

Once drawn, the aid graphic can be used with the PML command **AID CLEAR LINE 100**

 *Other geometric PML objects have a .draw() method. Refer to the Software Customisation Reference Manual for more information*

4.4 Adding colour to the model

If PML interacts with the model, applying colour is a good way of displaying information, such as:

- Which elements have been selected
- Which elements have a certain attribute value (e.g. status)
- Which elements have been changed or affected by the PML

For example, to add colour to an element with a PML command, type the following:

ENHANCE CE COL 10

Where CE is a selection (i.e. current element) and 10 is the required colour (i.e. white)

Once an element has been highlighted, the colour can be removed, type the following:

UNENHANCE CE

To remove all highlight colours, type:

UNENHANCE ALL

 *Running this command will remove all highlight, even if not created by the PML*

To highlight an element using PML objects, there are methods available on the global drawlist object, use the following:

```
!colour = object COLOUR(white)  
!!gphdrawlists.drawlists[1].highlight(!ce, !colour.code)
```

i Notice how a colour object can be used to work out what number a colour has.

If more drawlists are created and registered with the global drawlist object, they will be available for use. In this case, the first drawlist is the drawlist to the main 3D view.

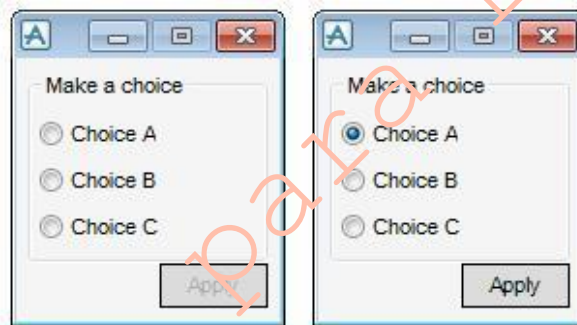
To remove the colour, use the following:

```
!!gphdrawlists.drawlists[1].unhighlight(!ce)
```

i If an element is highlighted by PML, it is a good idea to store the element reference. This will make it easier to remove the highlight once the PML has finished.

4.5 Adding feedback to a form

Feedback on a form can guide a user through the functionality. Rather than showing an alert box to instruct the user of an issue, controlling active gadgets or displaying messages on a form may support the the workflow better.



Worked Example – Adding feedback to the form

1. Lets develop the Element Collector form to enhance the feedback it provides to users.

Show the form by typing **show !!c3ex2**.

Add a space for messages at the bottom left of the form. This will consist of two paragraph gadgets and a line.

2. Consider what parts of the workflow are important to ensure the form does not error. Lets add a method that checks if the collection criteria has been provided.

Add a method to the form called **.checkCriteria()** and set it as the callback on the three criteria text boxes. Move the current callbacks to the **.fillGrid()** method. Also add it to the form initialisation callback.

3. The first check to make it that at least one of the text box has a value set. Update the method:

```
define method .checkCriteria()
    !this.collect.active = TRUE
    !this.message.val = ""

    if !this.types.val.eq("") .and(!this.filter.val.eq("")) .and(!this.scope.val.eq("")) then
        !this.collect.active = FALSE
        !this.message.val = "Please enter some collection criteria"
        return
    endif
endmethod
```

This method deactivates the Collect Elements button if no value has been entered and provides feedback. Show the form and test the feedback.

4. Another check might be to confirm that the entered criteria is valid. For example, has the user entered valid element types. Add the following code to the end of the **.checkCriteria()** method:

```
!errors = ""
do !type values !this.types.val.split()
    !elementType = object ELEMENTTYPE(!type)
    !owners = !elementType.parentTypes()
    if !owners.unset() then
        !errors = !errors & "" & !type
    endif
enddo

if !errors.length().gt(0) then
    !this.collect.active = FALSE
    !this.message.val = !errors & " - not valid element types"
    return
endif
```

Notice how the ELEMENTTYPE object is used to check if an element is valid. The method builds a string of the invalid element types and displays them to the user.

5. Test the form further and consider how the method can be extended to check the filter and scopes are valid. What other messages could be given to the user (e.g. if no elements are found)?

Exercise 3 – Adding feedback to the model

When elements are collected the list displays the names. To demonstrate these elements to the user, add a method to the form that highlights collected elements in the drawlist. The elements in the main drawlist can be obtained with the following:

```
!members = !!gphdrawlists.drawlists[1].members()
```

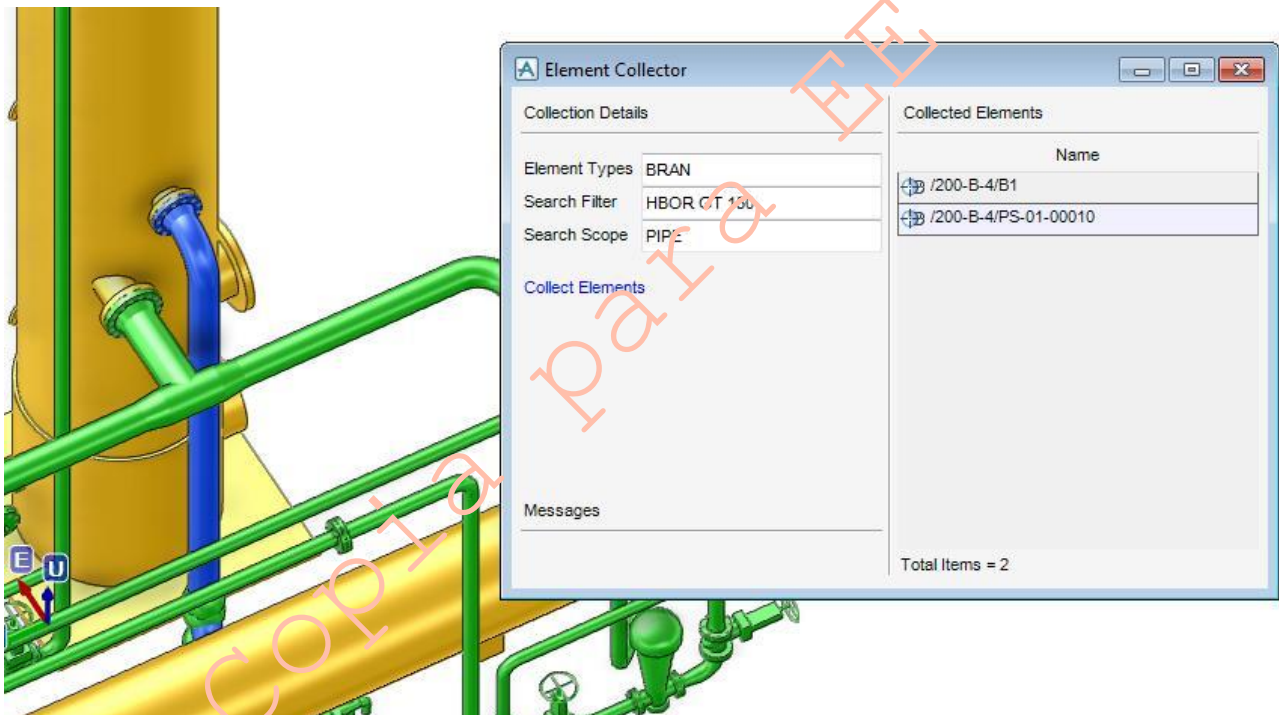
Add an ARRAY member to the form to store the elements that are highlighted

Before highlighting any elements, the method should check if any elements have been previously highlighted and unhighlight them.

This method can be extended by also labelling the highlighted elements with the name.

Test the form and method to ensure the highlighting works correctly.

 Remember, if a form is closed it should remove any highlights it has made





5 Developing new PML functionality

At the beginning any new PML development, it is important to understand the reasons and requirements behind it. The functionality will ultimately be for a specific user to aid their use of E3D. Identifying the user and their requirements first will help steer the development, ensuring it addresses the original need.

5.1 At the beginning of a development...

Before typing any lines of PML, answer the following questions:

- Who?
Who is the ultimate user of the development and whose requirements are being addressed?
- What?
What are the requirements of the end user and what will they expect the new functionality to provide? Ultimately, what are they trying to achieve?
- Why
Why is the functionality needed? What benefits will it bring? Identifying this information will help define the solution and will help create the test case scenario.

i *The final user of the functionality is the most important person. This should be considered at every stage of the development. Their involvement is important to ensure a successful development.*

5.2 Stages to the PML development

A typical PML development should pass through the following stages:

1. Address the Who, What and Why of the proposed new development.
2. Document this information and supply it to the final user for comment. Make the user aware of what functionality will be provided and how this functionality addresses their requirements.
3. Begin to develop a solution by making some initial decisions.
 - a) Is a form required?
 - i) If a form is required, what will it display?
 - ii) Produce a mock-up sketch (wire-frame) of the proposed layout
 - b) Do any new PML objects need developing?
 - i) What will the objects be required to do?
 - c) Will the PML need do any data manipulation, calculation or processing?
 - i) Develop a flow diagram showing the how the PML will function
 - d) What E3D modules will need to functionality?
 - i) This will dictate where the PML is stored and what information is available to the functionality at the time of use
4. Supply the user with more detail of how the development will work and how it will look and feel. Let the users provide feedback on the layout to ensure it is as they expect
5. Develop the prototype solution and setup an environment with a scenario for the user to test on. Take away and incorporate any additional feedback from the user. Finalise the solution.

- 6) Document, deploy and distribute the new development to the users

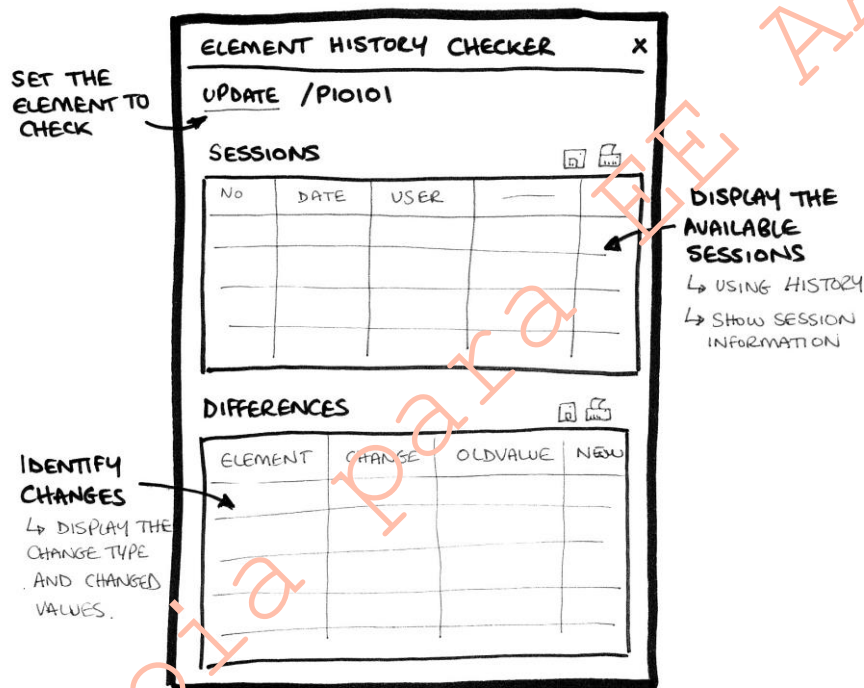
Getting the user to feed into the development process helps guide it and keeps it focussed on their requirements. Visual mock-ups and test environments allow the user to experience what will be delivered and should prevent any user-issues at the end. Instead of trying to guess what a user wants, get them involved and incorporate their feedback.

5.3 Using Wireline sketches

A wireline sketch is a great way of communicating the intended functionality, without having to develop a prototype. It can be used to stimulate discussions with the end user and to clarify what the user actually wants, rather than what they think they want.

The temptation might be to jump straight into writing PML, but a good sketch is quicker to produce and can say more than writing a functionality specification.

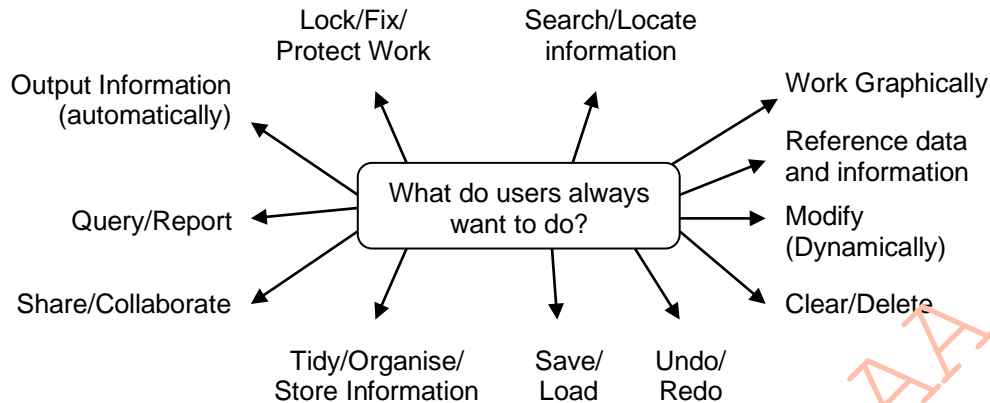
As an example, there is a requirement to give a user the ability to look at the history of an element. Rather than guessing what functionality is needed, the following quick sketch can start the discussion.



5.4 What do users always need to do?

There are certain aspects of functionality that users may expect from the development, but may not explicitly ask for. This expectation will be based on their experience with other programs, even standard Windows.

Not all of the following points will be relevant for every development, but it is worth considering them.



5.5 What is good user interface design?

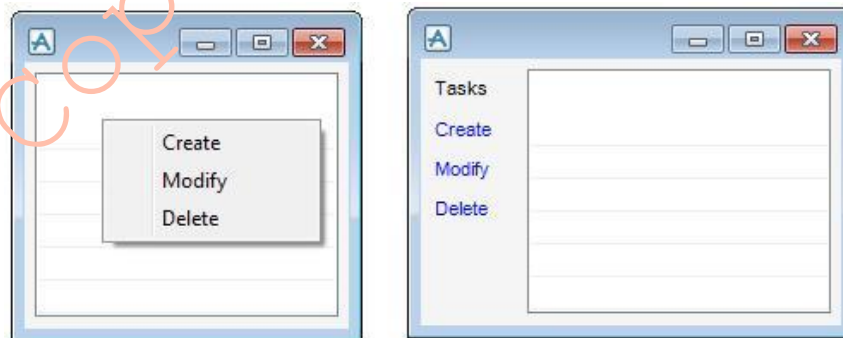
An interface and/or associated functionality can be assessed based on the experience of the end user. Ease of use under minimal guidance should be the benchmark. It is acceptable for the more difficult parts to be explained through documentation, but the aim should be for any user to pick up and use the main functionality on offer.

Although by definition a subjective topic, there are some aspects to form design that are always the case and should be considered when developing any interface:

5.5.1 How easy is the form to use?

The ultimate question and the first consideration when developing an interface. A user should be able to open and use the interface with minimal effort. The intent of the form should be clear and the route needs to be apparent.

Consider the which of the following two forms is easier to use:



Although a context menu will save space on a form, how many users would know to look for it? By moving this menu onto the form, the functionality is obvious and is clearly available to the users.

Some parts of the forms may need to be complex, but this should be kept to a minimum and assistance should be provided through documentation or additional feedback on the form.

The aim should be that a user will be able to use the basic form functionality without guidance

5.5.2 Have the correct gadgets been used?

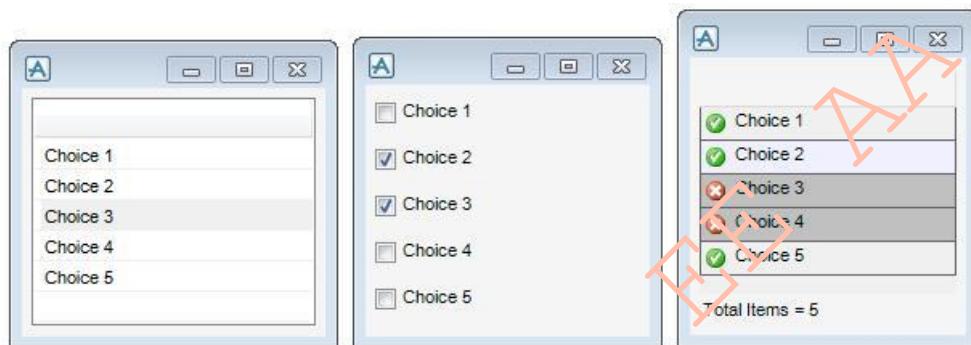
There are many different ways to develop an interface. Different combinations of gadgets and feedback can be used to achieve the same results.

For example, if the user has to make a choice from a set number of options this could be done with a list, option or combo gadget. It could also be achieved with toggles or radio toggles (within a frame).

The choice in gadget should reflect the user expectation and the required outcome. For a short list of fixed choices consider using radio toggles. For longer variable lists, consider using list or combo gadget.

i A combo gadget is good for long lists as a search mechanism could be written within its open callback.

The choice of which gadgets to use will be a trade-off between the desired user experience and the programming effort to achieve it.

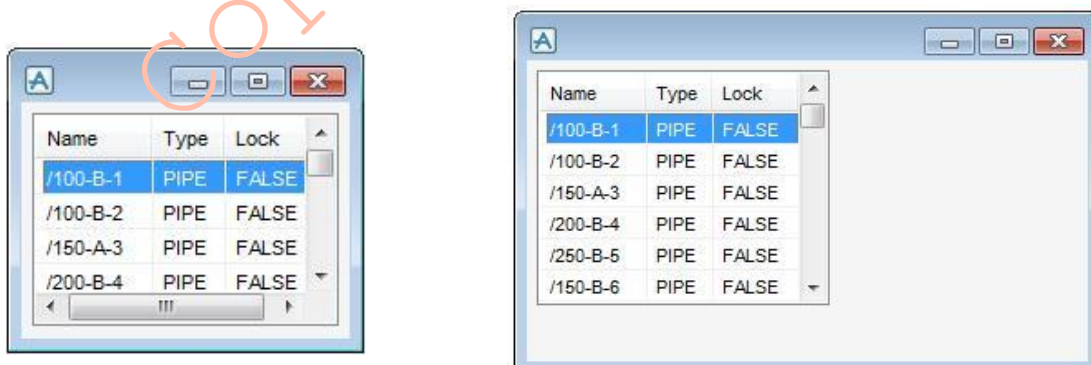


Consider the above three forms. They all provide the same set of multiple choices, but in different ways. As the toggles are defined during form definition, the number is fixed in the first form, while the choices on the others can be extended. However, the code required for the first two is considerably less than the third.

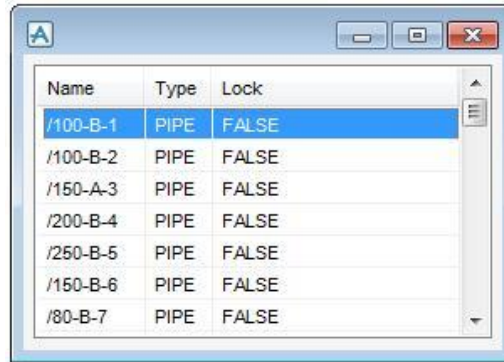
Which provides the greater user experience?

5.5.3 Can the form be resized and have gadgets been anchored?

If a form is designed to display information to the user then the available space should be considered. Is the form big enough to display all the possible information? A resizable form gives users the ability to resize a form to suit its contents. However, allowing a form to resize is only half of the consideration:



If a form can be resized then how the gadgets are docked or anchored needs to be considered. A docked or anchored gadget will resize or move with the resized form. This means that the above list would resize making all the information visible.



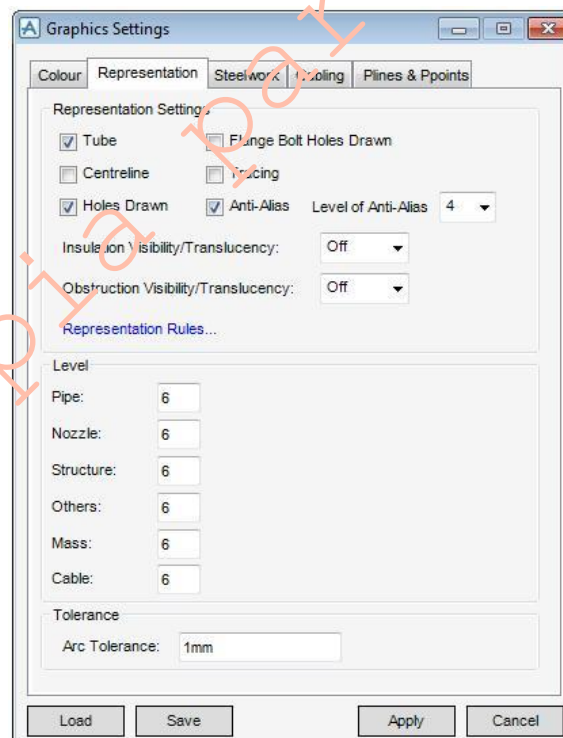
Name	Type	Lock
/100-B-1	PIPE	FALSE
/100-B-2	PIPE	FALSE
/150-A-3	PIPE	FALSE
/200-B-4	PIPE	FALSE
/250-B-5	PIPE	FALSE
/150-B-6	PIPE	FALSE
/80-B-7	PIPE	FALSE

5.5.4 Have progressive or staged disclosure been considered?

Progressive or staged disclosure are techniques of controlling how much functionality is displayed to a user on a form. This limits the choices presented to the user at any one time and delivers a cleaner user experience. The differences between the two techniques are as follows:

- **Progressive** – secondary or additional functionality (i.e. settings) are stored on additional forms or panels on the form. These are usually accessed by buttons from the main form.
- **Staged** – a “wizard-style” approach to progression through a form. The user is only presented with functionality as it is required, effectively stepping them through a process. This can be achieved by controlling the visibility of gadgets on the form, or by sequentially stepping through a series of forms.

For an example of progressive disclosure refer to the “Graphics Settings” form (on the **3DView** tab, in the **Settings** group, click the **Graphics** button). Options are available within the tabs, with the more advanced options found on secondary forms (e.g. “Representation Rules...” under the Representation tab).



Graphics Settings

Colour Representation Steelwork Tooling Plines & Ppoints

Representation Setting:

☒ Tube ☐ Flange Bolt Holes Drawn

☐ Centreline ☐ Tracing

☒ Holes Drawn ☒ Anti-Alias Level of Anti-Alias 4

Insulation Visibility/Translucency: Off

Obstruction Visibility/Translucency: Off

Representation Rules...

Level

Pipe: 6

Nozzle: 6

Structure: 6

Others: 6

Mass: 6

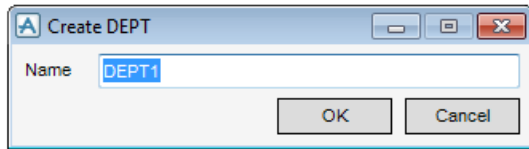
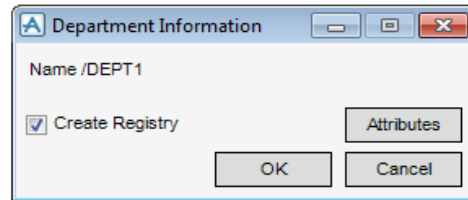
Cable: 6

Tolerance

Arc Tolerance: 1mm

Load Save Apply Cancel

For an example of staged disclosure refer to the “Create DEPT” form (on the **Tools** tab, in the **Explorers** group, select the **Create** option from the **Department** button options list in within E3D Draw). A series of forms are shown in sequence allowing the user to create a Draw hierarchy. At any time the user can break the process, or use progressive disclosure to update attributes.

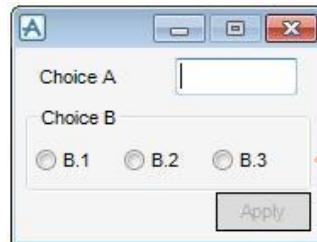
A small dialog box titled "Create DEPT" with a single text input field labeled "Name" containing the text "DEPT1". Below the input field are "OK" and "Cancel" buttons.A dialog box titled "Department Information" with a text input field labeled "Name /DEPT1". Below the input field is a checked checkbox labeled "Create Registry". To the right of the checkbox is an "Attributes" button. At the bottom are "OK" and "Cancel" buttons.

Although not applicable in every development, disclosure allows the user to be guided through a form. If the functionality has a sequence of predefined steps or a set of advanced settings, consider disclosure to tidy up the interface.

5.5.5 How many mouse clicks are required to use the form?

If a user has to make too many choices or interact with lots of form gadgets, this can interrupt the workflow and reduce the user experience.

Consider a form that has many options the user can choose. The user cannot progress with the form until they have worked through each option.

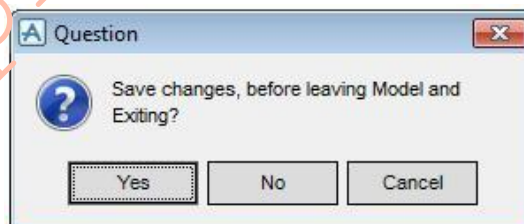
A form titled "Choice A" with a text input field. Below it is a section titled "Choice B" with three radio buttons labeled "B.1", "B.2", and "B.3". At the bottom right is an "Apply" button.

Does the user need to make every choice? A set of default values would allow the user to proceed straight away, or update the choices if required. This could be taken further by writing the users choices to file during the close event so they are available for next session.

5.5.6 Are pop-up alert boxes used?

Alert boxes or pop-ups are very useful to bring information to the users attention, but if overused can become annoying and disrupt the workflow.

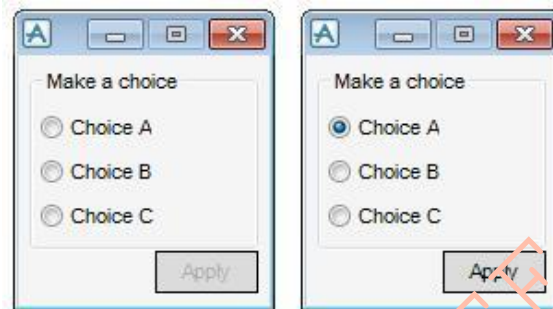
Consider the action of exiting E3D. If work has not been saved it is appropriate to prompt the user asking if they wish to save their work.

A dialog box titled "Question" with a question mark icon. The text inside says "Save changes, before leaving Model and Exiting?". At the bottom are "Yes", "No", and "Cancel" buttons.

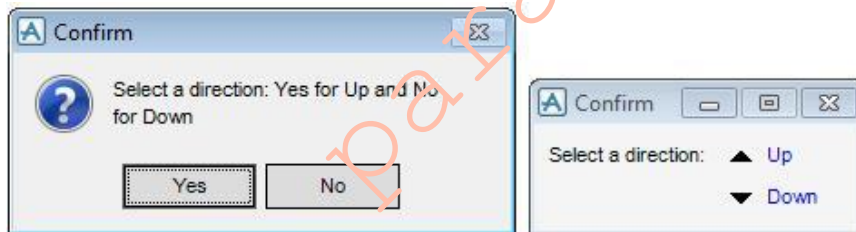
Alert boxes are useful if you want to disrupt the workflow of the form, but consider whether there are better ways of addressing the issue. For example, consider the following form. If the user has not made a choice, they receive a message telling to do so.



A more appropriate approach could be to prevent the Apply button from being clicked before a choice has been made. Additional feedback on the form would be necessary to direct the user.



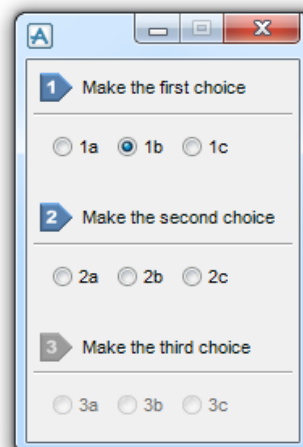
It can be tempting to try and make an alert box fit the question that the user needs to answer. A better solution would be to use a developed child form specifically for the question. This will avoid the classic “click Yes for Up” and give more control over its appearance and functionality.



5.5.7 Is there a flow through the form

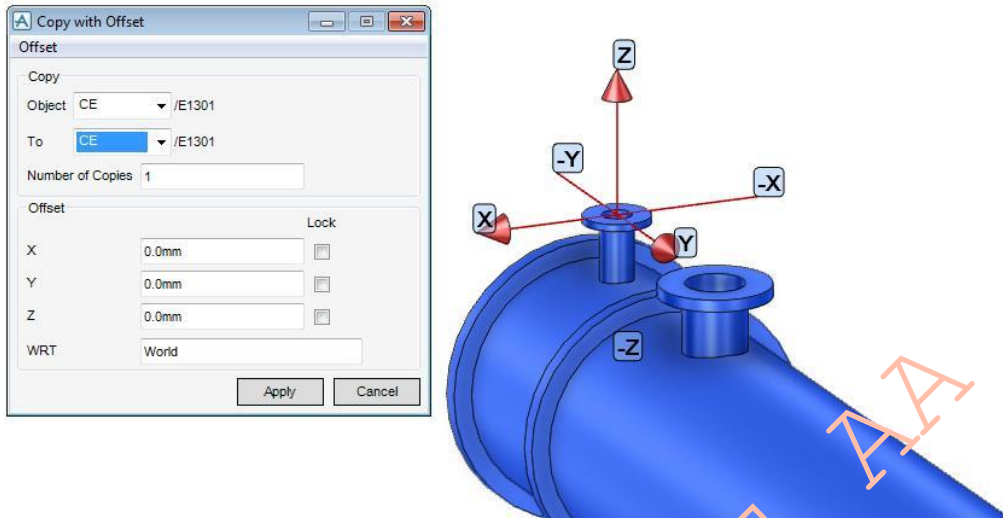
Typically, flow through a form should be as a user would read of text; left to right, from top to bottom. Users expect actions to be completed first at the top of the form and the “Apply” button to be at the bottom.

The flow through a form could be assisted with the use of titles and “steps”. Coupled with the staged activation of gadgets, the users can be led through the form.



5.5.8 How much feedback or assistance is offered?

While a form is being used, the most relevant information should always be available to the user. This could include an appropriate tooltip on a pixmap gadget or highlighting and tagging elements in the graphical view.



5.5.9 How reliable is the form?

Despite the layout and design of the form, users may still use the functionality in a different order. Designing a robust form involves a mixture of testing and predicting what could go wrong. Typically problems which can affect forms include:

- Not having write access to the chosen element
- At the wrong point in the hierarchy to create the required element
- A previous choice on the form has been missed
- Assumptions about the E3D environment (e.g. loaded defaults or forms) are incorrect

It is good practice that for every assumption made in the code, there should be an appropriate check and feedback. Do not wait until the “Apply” button is clicked to check the users inputs.

5.5.10 Is the form familiar?

As users make use of the forms, they can begin to recognise functionality and how forms function. To help this common features need to be provided, such as a button to refresh a form or an “Apply” button to complete the action

Within standard E3D, it is now common to see functionality appear on right docking forms with a “refresh” button at the top and “Tasks” at the bottom.

5.6 Developing a style

With any customisations, it is worth developing common style across them. This will help users recognise functionality and instinctively know how to use it.

This will include the style of icons used, the type and location of buttons on the form, the naming convention of displayed information, wording of feedback etc.

The AVEVA Training Example was developed to mimic the style of the the AVEVA Training guides to link the two together.



5.7 Supplied Exercises

The following exercises have been provided as a starting point for some new PML developments. They will provide an opportunity to apply the topics discussed so far.

For each, the who, what and why have already been addressed, along with some additional points to consider while developing a solution.

- ① *If there is a specific development that you would like to address instead of these exercises, please discuss this with the course tutor.*

Copia para EE AA

Exercise 4 – A development for the E3D Designer

I would like to be able to preview an element inside FODP Design without having to add it to the drawlist. This would mean that I could see elements on their own, rather than having to highlight them in the drawlist.



FODP Designer

Consider the following:

1. Consider the Who, What and Why of the request and develop this as a requirement spec. Identify what functionality is required to address the problem

2. Develop a wireline sketch to outline your proposed development. Highlight the major pieces of functionality and add notes to justify what is being provided

3. Develop the solution, giving extra consideration to the following:

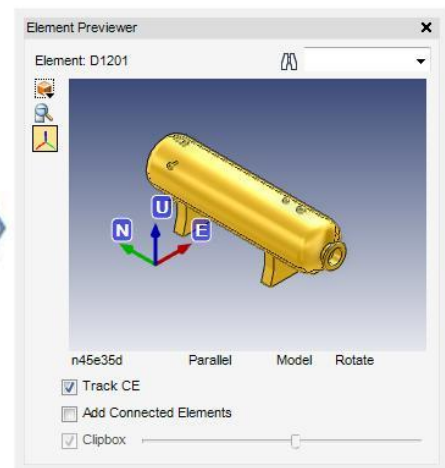
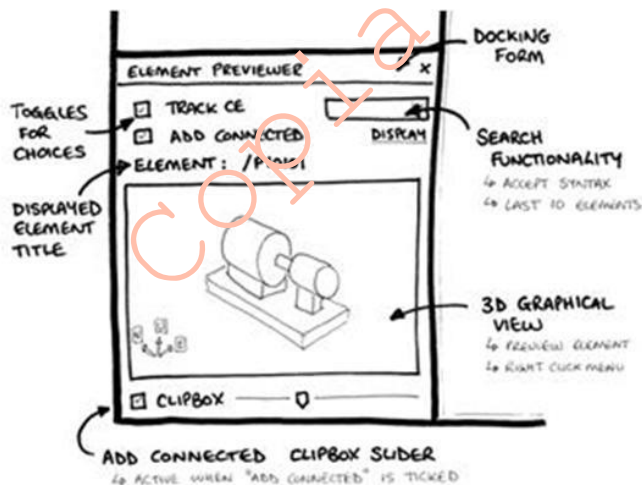
How will the user display the form? How will the form look (wireframe)?

What is the largest element that would be previewed? What level in the hierarchy?

Will the form track the current element, or will the user refresh the preview manually

Can the user type in the name of the element to preview? Can syntax be used (i.e. OWNER)?

What other useful pieces of information would be worth displaying? For example properties, connection elements, reserved volumes etc



4. Finish the development by creating some user documentation to describe its use. How will the development be considered a success?

Exercise 5 – A development for the E3D Checker

Consider the following:

There are a number of checks I always carry out within FODP, such as the name format. It would be great if there was a way of performing these usual checks and getting the results easily into Excel.



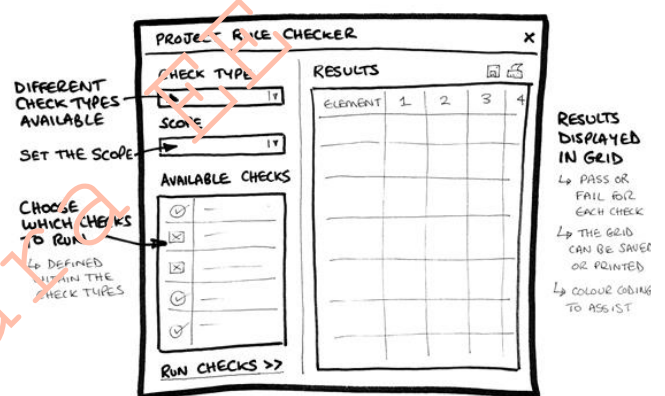
FODP Checker

1. Consider the Who, What and Why of the request and develop this as a requirement spec. Identify what functionality is required to address the problem.

2. Develop a wireline sketch to outline your proposed development. Highlight the major pieces of functionality and add notes to justify what is being provided .

3. Develop the solution, giving extra consideration to the following:

- What is the best way for the checker to perform the checks?
- How should the results of the checks be displayed?
- How will the checks be customised to make them customer specific?
- How can different checks be activated and deactivated?
- How will company and project specific checks be differentiated? Do they need to be?
- What additional functionality would help the check visualise the results of any checks? Could the use of colour, or icons help?
- What sort of checks could be supplied as default? For example:
 - If the element is a SCTN, is the SPREF set
 - If the element is a HVAC, are the DESP set
 - If UDA present, has a value been set
 - How will the development be tested or deemed a success?



Element	1	2	3	4
/C4D4-2	TRUE	TRUE	-	TRUE
/D3D4-1	TRUE	TRUE	-	TRUE
/D3D4-2	TRUE	TRUE	-	TRUE
/D3E3-1	TRUE	TRUE	-	TRUE
/D3E3-2	TRUE	TRUE	-	TRUE
/D4E4-1	TRUE	TRUE	-	TRUE
/D4E4-2	TRUE	TRUE	-	TRUE
/E3E4-1	TRUE	TRUE	-	TRUE
/E3E4-1/HA1-SUPP	TRUE	TRUE	-	TRUE
/E3E4-2	TRUE	TRUE	-	TRUE
/B-C-1A	FALSE	TRUE	-	TRUE
/B-BC-1A	FALSE	TRUE	-	TRUE
/C-BC-1A	FALSE	TRUE	-	TRUE

4. Finish the development by creating some user documentation to describe its use. How will the development be considered a success?

Exercise 6 – A development for the E3D Manager

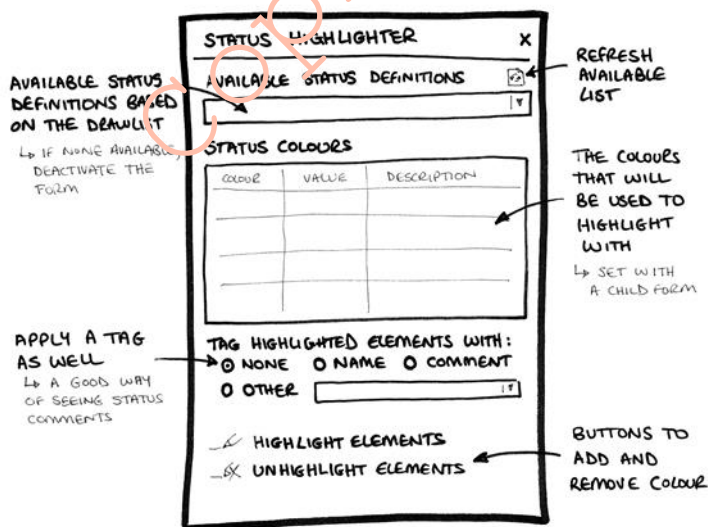
Consider the following:

As the project develops, I would like to be able to log into FODP and get a visual overview of the status. This could be for a range of disciplines, but would be based on the drawlist. It would be great for presentations.



Project Manager

1. Consider the Who, What and Why of the request and develop this as a requirement spec. Identify what functionality is required to address the problem
2. Develop a wireline sketch to outline your proposed development. Highlight the major pieces of functionality and add notes to justify what is being provided
3. Develop the solution, giving extra consideration to the following:
 - How can the available status definitions be identified?
 - What happens to the form if there are no status definitions available. What feedback will there be for the user?
 - How best will the highlight colours used be shown to the user?
 - How can the user update the highlight colours and save their preferences?
 - How can attention be drawn away from the elements not controlled by the chosen status definition?
 - What is the best way for the form to keep track of the elements it highlights?



Status Description	Status Value
Work Pending	0
Work Suspended	15
Work Started	20
Rework	25
Work Completed	40
Preliminary Release	60

Total Items = 8

Tag highlighted elements with:

☒ No Tag ☐ Name ☐ Status Comment ☐ Other

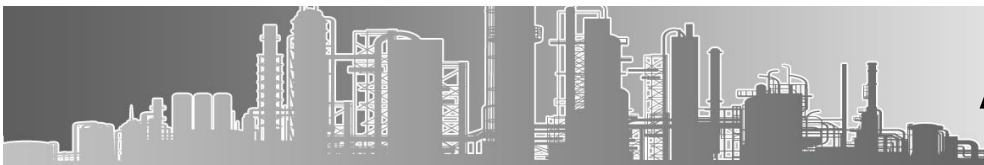
UDTYPE - User defined ele

Tasks

Highlight elements

Unhighlight elements

4. Finish the development by creating some user documentation to describe its use. How will the development be considered a success?



Appendix A – Example PML

Appendix A.1 - Provided form !!c3ex2

```
--
-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited
--
-- File: c3ex2.pmlfrm
--
-- Description:
-- A form that collects and presents elements to users
--
-- This form is the bases of worked examples and exercises 2,3,4 for TM1881
--
--
-- Object definition
--
setup form !!c3ex2 dialog resizable

-- Load the required .dll for the explorer .NET gadget
import 'GridControl'
handle ANY
-- Handle all errors, incase its already been loaded
endhandle
using namespace 'Aveva.Core.Presentation'

-- Set the form title
!this.formTitle = 'Element Collector'

-- Form gadget definition
para .title1 text 'Collection Details'
line .line1 at xmin ymax horizontal width 30

text .types 'Element Types' at xmin ymax+0.5 tagwidth 10 width 19 is string
text .filter 'Search Filter' at xmin ymax tagwidth 10 width 19 is STRING
text .scope 'Search Scope' at xmin ymax tagwidth 10 width 19 is string

button .collect linklabel 'Collect Elements' at xmin ymax+0.5 width 15

line .lineVert at xmax.line1+0.5 y0 anchor t+l+b vertical height 10

para .title2 at xmax+0.75 ymin text 'Collected Elements'
line .line2 at xmax.lineVert+0.5 ymax anchor t+l+r horizontal width 30

container .frame NOBOX PMLNETCONTROL 'grid' at xmin ymax anchor all width 30 height 9

-- Set the form members
member .grid is NETGRIDCONTROL
member .elements is ARRAY

exit

-- Default CONSTRUCTOR method - Setup the grid and set the callback on the collect button
--
define method .c3ex2()

-- Specify the namespace before using the .NET gadget
using namespace 'Aveva.Core.Presentation'

-- Create an instance of the control
!this.grid = object NETGRIDCONTROL()
!this.frame.control = !this.grid.handle()

-- Set the callback on the collect button
!this.collect.callback = '!this.collect()'

-- Ensure the grid gadget looks as required
!this.setupGrid()

endmethod
```

```

-----
-- Method .collect() - Collect the elements based on the criteria --
-----
define method .collect()

  -- If no elements types have been entered, stop the method
  if !this.types.val.length().eq(0) then
    return
  endif

  -- Setup a collection object
  !collection = object COLLECTION()
  !collection.types(!this.types.val.split())

  -- If a filter has been defined, add it to the collection object
  if !this.filter.val.length().gt(0) then
    !expression = object EXPRESSION(!this.filter.val)
    !collection.filter(!expression)
  endif

  -- If a scope has been defined, add it to the collection object
  if !this.scope.val.length().gt(0) then
    !collection.scope(!this.scope.val.dbref())
  endif

  -- Store the collection results
  !this.elements = !collection.results()

  -- Fill the grid with the collected elements
  !this.fillGrid()

endmethod

-----
-- Method .fillGrid() - Fill the grid with the collected elements --
-----
define method .fillGrid()

  -- Specify the namespace before using the .NET gadget
  using namespace 'Aveva.Core.Presentation'

  -- Create headings
  !headings = |Name|

  -- Get the collected elements as STRING objects (required for the grid)
  !data = !this.elements.evaluate(object BLOCK(|!this.elements[!evalIndex].string(|))

  -- Bind data to grid
  !nds = object NETDATASOURCE('Grid Control Example', !headings.split(), !data)
  !this.grid.bindToDataSource(!nds)

endmethod

-----
-- Method .setupGrid() - Format the grid as required --
-----
define method .setupGrid()

  !this.grid.columnExcelFilter(FALSE)
  !this.grid.setNameColumnImage()
  !this.grid.outlookGroupStyle(FALSE)
  !this.grid.fixedHeaders(FALSE)
  !this.grid.fixedRows(FALSE)
  !this.grid.columnSummaries(FALSE)
  !this.grid.autoFitColumns()
  !this.grid.editableGrid(FALSE)

endmethod

-----
-- End of form !!c3ex2 definition --
-----
-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited --
-----

```

Appendix A.2 - Example completed worked example for Chapter 3

```

-----
--
-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited
--
-- File:          elementCollection.pmlobj
--
-- Description:
--   An object collect and store E3D elements
--
--   This object represents a solution to worked example 2 in TM1403
--
-----
-- Object definition
-----
define object elementcollection
  member .elementTypes is ARRAY
  member .filter       is EXPRESSION
  member .scope        is DBREF
  member .elements     is ARRAY
endobject

-- Method .collect() ARRAY - Collect the elements as defined in the object
-----
define method .collect() is ARRAY

  -- Check that some element types have been defined
  if !this.elementTypes.size().eq(0) then
    return
  endif

  -- Setup the collection object with the element types
  !collection = object COLLECTION()
  !collection.types(!this.elementTypes)

  -- If a filter has been created, apply it
  if !this.filter.set() then
    !collection.filter(!this.filter)
  endif

  -- If a scope has been defined, use it
  if !this.scope.set() then
    !collection.scope(!this.scope)
  endif

  -- Store and return the collected elements
  !this.elements = !collection.results()
  return !this.elements
endmethod

-- End of object ELEMENTCOLLECTION definition
-----
-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited
-----

```

Appendix A.3 - Example solution to Exercise 2

```

-----
--
-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited
--
-- File:          formValueStore.pmlobj
--
-- Description:
--   An object to save and restore values from a PML form
--
--   This object represents a solution to worked example 2 in TM1403
--
-----
-- Object definition
--
-----
define object FORMVALUESTORE
  member .form      is FORM
  member .gadgets   is ARRAY
  member .values    is ARRAY
  member .file      is FILE
endobject

-- Default CONSTRUCTOR method - Not used directly
--
define method .formValueStore()
endmethod

-- Overloaded CONSTRUCTOR method - Setup the object with a form and populate the information
--
define method .formValueStore(!form is FORM)

  -- Store the owning form, so the object can refer to it
  !this.form = !form

  -- Store the form gadgets for easier use
  !this.gadgets = !this.form.gadgets()

  -- Create the settings file for the form
  var !user authuser
  !this.file = object FILE(|%E3DUSER%/| & !this.form.name().lowercase() & |-.| & !user & |.settings|)

  -- Populate the object with the information and try to populate the form
  !this.loadValues()

endmethod

-- Method .saveValues() - Save the values to the object and the settings file
--
define method .saveValues()

  -- Get rid of the previous saved values
  !this.values.clear()

  -- Initialise the storage array
  !lines = ARRAY()

  -- Loop through the gadgets and store the values
  do !gadget values !this.gadgets
    !this.values.append(!gadget.val)
    handle ANY
      !this.values.append(| |)
    elseif NONE
      -- If there is not problem with the value, store the line
      !value = !gadget.val
      if !value.objectType().eq(|STRING|) then
        !lines.append(|!!!| & !this.form.name() & |.| & !gadget.name() & |.val = '| & !value & '|)|
      else
        !lines.append(|!!!| & !this.form.name() & |.| & !gadget.name() & |.val = | & !value)
      endif
      !value.delete()
    endhandle
  enddo

  -- Write the settings file - effectively a macro file
  !this.file.writeFile(|OVER|, !lines)

endmethod
--(Continue on the next page)

```



```
-----  
-- Method .loadValues() - load the values from the settings file to the form and object --  
-----  
define method .loadValues()  
  
  -- Check if the settings file already exists  
  if !this.file.exists() then  
    -- If it exists, try and run it  
    $m "$!<this.file>"  
    handle ANY  
    endhandle  
  endif  
  
  -- Clear the stored values  
  !this.values.clear()  
  
  -- Update the stored values  
  do !gadget values !this.gadget  
    !this.values.append(!gadget.val)  
    handle ANY  
    !this.values.append(!I)  
    endhandle  
  enddo  
  
endmethod  
  
-----  
-- End of object FORMVALUESTORE definition --  
-----  
-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited --  
-----
```

Copia para EEAA

Appendix A.4 - Example completed worked example for Chapter 4

Additions to the form definition

```
!this.initCall = !!this.checkCriteria()!
```

Additions to the constructor method

```
!this.types.callback = !!this.checkCriteria()!  
!this.filter.callback = !!this.checkCriteria()!  
!this.scope.callback = !!this.checkCriteria()!
```

Complete method .checkCriteria()

```
define method .checkCriteria()  
  
!this.collect.active = TRUE  
!this.message.val = !!  
  
if !this.types.val.eq(!!) .and(!this.filter.val.eq(!!)) .and(!this.scope.val.eq(!!)) then  
!this.collect.active = FALSE  
!this.message.val = !Please enter some collection criteria!  
return  
endif  
  
!errors = !!  
do !type values !this.types.val.split()  
!elementType = object ELEMENTTYPE(!type)  
!owners = !elementType.parentTypes()  
if !owners.unset() then  
!errors = !errors & !! & !type  
endif  
enddo  
  
if !errors.length().gt(0) then  
!this.collect.active = FALSE  
!this.message.val = !errors & ! - not valid element types!  
return  
endif  
endmethod
```

Appendix A.5 - Example solution to Exercise 3

Additions to the form definition

```
!this.quitCall = (!this.tidy())  
member .highlighted is ARRAY
```

Addition to the end of method .fillGrid()

```
!this.highlightElements()
```

New methods added

```
define method .tidy()  
do !element values !this.highlighted  
  !!gphdrawlists.drawlists[1].unhighlight(!element)  
enddo  
!this.highlighted = ARRAY()  
  
AID CLEAR text 10101  
handle ANY  
endhandle  
  
endmethod  
  
define method .highlightElements()  
  
  !this.tidy()  
  
do !element values !this.elements  
  !!gphdrawlists.drawlists[1].highlight(!element, 5)  
  !this.highlighted.append(!element)  
  
  !name = !element.name  
  
  !pos = !element.position  
  handle ANY  
  !pos = !element.hpos  
  handle ANY  
  !volume = object VOLUME(!element)  
  !pos = !volume.from.midpoint(!volume.to)  
  endhandle  
endhandle  
  
  AID text NUM 10101 |$!<name>| AT $!pos  
  
enddo  
  
endmethod
```

Appendix A.6 - Example solution to Exercise 5

```

--
-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited
--
-- File:          elementPreviewer.pmlfrm
--
-- Description:
--   A form to track and display the current element
--
--   This form demonstrates a solution to the first part of exercise 3 in
--   TM-1882
--
-----
-- Form definition
-----

setup form !!elementPreviewer dialog docking

-- Update the standard form members
!this.formTitle = |Element Previewer|
!this.initCall = |!this.init()|
!this.firstShownCall = |!this.firstShown()|
!this.quitCall = |!this.close()|

-- Declare the method that will follow the current element
track |DESICE| call |!this.track()|

-- Set the standard icon size
!pix = |pixmap wid 16 hei 16|

-- Title above view to display element name
para .title at xmin y0 anchor t+l+r text |Element:| width 20

-- Define the four buttons down the L.H.S of the form
button .limitsCe at xmin ymax $!pix
button .walkDrawlist at xmin ymax $!pix
button .axis toggle at xmin ymax $!pix

-- Declare the view within a containing frame
frame .viewFrame panel at xmax.limitsCe+0.5 ymax.title-0.2 anchor all
view .volumeView at xmin ymax volume
width 35 height 11
limits auto
isometric 3
exit
exit

-- Setup the search gadgets
combo .searchBox at xmax - size y0 anchor t+r width 10
para .searchIcon at xmin - 1.35 * size ymin anchor t+r $!pix

-- Add the toggles to the end of the form
toggle .trackCe |Track CE| tagwid 10 at xmin.viewFrame ymax.viewFrame anchor b+l
toggle .addConn |Add Connected Elements| tagwid 20 at xmin ymax-0.1 anchor b+l+r
toggle .clip |Clipbox| tagwid 5 at xmin.trackCE ymax-0.1 anchor b+l

-- Define the slider
!sliderInfo = |horizontal range -100 100 step 1 val 10|
slider .slider |Clipbox size| at xmax ymin+0.2 anchor b+l+r $!sliderInfo width 29

-- Define the message gadgets
para .message at xmax.viewFrame - size ymin.trackce anchor b+l+r text | | width 20
para .messageIcon at xmin - 1.5 * size ymin anchor b+l pixmap width 16 height 16

-- Additional form members to store information
member .element is DBREF
member .drawlist is REAL
member .clipbox is GPHCLIPBOX
member .limits is ARRAY

exit

-----
-- CONSTRUCTOR Method - Set up the form and gadgets
-----

define method .elementPreviewer()

-- Setup some of the gadgets
!this.slider.tickStyle = 0
!this.slider.active = FALSE
!this.clip.active = FALSE
!this.clip.val = TRUE
-- Continues p.t.o
!this.axis.val = TRUE

```

```

!this.trackce.val = TRUE
!this.message.visible = FALSE
!this.messageIcon.visible = FALSE

!dtext[1] = ""
!dtext[2] = "Clear"
!this.searchBox.dtext = !dtext

-- Create local drawlist within the global object
!this.drawlist = !!gphDrawlists.createDrawList()

-- Run the setup methods (at end of definition)
!this.setIcons()
!this.setTooltips()
!this.setCallbacks()
!this.setUpView()

endmethod

-----
-- First Shown Method - Register the view gadget and attach the drawlist
-----
define method .firstShown()
-- Add 3D view to view system
!!gphViews.add(!this.volumeView)
-- Add local drawlist add to 3D view
!!gphDrawlists.attachView(!this.drawlist, !this.volumeView)
endmethod

-----
-- Initialisation Method - Update the drawlist and active the view
-----
define method .init()
-- Active the volume view
!this.volumeView.active = TRUE
-- Turn on holes drawn
!!gphDrawlists.drawlists[!this.drawlist].holes(TRUE)
!this.setDrawList(!!ce)
endmethod

-----
-- Killing Call Method - Detach the drawlist and then delete it
-----
define method .close()
!!gphDrawlists.detachView(!this.volumeView)
!!gphDrawlists.deleteDrawlist(!this.drawlist)
endmethod

-----
-- Method .track() - If we are tracking the current element, refresh the drawlist
-----
define method .track()
if !this.trackce.val then
!this.setDrawList(!!ce)
endif
endmethod

-----
-- Method .setDrawlist(DBREF) - Display the supplied element in the View
-----
define method .setDrawlist(!element is DBREF)

-- If the supplied element is unset, no point continuing
if !element.unset() then
return
endif

-- Store the supplied element
!this.element = !element

-- Get the drawlist associated with the view
!drawlist = !!gphDrawlists.drawlist(!this.drawlist)

-- Clear the drawlist is a reset is requested
!drawlist.removeall()

-- Add the required element to the drawlist, and update the title
!drawlist.add(!element)
!this.title.val = "Element: " & !element.flnn
-- Continues p.t.o

```

```
-- If connected elements are required
if !this.addConn.val then

    -- Collect all the members of the element with a valid
    !collection = object COLLECTION()
    !collection.scope(!element)
    !expression = object EXPRESSION(|NOT UNSET(CREF) AND NOT BADREF(CREF)|)
    !collection.filter(!expression)

    -- Loop through the collected elements and add the connected to the drawlist
    do !conn values !collection.results()
        !drawlist.add(!conn.attribute(|cref|))
    enddo

    -- Update the volume of the clipbox (if required)
    if !this.clip.val then
        !this.setUpClipBox()
    endif

else

    -- Otherwise, disable the clipbox
    !this.clipbox.active = FALSE
    !this.volumeView.clipping = FALSE

endif

-- Refresh the limits of the view
!this.limitsCE()

endmethod

-----
-- Method .limitsCE() - Refresh the limits of the view to the stored element
-----
define method .limitsCE()
    -- Refresh the limits of the view to ensure the element can be seen
    !!gphViews.limits(!this.volumeView, !this.element)
    !this.limits = !this.volumeView.limits
endmethod

-----
-- Method .walkDrawlist() - Walk to the elements in the drawlist
-----
define method .walkDrawlist()
    -- Get the drawlist associated with the view
    !drawlist = !!gphDrawlists.drawlist(!this.drawlist)
    -- Derive a volume object from the members of the drawlist
    !volume = object VOLUME(!drawlist.members())
    -- Derive a limits array in the expected format
    !limits[1] = !volume.from.east
    !limits[2] = !volume.to.east
    !limits[3] = !volume.from.north
    !limits[4] = !volume.to.north
    !limits[5] = !volume.from.up
    !limits[6] = !volume.to.up
    -- Apply the limits to the volume view
    !this.volumeView.limits = !limits
    handle ANY
        -- Incase the drawlist is empty or has no volume
    endhandle
endmethod

-----
-- Method .addConnected() - Update the form based on the add connected toggle
-----
define method .addConnected()

    -- If adding connected, activate the clip toggle (and maybe the slider)
    !this.clip.active = !this.addConn.val
    !this.slider.active = !this.addConn.val.and(!this.clip.val)

    -- Update the clipbox and refresh the form
    !this.setUpClipBox()
    !this.track()
endmethod

-----
-- Method .slide() - Update the size of the clip box
-----
define method .slide()

    -- Check that the "Add Connected" toggle has been checked
    if !this.addConn.val.not() then
        return
    endif
    -- Continues p.t.o
endmethod
```

```

-- Get the value of the slider
!value = !this.slider.val

-- Store the radius of the view
!radius = !this.volumeView.radius

-- Update the limits of the view
!limits = !this.limits
!modlimit[1] = !limits[1] - !value.power(2)
!modlimit[2] = !limits[2] + !value.power(2)
!modlimit[3] = !limits[3] - !value.power(2)
!modlimit[4] = !limits[4] + !value.power(2)
!modlimit[5] = !limits[5] - !value.power(2)
!modlimit[6] = !limits[6] + !value.power(2)
!this.volumeView.limits = !modlimit

-- Restore the view radius (stops the view from resizing)
!this.volume

-- Update the size of the clipbox
!this.volumeView.clipBoxXlen = !this.clipbox.box.xlength + !value.power(2)
!this.volumeView.clipBoxYlen = !this.clipbox.box.ylength + !value.power(2)
!this.volumeView.clipBoxZlen = !this.clipbox.box.zlength + !value.power(2)

-- Refresh the view
!this.volumeView.refresh()
endmethod

-----
-- Method .search() - Open callback from the combo gadget
-----

define method .search(!gadget is GADGET, !string is STRING)

-- If the user has typed in...
if !string.eq(|VALIDATE|) then

    -- Get the entered text and check if it is an element
    !suppliedName = !gadget.displayText()
    !elementCheck = !suppliedName.dbref()
    handle ANY
        -- Any problem, show the error message
        !this.message.visible = TRUE
        !this.messageIcon.visible = TRUE
        !this.message.val = !error.text
    elsehandle NONE
        -- If no problems, add the element name as second in the list and update the view
        !dtext = !gadget.dtext
        !dtext.insert(2, !suppliedName)
        !gadget.dtext = !dtext
        !this.setDrawlist(!elementCheck)
        !gadget.val = 1
        -- Ensure the error message is not visible
        !this.message.visible = FALSE
        !this.messageIcon.visible = FALSE
    endhandle

--...or if the user selects from the list
elseif !string.eq(|SELECT|) then

    if !gadget.val.eq(1) then
        -- If they picked the first one, do nothing (its blank)
    elseif !gadget.val.eq(!gadget.dtext.size()) then
        -- If they choose the last one, clear the list
        !dtext[1] = | |
        !dtext[2] = |Clear|
        !gadget.dtext = !dtext
        !gadget.val = 1
    else
        -- Otherwise, update the form with the chosen element
        !elementName = !gadget.dtext[!gadget.val]
        !this.setDrawlist(!elementName.dbref())
        !gadget.val = 1
    endif
endif

endmethod

-----
-- Method .setUpClipBox() - Update the size of the clip box based on the stored element
-----

define method .setUpClipBox()

!volume = object VOLUME(!this.element)
!this.clipbox.box = !volume.box()
-- Continues p.t.o

```

```

!this.clipbox.active = FALSE
!this.clipbox.set()
!this.clipbox.active = TRUE
!this.volumeView.clipping = TRUE
!this.slide()

endmethod

-----
-- Method .clipBox() - Active and refresh the clip box slider --
-----

define method .clipBox()
    !this.slider.active = !this.clip.val

    if !this.clip.val then
        !this.setUpClipBox()
    else
        !this.clipbox.active = FALSE
        !this.volumeView.clipping = FALSE
    endif
endmethod

-----
-- Method .setIcons() - Update the icons on the pixmap gadgets --
-----

define method .setIcons()
    !this.limitsCe .addPixmap(!pml.getPathName(|autocepopupicon.png|))
    !this.walkDrawlist.addPixmap(!pml.getPathName(|ng_zoomtodrawlist.png|))
    !this.axis .addPixmap(!pml.getPathName(|worldaxes16.png|))
    !this.searchIcon .addPixmap(!pml.getPathName(|id_search.png|))
    !this.messageIcon .addPixmap(!pml.getPathName(|exclamation-16.png|))
endmethod

-----
-- Method .setTooltips() - Update the tooltips on the pixmap buttons --
-----

define method .setTooltips()
    !this.limitsCe .setTooltip(|Limits CE|)
    !this.walkDrawlist.setTooltip(|Walk to drawlist|)
    !this.axis .setTooltip(|Toggle axis on/off|)
endmethod

-----
-- Method .setCallbacks() - Update the callbacks on the relevant gadgets --
-----

define method .setCallbacks()
    !this.limitsCe.callback = |!this.limitsCE()|
    !this.walkDrawlist.callback = |!this.walkDrawlist()|
    !this.axis.callback = |!this.volumeView.showAxes = !this.axis.val|
    !this.trackCe.callback = |!this.track()|
    !this.addConn.callback = |!this.addConnected()|
    !this.clip.callback = |!this.clipBox()|
    !this.slider.callback = |!this.slide()|
    !this.searchBox.callback = |!this.search()|
endmethod

-----
-- Method .setUpView() - Define the properties of the view and setup the clip box --
-----

define method .setUpView()

    -- Setup the view gadget
    !this.volumeView.borders = FALSE
    !this.volumeView.shaded = TRUE
    !this.volumeView.projection = |PARALLEL|
    !this.volumeView.radius = 100
    !this.volumeView.range = 500.0
    !this.volumeView.eyemode = FALSE
    !this.volumeView.step = 25
    !this.volumeView.showAxes = TRUE

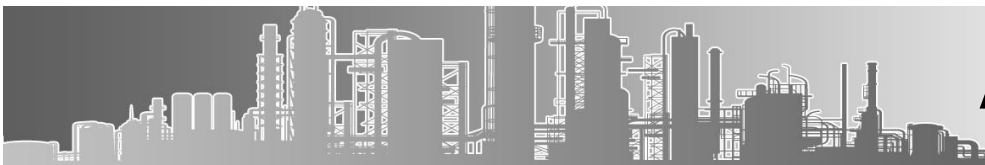
    -- Declare and link the view clipbox
    !this.clipbox = object GPHCLIPBOX()
    !this.clipbox.view = !this.volumeView
    !this.clipbox.capOn()

endmethod

-----
-- End of Form definition and methods --
-----

-- (c) Copyright 2011 to Current Year AVEVA Solutions Limited
-----

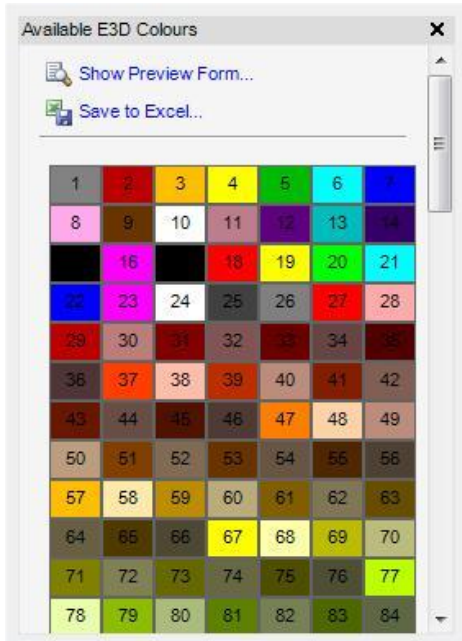
```

Appendix B – Supporting PML Toolbar

This appendix explains the PML utility, supplied to support the development of PML forms and the training course. All the PML for this utility will be found in the supplied files, within the Utilities folder.

Appendix B.1 - Available E3D Colours form



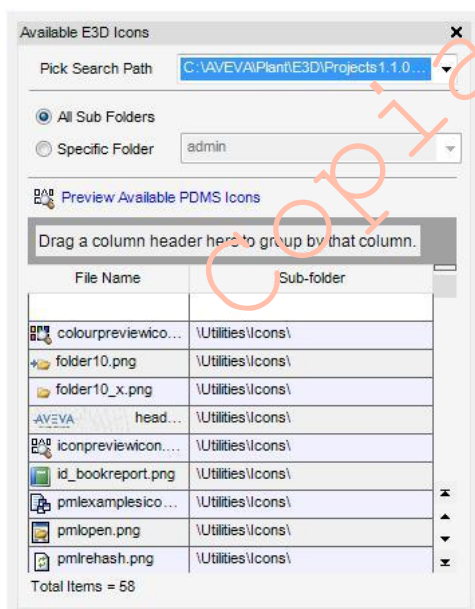
This form will display the available colours within the E3D session. Dynamically built, it will always be correct for the current session.

If the preview form is shown, a larger version of the coloured button is available along with more information about the colour. To update the colour preview, choose a different colour from the main form.

To get a record of the colours, they can be saved to an Excel spreadsheet. This will record the colour name, name and if it's a mix colour the RGB values.

The purpose of this form is to aid with setting colours (e.g. highlighting, gadgets etc) within any customisation

Appendix B.2 - Available E3D Icons form



This form will search the chosen PMLLIB search path for .png image files and display them in the grid control gadget. A preview of the image will be displayed and it will be possible to gain a larger preview from the context menu. PMLLIB variable can refer to more than one directory. As path separator might be used blank space (' ') or semicolon symbol(';'). Check PMLLIB and modify form definition (code line 84).

As the PMLLIB search paths could be large and contain many images, it is possible to only search specific sub folders. Standard grid control functionality is also available to help filter the search results.

The purpose of the form is to visualise all the images available in the PMLLIB search path. As the images are within the search path, these images are therefore available for use in any customisation.

Depending on the size of pml.index file, this form can take a while to collect and display all the information.

Appendix B.3 – Training Examples form

AVEVA Training Examples

AVEVA
CONTINUOUS PROGRESSION

Available Training Courses

TM 1880 TM 1881 TM 1882

Selected Course:
TM-1882 Applied

Available Example Types

Available FORM Examples

Available Examples

- Show Alert Box Example
- Show No Alert Box Example
- Show Replace Alert Example
- Show Easy To Use 1 Example
- Show Easy To Use 2 Example
- Show Form Flow Example
- Show Form Resize Example
- Show Gadget 1 Example
- Show Gadget 2 Example
- Show Gadget 3 Example
- Show Mouse Clicks Example

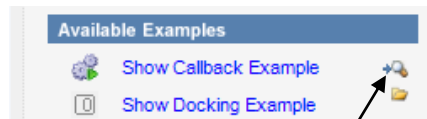
TRAINING

This form will provide quick access to all the training examples supplied as part of this course. Split into types, each example can be shown inside E3D or opened in the default Windows Program.

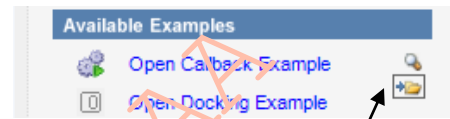
The form builds itself based on the contents of a .xls file saved within the same folder as the form definition.

By making choices in the top two sections, different examples will be available at the bottom of the form.

The examples can be displayed within E3D or their definition opening in the default Windows program. Choosing between the icons on the right will switch this mode:



In "Show" mode



In "Open" mode

The purpose of the form is to speed up access to the training examples and to prompt their future use as reference.

As you attend further courses, more of the options across the top will become available