White Paper

The Modular Approach to Developing TM1 TurboIntegrator Processes

What You'll Learn in This White Paper:

- Limitations of the TurboIntegrator language and how to work around them
- The non-modular approach to developing TurboIntegrator processes
- The modular approach to developing TurboIntegrator processes
- The advantages of the modular approach
- Key principles of the modular approach

TurboIntegrator is the Extract, Transform & Load (ETL) utility for use with TM1. TurboIntegrator, or TI, is an extremely powerful and fast utility which allows data and metadata to be updated in TM1.

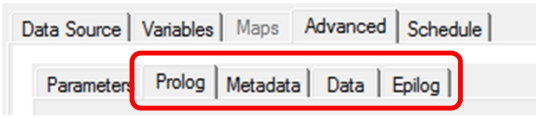
This white paper assumes that the reader has a working knowledge of TM1 and the TurboIntegrator language. It is not intended to be an instruction manual on the TI language, but rather to describe a best practice approach to developing TI processes using a modular approach to development.

As background to the upcoming sections in this guide, the following is a brief overview of the execution of a TI process.

The TurboIntegrator Language

The TurboIntegrator language is a scripting language which contains four distinct coding sections as follows:

- Prolog
- Metadata
- Data
- Epilog



The language is structured this way as TI is most commonly used to load data from an external data source and these steps align with the data loading sequence. However, TI is not limited to just loading data into cubes, it is also capable of manipulating metadata within TM1 such as updating or rebuilding dimensions, and can also perform many other functions. It is not mandatory to include code in all of the sections listed above.

Prolog

The Prolog section of a TI process is always the first section of code to be executed and is executed only once when the process is initiated. If the TI process contains a data source of any type, the code in the Prolog section will execute before the first line of input from the data source is retrieved. This allows the process to perform any pre-load functions such as clearing a section of a cube.

Metadata

The Metadata section of a TI process is executed after the Prolog section has finished. The Metadata section is an implicit loop and each line of code on the tab will execute once for each record of the data source. The purpose of the metadata tab is to update any metadata within the TM1 environment, prior to loading data in the Data section. Only metadata operations should be performed in the Metadata section, for example inserting an element into a dimension.

The code in the Metadata section will continue to execute until the final record from the data source is processed. During the execution of the Metadata section, all metadata changes are made in a copy of the object that they are affecting. After the final record is processed, the data source is closed and all metadata updates are committed to the server.

Note that if the Metadata section does not contain any code, then the data source will not be processed at all for this section.

Data

Following the completion of the Metadata section, the Data section of a TI process is then executed. As per the Metadata section, the Data section is an implicit loop and each line of code on the tab will execute once for each record of the data source. The TI process is effectively making a second pass over the data source but this time through the intention is to execute code that updates data in the system such as writing cells of data to the cube.

The code in the Data section will continue to execute until the final record from the data source is processed. Once again, if the Data section does not contain any code, then the data source will not be processed at all for this section.

Epilog

Finally, after the completion of the Data section, the code in the Epilog section is executed. The Epilog section is intended to execute any final commands required once all data from the data source has been processed. This section of code is commonly used for cleaning up any temporary objects created during the execution of the other three sections of code.

Limitations of TurboIntegrator

TurboIntegrator is an extremely powerful and fast way to load data into TM1. However, there are some limitations of the language that can lead to the development of applications that do not conform to best practice.

While the TI language includes such constructs as loops, branches and other coding mechanisms that are familiar to many programming languages, the language does not include the use of subroutines. Subroutines allow a programmer to define separate functions and/or procedures to perform often repeated tasks and simply call them via a function or procedure call.

Therefore if there is a unique piece of code that needs to be executed many times in different sections of a TI process then the relevant sections of code need to be repeated at multiple points in the TI process. Furthermore there are many instances where the same general functionality is required across many different TI processes, such as clearing out cubes, which leads to further code repetition across processes.

Code repetition leads to the following undesirable outcomes:

Complexity: Processes can often be confusing to anyone but the original author due to the repetition of similar or identical sections of code.

Increase in code size: Processes become bloated with portions of code that are repeated over and over.

Maintenance overhead: Changes required to an often used piece of code will need to be made in every place the code is used. This could include multiple processes and/or multiple sections within a process.

Lack of consistency: If code changes are not correctly synchronised in all areas, the system could start to behave inconsistently or incorrectly.

Impediment to portability: To use similar functionality in multiple places the code needs to be copied and pasted between relevant processes. The programmer will need to read through the source code to specifically identify relevant sections to be copied. This can be very time consuming especially for large and/or complex processes.

Testing overhead: Each process or portion of process functionality must be individually unit tested and user tested leading to longer testing times.

Longer development time: All of the points listed above lead to longer than necessary development time.

Working around the limitations of TurboIntegrator

The previous section outlined the limitations of TurboIntegrator. However there is a way of working with TM1 which achieves a similar outcome to traditional procedural languages.

The way to achieve this is to create a separate TI process for each often used piece of code and call that process from the main process as needed. Using a separate process allows code to be written with the same flexibility of a language that supports subroutines. By using parameters within TI processes arguments can be passed between processes in a similar way to traditional procedure or function calls in other languages. The process behaves the same way as a subroutine for all intents and purposes.

The benefits of working in this more modular way are as follows:

Simplicity: Eliminate code repetition from TI processes making them clearer and easier to understand.

Reduction in code size: Reduce the overall size of processes as multiple lines of code will be replaced with a single call to a standard process.

Reduced maintenance: Whenever a change is required to a core piece of functionality, it only needs to be updated in one place. This reduces the maintenance to a single piece of code rather than a large number of code snippets in multiple locations.

Consistency: As the code that supports each piece of unique functionality only exists in one place it eliminates the possibility of any inconsistency in how standard operations are performed.

Improved portability: By re-using the same code via a process call, it can be used many times over without the need for repetition. If a TM1 environment has multiple TM1 instances the processes can easily be synchronised across all of them.

Enhanced logging: By calling separate TI processes to perform specific tasks, any errors encountered will be logged from the process that encounters the error. This assists in isolating the section of code that requires debugging in the event of an error.

Library of core functions: Over time, as more and more standard processes are created, a library of core processes is created. The more code that can be executed via calls to existing library processes, the less new code that needs to be written and the faster new processes can be created.

Code sharing: Code can easily be shared across different organisations and user communities. By modularising code into separate processes, it is easy to share code or use code shared by others. Each library process is stored in its own .pro file so it can easily be copied, emailed or FTP'ed.

Well documented: Each process has its own unique name and contains named parameters. This makes finding the relevant library process easy as its function should be clearly indicated by the process name. Furthermore, each of the named parameters within the process can be defined as mandatory vs. non-mandatory parameters and can have default values.

Reduced testing time: Each process will be tested and debugged at the time it is originally written and can then be used by many other processes. The specific functionality of the process does not need to be explicitly re-tested every time it is used.

Faster development: All of the points listed above all lead to faster application development time.

Sample Business Requirement

The remaining sections of this document are intended to compare the non-modular and modular approaches to development and describe the modular approach in more detail. These sections will use the following sample business requirement as a basis for examples and explanations:

> A process is required to load general ledger data into TM1 from a file extracted from a source GL system. The file contains one month of general ledger movements. This process needs to zero out the relevant month and year within the general ledger cube before loading the data from the file. For simplicity of this example any new general ledger codes will be ignored by the data load.

The Non-Modular Approach

The non-modular or traditional approach to developing TI processes would generally result in the creation of a single process to perform a particular piece of functionality. The resulting code would make calls to the inbuilt TM1 functions but would not use the concept of a user defined subroutine.

Using the non-modular approach to developing the functionality described in the previous section would result in one process that does everything from zeroing out the relevant section of the target cube to loading the data and cleaning up after itself.

The Prolog section of this process would likely include the following:

- Create a view and relevant subsets for use in zeroing out the cube
- Zero out the cube
- Clean up temporary views and subsets

The Data section of this process will likely include the following:

- Perform any concatenation, decomposition or derivation of input fields to map onto the relevant dimension elements in the target cube
- If data accumulation is required then retrieve the existing value and add it to the data value from the source file
- Send data to the cube

There will be no code in the Metadata or Epilog sections in this example.

The code in the Prolog section may look like this:

```
#==========
# Constants
#==========

cCubeName = 'plan_Report';
cViewName = 'TI Zero Out';
cSubName = cViewName;

cCurrency = 'local';
cVersion = 'actual';
cTime = pMonth | '-' | pYear;


#=========================================
# Zero out current months data prior to load
#=========================================

#---------------------------------------
# Create view to be used for zeroing out
#---------------------------------------

# Delete view if it already exists
If(ViewExists(cCubeName, cViewName) = 1);
  ViewDestroy(cCubeName, cViewName);
EndIf;

# Create view
ViewCreate(cCubeName, cViewName);

#-----------------------------------
# Create subset for version dimension
#-----------------------------------

sDimName = 'plan_Report';

# Delete subset if it already exists
If(SubsetExists(sDimName, cSubName) = 1);
  SubsetDestroy(sDimName, cSubName);
EndIf;

# Create subset
SubsetCreate(sDimName, cSubName);

# Insert relevant elements into subset
SubsetElementInsert(sDimName, cSubName, cVersion, 1);

# Assign subset to view
ViewSubsetAssign(cCubeName, cViewName, sDimName, cSubName);

#--------------------------------
# Create subset for time dimension
#--------------------------------

sDimName = 'plan_Time';

# Delete subset if it already exists
If(SubsetExists(sDimName, cSubName) = 1);
  SubsetDestroy(sDimName, cSubName);
EndIf;

# Create subset
```

```
SubsetCreate(sDimName, cSubName);

# Insert relevant elements into subset
SubsetElementInsert(sDimName, cSubName, cTime, 1);

# Assign subset to view
ViewSubsetAssign(cCubeName, cViewName, sDimName, cSubName);

#----------------------
# Zero out data in cube
#----------------------

ViewZeroOut(cCubeName, cViewName);

#--------------------------
# Clean up views and subsets
#--------------------------

ViewDestroy(cCubeName, cViewName);

sDimName = 'plan_Report';
SubsetDestroy(sDimName, cSubName);

sDimName = 'plan_Time';
SubsetDestroy(sDimName, cSubName);
```

The Data section of this process may look like this:

```
# Retrieve existing value from cube
nCurrentValue = CellGetN(cCubeName, vBusinessUnit, vDepartment, vAccount, cCurrency,
                         cVersion, cTime);

# Accumulate data
nNewValue = nCurrentValue + vLocal;

# Send new value to cube
CellPutN(nNewValue, cCubeName, vBusinessUnit, vDepartment, vAccount, cCurrency,
         cVersion, cTime);
```

Note that for illustrative purposes this process has been kept relatively simple, for example no validation of elements takes place before sending data to the cube.

The code shown above will achieve the desired outcome as specified in the original business requirement. However, there are a number of areas this process could be improved and also lead to the creation of library processes that could be used in a broad number of processes in the future.

The Modular Approach

The modular approach would result in a somewhat different set of processes than the one described in the previous section. While the non-modular approach achieves the desired business outcome in a single process, it has solved one and only one problem. Any requirements in the future to build a similar process for a slightly different purpose would require it to be built from scratch, potentially copying and pasting large sections of code from the original.

The modular approach seeks to abstract code that defines actions or behaviour that may be common to multiple processes and separate that code into its own unique process definition. The goal is to create processes that are standardised and flexible so that they may be able to be used in a wide variety of circumstances.

Firstly, there is some obvious code repetition in the process from the previous section. Consider the following two code samples from the Prolog section of this process:

```
#-----------------------------------
# Create subset for version dimension
#-----------------------------------

sDimName = 'plan_Report';

# Delete subset if it already exists
If(SubsetExists(sDimName, cSubName) = 1);
  SubsetDestroy(sDimName, cSubName);
EndIf;

# Create subset
SubsetCreate(sDimName, cSubName);

# Insert relevant elements into subset
SubsetElementInsert(sDimName, cSubName,
                    cVersion, 1);
```

```
#---------------------------------
# Create subset for time dimension
#---------------------------------

sDimName = 'plan_Time';

# Delete subset if it already exists
If(SubsetExists(sDimName, cSubName) = 1);
  SubsetDestroy(sDimName, cSubName);
EndIf;

# Create subset
SubsetCreate(sDimName, cSubName);

# Insert relevant elements into subset
SubsetElementInsert(sDimName, cSubName,
                    cTime, 1);
```

Note that these two code samples are virtually identical except for the dimension name and the element name used to create the subset. This code could be immediately improved by separating this code into a separate library process and called from the original. The new process would like the following:

Parameters:

```
pDimName
pSubName
pElemName
```

Prolog:

```
# Delete subset if it already exists
If(SubsetExists(pDimName, pSubName) = 1);
  SubsetDestroy(pDimName, pSubName);
EndIf;

# Create subset
SubsetCreate(pDimName, pSubName);

# Insert relevant elements into subset
SubsetElementInsert(pDimName, pSubName, pElemName, 1);
```

Note that this process only requires code in the Prolog section; no code is required in the Metadata, Data or Epilog sections. By replacing the code in the original process with a call to the new library process, the code samples shown earlier would now look as follows:

```
#-----------------------------------
# Create subset for version dimension
#-----------------------------------

sDimName = 'plan_Report';
ExecuteProcess('Lib.Dim.Sub.Create',
               'pDimName', sDimName,
               'pSubName', cSubName,
               'pElemName', cVersion);
```

```
#---------------------------------
# Create subset for time dimension
#---------------------------------

sDimName = 'plan_Time';
ExecuteProcess('Lib.Dim.Sub.Create',
               'pDimName', sDimName,
               'pSubName', cSubName,
               'pElemName', cTime);
```

As can be seen, the code is now simpler and smaller. In addition, there is now a standard library process for creating a subset and inserting a single element that can be used by any other process that requires similar functionality.

There are still further improvements that can be made. The section of code for creating and zeroing out the view is not repeated in this TI, but similar functionality will likely be required by other TI's.

Therefore, the code for creating and zeroing out a view should also be abstracted into a process of its own that can be used by other processes.

The code for zeroing out a view should be flexible enough to be used across a wide variety of circumstances e.g. cubes with different numbers of dimensions. In this case, the library process for zeroing out a cube will be a little more complex than the original code as it has to deal with a greater number of potential circumstances.

Before describing the ideal solution, consider a library process that does exactly what the original process did i.e. zero out a cube by specifying two dimension elements as filters:

Parameters:

```
pCubeName
pDimName1
pElemName1
pDimName2
pElemName2
```

Prolog:

```
#==========
# Constants
#==========

cProcName = GetProcessName();
cViewName = 'TI Zero Out - ' | cProcName;
cSubName = cViewName;


#============
# Create View
#============

# Delete view if it already exists
If(ViewExists(pCubeName, cViewName) = 1);
  ViewDestroy(pCubeName, cViewName);
EndIf;

# Create view
ViewCreate(pCubeName, cViewName);


#==================================
# Create subsets and assign to view
#==================================

# Dimension 1
ExecuteProcess('Lib.Dim.Sub.Create', 'pDimName', pDimName1, 'pSubName', cSubName,
               'pElemName', pElemName1);
ViewSubsetAssign(pCubeName, cViewName, pDimName1, cSubName);

# Dimension 2
ExecuteProcess('Lib.Dim.Sub.Create', 'pDimName', pDimName2, 'pSubName', cSubName,
               'pElemName', pElemName2);
ViewSubsetAssign(pCubeName, cViewName, pDimName2, cSubName);


#======================
# Zero out data in cube
#======================

ViewZeroOut(pCubeName, cViewName);


#===========================
# Clean up views and subsets
#===========================

ViewDestroy(pCubeName, cViewName);
```

```
SubsetDestroy(pDimName1, cSubName);
SubsetDestroy(pDimName2, cSubName);
```

If a library process was created using this code, the original process would simply need to make a call to this new process as follows:

```
ExecuteProcess('Lib.Cube.ZeroOut', 'pCubeName', cCubeName, 'pDimName1', 'plan_Report',
               'pElemName1', cVersion, 'pDimName2', 'plan_Time', 'pElemName2', cTime);
```

However, this new library process has some pretty big limitations:

- It can only zero out cubes where two dimensions are used for filtering (no more, no less)
- Only one element can be specified per dimension

These limitations are fine for the sample business problem described earlier, but are too restrictive to allow this process to be used in other processes that require similar functionality. With a few adjustments this process can be improved to have much greater flexibility and therefore a greater scope of use.

Firstly, the limitation of only one element per dimension can be addressed by modifying the library process that was defined earlier to create a subset. The original process definition allowed the creation of a subset with a single element only; this could be improved by allowing more than one element to be specified. To allow maximum flexibility, this process should not restrict how many or how few elements can be specified. This can be achieved by using a delimiter to separate element names and specify multiple elements in a single variable. The process could be changed as follows:

Original Process	Process with Modifications (changes in red)

Parameters:
pDimName
pSubName
pElemName
 	
Prolog:
# Delete subset if it already exists 	
If(SubsetExists(pDimName, pSubName) = 1);
  SubsetDestroy(pDimName, pSubName);
EndIf;
 	
# Create subset 	
SubsetCreate(pDimName, pSubName);
 	
# Insert relevant elements into subset 	
SubsetElementInsert(pDimName, pSubName,
pElemName, 1);
	Parameters:
	pDimName
	pSubName
	pElements
	pDelimiter (default value: |)
	
	Prolog:
	# Delete subset if it already exists
	If(SubsetExists(pDimName, pSubName) = 1);
	  SubsetDestroy(pDimName, pSubName);
	EndIf;
	
	# Create subset
	SubsetCreate(pDimName, pSubName);
	
	# Insert each element from supplied list into
	the subset
	sElements = pElements;
	
	nDelimiterIndex = 1;
	
	While(nDelimiterIndex <> 0);
	
	  nDelimiterIndex = Scan(pDelimiter,
	sElements);
	
	  If(nDelimiterIndex <> 0);
	
	    sElement = SubSt(sElements, 1,
	nDelimiterIndex - 1);
	    sElements = SubSt(sElements,
	nDelimiterIndex + 1,
	                      Long(sElements));
	
	  Else;

```
    sElement = sElements;

  EndIf;

  # Insert relevant elements into subset
  SubsetElementInsert(pDimName, pSubName,
sElement, 1);

End;
```

Note that this process still supports the original requirement of creating a subset with a single element but now also supports the creation of a subset with multiple elements. Note that the pElemName parameter in the original process has been replaced by two parameters in the new process, one for the list of elements: pElements, and one to specify the delimiter between elements: pDelimiter.

In addition, as the pDelimiter parameter has a default value, it does not need to be specified in the process call unless there is a requirement to use a delimiter that is different to the default. An example call to the modified process above when using the default delimiter of | would be as follows:

```
ExecuteProcess('Lib.Dim.Sub.Create', 'pDimName', sDimName, 'pSubName', cSubName,
               'pElements', 'actual|budget');
```

Note that two elements: actual and budget have been supplied to the process call and are separated using the default | character. If there was a possibility that some of the supplied element names contained the | character then you could make a call using a delimiter other than the default, for example:

```
ExecuteProcess('Lib.Dim.Sub.Create', 'pDimName', sDimName, 'pSubName', cSubName,
               'pElements', 'actual:budget', 'pDelimiter', ':');
```

In this example, the call to the process specified a delimiter of : instead of |. Note the additional pDelimiter argument in this process call to override the default.

So this solves the first limitation, subsets can now be created with multiple elements through a standard process. To solve the second limitation the process defined earlier for zeroing out a section of a cube needs to be modified further. The original parameters for this process were:

```
pCubeName
pDimName1
pElemName1
pDimName2
pElemName2
```

Even if the parameter list was modified slightly to allow multiple elements to be specified...

```
pCubeName
pDimName1
pElements1
pDimName2
pElements2
```

...the process call would still be limited to two dimensions.

The number of parameters could be increased as follows...

```
pCubeName
pDimName1
pElements1
pDimName2
pElements2
pDimName3
pElements3
pDimName4
pElements4
```

...however this approach will always result in a fixed number of arguments.

Even if the number of parameters was set to the same number as the cube that has the largest number of dimensions, there is still a possibility that a new cube could be created in the future that exceeds the current maximum. In addition, there will be many examples where a call to this process may want to specify fewer dimensions, after all this process is designed to zero out section of a cube, in doing so you could you use as few as one dimension to filter.

For maximum flexibility the process needs to allow a flexible number of arguments.

It is not possible to define a TI process that has a flexible number of parameters. You can define parameters with default values which don't need to be specified by calls to the process, however the resulting TI will still have to interrogate these parameters to check whether a value has been supplied or not. Also, this approach will result in code repetition as the process will need to perform the same action on each parameter.

The alternative to using separate parameters for each argument is to use a single parameter that contains a list of arguments, and use a delimiter to separate each argument. This approach has already been demonstrated in the definition of the earlier process that created a subset with multiple elements. However in the case of zeroing out a cube view the requirement will be to have multiple "sets" of arguments. This is because there will need to be a flexible number of dimensions supplied as arguments but also a flexible number of elements supplied within each dimension as well.

Consider an example where a cube is to be filtered using three dimensions, the first dimension requires only 1 element, the second dimension requires 2 elements and the third dimension requires 5 elements (8 elements in total). The TI parameter required cannot simply contain a list of the 8 elements as it is possible that the same element name exists in multiple dimensions and the TI would have no way of determining the correct dimension to filter.

The way of addressing the flexibility required in this example is to use multiple delimiters to separate each "set" of arguments. Consider the following:

- Use one delimiter to separate the sets of arguments e.g. :
- Use a different delimiter to separate the individual components of each set of arguments e.g. |

Using the example from above the parameter string would look as follows:

```
Dim1|Elem1:Dim2|Elem2|Elem3:Dim3|Elem4|Elem5|Elem6|Elem7|Elem8
```

The library zero out process would now simply need to use a loop to split this string into its components starting with the first delimiter and then the second. For example, the process would first break the above string into three sets of arguments as follows:

```
Dim1|Elem1
Dim2|Elem2|Elem3
Dim3|Elem4|Elem5|Elem6|Elem7|Elem8
```

In our example, the first argument will always be the dimension name followed by one or more element names. The code would therefore need to split the dimension name from the element names first, for example:

```
Dim3
Elem4|Elem5|Elem6|Elem7|Elem8
```

And finally separate each element from the elements string into individual elements.

For most flexible use, the delimiter characters should themselves be parameters to allow the calling process to define its own delimiters. This is of particular importance because different data sets may have arguments that contain the default delimiter character within them.

This approach has advantages for both the calling routine and the library process itself as the number of dimensions and elements that can be specified in a process call is not limited by the process itself. The only limitations will be on the physical string length that TM1 can handle for a particular parameter (approx. 64,000 characters in Unicode versions of TM1) which is more than sufficient for most purposes.

Now consider the following set of parameters:

```
pCubeName
pFilter
pDelimDim   (default value :)
pDelimElem  (default value |)
```

The pFilter parameter now becomes the "intelligent" parameter that can accept multiple dimensions containing multiple elements without limiting how many dimensions or how many elements can be used. The pDelimDim parameter is the string that will be used to delimit the main set of arguments, and pDelimElem will be used to delimit the elements in a dimension. In our earlier example the process call would now look as follows:

```
ExecuteProcess('Lib.Cube.ZeroOut', 'pCubeName', cCubeName, 'pFilter',
               'plan_Report|actual:plan_Time|jan-2004');
```

Note that we have used the default delimiters in this example so there was no need to supply the pDelimDim or pDelimElem parameters.

The TI code in the zero out process now simply needs to interpret the pFilter parameter using the specified delimiters to determine the dimensions and elements to use as a filter when zeroing out the view.

The resulting code from the zero out process would now be:

```
#==========
# Constants
#==========

cProcName = GetProcessName();
cViewName = 'TI Zero Out - ' | cProcName;
cSubName = cViewName;


#============
# Create View
#============

# Delete view if it already exists
If(ViewExists(pCubeName, cViewName) = 1);
  ViewDestroy(pCubeName, cViewName);
EndIf;

# Create view
ViewCreate(pCubeName, cViewName);


#=============================================================
# Deconstruct filter, create relevant subsets and assign to view
#=============================================================

#-----------------------------------------------------
# Step 1: Split filter into dimension|elements components
#-----------------------------------------------------

sFilter = pFilter;
nDelimDimIndex = 1;

While(nDelimDimIndex <> 0);

  nDelimDimIndex = Scan(pDelimDim, sFilter);

  If(nDelimDimIndex <> 0);

    sArgument = SubSt(sFilter, 1, nDelimDimIndex - 1);
    sFilter = SubSt(sFilter, nDelimDimIndex + 1, Long(sFilter));

  Else;

    sArgument = sFilter;

  EndIf;

  #-----------------------------------------------------
  # Step 2: Split argument into dimension and elements
  #-----------------------------------------------------

  nDelimElemIndex = Scan(pDelimElem, sArgument);
  sDimension = SubSt(sArgument, 1, nDelimElemIndex - 1);
  sElements = SubSt(sArgument, nDelimElemIndex + 1, Long(sArgument));

  #----------------------------------------
  # Step 3: Create subset and assign to view
  #----------------------------------------

  ExecuteProcess('Lib.Dim.Sub.Create', 'pDimName', sDimension, 'pSubName', cSubName,
                 'pElements', sElements);
  ViewSubsetAssign(pCubeName, cViewName, sDimension, cSubName);

End;


#======================
# Zero out data in cube
#======================

ViewZeroOut(pCubeName, cViewName);


#============================
# Clean up views and subsets
#============================

# Clean up view
ViewDestroy(pCubeName, cViewName);
```

```
# Clean up subsets
sFilter = pFilter;
nDelimDimIndex = 1;

While(nDelimDimIndex <> 0);

  nDelimDimIndex = Scan(pDelimDim, sFilter);

  If(nDelimDimIndex <> 0);

    sArgument = SubSt(sFilter, 1, nDelimDimIndex - 1);
    sFilter = SubSt(sFilter, nDelimDimIndex + 1, Long(sFilter));

  Else;

    sArgument = sFilter;

  EndIf;

  nDelimElemIndex = Scan(pDelimElem, sArgument);
  sDimension = SubSt(sArgument, 1, nDelimElemIndex - 1);

  SubsetDestroy(sDimension, cSubName);

End;
```

Notice that the library process to clear a cube view itself calls the library process to create a subset.

Although our generic library process to zero out cube data is quite long and detailed, importantly it is flexible enough to be used in any process on any server that requires a view to be created and cleared. As the code is re-usable and portable and can be discretely unit tested it can be in effect equivalent to a function call in other programming languages and can be used with great effect to simplify other processes and reduce the overall code base and code complexity within a TM1 system.

The resulting code in the Prolog section from the original process is now reduced significantly to:

```
#=========
# Constants
#=========

cCubeName = 'plan_Report';
cCurrency = 'local';
cVersion = 'actual';
cTime = pMonth | '-' | pYear;


#================================
# Zero out curent months data prior to load
#================================

sFilterVersion = 'plan_Report|' | cVersion;
sFilterTime = 'plan_Time|' | cTime;
sFilter = sFilterVersion | ':' | sFilterTime;

ExecuteProcess('Lib.Cube.ZeroOut.2', 'pCubeName', cCubeName, 'pFilter', sFilter);
```

The Data section remains unchanged.

Modular vs. Non-Modular

This section revisits the advantages of the modular approach that were described in the earlier sections with respect to the example business problem:

Simplicity: The repetition of the code to create two separate dimension subsets was moved into a separate library process and the code to zero out the view was moved to a separate process. This made the final code in the main data load process simple and easy to read.

Reduction in code size: Following on from the previous point, the code in the main process was reduced significantly using the modular approach.

Reduced maintenance: Library processes were created for building a subset and zeroing out a cube. Both of these processes can now be called from any other process in the system.

Consistency: The code for building subsets and zeroing out cubes has now been standardised and stored in a single place.

Improved portability: Code is now stored in a separate .pro file and can easily be synchronised across other TM1 environments.

Enhanced logging: Any errors encountered by the library processes will be logged by that process making it easier to find errors when they occur.

Library of core functions: The library now contains two core functions.

Code sharing: Code is now stored in a separate .pro file and can easily be shared with others.

Well documented: Each process has its own unique name and contains named parameters. The zero out process in particular uses default values for parameters.

In addition to the points listed above, an important point needs to be noted here. In the case of the sample business problem used in this document, it could be argued that it actually took more lines of code in the three processes that resulted from the modular approach compared to the single process used in the non-modular approach.

This is certainly true in this case because we used additional code in the modular library processes to make them more flexible and allow them to be used in a broader range of circumstances in the future. However, the modular approach is designed to pay back when used in the context of an entire system or systems. The more processes that use calls to standard processes, the less new code is required and therefore the investment in time in effort in creating the library process pays back over and over again as that time is saved when developing new processes, modifying an existing system or developing an entirely new system.

Key Principles When Using the Modular Approach

Flexibility

Make library processes as flexible as possible so that they may be used in the largest number of circumstances. In the short term this may require more complexity to be developed in the library process, but in the long term its use will be greater and therefore have greater payback. You will only have to code this process once, and this complexity will be hidden from the person simply calling the function, all the caller needs to know is what does this process do and what parameters are required, the complexity of the process is not their concern. Conceptually, this is quite similar to object oriented programming principles where the complexity is hidden within the object and a simple interface is published for all those who would like to use the objects. In the case of TI, the interface is the set of parameters you define on your generic process.

Error handling

Make sure that library processes handle errors effectively. As library processes will be used over and over it is worth the additional effort to handle any potential errors efficiently and effectively. For example, what if a call to a library process which requires a list of dimension elements as a parameter contained some element names that were not in the dimension being processed? Should the process add the elements that do exist and ignore the ones that don't? Should it return an error code to the calling process and not create the subset? Questions such as these should be considered to ensure the library of standard processes is robust.

Modification of existing library processes

If an existing library process requires modification, make sure the change doesn't cause problems for existing processes that are using it. When using a library process call, developers should feel confident that the process being called works both now and in the future. Ensure changes to library processes are rigorously tested. Similarly, if an additional parameter is to be added to an existing library process, make sure that either a default value is assigned to the new parameter or the code in the library process is robust enough to still work if this new parameter is omitted. This will ensure that all existing calls to the library process will continue to function as originally intended.

Processes with a variable number of arguments

In the case of library processes that require a variable number of arguments, use a parameter for the argument list and another parameter for the list delimiter. Then write code that cycles through the list to retrieve each argument. You should define parameters to be flexible enough to handle the worst case scenario, even if that scenario rarely occurs.

Process Library

It may make sense to create additional library processes to avoid code repetition in other library processes. This is actually quite common, the example of creating a subset as part of the view zero out process is one such example. There is no problem in having a library process that only ever gets called from another library process; the intention of the modular approach is to create an efficient, easily maintainable and elegant system that other developers can easily understand.

What Standard Processes Should Be Defined

There are many obvious processes for standardisation that could be used across any TM1 system. Tasks such as zeroing out a section of a cube, unwinding a dimension hierarchy, and snap-shotting

data from one version to another are just a few examples. There will also inevitably be standard processes that are specific to particular organisations or industries.

Cubewise has developed a library of standard processes that can be used across any TM1 system. A complete listing of Cubewise standard processes are included Cubewise.

When should a separate process be created

When considering whether to create a new library process, use the following as a general rule of thumb:

- Are there sections of code that are being repeated multiple times in a process or across a number of processes?
- Is there functionality you are building now that is likely to be required by other processes in the future?

If the answer to either of these questions is yes, consider creating a separate library process.