White Paper

A Best Practice Guide to Developing TM1 Turbo Integrator Processes

What You'll Learn in This Chapter:

- General principles for best practice process design
- Guidelines for data sources, variables and maps
- Guidelines for advanced scripting
- Naming conventions
- Coding conventions
- Considerations for cube logging
- Debugging and error management

TM1 is an extremely flexible application development platform and includes all of the tools required to develop a complete application from end to end. The flexibility of the environment allows applications to be developed very quickly; application prototypes can be developed in a matter of hours and full applications can be developed in just days or weeks.

The TM1 development environment is also very fast to learn with newcomers to TM1 capable of developing their own applications within just a couple of days of training. The system is very intuitive, especially for those who have developed complex spreadsheet models in the past.

The flexibility described above enables applications to be built quickly and easily, however this level of flexibility can also have some undesirable side effects if projects are not developed or managed well. It is very easy to create "bad" applications that may solve specific business problems, but leave little room for future development and improvement.

To ensure that the TM1 platform is fully leveraged and systems are developed to a high standard there are two key principles that are recommended:

- Implementation of systems that follow best practice development principles
- Management of TM1 projects using a methodology that complements the technology

This document is intended to address the first point; that is to describe an approach to developing TM1 systems using best practice principles.

Why Use Best practice?

Using a best practice approach to developing systems is a way of developing systems that are:

- Efficient
- Consistent
- Reliable
- Easy to understand
- Easy to maintain
- Easy to extend

The intention of best practice is to establish a series of development guidelines and principles that are the result of the collective experience and learning of many TM1 developers over many TM1 projects and systems. Often this is the result of trying many different approaches to solving problems and settling on the approach that is most suitable in the widest variety of circumstances.

Best practice is also a way of making the overall development approach consistent across the entire TM1 system (or systems). This enables the system to be easily understood, maintained and extended by multiple developers with potentially different backgrounds and experience.

It is not the intention of best practice to dictate how problems should be solved, that will always be the creative responsibility of the developer or developers involved in developing TM1 systems. Instead it is a framework of guidelines and principles that allows the creative process of developing systems to result in an efficient and consistent outcome that is easily understood by fellow developers and allows the systems to be easily maintained and extended in the future.

In addition, using best practice allows the ongoing maintenance and development of the systems to happen faster and be more efficient. This is because new developers will be working with a system that is easy to understand and therefore faster to learn. It will also result in the build up of a library of standardised code that acts as a blueprint for future development.

Introduction

Like any programming or scripting language, there are infinite ways to achieve a particular outcome when building turbo integrator process. Two different developers presented with the same business problem are unlikely to come up with the exact same design; in some cases their respective designs may be vastly different.

The turbo integrator development environment imposes some constraints on the way processes are designed by separating the development steps into different sections or tabs within the process. However, there are parts of the language that are very unconstrained, e.g. variables don't have to be declared before they are used, and TM1 does impose strict rules on naming variables.

This white paper describes the best practice approach to developing TM1 TurboIntegrator processes. It steps through the components of the TI process is a similar order to the development of a TI process.

Data Sources

TI processes can have data sources of different types including

- ODBC
- Text
- ODBO
- TM1
- SAP

ODBC Data Source

Firstly, there is a consideration as to the level of complexity within the SQL statement versus the complexity of the TI code that will be used to process the data. Often times a decision will need to be made between the following: (i) place the majority of the complexity of the data load in the SQL statement and make the TI code as simple as possible, or (ii) place the majority of the complexity of the data load in the TI code and make the SQL statement as simple as possible.

For example: when reading data from a transactional table with many records that need to be summarised in TM1, is it better to sum the data in the SQL statement or accumulate the data in the TI? There are many considerations that should be taken into account when making this decision including:

- **Speed:** Is it faster to sum data in SQL or accumulate in TI? Depending on the size of the source table, the level of filtering in the WHERE clause, the indexing of the underlying table(s) etc. it may be faster to accumulate data in SQL or it may be faster to read all of the underlying transaction records from the data source and accumulate them in TM1. Some testing and trial and error may be required here to determine the fastest method.

- **Maintainability:** Are skilled SQL personnel readily available to assist with the initial development and ongoing maintenance of the SQL statement? Are skilled TM1 personnel readily available to assist with the initial development and ongoing maintenance of the TI code? Consideration of what skill sets are available during the development phase and the ongoing maintenance phases is important.

- **Complexity:** Would it be more complex to sum the data in SQL or accumulate in TI? If one method is far more complex than the other it would likely be better to use the less complex of the two.

- **Lookup and Reference Data:** If the process needs to look up data from other TM1 cubes and use that information as part of the accumulation process, then it would be better to accumulate the data in TI.

In addition, if there are many TI processes that access data from ODBC data sources, the considerations above should be made with the bigger picture in mind. Overall it would be best to take a consistent approach across all processes that have ODBC data sources. This may not always be practical and should be used more as a rule of thumb than a hard and fast decision.

Secondly, when using ODBC data sources, the SQL statement should be set to retrieve exactly the amount of data required by the TI process: no more, no less. Often the SQL statement used when building a TI process has been copied from an existing report or other process in the source system and is used as a starting point within the TI rather than writing the SQL from scratch. This approach is fine, but the SQL should be refined where possible to retrieve only the records and fields required by the TI process. Any additional records or fields that are retrieved from the data source but are not used in the TI will just slow the data retrieval process down and should be avoided.

Thirdly, consideration should be given to the field names that are used when retrieving data from the ODBC data source. Any field names that contain spaces or other punctuation, or are TM1 reserved words will need to be manually renamed in the Variables tab of the TI process. While renaming variables is an allowable feature within TI it can lead to maintenance issues later on if not managed carefully. Consider the following example:

A process is required to load one month of general ledger movements at a time and load them into the General Ledger cube in TM1. The initial SQL statement is as follows:

```
select
  gl.year,
  gl.month,
  gl.account,
  gl.monthly movement
from
  general_ledger as gl
where
  gl.year = ?pYear? and
  gl.month = ?pMonth?
```

Once this SQL statement is "Previewed" for the first time in TI, TM1 will attempt to use the existing field names from the data source as long as they don't contain spaces, punctuation or are TM1 reserved words. In this example year and month are reserved function names and monthly movement contains a space so only the account field will retain its original name. The other fields will be assigned names like V1, V2 etc. Of course V1 and V2 are not very descriptive and are not in keeping with best practice development principles.

So, there are two options: (i) rename the variables from V1, V2 to something more meaningful such as vYear and vMonth in the variables tab, or (ii) rename the variables to something more meaningful in the SQL statement.

Renaming the variable names in the Variables tab has one major pitfall. If additional fields are added to the SQL statement in the future or the order of the fields is changed for any reason, then TM1 won't know which original field names to match with which renamed variables (as the relationship was based on the field position which has now changed). This would require the field names to have to be renamed again when changing the SQL statement, or worse the fields may become jumbled and the year field may map to the vMonth variable and so on.

For this reason, it is recommended best practice to rename the fields in the SQL statement itself and not in the Variable tab in TI. This will avoid any problems when new fields are added or the order of fields is changed for any reason. The SQL from the earlier example would now look like:

```
select
  gl.year as vYear,
  gl.month as vMonth,
  gl.account as vAccount,
  gl.monthly movement as vMovement
from
  general_ledger as gl
where
  gl.year = ?pYear? and
  gl.month = ?pMonth?
```

Text Data Source

Text data sources don't offer the same level of flexibility as the ODBC data source when it comes to the fields and records in the data source and the variable names for each of the fields. The TI process must work with what is in the text file "as is" and perform the necessary steps in the TI code itself to achieve the desired outcome.

When specifying the data source it is important to access the source text file via a file path that will work both now and in the future even if the underlying file system environment changes. When specifying the path to the relevant file, two paths must be entered: (i) Data Source Name and (ii) Data Source Name On Server. The first path is used during the development of the TI process for the purpose on previewing the data from the source file and establishing the variables that will result from the source file. This path is not used during the execution of the process. The second path is used during the execution of the process.

The Data Source Name file path should always be specified relative to the workstation(s) of the developer(s) creating and maintaining the process. The Data Source Name On Server file path should always be specified relative to the TM1 server itself.

There are two choices when specifying the path to the source file: (i) use a path that contains a drive mapping to the desired folder, or (ii) use the Universal Naming Convention (UNC) path to the file. For example:

- **Drive Mapping:** t:\transfer\general ledger\import\gl_2011_jan.csv
- **UNC Path:** \\server_name\share_name\general ledger\import\gl_2011_jan.csv

In the two examples given above, it is actually the same file but accessed via two different mechanisms. There are advantages and disadvantages to both approaches.

The advantage of using a drive mapping is that if the server that contains the source data file changes in the future then the t: drive just needs to be re-mapped to the new server and the TI will continue to work. The disadvantage of using a drive mapping is that drive mappings can differ for different machines and/or different usernames on the network. If one developer uses a t: drive mapping to the source files but another developer uses a u: drive then the second developer will experience errors when trying to modify the process. If drive letters are used it is recommended that the drive is mapped consistently for all affected users via a login script rather than being manually mapped by the users themselves.

The advantage of using the UNC path is that no matter who accesses the TI process, it will access the respective data file regardless of any drive mapping that the user or workstation may have in place. That is because the file path is a more specific and direct mapping to the source file location.

The disadvantage of using the UNC path is that if the server changes in the future the TI process will need to be updated to reflect a change in server name or share name.

In either case, it is best not to hard code the Data Source Name On Server path in the TI process. The reason for this is that if the server that contains the source file changes and that same server is used in many other TI processes, then all of the affected TI's will need to be updated one by one. It is far better to store the respective paths in a system or control cube, retrieve the path during the Prolog section of the TI process and reset the data source to this path before the Metadata section of the TI commences. This can be achieved by using the DataSourceNameForServer function in TI. In the event that the source server changes it only needs to be updated in one place in the system or control cube.

Another consideration when using Text data source is variable names for fields from the data source. If the source file does not contain field names in the first row of the file then there is only really one option which is to rename variables from V1, V2 etc in the variables tab to something more meaningful. If the text file does contain variable names then the names from the source file should be used where possible. Of course if these are spaces, punctuation or reserved words in the file then TI will still convert these to V1, V2 etc.

For similar reasons as were specified in the section on ODBC data sources, it is best not to rename variables directly in the Variables tab. If future versions of the source file change by adding additional fields or changing the order of the fields then a similar problem will arise.

TM1 Data Source

Both TM1 cube views and dimension subsets can be used as data sources for TI processes. To use a cube view or dimension subset it must be public.

For best practice, TI processes should always use "process specific" views and subsets as opposed to using existing views and subsets that may already be in use for slicing and reporting. Bear in mind that subsets and views often serve multiple purposes in any given TM1 application and you can't always be sure that the local administrator won't change or delete views and/or subsets.

To create "process specific" views and subsets, write code in the Prolog section that creates them and assigns them to the process as the data source.

Variables from the Data Source

Variable Names

The default name for a variable from the data source is the name of the field (if an ODBC data source is used or if an ASCII data source is used where a header row is present). If a TM1 cube view or subset is used then the default variable name is the name of a dimension or the word "Value"

If the field name from the data source contains spaces, punctuation or is a TM1 reserved word then the field name will default to names like V1, V2 etc. You have the option of renaming these variables in the Variables tab.

The naming convention for variable names will be covered later in this chapter, however at this stage it is important to highlight that the names of these variables should be self documenting. That is, anyone reading the code should immediately be able to derive the nature of the variable from its name. This will make code more understandable and therefore easier to maintain and extend in the future.

If the variable names from the source system consistently conform to self-documented names then they can remain as they are from the source. However, it is often the case that variable name from the source convey little meaning, in which case the variables should be renamed to something meaningful that conforms to variable naming standards.

User-Defined Variables

User defined variables are most relevant when using the "wizard" approach, however they are also useful when writing pure script. For example, if a user-defined variable is required in both the Metadata and Data tabs it only needs to be defined once in the Variables tab, as opposed to being defined once in each of the Metadata and Data tabs using the Scripting approach. User defined variables should follow the same variable naming standards as all other variables.

Using the Maps Tab

The Maps section of a TI process is only used when the "wizard" approach to building TI is used. However, when using the Maps section there are a number of considerations as follows.

Recreate Versus Update

There are many options within the maps tab depending on whether you are building dimensions, updating attributes, loading cubes etc. When using the "wizard" approach to building a dimension, probably the most important consideration (after the initial creation of the dimension) is whether to use Recreate or Update when performing subsequent updates of the dimension.

Using the Recreate option will remove all of the existing elements from the dimension and rebuild it from scratch. This poses a risk to data integrity within the TM1 system because the recreated dimension may no longer include some of the elements that were in the original dimension. If there

was historical data stored against any of these elements then the data will now be lost from the system. Furthermore, the loss of this data will not be logged in the TM1 transaction log; it will simply cease to exist, making it very difficult to track down the root cause of the problem in future.

For this reason, you should always use Update as opposed to Recreate as this approach preserves any historical data that may be present in any cube that contains the dimension.

For example, if the source system you are reading from has inactivated or deleted a cost centre and that cost centre contains historical data, the Recreate option will potentially result in loss of data where the Update option will not.

Accumulating Values

When using the option to "Accumulate Values" when loading cubes, you should, in almost every circumstance, zero-out the portion of the cube you are loading before the load begins. Failure to zero-out will result in double or triple-counting results if the load is run several times.

There are a small number of circumstances where this may not be the case (for example, when performing trickle loads where only changed records are being loaded).

Advanced Scripting

Before any further discussion of best practice, a quick recap of how a process executes:

- **Step 1:** Parameters (if present) are collected and bound to their relevant variables
- **Step 2:** The script in the Prolog tab is executed once before the first line of the data source is read
- **Step 3:** The data source is read starting at the first record. Note that for ODBC data sources any variables that were collected in the Parameters tab and/or any variables set up in the Prolog tab are substituted into the SQL statement at this point (if the SQL is set up to be parameter driven)
- **Step 4:** If there is code in the Metadata tab, then the code is executed against every record from the source data (first pass through source data)
- **Step 5:** If there is code in the Data tab, then the code is executed against every record from the source data (second pass through source data)
- **Step 6:** The script in the Epilog tab is executed once after the final record of the data source is read during execution of the Data tab.

Parameters

Parameters should be used to allow for writing more generic processes that can be used in a wider range of circumstances. For example a load of data from a general ledger system should be flexible enough to run for any combination of Year and Month and therefore use parameters to control this.

Parameter driven processes can be called from other processes using ExecuteProcess and supplying the relevant parameter values. This can be useful when running multiple different processes that require the same set of inputs.

For example you could have one central process that reads the current reporting month and year from a control cube and then calls each subsequent process using ExecuteProcess and supplying these values. This saves each subsequent process from having to read the control cube themselves and can even reduce locking in the process. See chapter on "The Modular Approach to Developing TurboIntegrator Processes" for more information on using parameter driven process.

Prolog

Pre-processing activities, such as creating subsets and/or views, or zeroing out portions of a cube, should be done in the Prolog.

All views and subsets required by the process should be dynamically created in the Prolog rather than using pre-created views and subsets that may be used (and therefore modified) by other users.

Metadata

As the name implies, the Metadata tab should be used to perform metadata related updates such as updates to dimension structures. The exception to this rule is the setting of attribute values which should be performed in the Data tab (an update of an attribute value is stored as a cell in a control cube after all).

Data

All data updates such as CellPutN and CellPutS must occur in the Data tab. Per the point above, updates to attribute values should occur here also.

Epilog

Post-processing activities should be done in the Epilog.

Naming Conventions

An important part of the best practice approach to developing TurboIntegrator processes is to define conventions for naming variables. There are a number of reasons to define these conventions as they aid in the ability to intuitively understand what code is doing without having to track through code. For example, consider a piece of code that derives a revenue figure from a sales quantity and a price. If there were no conventions for variables names this code may look as follows:

```
x = y * z;
```

Where x is revenue, y is quantity and z is price. This code will indeed derive the revenue figure as intended, but it won't be immediately obvious to the reader of this code that this was the intention. They would have to backtrack through the code to determine what x, y and z were intended for. Consider the following alternative:

```
revenue = quantity * price;
```

This statement performs the exact same operation; however it is now immediately obvious what the intention of the statement is. A variable name should always be self documenting; the name of the variable should be descriptive enough to clearly convey its meaning to anyone reading the code.

Furthermore, variables can be initiated from a number of different sources in a TI process. They can be parameters supplied to the TI process, they can be variables from a data source, they can be defined directly in code and they may have constant or variable values. Where a variable is defined also assists the reader in understanding the intention of the code.

Using the earlier example, what if the quantity was a variable from the data source and price was a parameter that was supplied to the process? It would be helpful to the reader to be able to determine this by looking at the name of the variable. It is fairly common practice in many languages to prefix the variable name with the variable "type", for example in Visual Basic the following is common:

```
dim intQuantity as integer
dim dblPrice as double, dblRevenue as double
```

In this case the **int** in intQuantity indicates that the variable is of type integer and the **dbl** in dblPrice and dblRevenue indicates that the variables are of type double. In TI, there are only two possible data types available: numeric or string. However, we also want to know where the variable was defined. Using a similar syntax to the VB example, our earlier TI code could be changed as follows:

```
nRevenue = vQuantity * pPrice;
```

In this example we have used a single character prefix to indicate the source of the variable or its data type. The n in nRevenue indicates that the variable contains a numeric value (as opposed to string); the v in vQuantity indicates that this variable was initiated from the Variables tab and the p in pPrice indicates that the price was initiated from the Parameters tab.

It is now clearly obvious to the reader what the statement is intended to achieve and where the variables were derived from.

Therefore, the following naming convention is recommended for best practice:

vVariableName

This naming convention is known as the Hungarian naming convention and has the following properties:

- **Prefix**
  A prefix is used to indicate the nature of the variable. The following prefixes should be used in TI:

Prefix	Description	Examples
p	**Parameter:** All variables defined in the Parameters tab should be prefixed with a "p".	pMonth pYear
v	**Variables Tab:** Any variables defined in the Variables tab should be prefixed with a "v".	vQuantity vDescription
c	**Constant:** Although TI does not specifically support constants, any variables that are set up once and never change their values should be prefixed with a "c".	cCube cVersion
n	**Numeric:** All variables that contain numeric values should be prefixed with an "n".	nRevenue nIndex
s	**String:** All variables that contain string values should be prefixed with an "s".	sDescription sAlias

- **Variable Name**
  The variable name indicates the purpose of the variable and should be descriptive without being too verbose. The variable name should start with an initial uppercase letter followed by lower case characters. If the variable name has more than one word then each word should have its initial letter capitalised.

Coding Conventions

Avoid Code Repetition

Repetition of blocks of code that perform the same thing should be avoided. Code repetition leads to additional maintenance overhead and makes code more complex than is necessary.

Use loops where possible to avoid the repetition of blocks of code. Where loops aren't appropriate, consider using a separate parameter driven process that can be called from the first process using ExecuteProcess. See paper on "The Modular Approach to Developing TurboIntegrator Processes" for more information on using separate processes to avoid code repetition.

Use Constants for Object Names

When using functions that require cube, view, dimension and subset names, define the names of the objects as constants and use the constants in the function rather than the object name. For example:

Rather than using:

```
ViewZeroOut('Ledger', 'ZeroOut');
```

Use this instead:

```
cCube = 'Ledger';
cView = 'ZeroOut';
...
ViewZeroOut(cCube, cView);
```

If you need to change the name of the object it can be done in one place rather than in every function call.

If you use a cube name in both the metadata and data tabs, define it once in the prolog tab rather than defining it in both the metadata and data tabs. Variables defined in the prolog tab persist in both the metadata and data tabs, variables defined in the metadata and data tabs persist only in their respective tabs.

Use Indentation to Make Code More Readable

When code is written without indentation is makes it difficult to easily identify where the code contained in a particular loop or branch starts and ends. For example:

```
cSubName = 'Level0';

vSubSize = SubsetGetSize(cDimName, cSubName);
vSubIndex = 1;

While(vSubIndex <= vSubSize);
If(…);
DoSomething(…);
EndIf;
End;
```

Even in this very basic example, it is difficult to follow the flow of the code. Readability is greatly improved when code within loops and branches is indented. When loops and branches are nested within other loops and branches they should be further indented. For example:

```
cSubName = 'Level0';

vSubSize = SubsetGetSize(cDimName, cSubName);
vSubIndex = 1;

While(vSubIndex <= vSubSize);
  If(…);
    DoSomething(…);
  EndIf;
End;
```

In this example it is much easier to see where the If statement starts and ends and where the While loop starts and ends. We recommend the size of the indent to be two spaces with each layer of nesting being indented an additional two spaces.

Use Comments Throughout Code

The only way for someone reading a section of code that does not contain comments to understand its purpose is to work through the code line by line, statement by statement. Even then, the code does not necessarily indicate any assumptions or thoughts that the original developer had at the time of writing the code and can be very time consuming and tedious. In order for code to be easily maintained and extended in the future, it is imperative that any subsequent developers can make sense of what was originally created.

The best way to explain the intention of a section of code is to write a comment in line with the code itself. Comments should be used to describe what the code is trying to achieve and detail any assumptions or other important thoughts that the developer had when writing the code. An important consideration is to explain the "why" e.g. why the code is constructed in the manner it is or why choices are made. It is usually possible to follow what code does but often not why.

If necessary, comments can be written across multiple lines so the developer is not restricted when explaining the code. As with most languages, comments are ignored by TI when the process is executed and won't have any adverse impact on performance of the process. Where there are major blocks of code a header comment should be used to clearly highlight its purpose.

Use appropriate spacing

Spaces should be used to make code more readable. Whitespace is ignored by the compiler so it can be used freely throughout code. Spaces should be used before and after all operators, after commas and between pipes:

Rather than using:

```
vCubeName='Ledger';
```

Use this instead:

```
vCubeName = 'Ledger';
```

Cube Logging

There is no doubt that TI processes run fastest when cube logging is turned off, however it is not always appropriate to simply turn cube logging off during a process without considering other factors. The most important consideration is to ensure that any user that is updating the cube (e.g. budget/forecast data entry) doesn't lose the logging of transactions as the transaction log is the mechanism that TM1 uses for recovering after a server crash.

While the cube locking mechanism within TM1 should mitigate the risks of the previous point, consider the following process execution:

- The process is written so that cube logging is switched off in the prolog section and switched back on in the epilog section
- The process experiences a fatal error during processing of the data section
- The epilog section of the process never runs due to the fatal error and therefore the logging is not turned back on

In this example cube logging has been inadvertently switched off. Until this is detected by the administrator, all subsequent updates will not be logged.

Another alternative is to have a chore containing three processes, the first to turn logging off, the second to perform the data load and the third to turn logging back on. This will mitigate risks of the earlier example as the third process will run even if the second one crashes. There is still some small risk of data loss, as another update could sneak in between the execution of the second and third processes and perform some un-logged updates.

The important thing when considering cube logging is to make best efforts to ensure that no data is lost in the event of a server crash. For overnight processing where there is less competition when updating cubes, it is easier to control as you can have a series of overnight processes run without logging and then turn cube logging back on as the last step.

For processing that occurs during the day, utmost care should always be taken to ensure there is no data loss in the event of a server crash.

Error Control and Debugging

Error Control

TurboIntegrator processes should be developed so that they anticipate potential errors and take steps to handle these errors as effectively as possible.

Consider a very simple parameter driven process that is to build a subset based on the caller providing a dimension name, a subset name and an element name. Assume that the parameters to this process are as follows:

```
pDimension        e.g. Product
pSubset           e.g. MyProducts
pElement          e.g. Chair
```

In the absence of any debugging or error control in the TI, the Prolog would probably look something like this:

```
# Create subset and insert element
SubsetCreate(pDimension, pSubset);
SubsetElementInsert(pDimension, pSubset, pElement, 1);
```

Now assume that this process is called by another process but the supplied parameter for the dimension name is invalid i.e. the dimension does not exist. The process would encounter an error on the SubsetCreate function call and the process would abort. This is not a particularly bad thing, as we certainly shouldn't be continuing on with the process if the dimension does not exist.

But what if the dimension name was OK but the element name was invalid i.e. the element name does not exist in the dimension? In this case the SubsetCreate statement would execute fine, but the SubsetElementInsert function would cause an error. In this case, the process would have created an empty subset with no elements.

Now assume that this sample process was just one of a chain of processes that was being executed. If the subset created by the sample process was to be used in other processes in the chain then there would be potential knock-on effects for the remaining processes.

A better way to develop this process would be to anticipate the potential errors and only create the subset and insert the element if all supplied parameters were valid. The revised code in the Prolog would then look something like this:

```
# Validate dimension exists
If(DimensionExists(pDimension) = 1);
  # Validate subset does not exist and the subset name is not blank
  If(SubsetExists(pDimension, pSubset) = 0 & pSubset @<> '');
    # Validate element exists in specified dimension
    If(DimIx(pDimension, pElement) <> 0);
      # Create subset and insert element
      SubsetCreate(pDimension, pSubset);
      SubsetElementInsert(pDimension, pSubset, pElement, 1);
    EndIf;
  EndIf;
EndIf;
```

This process will now only create the subset if all supplied parameters are valid. However, this process will now end successfully even if the supplied parameters are not valid. The intention was to only create the subset if all parameters were valid, however the process should still end in error if any of the parameters were not valid. A slight tweak to the code can achieve this:

```
# Assume parameters are invalid until proven otherwise
nValid = 0;

# Validate dimension exists
If(DimensionExists(pDimension) = 1);
  # Validate subset does not exist and the subset name is not blank
  If(SubsetExists(pDimension, pSubset) = 0 & pSubset @<> '');
    # Validate element exists in specified dimension
    If(DimIx(pDimension, pElement) <> 0);
      # Create subset and insert element
      SubsetCreate(pDimension, pSubset);
      SubsetElementInsert(pDimension, pSubset, pElement, 1);
      # Parameters are all valid
      nValid = 1;
    EndIf;
  EndIf;
EndIf;

# If any parameters weren't valid then end in error
If(nValid = 0);
  ProcessQuit;
EndIf;
```

Notice the additional variable nValid which is used to determine if all parameters are valid. If they are not all valid the process makes a call to ProcessQuit which terminates the process with an error status. There are also other in-built TI functions that can be used to control how the process terminates. Each of these processes will terminate the process slightly differently, you should choose the one that is most appropriate based on the error detected:

- **ProcessBreak**
  This function stops processing source data and proceeds to the Epilog portion of a process

- **ProcessError**
  This function causes an immediate termination of a process.
  Processes terminated with this function are flagged with an error status.

- **ProcessQuit**
  This function terminates a TurboIntegrator process.

Debugging

As with most programming languages, one of the best ways to debug a TI process that is not producing the desired outcome is to interrogate the values of the variables during process execution. In TI, this is usually done by writing the values of variables out to a text file during process execution. Debug files can be further enhanced by outputting tailored error and/or status messages.

Analysing the contents of the debug file gives the developer more information when they are trying to locate the cause of a problem. This can be particularly helpful when there are multiple processes running together either via a chore or via calls to the ExecuteProcess function.

Furthermore, the process of debugging usually only happens during initial development or when a problem is found. Therefore it is best to make the debugging capability optional during process execution.

So, returning to our earlier sample TI process for creating a subset and inserting an element, there are further improvements that can be made.

Firstly, to allow debugging to be optional during process execution an additional parameter is required to switch debug mode off or on. In our sample process we will call this parameter pDebug. When creating debug functionality there are three possible ways you may wish to run the process:

- Run the process normally with no debugging
- Run the process normally and write information out to the debug file
- Run the process and write information out to the debug file but don't perform any updates

We will assign the three options above a numeric value for simplicity. The first option will be 0, the second option will be 1 and the third option will be 2. So now our complete set of parameters is as follows:

```
pDimension        e.g. Product
pSubset           e.g. MyProducts
pElement          e.g. Chair
pDebug            e.g. 0, 1 or 2
```

The Prolog section of the process will now interrogate the pDebug parameter to determine if debug messaging is to be produced and/or if updates are to be performed. If the debug mode is 0 or 1 then updates will be performed. If the debug mode is 1 or 2 then debug messaging will be produced.

We will also modify the process to include a process start and finish time and write out the supplied parameters to the debug file. The resulting process will now look as follows:

```
# Initialise Debug
If(pDebug >= 1);
  cProcess = GetProcessName();
  cTimeStamp = TimSt(Now, '\Y\m\d\h\i\s');
  cDebugFile = GetProcessErrorFileDirectory | cProcess | '.' | cTimeStamp | '.debug';
  # Log start time
  AsciiOutput( cDebugFile, 'Process Started: ' | TimSt( Now, '\d-\m-\Y \h:\i:\s' ) );
  # Log parameters
  AsciiOutput( cDebugFile, 'Parameters: pDimension : ' | pDimension );
  AsciiOutput( cDebugFile, '            pSubset    : ' | pSubset );
  AsciiOutput( cDebugFile, '            pElement   : ' | pElement );
EndIf;

# Assume parameters are invalid until proven otherwise
nValid = 0;

# Validate dimension exists
If(DimensionExists(pDimension) = 1);
  # Validate subset does not exist and the subset name is not blank
  If(SubsetExists(pDimension, pSubset) = 0 & pSubset @<> '');
    # Validate element exists in specified dimension
    If(DimIx(pDimension, pElement) <> 0);
      # Create subset and insert element
      If(pDebug <= 1);
        SubsetCreate(pDimension, pSubset);
        SubsetElementInsert(pDimension, pSubset, pElement, 1);
      EndIf;
```

```
      # Parameters are all valid
      nValid = 1;
    ElseIf(pDebug >= 1);
      AsciiOutput(cDebugFile, 'Invalid element: ' | pElement);
    EndIf;
  ElseIf(pDebug >= 1);
    AsciiOutput(cDebugFile, 'Invalid subset: ' | pSubset);
  EndIf;
ElseIf(pDebug >= 1);
  AsciiOutput(cDebugFile, 'Invalid dimension: ' | pDimension);
EndIf;

If(pDebug >= 1);
  # Log finish time
  AsciiOutput( cDebugFile, 'Process Started: ' | TimSt( Now, '\d-\m-\Y \h:\i:\s' ) );
EndIf;

# If any parameters weren't valid then end in error
If(nValid = 0);
  ProcessQuit;
EndIf;
```

Notice that this process is now quite a lot larger than the original two line process we started with. However, we now have a process that will only create the subset if the supplied parameters are correct and the capability to analyse the execution of the process through the messaging in the debug file.