

MapleMBSE Configuration Guide

**Copyright © Maplesoft, a division of Waterloo Maple Inc.
2018**

MapleMBSE Configuration Guide

Contents

1 Overview	1
1.1 Overview of MapleMBSE Mapping	1
1.2 MSE Configuration Editor	2
1.3 Notation	2
1.4 Overview of an MSE Configuration File	3
2 Getting Started	5
3 EcoreImport	13
4 Query Path Expression	15
4.1 Query Path Expression Definition	15
5 Data Source	19
6 SyncTable Schema	21
6.1 SyncTable Schema Definition	21
6.2 Examples of SyncTable Schema	22
6.3 Mapping the attribute values of the model elements to the columns	22
6.4 Mapping the dimensions to the records	23
6.5 Alternative and Group dimensions	24
6.6 ReferenceDecomposition and ReferenceQuery	28
Mapping reference values with ReferenceDecomposition and ReferenceQuery	28
ReferenceDecomposition by Example	28
References by Dimensions or ReferenceQuery	30
6.7 Key Columns defined in SyncTable Schema	31
7 SyncTable	33
8 Laying out SyncViews	35
8.1 Setting up a Workbook and Worksheets	35
8.2 Worksheet template and View Layout	36
Table View Layout	36
Matrix View Layout	38

List of Figures

Figure 5.1: Relationship between model elements	19
Figure 6.1: SimpleTree	22
Figure 6.2: SyncTable From Simple Tree	23
Figure 6.3: SyncTable From Simple Tree (add record keyword to top level Dimension	24
Figure 6.4: Simplified Model Number Two: Using Alternative and Group	24
Figure 6.5: Tree From Simplified Model Two	25
Figure 6.6: Table Made From The Tree of Simplified Model Two	26
Figure 6.7: Another Tree Made From Simplified Model Two	27
Figure 6.8: Another Table Made From Simplified Model Two	27
Figure 6.9: Target Model	30
Figure 6.10: Illustration of ReferenceDecomposition	30

1 Overview

In this chapter:

- *Overview of MapleMBSE Mapping (page 1)*
- *Overview of an MSE Configuration File (page 3)*

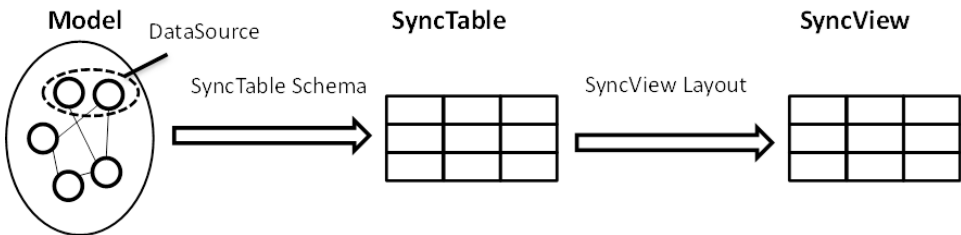
1.1 Overview of MapleMBSE Mapping

Mapping model information from diagram-based model form into table form requires a two step process.

First, a **SyncTable Schema** must be defined to convert the model to an intermediate table structure called a **SyncTable**.

A SyncTable Schema specifies how to find objects in a model starting with an object given by a **DataSource**. A pair of a DataSource and a SyncTable Schema defines one SyncTable.

Next, the **SyncView Layout** must be defined for how the SyncTable is displayed on a spreadsheet by specifying a layout and which columns of the SyncTable to include or omit. The resulting part of the spreadsheet displaying the SyncTable is called **SyncView**. The schematic flow of displaying a model in an Excel spreadsheet is shown in the figure below.



The definition of a DataSource, a SyncTable Schema, and a SyncView Layout is called an **MSE configuration**. The language used to define an MSE configuration is called MSE configuration language. In this guide we provide the specification of the MSE configuration language. For notation used in the specification, see *Notation (page 2)*. MSE configuration files are text files that can be edited and created with any text editor. However, it is recommended to use MSE Configuration Editor which provides convenient syntax highlighting and checking. For the installation instructions, see *MSE Configuration Editor (page 2)*. Examples in this guide use MSE Configuration Editor.

1.2 MSE Configuration Editor

MapleMBSE Configuration Editor (a.k.a MSE Editor) is provided in the same package as MapleMBSE-Editor_<VERSION>.zip.

The MSE Editor is an Eclipse add-on, and you can install with the following steps:

1. Launch Eclipse (Oxygen)
2. Select **Help**, then **Install New Software**
3. Click **Add** to display the Add Repository window.
4. In the Add Repository widow click **Local**
5. Select MapleMBSE-Editor_2018.0.zip then click **OK**.
6. Select "MapleMBSE MSE Editor" and then follow the instructions shown in the dialog.

To use the editor, you need to add an MSE file to your Eclipse workspace. Double-clicking an MSE file in the workspace launches the editor.

1.3 Notation

The formal grammar of MSE Configuration Language is given using a simple Extended Backus-Naur Form (EBNF) notation. Each rule in the grammar defines one symbol, in the form:

`symbol ::= expression`

The following notations are used in expressions.

Notation	Usage
'string'	literal string matching the string between the quotes
(expression)	expression is treated as a unit
A*	0 or more occurrences of A
A+	1 or more occurrences of A
A?	0 or 1 occurrence of A
A B	A or B
<A>	name of an element of type A

For reference see <https://www.w3.org/TR/2008/REC-xml-20081126/#sec-notation>

1.4 Overview of an MSE Configuration File

The following is the formal definition of the configuration file.

```
MSEConfiguration ::= EcoreImport*
```

```
WorkbookInstance &  
  ( DataSource  
    | SyncTableSchema  
    | SyncTable  
    | WorksheetTemplate  
  ) *
```

In MSEConfiguration, EcoreImports come first, and then other elements can be specified in any order. The definitions of the elements are given in the following chapters. The following is an example of the procedure for writing MSE Configuration file.

1. Define a Data Source and a SyncTable Schema.
2. Define a SyncTable with the pair of Data Source and SyncTable Schema.
3. Define the view and the layout of the SyncTable on WorksheetTemplate.
4. Define a worksheet in the WorkbookInstance with the pair of the WorksheetTemplate and SyncTable.

2 Getting Started

The goal of this section is to introduce the elements of the configuration and template files and how they are connected together by defining a simple configuration file. The details about the elements are given in the following sections.

A configuration file defines what data from a model is accessible and how it is presented in Excel. In order to do that, the configuration file must define the following elements.

- The content of the Excel workbook: how many and what types of worksheets it has.
- For each worksheet, define the area that is associated with the model data - the SyncView area and how it is displayed.
- For each SyncView area define what model data is displayed.

In this example, we want to define a configuration that allows us to view and update top-level packages in a UML model. The first step is to import the definition of a UML metamodel. A metamodel, called Ecore, defines types of elements a UML model may have and their relationship. The definitions inside the configuration file that allow us to access different elements of a model rely on the structures defined by the imported metamodels. To import an Ecore metamodel, use an **EcoreImport** construct as follows.

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML"
```

There are two steps in converting model data into its representation in Excel. First, we define a **SyncTable Schema** that converts the data into an intermediate table called a **SyncTable**. In the second step, the **SyncView Layout** defines how a SyncTable is displayed on a spreadsheet by specifying which SyncTable columns will be displayed, as well as their placement and the layout. The resulting part of the spreadsheet displaying the SyncTable is called **SyncView**.

A SyncTable Schema defines how a set of model elements is mapped to a table structure. A basic structure of a SyncTable schema is a dimension. Each dimension corresponds to a model element. The first dimension of a SyncTable Schema is a Top Level Dimension. It represents the type of element to which the schema applies. Each following dimension is defined with respect to the preceding one. In this example, we define a SyncTable Schema called PackagesTable.

```
synctable-schema PackagesTable {
```

```
|  
}
```

Note the configuration file editor performs some testing of the correctness of the defined structures. The syntax error highlighting the closing bracket indicates that definition is incomplete without defining a dimension.

We define the top level dimension to be an element of a Package type.

```
synctable-schema PackagesTable {  
  dim [Package]  
}
```

A dimension consists of columns. Each column represents an attribute of the element that the dimension describes. To identify the element some of the columns must be designated as key columns. They must represent the attributes of the element that would allow you to identify it uniquely. Without the definition of the key column(s) the definition of the dimension is incomplete. It is indicated by a syntax error.

For a package, its name can identify it uniquely. We define a key column that corresponds to the 'name' attribute of a Package class.

```
synctable-schema PackagesTable {  
  dim [Package] {  
    key column /name as PackageName  
  }  
}
```

This SyncTable schema definition allows you to view, add and delete packages by referring to their name. To create a SyncTable, the schema must be applied to a **Data Source**. A Data Source defines a set of model elements. The Data Source representing the top-level data structure of a model has the name **Root**. This is a reserved name. The Root is declared as follows.

```
data-source Root[Model]
```

The declaration specifies that the type of the top-level data structure is Model. You can see the types and the structure of a UML model by opening the .uml file in a text editor. For example, the following is a snippet from UserGuide.uml in the installer, found in the **<MapleMBSE>\Examples\UserGuide** directory, where **<MapleMBSE>** is your MapleMBSE installation directory.

```
<?xml version="1.0" encoding="UTF-8"?>  
<uml:Model xmi:version="20131001"  
  xmlns:xmi="http://www.omg.org/spec/XMI/20131001"  
  xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML"  
  xmi:id="_D2UUEM_MEee6666BhKb4Cg" name="UserGuide">  
    <packagedElement xmi:type="uml:Package" xmi:id="_Oeqy0M_MEee6666BhKb4Cg"  
      name="Package1" visibility="public">  
      ...  
    </packagedElement>  
    <packagedElement xmi:type="uml:Association" ...>  
    ...  
  </packagedElement>
```

```
...
</uml:Model>
```

The text representation of the model is written in XML. The model and its content are represented by XML elements. The top-level element is defined by the start and end tags:

```
<uml:Model ...>
...
</uml:Model>
```

The element is a "uml:Model" that is of a type Model defined by the "uml" namespace. The "uml" namespace is defined among the attributes of the Model element.

```
xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML"
```

The definition matches the EcoreImport we are using in the configuration file. So the type "Model" used in the definition of the Root Data Source is the same as "uml:Model" in the model file. We want to apply the PackageTable schema to define packages inside a model, and the type of the data source it applies to is Package. We define a data source that represents packages in a model as follows.

```
data-source topPackages=Root/packagedElement[Package]
```

Looking at the sample model we see that elements inside the Model element are defined by the tag packagedElement. They can be of different types, such as Package or Association. The defined data source is called topPackages. It chooses structural elements of type Package inside the Root. These structural elements are collectively called packagedElement.

To create a SyncTable we apply the SyncTable schema to the Data Source.

```
synctable packagesTable = PackagesTable<topPackages>
```

The next step is to define how the SyncTable is represented in an Excel worksheet. We do this by defining a **Worksheet Template**. We define a Worksheet Template called Packages.

```
worksheet-template Packages|
```

The template uses one argument p of type PackagesTable.

```
worksheet-template Packages(p : PackagesTable) {
|
}
```

The Worksheet Template must define where the SyncView for the given argument is placed and what orientation it has. In this example we choose the SyncView for the argument p to be a vertical table (named tab1) and start in cell B3 (row 3, column 2). The section defining tab1 is called **SyncView Layout**.

```
worksheet-template Packages(p : PackagesTable) {  
  vertical table tab1 at (3,2) = p {  
    |  
  }  
}
```

We also need to specify which columns of the SyncTable should be included in the SyncView. A SyncTable column becomes a field in a SyncView record. For the definitions of records and fields see the **Operations Overview** section in Chapter 2 of the **MapleMBSE User Guide**. Some fields in a record must be marked as key fields to indicate that those fields are used to identify the record uniquely. In the PackageTables schema there is only one column, PackageName. It is a key column and should be used as a key field. PackageName is of the String data type. The definition of the SyncView layout is then as follows.

```
worksheet-template Packages(p : PackagesTable) {  
  vertical table tab1 at (3,2) = p {  
    key field PackageName : String  
  }  
}
```

We also want to indicate that the column should be sorted in ascending order when a model data is loaded or when sort operation is performed after adding new data. We do so by specifying the name of the field in the sort keys.

```
worksheet-template Packages(p : PackagesTable) {  
  vertical table tab1 at (3,2) = p {  
    key field PackageName : String  
    sort-keys PackageName  
  }  
}
```

Finally, we need to define a workbook that consists of a worksheet based on the defined template applied to an instance of a SyncTable.

```
workbook {  
  worksheet Packages(packagesTable)  
}
```

The final content of the cofiguration file is as follows.

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML"

synctable-schema PackagesTable {
    dim [Package] {
        key column /name as PackageName
    }
}

data-source Root[Model]
data-source topPackages=Root/packagedElement[Package]

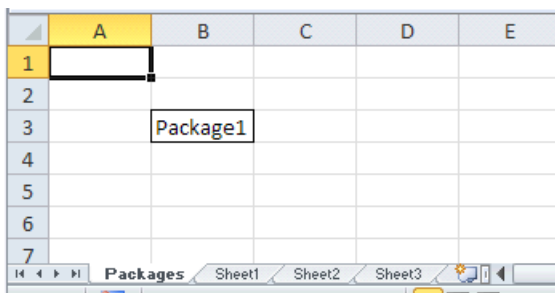
synctable packagesTable = PackagesTable<topPackages>

worksheet-template Packages(p : PackagesTable) {
    vertical table tab1 at (3,2) = p {
        key field PackageName : String
        sort-keys PackageName
    }
}

workbook {
    worksheet Packages(packagesTable)
}
```

The resulting file can be found in GettingStarted.MSE, in the MapleMBSE Configuration Editor Package.

You can use the configuration file with any UML model. For example, opening MapleMBSE with this configuration file and **<MapleMBSE>\Example\UserGuide\UserGuide.uml**, where **<MapleMBSE>** is the location where MapleMBSE is installed, gives the following result.



	A	B	C	D	E
1					
2					
3		Package1			
4					
5					
6					
7					

The SyncView area of the worksheet can be highlighted by choosing the name of the corresponding SyncView in the name box. The SyncView name has the following format.

MapleMBSE_SyncView_<Worksheet Name>_<SyncView Layout Name>

_MapleMBSE_SyncView_Packages_tab1							
	A	B	C	D	E	F	G
1							
2							
3		Package1					
4							
5							
6							

You can add new packages to the model by adding rows in the SyncView area or by entering them in the insertion area (cell B4). See the *MapleMBSE User Guide, Chapter 2, Adding Model Elements* for more details.

For convenience, it is good to add a heading to the column explaining what it is and maybe change the width of the column. Any such formatting changes done when editing a model are not saved with the model data. Instead, they should be done in a separate file called **Template File**. A template file is an Excel file that has the same base name as the configuration file and is placed in the same folder. MapleMBSE looks for the sheets in the workbook that match the names of the worksheets defined by the configuration file and loads the specified SyncViews into that sheet. To define a template file for our example, we need to create an Excel file with the name that matches the name of the configuration file and contains a sheet called Packages.

Tip: the template file for this example can be found in GettingStarted.xls, in the same place with GettingStarted.MSE.

We define a new Excel workbook. We name one of the sheets Packages, and delete others. The data with the package name is displayed in column B starting with row 3. We can define the heading for the column in cell B2 and increase the width of the column.

	A	B	C	D
1				
2		Package Name		
3				
4				
5				
6				

Excel status bar: Packages

We save the template file with the same base name as the configuration file and in the same folder. Now if we open MapleMBSE with the configuration file and the example model UserGuide.uml we get the following.

	A	B	C	D
1				
2		Package Name		
3		Package1		
4				
5				
6				

Navigation icons: Home, Insert, Page Layout, Formulas, Data, Review, View, Send, Print, Help, Packages

Another way to create a template could be to open a model with the configuration file as we did before, then save it as an Excel file using **Add-Ins > MapleMBSE > Export To Excel File**. This way we have the right number of sheets with their names. It is also easier to judge where the headings need to be added and how wide the columns should be. Any model data loaded in the tables should be removed. If it is left in the template it may create confusion when MapleMBSE uses the template. MapleMBSE will load SyncViews according to the specifications in the configuration file, so some data may be overwritten and some may not, depending on the model file with which the template is opened.

3 EcoreImport

EcoreImport declares the type of model to be edited with the configuration file. A type of model is defined by specifying an IRI of a metamodel definition. A metamodel, called Ecore, defines types of elements a model may have and their relationship. Model elements and their attributes are queried using the structural elements defined by EcoreImports. The formal syntax of EcoreImport declaration is as follows.

```
EcoreImport ::= 'import-ecore' '''IRI''' (as ID)
```

where IRI is an identifier of the Ecore metamodel in the form of IRI (International Resource Identifier). Different types of models have their own metamodels. The following is a list of the available Ecore models.

Type	IRI
UML	http://www.eclipse.org/uml2/4.0.0/UML
SysML	http://www.eclipse.org/papyrus/sysml/1.4/SysML
Teamwork Cloud	http://www.nomagic.com/magicdraw/UML/2.5
Rhapsody	http://w3.ibm.com/Rhapsody/api/
MapleMBSE metamodel	http://maplembse.maplesoft.com/common/1.0

4 Query Path Expression

4.1 Query Path Expression Definition

Query Path Expression is an expression that queries the model for model elements and attribute values. It is used in defining Data Sources and SyncTable Schemas. The formal syntax definition is as follows.

```
QueryPathExpression ::= ( LocalQueryExpression )+ ( '@' ReferenceDecompositionId )?
```

```
LocalQueryExpression ::= ( ( '/' AttributeId ) | ( '.' ReferenceId ) ) Qualifier?
```

```
AttributeId ::= ( EcoreImportId '::' )? <Attribute>
```

```
ReferenceId ::= ( EcoreImportId '::' )? <Reference>
```

```
Qualifier ::= '[' ClassifierId ( '|' AttributeFilter ( ',' AttributeFilter)* )? ']'
```

```
ClassifierId ::= ( EcoreImportId '::' )? <Classifier>
```

```
AttributeFilter ::= AttributeId '=' ' "' <Expression> ' "'
```

ReferenceDecompositionId refers to ID of a ReferenceDecomposition defined in *ReferenceDecomposition and ReferenceQuery* (page 28).

<Classifier>, <Attribute>, <Reference> mean names of elements of the corresponding types. The names and their types are defined by a metamodel (via EcoreImport). In Query Path Expressions we distinguish the following three types.

- **Classifier**

A type of an element. For example, a UML model may have elements of type Class.

Class is a classifier. An element contains subelements which can be of two types: attributes and references.

- **Attribute**

A subelement that belongs to the element.

- **Reference**

A subelement that refers to another element.

To illustrate these types and their relations consider the example code in the Figure below. The code is a snippet from the UML example model from the MapleMBSE User Guide.

Tip: The model file, `UserGuide.uml`, can be opened using any text editor. It can be found in the installation folder `<MapleMBSE>/Example/UserGuide`, where `<MapleMBSE>` is the location of your MapleMBSE installation.

```
...
<packagedElement xmi:type="uml:Class" xmi:id="_vXfqAM_MEee6666BhKb4Cg"
name="Class1">
  <ownedAttribute xmi:id=" _AH4akdBoEee6666BhKb4Cg" name="Property1"
visibility="protected" type="tsYvoNBnEee6666BhKb4Cg" aggregation="shared"
association=" _AH3McNB0Eee6666BhKb4Cg">
    <lowerValue xmi:type="uml:LiteralInteger"
xmi:id=" _AH4aktBoEee6666BhKb4Cg" value="1"/>
    <upperValue xmi:type="uml:LiteralUnlimitedNatural"
xmi:id=" _AH5BoNB0Eee6666BhKb4Cg" value="1"/>
  </ownedAttribute>
  ...
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="tsYvoNBnEee6666BhKb4Cg"
name="Class2" visibility="private"/>
...
<packagedElement xmi:type="uml:Association"
xmi:id=" _AH3McNB0Eee6666BhKb4Cg" name="aggregation_class2_in_class1"...>
  ...
</packagedElement>
...
```

The text representation of the model is written in XML. The model and its content are represented by XML elements. An element can be defined as an empty element with attributes.

```
<element ... />
```

Or if it contains other elements it can be defined using the start and end tags.

```
<element> ... </element>
```

The classifiers are highlighted in blue: "uml:Class", "uml:LiteralInteger", "uml:LiteralUnlimitedInteger", "uml:Association". The "uml" namespace is defined in the definition of the Model element, see *Getting Started (page 5)*. Consider the ownedAttribute element Property1 in Class1. The attributes of the element are highlighted in green: name, visibility, aggregation, lowerValue, upperValue. The references of Property1 are highlighted in orange: type and association. You can see that the values of the references are the IDs of the elements they refer to.

The names of Classifiers, Attributes, and References can be written with or without `EcoreImportId` depending on how `EcoreImport` was declared. If there is only one `EcoreImport` in a configuration file and it was declared without an ID:

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML"
```

`EcoreImportId` is not necessary. In this case, a query path expression that queries elements of a package can be written as follows.

```
/packagedElement
```

If an `EcoreImport` was declared with an ID:

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML" as uml
```

`EcoreImportId` must be used to refer to classifiers, attributes, or references defined by the corresponding model. The same Query Path Expression takes the form.

```
/uml::packagedElement
```

In the following examples we omit `EcoreImportId`. The above examples of Query Path Expressions query all elements in a package. For the example in the Figure above, it would include elements of types `Class` and `Association`. If we want to specify that only elements of `Class` type should be queried we need to specify a `Qualifier`:

```
/packagedElement[Class]
```

A `qualifier` can include one or more `FeatureFilters`. For example, to query a class inside a package called `Class1`, the following Query Path Expression can be used.

```
/packagedElement[Class|name="Class1"]
```

The examples we have considered so far consisted of single `LocalQueryExpressions`. `LocalQueryExpressions` can be combined to query nested objects. Each subsequent `LocalQueryExpression` applies to the result of the previous `LocalQueryExpression`. For example, to query attributes (the `ownedAttribute` elements) inside `Class1` inside a package, the following Query Path Expression can be used.

```
/packagedElement[Class|name="Class1"]/ownedAttribute
```

So far, we have only used attributes in query expressions. To query a type of an `ownedAttribute` in a class a reference must be used.

```
/packagedElement[Class|name="Class1"]/ownedAttribute.type
```

The result of the query is the element that the 'type' reference refers to. For `Property1`, it would return class `Class2`. Another way to specify a reference is to add the specification of `ReferenceDecomposition` at the end of the Query Path Expression.

```
/packagedElement[Class|name="Class1"]/ownedAttribute.type @  
ReferenceDecompositionId
```

`ReferenceDecomposition` is defined in Chapter 6, see *ReferenceDecomposition and ReferenceQuery* (page 28). `ReferenceDecomposition` is a description of the referenced object. For display purposes there is no difference between a reference query with and without the use of `ReferenceDecomposition`. However, when updating a field specified by a reference without a `ReferenceDecomposition`, the updates apply to the referenced object. Whereas, with a `ReferenceDecomposition` the updates may change which object the reference

points to. It is not recommended to use references without `ReferenceDecompositions`. If necessary, they should only be used in read-only worksheets.

5 Data Source

Data Source defines a set of model elements. A Data Source is combined with a SyncTable Schema to create a SyncTable for the model elements defined by the Data Source. The following is the formal definition of Data Source.

```
DataSource ::= PrimaryDataSource | ChainedDataSource
PrimaryDataSource ::= 'data-source' 'Root' Qualifier
ChainedDataSource ::= 'data-source' ID '=' DataSource ObjectQueryExpression
```

Root is a reserved Data Source name that refers to the top-level model element. The type of the top-level model element depends on the type of a model. The following are definitions of the Root Data Source based on the type of model.

- UML, SysML, Teamwork Cloud

```
data-source Root[Model]
```

- Rhapsody

```
data-source Root[REProject]
```

A ChainedDataSource applies Query Path Expression to the result of the parent Data Source. Consider the example in the Figure below based on UserGuide.uml model (found in <MapleMBSE>/Example/UserGuide, where <MapleMBSE> is the MapleMBSE installation directory). The Figure below shows the relationship between the elements. For each element its classifier is given in *italics*. The elements enclosed in boxes with dashed lines are included in the corresponding data sources defined below.

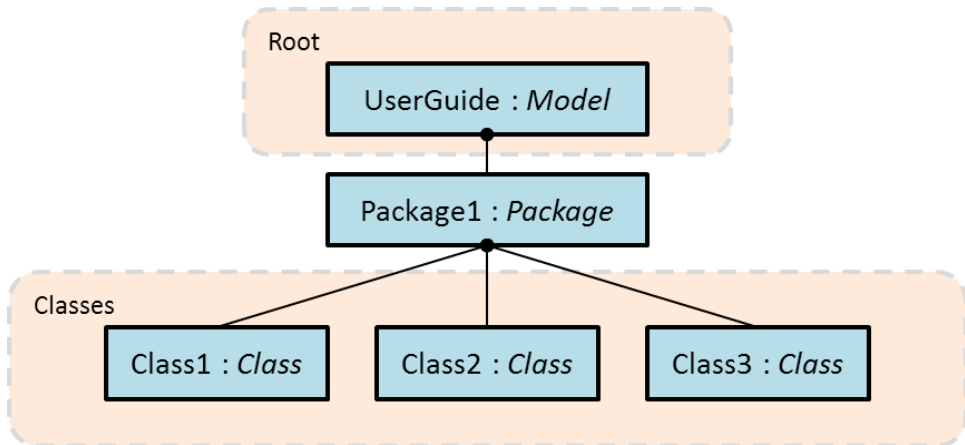


Figure 5.1: Relationship between model elements

- **Primary Data Source**

The example is a UML model, so the Primary Data Source is defined as follows.

```
data-source Root[Model]
```

- **Chained Data Source**

The following Data Source defines a set of all classes in Package1. The Data Source called "classes" is defined by applying an Query Path Expression to a previously defined data source (Root, in this

```
data-source classes = Root/packagedElement[Package|name="Package1"]/packagedElement[Class]
```

6 SyncTable Schema

6.1 SyncTable Schema Definition

SyncTable schema specifies how model elements are mapped to a logical table. With data sources explained in Chapter 5, model elements are first organized as trees, and then mapped to tables. Such tree nodes are defined by *dimensions* in SyncTable schema, which identifies a model element by key columns. The formal syntax of SyncTable Schema is defined as:

```
SyncTableSchema ::= 'syncable-schema' ID ( '(' SyncTableParam ( ',' SyncTableParam ) *  
' ' ) ) ?  
'{' TopLevelDimension  
AbstractDimension*  
}'
```

```
SyncTableParam :: ID ':' SyncTableSchemaId
```

```
TopLevelDimension ::= ('record')? 'dim' Qualifier '{' DimensionMember* '}'
```

```
AbstractDimension ::= SuccessiveDimension | DimensionGroup
```

```
SuccessiveDimension ::= ('record')? dim QueryPathExpression '{' DimensionMember* '}'
```

```
DimensionGroup ::= ('alternative'|'optional'|'group') '{' DimensionMember* '}'
```

```
DimensionMember ::= PropertyMapping | ReferenceDecomposition
```

```
PropertyMapping ::= AttributeColumn | ReferenceQuery
```

```
AttributeColumn ::= KeyAttributeColumn | NonkeyAttributeColumn
```

```
KeyAttributeColumn ::= 'key' 'column' ObjectQueryExpression 'as' ID
```

```
NonkeyAttributeColumn ::= 'column' ObjectQueryExpression 'as' ID
```

where `SyncTableSchemaId` is ID of a `SyncTableSchema`, and `TopLevelDimension` appears first as defined in the formal syntax, and we need to put a qualifier to specify what model element types are selected, then `SuccessiveDimension` follows in which we put a Query Path Expression to query what model elements are selected as dimensions. In the later sections, we explain how to specify SyncTable schemas by examples.

6.2 Examples of SyncTable Schema

First, we show a simple SyncTable Schema as follows:

```
synctable-schema PkgCls {  
  dim [REPackage] {  
    key column /name as PkgName  
    column /description as PkgDesc  
  }  
  record dim /nestedElements[REClass] {  
    Key column /name as ClsName  
  }  
}
```

Here we define a SyncTable Schema with an ID called PkgCls and it consists of two Dimensions. '[REPackage]' in the top level dimension means it picks up REPackage model elements, and it must be consistent with that in data sources. The next dimension picks up REClass elements in nestedElements feature of the top level dimension. By applying this schema to Pkg 1, Pkg 2 of the data source having **Figure 5.1**, we obtain two trees as shown in **Figure 6.1**, where Pkg1 and Pkg2 belong to the top level dimensions; and Cls1 and Cls2 belong to the next dimensions.

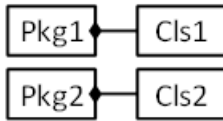


Figure 6.1: SimpleTree

6.3 Mapping the attribute values of the model elements to the columns

The trees in the example above are translated into tables by the column definitions. The top level dimension has PkgName and PkgDesc columns, and they are filled with the QPEs of

"/name" and "/description", respectively. And the next dimension have ClsName column, which is filled with the QPE of "/name". Then the tree in **Figure 6.1** is translated to:

Pkg1	PkgDesc1	Cls1
Pkg2	PkgDesc2	Cls2

Figure 6.2: SyncTable From Simple Tree

More formally speaking, each path in the trees is translated into record, and then we have two records from the paths of Pkg1-Cls1 and Pkg2-Cls2. Note that synctable schema determines all of the columns in a static way. They are, in this example, PkgName, PkgDesc, and ClsName, and the number is three.

6.4 Mapping the dimensions to the records

Let us look at how dimensions are mapped to records in more detail by comparing with the example below. The only difference from the previous example is the **record** keyword in the top level dimension highlighted with bold font.

```
synctable-schema PkgCls2 {
  record dim [REPackage] {
    key column /name as PkgName
    column /description as PkgDesc
  }
  record dim /nestedElements[REClass] {
    key column /name as ClsName
  }
}
```

If any other conditions are the same as the above, the trees generated by this schema are exactly the same as in **Figure 6.1**. However, because the top level dimension has a record keyword, the table has more records as shown in **Figure 6.3**. The added records are the first and third rows, which come from the top level dimension. Note that the last dimension (in this example, that is the one corresponding to Cls) always creates records even if it is missing. In this table, the rightmost column in the first and third rows is specially treated as EMPTY. They will be shown as blank cells with light gray backgrounds, and distinguished from the usual blank cells

Pkg1	PkgDesc1	
Pkg1	PkgDesc1	Cls1
Pkg2	PkgDesc2	
Pkg2	PkgDesc2	Cls2

Figure 6.3: SyncTable From Simple Tree (add record keyword to top level Dimension

Note that each record corresponds to one model element. In this example, the first record corresponds to Pkg1, and the second one corresponds to Cls1 while the previous example does not have any records corresponding to Pkg1 nor Pkg2. Therefore, in this example you can add or delete packages by adding or removing a row while in the previous example you cannot. In this sense, record keyword plays a vital role that determines which model elements can be added or deleted by users.

6.5 Alternative and Group dimensions

Next we move on to how to organize tree structures by using the following example model. For the sake of simplicity, we denote model elements with lowercases with numbers (e.g. a1) and its types with uppercases (e.g A) in this example.

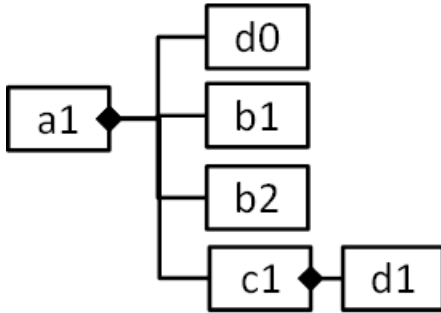


Figure 6.4: Simplified Model Number Two: Using Alternative and Group

Let us consider the following configuration:

```

synctable-schema alternativeExample {
  dim [A] {
    key column /name as Aname
  }
  alternative {

```

```
record dim /nestedElements[B] {  
key column /name as Bname  
}  
record dim /nestedElements[C] {  
key column /name as Cname  
}  
record dim /nestedElements[D] {  
key column /name as Dname  
}  
}  
}  
}
```

It generates a tree as show in Figure 12**Figure 6.5**.

The top level dimension selects type A by [A], and then the root of the tree is a1. In the following dimensions, it selects /nestedElements[B], /nestedElements[C], or /nestedElements[D] because these are in alternative { ... } clause. That means that if /nestedElements[B] is matched, the second dimension is used; if /nestedElements[C] is matched, the third dimension is used; and if /nestedElements[D] is matched, the forth dimesion is used. Therefore, d0, the first model element in the nestedElements feature, is applied to the forth dimension; b1 and b2 are applied to the third dimension; and c1 is applied to the forth dimension. And then, we obtain a tree shown in Figure 12**Figure 6.5**.

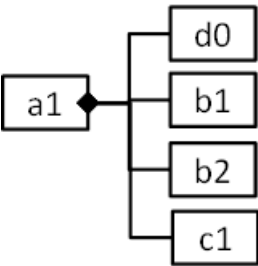


Figure 6.5: Tree From Simplified Model Two

This tree will be turned into a table as shown in Figure 13 **Figure 6.6**. It has four columns consisting of Aname, Bname, Cname, and Dname. Since the top level dimension does not have "record" keyword, it does not have a record of "a1". Instead it creates four records for "b1", "b2", "c1", and "d0" corresponding to the tree nodes under "a1" in Figure12**Figure 6.5**. Notice that the record for "c1" in the third row fills Aname and Cname columns, and Bname column is specially treated as VOID, which looks blank but filled with thick gray background. Likewise, the second (Bname) and third (Cname) columns in the forth row are also filled with VOID.

a1	b1		
a1	b2		
a1		c1	
a1			d0

Figure 6.6: Table Made From The Tree of Simplified Model Two

Let us move on to the next example using "group" as shown below:

```
synctable-schema groupExample {  
  dim [A] {  
    key column /name as Aname  
  }  
  alternative {  
    record dim /nestedElements[B] {  
      key column /name as Bname  
    }  
    group {  
      dim /nestedElements[C] {  
        key column /name as Cname  
      }  
      record dim /nestedElements[D] {  
        key column /name as Dname  
      }  
    }  
  }  
}
```

It generates a tree as shown in Figure 14 **Figure 6.7**.

The difference is that now "d1" belongs to "c1" instead of "a1" because the above configuration says B or C followed by D rather than B, C, or D. It means something like (B or (C, D)) in contrast with (B or C or D). That is, group keyword is something like parentheses in dimension definitions and alternative is like "or" operator.

Then this tree is translated to a table as shown in Figure 15**Figure 6.8**. Since the dimensions of B and D have a record keyword, it creates three records: "b1", "b2", and "d1", corresponding to the first, second, and third rows. The third and fourth columns that follow after "b1" and "b2", in the first and second rows, are EMPTY and the second column in the third row is VOID in this table.

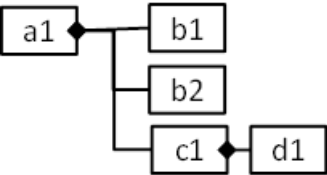


Figure 6.7: Another Tree Made From Simplified Model Two

a1	b1		
a1	b2		
a1		c1	d1

Figure 6.8: Another Table Made From Simplified Model Two

Since group keyword combines dimensions in alternative blocks, using it out of alternative does not give any effects. For example,

```
dim ... {}  
dim ... {}  
dim ... {}
```

and

```
dim ... {}  
group {  
  dim ... {}  
  dim ... {}  
}
```

give the same results.

6.6 ReferenceDecomposition and ReferenceQuery

Mapping reference values with ReferenceDecomposition and ReferenceQuery

ReferenceDecomposition is used for presenting references of model elements. The examples so far edit model elements themselves by querying them with QPEs, where we can track references as well. That means we always change values of such model elements instead of references to model elements.

First we specify the formal syntaxes of ReferenceDecomposition and ReferenceQuery as below:

```
ReferenceQuery ::= ('key')? 'reference-query' ObjectQueryPath
ReferenceDecomposition ::=
'reference-decomposition' ID '=' [ReferrableSyncTable] '{' ForeignColumn* '}'
ForeignColumn ::= KeyForeignColumn | NonkeyForeignColumn
KeyForeignColumn ::= 'foreign-key' 'column' [Column] 'as' ID
NonkeyForeignColumn ::= 'foreign' 'column' [Column] 'as' ID
```

In ReferenceDecomposition, you should specify all of the key columns in the referred table as KeyForeignColumn (that is, you should specify "foreign-key" for such key columns) because we should identify a record by such key columns. If the configuration does not satisfy this condition, it is not guaranteed to identify a unique record to make a reference.

ReferenceDecomposition by Example

We use the following configuration to explain ReferenceDecomposition.

```
import-ecore "http://w3.ibm.com/Rhapsody/api/"

data-source Root [REProject]
data-source Pkgs =
Root/nestedElements[REPackage|name="Structure"]/nestedElements[REPackage]
data-source Types =
Root/nestedElements[REPackage|name="Units"]/nestedElements[REPackage]

synctable attributes = AttsByName<Pkgs>(types)
synctable types = TypesByName<Types>

synctable-schema TypesByName {
    dim [REPackage] {
        key column /name as TypeName
    }
}
```

```

}

synctable-schema AttsByName(tps:TypesByName) {
  dim [REPackage] {
    key column /name as PkgName
  }
  dim /nestedElements[REClass] {
    key column /name as ClsName
  }
  dim /nestedElements[REAttribute] {
    key column /name as AttName
    reference-query .type @ typedecomp
    reference-decomposition typedecomp = tps {
      foreign-key column TypeName as Type
    }
  }
}

```

This configuration transforms the target model in Figure 16 **Figure 6.9** into a table as shown in Figure 17 **Figure 6.10**.

This example first introduces TypesByName synctable-schema, which itemize all of the types as "TypeName", and AttsByName refers to that type by "type" feature of "REAttribute". Note that AttsByName takes "tps" argument of "TypesByName", and in Line 7, "attributes" synctable takes "types" as an argument and then "attributes" synctable uses "types" synctable to refer to types by the ReferenceDecomposition in Lines 25-28. Let us look into these in the following section.

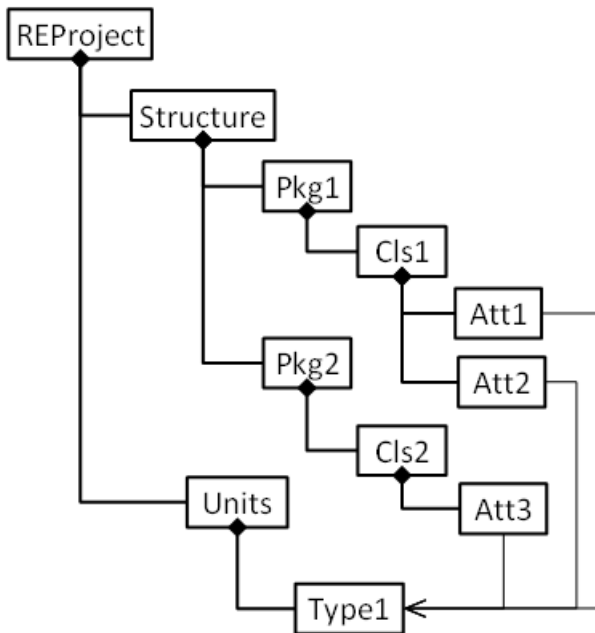


Figure 6.9: Target Model

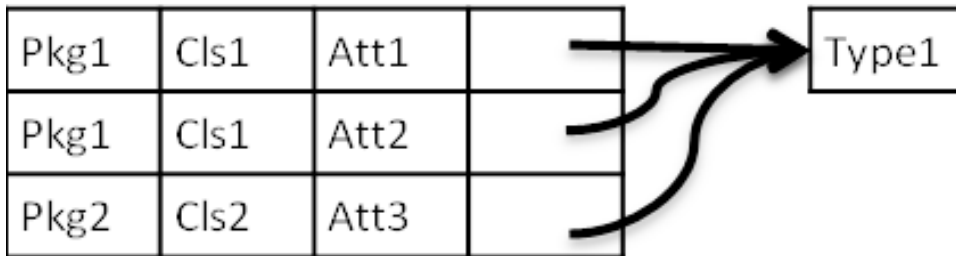


Figure 6.10: Illustration of ReferenceDecomposition

References by Dimensions or ReferenceQuery

Next, let us see how we identify references. As show in Line 25 of the previous section, we write "reference-query *QPE @ name*" in the dimension. Let us look in the part in the example of the previous section:

```
dim /nestedElements[REAttribute] {
key column /name as AttName
```

```
reference-query .type @ typedecomp
reference-decomposition typedecomp = tps {
foreign-key column TypeName as Type
}
}
```

In this example, we use the "type" feature of "REAttribute" as a reference to be decomposed. Thus, this reference refers to a type identified by the "TypeName" column of "tps" table. This dimension has "AttName" and "Type" columns and "AttName" column is associated with "name" feature of "REAttribute" of this dimension, and "Type" column is used to refer to "type" (see the reference-query) by "TypeName" column of "tps" table.

Otherwise, if the reference is associated with a dimension, we put "@ name" after the dimension definition as the example below:

```
dim [REInstance] { ... }
dim .otherClass[REClass] @ cls2 {
reference-decomposition cls2 = clsTbl {
foreign-key column PkgName as PkgName2
foreign-key column ClsName as ClsName2
}
column /description as ClsDesc2
}
```

where we use "cls2" as the name of the reference. And in the following "reference-decomposition cls2", we use "PkgName" and "ClsName" columns of "clsTbl" to present that reference. Therefore, this dimension has "otherClass" reference of REInstance (in the previous dimension), which refers to "REClass" class identified by "PkgName" (propagated by "PkgName2" column of this dimension) and "ClsName" (propagated by "ClsName2" column, likewise) columns of "clsTbl". Note that "clsTbl" is a parameter of the synctable-schema. Since "PkgName" and "ClsName" are key columns, we specify "foreign-key" keyword in the reference decomposition. In addition, we can edit "description" of the reference via "ClsDesc2."

6.7 Key Columns defined in SyncTable Schema

In a synctable-schema, we need to specify key columns to identify the recode by such key columns. So in every dimension, we need at least one key column and all of the model elements associated with this dimension must be uniquely identified by the defined key columns. Key columns in dimensions are one of the followings:

1. Columns defined by "key column"

2. All of "foreign-key" columns in ReferenceDecomposition that uses references by "**key** reference-query" or dimensions. If you use "reference-query" without "key" keyword, such "foreign-key" columns are not key columns.

7 SyncTable

A SyncTable is an intermediate structure created in the first step of converting model data into a table from shown in an Excel spreadsheet. The definition of a SyncTable consists in applying a SyncTable schema to a Data Source. The formal syntax of a SyncTable definition is as follows.

```
SyncTable ::= 'syncTable' ID '=' SyncTableSchemaId '<' Data-  
SourceId '>' ( '(' SyncTableId (',' SyncTableId)* ')' )?
```

SyncTableSchemaId is an ID of a SyncTableSchema.

DataSourceId is an ID of a DataSource.

SyncTableId is an ID of a SyncTable.

8 Laying out SyncViews

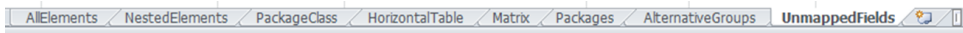
This chapter describes how SyncTables are presented as SyncViews. All of the SyncViews must be layed out in some Worksheet in a Workbook. The rest of the sections are organized as 1) how to set up Worksheets in a Workbook; 2) how to lay out SyncTables in a table; and how to lay out SyncTables in a matrix.

8.1 Setting up a Workbook and Worksheets

When MapleMBSE opens a model, it assigns one workbook to the model, and in *Workbook Instance*, we specify all of the worksheets managed by MapleMBSE. In each configuration, one and only one Workbook instance must be specified.

The example below comes from Example/UML/UserGuide.MSE, and it defines all of the worksheets

```
workbook {  
  worksheet AllElements(allElementsTable)  
  worksheet PackageClassProperty(classesInPackage) {label = "NestedElements"}  
  worksheet PackageClass(classesTable)  
  worksheet Horizontal(classesInPackage) {label = "HorizontalTable"}  
  worksheet Dependencies(dependenciesTable,dependentTable,supplierTable) {label =  
    "Matrix"}  
  lazy-load worksheet Packages(packagesTable) {label = "Packages"}  
  worksheet NestedPackageClass(nestedPackageClassTable) {label = "AlternativeGroups"}  
  
  worksheet UnmappedFields(nestedPackageClassTable) {label = "UnmappedFields"}  
}
```



In the example shown “workbook” is used to represent the arrangement of worksheets as shown, AllElements is the name of the worksheet template and allElementsTable is the name of the synctable that is created. By default, a worksheet is created with the name AllElements containing the information from the corresponding worksheet-template. MapleMBSE allows the user to create a name for the worksheet manually by using the ‘label’ attributes as shown in the above example.

Note: When a worksheet template is created with more than one parameter they should be mentioned with ‘,’ as shown above for the creating a Dependencies worksheet.

If "lazy-load" is specified before "worksheet", MapleMBSE will load syncviews of that worksheet when that worksheet is activated. By default, they are loaded at the startup and then the name of worksheet-template and its parameters follow. This means the worksheet

will be initialized with the specified worksheet-template. The details of worksheet-template is explained in the next section.

If the worksheet declaration has label="XXX", MapleMBSE regards "XXX" as the name of the worksheet. Otherwise, the name, "worksheet-template" is used as the name of the worksheet. For example, "AllElements" will be the name of the worksheet to initialize "worksheet AllElements(allElementsTable)". If the Excel template has the worksheet with the same name, MapleMBSE will use this worksheet to initialize the syncviews of worksheet-template. Otherwise, MapleMBSE will create a new worksheet with the same name.

8.2 Worksheet template and View Layout

A worksheet template is used to define how a SyncTable should be represented in the Excel worksheet. In a worksheet template, we can specify one or more View Layouts, each of which can be a table view layout or a matrix view layout, in the following subsections respectively.

The formal syntax of worksheet template is as follows.

```
WorksheetTemplate ::=  
'worksheet-template' ID '(' WorksheetTemplateParam (',' WorksheetTemplateParam)* ')'   
'{' ViewLayout* '}'
```

ViewLayout: TableViewLayout | MatrixViewLayout

where ID means the name of the worksheet template and should be referred by worksheet definitions explained in the previous section.

Table View Layout

Table view layout allows the user to define how the contents of the model should be displayed in the table. It has two possible arrangements: vertical or horizontal. The syntax for table view layout is as show above. To define a table layout: you must speciify the arrangement of the table, cell address to define the location of table in excel and the order of the fields. A column can be populated with either mapped or unmapped fields; a mapped field displays the attributes or value assigned to it whereas an unmapped field is used to insert a blank column within the table. Based on how fields are declared in the syncntable schema as key column or column inside the view layout the are declared as key field or field respectively. It is necessary to mention the column type as string or integer for every field that is created except for the unmapped field.

The following code snippet comes from UserGuide.MSE which defines a table view layout.

```
worksheet-template AllElements(cls:AllElementsTable) {  
  vertical table tabl at (3,2) = cls {  
    key field PackageName : String
```

```

field PackageVisibility : String
key field TopClassName : String
field TopVisibility : String
key field PropertyName : String
field PropertyVisibility : String
field AggType : String
field PropertyTypePackage : String
field PropertyType : String
field PropertyTypeVisibility : String
sort-keys PackageName, TopClassName, PropertyName
}
}

```

In the example shown above, a worksheet template with ID `Allelements` is created for a synctable-schema, `AllelementsTable` that is assigned to a parameter '`cls`'. Line 2 defines that the table is arranged vertically and "(3, 2)" means it should be displayed from Row 3 and Column B in Excel as shown in the figure below. Line 3 to line 12 in the example defines the order in which fields have to be displayed in Excel, shown as Table View in the figure. Line 3 in example, key field is used for `PackageName` because it was specified as key column in the synctable schema for `AllelementsTable`. Column type for every field is mentioned as shown from line 3 to line 12, in the example shown `String` is the type for all fields, in case of integer type '`Int`' is used instead of `String`. '`sort-keys`' are used to indicate the columns that should be sorted in ascending order when model data is loaded or when new data is added to the table.

Predefined Row in the figure denotes that the Excel sheet can be formatted based on user preference before the model is loaded in the Excel sheet.

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2		Package Name	Package Visibility	Class Name	Class Visibility	Property Name	Property Visibility	Aggregation Type	Property Type Package	Property Type	Type Visibility	
3		Package1	public									
4		Package1	public	Class1	public							
5		Package1	public	Class1	public	Property1	protected	shared	Package1	Class2	private	
6		Package1	public	Class1	public	Property2	public	composite	Package1	Class3	public	
7		Package1	public	Class2	private							
8		Package1	public	Class3	public							
9												
10												
11												

The formal syntax of table view layouts is as follows.

```

TableViewLayout ::=
('vertical' | 'horizontal') 'table' ID
'at' CellAddr
'=' WorksheetTemplateParam
'{'
( 'import-order' INT )? & ( 'enable-import' BoolType )?
ViewColumn*
( SortKeys )?
'}'

```

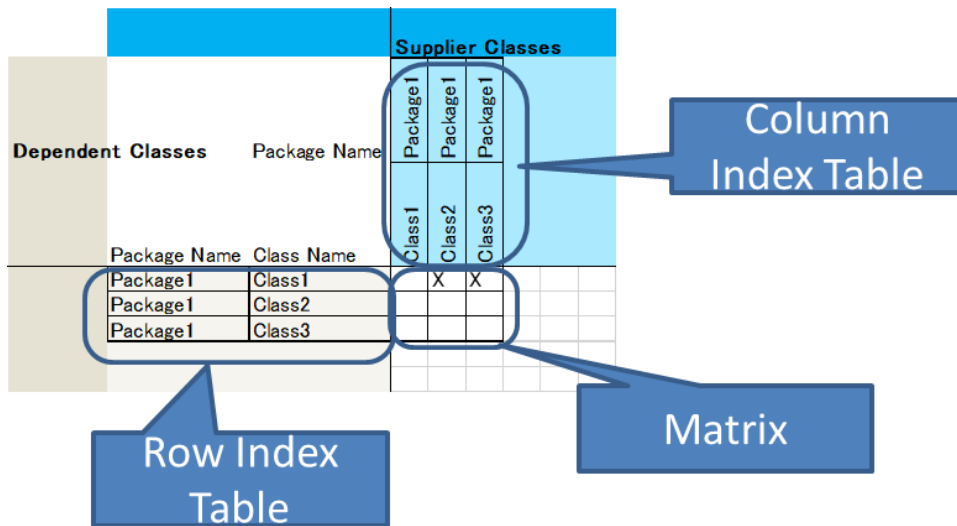
```

ViewColumn ::= MappedViewColumn | UnmappedViewColumn
MappedViewColumn ::= KeyViewColumn | NonkeyViewColumn
KeyViewColumn ::= 'key' 'ref'? 'field' [KeyColumn] ':' ViewColumnType
NonkeyViewColumn ::= 'ref'? 'field' [Column] ':' ViewColumnType
UnmappedViewColumn ::= 'unmapped-field'
ViewColumnType: 'String' | 'Int'

```

Matrix View Layout

A matrix view layout consists of three parts, row index table view part, column index table view part, and matrix part, as shown in the following example:



Row and column index tables identify which cell in a matrix should be selected to show a record of the synctable. The following code snippet comes from UserGuide.MSE which defined matrix view layout.

```

worksheet-template
Dependencies (mat:DependenciesTable, tabr:DependentTable, tabc:SupplierTable)
{
  matrix Matrix1 at (5,4) = mat {
    const-field "X"
    row-index = tabr {
      key field DependentPackageName : String
      key field DependentClassName : String

      sort-keys DependentPackageName, DependentClassName
    }
    column-index = tabc {
      key field SupplierPackageName : String
      key field SupplierClassName : String

      sort-keys SupplierPackageName, SupplierClassName
    }
  }
}

```

In this example, Lines 2 to 16 defines matrix layout with the name of "Matrix1" created by "mat", that is DependenciesTable (see parameters of Dependencies). "const-field" means a matrix cell should be filled with the specified value if and only if the corresponding record exists. You can specify some column instead of const-field. For example, if you specify "field DepName : String", you can edit DepName in matrix cells. However, you can specify only one field for a matrix view layout.

Lines 4 to 9 define a row index table, and Lines 10 to 15 define a column index table. By this configuration, DependentTable (row index table) needs to have DependentPackageName and DependentClassName columns, and SupplierTable (column index table) needs to have SupplierPackageName and SupplierClassName columns, and finally, DependenciesTable (matrix table) needs to have all of the columns, those are DependentPackageName, DependentClassName, SupplierPackageName, and SupplierClassName. In DependenciesSheet, we can show DependenciesTable in a vertical table as below:

Dependent Classes		Supplier Classes	
Package Name	Class Name	Package Name	Class Name
Package1	Class1	Package1	Class2
Package1	Class1	Package1	Class3

A synctable used to present in a Matrix **must have** all of the same key columns of row and column index tables. In this example, DependenciesTable (synctable to be shown in a matrix) must have DependentPackageName and DependentClassName that are the key columns of row index table (DependentTable), and SupplierPackageName and SupplierClassName that are key columns of column index table (SupplierTable). Notice that all of the column names must be unique.

The formal syntax of matrix view layout is as follows.

```
MatrixViewLayout ::=
'matrix' ID
'at' CellAddr
'=' [WorksheetTemplateParam]
'{'
( 'import-order' INT )? &
( 'enable-import' BoolType )?
ViewColumn
MatrixRowIndexViewLayout & MatrixColumnIndexViewLayout
'{'
```

```
MatrixRowIndexViewLayout ::=
'row-index' '=' [WorksheetTemplateParam]
'{'
ViewColumn*
( sortKeys=SortKeys )?
'{'
```

```
MatrixColumnIndexViewLayout ::=
'column-index' '=' [WorksheetTemplateParam]
'{'
ViewColumn*
( sortKeys=SortKeys )?
'{'
```