Contents

Standard SQL     2-50
Cloudscape       2-50
Oracle Oracle8   2-50

Stereotypes for MagicDraw constructs   2-51
Properties of code engineering set for DDL   2-52
  Properties for DDL script reverse engineering and generation   2-52

Supported SQL statements   2-55
Tips   2-57
  Short representation for primary key constraint   2-57
  Primary key constraint with overhead info   2-58
INTRODUCTION

OVERVIEW

MagicDraw code engineering provides a simple and intuitive graphical interface for merging code and UML models, as well as preparing both code skeletons out of UML models and models from code.

MagicDraw code engineering implements several cases where code engineering may be very useful:

- You already have code that needs to be reversed to a model.
- You wish to have the implementation of the created model.
- You need to merge your models and code.

The tool may generate code from models and create models out of code (reverse). Changes in the existing code can be reflected in the model, and model changes may also be seen in your code. Independent changes to a model and code can be merged without destroying data in the code or model.

MagicDraw UML code engineering supports Java, C++, CORBA IDL, DDL, XML Schema, WSDL, and C# languages also EJB 2.0 UML notation is supported. You may model EJB classes and generate descriptors for them. You may also reverse descriptors and will get a model describing your Enterprise Java Beans. Your models can be converted to any of those languages, or UML models can be created from the source code written in those languages. Also reverse from Java Bytecode and CIL is supported.

The Code Engineering Sets tool is MagicDraw tool managing center for all code engineering matters.

Code engineering is available only in Professional or Enterprise editions. In the following table you’ll find what languages are supported in different editions:

<table>
<thead>
<tr>
<th>Language</th>
<th>Professional Edition</th>
<th>Enterprise Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Java</td>
<td>+</td>
</tr>
<tr>
<td>Java Bytecode</td>
<td>Java</td>
<td>+</td>
</tr>
<tr>
<td>C++</td>
<td>C++</td>
<td>+</td>
</tr>
<tr>
<td>CORBA IDL</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>DDL/Database engineering</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>CIL</td>
<td>C#</td>
<td>+</td>
</tr>
</tbody>
</table>
CODE ENGINEERING SETS

You may manage code engineering through the Code Engineering Sets in the Browser tree. The Code Engineering Sets tree contains the list of all sets created in the project and instruments for managing those sets.

To add a new set

1. From the Code Engineering Sets shortcut menu, choose New.
2. Choose the language you want. (possible choices include: Java, Java Bytecode, C++, C#, CIL, CIL Disassembler, CORBA IDL, DDL (Cloudscape, DB2, Microsoft Access, Microsoft SQL Server, MySQL, Oracle, Pervasive, Pointbase, PostgreSQL, Sybase), EJB 2.0, XML Schema, and WSDL). The new set is created.

<table>
<thead>
<tr>
<th>Language</th>
<th>Professional Edition</th>
<th>Enterprise Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL Disassembler</td>
<td>C#</td>
<td>+</td>
</tr>
<tr>
<td>XML Schema</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>WSDL</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>C#</td>
<td>C#</td>
<td>+</td>
</tr>
<tr>
<td>EJB 2.0</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 1 -- Code engineering language options
Edit sets in the Round Trip Set dialog box. To open this dialog box

- Choose Edit from the set shortcut menu.

If you are performing round trip for the first time, the tip message box appears.

![Figure 2 -- Code Engineering Sets tip message box](image)

Disable the tip message box by deseleting the **Show this tip next time** check box.
The **Round Trip Set** dialog box allows you to manage entities to be added/removed to your set.

Specify **Working Directory** for displaying source files. This option indicates files and required sub-directories, where a code generation output goes. Type a path manually or by browsing in the directory tree, by clicking the ‘…’ button.

The **Working Package** option allows to define any package for reverse output or code generation. Model will be reversed or code generated from this specified package.

The **Round Trip Set** dialog box has two tabs: **Add Files** and **Add Data from Model**. The **Add Files** tab helps you manage the files of source code involved in your code engineering set.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>All files</td>
<td>Helps you find directories with the source files for the set.</td>
</tr>
<tr>
<td>Files of type</td>
<td>Contains possible file name extensions for the chosen language.</td>
</tr>
</tbody>
</table>
The **Add Data from Model** tab helps you manage elements located in the UML model.

**Figure 4 -- Round Trip Set dialog box. Add data from model tab**

The **All Data** list contains the hierarchy of UML model packages with model elements (or other packages) inside of them. Your code engineering set can be combined out of model and code elements.

The following buttons are available in the **Round Trip Set** dialog box:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>The selected file in the <strong>All Files</strong> or <strong>All Data</strong> list is added to the set.</td>
</tr>
<tr>
<td>Add All</td>
<td>All files in the opened or selected directory are added to the set.</td>
</tr>
<tr>
<td>Add Recursively</td>
<td>All files in the selected directory and its subdirectories are added to the set.</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes the selected entity from the set.</td>
</tr>
<tr>
<td>Remove All</td>
<td>Removes all entities from the set.</td>
</tr>
</tbody>
</table>

**GENERATE CODE**

You may generate code for the selected and prepared set and directly for model elements.
Code Generation for Set

Start code generation once the set or sets are prepared. For more details about creating and editing sets, see Section "Code Engineering Sets" on page 1-6.

- Choose Generate from the Code Engineering Sets item shortcut menu. It allows code generating for all created sets.
- Choose Generate from the selected set shortcut menu. It allows code generating only for the selected set.

The Code Generation Options dialog box appears.

![Code Generation Options dialog box](image)

The Code Generation Options dialog box allows you to specify the way your code will be generated.

Once generating options are specified for the set, code can be generated.

<table>
<thead>
<tr>
<th>Box name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Directory</td>
<td>Type the directory where the generated files will be saved.</td>
</tr>
<tr>
<td>'...'</td>
<td>The Set Output Directory dialog box appears. Select the directory for the generated files.</td>
</tr>
<tr>
<td>Set as Working Directory</td>
<td>The output directory is set as a working directory and files are saved to the working directory.</td>
</tr>
<tr>
<td>Reverse before generation</td>
<td>Changes your model according to changes in the existing code. WARNING: Exercise caution when selecting the Reverse before generation check box. If the model differs from the code, all differences in the model will be discarded. In such cases, you will lose some your work.</td>
</tr>
</tbody>
</table>
Code Generation for Model Element

All the classes contained in the component will be written to one file. However, code for the class can be generated in a different way. Select the class you wish to generate in the browser Data package and click Generate in the class shortcut menu. For packages and components, you may also select Generate, but you will not be able to specify the generation options. All the options related to that task will be set according to the default values.

If you have chosen framework generation for a single class or for packages, the Code Generation Options dialog box does not appear. The code is generated according to the default values.

If no errors occurred, you may view the results with your favorite file viewer or programming environment. Look for the files in the directory that you specified as your Working directory in the Round trip set dialog box or in the Project Options dialog box. Additional sub-directories could be created.

**REVERSE**

A reverse is an opposite operation to the code generation. The existing code can be converted to UML models with the help of MagicDraw reverse mechanism.

Prepare the sets in the exact same way that you did for code generation (see Section “Code Engineering Sets” on page 1-6)
Choose **Reverse** from the **Code engineering sets** item shortcut menu. It allows code reversing for all already created sets.

Choose **Reverse** from the selected set shortcut menu.

The UML model for the component can be reversed in the same way. Just select the component you are interested in from the browser and click **Reverse** on it shortcut menu.

Models can be reversed without creating a set.

**To reverse a model without creating a set**

1. From the **Tools** menu, choose **Quick Reverse**. The **Round Trip Set** dialog box appears.
2. Select the files from the **Round Trip Set** dialog box, **Add Files** tab.
3. Click **OK**. The **Reverse Options** dialog box appears.

---

**Figure 6 -- Reverse options dialog box**

<table>
<thead>
<tr>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIZUALIZATION</td>
<td></td>
</tr>
</tbody>
</table>
If you have a code set combined from several files, you may see changes you wish to model without reversing all the code. Only changed files should be reversed. This type of reversing can be done by clicking the Refresh button on the set shortcut menu, or by performing model refresh from the Code Engineering Sets dialog box.

## JAVA REVERSE TO SEQUENCE DIAGRAM

Java reverse to Sequence diagram functionality allows visualizing Java method implementation with UML Sequence diagram. Created from method Sequence diagram cannot be updated, every time new diagram should be generated.
To launch **Sequence Diagram from Java Source Wizard** and specify options needed for the reverse

- You are able to reverse any operation from the Browser: right click an operation, choose **Reverse Implementation** and launch **Sequence diagram from Java Source Wizard**.
- From the **Tools** menu, choose **Model Visualizer**, and then choose **Sequence Diagram from Java Source Wizard**.
- When reversing, in the **Reverse Options** dialog box, choose **Launch Model Visualizer** and then choose **Sequence Diagram from Java Source Wizard**.

The more detailed example of how this functionality works, see MagicDraw Tutorials.pdf, which is locate in `<MagicDraw installation directory>`, manual folder.

**Sequence Diagram from Java Source Wizard**

Sequence Diagram from Java Source Wizard is the primary tool for reversing sequence diagram from Java method. It contains four steps that are described below.

**STEP 1 SPECIFY NAME AND PACKAGE.**

![Sequence Diagram from Java Source Wizard](image)

*Figure 7 -- Sequence Diagram from Java Source Wizard*
In this step, type the name of the newly created sequence diagram. By default class name and selected operation name with a word “implementation” will be included in the sequence diagram name.

Also choose the package that will contain created sequence diagram. If you want to create a new package and place there a sequence diagram, click the New button and define package parameters in the Package Specification dialog box.

**STEP 2 SELECT OPERATION**

In this step, select an operation for which you want to create a sequence diagram. If the Java source file is not shown you must select it manually.

**IMPORTANT** To specify implementation files, we suggest, before reversing, to specify Java Default working directory in the Project Options dialog box (specify root folder where all source files can be found).
STEP 3 SELECT CLASSES FOR DIAGRAM

In the Select Classes for Diagram step, all referenced classes are displayed. Select the desired classes to show on the diagram.

- Select the **Analyze and split long expressions in diagram** check box if expression contains calls and cannot be displayed as call message. Then every call will be shown as separate call message with temporary variable initialization.
- Select the **Create return messages** check box, if you want to display return message for every call message.
- Select the **Wrap message text** check box and specify the maximum message text length in pixels, to wrap longer message.
STEP 4 SPECIFY SYMBOLS PROPERTIES

In this step, define symbols properties for lifelines and messages.

Figure 8 -- Sequence Diagram from Sequence Wizard. Specify Symbols Properties
GLOBAL OPTIONS FOR CODE ENGINEERING

Code engineering options for all sets in your project

From the Options menu, choose Project. The Project Options dialog box appears.

The Project Options dialog box has two main collections of customizable options, which are represented by the hierarchy tree on the left side of the dialog box:

- **Styles** – expands the tree hierarchy of all the styles defined within the project. You may use as many of these styles as you wish. See MagicDraw main User’s Manual, working with Projects Section.

- **Code engineering** – these options are found on the right side of the Project options dialog box:
  - **Default Working Directory** field - type the name or browse by clicking the button in the working directory.
- **Default Working Package** - allows to define any package for reverse output or code generation. Model will be reversed or code generated from this specified package.
- **Default Paths for References** - add specific profiles, modules, libraries to define where to search paths for references during reversion/code generation.
- **Default Encoding** - a list of available encodings appears.
- **Default language** drop-down box – select the default generation language.
- **Use Syntax Checker** check box – when selected, the syntax checker runs while Code Engineering is executed.
- **Directory for Temporal Files** - it can be Active Directory, System or define other by clicking “...” button.

<table>
<thead>
<tr>
<th>Tab name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code generation</td>
<td>Set code generation options using the fields listed in the right side of the Project options dialog box. The Code generation area contains boxes that have the same functionality as in the Code generations options dialog box (see Section “Generate Code” on page 1-9).</td>
</tr>
<tr>
<td>Reverse</td>
<td>Set reverse options for all reverse actions of the project using the options listed on the right side of the Project options dialog box. The Reverse area contains boxes that have the same functionality as in the Reverse options dialog box (see Section “Reverse” on page 1-11).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tab name</th>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Language Options</td>
<td>Generate opening bracket in new line</td>
<td>Opens a bracket in the new line that is being generated.</td>
</tr>
<tr>
<td></td>
<td>Generate spaces</td>
<td>Generates spaces inside an assignment and other operators.</td>
</tr>
<tr>
<td></td>
<td>Generate empty documentation</td>
<td>Comment brackets are placed in your code, unless class in the model has no documentation.</td>
</tr>
<tr>
<td></td>
<td>Automatic “import” generation</td>
<td>Automatic generation of &quot;import&quot; sentences according to classes that are referenced in the generated class.</td>
</tr>
<tr>
<td></td>
<td>Class count to generate import on demand</td>
<td>Specify number of classes imported from a single package until all statements importing a single class are substituted with a statement importing an entire package.</td>
</tr>
</tbody>
</table>
After selecting Java Doc processor, click the “...” button to open the Documentation Properties dialog box.

Two styles are available for documentation.

The ‘...' button is activated. Search a classpath for importing sentences generation in the Set classpath dialog box.

Figure 10 -- Set classpath dialog box

Click the Get from System button to get CLASSPATH variable defined by operating system or click the Add button and select the classpath directory in the Add Classpath dialog box.

Available choices 1.4 or 5.0

Add the specific header to all your code files. Click the button and enter header text in the Header screen. You may also define $DATE, $AUTHOR, and $TIME in the header.

Opens a bracket in the newly generated line.

Spaces inside an assignment and other operators are generated.

Comment brackets are placed in your code, unless class in the model has no documentation.

Select check box to generate methods body into class.

Two styles are available for documentation.

Select check box. The ‘...' button is activated. Click the ‘...' button and then specify the path for the includes in the Set Include Path dialog box.
**Tab name** | **Element name** | **Function**
--- | --- | ---
Use explicit macros | Select check box. The ‘…” button is activated, click it and in the C++ Macros dialog box use a set of predefined macros.
Header | Add the specific header to all your code files. Click the “…” button and enter header text in the Header screen. You may also define $DATE, $AUTHOR, and $TIME in the header.
**CORBA IDL 3.0 Language Options**
Generate documentation | Includes the documentation of an element in the comment.
Generate opening bracket in new line | Opens a bracket in the new line generating.
Generate spaces | Spaces inside an assignment and other operators are generated.
Generate empty documentation | Comment brackets are placed in your code, unless class in the model has no documentation.
Generate imports | Generation of "import" statements for classes that are referenced in the generated class.
Generate preprocessor directives | Generates pre-processors directives.
Documentation Style | Three styles are available for documentation.
Header “…” | Add the specific header to all your code files. Click the “…” button and enter header text in the Header screen. You may also define $DATE, $AUTHOR, and $TIME in the header.
Set Include Path | Specify the path for the "includes". Click the "…” button to open the Select Folder dialog box.
**DDL Language Options**
Generate opening bracket in new line | Opens a bracket in the new line generating.
Generate spaces | Spaces inside an assignment and other operators are generated.
Generate documentation | Comment brackets are placed in your code, unless class in the model has no documentation.
Header | Add the specific header to all your code files. Click the button and enter header text in the Header screen. You may also define $DATE, $AUTHOR, and $TIME in the header.
## C# Language Options

Set the generated code style for C# programming languages.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate opening bracket in new line</td>
<td>Opens a bracket in the newly generated line.</td>
</tr>
<tr>
<td>Generate spaces</td>
<td>Generates spaces inside an assignment and other operators.</td>
</tr>
<tr>
<td>Generate empty documentation</td>
<td>Comment brackets are placed in your code, unless class in the model has no documentation.</td>
</tr>
<tr>
<td>Generate required &quot;using&quot; directives</td>
<td>Automatic generation of &quot;using&quot; directives. This option facilitates the usage of namespaces and types defined in other namespaces.</td>
</tr>
</tbody>
</table>
| Concatenate namespace names                      | If not selected namespace names are separated into several lines. e.g. namespace A {
                                                                                     namespace B
                                                                                     }

### Documentation:

- **Processor**
  - Use C# XML processor then generates c# xmi documentation for commenting the code.
- **Style**
  - Select one of the listed comment styles.

### Header

- Adds the specific header to all your code files.
  - Click the '...' button and type header text in the Header dialog box.
  - You may also define $DATE, $AUTHOR, and $TIME in the header.

### Conditional Symbols

- Add the conditional symbols, which can not be recognized and should be skipped during reverse.
  - Click the '...' button and add conditional symbols in the Define Conditional Symbols dialog box.
Java Documentation Properties dialog box

To open the Java Documentation Properties dialog box

In the Project Options dialog box, Java Language Options group, select the Java Doc processor in the Documentation field and click the “…” button to open the Documentation Properties dialog box.

![Documentation Properties dialog box](image)

**Box Name** | **Function**
--- | ---
Tag Name | Type a tag name.
Value | Type the value of the tag.
Generate | The selected tag will be placed in the generated code as a comment before classifier (class or interface), operation or attribute.
Up | Moves the selected item up the list.
Down | Moves the selected item down the list.
Add | Adds a new item in the list.
Remove | Removes the selected item from the list.
OK | Saves changes and closes the dialog box.
Cancel | Closes the dialog box without saving changes.
Help | Displays MagicDraw Help.
Round Trip

MagicDraw round trip keeps your code and model synchronized, and because Round trip traces all the model and code changes, you may freely change entity specifications without discarding code changes made outside the tool.

For example, Round Trip prevents a job from being damaged by code additions or changes when these steps are followed:

Within the tool, class Base is created.
1. Operation getInstance is added to class.
2. Code is generated
3. With external tool, programmer adds code to that operation.
4. With MagicDraw UML, operations name is changed to Instance.
5. Code is generated.

If the tool rewrites the whole code, these changes are made without corrupting the programmer’s job. The name of the operation is changed, but the internals remain the same.

Round trip catches all changes in your project and controls the following actions:

- If the source code is not changed, it is not allowed to refresh UML model. The Refresh command from the set shortcut menu is unavailable.
- If the model is changed but the code remains the same (new members were added or their headers were changed), refresh is not allowed, and the Refresh command from the set shortcut menu is unavailable. When generating code according to changes, all changes in the model are written to the signatures of class members, leaving the old implementation in place.
- If the code is changed but the model remains the same, refresh can be executed: code will be reversed to the UML models. If the Code Generation Options dialog box appears when you are attempting to generate code, you may select a code action that differs from the UML model.
- If the code and model are changed while refreshing, all changes in the code are treated as new items and added to the model.
- If data in the model file is deleted, it will be restored while refreshing, even when the code has not been changed or the data itself is unimportant.
Type Mapping Table

Languages supported by MagicDraw UML have their own built-in types. One language’s type might have no matches in another language, or it might have multiple matches. Additionally, some names are interpreted differently in different languages. When performing code generation, therefore, problems may occur when switching between different languages. To avoid this, MagicDraw UML uses type-mapping tables to manage mapping between languages. It describes the rules of how one language’s built-in types are converted to those of another language.

FILES OF PROPERTIES

The code can be generated out of prepared UML models. The mapping between the identifiers, used in the UML model and the language to which the model is being generated, should be implemented. This mapping includes the following sections:

- Build-in types (their default values)
- Generalization types
- Possible class declarations. Attributes and operations declaration and visibility modifiers
- Code generation options.

The separate prop file is created for every language that is supported by MagicDraw. Files are located in the `<MagicDraw installation directory>/data` folder. The file name pattern is `lang.prop`, where `lang` stands for the name of the programming language.

<table>
<thead>
<tr>
<th>Supported language</th>
<th>File of Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA</td>
<td>java.prop</td>
</tr>
<tr>
<td>C++</td>
<td>C++.prop</td>
</tr>
<tr>
<td>CORBA IDL</td>
<td>idl.prop</td>
</tr>
<tr>
<td>JAVABytecode</td>
<td>javabytescode.prop</td>
</tr>
<tr>
<td>DDL</td>
<td>ddl.prop</td>
</tr>
<tr>
<td>CIL</td>
<td>cil.prop</td>
</tr>
<tr>
<td>CIL Disassembler</td>
<td>cil disassembler.prop</td>
</tr>
<tr>
<td>C#</td>
<td>c#.prop</td>
</tr>
<tr>
<td>EJB</td>
<td>ejb.prop</td>
</tr>
<tr>
<td>EJB 2.0</td>
<td>ejb20.prop</td>
</tr>
<tr>
<td>IDL</td>
<td>idl.prop</td>
</tr>
</tbody>
</table>
Files of language properties are separated into sections where all related elements are grouped. You may edit existing entities, add new ones, and change the default values.

We strongly recommend that you edit default values only. In general, all the sections have the list of possible and default values for the element.

<table>
<thead>
<tr>
<th>Supported language</th>
<th>File of Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Schema</td>
<td>xmlschema.prop</td>
</tr>
<tr>
<td>WSDL</td>
<td>wsdl.prop</td>
</tr>
</tbody>
</table>
MagicDraw UML has the tools that forward engineer UML constructs to DDL script. The resulting DDL script depends on the selected DDL language dialect.

**Retrieve DB Info dialog box**

*IMPORTANT!* Since MagicDraw version 9.0, Retrieve DB Info dialog box has been moved to the Round Trip Set dialog box.

The Retrieve DB info function allows you to retrieve the information from the existing database or ODBC source. Database structure is retrieved as UML model, using class diagram elements.

The mapping of the DB to UML is the same as in DDL reverse engineering.

**NOTE** As the retrieve DB info function uses JDBC bridge, you may need to have JDBC drivers. All types of drivers are valid for making a connection. The driver should be able to retrieve the database metadata information. The same applies to the database; it should be able to provide this information.

**To open the Retrieve DB Info dialog box**

1. From the Code Engineering Sets shortcut menu, choose New.
2. Choose DDL and then choose the database vendor you need. (possible choices include: Cloudscape, DB2, Microsoft Access,
3 The new set is created.

4 From the created set shortcut menu, choose Edit. The Round Trip Set dialog box appears. Select the Reverse from DB option.
From the **Tools** menu, choose **Quick Reverse** and then choose DDL and required database vendor. The **Round Trip Set** dialog box appears. Select the **Reverse from DB** option.

**Box name** | **Function**
--- | ---
**Recently Used** | Contains the list of the recently used reverse templates. Choose the one you need and click **Apply**.
**DB Connection URL** | The connection URL for the selected profile.
**Driver Files** | Contains .jar and .zip files or directories with JDBC driver’s classes. To choose the files or directories you want to add or remove, click the “…” button. The **Select Files and/or Directories** dialog box appears.

**NOTE** If the driver file is empty, **Driver Class** is searched from the classpath.
This section describes to what data model constructs MagicDraw constructs are converted.

## Packages

The Database is represented as a Package with the `<database>` stereotype. Each View or Table can be assigned to a Schema where the Schema is represented as a Package with the `<schema>` stereotype.
A Package, depending on a package stereotype, is mapped to one of the following DDL constructs:

- A Database, if the package has the `<<database>>` stereotype. Elements: `CREATE DATABASE <database_name>`
  Database name is equal to the package name.

- A Schema, if the package has the `<<schema>>` stereotype. Elements: `CREATE SCHEMA [<database_name>.]<schema_name>`
  Schema name is equal to the package name. Schema database name is the name of a package that contains schema package.

- Otherwise it is not mapped.

**NOTE**
If a package has no stereotype and the EnableDefaultStereotypes property is true, the `<<database>>` stereotype is used for the first level packages, and the `<<schema>>` stereotype is used for the second level packages.

![Figure 13 -- Package mapping example](image)

**Figure 13 -- Package mapping example**

DDL script, generated for example showed in Package mapping example, creates one Database and two Schemas:

```sql
CREATE DATABASE DB1;
CREATE SCHEMA DB1.Schema1;
CREATE SCHEMA DB1.Schema2;
```

**Classes**

A class, depending on class stereotype, is mapped to one of the following DDL constructs:

- Table, if the class has the `<<table>>` stereotype. Elements: `CREATE TABLE [<schema_name>.]<table_name> (<column_and_constraint_list>)`
  Table name is equal to the class name. Table schema name is the name of package that contains table class.
• View, if the class has the <<view>> stereotype.

Elements: CREATE VIEW [schema_name].view_name
[(<column_list>)] AS SELECT <derived_column_list> FROM
<table_list>.

View name is equal to the class name. The view schema name is the name of a package that contains view class. Table list within view “FROM” clause are derived from dependencies between the view class and tables classes.

• Otherwise it is not mapped.

NOTE If a class has no stereotype and the EnableDefaultStereotypes property is true, the class is treated as a class with the <<table>> stereotype.

Figure 14 -- Class mapping example

DLL script, generated for classes is showed in Class mapping example. It has two Table definition statements (CREATE TABLE) and view definition statement (CREATE VIEW):

```sql
CREATE TABLE Person (  
id integer,  
socialId number (10),  
lastName varchar (20),  
firstName varchar (10),  
sex char (1)  
);  
CREATE TABLE Account (  
accountNo integer,  
balance float (5) DEFAULT 0.0,  
personalId integer);  
CREATE VIEW ImportantClient
AS SELECT P.*, Account.balance as total
FROM Person, Account;
```
Attributes

An attribute of a class with the `<<table>>` or the `<<view>>` stereotype, depending on an attribute stereotype, is mapped to one of the following DDL constructs:

- A Column of a Table, if the class that contains an attribute has the `<<table>>` stereotype.
  A Column name is equal to the name of an attribute. A Column type is equal to the type of an attribute. Column default value is equal to the initial value of an attribute (if any).

- A Column of a View, if the class that contains an attribute has the `<<view>>` stereotype.
  Elements: `[<column_expression> AS] <column_name>`
  A Column name is equal to the name of an attribute. Column expression is equal to the initial value of an attribute (if any).

![Figure 15 -- Attributes mapping example](image)

Col1 attribute of the Table1 class (see Attributes mapping example) is mapped to the col1 column of a Table1 table, and col1 attribute of the View1 class is mapped to the col1 column of a View1 view. There is DDL script for attributes mapping example:

```sql
CREATE TABLE Table1 (  
  col1 integer DEFAULT 123  
);  
CREATE VIEW View1  
AS SELECT Table1.col1 AS v1  
FROM Table1;
```

Operations

An operation (method) of a class with the `<<table>>` stereotype, depending on an operation’s stereotype, is mapped to one of the following DDL constructs:

- An Index for a Table, if the operation has the `<<index>>` stereotype;
  Elements: `CREATE INDEX [<schema_name>.]<index_name> ON <table_name>(<column_list>)`
  An Index name is equal to the name of an operation. Names of comma-delimited set of column for the index are equal to the parameter names of an operation.
• Tags with names: trigger action time, trigger event and triggered action has meaning in DDL CG have meaning in the DDL CG if are specified for an operation with stereotype «<». The "implementation" tag specified for operation with stereotype trigger has no meaning for the DDL code generation since MD 6.0 version. The "implementation" tag was split to three tags: "trigger action time", "trigger event" and "triggered action". Values of these tags will be used then generating definition of a trigger.

::= CREATE TRIGGER ON [ REFERENCING ]

• A constraint, if the Operation has the «<<PK>>, «<<unique>>, or «<<check>>» stereotype.

Elements: [ALTER TABLE <<table_name>> ADD] CONSTRAINT <<constraint_name>>

Constraint name is equal to the name of an operation.

• PRIMARY KEY («pk_column_list»)

Primary key column list contains all attributes with the «<<PK>>» stereotype.

• UNIQUE («unique_column_list»)

Unique constraint element «unique_column_list» is generated from operation’s parameter names.

• CHECK («check_expression»)

Check constraint element «check_expression» is generated from tag named Implementation. Otherwise it is not mapped.

NOTES:

IndexNamePrefix property specifies an optional naming standard that is added to the beginning of the name for each generated index.

TriggerNamePrefix property specifies an optional naming standard that is added to the beginning of the name for each generated trigger.

An example of the DDL script for operations mapping:
CREATE TABLE TableB (  
id integer
CONSTRAINT checkColumn CHECK(/*<check_expression>**/),
  b1 integer,
  b2 integer,
CONSTRAINT pk1 PRIMARY KEY (id),
CONSTRAINT checkTable CHECK(/*<check_expression>**/),
CONSTRAINT unique1 UNIQUE(b1,b2)
);

CREATE TABLE TableA (  
fk integer,
a1 integer DEFAULT 123,
CONSTRAINT fk1 FOREIGN KEY (fk) REFERENCES TableB(id)
);
CREATE INDEX indexOnA1 ON TableA(a1);
CREATE TRIGGER trigger1 ON TableA /*<triggered_SQL_statement>*/;

Relationship cardinalities

**ONE-TO-MANY (1:N) RELATIONSHIP**

One-to-many (1:N) relationship is mapped to dependency with `<FK>` stereotype.

For 1:N relationship allowed cardinalities for end A are 0, 1, N. For end B allowed cardinalities are 0, 1 (see Cardinalities of 1:N relationship -- Cardinalities of 1:N association).

Concrete cardinalities of A and B ends are mapped to different data model constraints (if any):

1. If A end does not allow N cardinality, the UNIQUE constraint is assigned to _b_id_ column.

   When _b_id_ column has the UNIQUE constraint, every instance of A class references a unique instance of B class (if any). Given B class instance is associated with the unique instance of A class (if any). This means that A end has no N cardinality.
If B end does not allow 0 cardinality, NOT NULL constraint is assigned to b_id column.

When b_id column has NOT NULL constraint, every A class references some B class. This means that ca role does not allow 0 cardinality.

Figure 18 -- 1:N relationship and mapped representation

Script creating 1:N relationship:

```sql
CREATE TABLE Company (
    id INTEGER NOT NULL PRIMARY KEY);
CREATE TABLE Employee (c_id INTEGER NOT NULL,
    CONSTRAINT FK_1 FOREIGN KEY (c_id) REFERENCES Company (id));
```

**ONE-TO-ONE (1:1) RELATIONSHIP**

One-to-one (1:1) relationship is handled as a special case of one-to-many relationships, where end B allowed cardinalities are 0, 1 and end A allowed cardinalities are 0, 1 (see Cardinalities of 1:N relationship16 -- Cardinalities of 1:N association).

1:1 relationship is mapped to a dependency with <<FK>> stereotypey. The 1:1 cardinality must be forced through constraints.

Figure 19 -- 1:1 relationship and mapped representation

A script that creates a 1:1 relationship:

```sql
CREATE TABLE Company ( 
    id INTEGER NOT NULL PRIMARY KEY 
);
CREATE TABLE Director ( 
    c_id INTEGER NOT NULL UNIQUE, 
    CONSTRAINT FK_1 FOREIGN KEY (c_id) REFERENCES Company (id) REFERENCES Company (id) 
);
Nevertheless, this DDL script with these constrains does not ensure a strict 1:1 relationship – Company table may have rows that do not have their counterpart rows within Director table.

**MANY-TO-MANY (N:M) RELATIONSHIP**

Many-to-many (N:M) associations are not handled by MagicDraw UML. The N:M relationship can be achieved by using two 1:N relationships with intermediate table.

### Inheritance

**SINGLE INHERITANCE**

![Diagram showing single inheritance](image)

Single inheritance can be modeled with the 1:1 relationship, by creating the foreign key constraint on primary key in the derived class.

A script that creates a single inheritance:

```sql
CREATE TABLE Employee (  
    id INTEGER NOT NULL PRIMARY KEY  
);  
CREATE TABLE Director (  
    id INTEGER NOT NULL PRIMARY KEY,  
    CONSTRAINT FK_1 FOREIGN KEY (id) REFERENCES Employee(id)  
);  
NOTE The Employee table may have rows that do not have their counterpart rows within Director table.
```

**MULTIPLE INHERITANCE**

The mapping of a multiple inheritance is similar to the mapping of a single inheritance.
A script that creates a multiple inheritance:

```sql
CREATE TABLE Base1 {
    id INTEGER NOT NULL PRIMARY KEY
};
CREATE TABLE Base2 {
    id INTEGER NOT NULL PRIMARY KEY
};
CREATE TABLE Derived {
    b1_id INTEGER NOT NULL,
    b2_id INTEGER NOT NULL,
    PRIMARY KEY (b1_id, b2_id),
    CONSTRAINT FK_1 FOREIGN KEY (b1_id) REFERENCES Base1(id),
    CONSTRAINT FK_2 FOREIGN KEY (b2_id) REFERENCES Base2(id)
};
```

Not supported UML constructs

Constructs that are not mapped into DDL script, because this would lead to a generation of an illegal DDL code:

- Duplicated names are not allowed.

**NOTE**
Uppercase and lowercase letters are equivalent.

- Database package cannot contain two schema packages with the same name.
- Schema package cannot contain two UML constructs that are mapped to the schema elements such as table classes, view...
classes, index operations, and trigger operations that have the same name.

- Table class cannot have two column attributes or constraint operations with the same name.
- View class cannot have two column attributes with the same name.
- Table class cannot have two operations with the <<PK>> stereotype, because a table can have only one primary key (if any).
- References to non-existing columns are illegal.
- The parameter name of an operation with a DDL stereotype (<<index>>, <<trigger>>, <<PK>>, <<unique>>, <<check>>) must be the name of an existing column attribute.
- Supported attribute multiplicity can be [not specified], [0..1], and [1]. All other attribute multiplicities are not supported.

### REVERSE ENGINEERING FOR DDL SCRIPT

Information about a specific database structure acquired reversing DDL script or from JDBC is mapped to the MagicDraw UML constructs as described below.

**Database**

A Database is a system for a data storage and controlled access to the stored data. It is the biggest construct that a data model supports.

A package, which is used with the <<database>> stereotype, represents a database in the MagicDraw UML model. The database that is modeled as a package must have a name.

Example of a DDL script:

```sql
CREATE DATABASE BankDB;
```

Representation using UML concepts:

```uml
<<database>>
BankDB
```

*Figure 22 -- Database example*
See also: CurrentDatabaseName property

### Schema

A package with the `<schema>` stereotype within the package with the `<<database>>` stereotype represents a database schema.

Example of a DDL script:

```sql
CREATE SCHEMA Public;
```

Example rewritten using a qualified schema name “BankDB.Public”:

```sql
CREATE SCHEMA BankDB.Public;
```

Representation using UML concepts:

![Figure 22 – Schema example](image)

#### NOTE
There can be more than one schema associated to a database.

See also: CurrentSchemaName property

### Table

A table is the basic modeling structure of a relational database. It represents a set of records of the same structure, also called rows. Each of these records contains data. Information about the structure of a table is stored in the database itself.

A class with the `<table>` stereotype represents a relational table in a schema of a database.

Example of a DDL script for table:

```sql
CREATE TABLE Account (
    accountNo INTEGER NOT NULL,
    personId INTEGER NOT NULL,
    balance FLOAT(5) DEFAULT 0.0 NOT NULL
);
```

This example may be rewritten instead of “Account” using qualified table name “BankDB.Public.Account”.

Representation using UML concepts:
Hosting the table in the schema package creates an association of a table to a schema.

**Column**

A table contains columns. A column must have a defined name and data type; a default value and several constraints are optional.

Example: balance FLOAT(5) DEFAULT 0.0

A column is represented as an attribute. A name, data type, and initial value of an attribute are set according to the name, data type, and default value of the column. 

The stereotype of an attribute is set according to column constraints. Stereotype can be not specified, <<unique>>, or <<PK>>.

**Constraint**

A constraint is a rule applied to the structure of a database. This rule extends the structure of a database and can be applied to a column or a table.

In general, a constraint may be represented as an operation with an appropriate stereotype and parameter list containing the list of column names that constraint concerns. An operation name is equal to the name of a constraint, but, when a constraint has no name specified, operation is unnamed.

All listed constraints (null, not null, uniqueness, primary key, foreign key, and check) are implemented in the following example:

```
CREATE TABLE Person (
    id INTEGER NOT NULL PRIMARY KEY,
    socialId NUMBER(10) NOT NULL UNIQUE
    CONSTRAINT checkSocialId CHECK(socialId>0),
    lastName VARCHAR(20) NOT NULL,
    firstName VARCHAR(10) NOT NULL,
    sex CHAR(1) NULL
);
```

```
CREATE TABLE Account (
    accountNo INTEGER NOT NULL,
    balance FLOAT(5) DEFAULT 0.0 NOT NULL
    CONSTRAINT checkBalance CHECK(balance>=0),
```
NOT NULL, NULL

A Null constraint is not defined in SQL-92 standard but some dialects use it. Microsoft SQL Server allows setting a flag, which indicates that all columns have Not Null constraint, which is set by default. A Null constraint that is assigned for a column overrides the default Not Null constraint and allows null values.

Not Null constraint indicates that the value for an attribute is required; Null constraint indicates that the value is optional.

Example:

```sql
firstName VARCHAR(10) NOT NULL,
sex CHAR(1) NULL
```

Not null constraint and null constraint are modeled as the multiplicity of an attribute.

Multiplicity may be indicated by placing a multiplicity indicator in brackets after the name of an attribute. A multiplicity of 0..1 provides a possibility of null values, for an example: sex [0..1]: CHAR(1)

In the absence of a multiplicity indicator, an attribute holds exactly one value.

See also: ColumnDefaultNullability, AttributeDefaultMultiplicity, GenerateNullConstraint, GenerateNotNullConstraint properties.

Example of a DDL script:

```sql
CREATE TABLE NullableExample ( 
  dontcare INTEGER,
  optional INTEGER NULL,
  required INTEGER NOT NULL
);
```

Representation using UML concepts:
UNIQUENESS

Uniqueness constraint indicates that the value of a column must be unique within the table.

Uniqueness constraint is modeled as the <<unique>> stereotype applied to an attribute and/or operation with the <<unique>> stereotype and a parameter list.

Example of a DDL script:

```sql
CREATE TABLE UniqueExample (
  col1 INTEGER CONSTRAINT uniqueColumn UNIQUE,
  col2 INTEGER,
  col3 INTEGER,
  CONSTRAINT uniqueCombination UNIQUE (col2, col3)
);
```

Representation using UML concepts:

```uml
<<table>>
NullableExample

-<null>=integer
-<null>=optional:integer
-<null>=required:integer
```

Figure 25 -- Not null, null constraints example

PRIMARY KEY

Primary Keys uniquely identify a row in a table. They mark an attribute as the Primary Key or the part of the Primary Key. The attribute must be of a scalar type. If more than one Primary Key attribute is identified, a concatenated primary key is generated.

A Primary Key is represented as the <<PK>> stereotype on an attribute and/or an operation with the <<PK>> stereotype and the parameter list.

Example of a DDL script:

```sql
CREATE TABLE PKColumnExample (
  col1 INTEGER Constraint pkColumn PRIMARY KEY
);
CREATE TABLE PKColumnExample (
  col2 INTEGER,
  col3 INTEGER,
```
CONSTRAINT pkCombination PRIMARY KEY (col2, col3);

Representation using UML concepts:

![UML concept diagram](image)

FOREIGN KEY

A Foreign Key constraint represents a relationship to another table.

Foreign Key constraint is represented as a dependency with the "<<FK>>" stereotype to the target table.

CONSTRAINT <fkname> FOREIGN KEY (<linkCols>) REFERENCES <targetTable>(<targetCols>)

<table>
<thead>
<tr>
<th>UML concept</th>
<th>FK element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency name</td>
<td>&lt;fkname&gt;</td>
<td>fk</td>
</tr>
<tr>
<td>Dependency source class</td>
<td>&lt;linkTable&gt;</td>
<td>FKLink</td>
</tr>
<tr>
<td>&quot;FK columns&quot; tagged value on the dependency</td>
<td>&lt;linkCols&gt;</td>
<td>t1,t2</td>
</tr>
<tr>
<td>Dependency target class</td>
<td>&lt;targetTable&gt;</td>
<td>FKTarget</td>
</tr>
<tr>
<td>&quot;PK columns&quot; tagged value on the dependency</td>
<td>&lt;targetCols&gt;</td>
<td>t1,t2</td>
</tr>
</tbody>
</table>

Example of a DDL script:

```sql
CREATE TABLE FKTarget1(
    t INTEGER PRIMARY KEY
);
CREATE TABLE FKLink1(
    r INTEGER CONSTRAINT fk1 REFERENCES FKTarget1(t),
);
CREATE TABLE FKTarget (t1 INTEGER,
                        t2 INTEGER,
                        PRIMARY KEY (t1, t2));
CREATE TABLE FKLink (r1 INTEGER,
                      r2 INTEGER,
                      CONSTRAINT fk2 FOREIGN KEY (r1,r2) REFERENCES FKTarget(t1,t2));
```
Representation using UML concepts:

```
<< table >>
FKTarget1
<< PK >> t1 : integer
<< FK >> (columns = t1)

<< table >>
FKLink1
- r1 : integer

<< table >>
FKTarget
- t1 : FKTarget
- t2 : integer
<< FK >> (columns = (t1, t2))
```

**Figure 28 -- Foreign key constraint example**

### Check

The Check constraint checks the value of data according to a given expression.

The Check constraint is represented as an operation with the `<<check>>` stereotype. Operation’s name is equal to the name of the check constraint or, if the constraint name is not specified, the name “unnamed” is generated.

There are a column check constraint and table check constraint:

For column check constraint operation’s parameter list contains one parameter with the name that equals to the name of the attribute for which check constraint is assigned.

Example: Table check constraint operation has no parameters.

```sql
CREATE TABLE CheckExample (  
    balance INTEGER CONSTRAINT checkBalance CHECK(balance>=0) 
    start INTEGER, -- period start balance  
    income INTEGER CHECK(income>=0), -- unnamed check constraint  
    outcome INTEGER,  
    CONSTRAINT checkInOut CHECK(start+income-outcome = balance)  
);  
```

Representation using UML concepts:

```
<< table >>
CheckExample
- balance : INTEGER 
- income : INTEGER 
- outcome : INTEGER

<< check >>
+ checkBalance( balance )
+ checkInOut() 
+ unnamed1( income )
```

**Figure 29 -- Check constraint example**
UNNAMED CONSTRAINT REPRESENTATION AS A STEREOTYPE OF AN ATTRIBUTE

Some column characteristics (column primary key constraint, column uniqueness constraint) that apply to one specific column may be represented as stereotype of that attribute.

Example DDL script:

```
CREATE TABLE ConstraintExample1 (
    col1 INTEGER PRIMARY KEY,
    col2 INTEGER UNIQUE
);
```

Representation using UML concepts:

```
<<table>>
ConstraintExample1
<<PK>>
- col1 : INTEGER
<<unique>>
- col2 : INTEGER
```

Only one constraint can be represented as stereotype, because only one stereotype can be assigned for the attribute. If there are other constraints, they should be represented as operations.

Although attribute may have several such constraints assigned, in practice there is no need for more than one. If attribute has primary key constraint it is unique too.

Example DDL script:

```
-- here unique constraint is unnecessary
CREATE TABLE ConstraintExample2 (
    col1 INTEGER PRIMARY KEY UNIQUE
);
```

Representation using UML concepts:

```
<<table>>
ConstraintExample2
<<PK>>
- col1 : INTEGER
<<unique>>
+ unnamed1( col1 )
```

Index

An Index is a physical data structure that speeds up the access to data. It does not change the quality or the quantity of data retrieved. The index specifies the columns
included and optionally the uniqueness of the index. An index can include multiple columns or just a single column.

An index is represented as an operation with the <<index>> stereotype and parameter list. The operation name is equal to the index name and the parameter list of the operation is a column names for the index. The uniqueness of the index is ignored (not mapped).

DDL script for an index example that creates the Person table and two indexes:

```sql
CREATE TABLE Person (
    id INTEGER NOT NULL PRIMARY KEY,
    socialId NUMBER(10) NOT NULL UNIQUE,
    lastName VARCHAR(20) NOT NULL,
    firstName VARCHAR(10) NOT NULL,
    sex CHAR(1)
);
CREATE UNIQUE INDEX bySocialId ON Person(socialId);
CREATE INDEX byName ON Person(lastName,firstName);
```

Representation using UML concepts:

```
<<table>>
Person
+firstName[1] : VARCHAR ( 10 )
+lastName[1] : VARCHAR ( 20 )
+sex : char ( 1 )
+id[1] : INTEGER
+socialId[1] : NUMBER ( 10 )
+byName( lastName, firstName )
+bySocialId( socialId )
```

Figure 32 -- Index example

**Trigger**

A trigger is an activity executed by the DBMS as a side effect or instead of a modification of a table or view to ensure consistent system behavior on data operations.

A Trigger is represented as an operation with the <<trigger>> stereotype.

DDL script for trigger example that creates trigger:

```sql
CREATE TRIGGER logActions BEFORE INSERT OR DELETE OR UPDATE
ON Person
<triggered SQL statement>;
```

Representation using UML concepts:
**View**

A View is a construct for creating a virtual table based on one or more existing tables or views.

A class with the <<view>> stereotype represents a view in a schema of a database. The View column is modeled as an attribute. Relationships between the View and its underlying tables ("FROM" clause) are modeled as dependencies with <<reference>> stereotype. Referenced table alias is modeled as the name of the dependency, referenced column list as the "columns" tagged value on the dependency.

<table>
<thead>
<tr>
<th>View element</th>
<th>UML concept</th>
<th>Script example</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>Class with the &lt;&lt;view&gt;&gt; stereotype</td>
<td>CREATE VIEW</td>
</tr>
<tr>
<td>View name</td>
<td>Class name</td>
<td></td>
</tr>
<tr>
<td>Column</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>Attribute name</td>
<td></td>
</tr>
<tr>
<td>Derived column</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>- &lt;expression&gt; AS &lt;name&gt;</td>
<td>default value &lt;expression&gt;, name &lt;name&gt;</td>
<td>Account.balance AS total</td>
</tr>
<tr>
<td>- *</td>
<td>Attribute name</td>
<td>*</td>
</tr>
<tr>
<td>- &lt;typename&gt;.*</td>
<td>Attribute name</td>
<td>P*</td>
</tr>
<tr>
<td>- &lt;typename&gt;.&lt;columnname&gt;</td>
<td>Attribute name</td>
<td>Balance</td>
</tr>
<tr>
<td>- &lt;typename&gt;.&lt;columnname&gt;</td>
<td>Attribute name &lt;typename&gt;.&lt;columnname&gt; and default value</td>
<td>Account.balance</td>
</tr>
<tr>
<td>- &lt;expression&gt; Generated unique attribute</td>
<td>name and default value</td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td>Table reference</td>
<td>Dependency</td>
<td></td>
</tr>
</tbody>
</table>
DDL script for view example:

```sql
CREATE TABLE Person (  
id INTEGER NOT NULL,  
socialId NUMBER(10) NOT NULL,  
lName VARCHAR(20) NOT NULL,  
firstName VARCHAR(10) NOT NULL,  
sex CHAR(1)  
);
CREATE TABLE Account (  
accountNo INTEGER NOT NULL,  
balance FLOAT(5) DEFAULT 0.0 NOT NULL,  
personalId INTEGER NOT NULL  
);
CREATE VIEW ImportantClient  
AS SELECT P.*, Account.balance as total  
FROM Person as P(id,firstName,lastName), Account  
WHERE balance >= 1000000.00 AND personId = P.id;
```

Representation using UML concepts:

---

**DDL DIALECTS**

This section reviews DDL implementations from different vendors. Specific implementation usually states compliance to some level of SQL standard and provides some extensions.
Standard SQL2

For SQL2 statements supported by MagicDraw UML see Section Supported SQL statements, “Supported SQL statements”, on page 55.

MagicDraw UML schema package is located within a database package. Database definition statement is not the part of the SQL2 standard - it is an analogue of a Database (a Catalog).

NOTE A Catalog has no explicit definition statement. If a database package for a Catalog does not exist, it should be created (when it is referred for the first time).

Cloudscape

Informix Cloudscape v3.5 dialect has no database definitions statement. A database package with the name specified by CurrentDatabaseName property is used.

This dialect has CREATE INDEX and CREATE TRIGGER statements that are not the part of a SQL2 standard but that should be taken into account while reversing DDL script of this dialect.

This dialect has some syntax differences from SQL2 standard because of extensions (e.g. some schema definition statements can have PROPERTIES clause). These extensions are ignored while reversing.

Oracle Oracle8

Oracle Oracle8 dialect has CREATE DATABASE, CREATE INDEX, and CREATE TRIGGER statements that are not the part of SQL2 standard but that should be taken into account while reversing DDL script of this dialect.

This dialect has some syntax differences from SQL2 standard because of extensions (e.g. some schema definition statements can have STORAGE clause). These extensions are ignored while reversing.

Oracle Oracle8 has object oriented DDL statements (CREATE TYPE and CREATE TYPE BODY). Additional object oriented Oracle8 schema objects are Object Types, Nested Object Types, Nested Table, VARRAY, Object Tables, and Object Views.
### Stereotypes for MagicDraw Constructs

The following table lists stereotypes that are used with MagicDraw UML constructs to represent a database structure:

<table>
<thead>
<tr>
<th>Model Item</th>
<th>Stereotype</th>
<th>Description</th>
<th>Default item stereotype for forward engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package</strong></td>
<td>&lt;&lt;database&gt;&gt;</td>
<td>See Section “Database” on page 2-39</td>
<td>If EnableDefaultStereotypes property is true, &lt;&lt;database&gt;&gt; stereotype is used for first level packages, and &lt;&lt;schema&gt;&gt; stereotype is used for second level packages; Otherwise none.</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;schema&gt;&gt;</td>
<td>See Section “Schema” on page 2-40</td>
<td></td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td>&lt;&lt;table&gt;&gt;</td>
<td>See Section “Table” on page 2-40</td>
<td>If EnableDefaultStereotypes property is true, &lt;&lt;table&gt;&gt; stereotype is used; Otherwise none.</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;view&gt;&gt;</td>
<td>See Section “View” on page 2-48</td>
<td></td>
</tr>
<tr>
<td><strong>Attribute</strong></td>
<td>&lt;&lt;PK&gt;&gt;</td>
<td>See Section “Primary Key” on page 2-43</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;unique&gt;&gt;</td>
<td>See Section “Uniqueness” on page 2-43</td>
<td></td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>&lt;&lt;PK&gt;&gt;</td>
<td>See Section “Primary Key” on page 2-43</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;unique&gt;&gt;</td>
<td>See Section “Operations” on page 2-33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;check&gt;&gt;</td>
<td>See Section “Check” on page 2-45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;index&gt;&gt;</td>
<td>See Section “Index” on page 2-46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;trigger&gt;&gt;</td>
<td>See Section “Trigger” on page 2-47</td>
<td></td>
</tr>
<tr>
<td><strong>Dependency</strong></td>
<td>&lt;FK&gt;</td>
<td>See Section “Relationship cardinalities” on page 2-35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;reference&gt;&gt;</td>
<td>See Section “Relationship cardinalities” on page 2-35</td>
<td></td>
</tr>
</tbody>
</table>
PROPERTIES OF CODE ENGINEERING SET FOR DDL

There are two separate properties sets, stored as properties of code engineering set for DDL:

- Properties for DDL script generation,
- Properties for DDL script reverse engineering.

Properties for DDL script reverse engineering and generation

![Diagram of CG Properties Editor dialog box. DDL properties]

<table>
<thead>
<tr>
<th>Property name</th>
<th>Values list</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Target DDL Script File Name</td>
<td>script</td>
<td></td>
</tr>
<tr>
<td>Target DDL Dialect Name</td>
<td>Standard SQL</td>
<td></td>
</tr>
<tr>
<td>Enable Default Stereotypes</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Generate Extended Index Name</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Generate Extended Trigger Name</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Generate Drop Statements</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Attribute Default Multiplicity</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Generate Null Constraint</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Generate Not Null Constraint</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Generate Index for Primary Key</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Generate Index for Unique</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Generate Quoted Identifiers</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>Generate Qualified Names</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Default Catalog Name</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Default Schema Name</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 35 -- CG Properties Editor dialog box. DDL properties*
<table>
<thead>
<tr>
<th>Property name</th>
<th>Values list</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column default nullability</td>
<td>Dialect default (default), not specified, NULL, NOT NULL</td>
<td>If column has no NULL or NOT NULL constraint specified, the value of this property is used.</td>
</tr>
<tr>
<td>Create catalog sets current catalog</td>
<td>True (default), false</td>
<td>Specifies whether create catalog statement changes current catalog name.</td>
</tr>
<tr>
<td>Create schema sets current schema</td>
<td>True (default), false</td>
<td>Specifies whether create schema statement changes current schema name.</td>
</tr>
<tr>
<td>Default catalog name</td>
<td>DefaultCatalogNone (default), DefaultCatalogPackage, any entered by the user</td>
<td>Specifies current database name. Used when DDL script does not specify database name explicitly.</td>
</tr>
<tr>
<td>Default schema name</td>
<td>DefaultSchemaNone (default), DefaultSchemaPackage, any entered by the user</td>
<td>Specifies current schema name. Used when DDL script does not specify schema name explicitly.</td>
</tr>
<tr>
<td>Drop statements</td>
<td>Deferred (default), Immediate, Ignored</td>
<td>Specifies whether execution of drop statements may be deferred, or must be executed, or must be ignored. Deferred drop may be enabled if elements are recreated later. This will save existing views. Attribute stereotypes, multiplicity and default value always are not dropped immediately.</td>
</tr>
<tr>
<td>Map Null/not Null constraints to</td>
<td>Stereotypes (default), Multiplicity</td>
<td>When parsing DDLs, the null/not null constraints are modeled as either stereotypes or multiplicity.</td>
</tr>
<tr>
<td>Map foreign keys</td>
<td>True (default), false</td>
<td>A dependency with &lt;&lt;-FK-&gt; stereotype is created, to represent Foreign Key.</td>
</tr>
<tr>
<td>Map indexes</td>
<td>True (default), false</td>
<td>An operation with &lt;&lt;-index-&gt; stereotype is added into class, to represent index.</td>
</tr>
<tr>
<td>Map triggers</td>
<td>True (default), false</td>
<td>An operation with &lt;&lt;-trigger-&gt; stereotype is added into class to represent trigger.</td>
</tr>
<tr>
<td>Map views</td>
<td>True (default), false</td>
<td>A class with &lt;&lt;-view-&gt; stereotype is created to represent view.</td>
</tr>
<tr>
<td>Generation features</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default attribute multiplicity</td>
<td>0, 0..1, any entered by user</td>
<td>If the attribute multiplicity is not specified, the value of this property is used.</td>
</tr>
<tr>
<td>Feature</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Generate Null constraint</td>
<td>True, false (default)</td>
<td>If true, generates NULL constraint for column attribute with ([0..1]) multiplicity. If DBMS, you use, support NULL, you can enable this to generate NULL constrains. See also: GenerateNotNullConstraint, AttributeDefaultMultiplicity</td>
</tr>
<tr>
<td>Generate extended index name</td>
<td>True, false (default)</td>
<td>If true, generates index name of the form: TableName_IndexName.</td>
</tr>
<tr>
<td>Generate extended trigger name</td>
<td>True, false (default)</td>
<td>If true, generates trigger name of the form: TableName_TriggerName.</td>
</tr>
<tr>
<td>Generate index for primary key</td>
<td>True (default), false</td>
<td>If the DBMS, you use, requires explicit indexes for primary key, you may enable explicit index creation using this flag. See also: GenerateIndexForUnique</td>
</tr>
<tr>
<td>Generate index for unique</td>
<td>True (default), false</td>
<td>If the DBMS, you use, requires explicit indexes for primary key or unique columns, may enable explicit index creation using this flag. See also: GenerateIndexForPK</td>
</tr>
<tr>
<td>Generate not Null constraint</td>
<td>True (default), false</td>
<td>If true, generates NOT NULL constraint for column attribute with ([1]) multiplicity. If you set GenerateNullConstraint, you may wish to do not generate NOT NULL constrain. See also: GenerateNullConstraint, AttributeDefaultMultiplicity</td>
</tr>
<tr>
<td>Generate qualified names</td>
<td>True (default), false</td>
<td>If value of Generate Qualified Names check box is true, package name is generated before the table or view name. For example: (&lt;&lt;\text{database}&gt;&gt;) package MQOnline includes (&lt;&lt;\text{table}&gt;&gt;) class libraries. Then check box Generate Qualified Names is selected as true in generated source would be written: DROP TABLE MQOnline.libraries; Then check box Generate Qualified Names is selected as false, in generated source would be written: DROP TABLE libraries;</td>
</tr>
<tr>
<td>Generate quoted identifiers</td>
<td>True, false (default)</td>
<td>Specifies whether DDL code generator should generate quoted names of identifiers.</td>
</tr>
</tbody>
</table>
### SUPPORTED SQL STATEMENTS

This section lists SQL statements that MagicDraw UML supports (that are parsed and mapped into UML constructs).

The following table provides SQL2 SQL schema statements that are supported and that are NOT supported in MagicDrawTM UML:

<table>
<thead>
<tr>
<th>SQL schema statement</th>
<th>Supported</th>
<th>(Yes/No)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQL schema definition statement</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schema definition</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Table definition</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>View definition</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Alter table statement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Grant statement</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Domain definition</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Assertion definition</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Character set definition</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Collation definition</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Translation definition</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>SQL schema manipulation statement</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drop table statement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop view statement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Revoke statement</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Alter domain statement</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Drop assertion statement</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Drop domain statement</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Drop character set statement</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Drop collation statement</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Drop translation statement</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Some SQL schema statements (e.g. schema definition, table definition) allow implicit catalog name and unqualified schema name. In addition to SQL schema statements, the following SQL session statements must be supported:

Object creation mode: The Object Creation Mode combo box has the following options:
- only CREATE statements
- DROP & CREATE statements
- CREATE OR REPLACE statements (only for Oracle dialect; default for this dialect)
- DROP IF EXISTS & CREATE statements (only for MySQL dialect; default for this dialect).
- Set catalog statement - sets the current default catalog name.
- Set schema statement - sets the current default unqualified schema name.

MagicDraw supports the following widely used by dialects statements that are not the part of SQL2:

- Database definition statement (CREATE DATABASE) that creates database
- Index statements (CREATE INDEX, DROP INDEX) that create an index on table and remove it
- Trigger statements (CREATE TRIGGER, DROP TRIGGER) that create a trigger on table and remove it.

The following table provides details on mapping on the supported SQL schema manipulation statements into MagicDraw constructs:
**TIPS**

**Short representation for primary key constraint**

When primary key is made out of several columns, and columns order is not important, short primary key representation may be used. Columns that make up the primary key must be marked with `<<<PK>>` stereotype. Because a table can contain only one (if any) primary key, these columns are concatenated into one primary key.

Short primary key representation using UML concepts and generated DDL script:
CREATE TABLE ShortPKExample (  
col1 INTEGER,  
col2 INTEGER,  
PRIMARY KEY (col1, col2)  
);

If the order of columns within the primary key is important, constraint representation as an operation with parameters must be used:

CREATE TABLE NoShortPKExample (  
col1 INTEGER,  
col2 INTEGER,  
PRIMARY KEY (col2, col1)  
);

Primary key constraint with overhead info

Consider such primary key definition using UML concepts and generated DDL script:

CREATE TABLE PKWithOverheadExample (  
col1 INTEGER UNIQUE,  
col2 INTEGER,  
PRIMARY KEY (col1, col2)  
);
This representation is valid, but the primary key contains overhead info (specifically column col2). Unique column alone may be used as a valid primary key:

```sql
CREATE TABLE PKWithoutOverheadExample (  
  col1 INTEGER PRIMARY KEY,  
  col2 INTEGER
);
```

Differences between UML Profile for CORBA specification and mapping in MagicDraw is listed below. Most differences are present because MagicDraw does not fully support UML 1.4.

- CORBA IDL exception is mapped to Core::Class, instead of CommonBehavior::Exception. MagicDraw does not support CommonBehavior::Exception.

- Constraints defined in UML Profile for CORBA specification are not checked in MagicDraw.

- For storing constants, UML class instead of UtilityClass is used.

- Use simplified “1” multiplicity instead of “1...1”.

- Stereotype CORBAAnonymousFixed is introduced. It is used to represent anonymous fixed types. Fixed types without names are mapped to inner classes with stereotype CORBAAnonymousFixed. These classes are binded with CORBA::fixed classes. IDL Code:

```idl
struct baz
{
    fixed <8, 4> high_scale;
    fixed <8, 2> low_scale;
};
```

This code is mapped to the following diagram:
MagicDraw presents a CORBA IDL diagram, which simplifies the creation of standard CORBA IDL elements. The following elements are available in the CORBA IDL diagram:

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut key</th>
<th>Model Element</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="M" /></td>
<td>M</td>
<td>CORBA IDL Module</td>
</tr>
<tr>
<td><img src="image" alt="I" /></td>
<td>I</td>
<td>CORBA IDL Interface</td>
</tr>
<tr>
<td><img src="image" alt="V" /></td>
<td>V</td>
<td>CORBA IDL Value</td>
</tr>
<tr>
<td><img src="image" alt="SHIFT+P" /></td>
<td>SHIFT+P</td>
<td>Class by Pattern</td>
</tr>
<tr>
<td>Button</td>
<td>Shortcut key</td>
<td>Model Element</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td>Generalization</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Truncatable Generalization</td>
</tr>
<tr>
<td></td>
<td>VS</td>
<td>Value Support Generalization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CORBA IDL Association</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>Association</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Aggregation</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>Composition</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>Interface</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Dependency</td>
</tr>
<tr>
<td>Button</td>
<td>Shortcut key</td>
<td>Model Element</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><img src="image1" alt="Binding Dependency" /></td>
<td>B</td>
<td>Binding Dependency</td>
</tr>
<tr>
<td><img src="image2" alt="Containment" /></td>
<td>SHIFT+C</td>
<td>Containment</td>
</tr>
<tr>
<td><img src="image3" alt="Separator" /></td>
<td>W</td>
<td>Separator</td>
</tr>
</tbody>
</table>
In MagicDraw, EJB-UML mapping is based on UML Profiles for EJB. The UML Profiles for EJB is a mapping from Unified Modeling Language (UML) to the Enterprise JavaBeans (EJB) architecture. The mapping specifies the standard representation for elements of the EJB architecture in UML models.

UML Profiles for EJB consist of Java, EJB Design, and EJB Deployment profiles. The Java Profile defines the subset of Java constructs that are needed to support the EJB Profiles. EJB Design Profile defines how to model EJB applications using the interfaces and the implementation classes of an enterprise bean. The EJB Deployment Profile defines mapping of EJB components contained in EJB-JAR archives and deployment on Application Servers to UML.

### Java Profile

This section describes stereotypes and tagged values introduced in the Java Profile.

#### Stereotypes

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Applies To</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>«JAR»</td>
<td>Artifact, Component</td>
<td>Specializes the standard UML Stereotype «file». Indicates that the Artifact or Component represents a JAR.</td>
</tr>
<tr>
<td>«JARInclude»</td>
<td>Dependency</td>
<td>Indicates that the supplier of the Dependency, a JAR, Java .class file, or other file, is included “by value” (i.e., copied into) the JAR that is the client of the Dependency</td>
</tr>
<tr>
<td>«JARReference»</td>
<td>Usage</td>
<td>Indicates that the supplier is a model element that is not contained in the archive but is referenced by it in the META-INF/MANIFEST.MF file</td>
</tr>
<tr>
<td>«JavaSourceFile»</td>
<td>«file»</td>
<td>Specializes the standard UML Stereotype «file». Indicates that the Artifact describes a Java Source File.</td>
</tr>
</tbody>
</table>

**NOTE** The compiled “*.class” file is not modeled by this profile.

#### Tagged Values

Tagged Values of the Java Profile are included in current implementation of MagicDraw but are not used in EJB 2.0 code engineering.

<table>
<thead>
<tr>
<th>Tagged Value</th>
<th>Applies To</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaStrictfp</td>
<td>Class</td>
<td>A Boolean value indicating whether or not the Java Class is FP-strict</td>
</tr>
</tbody>
</table>
### Stereotypes

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Applies To</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>«EJBBusiness»</td>
<td>Operation</td>
<td>Indicates that the Operation represents an <em>instance-level</em> business method, a method that supports the “business logic” of the EJB. Contrast with <em>class-level</em> business methods on home interfaces, which are stereotyped «EJBHome».</td>
</tr>
<tr>
<td>«EJBCmpField»</td>
<td>Attribute, AssociationEnd</td>
<td>Indicates that the Attribute or Association End represents a container-managed field for an EJB Entity Bean with container-managed persistence.</td>
</tr>
<tr>
<td>«EJBCmrMethod»</td>
<td>Dependency</td>
<td>A UML Dependency with the «EJBRelationshipRole» Association End as the client and the abstract get/set Methods of the relationship as the suppliers. This connects the EJB Relationship to the pair of container managed relationship methods, and allows the decoupling of the role name to the name of the CMR fields. (EJB 2.0 only)</td>
</tr>
<tr>
<td>«EJBCreate»</td>
<td>Operation</td>
<td>Indicates that the Operation represents an EJB Create Method.</td>
</tr>
<tr>
<td>Stereotype</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>«EJBEnterpriseBean»</td>
<td>Class</td>
<td>Specializes the standard UML Stereotype «implementationClass». An abstract Class that represents an EJB Enterprise Bean. The stereotype is applied to the implementation class for the EJB.</td>
</tr>
<tr>
<td>«EJBEntityBean»</td>
<td>Class</td>
<td>Indicates that the Class represents an EJB Entity Bean. Specializes «EJBEnterpriseBean».</td>
</tr>
<tr>
<td>«EJBFinder»</td>
<td>Operation</td>
<td>Indicates that the Operation represents an EJB Finder Method.</td>
</tr>
<tr>
<td>«EJBHome»</td>
<td>Operation</td>
<td>Indicates that the Operation represents an EJB Home Method, either local or remote, which is a class-level “business” method, as opposed to a “create”, “finder”, etc. method. Contrast with «EJBBusiness».</td>
</tr>
<tr>
<td>«EJBLocalMethod»</td>
<td>Operation</td>
<td>Indicates that the Operation represents a method that is exposed on either the local or local-home interface. The former case is assumed if the EJB Business stereotype is also present. (EJB 2.0 only)</td>
</tr>
<tr>
<td>«EJBLocalReference»</td>
<td>Dependency</td>
<td>A stereotyped Dependency representing an EJB Local Reference, where the client is an EJB-JAR and the supplier is an EJB Enterprise Bean. (EJB 2.0 only)</td>
</tr>
<tr>
<td>«EJBMessengerDrivenBean»</td>
<td>Class</td>
<td>Indicates that the Class represents an EJB Message Driven Bean. Specializes «EJBEnterpriseBean». (EJB 2.0 only)</td>
</tr>
<tr>
<td>«EJBPrimaryKey»</td>
<td>Usage</td>
<td>Indicates that the supplier of the Usage represents the EJB Primary Key Class for the EJB Enterprise Bean represented by the client.</td>
</tr>
<tr>
<td>«EJBPrimaryKeyField»</td>
<td>Attribute. AssociationEnd</td>
<td>Specializes «EJBcmpField». Indicates that the Attribute or Association End is the primary key field for an EJB Entity Bean with container-managed persistence.</td>
</tr>
<tr>
<td>«EJBRealizeHome»</td>
<td>Abstraction</td>
<td>Indicates that the supplier of the Abstraction represents an EJB Remote Home Interface for the EJB Enterprise Bean Class represented by the client.</td>
</tr>
<tr>
<td>«EJBRealizeLocal»</td>
<td>Abstraction</td>
<td>Indicates that the supplier of the Abstraction represents an EJB Local Interface for the EJB Enterprise Bean Class represented by the client. (EJB 2.0 only)</td>
</tr>
<tr>
<td>«EJBRealizeLocalHome»</td>
<td>Abstraction</td>
<td>Indicates that the supplier of the Abstraction represents an EJB Local Home Interface for the EJB Enterprise Bean Class represented by the client. (EJB 2.0 only)</td>
</tr>
<tr>
<td>«EJBRealizeRemote»</td>
<td>Abstraction</td>
<td>Indicates that the supplier of the Abstraction represents an EJB Remote Interface for the EJB Enterprise Bean Class represented by the client.</td>
</tr>
</tbody>
</table>
**Tagged Values**

<table>
<thead>
<tr>
<th>Tagged Value</th>
<th>Applies To</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJBAbstractSchemaName</td>
<td>Class «EJBEntityBean»</td>
<td>A String representing the abstract name used for a schema associated with a CMP 2.0 entity bean.</td>
</tr>
<tr>
<td>EJBAcknowledgeMode</td>
<td>Class «EJBMessageDrivenBean»</td>
<td>An enumeration with values Auto Acknowledge or Dups OK Acknowledge indicating the type of message acknowledgment used for the onMessage message of a message-driven bean that uses bean-managed transaction demarcation.</td>
</tr>
<tr>
<td>EJBCOMPONENTINTERFACE</td>
<td>Dependency «EJBReference» or «EJBLocalReference»</td>
<td>The name of the EJB Enterprise Bean’s “Component” Interface (local or remote).</td>
</tr>
<tr>
<td>EJB CmpVersion</td>
<td>Class «EJBEntityBean»</td>
<td>An enumeration with values 1.x or 2.x. Indicates the type of CMP used by the EJB Entity Bean</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EJB Cmr FieldType</td>
<td>AssociationEnd «EJBRelationshipRole»</td>
<td>Type expression property of the supplier UML Association End «EJBRelationshipRole».</td>
</tr>
<tr>
<td>EJB Display Name</td>
<td>Class «EJBEnterpriseBean»</td>
<td>A String with the name for the EJB to be displayed by tools</td>
</tr>
<tr>
<td>EJB Env Entries</td>
<td>Class «EJBEnterpriseBean»</td>
<td>A string of XML tags, designating the environment entries used by the EJB Enterprise Bean</td>
</tr>
<tr>
<td>EJB Home Interface</td>
<td>Dependency «EJBReference» or «EJBLocalReference»</td>
<td>The name of the EJB Enterprise Bean’s Home Interface (local or remote).</td>
</tr>
<tr>
<td>EJB Link</td>
<td>Dependency «EJBReference» or «EJBLocalReference»</td>
<td>The name of the referenced EJB Enterprise Bean, if it is in the same Archive or another archive in the same J2EE Application Unit, or a path to the bean.</td>
</tr>
<tr>
<td>EJB Message Driven Destination</td>
<td>Class «EJBMessageDrivenBean»</td>
<td>An XML tag, designating the type of destination and the durability used by the EJB Message Driven Bean.</td>
</tr>
<tr>
<td>EJB Message Selector</td>
<td>Class «EJBMessageDrivenBean»</td>
<td>A String specifying the JMS message selector to be used in determining which messages a message-driven bean is to receive.</td>
</tr>
<tr>
<td>EJB Name In JAR</td>
<td>Class «EJBEnterpriseBean»</td>
<td>The name used for the EJB Enterprise Bean in the EJB-JAR. Defaults to the name of the EJB Remote Interface.</td>
</tr>
<tr>
<td>EJB Persistence Type</td>
<td>Class «EJBEntityBean»</td>
<td>An enumeration with values Bean or Container. Indicates whether the persistence of the EJB Entity Bean is managed by the EJB Entity Bean or by its container, respectively.</td>
</tr>
<tr>
<td>EJB Query String</td>
<td>Operation «EJB Finder» or «EJB Select»</td>
<td>The EJB QL statement corresponding to the method. It is ignored for BMP Entity Beans</td>
</tr>
<tr>
<td>EJB Reentrant</td>
<td>Class «EJBEntityBean»</td>
<td>A Boolean value indicating whether or not the EJB Entity Bean can be called re-entrantly.</td>
</tr>
</tbody>
</table>
EJB Deployment Profile

Stereotypes

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Applies To</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>«EJB-JAR»</td>
<td>Artifact, Component</td>
<td>Specializes the Stereotype «JAR». Indicates that the Artifact represents an EJB JAR.</td>
</tr>
<tr>
<td>Stereotype</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>«EJBClientJAR»</td>
<td>Usage</td>
<td>Indicates that the client of the Usage represents an ejb-client-jar for the EJB-JAR represented by the supplier of the Usage.</td>
</tr>
<tr>
<td>«EJBContainerTransaction»</td>
<td>Actor</td>
<td>Indicates the type of transaction used for associated Methods. It is combined with a transaction type stereotype, such as «EJBRequired». (See the following Table.) Container transactions are specified using one or more dependencies, stereotyped «EJBMethod», from an EJB Enterprise Bean client to an EJB Container Transaction Actor supplier.</td>
</tr>
<tr>
<td>«EJBEnterpriseBeanDeployment»</td>
<td>Component</td>
<td>Indicates that the Component represents an EJB Enterprise Bean. It resides in an EJB-JAR. It has tagged values for overriding default settings in the corresponding EJB Enterprise Bean Class. For example, these tagged values can be used to create a new EJB by specifying different EJB Home and EJB Remote interfaces for a shared EJB Enterprise Bean Class.</td>
</tr>
<tr>
<td>«EJBEntityBeanDeployment»</td>
<td>Component</td>
<td>Specializes the standard Stereotype «EJBEnterpriseBean». Indicates that the Component represents an EJB Entity Bean.</td>
</tr>
<tr>
<td>«EJBExcludeList»</td>
<td>Actor</td>
<td>Indicates an Actor that represents an Exclude List. An exclude list is specified using one or more dependencies, stereotyped «EJBMethod», from an EJB Enterprise Bean client to an EJB Exclude List Actor supplier.</td>
</tr>
<tr>
<td>«EJBMessageDrivenBeanDeployment»</td>
<td>Component</td>
<td>Specializes the standard Stereotype «EJBEnterpriseBean». Indicates that the Component represents an EJB Message Driven Bean.</td>
</tr>
<tr>
<td>«EJBMethod»</td>
<td>Dependency</td>
<td>Indicates a Dependency between an EJB Component client and an EJB Method Permission (or EJB Container Transaction or EJBExcludeList) supplier that completes a Method Permission (or Container Transaction or Exclude List) specification. It contains a method specification, either “style 1, 2, or 3”, according to the EJB 2.0 specification.</td>
</tr>
<tr>
<td>Class Name</td>
<td>Role</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>«EJBMethodPermission»</td>
<td>Actor</td>
<td>Indicates an Actor that represents a Method Permission. It contains a tag value indicating whether or not the method is “unchecked”. One or more method permissions are specified using one or more dependencies, stereotyped «EJBMethod», from an EJB Enterprise Bean to an EJB Method Permission Actor.</td>
</tr>
<tr>
<td>«EJBOverrideHome»</td>
<td>«reside»</td>
<td>Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Home Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.</td>
</tr>
<tr>
<td>«EJBOverrideLocal»</td>
<td>«reside»</td>
<td>Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Local Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.</td>
</tr>
<tr>
<td>«EJBOverrideLocalHome»</td>
<td>«reside»</td>
<td>Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Local Home Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.</td>
</tr>
<tr>
<td>«EJBOverridePrimaryKey»</td>
<td>«reside»</td>
<td>Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier Java Class overrides the EJB Primary Key Class that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.</td>
</tr>
<tr>
<td>«EJBOverrideRemote»</td>
<td>«reside»</td>
<td>Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Remote Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.</td>
</tr>
<tr>
<td>«EJBRoleName»</td>
<td>Actor</td>
<td>Indicates the name of a security role used in the definitions of Method Permissions, etc.</td>
</tr>
</tbody>
</table>

© 2005 No Magic, Inc.
The **EJBContainerTransaction** Actor model elements always receive a second stereotype, which identifies the type of transaction.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>«EJBMandatory»</td>
<td>Transactions are mandatory for this method</td>
</tr>
<tr>
<td>«EJBNever»</td>
<td>Transactions are never used for this method</td>
</tr>
<tr>
<td>«EJBNotSupported»</td>
<td>Transactions aren’t supported for this method</td>
</tr>
<tr>
<td>«EJBRequired»</td>
<td>Transactions are required for this method</td>
</tr>
<tr>
<td>«EJBRequiresNew»</td>
<td>A new transaction is required for this method</td>
</tr>
<tr>
<td>«EJBSupports»</td>
<td>Transactions are supported for this method</td>
</tr>
</tbody>
</table>

### Tagged Values

<table>
<thead>
<tr>
<th>Tagged Value</th>
<th>Applies To</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJBDisplayName</td>
<td>Artifact «EJB-JAR»</td>
<td>A String with the name for the Archive to be displayed by tools</td>
</tr>
<tr>
<td>EJBMethodDescriptor</td>
<td>Dependency «EJBMethod»</td>
<td>A String with the subset of XML required to specify a complete deployment descriptor &lt;method&gt; tag for a method or set of methods, using “Style 1, 2, or 3” syntax, as defined by the EJB 2.0 Specification.</td>
</tr>
<tr>
<td>EJBUnchecked</td>
<td>Actor «EJBMethodPermission»</td>
<td>A Boolean that when “true”, indicates that permission to invoke the specified methods should not be checked before invocation. Default is “false”</td>
</tr>
<tr>
<td>EJBUseCallerIdentity</td>
<td>Component «EJBEnterpriseBean»</td>
<td>A <strong>Boolean</strong> indicating whether or not to use the caller’s identity when invoking the bean methods.</td>
</tr>
</tbody>
</table>
USING MAGICDRAW EJB 2.0

Reverse engineering

1 Create EJB 2.0 code engineering set.

   NOTE If the project does not contain profiles required for EJB code engineering, you will be asked whether you would like to import the EJB 2.0 template. It is recommended to select Yes.

2 Add java files and EJB deployment descriptor file that are part of concrete EJB. EJB deployment descriptor file ejb-jar.xml must be in META-INF directory.

3 Reverse those files.

After the reverse, you will get the following result:
Model elements representing the code will be created. Stereotypes and tagged values representing EJB tags will be applied to appropriate model elements.

Code generation

We will create a simple example and we will go step by step from the beginning to the generated code. As an example we will take a reservation system for a theater ticket reservation. It will consist of single session enterprise bean ReservationBean. You may find this example in the <MagicDraw installation directory>, examples folder.

1 Create a new MagicDraw project from EJB 2.0 template.

2 Create a new class diagram.

3 Create an interface Reservation that extends javax::ejb:EJBObject interface in the class diagram.

4 Create operation bookSeats (playName : java::lang::String, userName : java::lang::String, quantity : java::lang::String) in the bean’s class. The operation should throw java::rmi::RemoteException. To specify which exceptions the method throws, open the Operation Specification dialog box, click the Language Properties button and in the CG Properties Editor dialog box, Java tab specify the Throws Exceptions property (java.rmi.RemoteException).

5 Create a class with name ReservationBean. The class will be an enterprise session bean.

6 Make the class realizes SessionBean interface from javax.ejb package.

7 Open the ReservationBean’s specification and add EJBSessionBean stereotype to the class and EJBNameInJAR tagged value - ReservationEJB.
8 Create operation bookSeats( playName : java::lang::String, userName : java::lang::String, quantity : java::lang::String ) in the bean’s class. You can copy the operation from the Reservation interface (select the operation in the Reservation interface press Ctrl button and drag the operation to bean’s class.

9 Create an interface ReservationHome that extends javax::ejb::EJBHome interface and has operation public Reservation create() throws CreateException, RemoteException.

10 Create abstraction links from the bean’s class to the Reservation and Reservation Home interfaces. On the abstraction between ReservationBean and Reservation add stereotype EJBRealizeHome. On the abstraction between ReservationBean and ReservationHome interface add stereotype EJBRealizeRemote.

11 Create a new EJB 2.0 code engineering set.

12 In the containment tree select ReservationBean, Reservation and ReservationHome and drag them to the set.

13 In the ComponentView select ReservationBean.java component, open specification for it and add stereotype EJBSessionBeanDeployment.

14 In the Component View package create new component ejb-jar.jar. Open specification and add stereotype EJB-JAR to it.

15 Create a new Implementation Diagram.

16 Drag ReservationBean.java and ejb-jar.jar components to the diagram.

17 Create dependency link from the ReservationBean.java component to ejb-jar.jar component and add stereotype implement to the dependency.

18 Drag ejb-jar.jar component from the in the Containment Tree from Component View to the EJB 2.0 code engineering set.

19 Generate the code.
XML SCHEMA

Reference: [http://www.w3.org/TR/xmlschema-2/](http://www.w3.org/TR/xmlschema-2/)

**XML SCHEMA MAPPING TO UML ELEMENTS**

**Defined stereotypes**

<table>
<thead>
<tr>
<th>Stereotype name</th>
<th>Base Stereotype</th>
<th>Applies to</th>
<th>Defined TagDefinitions</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSD component</td>
<td>Class Attribute</td>
<td>id – string</td>
<td>The base and abstract stereotype for all XML Schema stereotypes used in UML profile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AssociationEnd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Generalization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Component</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XSD attribute</td>
<td>fixed – some fixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>element value</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>form - *(qualified</td>
<td>unqualified)*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>refString – string</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>representation of</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reference to other</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>attribute.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ref – actual reference</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>to other attribute</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>use - *(optional</td>
<td>prohibited</td>
<td>required)* : optional</td>
</tr>
<tr>
<td>XML Schema Element</td>
<td>XSD Component</td>
<td>Attribute</td>
<td>Association End</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------</td>
<td>-----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| XSDelement         | XSD component | Attribute | AssociationEnd   | abstract – (true | false)  
|                    |               |           |                  | block - (extension | restriction | substitution) 
|                    |               |           |                  | final - (extension | restriction) 
|                    |               |           |                  | fixed – some fixed element value 
|                    |               |           |                  | form - (qualified | unqualified)  
|                    |               |           |                  | nillable – (true | false)  
|                    |               |           |                  | refString – string representation of reference to other attribute. 
|                    |               |           |                  | ref – actual reference to other attribute 
|                    |               |           |                  | substitutionGroup – actual reference to UML ModelElement 
|                    |               |           |                  | substitutionGroupString – string representation of substitution group 
|                    |               |           |                  | key_unique_keyRef – a list of referenced UML Attributes 
|                    |               |           |                  | sequenceOrder – a number in sequence order |
| XSDcomplexType     | XSD component | Class     |                  | block – (extension | restriction) 
|                    |               |           |                  | final – (extension | restriction) 
|                    |               |           |                  | mixed – (true | false)  
| XSDsimpleContent   | Class         |           |                  | simpleContentId – string |
| XSDcomplexType     | XSD component | Class     |                  | complexContentId – string 
| XSDcomplexType     | XSD component | Class     |                  | complexContentMixed |
| XSDgroup           | XSD component | Class     |                  | |
| XSDgroupRef        | XSD component | Attribute | AssociationEnd   | sequenceOrder – a number in sequence order |
| XSDall             | Class         |           |                  | allId – string 
|                    |               |           |                  | maxOccurs 
<p>|                    |               |           |                  | minOccurs |</p>
<table>
<thead>
<tr>
<th>Component Name</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSDChoice</td>
<td>Class</td>
<td>choiceId – string maxOccurs minOccurs sequenceOrder – a number in sequence order</td>
</tr>
<tr>
<td>XSDSequence</td>
<td>Class</td>
<td>sequenceld – string maxOccurs minOccurs sequenceOrder – a number in sequence order</td>
</tr>
<tr>
<td>XSDRestriction</td>
<td>XSDcomponent</td>
<td>Generalization</td>
</tr>
<tr>
<td>XSDExtension</td>
<td>XSDcomponent</td>
<td>Generalization</td>
</tr>
<tr>
<td>XSDAttributeGroup</td>
<td>XSDcomponent</td>
<td>Class</td>
</tr>
<tr>
<td>XSDsimpleType</td>
<td>XSDcomponent</td>
<td>Class</td>
</tr>
<tr>
<td>XSDList</td>
<td>XSDcomponent</td>
<td>Class</td>
</tr>
<tr>
<td>XSDUnion</td>
<td>XSDcomponent</td>
<td>Class</td>
</tr>
<tr>
<td>XSDAnnotation</td>
<td>XSDcomponent</td>
<td>Comment</td>
</tr>
<tr>
<td>XSDAny</td>
<td>XSDcomponent</td>
<td>Attribute</td>
</tr>
<tr>
<td>XSDAnyAttribute</td>
<td>XSDcomponent</td>
<td>Attribute</td>
</tr>
<tr>
<td>XSDSchema</td>
<td>XSDcomponent</td>
<td>Class</td>
</tr>
<tr>
<td>XSDNotation</td>
<td>XSDcomponent</td>
<td>Attribute</td>
</tr>
<tr>
<td>XSDRedefine</td>
<td>XSDcomponent</td>
<td>Class</td>
</tr>
</tbody>
</table>
XML Schema

- XML schema attribute maps to UML Attribute with stereotype XSDAttribute.
- default maps to initial UML Attribute or AssociationEnd value.
- annotation – to UML Attribute or AssociationEnd documentation.
- name – to UML Attribute or AssociationEnd name.
- type or content simpleType – to UML Attribute or AssociationEnd type.
Other attributes or elements maps to corresponding tagged values.

```xml
<attribute
  default = string
  fixed = string
  form = (qualified | unqualified)
  id = ID
  name = NCName
  ref = QName
  type = QName
  use = (optional | prohibited | required) : optional
  (any attributes with non-schema namespace . . . )>
  Content: (annotation?, (simpleType?))
</attribute>
```

Example:
```xml
<xs:attribute name="age" type="xs:positiveInteger" use="required"/>
```

ref value is generated from ref or refString TaggedValue.

One of ref or name must be present, but not both.

If ref is present, then all of <simpleType>, form and type must be absent.

type and <simpleType> must not both be present.

attribute UML Model example

```xml
  <xs:attribute name="address" fixed="fixed_value", form=qualified, use=optional">
  <xs:attribute name="name" string = mnde{fixed=fixed_value, form=qualified, use=optional}>
  <xs:attribute name="surname" string
</xs:schema>
```
<xs:attribute name = "name" type = "xs:string" default = "minde" fixed = "fixed_value" form = "qualified" use = "optional" >
   <xs:documentation >name attribute documentation</xs:documentation>
   </xs:annotation>
</xs:attribute>
<xs:attribute name = "address" fixed = "fixed_value" form = "qualified" use = "optional" >
   <xs:annotation >
      <xs:documentation >surname attribute documentation</xs:documentation>
   </xs:annotation>
</xs:attribute>
<xs:attribute name = "surname" type = "xs:string" />
<xs:attributeGroup name = "attr_group" >
   <xs:attribute ref = "nm:name" >
      <xs:annotation >
         <xs:documentation >reference documentation</xs:documentation>
      </xs:annotation>
   </xs:attribute>
   <xs:attribute ref = "nm:surname" />
</xs:attributeGroup>
</xs:schema>

**element**

Maps to UML Attribute or UML AssociationEnd with stereotype XSDelement.

- annotation – to UML Attribute or UML AssociationEnd documentation.
- *default* - to initial UML Attribute or UML AssociationEnd value.
- maxOccurs - to multiplicity upper range. Value *unbounded* maps to asterisk in UML.
- minOccurs – to multiplicity lower range.
- name – to UML Attribute or UML AssociationEnd name.
- type or content (simpleType | complexType) – to UML Attribute or UML AssociationEnd type.

Other properties maps to corresponding tagged values.
XML Representation Summary: **element** Element Information Item

```xml
<element
  abstract = boolean : false
  block = (#all | List of (extension | restriction | substitution))
  default = string
  final = (#all | List of (extension | restriction))
  fixed = string
  form = (qualified | unqualified)
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded) : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  nillable = boolean : false
  ref = QName
  substitutionGroup = QName
  type = QName
  (any attributes with non-schema namespace . . .)>
  Content: (annotation?, ((simpleType | complexType)?, (unique | key | keyref
</element>
```

ref value is generated from ref or refString TaggedValue.

One of ref or name must be present, but not both.

If ref is present, then all of `<complexType>`, `<simpleType>`, `<key>`, `<keyref>`, `<unique>`, nillable, default, fixed, form, block and type must be absent, i.e. only minOccurs, maxOccurs, id are allowed in addition to ref, along with `<annotation>`.

**Example**

```xml
<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
<xs:element name="gift">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="birthday" type="xs:date"/>
      <xs:element ref="PurchaseOrder"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```
element UML Model example

```xml
  <xs:element name = "name" type = "xs:string" default = "minde" id = "elementID" abstract = "true" block = "extension" nillable = "true" substitutionGroup = "nm:count" >
    <xs:annotation >
      <xs:documentation >element name documentation</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name = "count" >
    <xs:annotation >
      <xs:documentation >element count documentation</xs:documentation>
    </xs:annotation>
    <xs:restriction base = "xs:integer" />
  </xs:element>
  <xs:element name = "hour" type = "xs:integer" />
  <xs:element name = "minute" type = "xs:integer" substitutionGroup = "nm:count" />
  <xs:element name = "surname" type = "xs:string" minOccurs = "1" maxOccurs = "unbounded" />
  <xs:complexType name = "some_type" >
    <xs:all >
      <xs:element ref = "nm:hour" minOccurs = "0" maxOccurs = "1" >
        <xs:annotation >
          <xs:documentation >hour ref
documentation</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element ref = "nm:minute" minOccurs = "0" maxOccurs = "1" />
    </xs:all>
  </xs:complexType>
</xs:schema>
```
complexType

Complex type maps to UML Class with stereotype XSDcomplexType.

- abstract - to UML Class abstract value(true | false).
- annotation - to UML Class documentation.
- attribute – to inner UML Class Attribute or UML Association End.
- attributeGroup – to UML AssociationEnd or UML Attribute with type XSDattributeGroup.
- name – to UML Class name.

This class also can have stereotypes XSDsimpleContent, XSDcomplexType, XSDall, XSDchoice, XSDsequence.

No stereotype – the same as “XSDsequence”.

Generalization between complex type and other type has stereotype XSDrestriction or XSDextension. We assume stereotype XSDextension if generalization do not have stereotype.

Some complex mapping:

- complexType with simpleContent – to UML Class. This class must be derived from other class and can must have stereotype XSDsimpleContent.
- complexType with complexContent – to UML Class. This class must be derived from other class and must have stereotype XSDcomplexType.
- complexType with group, all, choice or sequence – to UML class with appropriate stereotype.

```xml
<complexType
  abstract = boolean : false
  block = (#all | List of (extension | restriction))
  final = (#all | List of (extension | restriction))
  id = ID
  mixed = boolean : false
  name = NCName
    (any attributes with non-schema namespace . . .)> Content: (annotation?, (simpleContent | complexContent | ((group | all
    ((attribute | attributeGroup)*, anyAttribute?))))))
</complexType>
```
When the `<simpleContent>` alternative is chosen, the following elements are relevant, and the remaining property mappings are as below. Note that either `<restriction>` or `<extension>` must be chosen as the content of `<simpleContent>`.

```xml
<simpleContent
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</simpleContent>
```

When the `<complexContent>` alternative is chosen, the following elements are relevant (as are `<attributeGroup>` and `<anyAttribute>` elements, not repeated here), and the additional property mappings are as below. Note that either `<restriction>` or `<extension>` must be chosen as the content of `<complexContent>`, but their content models are different in this case from the case above when they occur as children of `<simpleContent>`.

```xml
<complexContent
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | totalDigits | fractionDigits | length | minLength | maxLength | whiteSpace | pattern)*)?, ((attribute | attributeGroup)*, anyAttribute?))
</complexContent>
```

When the `<complexType>` alternative is chosen, the following elements are relevant (as are the `<attributeGroup>` and `<anyAttribute>` elements, not repeated here), and the additional property mappings are as below. Note that either `<restriction>` or `<extension>` must be chosen as the content of `<complexType>`, but their content models are different in this case from the case above when they occur as children of `<simpleContent>`.

The property mappings below are also used in the case where the third alternative (neither `<simpleContent>` nor `<complexContent>`) is chosen. This case is understood as shorthand for
complex content restricting the `ur-type definition`, and the details of the mappings should be modified as necessary.

```xml
<complexContent
  id = ID
  mixed = boolean
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</complexContent>

<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (group | all | choice | sequence)?, ((attribute | attributeGroup)*, anyAttribute?))
</restriction>

<extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((group | all | choice | sequence)?, ((attribute | attributeGroup)*, anyAttribute?)))
</extension>
```
complexType UML Model example

```xml
  <xs:complexType name = "my_Type2" block = "extension" final = "extension" mixed = "true" >
    <xs:annotation>
      <xs:documentation >my_type2 documentation</xs:documentation>
    </xs:annotation>
    <xs:attribute name = "name" />
    <xs:anyAttribute name = "unnamed attr1" />
  </xs:complexType>
</xs:schema>
```
<xs:complexContent id = "contentID" mixed = "false" >
  <xs:extension base = "nm:my_Type" >
    <xs:attribute name = "surname" type = "xs:string" />
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name = "my_Type3" >
  <xs:complexContent >
    <xs:restriction base = "nm:my_Type" >
      <xs:all >
        <xs:element name = "order" type = "xs:string" />
        <xs:element name = "order1" type = "xs:string" />
      </xs:all>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name = "my_Type4" >
  <xs:simpleContent >
    <xs:restriction base = "xs:string" >
      <xs:minLength value = "2" />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name = "my_Type5" >
  <xs:simpleContent >
    <xs:extension base = "xs:string" >
      <xs:attribute name = "attri" type = "xs:string" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name = "my_Type" abstract = "true" block = "extension" final = "extension" id = "myTypeID" mixed = "true" >
  <xs:annotation >
    <xs:documentation >my_type documentation</xs:documentation>
  </xs:annotation>
  <xs:attribute name = "name" type = "xs:string" />
  <xs:attributeGroup ref = "nm:attr_group" />
  <xs:anyAttribute />
</xs:complexType>
<xs:attributeGroup name = "attr_group" />
</xs:schema>

attributeGroup

attributeGroup maps to simple UML Class with stereotype XSDAttributeGroup.
- name – to UML Class name
- annotation – to UML Class documentation
- attribute – to inner UML Attribute or AssociationEnd with XSDAttribute stereotype.
• attributeGroup - inner attributeGroup always must be just reference. Such reference maps to Attribute or AssociationEnd with type of referenced attributeGroup. The opposite Association End kind must be aggregated and it must be navigable.

• anyAttribute – to inner UML Attribute with stereotype XSDanyAttribute.

If reference is generated, name is not generated.

**XML Representation Summary: attributeGroup Element Information Item**

```xml
<attributeGroup
  id = ID
  name = NCName
  ref = QName
  (any attributes with non-schema namespace . . .)>  
  Content: (annotation?, ((attribute | attributeGroup)*, anyAttribute?)
</attributeGroup>
```

When an `<attributeGroup>` appears as a daughter of `<schema>` or `<redefine>`, it corresponds to an attribute group definition as below. When it appears as a daughter of `<complexType>` or `<attributeGroup>`, it does not correspond to any component as such.

**attributeGroup UML Model example**

```xml
  <xs:attributeGroup name = "global_attr_group" >
    <xs:attribute name = "address" type = "xs:string" />
  </xs:attributeGroup>
  <xs:attributeGroup name = "attr_group_name" >
    <xs:annotation >
      <xs:documentation >attribute group documentation</xs:documentation>
    </xs:annotation> 
    <xs:attribute name = "surname" type = "xs:string" />
  </xs:attributeGroup>
</xs:schema>
```
simpleType

Maps to UML Class with stereotype XSDsimpleType.

XML Representation Summary: *simpleType* Element Information Item

```
<simpleType
    final = (#all | (list | union | restriction))
    id = ID
    name = NCName
    {any attributes with non-schema namespace . . .}>  
    Content: (annotation?, (restriction | list | union))
</simpleType>
<restriction
    base = QName
    id = ID
    {any attributes with non-schema namespace . . .}>  
    Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxEx |
        maxInclusive | totalDigits | fractionDigits | length | minLength | maxLen |
        whiteSpace | pattern)*))
</restriction>
<list
    id = ID
    itemType = QName
    {any attributes with non-schema namespace . . .}>  
    Content: (annotation?, (simpleType?))
</list>
<union
    id = ID
    memberTypes = List of QName
    {any attributes with non-schema namespace . . .}>  
    Content: (annotation?, (simpleType*))
</union>
```
Example

```xml
<xs:simpleType name="farenheitWaterTemp">
  <xs:restriction base="xs:number">
    <xs:fractionDigits value="2"/>
    <xs:minExclusive value="0.00"/>
    <xs:maxExclusive value="100.00"/>
  </xs:restriction>
</xs:simpleType>
```

The XML representation of a simple type definition.

restriction

To specify restriction generalization must be used between this class and super class. This generalization has or do not have XSD:restriction stereotype. Restriction id and annotation maps to Generalization properties.

In order to have inner simpleType element, parent of this Generalization must be inner Class of outer UML Class.

list

UML Class must have additional stereotype XSD:list.

Binding between this class and XSD:list must be provided.

“itemsType” maps to UML TemplateArgument from Binding.

union

UML Class must have additional stereotype XSD:union.

“memberTypes” and inner simpleTypes maps to several UML Generalizations between this simple type and members types.

In order to have inner simpleType element, parent of this Generalization must be inner Class of outer UML Class.
simpleType UML Model example

restriction example

```xml
  <xs:simpleType name = "farenheitWaterTemp" >
    <xs:annotation >
      <xs:documentation >documentation of simple type</xs:documentation>
    </xs:annotation >
  </xs:simpleType>

  <xs:simpleType name = "sex" >
    <xs:restriction base = "xs:string" >
      <xs:annotation >
        <xs:documentation >documentation of restriction</xs:documentation>
      </xs:annotation >
    </xs:simpleType>

  <xs:annotation >
    <xs:documentation >pattern doc</xs:documentation>
  </xs:annotation >
</xs:schema>
```
<xs:annotation>
  <xs:documentation>white space</xs:documentation>
</xs:annotation>
</xs:pattern>
<xs:whiteSpace id = "white_spaceid" fixed = "true" value = "preserve" >
  <xs:annotation >
    <xs:documentation >white space</xs:documentation>
  </xs:annotation>
</xs:whiteSpace>
</xs:documentation>
</xs:whiteSpace>
<xs:maxLength id = "maxlengthID" fixed = "false" value = "50" >
  <xs:annotation >
    <xs:documentation >max length</xs:documentation>
  </xs:annotation>
</xs:maxLength>
<xs:minLength id = "minlengthID" fixed = "true" value = "2" >
  <xs:annotation >
    <xs:documentation >min length</xs:documentation>
  </xs:annotation>
</xs:minLength>
<xs:length id = "lengthID" fixed = "true" value = "10" >
  <xs:annotation >
    <xs:documentation >length</xs:documentation>
  </xs:annotation>
</xs:length>
<xs:fractionDigits id = "fractionDigitsID" fixed = "true" value = "1" >
  <xs:annotation >
    <xs:documentation >fraction digits</xs:documentation>
  </xs:annotation>
</xs:fractionDigits>
<xs:totalDigits id = "totalDigitsID" fixed = "false" value = "8" >
  <xs:annotation >
    <xs:documentation >total digits</xs:documentation>
  </xs:annotation>
</xs:totalDigits>
<xs:maxInclusive id = "maxinclusiveid" fixed = "true" value = "100" >
  <xs:annotation >
    <xs:documentation >max inclusive</xs:documentation>
  </xs:annotation>
</xs:maxInclusive>
<xs:minInclusive id = "mininclusiveid" fixed = "true" value = "100" >
<xs:annotation>
  <xs:documentation>min inclusive documentation</xs:documentation>
</xs:annotation>
<xs:minInclusive>
  <xs:annotation>
    <xs:documentation>min inclusive documentation</xs:documentation>
  </xs:annotation>
  <xs:maxExclusive id="maxexclusiveid" fixed="true" value="101">
    <xs:annotation>
      <xs:documentation>max exclusive documentation</xs:documentation>
    </xs:annotation>
  </xs:maxExclusive>
</xs:minInclusive>
<xs:minExclusive id="id" fixed="true" value="99">
  <xs:annotation>
    <xs:documentation>min exclusive documentation</xs:documentation>
  </xs:annotation>
</xs:minExclusive>
</xs:restriction>
<xs:simpleType name="dayTime">
  <xs:annotation>
    <xs:documentation>day time documentation</xs:documentation>
  </xs:annotation>
  <xs:restriction>
    <xs:annotation>
      <xs:documentation>restriction documentation</xs:documentation>
    </xs:annotation>
    <xs:simpleType base="xs:number"/>
    <xs:enumeration value="day">
      <xs:annotation>
        <xs:documentation>day value</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
    <xs:enumeration value="night">
      <xs:annotation>
        <xs:documentation>night value</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="sex" final="restriction">
  <xs:annotation>
    <xs:documentation>documentation of simple type restriction documentation</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration id="some_id" value="male"/>
    <xs:enumeration value="female">
      <xs:annotation>
        <xs:documentation>female value</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
  </xs:restriction>
</xs:simpleType>
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="my_number_list2">
    <xs:list />
  </xs:simpleType>

  <xs:simpleType name="my_number_list">
    <xs:annotation>
      <xs:documentation>my list documentation</xs:documentation>
    </xs:annotation>
    <xs:list itemType="xs:boolean" />
  </xs:simpleType>
</xs:schema>
union example

```xml
    <xs:simpleType name = "my_simple_union" >
        <xs:union id = "unionID" memberTypes = "xs:string xs:number" />
    </xs:simpleType>
    <xs:simpleType name = "my_simple_union2" >
        <xs:annotation >
            <xs:documentation >very important documentation</xs:documentation>
        </xs:annotation>
        <xs:union id = "unionID" memberTypes = "xs:number" >
            <xs:simpleType >
                <xs:restriction base = "xs:number" />
            </xs:simpleType>
        </xs:union>
    </xs:simpleType>
</xs:schema>
```

**minExclusive**

Maps to UML Attribute with stereotype XSD\texttt{minExclusive}. Name and type of such attribute does not make sense.

- value – to Attribute initial value.
**XML Representation Summary: minExclusive Element Information Item**

```
<minExclusive
   fixed = boolean : false
   id = ID
   value = anySimpleType
   {any attributes with non-schema namespace ...}>
   Content: (annotation?)
   {value} `must` be in the `value space` of {base type definition}.
</minExclusive>
```

**Example**

The following is the definition of a `user-derived` datatype which limits values to integers greater than or equal to 100, using `minExclusive`.

```
<simpleType name='more-than-ninety-nine'>
   <restriction base='integer'>
      <minExclusive value='99'/>
   </restriction>
</simpleType>
```

Note that the `value space` of this datatype is identical to the previous one (named 'one-hundred-or-more').

**minExclusive UML Model example**

For an example, see Section “simpleType UML Model example” on page 5-107.

**maxExclusive**

Maps to UML Attribute with stereotype XSDmaxExclusive. Name and type of such attribute does not make sence.

- value – to Attribute initial value.

**XML Representation Summary: maxExclusive Element Information Item**

```
<maxExclusive
   fixed = boolean : false
   id = ID
   value = anySimpleType
   {any attributes with non-schema namespace ...}>
   Content: (annotation?)
   {value} `must` be in the `value space` of {base type definition}.
</maxExclusive>
```
Example

The following is the definition of a ·user-derived· datatype which limits values to integers less than or equal to 100, using ·maxExclusive·.

```xml
<simpleType name='less-than-one-hundred-and-one'>
  <restriction base='integer'>
    <maxExclusive value='101'/>
  </restriction>
</simpleType>
```

Note that the ·value space· of this datatype is identical to the previous one (named ‘one-hundred-or-less’).

maxExclusive UML Model example

For an example, see Section “simpleType UML Model example” on page 5-107.

minInclusive

Maps to UML Attribute with stereotype XSDminInclusive. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

**XML Representation Summary: minInclusive Element Information Item**

```xml
<minInclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</minInclusive>

{value} ·must· be in the ·value space· of {base type definition}.
```

Example

The following is the definition of a ·user-derived· datatype which limits values to integers greater than or equal to 100, using ·minInclusive·.

```xml
<simpleType name='one-hundred-or-more'>
  <restriction base='integer'>
    <minInclusive value='100'/>
  </restriction>
</simpleType>
```

minInclusive UML Model example
For an example, see Section “simpleType UML Model example” on page 5-107.

**maxInclusive**

Maps to UML Attribute with stereotype XSD:maxInclusive. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

**XML Representation Summary:** maxInclusive Element Information Item

```
<maxInclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace ...}>
  Content: (annotation?)
</maxInclusive>
{value} ·must· be in the ·value space· of {base type definition}.
```

**Example**

The following is the definition of a ·user-derived· datatype which limits values to integers less than or equal to 100, using ·maxInclusive·.

```
<simpleType name='one-hundred-or-less'>
  <restriction base='integer'>
    <maxInclusive value='100'/>
  </restriction>
</simpleType>
```

**maxInclusive UML Model example**

For an example Section “simpleType UML Model example”.

**totalDigits**

Maps to UML Attribute with stereotype XSD:totalDigits. Name and type of such attribute does not make sence.

- value – to Attribute initial value.
XML Representation Summary: totalDigits Element Information Item

```xml
<totalDigits
  fixed = boolean : false
  id = ID
  value = positiveInteger
  (any attributes with non-schema namespace . . .)> Content: (annotation?)
</totalDigits>
```

Example

The following is the definition of a ‘user-derived’ datatype which could be used to represent monetary amounts, such as in a financial management application which does not have figures of $1M or more and only allows whole cents. This definition would appear in a schema authored by an “end-user” and shows how to define a datatype by specifying facet values which constrain the range of the ·base type· in a manner specific to the ·base type· (different than specifying max/min values as before).

```xml
<simpleType name='amount'>
  <restriction base='decimal'>
    <totalDigits value='8'/>
    <fractionDigits value='2' fixed='true'/>
  </restriction>
</simpleType>
```

totalDigits UML Model example

For an example, see Section “simpleType UML Model example” on page 5-107.

fractionDigits

Maps to UML Attribute with stereotype XSDfractionDigits. Name and type of such attribute does not make sence.

- value – to Attribute initial value.

XML Representation Summary: fractionDigits Element Information Item

```xml
<fractionDigits
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  (any attributes with non-schema namespace . . .)> Content: (annotation?)
</fractionDigits>
```
Example
The following is the definition of a user-derived datatype which could be used to represent the magnitude of a person's body temperature on the Celsius scale. This definition would appear in a schema authored by an "end-user" and shows how to define a datatype by specifying facet values which constrain the range of the base type.

```
<simpleType name='celsiusBodyTemp'>
  <restriction base='decimal'>
    <totalDigits value='4'/>
    <fractionDigits value='1'/>
    <minInclusive value='36.4'/>
    <maxInclusive value='40.5'/>
  </restriction>
</simpleType>
```

fractionDigits UML Model example
For an example, see Section “simpleType UML Model example” on page 5-107.

length
Maps to UML Attribute with stereotype XSDlength. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: length Element Information Item

```
<length
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</length>
```

Example
The following is the definition of a user-derived datatype to represent product codes which must be exactly 8 characters in length. By fixing the value of the length facet we ensure that type derived from productCode can change or set the values of other facets, such as pattern, but cannot change the length.

```
<simpleType name='productCode'>
  <restriction base='string'>
    <length value='8' fixed='true'/>
  </restriction>
</simpleType>
```
length UML Model example
For an example, Section “simpleType UML Model example” on page 5-107.

minLength
Maps to UML Attribute with stereotype XSDminLength. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: minLength Element Information Item

<table>
<thead>
<tr>
<th>Element</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>minLength</td>
<td>fixed</td>
<td>boolean: false</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>nonNegativeInteger</td>
</tr>
<tr>
<td></td>
<td>Content:</td>
<td>(annotation)</td>
</tr>
</tbody>
</table>

Example
The following is the definition of a ·user-derived· datatype which requires strings to have at least one character (i.e., the empty string is not in the ·value space· of this datatype).

```
<simpleType name="non-empty-string">
  <restriction base='string'>
    <minLength value='1'/>
  </restriction>
</simpleType>
```

minLength UML Model example
For an example, see Section “simpleType UML Model example”

maxLength
Maps to UML Attribute with stereotype XSDmaxLength. Name and type of such attribute does not make sense.

- value – to Attribute initial value.
XML Representation Summary: **maxLength** Element Information Item

```xml
<maxLength
    fixed = boolean : false
    id = ID
    value = nonNegativeInteger
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
</maxLength>
```

Example

The following is the definition of a `user-derived` datatype which might be used to accept form input with an upper limit to the number of characters that are acceptable.

```xml
<simpleType name='form-input'>
    <restriction base='string'>
        <maxLength value='50'/>
    </restriction>
</simpleType>
```

**maxLength UML Model example**

For an example, see Section “simpleType UML Model example”

---

**whiteSpace**

Maps to UML Attribute with stereotype XSDwhiteSpace. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: **whiteSpace** Element Information Item

```xml
<whiteSpace
    fixed = boolean : false
    id = ID
    value = (collapse | preserve | replace)
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
</whiteSpace>
```

Example

The following example is the datatype definition for the `token` `built-in` `derived` datatype.

```xml
<simpleType name='token'>
    <restriction base='normalizedString'>
        <whiteSpace value='collapse'/>
    </restriction>
</simpleType>
```
whiteSpace UML Model example
For an example, see Section “simpleType UML Model example” on page 5-107.

pattern
Maps to UML Attribute with stereotype XSDpattern. Name and type of such attribute does not make sense.

- value – to Attribute initial value or TaggedValue with name ‘value’.

XML Representation Summary: pattern Element Information Item

```
<pattern id = ID value = anySimpleType
       {any attributes with non-schema namespace . . .}>Content: (annotation?)
</pattern>
{value} ‘must’ be a valid ‘regular expression’.
```

Example
The following is the definition of a ‘user-derived’ datatype which is a better representation of postal codes in the United States, by limiting strings to those which are matched by a specific ‘regular expression’.
```
<simpleType name='better-us-zipcode'>
  <restriction base='string'>
    <pattern value='[0-9]{5}(-[0-9]{4})?'/>
  </restriction>
</simpleType>
```

pattern UML Model example
For an example, Section “simpleType UML Model example” on page 5-107.

enumeration
Maps to UML Attribute with stereotype XSDenumeration.

- value – to Attribute name.
XML Representation Summary: `enumeration` Element Information Item

```xml
<enumeration
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</enumeration>
```

Example

The following example is a datatype definition for a `user-derived` datatype which limits the values of dates to the three US holidays enumerated. This datatype definition would appear in a schema authored by an "end-user" and shows how to define a datatype by enumerating the values in its `value space`. The enumerated values must be type-valid literals for the `base type`.

```xml
<simpleType name='holidays'>
  <annotation>
    <documentation>some US holidays</documentation>
  </annotation>
  <restriction base='gMonthDay'>
    <enumeration value='--01-01'>
      <annotation>
        <documentation>New Year's day</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--07-04'>
      <annotation>
        <documentation>4th of July</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--12-25'>
      <annotation>
        <documentation>Christmas</documentation>
      </annotation>
    </enumeration>
  </restriction>
</simpleType>
```

```
enumeration UML Model example

For an example, see Section “simpleType UML Model example” on page 5-107.
```
unique

Maps to UML Attribute added into some UML Class.

```xml
<unique
    id = ID
    name = NCName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (selector, field+))
</unique>
```

unique UML Model example

For an example, see Section “keyref UML Model example” on page 5-122.

key

Maps to UML Attribute added into some UML Class.

- name – to Attribute name.
- id – to TaggedValue

```xml
<key
    id = ID
    name = NCName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (selector, field+))
</key>
```

key UML Model example

For an example, see Section “keyref UML Model example” on page 5-122

keyref

Maps to UML Attribute added into some UML Class.

- refer – to value of “refer” or “referString” TaggedValue.
- name – to Attribute name.
- id – to TaggedValue

```xml
<keyref
    id = ID
    name = NCName
    refer = QName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (selector, field+))
</keyref>
```
keyref UML Model example

```xml
  <xs:element name = "vehicle" >
    <xs:complexType >
      <xs:all />
      <xs:attribute name = "plateNumber" type = "xs:integer" />
      <xs:attribute name = "state" type = "nm:twoLetterCode" />
    </xs:complexType>
  </xs:element>
  <xs:element name = "state" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "code" type = "nm:twoLetterCode" />
      </xs:sequence >
    </xs:complexType>
  </xs:element>
  <xs:element name = "car" type="carType"> 
    <xs:complexType >
      <xs:sequence >
        <xs:attribute name = "plateNumber" type = "integer" />
        <xs:attribute name = "regState" type = "twoLetterCode" />
      </xs:sequence >
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:element ref="nm:vehicle" maxOccurs="unbounded" />
<xs:element ref="nm:person" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
<xs:unique name="reg">
  <xs:annotation>
    <xs:documentation>unique documentation</xs:documentation>
  </xs:annotation>
  <xs:selector xpath="./vehicle" />
  <xs:field xpath="@plateNumber" />
</xs:unique>
</xs:element>
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="car">
        <xs:complexType>
          <xs:sequence />
          <xs:attribute name="regPlate" type="xs:integer" />
          <xs:attribute name="regState" type="nm:twoLetterCode" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="root">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="nm:state" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
  <xs:key name="state">
    <xs:selector xpath="./state" />
    <xs:field xpath="code" />
  </xs:key>
  <xs:keyref name="vehicleState" refer="nm:state">
    <xs:selector xpath="./vehicle" />
    <xs:field xpath="@state" />
  </xs:keyref>
  <xs:key name="regKey">
    <xs:annotation>
      <xs:documentation>key documentation</xs:documentation>
    </xs:annotation>
    <xs:selector xpath="./vehicle" />
    <xs:field xpath="@state" />
    <xs:field xpath="@plateNumber" />
  </xs:key>
  <xs:keyref name="carRef" refer="nm:regKey">
    <xs:annotation>
      <xs:documentation>key ref documentation</xs:documentation>
    </xs:annotation>
    <xs:selector xpath="./vehicle" />
    <xs:field xpath="@state" />
    <xs:field xpath="@plateNumber" />
  </xs:keyref>
</xs:element>
</xs:element>
selector and field

Maps to UML TaggedValues named “selector” and “field” of UML Attribute representing key, keyRef or unique. “selector” tag has value representing “xpath” and “field” - list of values representing field “xpath”. ID values shall be skipped and annotation documentation will be applied to tagged value according to annotation rule (see:annotation). For field values annotation documentation shall be merged in one.

Example

```
<xs:key name="fullName">
  <xs:selector xpath="%//person"/>
  <xs:field xpath="forename"/>
  <xs:field xpath="surname"/>
</xs:key>

<xs:keyref name="personRef" refer="fullName">
  <xs:selector xpath="%//personPointer"/>
  <xs:field xpath="@first"/>
  <xs:field xpath="@last"/>
</xs:keyref>

<xs:unique name="nearlyID">
  <xs:selector xpath="%//*"/>
  <xs:field xpath="@id"/>
</xs:unique>
```

XML representations for the three kinds of identity-constraint definitions.
Example

```xml
<xs:element name="state">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="code" type="twoLetterCode"/>
      <xs:element ref="vehicle" maxOccurs="unbounded"/>
      <xs:element ref="person" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:key name="reg"> <!-- vehicles are keyed by their plate within states -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@plateNumber"/>
  </xs:key>
</xs:element>

<xs:element name="root">
  <xs:complexType>
    <xs:sequence>
      . . .
      <xs:element ref="state" maxOccurs="unbounded"/>
      . . .
    </xs:sequence>
  </xs:complexType>

  <xs:key name="state"> <!-- states are keyed by their code -->
    <xs:selector xpath=".//state"/>
    <xs:field xpath="code"/>
  </xs:key>

  <xs:keyref name="vehicleState" refer="state"> <!-- every vehicle refers to its state -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@state"/>
  </xs:keyref>

  <xs:key name="regKey"> <!-- vehicles are keyed by a pair of state and plate -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@state"/>
    <xs:field xpath="@plateNumber"/>
  </xs:key>

  <xs:keyref name="carRef" refer="regKey"> <!-- people's cars are a reference -->
    <xs:selector xpath=".//car"/>
    <xs:field xpath="@regState"/>
    <xs:field xpath="@regPlate"/>
  </xs:keyref>
</xs:element>
```
A state element is defined, which contains a code child and some vehicle and person children. A vehicle in turn has a plateNumber attribute, which is an integer, and a state attribute. State's codes are a key for them within the document. Vehicle's plateNumbers are a key for them within states, and state and plateNumber is asserted to be a key for vehicle within the document as a whole. Furthermore, a person element has an empty car child, with regState and regPlate attributes, which are then asserted together to refer to vehicles via the carRef constraint. The requirement that a vehicle's state match its containing state's code is not expressed here.

selector and field UML Model example

For an example, see Section “keyref UML Model example” on page 5-122
“applInfo” shall have: “content” merged into one tag “applInfoSource” comment, but tag value shall represent first matched “applinfo”

XML Representation Summary: annotation Element Information Item

```
<annotation
  id = ID
  (any attributes with non-schema namespace . . .)>
  Content: (applinfo | documentation)*
</annotation>
<applinfo
  source = anyURI>
  Content: ((any))*
</applinfo>
<documentation
  source = anyURI
  xml:lang = language>
  Content: ((any))*
</documentation>
```

Example

```
<xs:simpleType fn:note="special">
  <xs:annotation>A type for experts only</xs:annotation>
  <xs:appinfo>checkForPrimes</xs:appinfo>
</xs:annotation>
```

XML representations of three kinds of annotation.

annotation UML Model example

```xml
<xs:schema xmlns:nom = "http://nomagic.com">
  <xs:annotation>
    the documentation for this schema
    {applInfoContent=infoContent,
     applInfoSource=infoSource,
     xml:lang=EN,
     source=documentation source}
  </xs:annotation>
</xs:schema>
```

```
  <xs:annotation>
    <xs:appinfo source = "infoSource">infoContent</xs:appinfo>
  </xs:annotation>
</xs:schema>
```
Compositors

Complex type maps to UML Class with stereotype XSDcomplexType. In order to have some group in complex type, the same UML Class also must have XSDall, XSDchoice or XSDsequence stereotype.

UML model can have ModelClass just with single stereotype XSDall, XSDchoice or XSDsequence. In this case such class maps to inner part of other group.

Elements order in sequence group is very important. Such elements are ordered according values of TaggedValue sequenceOrder.

Example

XML representations for the three kinds of model group, the third nested inside the second.
compositors UML Model example

```xml
<?xml version='1.0' encoding='Cp1252'?>
  <xs:group name = "myGroup" >
    <xs:annotation >
      <xs:documentation >my group documentation</xs:documentation>
    </xs:annotation >
    <xs:sequence minOccurs = "2" maxOccurs = "1" >
      <xs:choice >
        <xs:element name = "number" type = "xs:string" />
      </xs:choice>
      <xs:group ref = "nm:myGroup3" minOccurs = "0" maxOccurs = "1" >
        <xs:element name = "address" type = "xs:string" />
      </xs:group>
    </xs:sequence>
  </xs:group>  
</xs:schema>
```
Maps to UML Class with stereotype XSDgroup.

This class also may have stereotype XSDall, XSDsequence or XSDchoice.

If group has ref attribute, such group definition maps to UML Attribute or UML Association End. UML Attribute must have XSDgroupRef stereotype. This stereotype may be omitted for AssociationEnd.

**XML Representation Summary:** group Element Information Item

```xml
<group
    name = NCName
    Content: (annotation?, (all | choice | sequence))
</group>
<group
    ref = QName
    maxOccurs = (nonNegativeInteger | unbounded) : 1
    minOccurs = nonNegativeInteger : 1>
    Content: (annotation?)
</group>
```
Example

```
<xs:group name="myModelGroup">
  <xs:sequence>
    <xs:element ref="someThing"/>
    ...
  </xs:sequence>
</xs:group>

<xs:complexType name="trivial">
  <xs:group ref="myModelGroup"/>
  <xs:attribute .../>
</xs:complexType>

<xs:complexType name="moreSo">
  <xs:choice>
    <xs:element ref="anotherThing"/>
    <xs:group ref="myModelGroup"/>
  </xs:choice>
  <xs:attribute .../>
</xs:complexType>
```

group UML Model example
For an example, see Section “compositors UML Model example” on page 5-129.

any and anyAttribute
Maps to UML Attribute with stereotype XSDany or XSDanyAttribute.
maxOccurs - to multiplicity upper range. Value unbounded maps to asterisk in UML.
minOccurs – to multiplicity lower range.
annotation maps to Attribute documentation
Other properties to TaggedValues.

XML Representation Summary: any Element Information Item

```
<any
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded) : 1
  minOccurs = nonNegativeInteger : 1
  namespace = ((##any | ##other) | List of (anyURI | (#targetNamespace | ##local)) ) : ##any
  processContents = (lax | skip | strict) : strict
  (any attributes with non-schema namespace . . .)
  Content: (annotation?)
</any>

<anyAttribute
  id = ID
  namespace = ((##any | ##other) | List of (anyURI | (#targetNamespace | ##local)) ) : ##any
  processContents = (lax | skip | strict) : strict
  (any attributes with non-schema namespace . . .)
  Content: (annotation?)
</anyAttribute>
```
Example

XML representations of the four basic types of wildcard, plus one attribute wildcard.

**any and anyAttribute UML Model example**

```xml
<?xml version='1.0' encoding='Cp1252'?>

<xs:group name = "my_type" >
  <xs:choice >
    <xs:any id = "anyID" namespace = "http://bla" processContents = "strict" minOccurs = "0" maxOccurs = "1" >
      <xs:annotation >
        <xs:documentation >any documentation</xs:documentation>
      </xs:annotation>
    </xs:any>
  </xs:choice>
</xs:group>

<xs:anyAttribute id = "anyID" namespace = "http:\bla.bla.bla" processContents = "skip" >
  <xs:annotation >
```

```xml
<!DOCTYPE xs:schema [>
</xs:schema>
<xs:documentation>any attribute</xs:documentation>
</xs:annotation>
</xs:anyAttribute>
</xs:attributeGroup>
</xs:schema>

### schema

Maps to UML Class with stereotype XSDschema.

All schema global attributes and elements are mapped to UML Attributes of this class.

Name of this class should match file name or must be assigned to the component, which represents file.

“xmlns” xml tags maps to an permission link with stereotype `<<<xmlns>>` and name, representing given prefix. Permission client is schema class and supplier package with name equal to the “xmlns” value.

**XML Representation Summary:** **schema** Element Information Item

```xml
<schema
  attributeFormDefault = (qualified | unqualified) : unqualified
  blockDefault = (#all | List of (extension | restriction | substitution)) :
  elementFormDefault = (qualified | unqualified) : unqualified
  finalDefault = (#all | List of (extension | restriction)) : ''
  id = ID
  targetNamespace = anyURI
  version = token
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ((include | import | redefine | annotation)*, ((simpleType | complexType | group | attributeGroup) | element | attribute | notation), annotation*)*)
</schema>
```

**Example**

```xml
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.com/example">
  . . .
</xs:schema>
```

The XML representation of the skeleton of a schema.
schema UML Model example

```xml
<xs:schema xmlns:nm = "http://nomagic.com"
xmlns:xs = "http://www.w3.org/2001/XMLSchema"
xmlns = "http://nomagic.com"
attributeFormDefault = "qualified",
blockDefault = "extension",
elementFormDefault = "unqualified",
finalDefault = "extension",
targetNamespace = "http://nomagic.com"
version = "1.2",
xml:lang = "EN"
/>`
**XML Representation Summary: notation Element Information Item**

```
<notation
  id = ID
  name = NCName
  public = anyURI
  system = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</notation>
```

**Example**

```
<xs:notation name="jpeg" public="image/jpeg" system="viewer.exe">
```

The XML representation of a notation declaration.

**notation UML Model example**

```
<<XSschema>>
  schema
<<XSNotation>>-jpeg(system=viewer.exe, public=image/jpeg)
```

```
<xs:schema xmlns:nn = "http://nomagic.com"
  xmlns:xs = "http://www.w3.org/2001/XMLSchema"
  targetNamespace = "http://nomagic.com" >
  <xs:notation name = "jpeg" public = "image/jpeg" system = "viewer.exe"
  />
</xs:schema>
```

**redefine**

Maps to UML Class with stereotype XSDredefine. This class has inner UML Classes as redefined elements. Every redefined element must be derived from other UML class with stereotype XSDsimpleType, XSDcomplexType, XSDgroup, XSDattributeGroup. The name of this class shall match “schemaLocation” value.

If two “redefine” with the same schema location appears, they shall be merged to the one and the same class with a name “schemaLocation”.

Redefine Class must be inner class of XSDschema Class.

- annotation - to XSDredefine UML Class documentation
- schemaLocation – to XSDredefine UML Class name.
XML Representation Summary: redefine Element Information Item

```xml
<redefine
  id = ID
  schemaLocation = anyURI
  (any attributes with non-schema namespace . . .)>
  Content: (annotation | (simpleType | complexType | group | attributeGroup))*
</redefine>
```

Example

v1.xsd:
```xml
<xs:complexType name="personName">
  <xs:sequence>
    <xs:element name="title" minOccurs="0"/>
    <xs:element name="forename" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="addressee" type="personName"/>
```

v2.xsd:
```xml
<xs:redefine schemaLocation="v1.xsd">
  <xs:complexType name="personName">
    <xs:complexContent>
      <xs:extension base="personName">
        <xs:sequence>
          <xs:element name="generation" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:redefine>
<xs:element name="author" type="personName"/>
```

The schema corresponding to v2.xsd has everything specified by v1.xsd, with the personName type redefined, as well as everything it specifies itself. According to this schema, elements constrained by the personName type may end with a generation element. This includes not only the author element, but also the addressee element.
redefine UML Model example

```xml
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:nm="http://nomagic.com"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://nomagic.com">
    <xs:redefine schemaLocation="http://nomagic.com">
        <xs:simpleType name="string">
            <xs:annotation>
                <xs:documentation>my documentation</xs:documentation>
            </xs:annotation>
            <xs:restriction base="xs:string" />
        </xs:simpleType>
    </xs:redefine>
</xs:schema>
```

import

Maps to UML Permission with stereotype XSDimport. Permission client must be schema class stereotypes `<XSDschema>` Component, supplier namespace Package XSDnamespace.

- namespace maps to supplier name.
- annotation maps to UML Attribute documentation
- schemaLocation maps to TaggedValue.

**XML Representation Summary:** `import` Element Information Item

```xml
<import
  id = ID
  namespace = anyURI
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</import>
```

**Example**

The same namespace may be used both for real work, and in the course of defining schema components in terms of foreign components:

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:html="http://www.w3.org/1999/xhtml"
  targetNamespace="uri:mywork" xmlns:my="uri:mywork">
  <import namespace="http://www.w3.org/1999/xhtml"/>
  <annotation>
    <documentation>
      <html:p>
        [Some documentation for my schema]
      </html:p>
    </documentation>
  </annotation>
  ...
  <complexType name="myType">
    <sequence>
      <element ref="html:p" minOccurs="0"/>
    </sequence>
    ...
  </complexType>
  <element name="myElt" type="my:myType"/>
</schema>
```

The treatment of references as *QNames* implies that since (with the exception of the schema for schemas) the target namespace and the XML Schema namespace differ, without massive redeclaration of the default namespace *either* internal references to the names being defined in a schema document or the schema declaration and definition elements themselves must be explicitly qualified. This example takes the first option -- most other examples in this specification have taken the second.
import UML Model example

  <xs:import namespace="http://www.w3.org/1999/xhtml" schemaLocation="http://www.w3.org/1999/xhtml" />
</xs:schema>

include

Maps to UML Component with stereotype XSDinclude. Component must be added into xsd file component.

- annotation maps to UML Component documentation
- schemaLocation maps to UML Component name.

XML Representation Summary: include Element Information Item

```
<include
  id = ID
  schemaLocation = anyURI
  (any attributes with non-schema namespace . . .)>
  Content: (annotation?)
</include>
```
include UML Model example

```
>
  <xs:include schemaLocation = "http://nomagic.com/schema.xsd" />
</xs:schema>
```

### XML schema namespaces

Maps to UML Package with stereotype XSDnamespace. In order to define “xmlns” attribute in the schema file, Permission between XSDnamespace package and XSDschema class must be added into the model.

- The Permission name maps to namespace shortcut.

**Example**

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.com/example">
  ...
</xs:schema>
```

The XML representation of the skeleton of a schema.

In order to generate such namespaces:

- UML model must have Package with name “http://www.w3.org/2001/XMLSchema”
- UML model must have Package with name “http://www.example.com/example”
- Permission with name “xs” must be added into model between XMLSchema Class and Package “http://www.w3.org/2001/XMLSchema”.
- Permission without name must be added into model between XMLSchema Class and Package “http://www.w3.org/2001/XMLSchema”.
XML schema namespaces UML Model example

For an example, see Section “schema UML Model example” on page 5-134.

**XSD file creation with MagicDraw**

New XML Schema code engineering language is added into MagicDraw engine in order to generate/reverse XSD files.

Code Engineering Project of this language has such Language Properties:

- Default Target XSD File Name – the name of default xsd file.

This CE set has one RT Component by default. Selected by user classes will be added into this component. Component is mapped to xsd file. RT Component does not have inner RT Components.

TODO in Magicdraw

add encoding property to project
Schema Components Diagram (non-normative)