## CONTENTS

### GETTING STARTED  6
- Introduction  6
- Installing Cameo Data Modeler Plugin  6

### ENTITY-RELATIONSHIP (ER) MODELING AND DIAGRAMS  7
- Introduction  7
- Basic Concepts  7
- Business Entity-Relationship Diagrams  8
- Identifying Relationships and Dependent Entities  9
- Constraints between Relationships  9
- Generalization and Specialization  9
- Key Modeling  14
- Virtual Entities  14
- Importing CA ERwin® Data Modeler Projects  15
  - Importing Data Models  15
  - Imported Elements  16

### DATABASE SUPPORT  18
- Introduction  18
- SQL Diagrams  18
  - Crow’s Foot Notation in SQL Diagrams  19
- Database Modeling  19
  - Common SQL Element Properties  19
  - Top Level Elements  20
    - Database 23
    - Schema 23
    - Catalog 23
    - GLOBALS 23
  - Tables, Columns, and Views  24
    - Persistent Table 25
    - Temporary Table 25
    - View 26
    - Column 27
  - Modeling Types  28
    - Predefined Type Libraries 28
    - Type Usage 29
    - User Defined Types 30
  - Sequences and Autoincrement Columns  34
  - Constraints  37
    - Implicit Primary Key, Unique, Check Constraint, and Index Modeling 38
    - Explicit Primary Key, Unique, Check Constraint, and Index Modeling 40
    - Foreign Keys 41
    - Nullability Constraint 44
    - Assertion 44
    - Triggers 45
  - Routines  46
    - Procedure 47
    - Function 48
    - Method 48
    - Parameter 48
    - Cursor and Routine Result Table 49
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Control</td>
<td>50</td>
</tr>
<tr>
<td>User 50</td>
<td></td>
</tr>
<tr>
<td>Group 50</td>
<td></td>
</tr>
<tr>
<td>Role 51</td>
<td></td>
</tr>
<tr>
<td>Privilege 51</td>
<td></td>
</tr>
<tr>
<td>Role Authorization</td>
<td>51</td>
</tr>
<tr>
<td>Oracle Database Modeling Extensions</td>
<td>52</td>
</tr>
<tr>
<td>Database Code Engineering</td>
<td>52</td>
</tr>
<tr>
<td>Code Engineering Set</td>
<td>53</td>
</tr>
<tr>
<td>Properties of Code Engineering Set for DDL</td>
<td>54</td>
</tr>
<tr>
<td>Supported SQL Statements</td>
<td>57</td>
</tr>
<tr>
<td>DDL Dialects</td>
<td>58</td>
</tr>
<tr>
<td>Standard SQL2</td>
<td>58</td>
</tr>
<tr>
<td>Oracle 58</td>
<td>59</td>
</tr>
<tr>
<td>Cloudscape</td>
<td></td>
</tr>
<tr>
<td>TRANSFORMATIONS</td>
<td>60</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>UML to SQL Transformation</td>
<td>61</td>
</tr>
<tr>
<td>Transformation Procedure</td>
<td></td>
</tr>
<tr>
<td>Conversion of Classes</td>
<td></td>
</tr>
<tr>
<td>Primary Keys Autogeneration</td>
<td></td>
</tr>
<tr>
<td>Conversion of Associations</td>
<td></td>
</tr>
<tr>
<td>Conversion of Identifying Associations</td>
<td></td>
</tr>
<tr>
<td>Conversion of Multivalued Properties</td>
<td></td>
</tr>
<tr>
<td>Conversion of Generalizations</td>
<td></td>
</tr>
<tr>
<td>Package Hierarchy Reorganization</td>
<td></td>
</tr>
<tr>
<td>Sequence Autogeneration</td>
<td></td>
</tr>
<tr>
<td>Type Mapping</td>
<td>64</td>
</tr>
<tr>
<td>UML to SQL Type Map</td>
<td></td>
</tr>
<tr>
<td>Transformation Properties</td>
<td>66</td>
</tr>
<tr>
<td>ER to SQL (Generic / Oracle) Transformations</td>
<td>68</td>
</tr>
<tr>
<td>Identifying Relationships</td>
<td>68</td>
</tr>
<tr>
<td>Key Transformation</td>
<td>68</td>
</tr>
<tr>
<td>Virtual Entity Transformation</td>
<td>69</td>
</tr>
<tr>
<td>Tracing between Data Model Layers</td>
<td>69</td>
</tr>
<tr>
<td>SQL to UML Transformation</td>
<td>70</td>
</tr>
<tr>
<td>Type Mapping</td>
<td>70</td>
</tr>
<tr>
<td>Transformation Results</td>
<td>70</td>
</tr>
<tr>
<td>UML to XML Schema Transformation</td>
<td>73</td>
</tr>
<tr>
<td>Type Mapping</td>
<td>74</td>
</tr>
<tr>
<td>Transformation Results</td>
<td>75</td>
</tr>
<tr>
<td>XML Schema to UML Transformation</td>
<td>77</td>
</tr>
<tr>
<td>Type Mapping</td>
<td>77</td>
</tr>
<tr>
<td>Transformation Results</td>
<td>77</td>
</tr>
<tr>
<td>ENTITY-RELATIONSHIP AND SQL REPORT</td>
<td>79</td>
</tr>
<tr>
<td>XML SCHEMAS</td>
<td>82</td>
</tr>
<tr>
<td>Introduction</td>
<td>82</td>
</tr>
</tbody>
</table>
XML Schema Mapping to UML Elements  83
Defined stereotypes  83
attribute  85
element  87
complexType  90
attributeGroup  94
simpleType  96
restriction  96
list  96
union  97
minExclusive  102
maxExclusive  103
minInclusive  103
maxInclusive  104
totalDigits  104
fractionDigits  105
length  106
minLength  106
maxLength  107
whiteSpace  107
pattern  108
enumeration  108
unique  109
key  110
keyref  110
selector and field  113
XML representations for the three kinds of identity-constraint definitions 114
annotation  115
compositors  117
group  119
any and anyAttribute  120
schema  122
notation  123
redefine  124
import  126
include  128
XML schema namespaces  128
GETTING STARTED

Introduction

Cameo Data Modeler plugin provides data-related modeling for MagicDraw. It includes entity-relationship, database and XML schema modeling features.

This plugin enables you to draw entity-relationship diagrams (using the crow’s foot notation). This is a full-featured variant of ER diagram (including extended entity-relationship concepts - like generalization), providing a spectrum of capabilities for logical data modeling.

This plugin provides SQL database modeling / diagramming and DDL script generation / reverse features. It supports 11 flavors of databases (including Standard SQL, Oracle, DB2, Microsoft SQL Server, MySQL, PostgreSQL), has separate type libraries for them, carries additional modeling extensions for Oracle databases, Transformations from / to plain UML models and from ER models are provided.

This plugin provides XML schema modeling / diagramming and schema file (*.xsd) generation / reversing features. Transformations from / to plain UML models are provided.

NOTES

- Cameo Data Modeler plugin replaces previous (free) Data Modeling Notations plugin that supported the business entity-relationship diagram, a simplified version of entity-relationship diagram, usable for high level, abstract domain data modeling.
- This plugin repackages database and XML schema modeling functionality, which was previously available only in MagicDraw Architect and Enterprise editions.

Installing Cameo Data Modeler Plugin

To install Cameo Data Modeler plugin

1. From the Help menu, select Resource/Plugin Manager.
2. Select Cameo Data Modeler plugin to download and install it.

Note that when you install the plugin, you get an evaluation key automatically. This key is good for 7 days. Afterwards you need to purchase a license for a plugin to work on diagrams provided by the plugin (when initial license expires, diagrams are switched to the read-only mode).

For more information on how to work with the Resource/Plugin Manager dialog, see MagicDraw User's Manual.pdf.
ENTITY-RELATIONSHIP (ER) MODELING AND DIAGRAMS

Introduction

Cameo Data Modeler plugin brings in the following:

- Entity Relationship profile.
- Entity Relationship diagram.
- Template for new ER project creation.
- Sample, demonstrating ER modeling features.
- ER to SQL (Oracle and Generic) transformation and accompanying traceability features.
- Entity-Relationship and SQL report.

Entity-Relationship diagram, as the name suggests, allows specifying entities and relationships between them. It is useful for the abstract domain modeling - to provide structure for data in the domain. It is much more abstract and implementation-independent than the SQL diagram, which shows the concrete implementation of the data structure in the database.

Basic Concepts

An entity is any thing that is capable of an existence. An entity usually refers to some aspect of the real world, which can be distinguished from other aspects of the real world (a person, place, customer transaction, order...).

An entity is represented by a box shape on the diagram. An entity has two compartments where properties (columns) of the entity can be specified. The upper compartment holds primary key properties of the entity; lower - other properties.

A relationship between entities describes how entities are associated. Relationships are represented by lines, connecting entities. Relationship end adornments indicate multiplicities of these ends. Multiplicity is the number of entity instances that can be associated with a number of another entity instances. Relationship multiplicity is represented by three symbols (so called “crow’s foot notation” or “Information Engineering notation” - see Table 1 on page 7).

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>Zero</td>
<td></td>
</tr>
<tr>
<td>Vertical</td>
<td>One</td>
<td></td>
</tr>
<tr>
<td>Crow’s foot</td>
<td>Many</td>
<td></td>
</tr>
</tbody>
</table>

Multiplicity lower bounds and upper bounds are paired into one adornment - see the possible pairings in Table 2 on page 8. Note that any lower bound, which is more that 0 is treated as 1 (this also includes lower bounds greater than 1 - such as e.g. 2). Also, any upper bound which is greater than 1 is treated as Many (this also includes upper bounds less than unlimited - such as e.g. 7).
**TABLE 2. Multiplicity bound pairings**

<table>
<thead>
<tr>
<th>Min</th>
<th>Max</th>
<th>Read As</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>One (optional)</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>One (mandatory)</td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>0</td>
<td>Many</td>
<td>Many (optional)</td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>1</td>
<td>Many</td>
<td>Many (mandatory)</td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**NOTES**

- Some authors use **Entity Type** term to signify the description of a set of entities and **Entity Instance** term to signify concrete exemplar from that set. **Entity** term used in this manual corresponds to **Entity Type**.
- Data modeling world frequently uses term **Cardinality** to denote the allowable numbers of entity instances, which can be associated. But with the rise of the UML, the more correct term **Multiplicity** was introduced and term **Cardinality** is only used to denote concrete numbers of entity instances, which are associated. Hence in the example Person [0..1]------[-0..*] Book, the ranges [0..1] and [0..*] are called multiplicities. And if we have person “John Doe” associated with books “Moby Dick” and “Origin of Species”, we have a cardinality of 2 for loaned books role (and 1 on an opposite end - current reader role). Note that cardinality is always concrete number while multiplicity denotes range of possible cardinalities.

---

**Business Entity-Relationship Diagrams**

There is a flavor of the ER diagrams, called Business ER diagrams - this is a simplified flavor of the ER diagram. This diagram only shows entities as boxes (without structure) and relationships between them. It is useful for high-level, abstract domain modeling - provide a structure for business data, or define business terminology.

These diagrams can be draw using the same ER diagram simply by suppressing both primary key and column compartments on all the entities. Convenient way to do this is to multiselect all the entities (hold down ALT and click any entity) and use the **Suppress All** button in the Shape Editing toolbar of the diagram.
Identifying Relationships and Dependent Entities

One-to-many (and, very rarely, one-to-one) relationship can be declared identifying. Identifying relationship is a “stronger” version of the relationship, indicating that the one entity (the one at the multiple end of the relationship) cannot exist without the entity on the other end.

You can create such relationships using buttons on a diagram pallet. You can also turn an existing relationship into identifying and back again. For this you can choose to do one of the following: either change the Is Identifying property value in the relationship Specification window or select the appropriate check box on its shortcut menu.

Identifying relationship is drawn as solid line. Non-identifying relationships use heavy dashes.

Closely related concept is dependent / independent entities. Dependent entities are those, which are at the multiple end of the identifying relationship. They cannot exist without the independent entity at the other end. In addition every inherited entity (if you are doing EER modeling) is considered to be dependent.

Dependent entity’s primary key includes the other entity’s key as part (this is implied, not shown in the model).

Dependent entities are automatically recognized and drawn with rounded corners.

Constraints between Relationships

You can place XOR constraints (there is also a rarely used OR constraint) between relationships using a corresponding toolbar button. Note that constraint must join relationships, that have at least one common end - not any arbitrary relationships.

Current implementation of constraints does not allow placing a constraint on more than 2 relationships.

Generalization and Specialization

ER diagram has a support for generalization / specialization modeling. Generalization and Specialization is really the same relationship, just the different direction of classification (generalization is bottom-up, specialization is top-down). Hence they use the same model element.
Generalizations can be joined into generalization sets (trees of generalizations), which allow specifying additional properties on a group of generalizations - such as disjointness and completeness constraints.

Disjointness and completeness constraints are specified using the **Is Disjoint** (true for disjoint, false for overlapping specialization) and **Is Covering** (true for total, false for partial specialization) properties. They can be set via the relationship shortcut menu or in the Specification window.
Hence there are 4 combinations of these two settings. The “breadloaf” symbol joining generalizations into a tree shows these 4 variations - see the following figures.

Figure 4 -- Example of overlapping and partial specialization in ER diagram

Figure 5 -- Example of overlapping and total specialization in ER diagram
In the specialization hierarchies, there can be several ways how entity instance is assigned to specific entity subtype. It can be determined by the user - when user himself decides to which subtype given instance belongs (user-defined specialization). Or it can be determined by actual data values of entity instance (attribute-defined specialization). The latter case can be further subdivided into two subcases - simple attribute-based discrimination (when discrimination is performed by doing simple attribute value comparison) and more complex predicate-based discrimination (when discrimination is specified using more complex, explicitly specified conditions).
Examples of these two cases are shown in the following figures.

**Figure 8 -- Example of attribute-based discriminator in ER diagram**

Discriminators are modeled as special constraints, placed on individual generalization relationships. The easiest way to access them is from the shortcut menu of the generalization.

Predicate-based discriminator is simpler - you just fill in the **Specification** field of the predicate with an appropriate expression text.

Attribute-based discriminator is more complex. First you have to specify columns, by which you will discriminate the entities into the corresponding subclasses. This is done by filling in the **Discriminator** field of the generalization set (you can specify one or several columns there). Then you have to fill in the **Template** field of the predicate. This template field holds an instance specification, which is used as template or etalon to differentiate the entity instances into appropriate subclasses. Fill in the slots for the same columns that you indicated on the generalization set.

**NOTE** Category (also know as union) concept is currently not explicitly supported. Total (but not partial) categories can be “simulated” using the total specialization tree, just visually reversed.
Key Modeling

Keys of the entity are marked by applying the corresponding stereotype («PrimaryKey», «AlternativeKey») on the necessary column(s). This can be done from the shortcut menu of the column.

![Example of key usage in ER diagram](Figure 10)

Primary key columns are grouped into a separate compartment. When the «PrimaryKey» stereotype is applied / unapplied, the column migrates between the two compartments.

In rare cases there is a need to specify several alternative keys on the same entry. This can be done, by filling the “Id” tag field of the key column with key identifier(s). Columns, having the same Id are considered to be belonging to the same key. Overlapping alternative keys can be specified in the same manner (column can have several Ids specified).

![Example of multiple overlapping alternative keys in ER diagram](Figure 11)

Inversion entries are specified analogously. Inversion entry is a non-unique (combination of) column(s), which nevertheless is used frequently to search for the entity. Marking columns as IE gives hints to database implementers about which indexes to specify.

![Example of inversion entry in ER diagram](Figure 12)

**NOTE** Though ER profile carries the «ForeignKey» stereotype, this stereotype is currently unused. It is reserved for future - for automatic foreign key derivation functionality. Users should not specify FK columns explicitly on their entities (FKs are implied), unless needed for some specific purpose - use at your own risk.

Virtual Entities

Virtual entities are entities that can be derived from information in other entities. They are marked with keyword «virtual» on the diagrams. Otherwise they can be handled in the same manner as other entities.
Virtual entities roughly correspond to views in databases.

If you need to specify exact way how virtual entities are derived from other entities, you can use Abstraction relationships from UML; derivation expression can be specified in the Mapping field.

Cameo Data Modeler Plugin for MagicDraw provides import functionality for data models created using CA ERwin® Data Modeler (henceforth will be referred as ERwin). ERwin is one of the leaders in the data modeling tools market.

Data models produced in ERwin have a two-layer structure consisting of logical and physical layers that are tightly synchronized. The physical layer semantically corresponds to the SQL modeling / diagramming / generation functionality in MagicDraw. The logical layer corresponds to ER diagrams, implemented by Cameo Data Modeler Plugin.

The import functionality only imports logical layer data from ERwin into ER diagrams / data model in MagicDraw. Cameo Data Modeler Plugin does not yet support import of physical layer data.

### Importing Data Models

Cameo Data Modeler supports model files produced in ERwin version 7.x. It is recommended that the newest v7.3 should be used since it has been heavily tested. Data models in ERwin must be saved in the *.xml format (choose the XML Standard File option in the Save As dialog).
To import an ERwin model:

2. Click File > Import From > CA ERwin Data Modeler v7.x. The Open file dialog will open.
3. Select an ERwin model file (*.xml). A new MagicDraw project will be created and logical model will be imported from the ERwin model file into that project.

After successful import, you can proceed to edit or manage the model using MagicDraw features.

If you want to include the ER model as part of a larger project in MagicDraw, you can use either module linking functionality (click File > Use Module) to attach the ER model to your main project model or project import functionality (click File > Import From > Another MagicDraw Project) to transfer the contents of this ER model to your main project model.

If you want to update an imported and edited ER model, for example, you have made changes to the ERwin model and want to import those changes into MagicDraw again, you can use the merge functionality (click Tools > Project Merge) to import the ERwin model into a new ER model and merge it with the model you have imported earlier.

## Imported Elements

<table>
<thead>
<tr>
<th>ERwin</th>
<th>Cameo Data Modeler</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any element</td>
<td>Any Element</td>
<td>• For each element, its name, definition, and notes are imported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Definitions are imported as MagicDraw documentation (special UML comments) and notes are imported as UML comments.</td>
</tr>
<tr>
<td>Entity</td>
<td>Entity</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Attribute</td>
<td>• The Null / Not Null setting is imported as UML multiplicities [0..1] / [1].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Attribute constraints and default value information is imported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Domain information is not imported because domains are not supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Attribute type information is imported - the standard primitive types are mapped to the UML primitive types.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Other types (which are not found in the model) are created on the fly.</td>
</tr>
<tr>
<td>Key</td>
<td>Key Marking on Attributes</td>
<td>• There is no separate standalone model element for a key in the Cameo Data Modeler ER diagrams. Instead, attributes belonging to a key are marked by applying a stereotype to them (PK, AK, or IE) as necessary.</td>
</tr>
<tr>
<td>Relationship</td>
<td>Association relationship</td>
<td>• Simple relationships are mapped to UML associations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Verb phrases are mapped to role names.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cardinality and null / not null settings are mapped to UML multiplicities ([0..1], [1], [0..<em>], [1..</em>]).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Referential integrity information is stored in a special stereotype / tag.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Key information is not imported since the current ER diagrams do not support FK modeling.</td>
</tr>
</tbody>
</table>
### Importing CA ERwin® Data Modeler Projects

<table>
<thead>
<tr>
<th>ERwin</th>
<th>Cameo Data Modeler</th>
<th>Comments</th>
</tr>
</thead>
</table>
| Generalization       | Generalization relationship        | • ERwin relationships, which are participating in the generalization tree, are mapped to UML generalizations.  
• Generalizations are joined into generalization trees.  
• Complete / incomplete and overlapping / exclusive settings are imported / supported.  
• Discriminating columns are imported / supported.  
• Referential integrity information is stored in a special stereotype / tag.  
• Verb phrase information is not imported. |
| Default Value        | Instance Specification             | • A standalone UML instance specification is created to hold value definition. This instance specification is (or can be) then referenced from attributes, default value fields. |
| Domain               | -                                  | • Domains are not yet supported in Cameo Data Modeler. |
| Validation Rule      | Constraint                         | • The Validation rule is stored as constraint body text. |
| Display              | ER diagram                         | • Due to geometric constraints and element size changes, the diagram layout will be slightly different.  
• Paths between elements can be re-routed. |
| User Defined         | Profile / Stereotypes / Tags        | • A custom UML profile is created for the user's property definitions. |
| Properties Dictionary|                                    |          |
| User Defined         | Tag Values                         | • A custom profile generated from the UDP dictionary is applied and user property information is stored in the tag values of the applied custom stereotypes. |
| Properties Dictionary|                                   |          |
DATABASE SUPPORT

Introduction

Cameo Data Modeler plugin brings the following:

- IMM Relational profile for SQL modeling support (the profile is named according to the OMG working group name).
- Extension profile for Oracle.
- SQL diagram, Oracle SQL diagram and customizations for profile.
- Code engineering (generation / reverse) features for working with database generation scripts.
- Primitive type libraries for database flavors.
- Template for new Database project creation.
- Sample, demonstrating database modeling features.
- UML / ER to SQL (Oracle and generic) and SQL to UML transformations and accompanying traceability features.
- Entity-Relationship and SQL report.
- Helper functionality for SQL diagrams - notation switch.

Cameo Data Modeler plugin provides support for database modeling and code engineering. It supports modeling of the database concepts at the level of SQL:1999 (SQL3) standard. A few rarely used concepts (like collation, translation) are not supported.

SQL Diagrams

Cameo Data Modeler provides a specialized diagram for database modeling. This diagram is called SQL Diagram and is located under Data Modeling diagram subgroup. This diagram provides means for creating and depicting various SQL elements.

In addition to the main SQL diagram, there is a slightly modified diagram for Oracle databases. It is called Oracle SQL Diagram and is located under the same Data Modeling diagram subgroup. This diagram is only slightly modified - it has an additional diagram button for the Materialized View modeling. Otherwise than that, it is identical to the main SQL diagram. If you are not modeling materialized views, you can freely use the generic diagram type instead of specialized one for Oracle modeling.
Crow’s Foot Notation in SQL Diagrams

Once Cameo Data Modeler plugin is applied to MagicDraw you can display the crow’s foot notation or use standard UML notation of associations (displaying multiplicities in text format) in the SQL diagram.

To display Multiplicities or crow’s foot notation in a SQL diagram:

1. Create the SQL diagram.
2. Draw two tables.
3. Create columns for the tables and some of them as primary keys.
4. Connect the table elements with the Foreign Key relationship.
5. Define Name, PK, and FK in the open Foreign Key dialog box.
6. Open the Project Options dialog box.
7. Select the General project options branch.
8. Change the Show relationship ends as property correspondingly to either No special notation or Crow’s feet. Multiplicities (Figure 14 on page 19) or crow’s foot notation (Figure 15 on page 19) will then be displayed on the Foreign Key ends.

Figure 14 -- Multiplicities on Foreign Key relationship in SQL diagram

Figure 15 -- Crow’s foot notation for Foreign Key relationship in SQL diagram

Database Modeling

This chapter covers modeling of various SQL elements - in detail and with examples.

Common SQL Element Properties

These properties are common and available for all SQL model elements in their Specification windows.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of this SQL model element.</td>
</tr>
<tr>
<td>Label</td>
<td>Label of SQL model element. Can be used for various referring purposes (both human and code referral).</td>
</tr>
<tr>
<td>Description</td>
<td>Longer text, describing this SQL element in more detail.</td>
</tr>
</tbody>
</table>
In addition to these SQL properties, some common, useful UML model properties are shown in the Specification windows (only in the Expert mode).

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TODO</td>
<td>Additional remarks about the further modifications, necessary for this element</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualified Name</td>
<td>Fully qualified name of this model element - names of all owning parent elements and this element, concatenated using “::” separators.</td>
</tr>
<tr>
<td>Owner</td>
<td>Model element, directly owning this element.</td>
</tr>
<tr>
<td>Applied Stereotype</td>
<td>Stereotypes, applied on this model element, extending element data over and above the standard UML functionality. SQL extension stereotypes can be seen here (implementing SQL model features, described in this document) as well as any additional extensions.</td>
</tr>
<tr>
<td>Image</td>
<td>Custom image can be set on each model element if necessary.</td>
</tr>
</tbody>
</table>

**Top Level Elements**

There are several top-level model elements, that serve as the containers for other model elements of the database model. Those are: Database, Schema, Catalog.

Top level elements are not strictly necessary to begin database modeling. You can start modeling database elements (like tables) in the standard UML package (even directly under root ‘Data’ model element). But top level elements help to provide context for those other elements and their naming and positioning in the database. So, at least one top level element should be present - either Schema element or Database element. Optimally both Database and Schema element should be present in the model (Schema package inside the Database package). Catalog modeling is less important, it can be skipped. Not all databases have support for catalogs.
When top-level element is created (either on the diagram or in the containment tree), a special dialog is shown for selecting database flavor.

![Database flavor selection dialog](image)

When DB flavor is chosen, the necessary profile for that DB flavor is attached to the project (providing standard data types for that DBMS and / or additional stereotypes for modeling extensions of that DB flavor). Then profile application relationship is created from the package that is being created (Database, Schema) to the attached DB profile. This marks the top level element as belonging to this DB flavor. Other DB elements, created under that top level element will be automatically considered as belonging to this DB flavor.

If you would like to switch database flavor after creating a top level element, you can do this in the following way.

To switch database flavor after creating a top level element

**IMPORTANT!** You must have the necessary module attached to your project (use File>Use Module menu, choose the necessary module from your `<install.root>/profiles` predefined location)

1. Right-click the top level element.
2. From the shortcut menu, select **Apply Profiles**.
3. Select the check box near the needed profile and clear the check box near the old profile.
4. Click **Apply**.
Top level elements can be explicitly drawn, on the diagram.

![Database Top Level Containers](image)

**Figure 17 -- Database top level containers (Database and Schema) on diagram pane**

However, showing top level elements on the diagram, and nesting their contents inside them is often clumsy, and consumes valuable diagram space. Showing them on the diagram pane is not necessary; it is enough to create them in the Containment tree (using the **New Element** command on the shortcut menu). Then, place your diagram inside the created containers, and the elements that you will be creating in your diagram, will go into the necessary container. See the following figure (logically equivalent to the previous one), showing a top level element just in the Containment tree and not displayed on the diagram pane.

![Database Top Level Containers in Containment Tree](image)

**Figure 18 -- Database top level containers (Database and Schema) in Containment tree, but not on diagram pane**

There is also one additional complication, stemming from the limitations of UML. UML does not allow placing UML properties (which are used for SQL sequence modeling), or operations (which are used for SQL stored procedure & function modeling) directly into packages. Properties and operations can only occur in classes. A special model element was introduced to work around this limitation - **GLOBALS** element (based on UML class). This intermediate element can be placed directly inside the top level element (usually Schema, but can
also be placed under Database) and then the necessary database elements - sequences, stored procedures can be placed inside it.

**Database**

**NOTE** Database is modeled as UML Package with Database stereotype applied.

Database is a top level element, representing entire database within DBMS.

Besides the standard SQL element properties, database has the following properties available in the Specification window:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor</td>
<td>Specifies the vendor and the version of the database software. These fields are used for information purposes only. They do not affect the generation or further modeling.</td>
</tr>
<tr>
<td>Version</td>
<td></td>
</tr>
</tbody>
</table>

**Schema**

**NOTE** SQL Schema is modeled as UML Package with Schema stereotype applied.

Schema element represents a collection of database elements - tables, indexes, stored procedures, etc. - grouped for particular purpose (such as data structures for some particular application).

**Catalog**

**NOTE** SQL Catalog is modeled as UML Package with Catalog stereotype applied.

Catalog element represents intermediate grouping level between database and schema. Catalogs are also reused for Oracle DB modeling - to implement Oracle packages.

**GLOBALS**

**NOTE** GLOBALS is modeled as UML Class with the «Globals» stereotype applied.

GLOBALS element is a special intermediate element to work around limitation of UML. UML does not allow placing UML properties (which are used for SQL sequence modeling), or UML operations (which are used for SQL stored procedure & function modeling) directly into packages. Properties and operations can only occur in classes.

To work around this limitation, GLOBALS element (based on UML class) was introduced. This intermediate element can be placed directly inside the top level element (usually Schema, but can also be placed under Database) and then the necessary database elements - sequences, stored procedures and functions can be placed inside it.

Name of GLOBALS model element is not important, but for the sake of tidiness it should be named "GLOBALS". There should be at most one such element per the container (Schema, Database, Package). This model element does not carry any additional useful properties; it serves just as a carrier of inner elements - sequences and routines.
Tables, Columns, and Views

Tables and their constituent columns are the main elements for describing database data structures. Tables store multiple rows of data each consisting of several columns. Each cell holds one data value (or is empty). All values of one column are of the same type. Correspondingly each table description consists of the table name and a set of column descriptions. Additionally, there are various kinds of constraints (including the all-important primary key and foreign key constraints), that can be applied on tables and triggers, specifying additional actions to be performed during data manipulation.

See "Constraints" on page 37 for constraints and "Triggers" on page 45 for triggers.

There can be various kinds of tables

- Normal persistent tables
- Temporary tables
- Views (derived tables)

The following figure illustrates various kinds of tables that can be modeled on the diagram.

![Figure 19 -- Various kinds of tables: persistent tables, temporary tables, and views](image)

Tables can have generalization relationships between them. These relationships correspond to the following SQL syntax in the create table statement:

```sql
CREATE TABLE <name> OF <UDT name> [UNDER <supertable>]
```

There can be at most 1 outgoing generalization. Generalizations are not widely supported in database management systems. As of v17.0.1 Cameo Data Modeler supports modeling of these structures. Generation of corresponding script code is not supported yet.
Persistent Table

NOTE SQL Persistent Table is modeled as UML Class with the «Persistent-Table» stereotype applied. For the sake of compactness, these tables are displayed with the «table» keyword (instead of the long form - «PersistentTable») on the diagram.

Persistent table is the most often used kind of table.

Besides the standard SQL element properties, persistent table has the following properties available in the Specification window (these properties are only available in Expert mode).

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined type</td>
<td>Points to structured user defined type, which serves as a base for the row type of the table.</td>
</tr>
<tr>
<td>Supertable</td>
<td>Points to the parent (base) table. Can only be used together with user-defined type.</td>
</tr>
<tr>
<td>Self Ref Column Generation</td>
<td>Describes the self-referencing column generation options. Can only be used together with user-defined type. Corresponds to the following subclause of SQL create table statement: REF IS &lt;column name&gt; [SYSTEM GENERATED</td>
</tr>
<tr>
<td>Referencing Foreign Keys</td>
<td>This is back reference from foreign keys, referencing this table. This field is for information purposes only. If you want to change it, change Referenced Table field of the foreign key instead.</td>
</tr>
<tr>
<td>Insertable</td>
<td>These are two derived (non editable) fields, describing table data editing capabilities. At the moment calculation of these properties is not implemented - they are always set to false</td>
</tr>
<tr>
<td>Updatable</td>
<td></td>
</tr>
</tbody>
</table>

Temporary Table

NOTE SQL Temporary Table is modeled as UML Class with the «TemporaryTable» stereotype applied. For the sake of compactness, these tables are displayed with the «temporary» keyword (instead of the long form - «TemporaryTable») on the diagram.

Temporary table is a kind of table, where data is held only temporary. There are two kinds of temporary tables. Local temporary table persists for the duration of user session and is visible only for the creator user. Global temporary table is long lived and visible for all users. Note that data in the global temporary table is different for different users and does not persist throughout user sessions (only global table definition persists).

Temporary tables are created using SQL create table statement (using TEMPORARY option):

```
CREATE (GLOBAL | LOCAL) TEMPORARY TABLE <table name> ... [ON COMMIT (PRESERVE | DELETE) ROWS]
```

Besides the standard SQL element properties and persistent table properties (see section above), temporary table has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Marks the table as local or global temporary table.</td>
</tr>
<tr>
<td>Delete On Commit</td>
<td>Regulates whether data is deleted or retained on commit.</td>
</tr>
</tbody>
</table>
**View**

**NOTE** SQL View is modeled as UML Class with the «ViewTable» stereotype applied. For the sake of compactness, views are displayed with the «view» keyword (instead of the long form - «ViewTable») on the diagram.

View is a table, whose data is derived from data of other tables (by applying some SQL query).

Views are created using SQL create view statement:

```
CREATE VIEW <name> [ [<view column list>] ]
AS <query expression>
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Note that since column definition list is optional in SQL syntax, specifying column definitions in the view is also optional (columns can be inferred from query expression of the view). However it is often a good idea to include column definitions, since this allows to see view data structure on the diagram / in the model at a glance, without parsing the query expression text.

Besides the standard SQL element properties and persistent table properties (see section above), view has the following properties available in the Specification window:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Expression</td>
<td>A query expression, defining how data is calculated / retrieved for this view. This is an SQL SELECT statement.</td>
</tr>
<tr>
<td>Check Type</td>
<td>Describes how check is performed on the data update through the view. Only meaningful for updateable views (which is rare).</td>
</tr>
</tbody>
</table>

Query expression of the view modeling deserves a special attention. Query expression, defining the view, is not just a simple string, but a (stereotyped) UML model element. By default query expression model object is stored within the view definition itself. There is a special constraint, automatically created inside the view, to hold this expression. When the view is created, **Query Expression** field (which is a tag of stereotype, applied on the view) is automatically pointed to this expression.

So by default you just need to fill in the **Body** text of the expression. To do that you need to double-click on the **Query Expression** field. This opens Specification window for the expression itself, where **Body** can be filled in. This is the default, no-hassle way to specify view. It is easy. But it has one deficiency. Views created this way do not have any model references to the underlying table model elements. This may be undesirable from the dependency tracking standpoint (in the dependency analysis). To remedy this, you can draw an additional **Dependency** relationships between the view and base tables.

There is also another way to model the query expression, defining the view. If you click on the ... button of the **Query Expression** field, this action opens the element selection dialog, allowing to retarget the **Query Expression** pointer choose another expression object, located somewhere else in the model. For example view definition expression can be located inside the **Abstraction** relationship, drawn from the view to the base table (Mapping field of the **Abstraction**).

To model view queries using abstractions:

1. Draw an abstraction relationship between a View and a Table.
2. In the abstraction’s Specification window, fill in the **Mapping** cell. This will be an inner UML OpaqueExpression model element with language and body cells. Set language to “SQL” and fill in the body with the necessary “SELECT ...” expression text.
3. Further open the Specification window of the mapping expression, and apply the «QueryExpressionDefault» stereotype.
4. Open the Specification window of the view. Click the ... button in the **Query Expression** cell. In the element Selection dialog navigate to the abstraction relationship and select the expression inside of it.

This way to model view query expressions is rather tedious - so it is not recommended for modeling novices. But it has an advantage of capturing the real relationship in the model between the view and the constituent table(s). Also query expression can be shown on the abstraction relationship (using note mechanism) instead of showing expression on the view.

In the following figure you can see a diagram that illustrates the alternative way of view modeling.

![Diagram](image)

**Figure 20 -- Alternative notation for modeling view derivation from tables**

**Column**

**NOTE**  SQL Column is modeled as UML Property with «Column» stereotype applied. For the sake of compactness, columns are displayed with the «col» keyword (instead of the long form - «Column») on the diagram.

Column model element describes one column of the table. In the most frequent case it's just a name of the column and a type. Additionally column can carry default value specification, column constraints.

Column definition syntax in SQL (in CREATE TABLE, ADD COLUMN statements):

```sql
<column name> [ <data type> ]
[ DEFAULT <value expression> ]
[ GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
  [ '(' <sequence options> ')' ]
  [ GENERATED ALWAYS AS <expression>] ]
[ <column constraint definition>... ]
```
Besides the standard SQL element properties, column has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Collectively these two fields describe the type of the column. Type could be any of the primitive types from the library or user defined type.</td>
</tr>
<tr>
<td>Type Modifier</td>
<td>Modifier provides additional parameters for the type - such as width of the character type (when type=varchar and modifier=&quot;(20)&quot; - column is of varchar(20) type). See Type Usage section for details.</td>
</tr>
<tr>
<td>Nullable</td>
<td>Marks column as nullable or not. Basically this is an in-line nullability constraint. See Constraints section for details.</td>
</tr>
<tr>
<td>Default Value</td>
<td>Carries the default value of the column. This is normally an opaque expression, allowing to specify the value of the column. However it can be switched to Identity Specifier. In this case it describes the autoincrement options of the column. See Sequences section.</td>
</tr>
<tr>
<td>Is Derived</td>
<td>Standard UML field, used to mark the column as derived (GENERATED ALWAYS AS &lt;expression&gt;). It works together with Default Value field.</td>
</tr>
<tr>
<td>Scope Check</td>
<td>Marks this column as scope checked to a particular table and allows choosing particular referential integrity ensuring action (RESTRICT CASCADE, etc).</td>
</tr>
<tr>
<td>Implementation Dependent</td>
<td>Marks this column as implementation dependent.</td>
</tr>
</tbody>
</table>

**Modeling Types**

Cameo Data Modeler provides the standard type libraries as well as ability to model user defined types (structured user defined types and composites - multiset, array data types). The types can then be used to specify columns of the tables and / or parameters of procedures and functions. There is also a special mechanism for using types with modifiers. This mechanism is common in the MagicDraw, however some explanation is necessary on how to use it in database modeling.

**Predefined Type Libraries**

Cameo Data Modeler provides predefined type libraries for database flavors it supports. Besides the standard SQL type library, there are type libraries for Oracle, DB2, MS SQL, MySQL, PostgreSQL, Sybase, Cloudscape (Derby), Pervasive, MS Access and Pointbase. The standard SQL type library is the main type library, and type libraries for each flavor import (a subset of) types from it and define additional types, specific for that flavor.

The necessary type library is imported when you create the Database or Schema element in your model and choose a flavor for it (See Database flavor selection dialog: “Database flavor selection dialog” on page 21).
Type Usage

Usage of a simple SQL type, such as `boolean`, is very simple. If you want to set it as a type of a column or operation parameter, you just need to specify it in the `type` field. However there are types (such as `varchar` or `numeric`) in SQL, which require additional data. There are two mechanisms to specify these kinds of types: either use the library type+ type modifier mechanism or create your own type element.

Let's take the standard `varchar` type as an example. It must have the maximum length data provided at each usage. Semantically there are many different types, one for each length limit - `varchar(20)`, `varchar(53)`, `varchar(255)` etc. Now the standard type library can not provide myriad of different `varchar` types. Library only provides the `varchar` type definition.

To specify that column is of `varchar(20)`

1. Set the `type` field of the column to `varchar` type from the library.
2. Set the `type modifier` field of the column to "(20)" (no quotes). Note that type modifier is a simple string - whatever is entered in this field, will be used in script generation verbatim, without additional checks. An example of more complex type modifier would be "(10, 2)" type modifier for `numeric` data type.

Alternative way to specify that column is of `varchar(20)` is to explicitly create a separate type in the model.
DATABASE SUPPORT

To specify that column is of varchar(20) in the alternative way

1. Create the necessary type (use one of the buttons in the SQL diagram, Primitive Types toolbar) - character string, fixed precision, integer, approximate, boolean, binary, date or XML types. In our case this would be character string type.

2. Set the length data in the dedicated tag (look up the **length** tag in the **Tags** section of the Specification window). Note that this is numeric field - you need to input number 20, and not the “(20)” string as was the case with type modifiers.

3. The name of your type can be whatever you like. For example **varchar_of_20**. The name is not important.

4. Inherit (draw generalization relationship) your type from the appropriate type from the type library. In this case, inherit **varchar_of_20** from **varchar** form the library. This information will be used for determining the proper type name during script generation (so, in the generated script you will see the proper type reference - **varchar(20)**).

5. This created type can now be specified in the **type** field of the column(s).

There would be one type in the model for each **varchar** length that you use in your database.

The second way is more tedious - you need to create quite a few types. So by default the first way is used. But the second way has several advantages, that may outweigh it’s deficiencies. First - there is one spot where parameters of the type can be changed. You can easily widen the varchar(20) fields to varchar(40) by editing just one place in the model. Secondly, you can define some additional parameters of the type - such as character set.

**User Defined Types**

---

**Figure 22 -- Examples of user defined types**
Besides the primitive / built-in types of the database, user can define additional types for his own schema.

**Distinct Type**

**NOTE** SQL Distinct type is modeled as UML DataType with «DistinctUserDefinedType» stereotype applied. For the sake of compactness, references are displayed with the «distinct» keyword (instead of the long form - «DistinctUserDefinedType») on the diagram.

Distinct type definition allows to redefine some primitive type in order to enforce the non-assignability rules. For example, two distinct types Meters and Yards can be defined on the base primitive type float. With this definition, system would enforce checks that yard fields / columns are not assigned to meter fields / columns without a conversion (explicit cast).

Besides the standard SQL element properties, distinct type has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predefined Representation</td>
<td>Points to some base primitive type.</td>
</tr>
</tbody>
</table>

**Domain**

**NOTE** SQL Domain is modeled as UML DataType with «Domain» stereotype applied. For the sake of compactness, domains are displayed with the «domain» keyword on the diagram.

Domain allows to define a more narrow set of values than the base primitive type allows. This narrowing is done by assigning additional constraints on the domain. Columns, whose types are set to the domain, can only assume values from this more narrow subset.

Besides the standard SQL element properties, domain has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predefined Representation</td>
<td>Points to some base primitive type.</td>
</tr>
<tr>
<td>Default Value</td>
<td>Default value for the column if no value is specified.</td>
</tr>
</tbody>
</table>

**Structured User Defined Type**

**NOTE** SQL Structured User Defined Type is modeled as UML DataType with «StructuredUserDefinedType» stereotype applied. For the sake of compactness, domains are displayed with the «structured» keyword (instead of the long form - «StructuredUserDefinedType») on the diagram.

Structured UDT defines a composite datatype. Each value of this type is a tuple of several values; each position in a tuple has a name. Structured UDT value is analogous to one row of the table. Structured UDTs allow single inheritance (multiple inheritance is not supported). Inheritance (subtype-supertype relationship) can be modeled using UML Generalization relationships.
Besides the standard SQL element properties, structured UDT has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instantiable</td>
<td>Defines</td>
</tr>
<tr>
<td>Final</td>
<td>Default value for the column if no value is specified.</td>
</tr>
<tr>
<td>Super</td>
<td>Shows base data types. This is a derived field, it is not editable. To make changes, use UML Generalization relationships.</td>
</tr>
</tbody>
</table>

Parts of the structured UDT (properties) are called attributes (compare - parts of the table definition are called columns). Attributes of structured UDT are created like columns of the table, that is, via the Attribute Definitions tab in the structured UDT Specification window or using an appropriate smart manipulation button on its shape.

Besides the standard SQL element properties, attribute has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Collectively these two fields describe the type of the attribute. The same considerations as for column type modeling apply.</td>
</tr>
<tr>
<td>Type Modifier</td>
<td>Carries the default value of the attribute.</td>
</tr>
<tr>
<td>Default Value</td>
<td>Marks this attribute as scope checked to a particular table and allows choosing particular referential integrity ensuring action (RESTRICT CASCADE, etc).</td>
</tr>
</tbody>
</table>

Besides attributes, Structured UDTs have a collection of methods - operations, performing actions on values of this type. Methods are covered in a separate section with stored procedures and functions (see Routines section).

**Array Type**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>The elementary type of the set elements.</td>
</tr>
<tr>
<td>Max Cardinality</td>
<td>The size limit of the array.</td>
</tr>
</tbody>
</table>

SQL Array type is modeled as UML DataType with «ArrayDataType» stereotype applied. For the sake of compactness, arrays are displayed with the «array» keyword (instead of the long form - «ArrayDataType») on the diagram.

Array type defines an array (that is, list of values, with the indexed, O(1) access to the n-th element) of the values of elementary type.

Besides the standard SQL element properties, array type has the following properties available in the Specification window.
Multiset Type

NOTE SQL Multiset type is modeled as UML DataType with «MultisetDataType» stereotype applied. For the sake of compactness, multisets are displayed with the «multiset» keyword (instead of the long form - «MultisetDataType») on the diagram.

Multiset type defines a set of elements of the elementary type.

Besides the standard SQL element properties, multiset has the following properties available in the Specification window:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>The elementary type of the set elements.</td>
</tr>
</tbody>
</table>

Reference Types

NOTE SQL Reference type is modeled as UML DataType with «ReferenceDataType» stereotype applied. For the sake of compactness, references are displayed with the «ref» keyword (instead of the long form - «ReferenceDataType») on the diagram.

Reference type defines a pointer to the data of the referred type.

Besides the standard SQL element properties, reference type has the following properties available in the Specification window:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referenced Type</td>
<td>The type of the data that is being referenced.</td>
</tr>
<tr>
<td>Scope Table</td>
<td>Limit the references to the data of the particular table.</td>
</tr>
</tbody>
</table>

Row Type

NOTE SQL Row Data Type is modeled as UML DataType with «RowDataType» stereotype applied. For the sake of compactness, row data types are displayed with the «row» keyword (instead of the long form - «RowDataType») on the diagram.

Represents one row of the table. The difference from structured UDT is that row type represents a value stored in the table, while structured UDT represents “free-floating” value during computation. For example it is meaningful to take address for the row., but not of the structured UDT value.

Parts of the row data type (properties) are called fields (compare - parts of the table definition are called columns). Fields for row data type are created like columns of the table, that is, via the Fields tab in the row data type Specification window or using an appropriate smart manipulation button on its shape.

Besides the standard SQL element properties, field has the following properties available in the Specification window:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Collectively these two fields describe the type of the field. The same considerations as for column type modeling apply.</td>
</tr>
<tr>
<td>Type Modifier</td>
<td></td>
</tr>
</tbody>
</table>
DATABASE SUPPORT
Database Modeling

Sequences and Autoincrement Columns

SQL has facilities to generate sequences of numbers (0, 1, 2, 3, ...). These sequences are often used to fill in values for identifier columns - to uniquely number the row data in the table. There are 2 separate facilities:

- Standalone sequence object. This generator is not tied to any other object. Programer must explicitly query it to get the next value from the sequence and then use the retrieved value appropriately (usually in the INSERT statement to insert value for id column). Usually there are separate sequences for each table; sometimes the same sequence is reused for several columns.

- Autoincrement columns. Column of the table can be designated as autoincrement. When row is inserted into the table, if value of such column is not explicitly provided, one is generated automatically.

Cameo Data Modeler has modeling support for both kinds of sequences.

To create a standalone sequence

Do one of the following:

- Select theGLOBALS element shape on a diagram pane and click an appropriate smart manipulation button.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope Check</td>
<td>Marks this field as scope checked to a particular table and allows choosing particular referential integrity ensuring action (RESTRICT CASCADE, etc).</td>
</tr>
<tr>
<td>Scope Checked</td>
<td></td>
</tr>
</tbody>
</table>

NOTES

- SQL Sequence is modeled as UML Property with «Sequence» stereotype applied. For the sake of compactness, sequences are displayed with the «seq» keyword (instead of the long form - «Sequence») on the diagram.

- Autoincrement parameters (start value, increment, etc.) data is stored as a separate model element - UML OpaqueExpression, with «IdentitySpecifier» stereotype applied. This element is set as defaultValue of the Property - either sequence property (when standalone sequences are modeled) or column property (when autoincrement table columns are modeled).

Figure 23 -- Example of sequence and autoincrement column modeling
• Right-click the GLOBALS element in the Containment tree and on its shortcut menu, select **New Element > Sequence.**

**NOTE** Since a standalone sequence is modeled as a UML Property, it can not be placed directly into the Schema package.

Autoincrement columns are also supported. To mark a column as autoincrement, you must switch the **Default Value** property value type from value expression to identity specifier.

To mark a column as autoincrement

1. Open the column Specification window.
2. Select the **Default Value** property.
3. Click the black-arrowed button next to the property value and select **Value Specification > IdentitySpecifier** as shown in the following figure.

![Figure 24 -- Marking column as autoincrement](image)

*Figure 24 -- Marking column as autoincrement*
After the switching, the **Autoincrement** property group appears in the Specification window of the column allowing to specify autoincrement data (start value, increment, etc.).

![Autoincrement property group](image)

**Figure 25 -- Additional properties in autoincrement column’s Specification window**

Besides the standard SQL element properties and sequences, an autoincrement column has the following properties available in the **Autoincrement** property group of the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start Value</strong></td>
<td>Starting value of the sequence counter.</td>
</tr>
<tr>
<td><strong>Increment</strong></td>
<td>Delta value of the sequence counter (can be negative - to count down).</td>
</tr>
<tr>
<td><strong>Minimum</strong></td>
<td>Lower bound of the counter (if any).</td>
</tr>
</tbody>
</table>
DATABASE SUPPORT

Database Modeling

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>Upper bound of the counter (if any)</td>
</tr>
<tr>
<td>Cycle Option</td>
<td>The counter can “wrap around” when it reaches the maximum (or minimum - for downwards counters)</td>
</tr>
</tbody>
</table>

Additionally sequence has an **Identity** field and column has the **Default Value** field, where textual representation of the counter options can be entered. This feature can be used for nice displaying of the counter configuration in the diagrams (the start, inc, min, max field data is normally not visible in the diagram). Some notation convention should be adopted how to map the counter data into the text representation. For example it could be: {<start>, <inc>, <min>-<max>, <c>}. Then the counter from 0 with +1 increment, min max of 0 and 1000 and cycle option would be displayed as “{0, +1, 0-1000, C}” string. At the moment this text representation is not automatically connected to the counter field values, so synchronization has to be done by hand.

**Constraints**

Tables have a multitude of constraints between them. These constraints enforce the proper business semantics for the data in the database tables (relationships between data in different tables, semantical constraints of the data in the table). There are these available constraint types:

- Primary key constraints - specifying column (or a combination of columns), which uniquely identify the row in the table.
- Unique constraints. They are very similar to primary key constraints - uniquely identify the row in the table. One of the unique constraints of the table is designated as primary.
- Foreign key constraints, which establish relationships between two tables.
- Nullability constraints (NOT NULL constraint) - a simple constraint on the column, indicating that column must have value
- Check constraints establish additional checking conditions on values in the column / table.
- Assertions provide more global check than a check constraint - spanning multiple tables
- Indexes are not constraints per se, but they are covered in this section because they are modeled similarly.

The primary keys, unique and check constraints, indexes can be modeled in two ways. One way is easy and simple but does not cover all the options provided by SQL. Another way is tedious, but provides full SQL coverage.
Implicit Primary Key, Unique, Check Constraint, and Index Modeling

NOTES

- SQL Primary Key (when implicitly modeled) is modeled as an additional «PrimaryKeyMember» stereotype applied on the SQL column. This variant is shown in the diagram as an additional «pk» keyword on the column in the diagram.
- SQL Unique Constraint (when implicitly modeled) is modeled as an additional «UniqueMember» stereotype applied on the SQL column. This variant is shown in the diagram as an additional «unique» keyword on the column in the diagram.
- SQL Check Constraint (when implicitly modeled) is modeled as an additional «CheckMember» stereotype applied on the SQL column. This variant is shown in the diagram as an additional «chk» keyword on the column in the diagram.
- SQL Index (when implicitly modeled) is modeled as an additional «IndexMember» stereotype applied on the SQL column. This variant is shown in the diagram as an additional «idx» keyword on the column in the diagram.

An easy way of modeling this kind of constraint is applying the «PrimaryKeyMember», «UniqueMember», «CheckMember», or «IndexMember» stereotype on the necessary column. PK, unique, and index markers can be applied on the column via its shortcut menu as shown in the following figure.

Figure 26 -- Quick application of PK, Unique, and Index markers
To apply a check constraint marker on a column:

1. Open the Specification window of the column.
2. Define the **Condition** property value in the **In Check Constraint** property group.

Thusly marked column is considered as a member of one-column constraint, specified in-line. It is by default an unnamed constraint. To specify its name, you need to define the **Primary Key Name**, the **Unique Key Names**, the **Check Name**, or the **Index Names** property value in the column Specification window.

In the SQL script (in CREATE TABLE, ADD COLUMN statements) this would correspond to the following part of the column specification:

```sql
<column name> [ <data type> ] ... 
[ [<constraint name>]<constraint>... ] 
<constraint> ::=  
  | UNIQUE | PRIMARY KEY  
  | CHECK '('<condition>')' 
```

If primary key, unique constraint or index must span several columns (in this case constraint is not in-line, near the column definition, but as a separate definition at the bottom of the table definition), all the columns must be marked with the appropriate «UniqueMember» / «IndexMember» stereotype and all must have the same name. Column can participate in several unique or

Various cases of quick constraint modeling are depicted in the following figure.

---

*Figure 27 -- Various situations, modeled with quick constraint notation*
Explicit Primary Key, Unique, Check Constraint, and Index Modeling

The quick, implicit way to model constraints does not cover some cases, allowed by SQL. Constraints in SQL can be marked as DEFERABLE, INITIALY DEFERRED; constraint in the database can be in active state (enforced) or disabled. Indexes have various configuration parameters.

Modeling with the help of «XYZMember» stereotypes does not allow to specify this additional information. In this case modeling with explicit constraint model elements is necessary. This can be done from the Specification window of table. There are separate tabs for creating these constraint elements: **Unique Constraints** (allows creating both primary keys and unique constraints), **Check Constraints, Indices**. Once created, additional properties of the constraints can be specified.

Besides the standard SQL element properties, primary key and unique constraint have following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td>Columns, constrained by this constraint (must come from the same table where constraint is located)</td>
</tr>
<tr>
<td>Inline</td>
<td>Whether constraint is defined near the column definition, or separately, at the bottom of the bale definition. Only one-column constraints can be inline.</td>
</tr>
<tr>
<td>Deferable</td>
<td>Marks the constraint as deferrable.</td>
</tr>
<tr>
<td>Initially Deferred</td>
<td>Marks the constraint as initially deferred.</td>
</tr>
<tr>
<td>Enforced</td>
<td>Whether constraint is actively checked in the database (can be changed with the SQL statements).</td>
</tr>
</tbody>
</table>

Check constraints have the same properties as primary key and unique constraints, and additionally have following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>Condition to be checked</td>
</tr>
</tbody>
</table>

Besides the standard SQL element properties, index has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td>Member columns of this index</td>
</tr>
<tr>
<td>Member Increment Type</td>
<td>For each member column, ASC or DESC ordering direction</td>
</tr>
<tr>
<td>Unique</td>
<td>Index is used to enforce uniqueness</td>
</tr>
<tr>
<td>System Generated</td>
<td>Index is system-generated.</td>
</tr>
</tbody>
</table>
Database Modeling

Foreign Keys

Foreign keys describe relationships between two tables. At the detailed understanding level, foreign key is a constraint on the (group of) columns in the source / referencing table, such that for each row in the source table their value combination (tuple) is equal to the value combination (tuple) of the (group of) columns for some row in the target / referenced table.

Foreign keys also have the two ways to be modeled. The main way is described below.

The main way to model foreign keys is to draw association relationship from the referencing table to the referenced table. The relationship can be simply draw in the diagram from the smart manipulator or from the button in the diagram toolbar.
When the FK association is draw, the following dialog pops up:

![Foreign Key dialog](image)

*Figure 29 -- Foreign Key dialog*

Note that you have to have the necessary columns in the tables (PK or unique columns in target table, referencing FK columns in the source table) before drawing the FK relationship. In this dialog, select the referenced columns (of the target table) in the first column of the table, and corresponding referencing columns (of the source table). Additionally, foreign key name can be specified.

When dialog is OK’d, foreign key association is created; «FK» stereotype is applied on the referencing association end and the selected column information is stored in tags.

If foreign key information has to be changes, this is done in the Specification window of the FK property. Besides the standard SQL element properties foreign key has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline</td>
<td>The same functionality as for the explicitly modeled PK, unique constraints</td>
</tr>
<tr>
<td>Deferable</td>
<td></td>
</tr>
<tr>
<td>Initially Deferred</td>
<td></td>
</tr>
<tr>
<td>Enabled</td>
<td></td>
</tr>
<tr>
<td>Match</td>
<td>Specifies how the columns in the referenced table and columns in the referencing table are matched (SIMPLE, FULL, PARTIAL match).</td>
</tr>
<tr>
<td>On Delete</td>
<td>Referential integrity enforcement action that is performed when the data in the referenced table is deleted (NO ACTION, RESTRICT, CASCADE, SET NULL, SET DEFAULT)</td>
</tr>
<tr>
<td>On Update</td>
<td>Referential integrity enforcement action that is performed when the data in the referenced table is updated (NO ACTION, RESTRICT, CASCADE, SET NULL, SET DEFAULT)</td>
</tr>
<tr>
<td>Referencing Members</td>
<td>Member columns of the constraint (from the same table where constraint is located). FK constrains values of these columns to point to the data in the referenced tables</td>
</tr>
</tbody>
</table>
The alternative way of modeling a foreign key is creating a UML constraint with the «ForeignKey» stereotype applied. This way is less desired than the main way, because it does not visualize relationship between tables. It is just a constraint in the table. This method may be used when human-readability is not critical, e.g., when database layout is generated with some custom automated script / transformation in the model.

To create a constraint with the «ForeignKey» stereotype

1. Select a table in the Containment tree.
2. Do one of the following:
   • Right-click the selected element and from its shortcut menu select New Element > Explicit Foreign Key.
   • Open the Explicit Foreign Keys tab in the table’s Specification window.

Besides the standard SQL element properties and properties that are available for other explicit constraints (that is, PK, unique, check constraints), explicit foreign key has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>Specifies how the columns in the referenced table and columns in the referencing table are matched (SIMPLE, FULL, PARTIAL match).</td>
</tr>
<tr>
<td>On Delete</td>
<td>Referential integrity enforcement action that is performed when the data in the referenced table is deleted (NO ACTION, RESTRICT, CASCADE, SET NULL, SET DEFAULT)</td>
</tr>
<tr>
<td>On Update</td>
<td>Referential integrity enforcement action that is performed when the data in the referenced table is updated (NO ACTION, RESTRICT, CASCADE, SET NULL, SET DEFAULT)</td>
</tr>
<tr>
<td>Referencing Members</td>
<td>Member columns of the constraint (from the same table where constraint is located). FK constrains values of these columns to point to the data in the referenced tables</td>
</tr>
</tbody>
</table>
Nullability Constraint

NOTE SQL NOT NULL constraint (if modeled explicitly, which is rare!) is modeled as UML Constraint with «NotNullConstraint» stereotype applied.

Nullability, or NOT NULL constraint forces the condition that the column must have value. Implicit NOT NULL constraint is modeled with the nullable field of the column (set nullable=false to specify NOT NULL). This is an usual and quick way to model these constraints.

Usually there is no need to model these constraints explicitly - create a separate model element for them. But in the more complex cases these constraints can be created by hand and the «NotNullConstraint» stereotype applied on them. This allows specifying non-inline constraints, or named constraints, or deferred constraints or inactive constrains.

NOT NULL constraint does not have any additional properties in the Specification window besides the properties that all table constraints have.

Assertion

NOTE SQL Assertion is modeled as UML Constraint with «Assertion» stereotype applied.

Assertion constraints are very similar to check constraints, but instead of being local to the table, they are global to the database. Assertions check some condition that must hold through several tables. Assertions are modeled as explicit constraints; there is no shorthand modeling form - assertion is always an explicit UML constraint.

To create an assertion

1. Select a schema or a database element in the Containment tree.
2. Right-click the selected element and from its shortcut menu select New Element > Assertion.
Besides the standard SQL element properties assertion has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Condition</td>
<td>The assertion body condition</td>
</tr>
<tr>
<td>Constrained Tables</td>
<td>List of the tables on which assertion runs</td>
</tr>
</tbody>
</table>

**Triggers**

**NOTE** SQL Trigger is modeled as UMLOpaqueBehavior with the «Trigger» stereotype applied.

Trigger describes some action that must be performed when some particular data manipulation action is being performed in the database table. Trigger can be fired when data is added, deleted or changed in the table and perform some action (update some other table, calculate additional values, validate data being updated or even change the data that is being updated).

Trigger is always defined for some table. You can define triggers in the Triggers tab of the table Specification window. Trigger has an event type (on what actions trigger is fired, that is, on insert, on update, or on delete), action time (before, after, instead of), and an actual body describing the actions.

Besides the standard SQL element properties, trigger has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Time</td>
<td>Specifies moment of time when trigger action is performed (before the specified event, after event, instead of event).</td>
</tr>
<tr>
<td>On Insert</td>
<td>The event that causes trigger firing.</td>
</tr>
<tr>
<td>On Update</td>
<td></td>
</tr>
<tr>
<td>On Delete</td>
<td></td>
</tr>
<tr>
<td>Trigger Column</td>
<td>List of columns, which cause trigger fire on update (must be from the same table as trigger is located). Used with On Update triggers to specify that only some column updates cause trigger fire.</td>
</tr>
<tr>
<td>Language</td>
<td>Trigger implementation language (should be SQL).</td>
</tr>
<tr>
<td>Body</td>
<td>Trigger body text (operations that are performed on trigger fire)</td>
</tr>
<tr>
<td>Time Stamp</td>
<td>Trigger creation timestamp</td>
</tr>
<tr>
<td>Action Granularity</td>
<td>Specifies whether trigger fires once per executed statement, or once per each affected row</td>
</tr>
<tr>
<td>When</td>
<td>Specifies additional precondition for trigger firing</td>
</tr>
<tr>
<td>New Row</td>
<td>These fields can be used for defining variable names for holding new row / table values and old row / table values - for referencing in trigger body.</td>
</tr>
<tr>
<td>New Table</td>
<td></td>
</tr>
<tr>
<td>Old Row</td>
<td>REFERENCING ((NEW</td>
</tr>
<tr>
<td>Old Table</td>
<td></td>
</tr>
<tr>
<td>Additional Actions</td>
<td>Additional action statements. This option is rarely used - it is non-standard and supported only by some databases. Usually triggers have just one body.</td>
</tr>
</tbody>
</table>
**Routines**

SQL supports several different kinds of routines. There are global routines, that are not bound to a particular type but belongs to the schema. There are two kinds of these routines - Procedures and Functions. And there are routines, that are bound to a particular structured user defined type - Methods. Each routine kind can have several parameters. Parameters have type and direction (in, out, inout). Functions and methods in SQL have return types - this is formalized in UML models by having an additional parameter with return direction kind.

There is an UML limitation, that UML does not allow to place UML operations (which are used to model SQL procedures and functions) directly in the UML packages (which are used to model SQL schemas). For this reason global routines are placed into a special container class - GLOBALS (see “GLOBALS” on page 23).

Routines can be external (written in some other languages and attached to database engine) or SQL routines. In the latter case, body of the routine can be specified in the model. Due to UML specifics, there are two ways to specify the routine body - by filling the UML method field or by filling the UML bodyCondition field of the operation. These two ways are visible in the Specification window under the field names Source (as method) and Source (as body condition). When specifying routine body, specify only one of these fields.

To use “as method” way

1. Right-click the GLOBALS element in the Containment tree and from its shortcut menu select New Element > Source. A source element (a UML OpaqueBehavior with the «Source» stereotype applied) under the GLOBALS element will be created in your schema.
2. In the Specification window of routine, edit the Source (as method) property value and in the opened dialog select the source element you’ve just created.
To use “as body condition” way, you simply have to fill the field. The routine body model element (in this case UML Constraint - holding UML OpaqueExpression) shall be created under your routine model element.

Besides the standard SQL element properties, all 3 kinds of routines have the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific Name</td>
<td>Additional name for the routine, uniquely identifying it throughout the system.</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Specifies whether routine is deterministic (always gives the same output with the same data input)</td>
</tr>
<tr>
<td>Parameter Style</td>
<td>SQL or GENERAL</td>
</tr>
<tr>
<td>SQL Data Access</td>
<td>Specifies how routine accesses SQL data (NO SQL</td>
</tr>
<tr>
<td>Source (as method)</td>
<td>Fields holding routine body text (chose only one).</td>
</tr>
<tr>
<td>Source (as body condition)</td>
<td></td>
</tr>
<tr>
<td>Creation TS</td>
<td>Routine creation and last edit timestamps</td>
</tr>
<tr>
<td>Last Altered TS</td>
<td></td>
</tr>
<tr>
<td>Authorization ID</td>
<td>Authorization identifier which owns this routine (owner of the schema at routine creation time)</td>
</tr>
<tr>
<td>Security</td>
<td>Determines the authorization identifier under which this routine runs. Tipically set to “INVOKER”, “DEFINER”, “IMPLEMENTATION DEPENDENT”</td>
</tr>
<tr>
<td>External Name</td>
<td>The name of the external language routine implementation method (if routine is non-SQL routine)</td>
</tr>
</tbody>
</table>

**Procedure**

**NOTE** SQL Procedure is modeled as UML Operation with «Procedure» stereotype applied. For the sake of compactness, procedures are displayed with the «proc» keyword (instead of the long form - «Procedure») on the diagram.

Procedure is an operation that can be SQL-invoked and performs some actions depending on the parameters supplied. Procedures are global for schema - they are created under the GLOBALS model element.

Besides the standard properties of SQL routines (see “Routines” on page 46), procedure has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Result Sets</td>
<td>If result set count returned by procedure is dynamic, this value limits count thereof (DYMANIC RESULT SETS &lt;max&gt; clause)</td>
</tr>
<tr>
<td>Old Save Point</td>
<td>Savepoint level indicator for procedure (false means that new savepoint must be established before the procedure is run)</td>
</tr>
</tbody>
</table>
FUNCTION

NOTE
SQL Function is modeled as UML Operation with «Function» or «BuiltInFunction» or «UserDefinedFunction» stereotype applied. By default the «UserDefinedFunction» is used, however if another kind can be freely used if it is necessary for modeling needs (e.g. if we are modeling some built in library and want to specify that functions are built-in and not user defined).

For the sake of compactness, functions are displayed with the «func» keyword (instead of the long form) on the diagram.

Function describes some operation that calculates and returns some value depending on the parameters supplied. Functions are global for schema - they are created under the GLOBALS model element.

Besides the standard properties of SQL routines (see section above), function has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Call</td>
<td>Specifies that function returns NULL when called with NULL parameter value (RETURNS NULL ON NULL INPUT clause)</td>
</tr>
<tr>
<td>Type Preserving</td>
<td>Specifies that function does not change the type of the supplied parameter (returns the same object)</td>
</tr>
<tr>
<td>Transform Group</td>
<td>Allows to specify TRANSFORM GROUP &lt;groups&gt; clause - single or multiple.</td>
</tr>
</tbody>
</table>

METHOD

NOTE
SQL Method is modeled as UML Operation with «Method» stereotype applied. For the sake of compactness, methods are displayed with the «func» keyword (instead of the long form - «Method») on the diagram.

Method is a function of the structured user defined type. It is created inside the structured UDT.

Besides the properties of SQL functions (see section above), method has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Specifies that function is a constructor (that is, it is used to construct values of the enclosing structured UDT).</td>
</tr>
<tr>
<td>Overriding</td>
<td>Specifies that function is overriding the same-named function from the parent structured UDT</td>
</tr>
</tbody>
</table>

PARAMETER

NOTE
SQL Parameter is modeled as UML Parameter with «Parameter» stereotype applied.

This model element specifies data inputs / outputs into routine calculations. Parameter has a type, direction (in / out / inout for usual parameters and a single parameter with direction return for functions) and default value.
Besides the standard SQL element properties, parameter has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Type of the parameter</td>
</tr>
<tr>
<td>Type Modifier</td>
<td></td>
</tr>
<tr>
<td>Default Value</td>
<td>Default value (used when value is not supplied during routine invocation).</td>
</tr>
<tr>
<td>Direction</td>
<td>Direction of data flow through the parameter (into the routine, out of the routine or both)</td>
</tr>
<tr>
<td>Locator</td>
<td>AS LOCATOR modifier of the type. Specifies that instead of value, means to locate value are transferred</td>
</tr>
<tr>
<td>String Type Option</td>
<td>Only valid when parameter type is XML type. Specifies underlying string datatype.</td>
</tr>
<tr>
<td>Cast Type</td>
<td>Additional options, specifying that return parameter is cast from another type (possibly with locator indication).</td>
</tr>
<tr>
<td>Cast Locator</td>
<td></td>
</tr>
</tbody>
</table>

**Cursor and Routine Result Table**

**NOTE** SQL Cursor is modeled as UML Parameter with the «Cursor» stereotype applied.

When routine does not return a scalar value but a collection of the table values, cursor is used, instead of the parameter. Cursor has a type. This type must be some table type instead of the scalar types used for parameters. It can be an actual table / view from the model, if cursor returns values from that table, or (if cursor returns data from some SELECT statement) can be a synthetic table. A Routine Result Table model element is used for this purpose (UML Class with «RoutineResultTable» stereotype applied). It's modeling is exactly the same as the normal tables - this is just an ephemeral table.

Besides the standard SQL element properties, cursor has the following properties available in the Specification window:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Type of the cursor. Should point to the routine result table.</td>
</tr>
<tr>
<td>Type Modifier</td>
<td></td>
</tr>
<tr>
<td>Direction</td>
<td>Direction of data flow through the parameter (into the routine, out of the routine, routine result)</td>
</tr>
</tbody>
</table>
Access Control

SQL has means to specify and control the rules of access to various data objects. This subset of SQL language is sometimes called Data Control Language. The relevant concepts are: User, Group, Role (3 different kinds of authorization subjects), Permission and Role Authorization (2 kinds of access control rules). Possible object types for access control varies depending on database flavor, but usually Tables, User-defined Types, Domains, Routines, Sequences can be specified as the target objects of access control.

User

NOTE SQL User is modeled as UML Actor with the «User» stereotype applied.

User object represents the single user person in the system, User is subject to access control rules.

Besides the standard SQL element properties, user has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owned Schema</td>
<td>Schemas that are owned by this user,</td>
</tr>
</tbody>
</table>

Group

NOTE SQL Group is modeled as UML Actor with the «Group» stereotype applied.

Group object represents a collection of Users. Group is subject to access control rules, and allows specifying access control rules on several users simultaneously.

Besides the standard SQL element properties, group has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Collection of users the group is made of.</td>
</tr>
<tr>
<td>Owned Schema</td>
<td>Schemas that are owned by this group,</td>
</tr>
</tbody>
</table>
Role

**NOTE** SQL Role is modeled as UML Actor with the «Role» stereotype applied.

Role object represents a specific role (typical activities) that can be played by users. Role is subject to access control rules, and allows specifying access control rules for all subjects, playing this role.

Besides the standard SQL element properties, role has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owned Schema</td>
<td>Schemas that are owned by this role,</td>
</tr>
</tbody>
</table>

Privilege

**NOTE** SQL Privilege is modeled as UML Dependency with the «Privilege» stereotype applied.

Privilege relationship expresses the fact that the permission to perform specified action on specified object (relationship target) is granted to specified grantee (relationship source). Grantee can be any authorization subject - Use, Group or another Role. Object can by another SQL object (the precise list of object types, that can be targeted by privileges, varies by database type).

Privilege corresponds to SQL grant privilege statement:

\[
\text{GRANT <action>[(<column list>)] ON <object> TO <grantee> [WITH HIERARCHY OPTION] [WITH GRANT OPTION]}
\]

Besides the standard SQL element properties, privilege has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Specifies action that is being granted (such as SELECT or UPDATE).</td>
</tr>
<tr>
<td>Action Objects</td>
<td>Specifies additional more narrow subobject list, on which the specified action is permitted (usually column list for SELECT or UPDATE).</td>
</tr>
<tr>
<td>Grantable</td>
<td>Specifies that this privilege can be further re-granted to other subjects by the recipients. Corresponds to WITH GRANT OPTION part of GRANT statement.</td>
</tr>
<tr>
<td>With Hierarchy</td>
<td>Specifies that this privilege applies to subobjects (subtables). Corresponds to WITH HIERARCHY OPTION part of the GRANT statement.</td>
</tr>
<tr>
<td>Grantor</td>
<td>Subject, who grants this privilege to the grantees.</td>
</tr>
</tbody>
</table>

Role Authorization

**NOTE** SQL Role Authorization is modeled as UML Dependency with «RoleAuthorization» stereotype applied.

Role authorization relationship expresses the fact that the specified role (relationship target) is granted to specified grantee (relationship source). Grantee can be any authorization subject - Use, Group or another Role.

Role authorization corresponds to SQL grant role statement:

\[
\text{GRANT <role> TO <grantee> [WITH ADMIN OPTION]}
\]
Besides the standard SQL element properties, role authorization has the following properties available in the Specification window.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grantable</td>
<td>Specifies that this role can be further re-granted to other subjects by the recipients. Corresponds to WITH ADMIN OPTION part of GRANT statement.</td>
</tr>
<tr>
<td>Grantor</td>
<td>Subject, who grants this role to the grantees.</td>
</tr>
</tbody>
</table>

**Oracle Database Modeling Extensions**

When the Oracle flavor is chosen for database top level element (schema or database), additional Oracle extensions are brought in. Elements that are in the scope of this schema or database element obtain additional Oracle-specific properties in the Specification windows (under the separate Oracle property group). These properties carry an additional information, that is then used when generating DDL scripts for Oracle.

Most often there is just one **Additional Properties** property - allowing entering free-form text that is then used when generating (this can be used to specify any extension texts - such as tablespace options for tables).

Oracle extensions provide means to model synonyms. Synonym is mapped as follows:

- Element of the same type (that is, table, materialized view, stored procedure, sequence) as the one being aliased is created. It is stereotyped as appropriate, but have no other data - just it's name is important.
- Additionally, stereotype «OraSynonym» will be applied on the element. It has ref:Element[1] tag for pointing to the element being aliased. Synonyms of synonyms are handled in the same way.

Oracle extensions provide means to model materialized views. Materialized view can be created from Oracle SQL diagram. It is an ordinary SQL view, but with the additional «OraMaterializedView» stereotype applied (in diagrams, a shortened keyword «materialized» is used for the sake of compactness).

**Database Code Engineering**

Cameo Data Modeler provides code engineering capabilities database script generation and reverse engineering. Database model can be generated in to the DDL script, which can then be run on the database engine to create database structures. And conversely database engine can export database structures into the DDL script, which can then be reverse-engineered into the database model. In addition to that Cameo Data Modeler provides reverse engineering directly from live database through JDBC connection.

Cameo Data Modeler code engineering supports the following database dialects for script generation and reversing:

- Standard SQL
- Oracle
- MS SQL Server
- DB2
- MySQL
- PostgreSQL
- Sybase
As was mentioned earlier, database modeling was significantly extended in the version v17.0.1. But database code engineering has remained at the same level as before. Hence currently not all database concepts, that can be modeled, can be subsequently generated or reverse engineered. This situation will be amended in the future.

### Code Engineering Set

Code engineering set for database scripts can be created in the same manner as CE sets for other code types (see “Code Engineering Sets” in [MagicDraw CodeEngineering UserGuide.pdf](#)). Right-click the Code engineering Sets, New, DDL, and then the appropriate database flavor. When the CE set is created, database model elements can be added to it and then DDL script file(s) can be generated OR the script files can be added to the CE set and then reverse-engineered into the database models. In addition to reversing from files, there is Reverse from DB radio button. Once it is switched, the options for JDBC connection configuring appear, allowing to set up connection to the live database.

<table>
<thead>
<tr>
<th>Box name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Recently Used</strong></td>
<td>Contains the list of the recently used reverse templates. Choose the one you need and click <strong>Apply</strong>.</td>
</tr>
<tr>
<td><strong>DB Connection URL</strong></td>
<td>The connection URL for the selected profile.</td>
</tr>
<tr>
<td><strong>Driver Files</strong></td>
<td>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 <strong>Select Files and/or Directories</strong> dialog appears. <strong>NOTE</strong> If the driver file is empty, Driver Class is searched from the classpath.</td>
</tr>
<tr>
<td><strong>Driver Class</strong></td>
<td>Contains the connection driver class. Click the ... button and the list of available driver classes that are available in the selected driver files is displayed. <strong>NOTE</strong> Only in the files that are selected in the <strong>Driver Files</strong> list, the system searches for driver classes.</td>
</tr>
<tr>
<td><strong>Username</strong></td>
<td>Type the username to connect to the database.</td>
</tr>
<tr>
<td><strong>Password</strong></td>
<td>Type the password to connect to the database.</td>
</tr>
<tr>
<td><strong>Catalog</strong></td>
<td>Contains a name of the selected Catalog. To retrieve the list of available Catalogs from the database, click the ... button and select the catalog. The catalog name appears in the <strong>Catalog</strong> text box. <strong>NOTE</strong> Only when all other properties in this dialog box are correctly defined, the list of catalogs can be retrieved.</td>
</tr>
<tr>
<td><strong>Schema</strong></td>
<td>Contains a name of the selected Schema. To retrieve the list of available Schemas from the database, click the ... button and select the schema. The schema name appears in the <strong>Schema</strong> text box. <strong>NOTE</strong> Only when all other properties in this dialog box are correctly defined, the list of schemas can be retrieved.</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

![Figure 32 -- DDL properties in CG Properties Editor dialog](image)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Value list</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>Property name</td>
<td>Value list</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: «Database» package “MQOnline” includes «Table» class “libraries”. Then check box Generate Qualified Names is selected as true in generated source would be written: CREATE TABLE MQOnline.libraries; Then check box Generate Qualified Names is selected as false, in generated source would be written: CREATE 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>
### 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).

### Properties for DDL script reverse engineering

<table>
<thead>
<tr>
<th>Property name</th>
<th>Value 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 stereotype tag values or multiplicity.</td>
</tr>
<tr>
<td>Map foreign keys</td>
<td>True (default), false</td>
<td>An association with «FK» stereotype on the association end is created, to represent Foreign Key.</td>
</tr>
<tr>
<td>Map indexes</td>
<td>True (default), false</td>
<td>A constraint with «Index» stereotype is added into table, to represent index.</td>
</tr>
<tr>
<td>Map triggers</td>
<td>True (default), false</td>
<td>An opaque behavior with «Trigger» stereotype is added into table to represent trigger.</td>
</tr>
</tbody>
</table>
Supported SQL Statements

This section lists SQL statements that are supported in the Cameo Data Modeler plugin. They are parsed and mapped into model constructs.

The following table provides SQL2 SQL schema statements and their support status in the Cameo Data Modeler plugin (Yes means that a statement can be generated into DLL script from model constructs and reverse engineered from script into model constructs).

<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></td>
</tr>
<tr>
<td>View definition</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Alter table statement</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Grant statement</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Domain definition</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Assertion definition</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Character set definition</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Collation definition</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Translation definition</td>
<td>No</td>
<td></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></td>
</tr>
<tr>
<td>Drop view statement</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Revoke statement</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Alter domain statement</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Drop assertion statement</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Drop domain statement</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Drop character set statement</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Drop collation statement</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Drop translation statement</td>
<td>No</td>
<td></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:

- Set catalog statement - sets the current default catalog name.
- Set schema statement - sets the current default unqualified schema name.

Cameo Data Modeler 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 SQL constructs.

<table>
<thead>
<tr>
<th>DDL Statement or Concept</th>
<th>Action, model Item</th>
<th>Description</th>
<th>Visible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alter table statement</td>
<td>Modify class</td>
<td>Elements: table name and alter table action. Alter table action – one of: add column, add table constraint, alter column, drop table constraint, drop column.</td>
<td>Yes</td>
</tr>
<tr>
<td>Add column definition</td>
<td>Define attribute</td>
<td>Elements: column definition.</td>
<td>Yes</td>
</tr>
<tr>
<td>Add table constraint definition</td>
<td>Define method</td>
<td>Elements: table constraint definition.</td>
<td>Yes</td>
</tr>
<tr>
<td>Alter column definition</td>
<td>Modify attribute</td>
<td>Elements: mandatory column name, default clause (for add default statement only).</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop table constraint definition</td>
<td>Delete method</td>
<td>Elements: constraint name, drop behavior</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop column definition</td>
<td>Delete attribute</td>
<td>Elements: column name, drop behavior</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop schema statement</td>
<td>Delete package</td>
<td>Elements: schema name, drop behavior</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop table statement</td>
<td>Delete class</td>
<td>Elements: table name, drop behavior</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop view statement</td>
<td>Delete class</td>
<td>Elements: table name, drop behavior</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop behavior</td>
<td>Action property</td>
<td>Modifiers: CASCADE, RESTRICT</td>
<td>No</td>
</tr>
</tbody>
</table>

**DDL Dialects**

This section reviews Cameo Data Modeler support for DDL script flavors from different vendors.

**Standard SQL2**

For SQL2 statements supported by Cameo Data Modeler see Section Supported SQL Statements, ”Supported SQL Statements”, on page 57.

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).

**Oracle**

Cameo Data Modeler Oracle DDL script generation is based on the Velocity engine. This provides ability to change generated DDL script by changing velocity template. In this chapter we will introduce how Oracle DDL generation works in MagicDraw, how to change template for some specific things.

Knowledge of the Velocity Template Language is necessary for understanding, editing, or creating templates. Velocity documentation can be downloaded from: http://click.sourceforge.net/docs/velocity/Velocity-UsersGuide.pdf.

For more information about Oracle DDL generation and customization, see MagicDraw OpenAPI UserGuide.pdf.
Oracle dialect

For more information about Oracle DDL 11g, see http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/toc.htm

Oracle dialect has CREATE DATABASE, CREATE INDEX, and CREATE TRIGGER statements that are not the part of SQL2 standard but that are 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.

Code engineering features for Oracle dialect are more extensive that code engineering for other dialects. In addition to the concepts, supported by Standard SQL generation, Oracle generation supports generation and reverse of:

- Sequences
- Synonym
- Structured user-defined types (with methods, map & order functions)
- Function and Procedure
- Users, Roles Grants
- Materialized Views

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.
TRANSFORMATIONS

Introduction

The Cameo Data Modeler plugin for MagicDraw provides a set of transformation for transforming between various kinds of data models. There are transformations for transforming:

- UML models to SQL models (2 flavors - generic and Oracle)
- ER models to SQL models (2 flavors - generic and Oracle)
- SQL models to UML models (suitable for all flavors of databases)
- UML models to XML schema models
- XML schema models to UML models

After the transformation, user can further refine the resulting model as necessary, and generate the artifact files from those models. Actual DDL scripts, XML schema files can be generated using the code engineering facilities.

Since the Cameo Data Modeler plugin provides more powerful modeling and generation features for Oracle database flavor (there are Oracle-specific modeling extensions, and code engineering features for Oracle database scripts cover more features), there are two separate transformation flavors as well - "ER to SQL(Generic)" and "ER to SQL(Oracle)".

Functionality of performing model transformations in MagicDraw is accessible by the **Model Transformations Wizard**. The wizard is used for creating new transformations.

To open the **Model Transformations Wizard**

Do one of the following:
- From the **Tools** menu, choose **Model Transformations**.
- Select one or more packages. From the shortcut menu, choose **Tools > Transform**.

The Cameo Data Modeler plugin for MagicDraw provides a set of transformation for transforming between various kinds of data models. There are transformations for transforming:

- UML models to SQL models (2 flavors - generic and Oracle)
- ER models to SQL models (2 flavors - generic and Oracle)
- SQL models to UML models (suitable for all flavors of databases)
- UML models to XML schema models
- XML schema models to UML models

After the transformation, user can further refine the resulting model as necessary, and generate the artifact files from those models. Actual DDL scripts, XML schema files can be generated using the code engineering facilities.

Since the Cameo Data Modeler plugin provides more powerful modeling and generation features for Oracle database flavor (there are Oracle-specific modeling extensions, and code engineering features for Oracle database scripts cover more features), there are two separate transformation flavors as well - "ER to SQL(Generic)" and "ER to SQL(Oracle)".

NOTE: As of version 17.0.1 the Generic-Oracle DDL(SQL) transformation is no longer available in MagicDraw. The transformation is no longer needed, because of unification of previously separate profiles for generic SQL and Oracle SQL modeling.

Functionality of performing model transformations in MagicDraw is accessible by the **Model Transformations Wizard**. The wizard is used for creating new transformations.

To open the **Model Transformations Wizard**

Do one of the following:
- From the **Tools** menu, choose **Model Transformations**.
- Select one or more packages. From the shortcut menu, choose **Tools > Transform**.

NOTE: For more information about this wizard, see “Model Transformation Wizard” in **MagicDraw UserManual.pdf**.
Each transformation has its own default type map for replacing data types from the source domain into the appropriate data types of the result domain. If this type map is not suitable, the default type map can be modified or an entirely different type map can be provided if necessary.

NOTE
- For more information on how to create your own transformation rules or change mapping behavior, see “Transformation Type Mapping” in the MagicDraw UserManual.pdf.
- For more information on how to set up the mapping, watch the “Transformations” online demo at www.magicdraw.com/viewlets.

UML to SQL Transformation

There are two very similar UML to DDL transformations:

1. UML to SQL(Generic)
2. UML to SQL(Oracle)

These transformations convert the selected part of a UML model with class diagrams into Generic or Oracle SQL models with or Oracle SQL diagrams respectively.

Transformation Procedure

UML to SQL(Generic / Oracle) transformation is based on the same copy mechanism as the other transformations are. It copies the source model part to the destination (unless the in-place transformation is performed), remaps types, and then performs model reorganization to turn the model into a SQL model.

Conversion of Classes

UML classes from the source model are converted into tables.

Each property of the source class becomes a column in the result table. If a property in the UML model had the explicit multiplicity specified, nullable=true (for \([0..1]\) multiplicity in source property) and nullable=false (for \([1]\) multiplicity in source property) marking is applied on result columns.

Operations contained in the source class are not copied into the result table.

Primary Keys Autogeneration

If a UML class in the source model had no primary key (it is declared by applying an appropriate stereotype), an ID column is generated and marked as the primary key.

The Autogenerated PK name template transformation property governs the name of the generated ID column. %t pattern in the name template is expanded to the current table name.

The Autogenerated PK type transformation property determines the type of the ID column. The default type is integer.

Conversion of Associations

One-to-one and one-to-many associations between classes in the source UML model are converted to foreign key relationships and to foreign key columns in the table, which is at the multiple end.
The **Autogenerated FK name template** transformation property governs the name of the generated FK column. A name template can use the following patterns:

- $%t$ is replaced by the name of the table, the foreign key is pointing to.
- $%k$ is replaced by the key name, this foreign key is pointing to.
- $%r$ is replaced by the name of the relationship role, which is realized by this foreign key.

Note that the type of the FK column matches the type of the PK column, to which this key is pointing.

Many-to-Many associations are transformed into the intermediate table. An intermediate table is generated for an association and has two FK relationships pointing to the tables at association ends. FK are generated in the same way as for one-to-many associations.

The **Autogenerated table name template** transformation property governs the name of the generated intermediate table ($%t1$ and $%t2$ are replaced by the names of the tables at relationship ends).

### Conversion of Identifying Associations

Some relationships in the source model are treated as identifying relationships. In case of identifying a relationship, the objects of the class, which is at the multiple end of the association, are not independent, that is, they can exist only in association with the objects at the singular end of the association. In the resulting SQL model, the FK of these relationships is included into the PK of the table.

The PK of the dependent table is composite and contains two columns as a result:

1. ID column of the table
2. FK to the independent table

Unfortunately UML models lack model data and notation to specify, which associations are identified. Hence transformation has to guess this. It uses the following heuristics - the composite associations are treated as identifying, while the other associations are not.

The **Treat composition relationships as identifying** transformation property governs these heuristics. If this property set to false, all associations are treated as not identifying.

### Conversion of Multivalued Properties

In UML models, properties can be multi-valued (e.g. [0..7], [2..*]). However in databases columns they can be only single-valued. Transformation uses two strategies to handle multi-valued properties in the source model.

If the upper multiplicity limit is small and fixed, e.g., [0..3], then columns are simply multiplied the necessary number of times. The result table will have multiple columns with sequence numbers appended to their names (like "phone1", "phone2", and "phone3" columns in the result for a single phone[0..3] property in the source).

The **Max Duplicated Columns** transformation property governs the maximum number of columns, that are generated using this strategy.

If the upper multiplicity upper bound is larger than this limit or unlimited, then an auxiliary value table is generated for such multi-valued properties. This table is FK-related to the main table of the class, and holds a “value” column for storing property values.

The **Value table name** transformation property governs the name of the generated table ($%t$ in this property is replaced by the name of the table and $%r$ - by the property name). So, the table name template “$%t_%r_VALUES$” gives a “Person_Phone_VALUES” table name for the Person::phone property).
Conversion of Generalizations

In UML, generalizations are used extensively, while SQL domain lacks the concept of generalizations. Hence during the transformation, generalization trees are transformed into different concepts to simulate the generalization approximately.

There are three different strategies for simulating generalizations in the result domain:

1. **Multiple Tables, Decomposed Object** strategy.
2. **Multiple Tables, Copy Down** strategy.
3. **One Table, Merged** strategy.

Specify the strategy for converting generalization trees in the Generalization Tree transformation strategy transformation property.

**Multiple Tables, Decomposed Object strategy**

This strategy consists of the following actions:

1. Each class is converted to a separate table.
2. Direct (not inherited) properties of the class are converted to the columns of the table.
3. A foreign key to the table of the base class is created. The table of the base class carries the inherited columns.
4. Primary keys of all the classes participating in a hierarchy tree are the same (there can be several hierarchy trees in the same transformation source, and each one is handled separately). PK of the specific tables is also a FK to the parent table.

This strategy is the closest one to UML and fits nicely from theoretical standpoint since there is no data duplication. The only problem of this approach is the performance of data retrieval and storage. During the storing operation, objects are decomposed into several parts, each stored in a different table (that is why the strategy is called Decomposed Object strategy), and for retrieving the object you have to query several tables (with resulting multi-level joins).

**Multiple Tables, Copy Down strategy**

This strategy consists of the following actions:

1. Each class is converted to a separate table.
2. The table of each class holds columns for properties of that class AND all the columns, copied from the base class (that is why this strategy is called Copy Down strategy).

As a result each table possesses the complete column set to carry data about an object of particular type. All the data of the object is stored in one table.

The weak point of this strategy is that the association relationships between tables are copied down also. Hence each association in the source can produce many foreign keys in the target. Writing SQL queries against this database layout is not very convenient. Also, if you want to retrieve all the objects of the particular class, you have to query several tables and union the results.

**One Table, Merged strategy**

This strategy consists of the following actions:

1. All the classes in the generalization hierarchy are folded into one large table.
2. All the properties of all the classes become table columns (note that columns that were mandatory in the specific classes become optional in the merged table).
3. A separate selector column is generated, which indicates the type of the object carried by the particular line.
The **Selector Column Name**, **Selector Column Type** and **Selector Column Type Modifier** transformation properties determine the selector column format.

This strategy is suitable for very small hierarchies usually of just one hierarchy level with a couple of specialization classes, each adding a small number of properties to the base class. E.g., general class "VehicleRegistration" and a couple of subclasses: “CarRegistration” and “TruckRegistration”.

This strategy suites simple cases well. It is simple and fast. However it does not scale for larger hierarchies and produces sparse tables (tables with many null values in the unused columns) in this case.

**Conclusions and future improvements**

Note that all hierarchies from the transformation source are converted using the same method. You cannot choose different strategies for each particular case of the generalization tree. This is considered as a future improvement for the transformations.

**Package Hierarchy Reorganization**

UML models usually have a moderately deep package nesting organization, while SQL models can have at most one package level - the schemas. Hence during the transformation, packages should be reorganized.

The **Change package hierarchy** transformation property governs the package reorganization. Possible choices for setting the property value are as follows:

1. **Leave intact** choice switches reorganization off.
2. **Flatten packages** choice forces flattening of the packages of the source, leaving only the top level packages in the destination.
3. **Strip packages** choice removes all packages of the source.

**Sequence Autogeneration**

There is a note indicating:

| NOTE | This feature is only available in UML to SQL(Oracle) transformations. Generic SQL models do not have sequence support yet. |

For each single-column PK in the destination a sequence object can be generated.

The **Autogenerate Sequences** transformation property governs the sequence autogeneration. Possible choices for setting the property value are as follows:

1. **Do not generate sequences** choice switches sequence generation off.
2. **Generate sequences for all single-column PKs** choice switches sequence generation on.
3. **Generate sequences for all autogenerated PKs** choice switches sequence generation on but only for those PKs that there automatically generated by the tool (but not for PKs which were declared by the user).

**Type Mapping**

Default type maps of the UML to SQL(Generic / Oracle) transformations remap the types that are commonly used in the UML modeling domain (such as String) into types that are suitable for SQL domain (such as varchar).

**UML to SQL Type Map**

The default type map for these transformations is stored in the **UML to SQL Type Map** profile and automatically gets attached to your project at the 3rd step (the type mapping step) of the transformation wizard. If necessary it can be changed.
The Default map carries the following type conversions.

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Result Type</th>
</tr>
</thead>
</table>
| String      | varchar (default)  
               | varchar2  
               | char varying  
               | character varying  
               | nvarchar  
               | nvarchar2  
               | nchar varying  
               | national char varying  
               | national character varying  
               | longvarchar  
               | long varchar  
               | char  
               | character  
               | nchar  
               | national char  
               | national character  
               | long char |
| short       | smallint |
| long        | number(20) |
| Integer     | integer |
| int         | int |
| float       | float |
| double      | double precision |
| date        | date |
| char        | char(default)  
               | character  
               | nchar  
               | national char  
               | national character |
| byte        | number(3) |
| boolean     | number(1) |
Transformation Properties

This is the complete list of properties available in UML to SQL(Generic / Oracle) transformation in the Model Transformation Wizard (for more information about this wizard, see "Model Transformation Wizard" in MagicDraw UserManual.pdf).

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autogenerated PK name template</td>
<td>If the class has no PK column in the ER model, this transformation parameter for the autogenerated column name will generate the PK. You may specify the pattern for the PK name. Default &quot;id_%t&quot;, where %t is replaced by the name of the table.</td>
</tr>
<tr>
<td>Autogenerated PK type template</td>
<td>Specifies the type of the autogenerated PKs. Default: integer.</td>
</tr>
<tr>
<td>Autogenerated FK name template</td>
<td>The foreign keys are automatically generated to implement the relationships between classes. This transformation parameter autogenerates a FK name. You may specify the pattern for the name. Default: &quot;fk_%t%k%r&quot;, where %t is replaced by the name of the table, the foreign key is pointing. The %k is replaced by the key name, to which this foreign key points. The %r is replaced by the name of the relationship, which is realized with this foreign key.</td>
</tr>
<tr>
<td>Property name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Autogenerated table name template</td>
<td>This transformation parameter autogenerates table name. You may specify the pattern for the name. Default “%t1_%t2”, where %t1 is replaced by the name of the first table, %t2 - second table. The %r pattern (name of relationship) is also supported.</td>
</tr>
<tr>
<td>Generated index for alternative keys</td>
<td>If “true”, generates index for «AK».</td>
</tr>
<tr>
<td>Index name template</td>
<td>If the above option is set to “true”, you may choose the template for the index name. Template may contain %g pattern, which will be replaced with AK group name. Default: indexof_%g</td>
</tr>
<tr>
<td>Change package hierarchy</td>
<td>Choose option for packages from transformation source: to strip all the package hierarchy, or flatten the package hierarchy down to the first level where each package is transformed into the schema. Default: Flatten packages</td>
</tr>
<tr>
<td>Treat composition relationship as identifying</td>
<td>If this option is set to “true”, the composition associations are treated as if the «identifying» stereotype were applied to them. Default: Flatten packages</td>
</tr>
<tr>
<td>Default association end multiplicity</td>
<td>If multiplicity was not specified in model, defined multiplicity will be set after transformation. Default: 1</td>
</tr>
<tr>
<td>Generalization Tree transformation strategy</td>
<td>Selects the strategy to be used for converting generalization trees. Default: Multiple Values, Decomposed Object</td>
</tr>
<tr>
<td>Selector Column Name</td>
<td>Name of selector column for the merged table strategy of generalization conversion. Default: typeSelector. Note: together with selector type and type modifier this gives typeSelector:char(255) column.</td>
</tr>
<tr>
<td>Selector Column Type</td>
<td>Type of the selector column for the merged table strategy of generalization conversion. Default: char</td>
</tr>
<tr>
<td>Selector Column Type Modifier</td>
<td>Type modifier of the selector column for the merged table strategy of generalization conversion. Default: 255</td>
</tr>
<tr>
<td>Max Duplicated Columns</td>
<td>Threshold for multivalue property conversion strategies - maximum number of columns for which the column duplication strategy is used. If exceeded, auxiliary value table is used. Default: 3</td>
</tr>
<tr>
<td>Value Table Name</td>
<td>Name of the value table (to be generated when converting multivalued properties). %t pattern is expanded to table name, %r - name of the original property. Default: %t_%r_VALUES (e.g Person_phone_VALUES table will be generated for Person::phone[0..5] property in the source)</td>
</tr>
<tr>
<td>Autogenerate Sequences*</td>
<td>Selects wherever and when sequences are generated for PKs. Default: Generate sequences for all single-column PKs</td>
</tr>
</tbody>
</table>
ER to SQL (Generic / Oracle) Transformations

Both generic and Oracle transformation flavors are very similar, so they will be described together. Furthermore, these transformations are based on and very similar to the UML to SQL (Generic / Oracle) transformations with several extensions, relevant to ER modeling.

Hence this chapter only describes this extended behavior of ER to SQL (Generic / Oracle) transformation. To see the full transformation feature set (which includes conversion of many-to-many relationships into an intermediate table, three different methods of transforming generalizations into table layouts, autogenerating primary keys, unique constraints and indexes, generating additional tables for multivalue attributes, type remapping between UML and database worlds, sequence generation, and package hierarchy flattening), please, see “UML to SQL Transformation” on page 61.

NOTE
Please note that the SQL model, produced by the transformation, is usually not optimal (e.g. all the generalizations are transformed using the same chosen strategy, while usually different strategies are chosen for each particular case - at the discretion of DBA). Hence it is frequently advisable to refine / edit the produced model after the transformation.

Identifying Relationships

Identifying relationships are transformed in the same way as in the UML to SQL transformation, that is, the foreign key of the transformation gets to be included into the primary key of the dependent entity (the one at the multiple end of the relationship). The difference in ER to SQL transformation case is that the ER model eliminates guessing, which relationships are identifying and which ones are not. UML to SQL transformation guesses, which UML associations should be identifying, by using a heuristic method - composition associations are treated as identifying (this heuristic is controlled by the Treat compositions as identifying transformation property). In ER models, identifying relationships are explicitly marked as such, hence there is no need to guess (“Identifying Relationships and Dependent Entities” on page 9 specifies how identifying relationships are modeled).

Key Transformation

Keys in ER models are transformed into constraints in a DDL model.

These are the rules for key transformations into DDL constraints:

1. The Primary key of the entity in the ER model is transformed into a primary key constraint in the SQL model.
2. The Alternative keys of the entities in the ER model are transformed into unique constraints in the SQL model.
3. The Inversion entries of the entities in the ER model are transformed into indexes in the SQL model.

Autogenerated Sequence Name*

Name of the generated sequences. %t pattern is expanded to table name.
Default: %t_SEQ

* - These properties are available only for UML to SQL (Oracle) transformation.
4. If key or entry in ER model has a name (identifier tag), this information is preserved in the SQL model. The corresponding key / index will also have a name in the SQL model.

Lets review an example of key modeling, which has been described in “Key Modeling” on page 14. After the transformation, the three entities of the ER model are transformed into the three tables of the SQL model respectively.

![Table: Person](image)

```sql
PK <ssn> varchar
- name : varchar
- surname : varchar
+ unique{(columns = name, surname)}
```

![Table: ShippingAddress](image)

```sql
PK <id> integer
- country : varchar
- city : varchar
- street : varchar
- nr : varchar
- postalCode : varchar
+ unique{add()}{columns = country, city, street, ...}
+ unique{post()}{columns = country, postalCode}
```

![Table: inventoryPartType](image)

```sql
PK <code> varchar
- name : varchar
+ indexof{('code', 'name')}
```

Figure 34 -- Example of key transformation results

**Virtual Entity Transformation**

Virtual entities of ER models can be transformed into different elements of SQL models:

- Tables (just as ordinary, non-virtual entities).
- SQL views (ER to SQL(Oracle) transformation has an additional choice of simple views or materialized views).

The choice is controlled by the Virtual Entities Transformed To transformation property.

**Tracing between Data Model Layers**

After the transformation, a relationship is established between the logical data model layer, which is represented by the ER model, and the physical data model layer, which is represented by a SQL model respectively. It is possible to navigate between the connected elements in the forward (ER -> SQL) and backward (SQL -> ER) directions using the dedicated submenu - Go To - on the element’s shortcut menu.

To go to the corresponding element in the forward direction

1. Right-click the element.
To go to the corresponding element in the backward direction

1. Right-click the element.
2. On its shortcut menu, click Go To > Traceability > Model Transformations > Transformed From.

The same tracing information is visible in the element’s Specification window and Properties panel under the Traceability tab. This information is also reflected in the Entity-Relationship and SQL Report using navigable references between the report section. Traceability information can also be depicted in a relation map or in a tabular format using the capabilities of the custom dependency matrix feature.

**SQL to UML Transformation**

The SQL models and diagrams will be transformed into the platform-independent UML models and UML class diagrams. SQL to UML transformation can be applied to SQL models of any database flavor.

**Type Mapping**

If there are types specified in the SQL model for elements, after transformation SQL types should be converted to UML types. Because of that, there is a type mapping from SQL types to UML types.

Mapping rules are based on dependencies, which contains the SQL to UML Type Map profile. This profile is automatically attached, when SQL to UML transformation is performed.

**Transformation Results**

The SQL stereotypes are discarded from tables, views, fields, and associations (except the PK stereotype).

Views are discarded in transformed class diagram.
There are additional properties to choose for SQL to UML transformation in the Model Transformation Wizard (for more information about the wizard, see “Model Transformations Wizard” in MagicDraw UserManual.pdf.)

**Figure 35 -- Model Transformation Wizard for SQL to UML transformation. Specify Transformation Details wizard step**

<table>
<thead>
<tr>
<th>Option name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use PK</td>
<td>Check box</td>
<td>If set to “true”, appropriate columns with the primary key stereotype are marked after transformation.</td>
</tr>
<tr>
<td>Use IE</td>
<td>Check box</td>
<td>If set to “true”, indexed columns with the inverted entity stereotype are marked after transformation.</td>
</tr>
<tr>
<td>Use AK</td>
<td>Check Box</td>
<td>If set to “true”, unique columns with the alternative key stereotype are marked after transformation.</td>
</tr>
</tbody>
</table>

The «IE» stereotype is applied to the columns in the UML model from indexes in the SQL.

The «AK» stereotypes are applied to the columns in the UML model from unique constraints in the SQL.

If the unique or index of the SQL contains more than one column, the group tag is created on the corresponding columns. The value of the tag is the name of the unique / index.

If the PK, unique constraint or index of the SQL contains more than one column, the orderInXXGroup tag is created on the corresponding columns. The value of the tag is the place number of the column in the PK, unique constraint or index (first column gets tag value=1, second column - 2, etc).

Before transformation:

```
<table>
  <tr>
    <td>Person</td>
  </tr>
  <tr>
    <td>ID</td>
  </tr>
  <tr>
    <td>Name</td>
  </tr>
  <tr>
    <td>Bank account</td>
  </tr>
  <tr>
    <td><unique>+salary(ID, Name)</td>
  </tr>
</table>
```
After transformation:

There are some foreign key cases, when after transformation, association with multiplicities are created in class diagram:

**Transforming foreign key, when the «unique» stereotype is set**

**Before transformation:**

- Person
  - ID
  - Name
  - Account
- Account
  - Number
  - Currency

**After transformation:**

**Transforming foreign key, when the «not null» stereotype is set**

**Before transformation:**

- Person
  - ID
  - Name
  - Account
- Account
  - Number
  - Currency

**After transformation:**
Transforming foreign key, when the «null» stereotype is set

Before transformation:

![Diagram showing UML to XML Schema transformation before «null» stereotype]

After transformation:

![Diagram showing UML to XML Schema transformation after «null» stereotype]

Transforming foreign key, when the «unique» and the «not null» stereotypes are set

Before transformation:

![Diagram showing UML to XML Schema transformation before «unique» and «not null» stereotypes]

After transformation:

![Diagram showing UML to XML Schema transformation after «unique» and «not null» stereotypes]

**UML to XML Schema Transformation**

The UML to XML Schema transformation helps to create the equivalent XML schema model from the given UML model.

Basically this transformation is the copying of a source UML model, and then applying the necessary stereotypes according to the XML schema modeling rules.
Type Mapping

This type map stores mapping between primitive UML data types and primitive XML Schema data types.

Figure 36 -- UML to XML schema type map (1)
Transformation Results

For each class in the transformation destination set, the «XSDcomplexType» stereotype is applied, unless this class is derived from the simple XML type (that is, one of the basic types, or type, stereotyped with XSDsimpleType). In that case a «XSDsimpleType» stereotype is applied.

If the class is derived from another class, which is stereotyped as «XSDcomplexType», additionally the «XSDcomplexContent» stereotype is applied on this class with «XSDextension» on the corresponding generalization relationship.

If the class is derived from another class, which is stereotyped as «XSDsimpleType», additionally the «XSDrestriction» stereotype is applied on the corresponding generalization relationship.

If the class is not derived from anything, and has attributes with the XSDelement tag, the «XSDcomplexContent» stereotype is applied on this class.

If the class is not derived from anything, and has no attributes with the XSDelement tag, no «XXXXContent» stereotype is applied on this class - the class has an empty content.

The UML datatypes in the transformation source set are transformed into the classes with the «XSDsimpleType» stereotype - unless after the type map this class appears to be derived from a class with the «XSDcomplexType» stereotype. Then the «XSDcomplexType» stereotype is used.

For each attribute of the class, which is NOT of the simple XML type (that is, one of the basic types, or type, stereotyped with the «XSDsimpleType») or has a multiplicity > 1, the «XSDelement» stereotype is applied.

For each composition association, linking 2 classes stereotyped as XML schema types, the stereotype on the association end is applied, the same as the rules for attributes.
Enumerations in the UML model are transformed into the enumerations in the XML Schema model (classes with the «XSDsimpleType» stereotype are derived by restriction from the XML string type, where all the elements of the original enumeration are converted into the attributes with an «XSDenumeration» stereotype).

For each package in the transformation set, the «XSDnamespace» stereotype is applied.

In each package, one additional class for the XML schema is created. The name of the schema class is constructed by taking the name of the package and then appending the .xsd to it (e.g. if the package in the source model set is named "user", then name the schema class "user.xsd" in the destination package).

The targetNamespace value is added to the schema class, with the name of it's parent (e.g. if the schema is placed in the "http://magicdraw.com/User" package, the targetNamespace=" http://magicdraw.com/User" is set on the schema class).

Schema class and the namespaces http://www.w3c.org/2001/XMLSchema [XML Schema profile] and its target namespace are linked using the xmlns relationships. The names of these links are: the same as target namespace, for the link to target namespace; "xs" for the XML Schema namespace.

Class diagrams are transformed into XML Schema diagrams.

The model elements, which have no meaning in the XML schemas, are discarded. This includes (without limitation) behavioral features of classes, interfaces, actors, use cases, states, activities, objects, messages, stereotypes and tag definitions.

There are additional properties to choose for UML to XML Schema transformation in the ModelTransformation Wizard (for more information about the wizard, see "Model Transformations" in MagicDraw UserManual.pdf.)

![Model Transformation Wizard](image.png)

Figure 38 -- Model Transformation Wizard for UML to XML Schema transformation. Specify Transformation Details
XML Schema to UML Transformation

The XML Schema to UML transformation helps to extract the abstract UML model from the XML schema model.

Type Mapping

Type maps store mapping between primitive UML data types and primitive XML Schema data types, the same applies for UML to XML Schema Transformation just in reversed order. For XML Schema to UML element type mapping diagram, see “Type Mapping” on page 74.

Transformation Results

The XML Schema diagrams are transformed to the Class diagrams.

Unnecessary stereotypes (XSDxxxx) are discarded from the classes.

Attributes of the classes are gathered if they were spread into several different classes.

Attributes of the classes may be realized as associations. In this case the main class gathers all the associations of the members.

The same principle is applied when elements are in a group, shared by two or more classes. Elements (attributes) are copied into both destination classes.

The attributes with the «XSDgroupRef» stereotype are treated as if the group relationship has been drawn and transformed accordingly - discarded in the UML model, and the group content (elements / attributes) placed in their place.

Simple XML schema types (classes with the «XSDsimpleType» stereotype), which after copying and type remap happen to be derived from any data type (UML DataType) or not derived from anything and are transformed into the UML data types.

Simple XML schema types, which are derived by restriction from string and are restricted by enumerating string values and are converted into enumerations in the UML diagrams.

The classes with the «XSDschema» stereotype are not copied into a destination model.

The «XDSkey», «XSDkeyref», and «XSDunique» stereotyped attributes are not copied into a destination model.
The «XDSany» , «XSDanyAttribute» stereotyped attributes are not copied into a destination model.

The «XDSnotation» stereotyped attributes are not copied into a destination model.

The «XDSlength» , «XDSminLength» , «XDSmaxLength» , «XSDpattern» , «XSDfractionDigits» , «XSDtotalDigits» , «XSmaxExclusive» , «XSmaxInclusive» , «XSminExclusive» , and «XSminInclusive» stereotyped attributes are not copied into a destination model.

The XML schemas (classes with the «XSDschema» stereotype) should not be transformed, but they may contain inner classes (anonymous types of schema elements). These inner classes are transformed using usual rules for UML type transformation - as if they were not inner classes but normal XML schema types.
MagicDraw provides a report template for generating reports of the data models. The report template is suitable for reporting both ER and SQL models. If your project contains both ER and SQL models, a unified report covering both models can be produced.

The report can be generated using the Report Wizard feature.

To generate a report

2. In the Select Template area, select Data Modeling > Entity-Relationship and DDL Report and then click Next >.
3. Click Next > again.
4. In the Select Element Scope area, define the scope for the report, using the buttons placed between the two lists, and then click Next >.
5. In the Output Options area, define the appropriate options.
6. Click Generate. Wait a moment while the report is generated (generation time depends on the selected scope).

The Report Wizard produces an .rtf file. This file contains sections for each reported model entity, its attributes, relationships with other entities (both simple relationships and generalization / specialization relationships), and keys. The SQL part of the file contains sections for each table (with its columns, constraints, indexes, and triggers), each standalone sequence, each global procedure or function, each user defined type (with its attributes and methods), and each authorization identifier (users, groups, roles, and permissions).

The report has a boilerplate beginning and includes a cover page, table of contents, and a table of figures. Sections such as “Purpose”, “Scope”, “Overview”, and “Revision History” can be customized by changing the predefined report variables (see the 3rd step of the report generation procedure, described above). The report also has an appendix containing all the diagrams in your model.
If the model contains both ER and SQL models and is linked by traceability references, the report will link (with active references) the appropriate report sections of entities and tables that are traceable in the model.

**Figure 39 -- Fragment of ER model report example**

<table>
<thead>
<tr>
<th>Entity Relationship Modeling</th>
<th>ER&amp;SQL Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date: March 26, 2010</td>
<td>Revision: 1.0</td>
</tr>
</tbody>
</table>

**Entity Purchase**

Transformed To: Purchase

**Attributes:**
- PONr: String
- quantity: Integer
- priceWithDiscount: Integer
- total: Integer
- year: Integer
- month: Month
- day: Integer

**Keys:**
- Primary (anonymous): PONr

**Relationships:**
- many(optional)-to-one(mandatory) Salesman (as NA): Identifying NA
- many(optional)-to-one(mandatory) Product (as NA): Identifying NA

Confidential <Your Company Name> 12
Table Virtual_Entities.Purchase

Transformed From: Purchase

Columns:
- PON: varchar
- quantity: integer
- priceWithDiscount: integer
- total: integer
- year: integer
- month: Month
- day: integer
- fk_Salesmanid: varchar NOT NULL
- fk_Productid: varchar NOT NULL

Keys/Constraints/Indexes/Triggers:
- Primary (anonymous): PON, fk_Salesmanid, fk_Productid
- Foreign (anonymous): to table Salesman, fk_Salesmanid=Salesman.id
- Foreign (anonymous)(1): to table Product, fk_Productid=Product.id
- year: integer
- fk_Salesmanid: varchar NOT NULL
- fk_Productid: varchar NOT NULL
- fk_Productid: varchar NOT NULL

Keys/Constraints/Indexes/Triggers:
- Primary (anonymous): PON, fk_Salesmanid, fk_Productid
- Foreign (anonymous): to table Salesman, fk_Salesmanid=Salesman.id
- Foreign (anonymous)(1): to table Product, fk_Productid=Product.id

Figure 40 -- Fragment of SQL model report example
XML SCHEMAS

Introduction

Reference: 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 on</th>
<th>Defined TagDefinitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSDcomponent</td>
<td>Class</td>
<td>Attribute, AttributeEnd, Binding, Generalization, Comment, Component</td>
<td>id – string&lt;br&gt;Details: The base and abstract stereotype for all XML Schema stereotypes used in UML profile</td>
</tr>
<tr>
<td>XSDAttribute</td>
<td>XSDComponent</td>
<td>Attribute</td>
<td>fixed – some fixed element value&lt;br&gt;form – *(qualified</td>
</tr>
<tr>
<td>XSElement</td>
<td>XSDComponent</td>
<td>Attribute, AttributeEnd</td>
<td>abstract – (true</td>
</tr>
<tr>
<td>XSDcomplexType</td>
<td>XSDComponent</td>
<td>Class</td>
<td>block – *(extension</td>
</tr>
<tr>
<td>XSDsimpleContent</td>
<td>Class</td>
<td></td>
<td>simpleContentId – string</td>
</tr>
<tr>
<td>XSDcomplexType</td>
<td>Class</td>
<td></td>
<td>complexContentId – string&lt;br&gt;complexContentMixed</td>
</tr>
<tr>
<td>XSDgroup</td>
<td>XSDComponent</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>Stereotype name</td>
<td>Base Stereotype</td>
<td>Applies on</td>
<td>Defined TagDefinitions</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>XSDgroupRef</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>sequenceOrder – a number in sequence order</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssociationEnd</td>
<td></td>
</tr>
<tr>
<td>XSDall</td>
<td>Class</td>
<td>allId – string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>maxOccurs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>minOccurs</td>
<td></td>
</tr>
<tr>
<td>XSDchoice</td>
<td>Class</td>
<td>choiceId – string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>maxOccurs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>minOccurs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sequenceOrder – a number in sequence order</td>
<td></td>
</tr>
<tr>
<td>XSDsequence</td>
<td>Class</td>
<td>sequenceld – string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>maxOccurs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>minOccurs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sequenceOrder – a number in sequence order</td>
<td></td>
</tr>
<tr>
<td>XSDrestriction</td>
<td>XSDcomponent</td>
<td>Generalization</td>
<td></td>
</tr>
<tr>
<td>XSDextension</td>
<td>XSDcomponent</td>
<td>Generalization</td>
<td></td>
</tr>
<tr>
<td>XSDAttributeGroup</td>
<td>XSDcomponent</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>XSDsimpleType</td>
<td>XSDcomponent</td>
<td>final - (#all</td>
<td>(list</td>
</tr>
<tr>
<td>XSDlist</td>
<td>XSDcomponent</td>
<td>Class</td>
<td>listId - string</td>
</tr>
<tr>
<td>XSDunion</td>
<td>XSDcomponent</td>
<td>Class</td>
<td>unionId - string</td>
</tr>
<tr>
<td>XSDannotation</td>
<td>XSDcomponent</td>
<td>Comment</td>
<td>appInfoSource</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>appInfoContent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xml:lang</td>
</tr>
<tr>
<td>XSDany</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>namespace – string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>processContents - (lax</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sequenceOrder – a number in sequence order</td>
</tr>
<tr>
<td>XSDanyAttribute</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>namespace – string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>processContents - (lax</td>
</tr>
<tr>
<td>XSDschema</td>
<td>XSDcomponent</td>
<td>Class</td>
<td>attributeFormDefault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>blockDefault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>elementFormDefault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>finalDefault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targetNamespace – reference to some ModelPackage version xml:lang</td>
</tr>
</tbody>
</table>
### XML SCHEMAS

**XML Schema Mapping to UML Elements**

<table>
<thead>
<tr>
<th>Stereotype name</th>
<th>Base Stereotype</th>
<th>Applies on</th>
<th>Defined TagDefinitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSDnotation</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>public system</td>
</tr>
<tr>
<td>XSDredefine</td>
<td>XSDcomponent</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>XSDimport</td>
<td>XSDcomponent</td>
<td>Permission</td>
<td>schemaLocation</td>
</tr>
<tr>
<td>XSDinclude</td>
<td>XSDcomponent</td>
<td>Component</td>
<td></td>
</tr>
<tr>
<td>XSDminExclusive</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDminInclusive</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDmaxExclusive</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDmaxInclusive</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDtotalDigits</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDfractionDigits</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDlength</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDminLength</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDmaxLength</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDwhiteSpace</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>fixed = boolean : false</td>
</tr>
<tr>
<td>XSDpattern</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>XSDenumeration</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>XSDunique</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>selector field</td>
</tr>
<tr>
<td>XSDkey</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>selector field</td>
</tr>
<tr>
<td>XSDkeyref</td>
<td>XSDcomponent</td>
<td>Attribute</td>
<td>selector field refer – UML Attribute referString - String</td>
</tr>
<tr>
<td>XSDnamespace</td>
<td>ModelPackage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmlns</td>
<td>Permission</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**attribute**

- 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
<xs:attribute
    default = "string"
    fixed = "string"
    form = "qualified" | "unqualified"
    id = "ID"
    name = "NCName"
    ref = "QName"
    type = "QName"
    use = "optional" | "prohibited" | "required"
>
    any attributes with non-schema namespace . . .
</xs: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 Schema Diagram](image-url)
XML SCHEMAS
XML Schema Mapping to UML Elements


   <xs:attribute name = "name" type = "xs:string" default = "minde"
   fixed = "fixed_value" form = "qualified" use = "optional" >
   <xs:annotation >
   <xs:documentation >name attribute
   </xs:annotation>  
   </xs:attribute>
   <xs:attribute name = "address" fixed = "fixed_value" form = "qualified" use = "optional" >
   <xs:annotation >
   <xs:documentation >surname attribute
   </xs:annotation>  
   </xs:attribute>
   <xs:simpleType >
     <xs:restriction base = "xs:string" />
   </xs:simpleType>
   <xs:attribute>
   <xs:attribute name = "surname" type = "xs:string" />
   <xs:attributeGroup name = "attr_group" >
     <xs:attribute ref = "nm:name" >
     <xs:annotation >
     <xs:documentation >reference
     </xs:annotation>
     </xs:attribute>
   </xs:attributeGroup>
   <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 SCHEMAS

XML Schema Mapping to UML Elements

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 | a
  (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>
```

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

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

<attributeGroup
  id = ID
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</attributeGroup>

<anyAttribute

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 `<complexType>`) 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.
<complexType
  id = ID
  mixed = boolean
  {any attributes with non-schema namespace . . .}>
  Content: {annotation?, (restriction | extension)}
</complexType>
<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
<?xml version='1.0' encoding='Cp1252'?>
  <xs:complexType name = "my_Type2" block = "extension" final = "extension" mixed = "true">
    <xs:annotation >
      <xs:documentation >my_Type2
    </xs:documentation>
  </xs:annotation>
</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.

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:

```
  <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.

Example:

```xml
<xs:simpleType name="farenheitWaterTemp">
  <xs:restriction base="xs:number">
    <xs:frac
ti
onDigits value="2"/>
    <xs:min\nEx
clusive value="0.00"/>
    <xs:max\nEx
clusive 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 XSDrestriction 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 XSDlist.
Binding between this class and XSD:list must be provided.

“itemsType” maps to UML TemplateArgument from Binding.

union

UML Class must have additional stereotype XSDunion.

“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.
XML SCHEMAS
XML Schema Mapping to UML Elements

Example:

Restriction example

```
  <xs:simpleType name="fahrenheitWaterTemp">
    <xs:documentation>documentation of simple type</xs:documentation>
    <xs:restriction base="xs:integer">
      <xs:documentation>documentation of restriction</xs:documentation>
      <xs:pattern id="pattern_id" value="[0-9]{5}|[0-9]{4}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```
XML SCHEMAS

XML Schema Mapping to UML Elements
<xs:annotation>
  <xs:documentation>min inclusive documentation</xs:documentation>
</xs:annotation>
</xs:simpleType>
</xs:restriction>
</xs:simpleType>
</xs:documentation>
<xs:annotation>
  <xs:documentation>max exclusive documentation</xs:documentation>
</xs:annotation>
</xs:simpleType>
</xs:annotation>
</xs:documentation>
<xs:annotation>
  <xs:documentation>max exclusive documentation</xs:documentation>
</xs:annotation>
</xs:simpleType>
</xs:annotation>
</xs:documentation>
<xs:annotation>
  <xs:documentation>min exclusive documentation</xs:documentation>
</xs:annotation>
</xs:documentation>
<xs:annotation>
  <xs:documentation>min exclusive documentation</xs:documentation>
</xs:annotation>
</xs:simpleType>
</xs:simpleType>
</xs:documentation>
<xs:annotation>
  <xs:documentation>day time documentation</xs:documentation>
</xs:annotation>
</xs:simpleType>
</xs:annotation>
</xs:documentation>
<xs:enumeration>
  <xs:enumeration value = 'day'
  <xs:documentation>rn restriction base = "xs:day" />
</xs:enumeration>
</xs:simpleType>
</xs:annotation>
</xs:documentation>
<xs:enumeration>
  <xs:enumeration value = 'night'
  <xs:documentation>rn restriction base = "xs:day" />
</xs:enumeration>
</xs:simpleType>
</xs:annotation>
</xs:enumeration>
</xs:annota...
list example

```xml
<?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:restriction base='xs:string' />
    </xs:simpleType>
  </xs:list>
  <xs:simpleType name='my_number_list1'>
    <xs:annotation>
      <xs:documentation>my list</xs:documentation>
    </xs:annotation>
  </xs:simpleType>
  <xs:list itemType='xs:boolean' />
</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 = "unionID2" 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 XSDminExclusive. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

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

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

XML Schema Mapping to UML Elements

Example:

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

```xml
<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’).

**maxExclusive**

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

- value – to Attribute initial value.

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

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

**minInclusive**

Maps to UML Attribute with stereotype XSDminInclusive. Name and type of such attribute does not make sense.
XML SCHEMAS
XML Schema Mapping to UML Elements

- value – to Attribute initial value.

**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>
```

**maxInclusive**

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

- value – to Attribute initial value.

**Example:**

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

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

**totalDigits**

Maps to UML Attribute with stereotype `XSDtotalDigits`. Name and type of such attribute does not make sense.
XML SCHEMAS

XML Schema Mapping to UML Elements

- value – to Attribute initial value.

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

```
<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 min/max values as before).

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

**fractionDigits**

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

- value – to Attribute initial value.

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

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

**Example**
XML SCHEMAS

XML Schema Mapping to UML Elements

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:

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

**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**

```xml
<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.

```xml
<simpleType name='productCode'>
  <restriction base='string'>
    <length value='8' fixed='true'/>
  </restriction>
</simpleType>
```

**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**

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

**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).

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

**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>
```

**whiteSpace**

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

- value – to Attribute initial value.
XML Representaion 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>
```

**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 Representaion Summary: **pattern** Element Information Item

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

**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`.

```xml
<simpleType name='better-us-zipcode'>
  <restriction base='string'>
    <pattern value='[0-9]{5}(-[0-9]{4})?'/>
  </restriction>
</simpleType>
```

**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 . . .}>
  {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='197-01-01'>
      <annotation>
        <documentation>New Year's day</documentation>
      </annotation>
    </enumeration>
    <enumeration value='197-07-04'>
      <annotation>
        <documentation>4th of July</documentation>
      </annotation>
    </enumeration>
    <enumeration value='197-12-25'>
      <annotation>
        <documentation>Christmas</documentation>
      </annotation>
    </enumeration>
  </restriction>
</simpleType>
```

`unique`

Maps to UML Attribute added into some UML Class.

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

unique UML Model example

For an example, see “keyref UML Model example” on page 111
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 “keyref UML Model example” on page 111

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>
```
XML SCHEMAS

XML Schema Mapping to UML Elements

keyref UML Model example

```xml
  <xs:element name = "personType" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "personRef" type = "ns:personRef" />
      </xs:sequence >
    </xs:complexType >
  </xs:element >

  <xs:element name = "vehicleType" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "vehicleRef" type = "ns:vehicleRef" />
      </xs:sequence >
    </xs:complexType >
  </xs:element >

  <xs:element name = "carType" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "carRef" type = "ns:carRef" />
      </xs:sequence >
    </xs:complexType >
  </xs:element >

  <xs:element name = "stateType" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "stateRef" type = "ns:stateRef" />
      </xs:sequence >
    </xs:complexType >
  </xs:element >

  <xs:element name = "vehicleRef" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "plateNumber" type = "xs:integer" />
        <xs:element name = "regState" type = "ns:twoLetterCode" />
      </xs:sequence >
    </xs:complexType >
  </xs:element >

  <xs:element name = "personRef" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "regPlate" type = "xs:integer" />
        <xs:element name = "regState" type = "ns:twoLetterCode" />
      </xs:sequence >
    </xs:complexType >
  </xs:element >

  <xs:element name = "stateRef" >
    <xs:complexType >
      <xs:sequence >
        <xs:element name = "regPlate" type = "xs:integer" />
        <xs:element name = "regState" type = "ns:twoLetterCode" />
      </xs:sequence >
    </xs:complexType >
  </xs:element >

  <xs:element name = "twoLetterCode" >
    <xs:simpleType >
      <xs:restriction base = "xs:string" >
        <xs:pattern value = "[A-Z]{2}" />
      </xs:restriction >
    </xs:simpleType >
  </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:keyref>
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 reapplied to tagged value according to annotation rule (see: annotation). For field values annotation documentation shall be merged in one.

Example
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:element>

<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:element>

<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 code s are a key for them within the document. Vehicle's plateNumber s 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 “keyref UML Model example” on page 111

annotation

Maps to UML Comment with or without stereotype XSDannotation.

Documentation's content maps to UML Comment body(name).

“documentation” maps as UML comment:

- “content” value shall be comment name
- “xml:lang” value – tag “xml:lang” value
- source value – tag “source” value

“appinfo” maps as tag value with name “appInfoSource”:

- “source” value will be tag value
- “content” will be documentation for tagged value

Appearing several annotation nodes on one element node, mapping shall be done in following way:

- “documentation” text shall be merged into one UML comment with merged content, but “content” and “xml:lang” tag values shall represent only first matched values
• “appInfo” shall have: “content” merged into one tag “appInfoSource” comment, but tag value shall represent first matched “appinfo”

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

```xml
<annotation
    id = "ID"
    {any attributes with non-schema namespace . . .}>
    Content: (appinfo | documentation)*
</annotation>
<appinfo
    source = anyURI>
    Content: {any}*
</appinfo>
<documentation
    source = anyURI
    xml:lang = Language>
    Content: {any}*
</documentation>
```

**Example**

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

XML representations of three kinds of annotation.

annotation UML Model example

```xml
<xsd:schema>
    schema
    {targetNamespace=http://nomagic.com, 
     xmlns:ns=http://nomagic.com}>
        <xs:annotation >
            <xs:appinfo source = "infoSource" >infoContent</xs:appinfo>
```
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

```
<xs:all>
  <xs:element ref="cats"/>
  <xs:element ref="dogs"/>
</xs:all>

<xs:sequence>
  <xs:choice>
    <xs:element ref="left"/>
    <xs:element ref="right"/>
  </xs:choice>
  <xs:element ref="landmark"/>
</xs:sequence>
```

XML representations for the three kinds of model group, the third nested inside the second.

compositors UML Model example
XML SCHEMAS
XML Schema Mapping to UML Elements

```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:annotation >
**XML SCHEMAS**

XML Schema Mapping to UML Elements

---

```xml
<xs:documentation>ref
documentation/<xs:documentation>
</xs:annotation>
</xs:group>
<xs:group ref="nm:myGroup2" minoccurs="0" maxoccurs="unbounded">
    <xs:annotation>
    
    </xs:documentation>ref another ref
documentation/<xs:documentation>
</xs:annotation>
</xs:group>
<xs:element name="name" type="xs:string"/>
<xs:element name="name2" type="xs:string"/>
<xs:element name="name1" type="xs:string"/>
<xs:any id="anyID" namespace="value" minoccurs="0" maxoccurs="1"/>
</xs:sequence>
</xs:group>
<xs:group name="myGroup3">
    <xs:choice>
        <xs:element name="address" type="xs:string"/>
    </xs:choice>
</xs:group>
<xs:group name="myGroup2">
    <xs:all>
        <xs:element name="surname" type="xs:string"/>
    </xs:all>
</xs:group>
</xs:group>
</xs:schema>

**group**

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:**

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

```xml
<xsd:group name="myModelGroup">
  <xsd:sequence>
    <xsd:element ref="someThing"/>
  </xsd:sequence>
</xsd:group>

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

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

group UML Model example

For an example, see “compositors UML Model example” on page 117

**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.
Example

```xml
<x:s:processContents="skip"/>
<x:s:namespace="#other" processContents="lax"/>
<x:s:namespace="http://www.w3.org/1999/XSL/Transform"/>
<x:s:namespace="#targetNamespace"/>
```

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

any and anyAttribute UML Model example

```
<<XSD:complexType>>
  <complexType name="any">
    <sequence>
      <element name="anyID" namespace="#http://bla.bla.bla" id="anyID" processContents="strict" minOccurs="0" maxOccurs="1"/>
    </sequence>
  </complexType>
<<XSD:complexType>>
  <complexType name="anyAttribute">
    <sequence>
      <element name="any" namespace="#http://bla"/>
    </sequence>
  </complexType>
```

```xml
<?xml version='1.0' encoding='Cp1252'?>
  <x:s:group name = "my_type" >
    <x:s:choice >
      <x:s:element name = "anyID" namespace = "http://bla" processContents = "strict" minOccurs = "0" maxOccurs = "1" >
        <x:s:annotation >
          <x:s:documentation >any
documentation</x:s:documentation>
        </x:s:annotation>
      </x:s:element>
    </x:s:choice>
  </x:s:group>
</x:s:schema>
```
XML SCHEMAS

XML Schema Mapping to UML Elements

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

```
<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

```
<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:mm = "http://nomagic.com"
xmlns:xs = 'http://www.w3.org/2001/XMLSchema"
xmns = 'http://nomagic.com"
attributeFormDefault = "qualified"
blockDefault = "extension"
elementFormDefault = "unqualified"
finalDefault = "extension"
targetNamespace = "http://nomagic.com"
version = "1.2"
xml:lang = 'EN" />
```

**notation**

Maps to UML Attribute with stereotype XSDnotation. This attribute must be added into UML class with stereotype XSDschema.

- name maps to UML Attribute name
- annotation maps to UML Attribute documentation.

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

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

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

The XML representation of a notation declaration.

notation UML Model example

```xml
<xs:schema xmlns:xm="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
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
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

```
<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:
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

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

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

include UML Model example

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

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.com/example">
  ...
</xsd: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 “schema UML Model example” on page 123.