MDA Transformations User Guide

Enterprise Architect is an intuitive, flexible and powerful UML analysis and design tool for building robust and maintainable software.

This booklet describes the Model Driven Architecture (MDA) Transformation facilities of Enterprise Architect.

Enterprise Architect - MDA Transformations User Guide

Printed: May 2010

Publisher

Sparx Systems

Managing Editor

Geoffrey Sparks

Technical Editors

Simon McNeilly

Special thanks to:

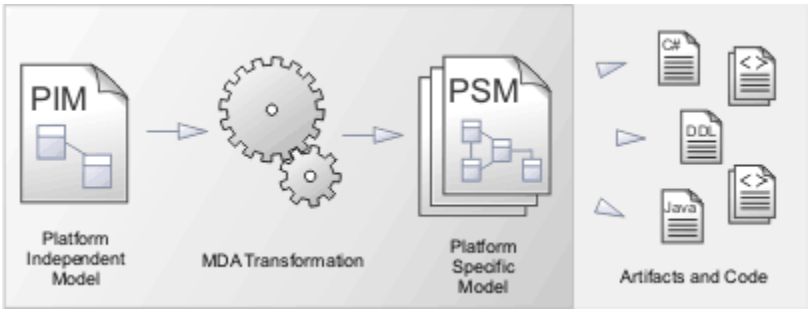
All the people who have contributed suggestions, examples, bug reports and assistance in the development of Enterprise Architect. The task of developing and maintaining this tool has been greatly enhanced by their contribution.

Table of Contents

Foreword

This user guide describes the Model Driven Architecture (MDA) Transformation facilities of Enterprise Architect.

MDA Transformations



Model Driven Architecture (MDA) Transformations provide a fully configurable way of converting model elements and model fragments from one domain to another. This typically involves converting Platform-Independent Model (PIM) elements to Platform-Specific Model (PSM) elements. A single element from the PIM can be responsible for creating multiple PSM elements across multiple domains.

Transformations are a huge productivity boost, and reduce the necessity of manually implementing stock Classes and elements for a particular implementation domain: for example, database tables generated from persistent PIM Classes. Enterprise Architect includes some basic built-in Transformations, such as PIM to Data Model, PIM to C#, PIM to Java and PIM to XSD. Sparx Systems will make further Transformations available over time, either as built in Transformations or as downloadable modules from the Sparx Systems website.

For a further productivity boost, Enterprise Architect can automatically generate code for your transformed Classes that target code languages. See the **Generate Code on result** 6 option on the Model Transformation dialog.

A Transformation is defined using Enterprise Architect's simple code generation template language, and involves no more than writing a template to create a simple intermediary source file. Enterprise Architect reads the source file and binds that to the new PSM.

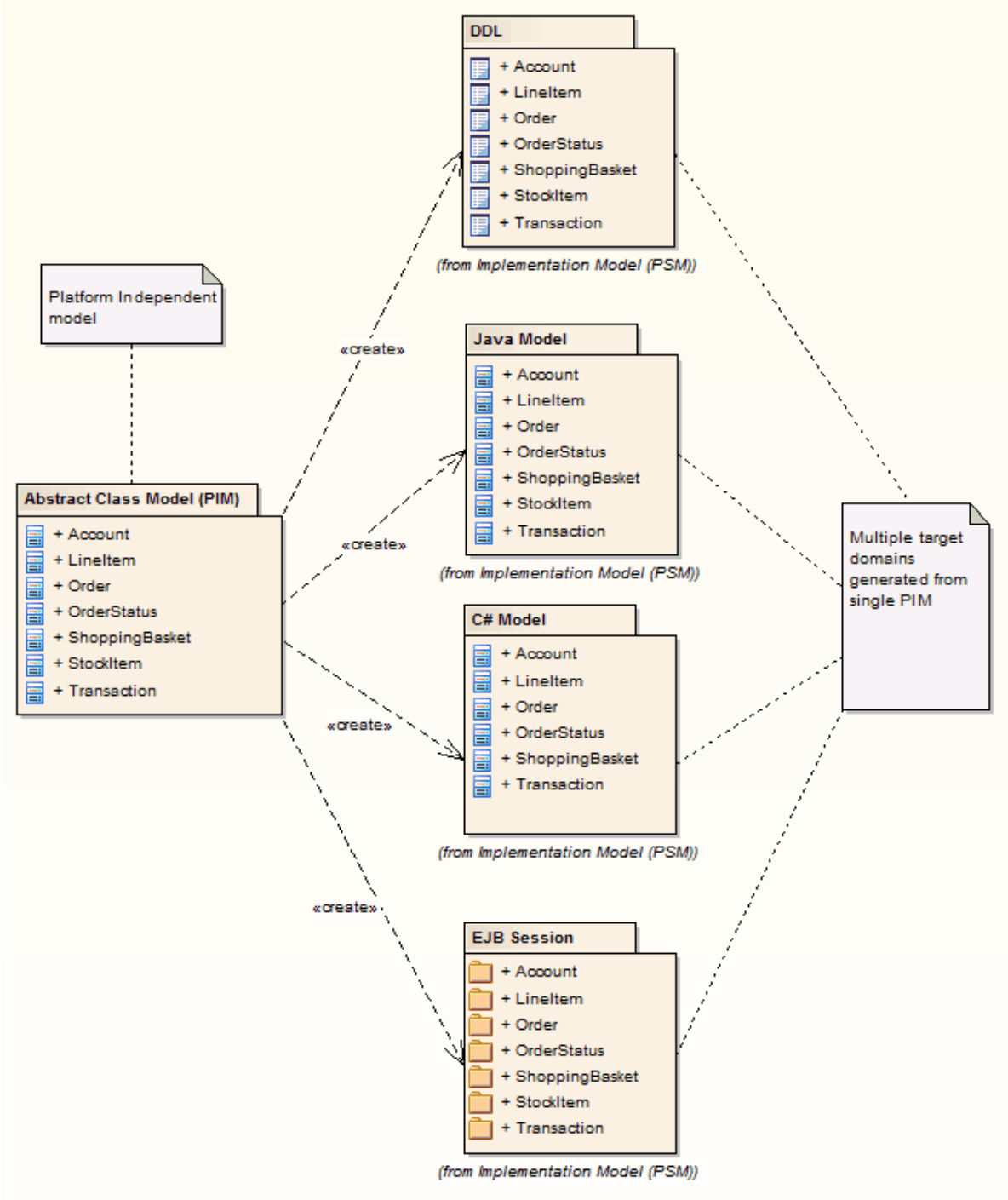
Enterprise Architect also creates internal bindings (*Transformation Dependencies*) between each PSM created and the original PIM. This is essential, as it enables you to forward synchronize from the PIM to the PSM many times, adding or deleting features as you go. For example, adding a new attribute to a PIM Class can be forward synchronized to a new column in the Data Model. You can observe the Transformation Dependencies for a package using the Traceability window. This enables you to check the impact of changes to a PIM element on the corresponding elements in each generated PSM, or to verify where a change required in a PSM should be initiated in the PIM (and also to reflect back in other PSMs). The Transformation Dependencies are a valuable tool in managing the traceability of your models (see *UML Model Management*).

Enterprise Architect does not delete or overwrite any element features that were not originally generated by the transform. Therefore, you can add new methods to your elements, and Enterprise Architect does not act on them during the forward generation process.

Note:

If you are using the Corporate, Business and Software Engineering, System Engineering or Ultimate edition, if security is enabled you must have **Transform Package** access permission to perform an MDA Transform on a package. For more information, see *User Security in UML Models.*

The following diagram highlights how Transformations work and how they can significantly boost your productivity:



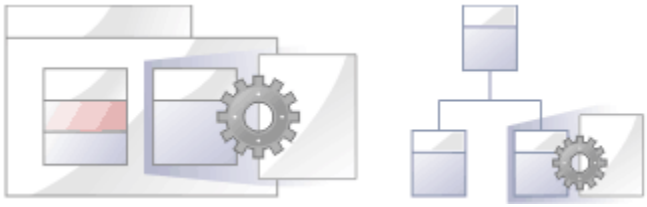
Transformations that are currently built-in include:

- **C# -** Converts a PIM to a standard C# implementation set
- **Data Model to ERD** - Transforms a Data Model to an Entity Relationship Diagram (ERD)
- **DDL -** Transforms platform-independent Class elements to platform-specific table elements
- **EJB Entity -** Transforms platform-independent Class elements to packages containing the Class and Interface elements that comprise an EJB Entity Bean
- **EJB Session -** Transforms platform-independent Class elements to packages containing the Class and Interface elements that comprise an EJB Session Bean
- **ERD to Data Model** - Transforms an Entity Relationship Diagram into a Data Model
- **Java -** Transforms platform-independent elements to Java language elements

- **JUnit** - Converts a Java model to a model where test methods are created for each public method of any original Class
- **NUnit -** Converts a .Net language specific model to a model where test methods are created for each public method of any original Class
- **WSDL -** Converts a simple representation of a WSDL interface into the elements required to generate that interface
- **XSD -** Transforms platform-independent elements to XSD elements.

Transformations are described in the following topics:

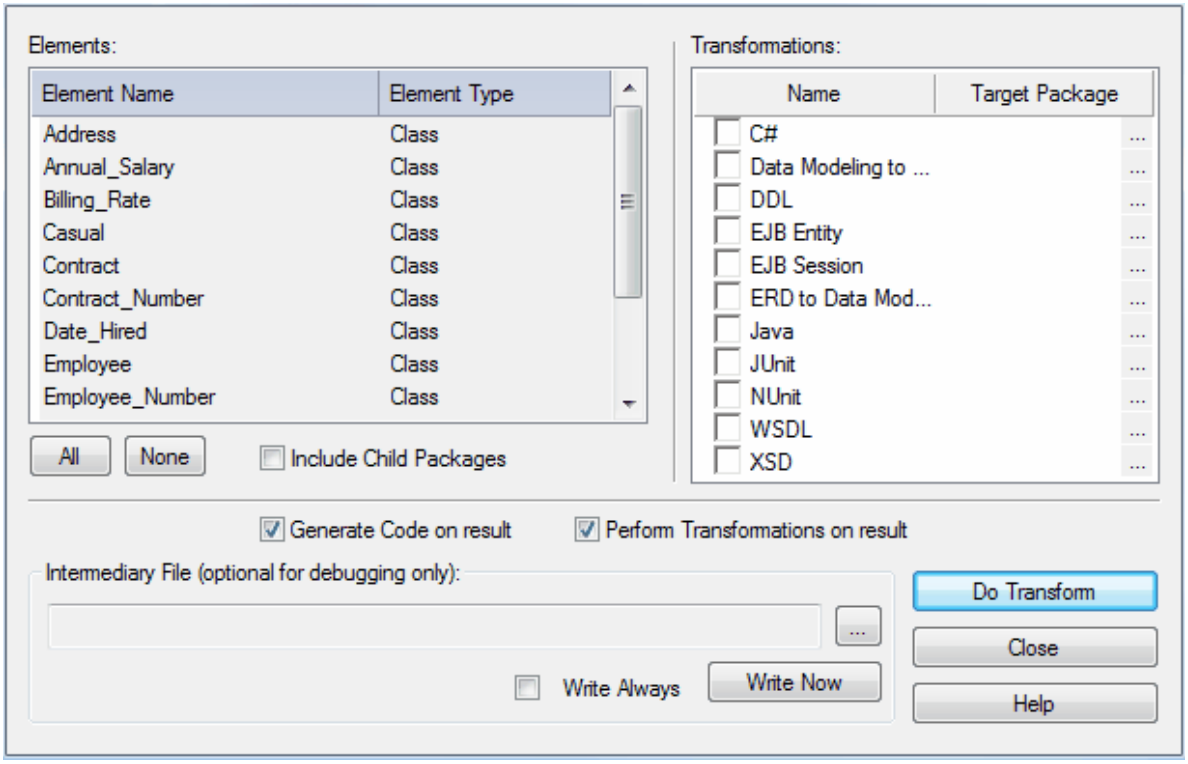
1 Transform Elements



There are two modes for initiating a Model Transformation, each of which can be started in two ways.

- To transform selected elements on a diagram, either:
  - Select the **Project | Transformations | Transform Selected Elements** menu option, or
  - From the context menu for the Classes on the diagram, select the **Transform** option.
- To transform elements in the package currently selected in the Project Browser, either:
  - Select the **Project | Transformations | Transform Current Package** menu option, or
  - From the context menu of the package in the Project Browser, select the **Transform Current Package** option.

The Model Transformation dialog displays.



When the dialog displays, all elements are selected and all transformations previously performed from any of these Classes are checked.

Option	Use to
Elements	Select (click on) the individual elements to be included in the transformation.
All	Select all of the elements from the list to be included in the transformation.
None	Deselect all of the elements from the list.

Option	Use to
Include child packages	Select to include elements in child packages of the selected package.
Transformations	Select which transformations to perform and the package each of them should be transformed to. (Use the **[...]** button to select the package in which the transformed elements are being created.)
Generate Code on result	Specify whether or not to automatically generate code for transformed Classes that target code languages. Automatically generating code helps boost productivity in development. With this option selected, the first time you transform to the selected Class Enterprise Architect enables you to select a filename to generate to. Subsequent transformations automatically generate any Class with a filename set.
Perform Transformations on result	Specify if transformations previously done on target Classes should be automatically executed. See Chaining Transformations 6 for more information.
Intermediary File	Specify the filename of the intermediary file (if any).
Write Always	Write the intermediary file to disk.
Write Now	Generate the intermediary file but do not perform the transform.
Do Transform	Execute the transform command.

1.1 Chaining Transformations

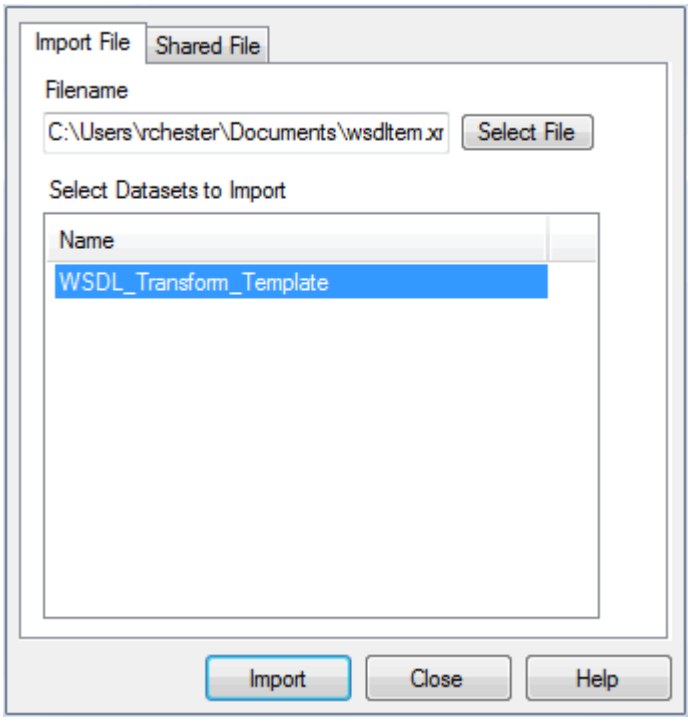
Chaining transformations provide an extra degree of flexibility and power to transformations. For example, you might have a situation where two transformations have a common element. This can be separated out into one transformation, and then the original transformations can be transformed from the common point. The separated transform could even produce a useful model itself.

Enterprise Architect provides for chaining transformations, by enabling transformations that have already been performed on target Classes to be performed automatically next time that Class is transformed to. To enable this, select the **Perform Transformations on result** checkbox in the Model Transformation dialog.

2 Import Transformations

You can transfer Transformation templates between models. To import a Transformation template, follow the steps below:

1. Select the **Tools | Import Reference Data** menu option. The Import Reference Data dialog displays.



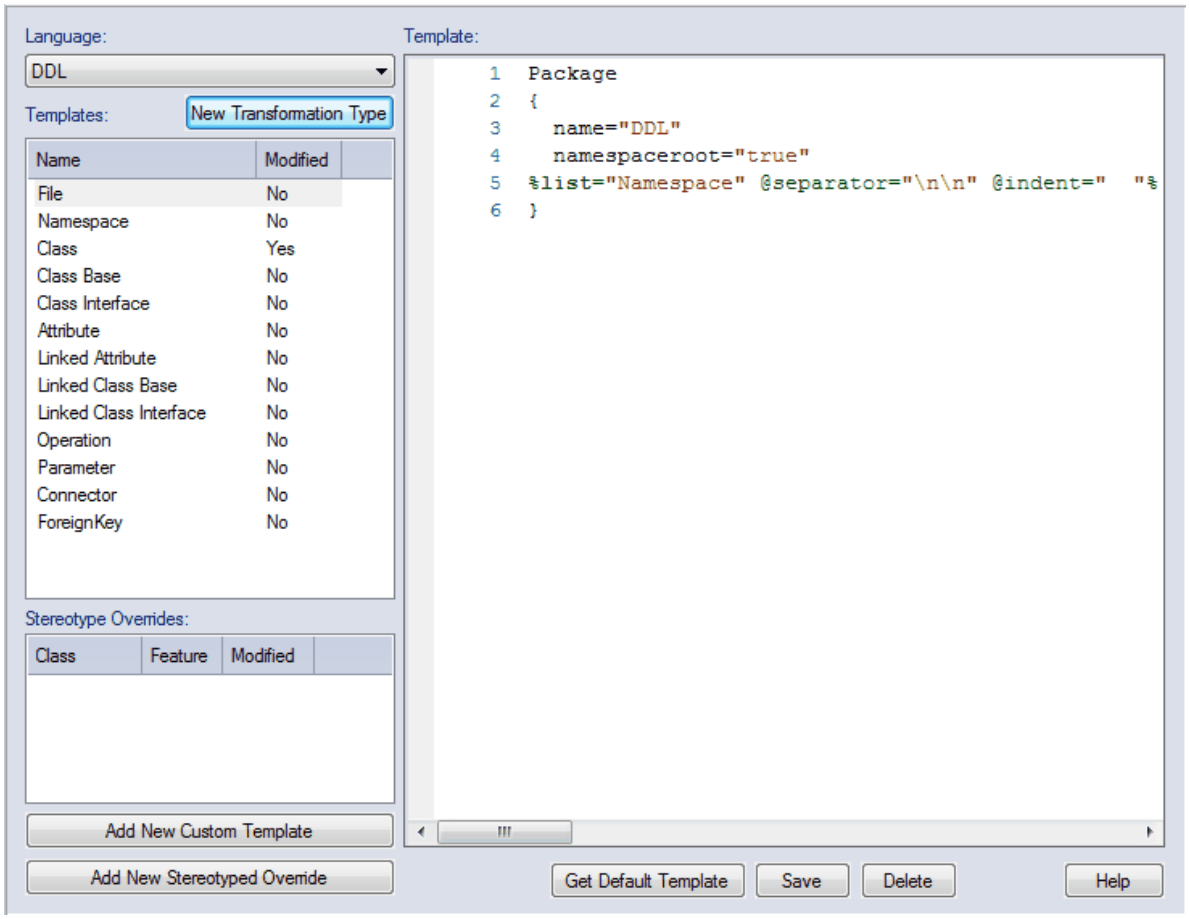
2. Click on the **Select File** button and browse to a .XML file containing the required Transformation template.
3. Select the name of one or more template datasets and click on the **Import** button.

3 Transformation Templates

Note that the Transformation Template mechanism is based very strongly on the Code Generation Template mechanism. For further information on Transformation Templates see the *Code Template Editor* section of *SDK for Enterprise Architect,* and also - for information on the Common Code Editor and intellisense - the *Code Editors* topic in *Using Enterprise Architect - UML Modeling Tool*.

To modify Transformation templates:

1. Select the **Settings | Transformation Templates** menu option. The Transformation Templates Editor displays.
2. In the **Language** field, type or select the name of the transformation to modify.
3. Select a template from the Templates list, and edit its contents in the editor pane.
4. Click on the **Save** button.



Option	Use to
Language	Select the name of the transformation.
Template	Display the contents of the active template. Provide the editor for modifying templates.
Templates	List the base transformation templates. The active template is highlighted. The **Modified** field indicates whether you have changed the default template for the current transformation.
New Transformation Type	Create a new transformation.

Option	Use to
Stereotype Overrides	List the stereotyped templates, for the active base template. The **Modified** field indicates whether you have modified a default stereotyped template.
Add New Stereotyped Override	Invoke a dialog for adding a stereotyped template, for the currently selected base template.
Add New Custom Template	Invoke a dialog for creating a custom stereotyped template.
Help	Launch the Enterprise Architect Help topic for this dialog.
Get Default Template	Update the editor display with the default version of the active template.
Save	Overwrite the active templates with the contents of the editor.
Delete	If you have overridden the active template, delete the override and replace it with the corresponding default transformation template.

4 Built-in Transformations

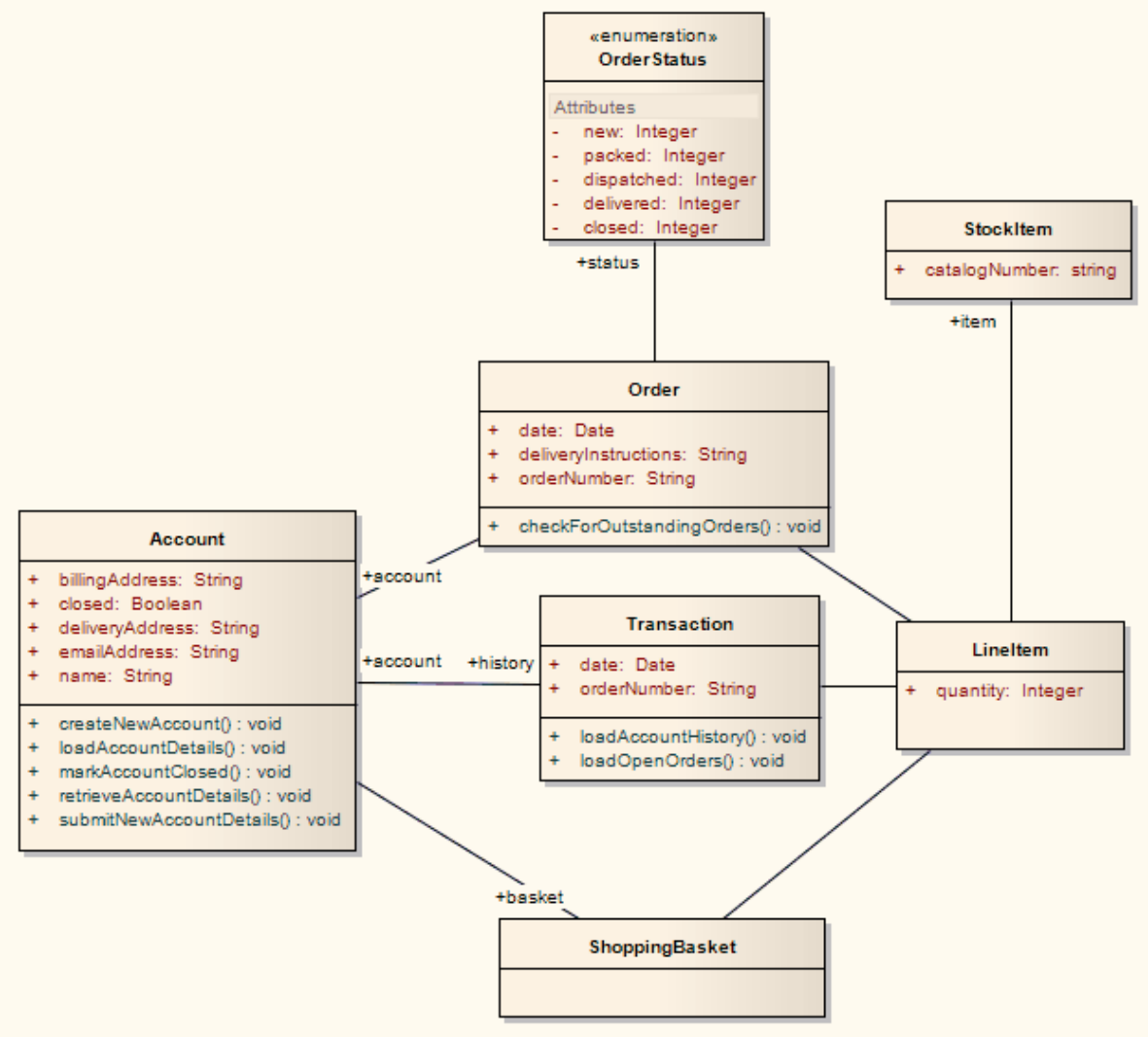
Enterprise Architect comes with some built-in transformation types. These transformations have been designed to be useful to as many users as possible, to be a good base to modify to include the specifics of your custom domain, and to be good examples of how to write transformations.

The following transformations are included in Enterprise Architect.

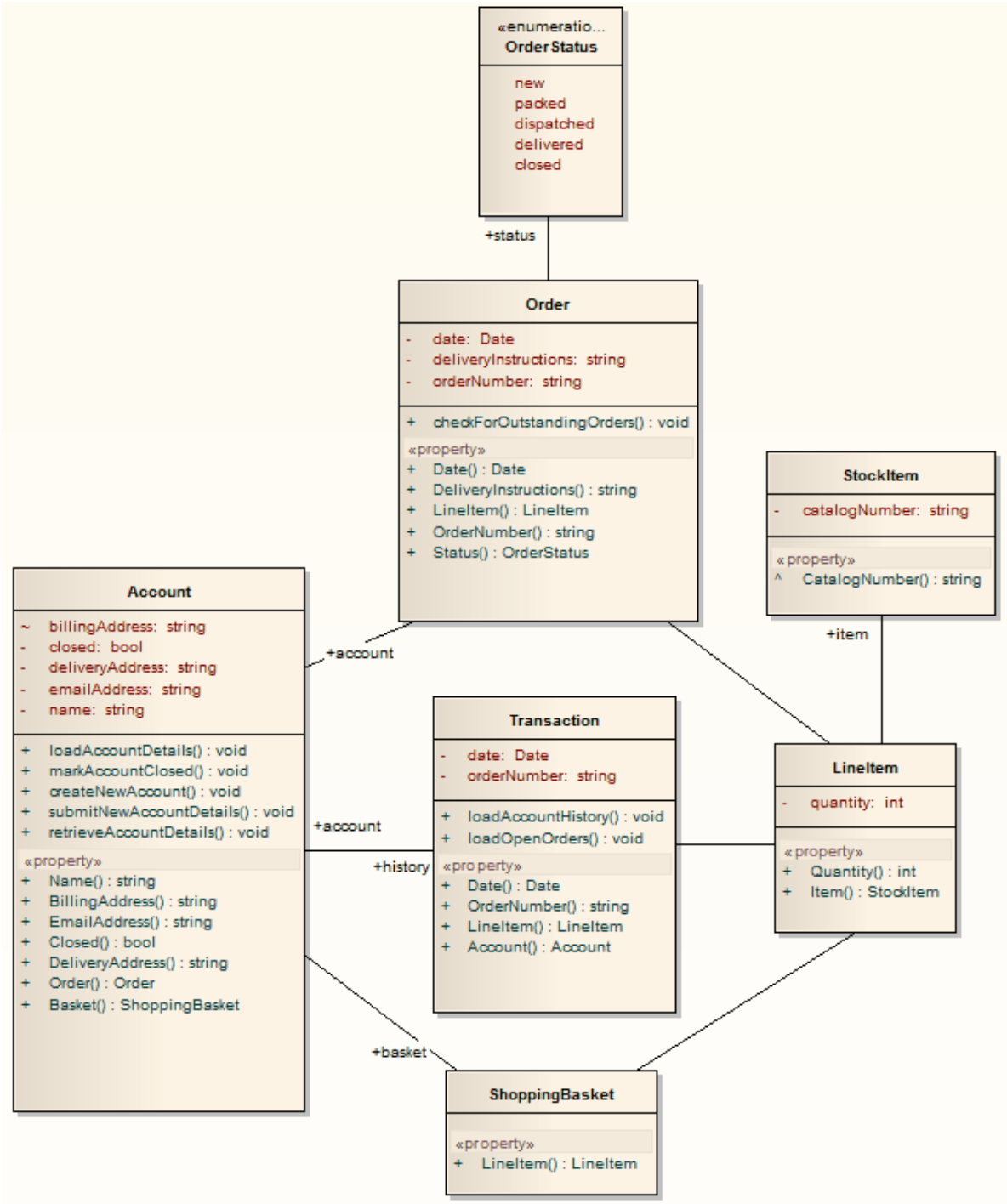
4.1 C# Transformation

The **C# transformation** converts Platform-Independent Model (PIM) elements to language-specific C# Class elements. The transformation converts PIM model types to C# types and creates encapsulation according to Enterprise Architect's options for creating properties from C# attributes, which you set on the C# Specifications page of the Options dialog (see the *Code Engineering Settings* topic in *Code Engineering Using UML Models)*.

The Platform-Independent Model (PIM):



After transformation, becomes the PSM:



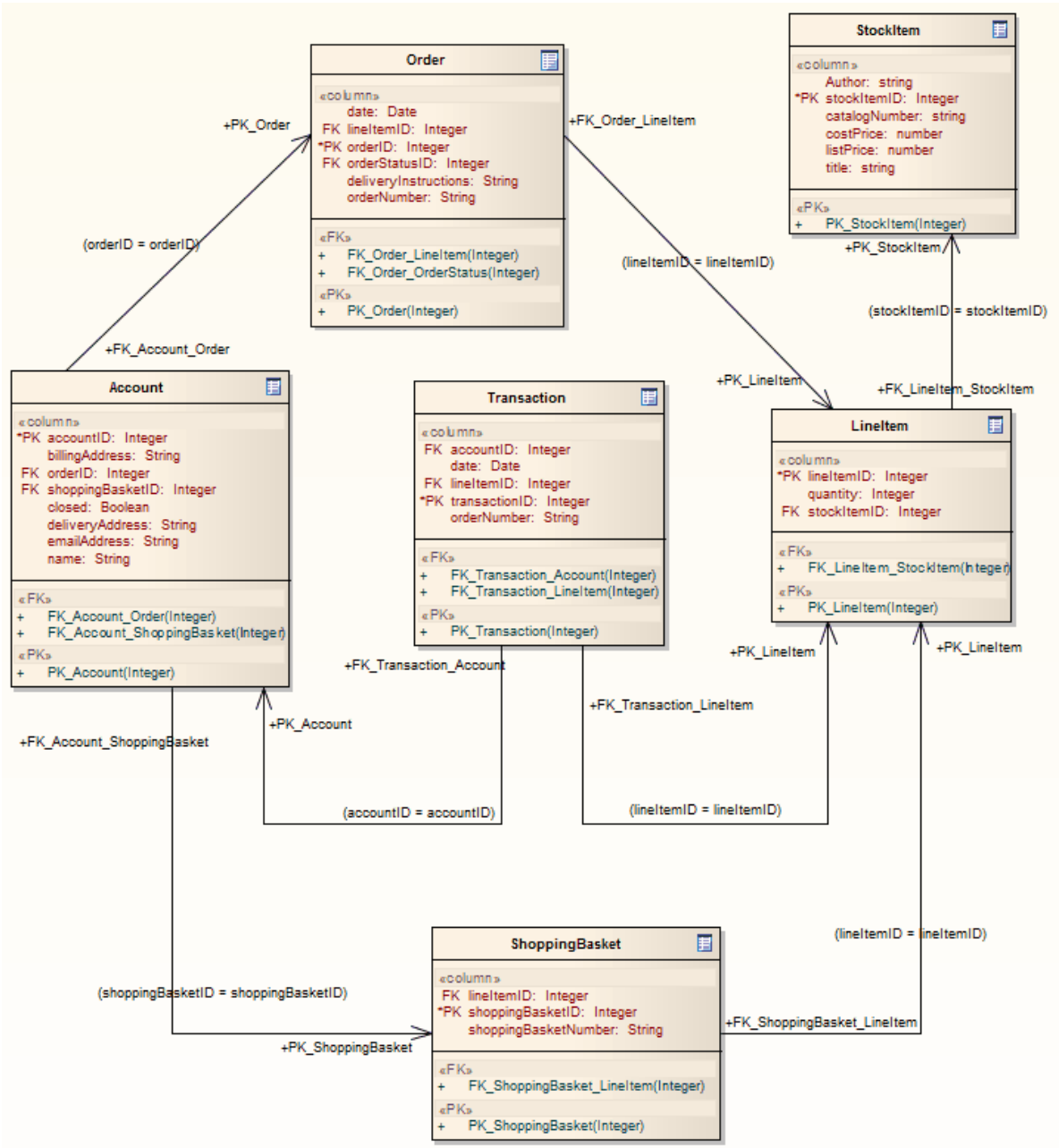
4.2 Data Model To ERD Transformation

The **Data Model to ERD transformation** is a reverse engineering of the **ERD to Data Model** 21 **transformation**.

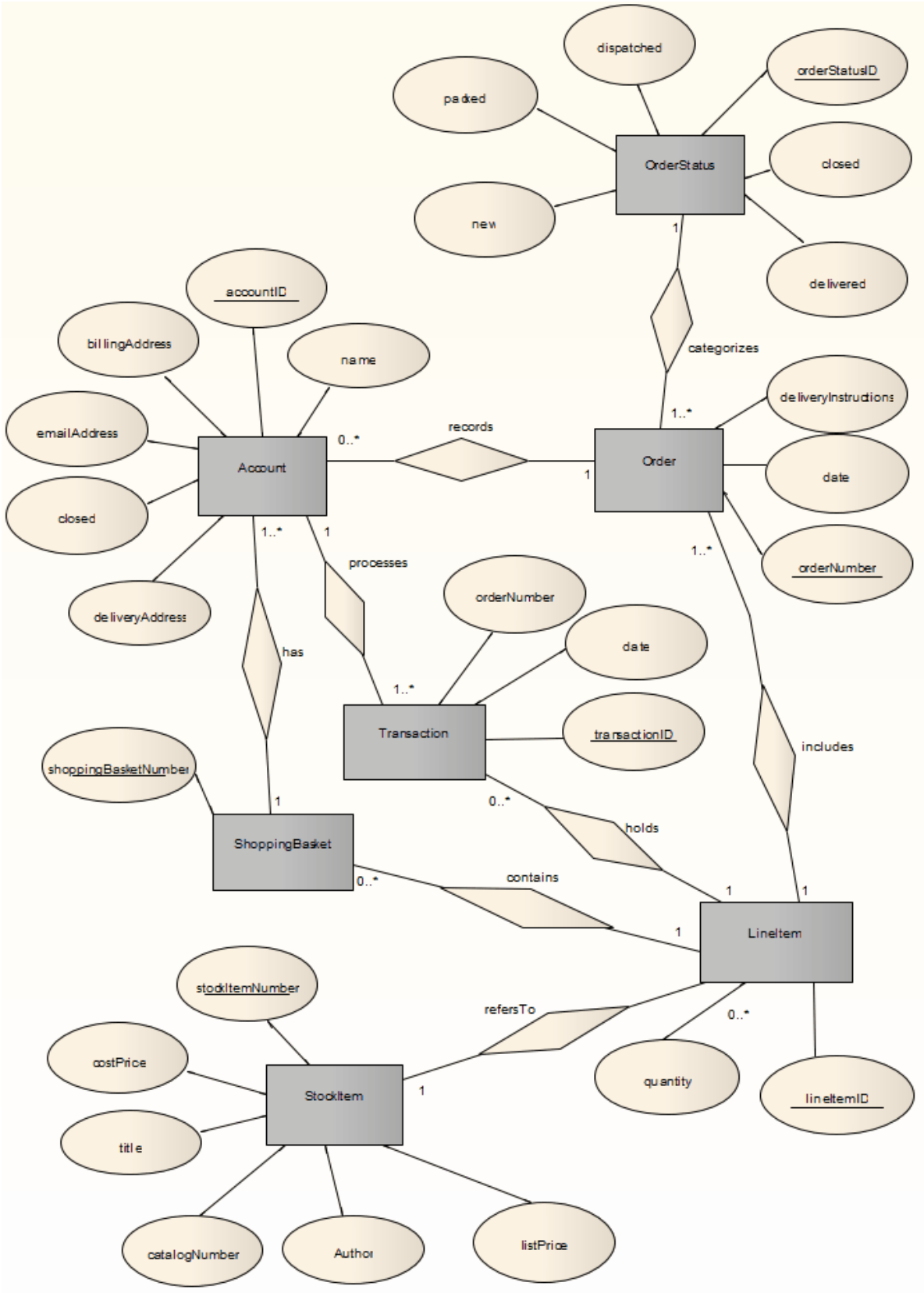
The transformation uses and demonstrates support in the intermediary language for the following database-specific concepts:

Entity	Mapped one-to-one onto table elements.
Attribute	Mapped one-to-one onto columns.
Primary Key	Comes from the PrimaryKey type of column.

The source Data Model diagram:



After transformation becomes the Entity Relationship Diagram:



Tip:

Sometimes you might want to limit the stretch of the diamond-shape Relationship connectors. Simply pick a Relationship connector, right-click to display the context menu, and select the **Bend Line at Cursor** option.

4.3 DDL Transformation

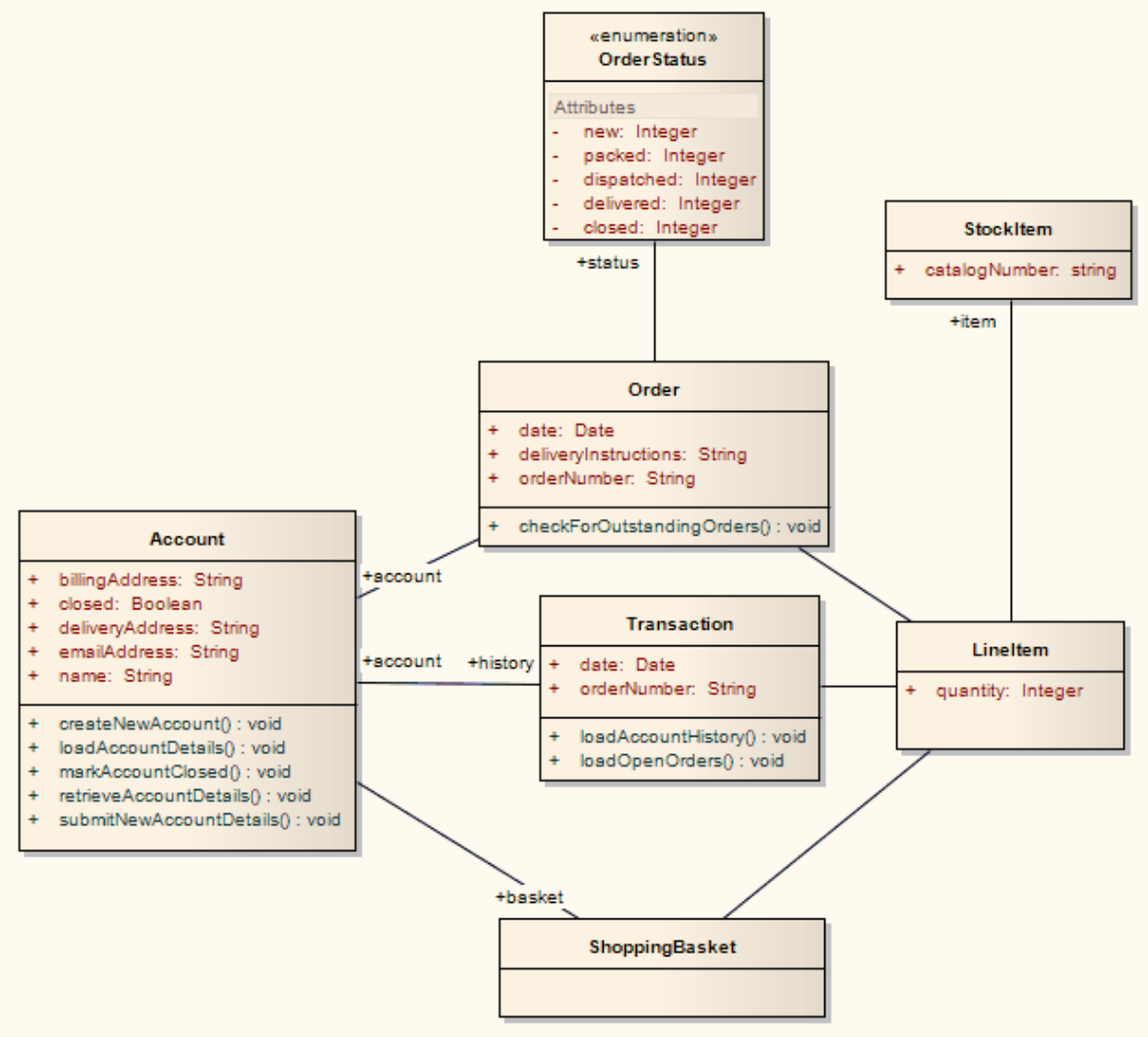
The purpose of the **DDL transformation** is to create a data model from the logical model, generating a model targeted at the default database type that is ready for DDL generation. The data model can then be used to automatically generate DDL statements to run in one of the Enterprise Architect supported database products.

The DDL Transformation uses and demonstrates support in the intermediary language 35 for the following database-specific concepts:

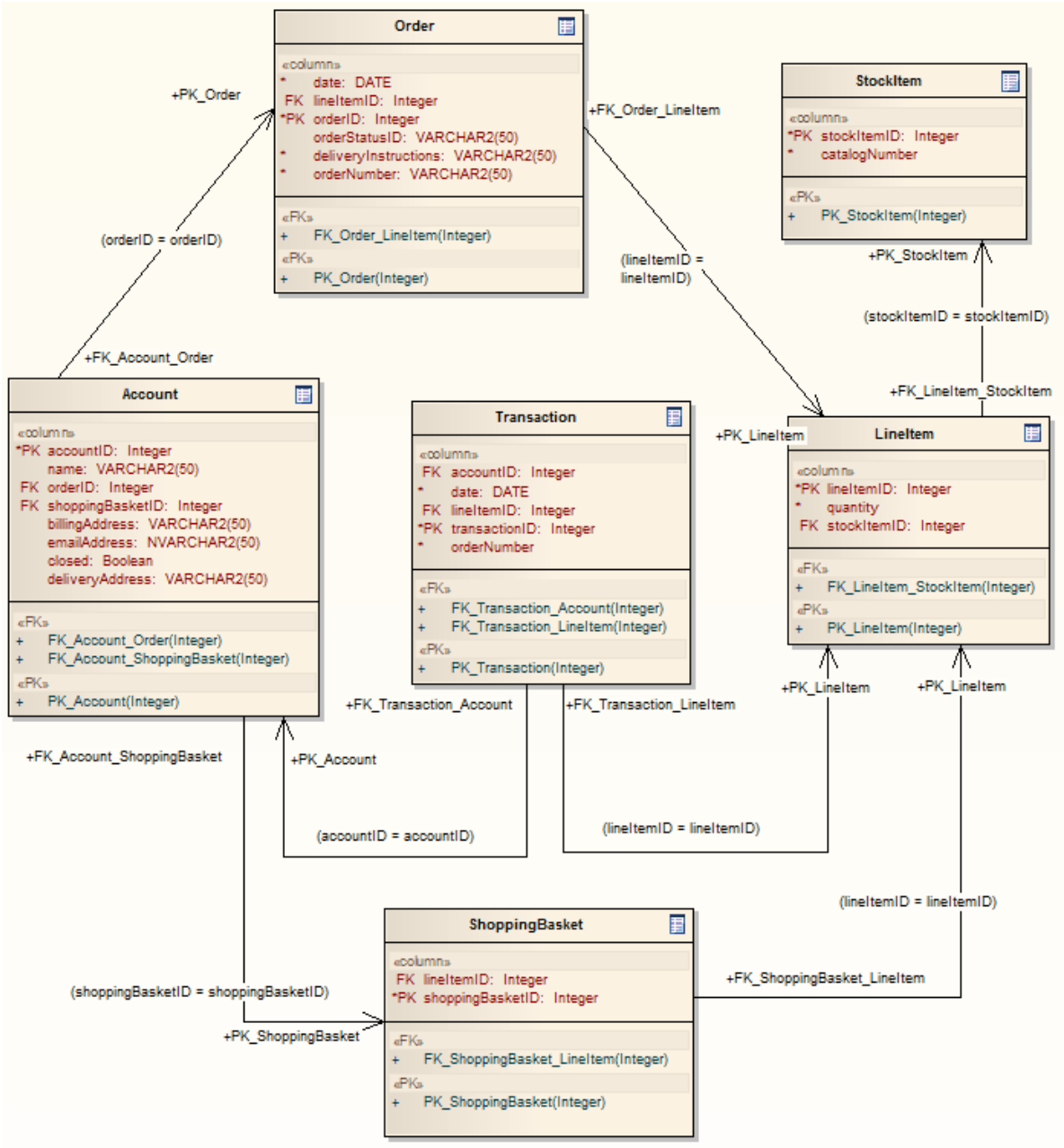
Table	Mapped one-to-one onto Class elements.
Column	Mapped one-to-one onto attributes.
Primary Key	Lists all the columns involved; this ensures that they exist in the Class and creates a primary key method for them.
Foreign Key	This is a special sort of connector. The *Source* and *Target* sections list all of the columns involved; this ensures that they exist and that a matching primary key exists in the destination Class, and that the transformation creates the appropriate foreign key.

The following two diagrams show a typical PIM to Data Model Transformation.

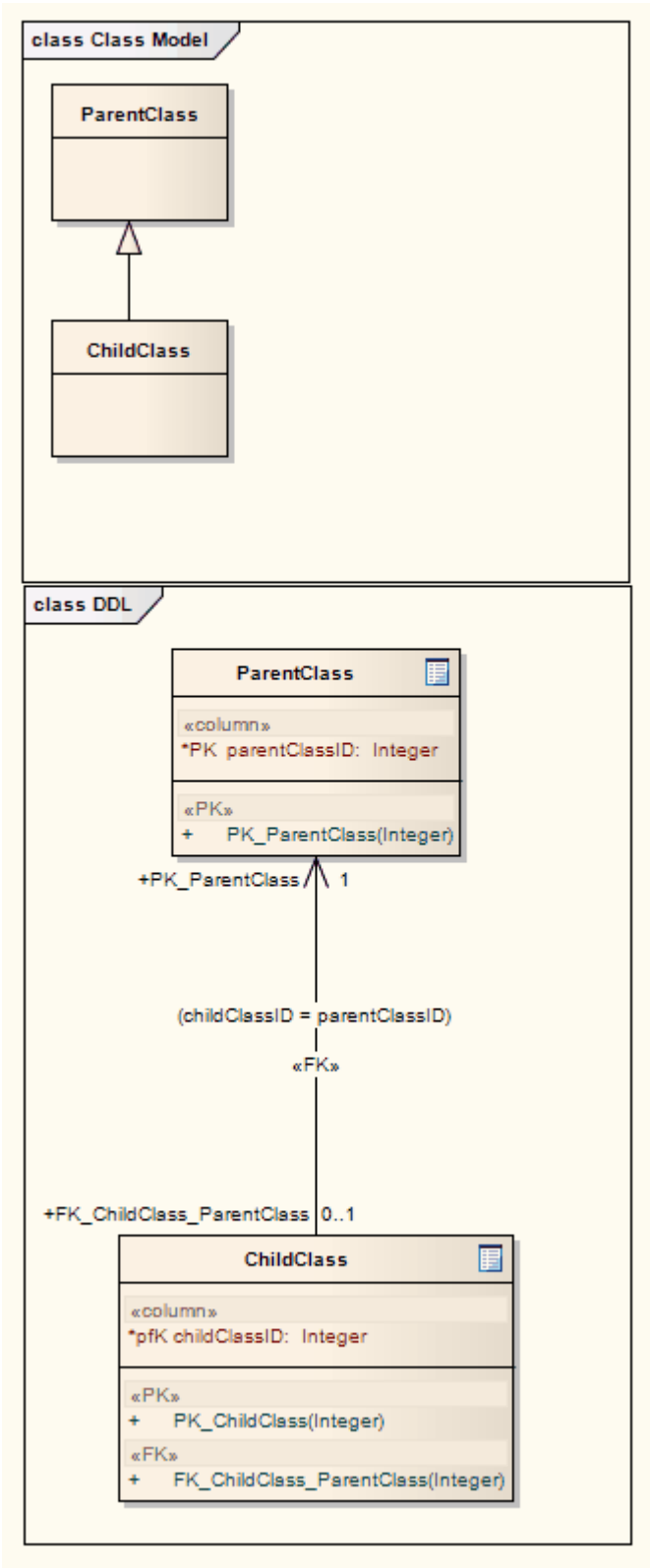
The Platform-Independent Model (PIM):



After transformation becomes the PSM:



Generalizations are handled by providing the child element with a foreign key to the parent element, as in the following diagram. Copy-down inheritance is not supported.



4.4 EJB Transformations

The purpose of the **EJB Session Bean transformation** and the **EJB Entity Bean transformation** is to reduce the work required in generating the internals of Enterprise Java Beans, thus enabling you to concentrate on modeling at a higher level of abstraction.

The **EJB Session Bean transformation** generates the following from a single Class element containing the attributes, operations and references required for code generation by the *javax.ejb.** package:

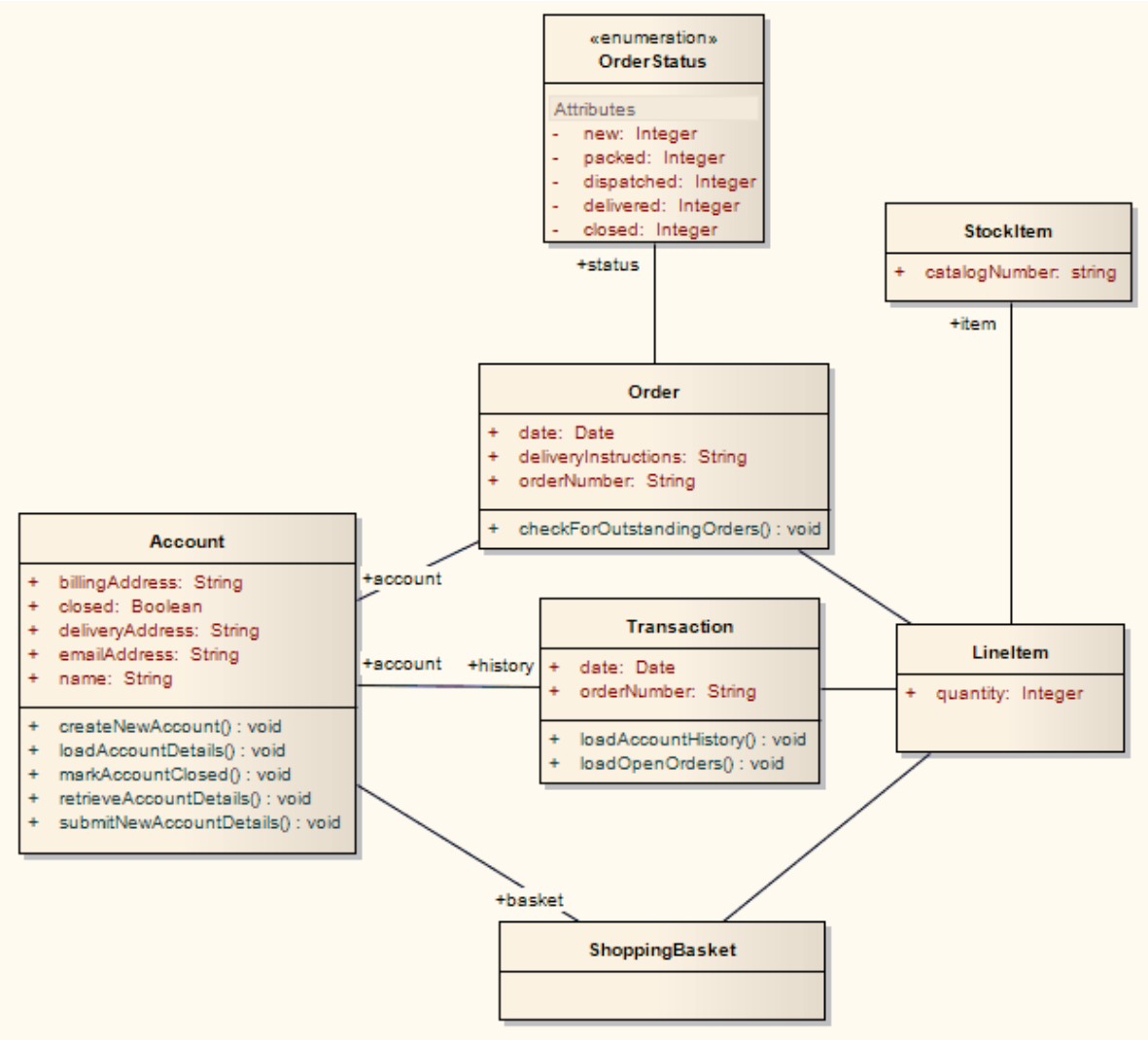
- An implementation Class element
- A home interface element
- A remote interface element.

The **EJB Entity Bean transformation** generates the following from a single Class element containing the attributes, operations and references required for code generation by the *javax.ejb.** package:

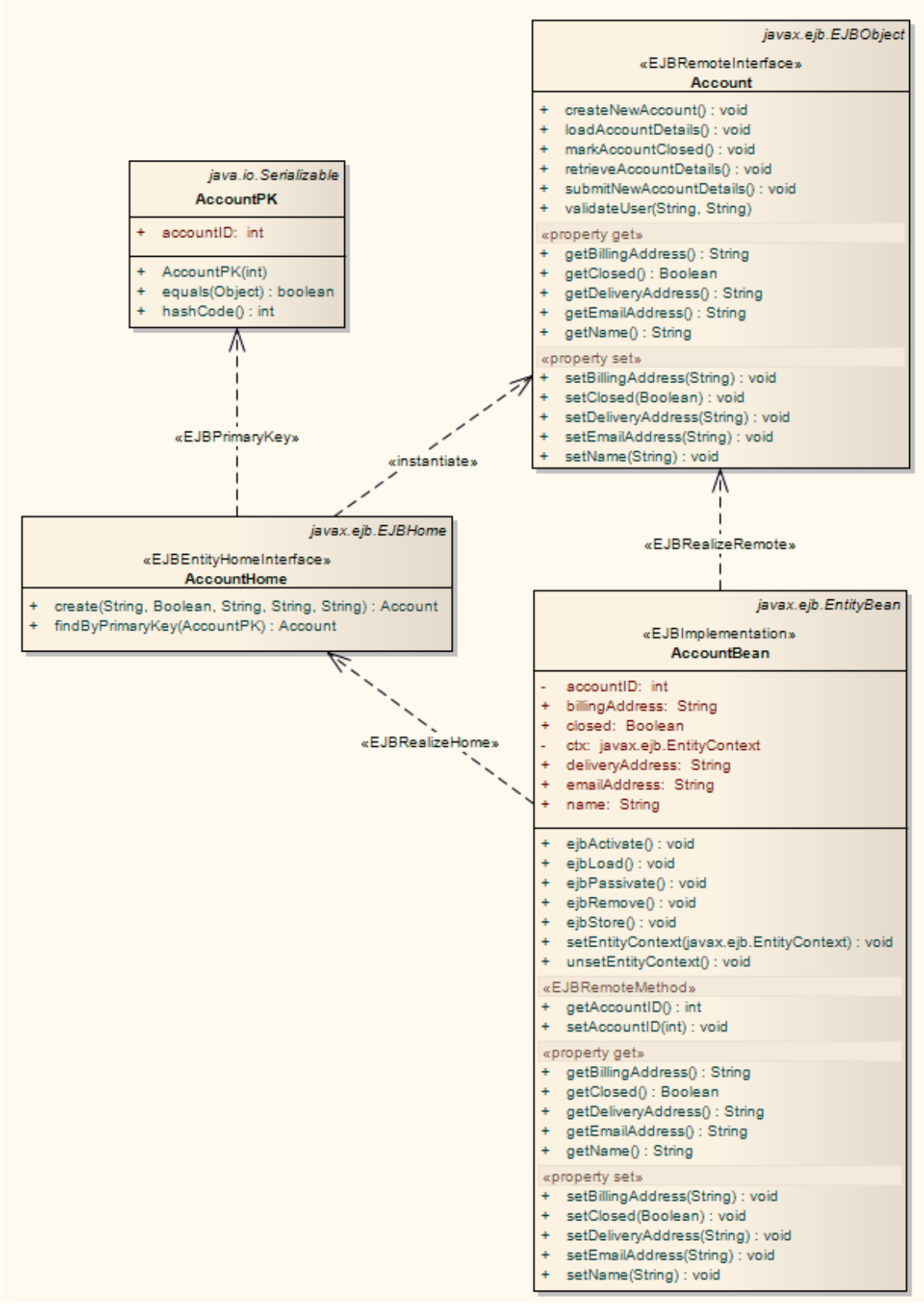
- An implementation Class element
- A home interface element
- A remote interface element
- A primary key element.

Both transformations also generate a META-INF package containing a deployment descriptor element.

The Platform-Independent Model (PIM):



After transformation generates a set of Entity Beans, where each one takes the following form (for the Account Class):



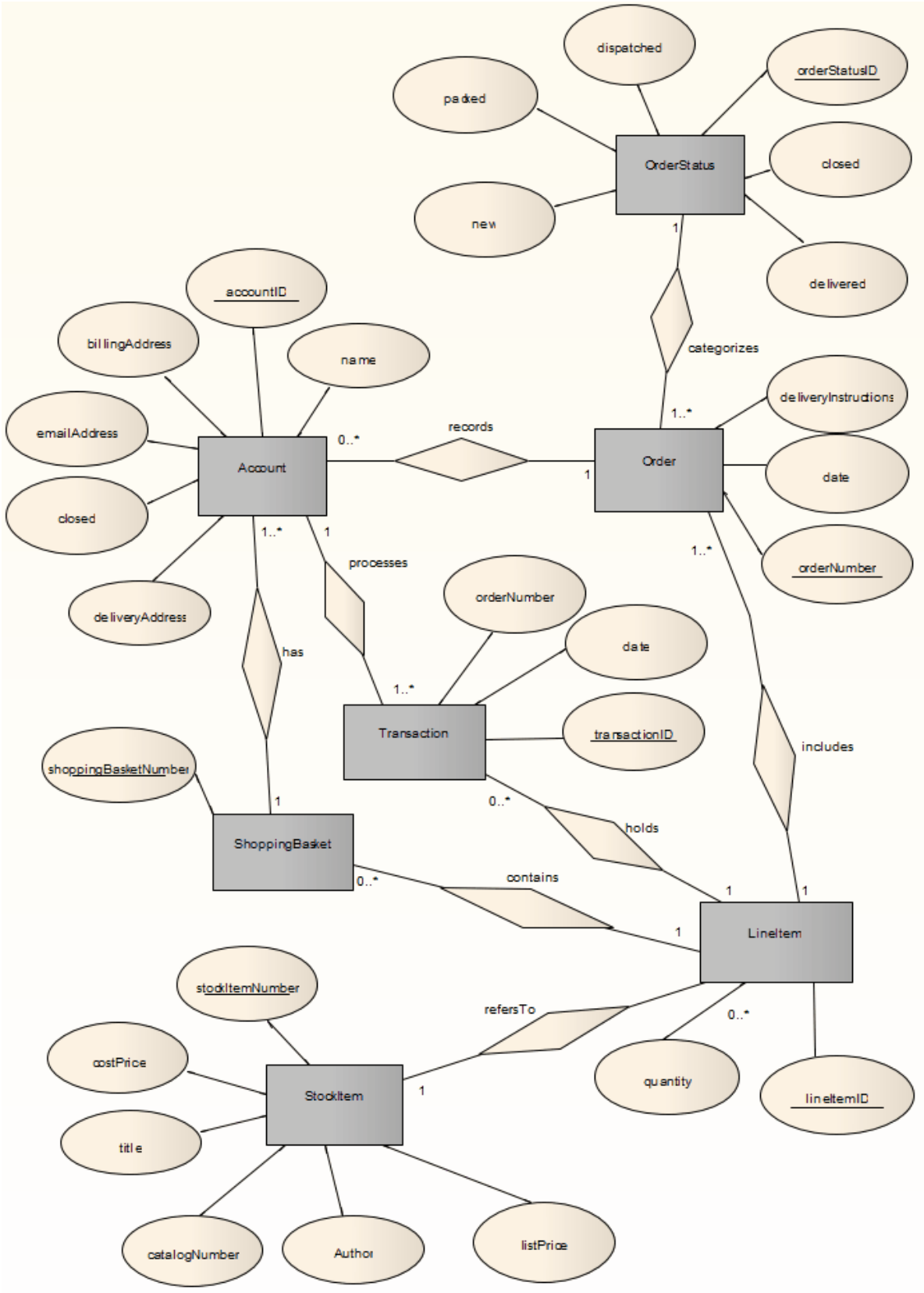
4.5 ERD To Data Model Transformation

The purpose of the **Entity Relationship Diagram (ERD) to Data Model transformation** is to create a data model from the ERD logical model, generating a model targeted at the default database type ready for DDL generation. Before doing the transformation, make sure you have defined the common data type for each attribute and selected a database type as the default database. The data modeling diagram can then be automatically generated. This data model can be used for generating DDL statements to run in one of the Enterprise Architect supported database products.

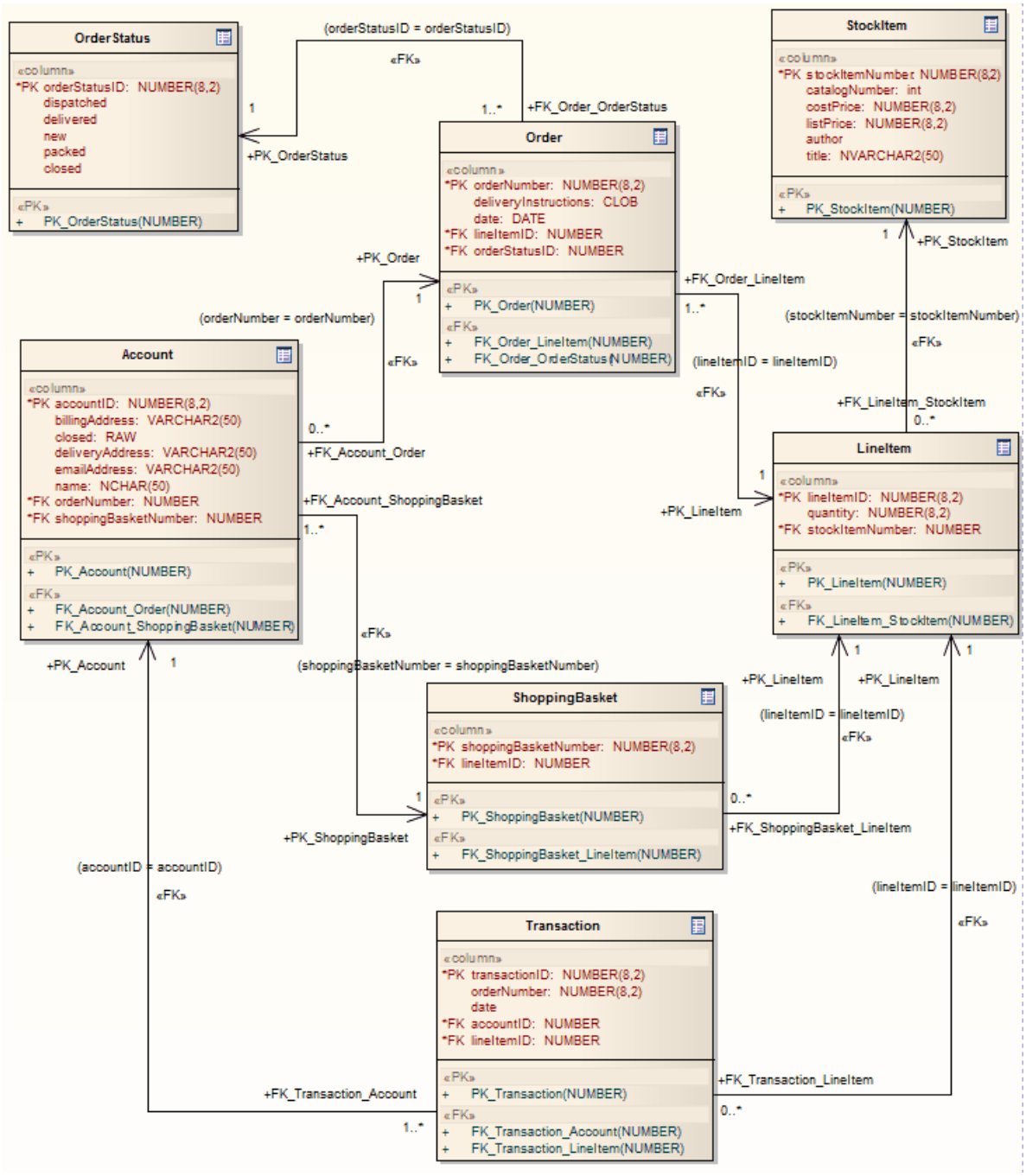
The transformation uses and demonstrates support in the intermediary language for the following database-specific concepts:

Table	Mapped one-to-one onto Entity elements.
Column	Mapped one-to-one onto attributes.
Primary Key	Comes from the *primaryKey* type of attribute.
Foreign Key	Make sure the primary key exists in the source Entity; the transformation then creates the appropriate foreign key.

The Source Entity Relationship Diagram:



After transformation becomes the Data Model Diagram (for an Oracle DBMS):

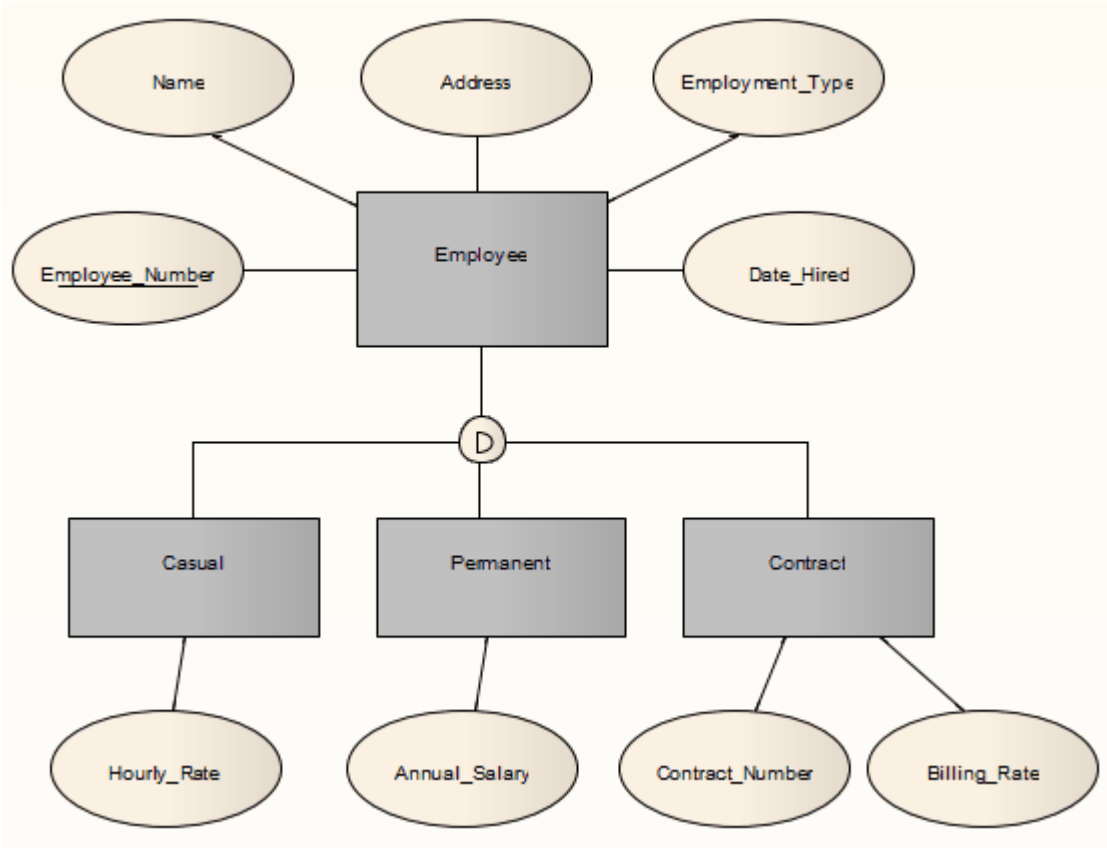


Tip:

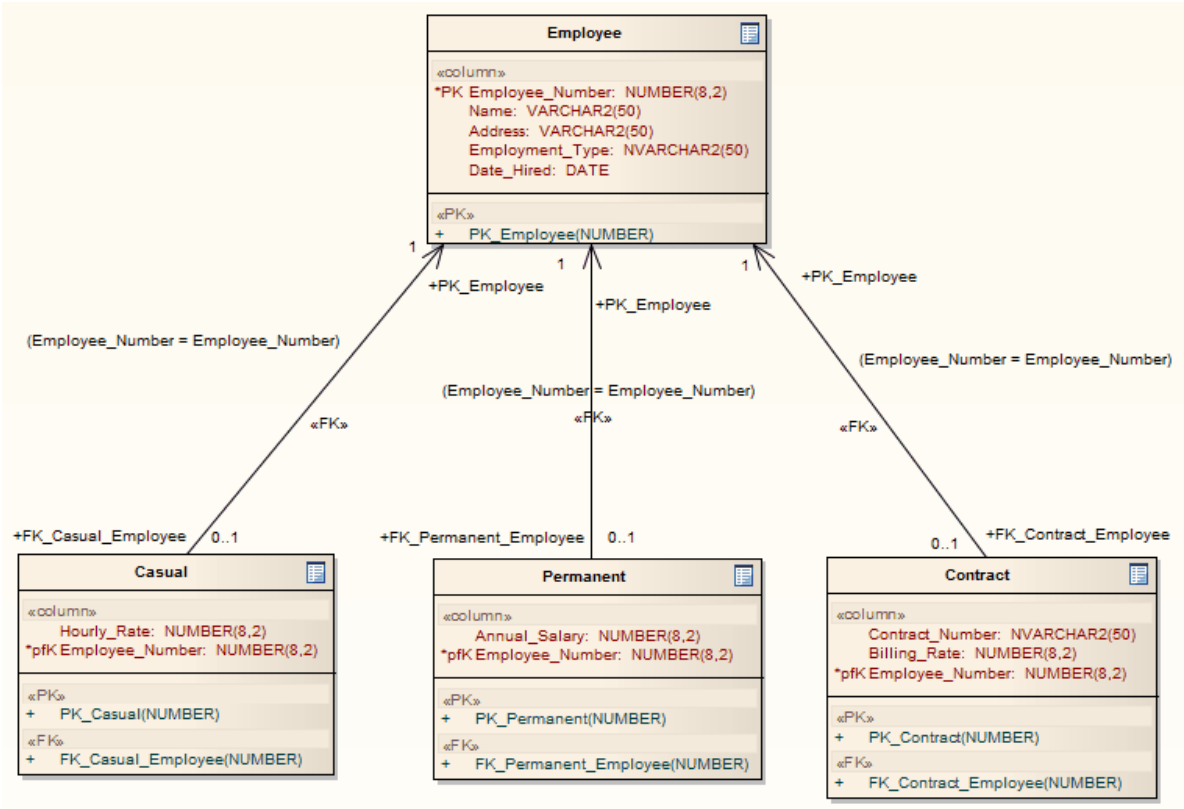
Sometimes you might go back to the ERD, make some changes and then want to do another transformation. To achieve better results, always delete the previous transformation package before doing the next transformation.

Generalization

Generalization can be handled in ERD technology, as illustrated by the following example. Note that the copy-down inheritance is currently supported with two levels only.



This transforms to:

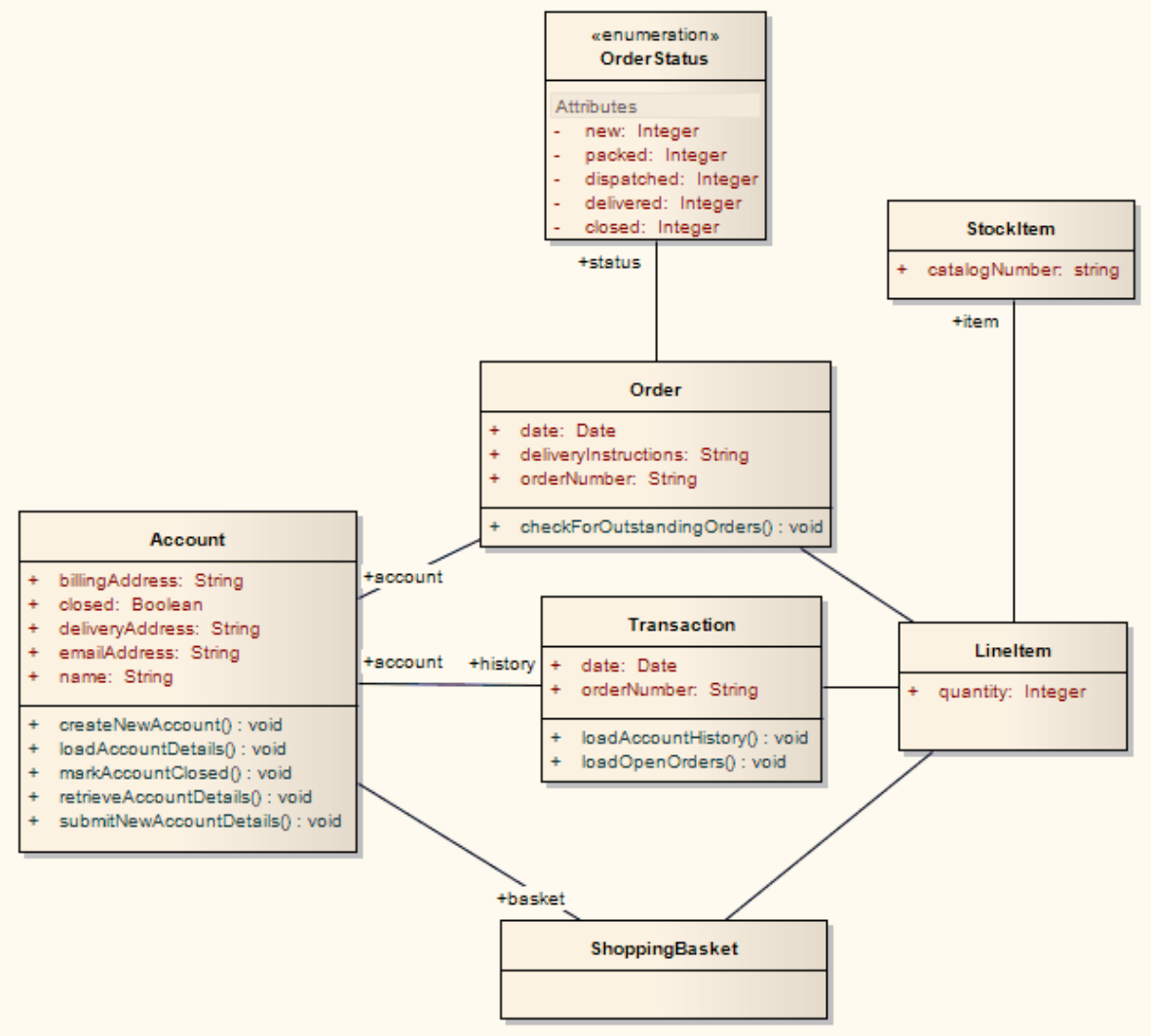


4.6 Java Transformation

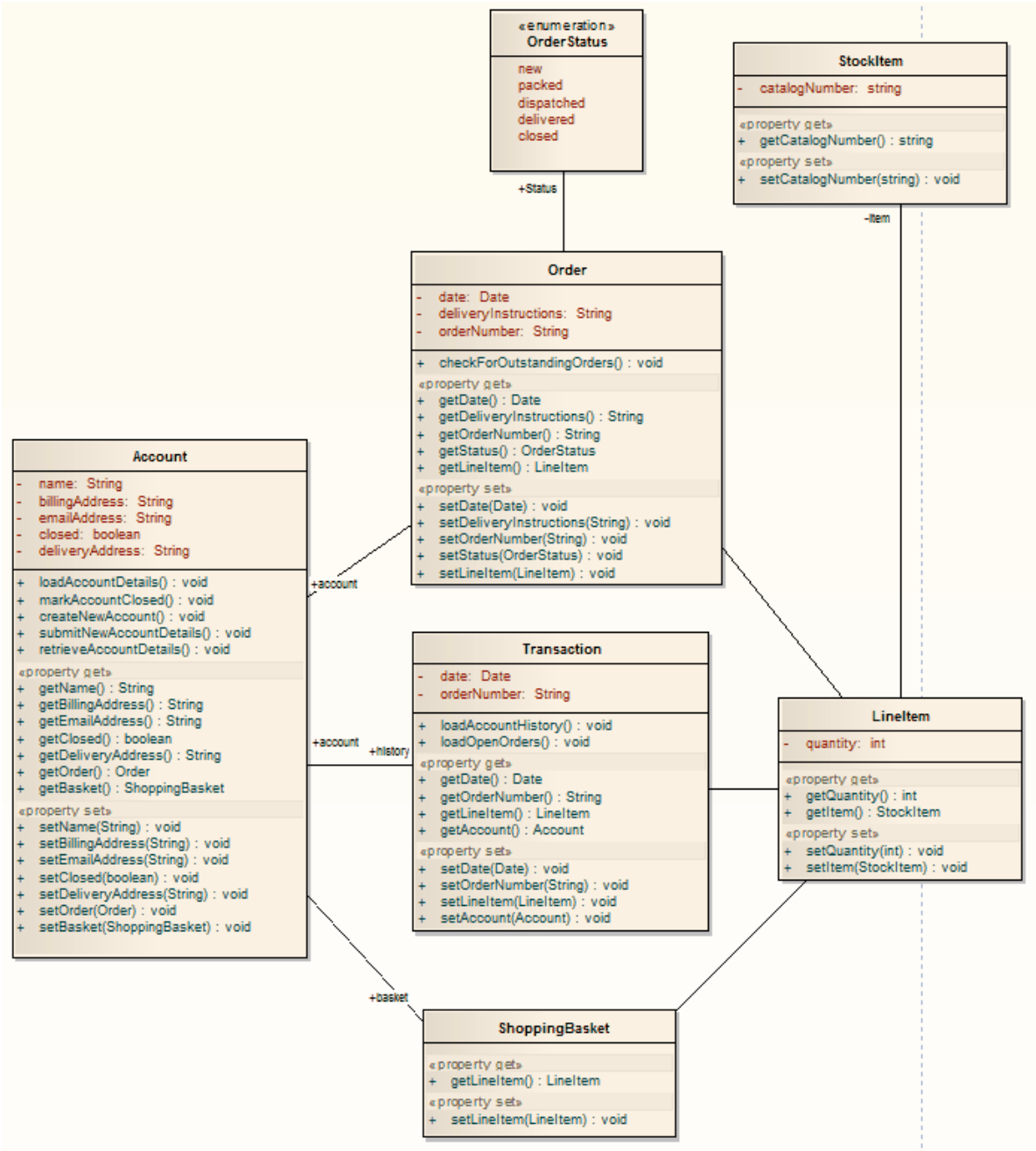
The purpose of the **Java transformation** is to convert Platform-Independent Model (PIM) elements to language-specific Java Class elements. The transformation converts the PIM model types to Java types and creates encapsulation according to Enterprise Architect's options for creating properties from Java attributes; that is, producing the getters and setters according to the rules you have defined. Notice that the *public* attributes in the PIM are converted to *private* attributes in the PSM.

You set the code generation options for Java code generation on the Java Specifications page of the Options dialog (see the *Code Engineering Settings* topic in *Code Engineering Using UML Models*).

The Platform-Independent Model (PIM):



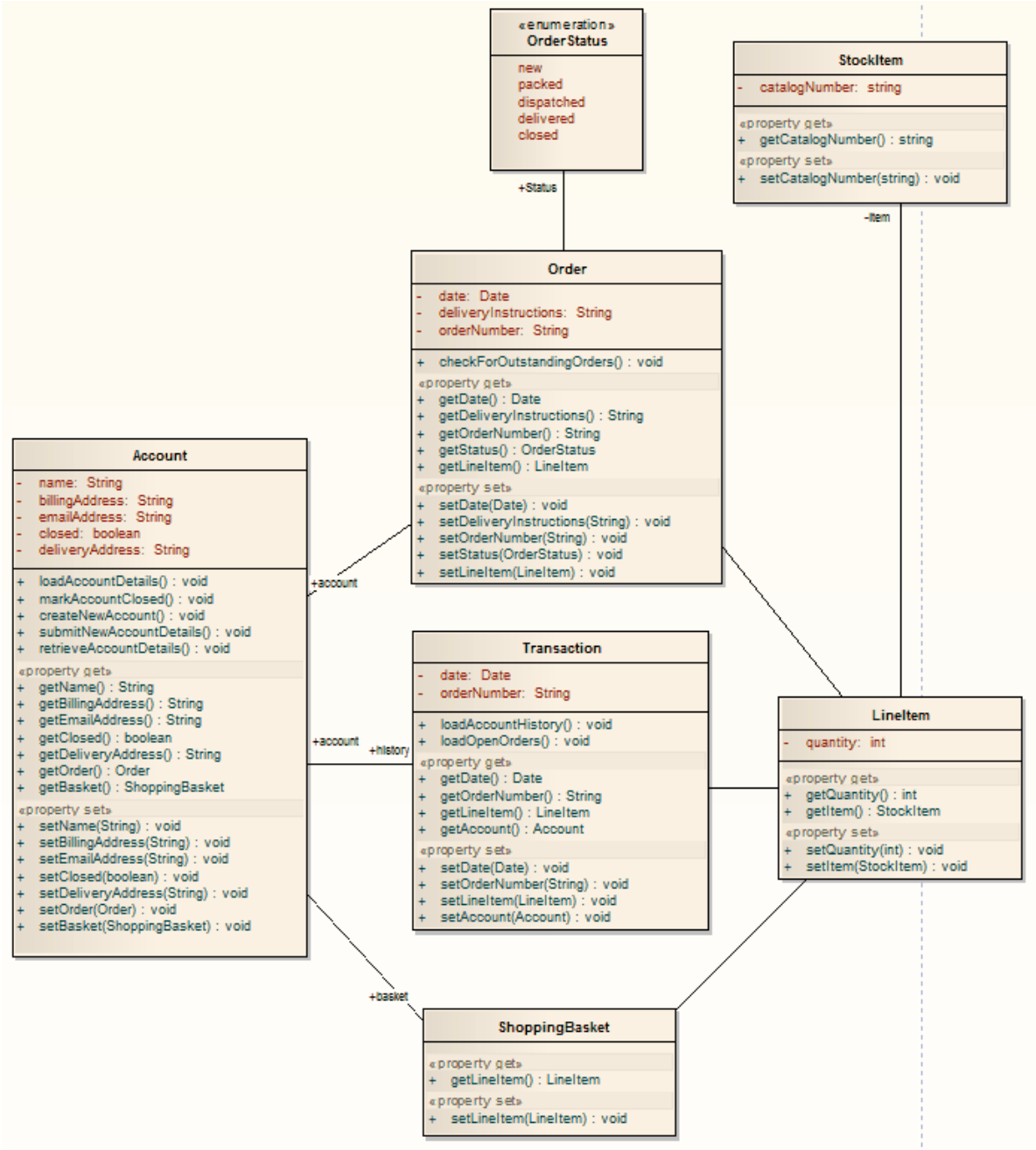
After transformation becomes the PSM:



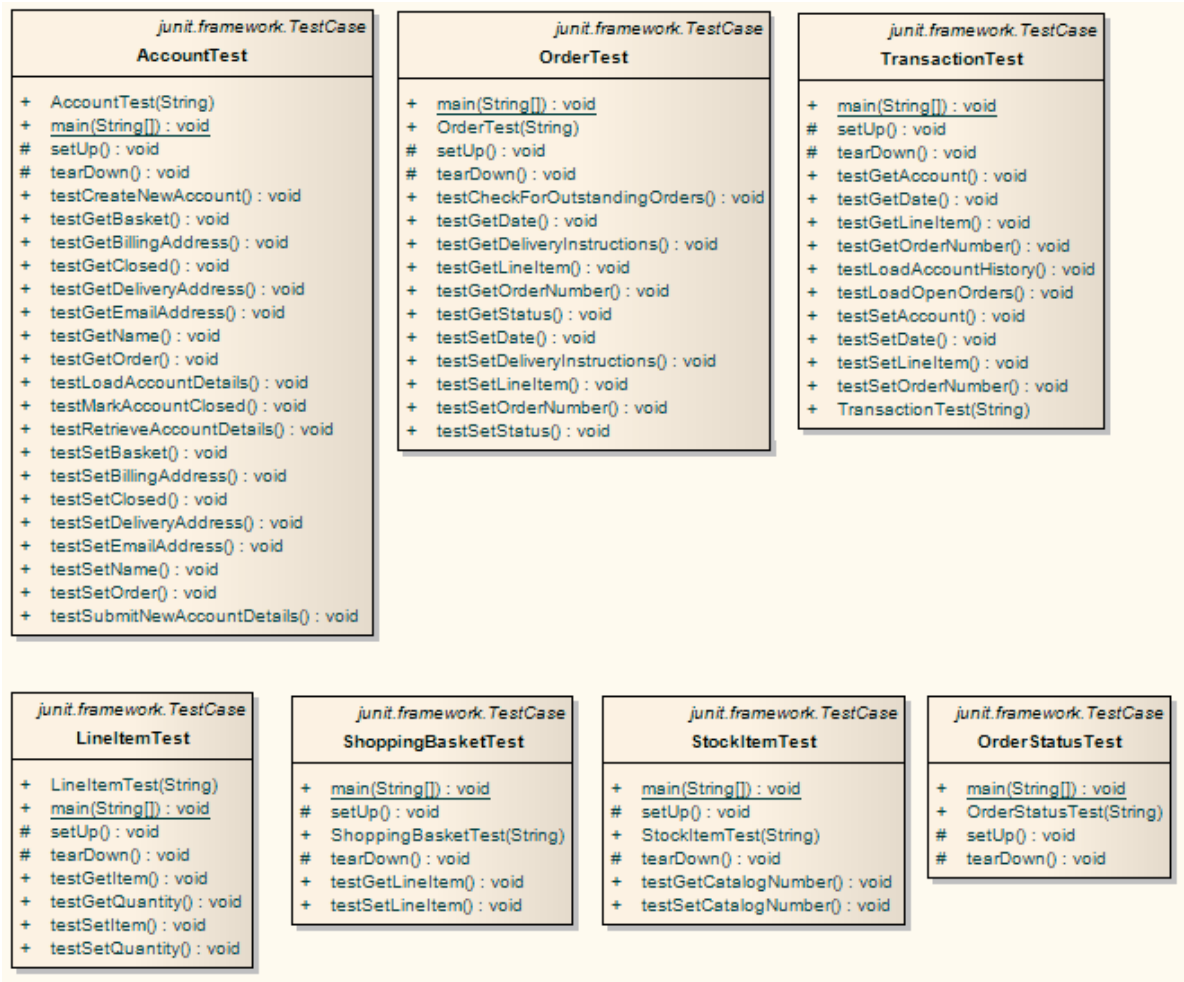
4.7 JUnit Transformation

The purpose of the **JUnit transformation** is to create a Class with test methods for all public methods of an existing Java Class. The resulting Class can then be generated and the tests filled out and run by JUnit.

The Java model originally transformed from the PIM:



After transformation becomes the PSM:

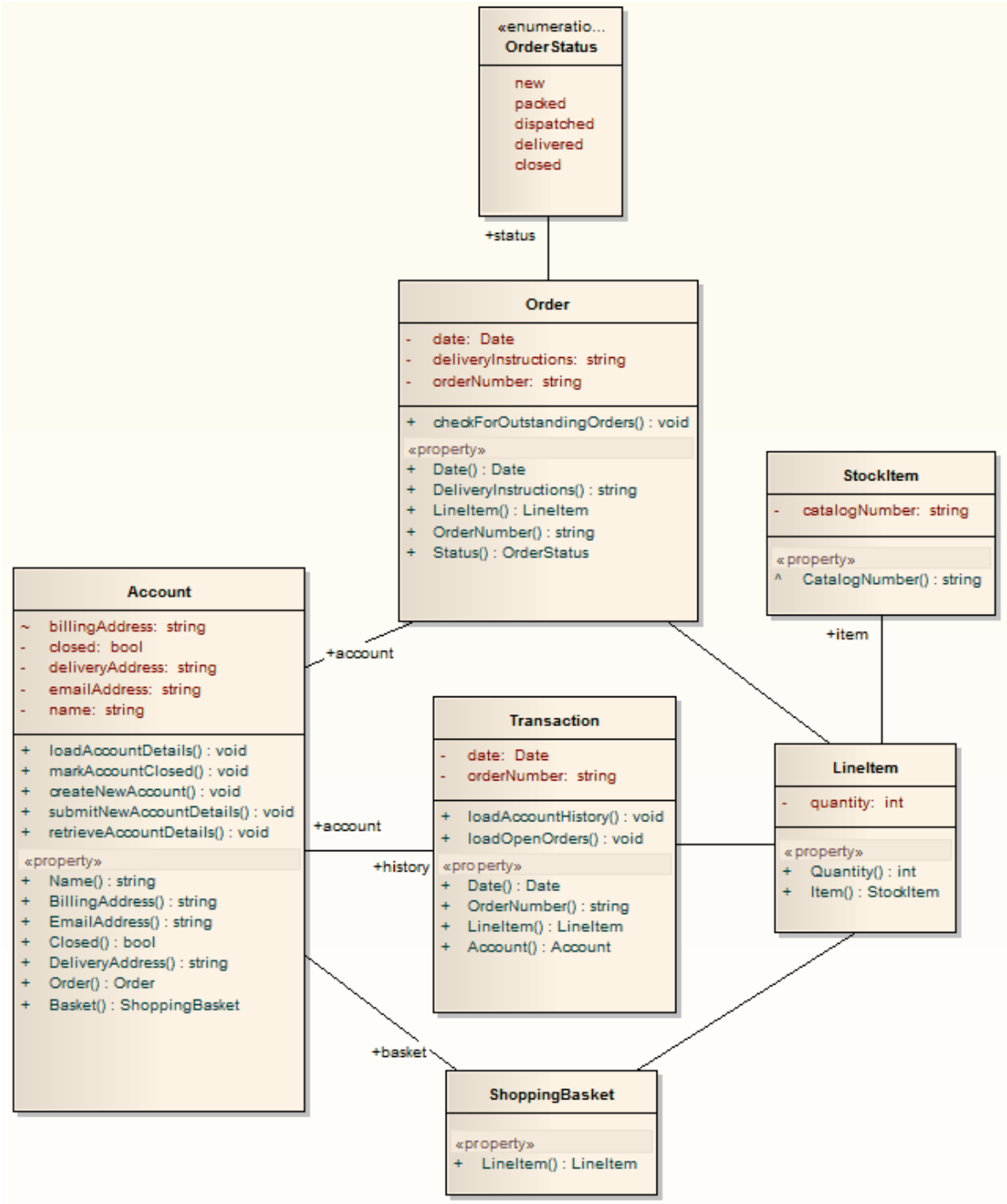


Note that for each Class in the Java model, a corresponding test Class has been created. Each of these test Classes contains a test method for every public method in the source Class, plus the methods required to appropriately set up the tests. It is your responsibility to fill in the details of each test. (See the *Unit Testing* topic in *Visual Execution Analyzer in Enterprise Architect.*)

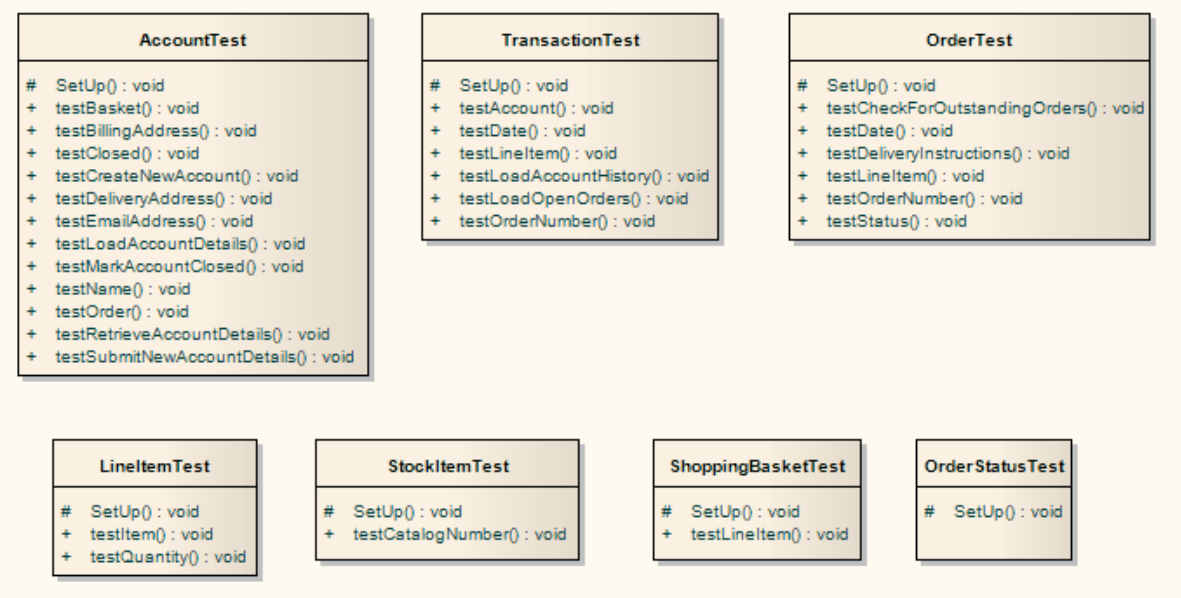
4.8 NUnit Transformation

The purpose of the **NUnit transformation** is to create a Class with test methods for all public methods of an existing .Net compatible Class. The resulting Class can then be generated and the tests filled out and run by NUnit.

The C# model originally transformed from the PIM:



After transformation becomes the PSM:

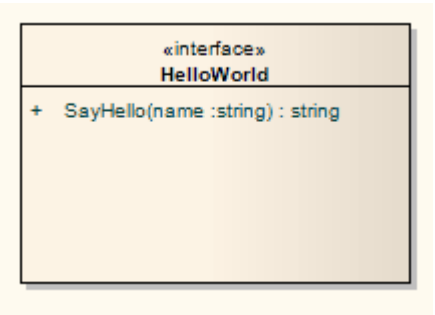


Note that for each Class in the C# model, a corresponding test Class has been created. Each of these test Classes contains a test method for every public method in the source Class, plus the methods required to appropriately set up the tests. It is your responsibility to fill in the details of each test. (See the *Unit Testing* topic in *Visual Execution Analyzer in Enterprise Architect.*)

4.9 WSDL Transformation

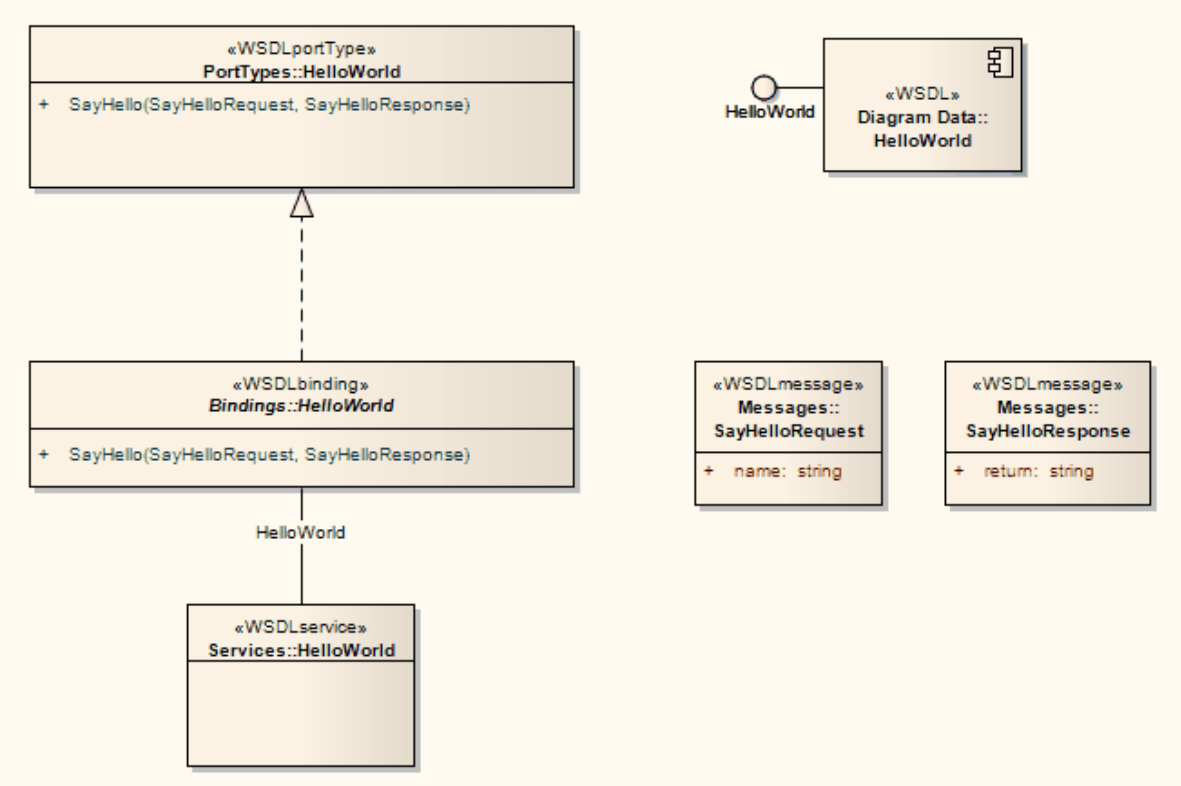
The purpose of the **WSDL transformation** is to create from a simple model an expanded model of a WSDL interface that is suitable for generation (see the *Model WSDL* topic in *Code Engineering Using UML Models*).

Take the following example interface:



This generates the corresponding WSDL component, service, port type, binding and messages as follows.

- Classes are handled in the same way as the XSD Transformation 31
- All in parameters are transformed into *messageParts* in the Request message
- The *return* value and all *out* and *return* parameters are transformed into *messageParts* in the Request message
- All methods where a value is returned are transformed into *Request-Response* operations while all methods not returning a value are transformed into *OneWay* operations
- The transformation does not handle generation of *Solicit-Response* and *Notification* methods or faults.



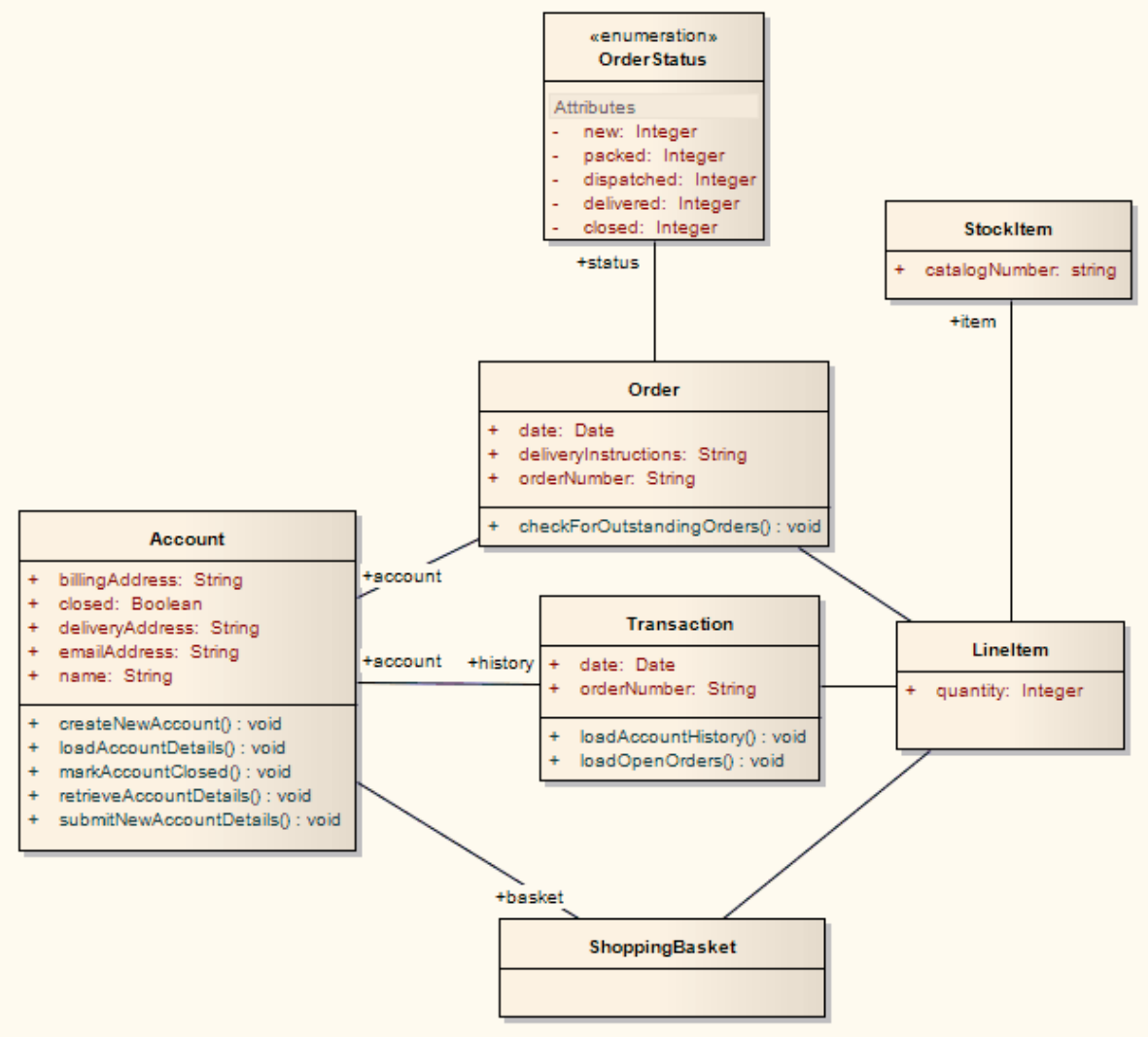
The resulting package can then have the specifics filled out using the WSDL editing capabilities of Enterprise Architect, and finally be generated using WSDL generation (see the *Generate WSDL* topic in *Code Engineering Using UML Models*).

4.10 XSD Transformation

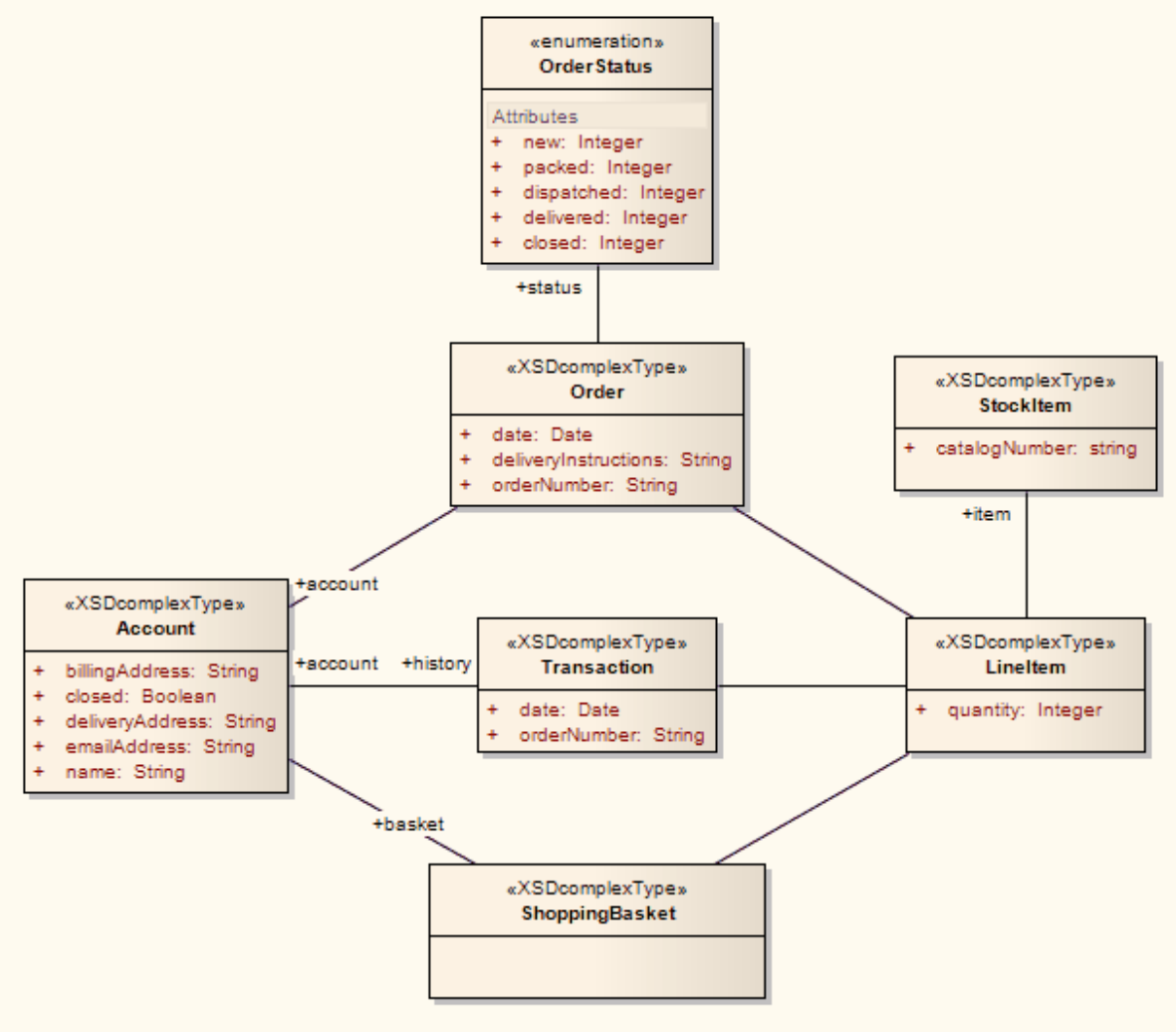
The purpose of the **XSD transformation** is to convert Platform-Independent Model (PIM) elements to UML Profile for XML elements as an intermediary step to creating an XML Schema. Each selected PIM Class element is converted to an «*XSDcomplexType*» element.

For more information, see the *XML Schema Generation* topic in *Code Engineering Using UML Models*.

The Platform-Independent Model (PIM):



After transformation becomes the PSM:



Which in turn generates the following XSD:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Account" type="Account"/>
    <xs:complexType name="Account">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="billingAddress" type="xs:string"/>
            <xs:element name="emailAddress" type="xs:string"/>
            <xs:element name="closed" type="xs:boolean"/>
            <xs:element name="deliveryAddress" type="xs:string"/>
            <xs:element ref="Order"/>
            <xs:element ref="ShoppingBasket"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="LineItem" type="LineItem"/>
    <xs:complexType name="LineItem">
        <xs:sequence>
            <xs:element name="quantity" type="xs:integer"/>
            <xs:element ref="StockItem"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Order" type="Order"/>
    <xs:complexType name="Order">
        <xs:sequence>
            <xs:element name="date" type="xs:date"/>
            <xs:element name="deliveryInstructions" type="xs:string"/>
```

```
                    <xs:element name="orderNumber" type="xs:string"/>
                    <xs:element ref="LineItem"/>
                    <xs:element name="status" type="OrderStatus"/>
                </xs:sequence>
        </xs:complexType>
        <xs:simpleType name="OrderStatus">
                <xs:restriction base="xs:string">
                    <xs:enumeration value="new"/>
                    <xs:enumeration value="packed"/>
                    <xs:enumeration value="dispatched"/>
                    <xs:enumeration value="delivered"/>
                    <xs:enumeration value="closed"/>
                </xs:restriction>
        </xs:simpleType>
        <xs:element name="ShoppingBasket" type="ShoppingBasket"/>
        <xs:complexType name="ShoppingBasket">
                <xs:sequence>
                    <xs:element ref="LineItem"/>
                </xs:sequence>
        </xs:complexType>
        <xs:element name="StockItem" type="StockItem"/>
        <xs:complexType name="StockItem">
                <xs:sequence>
                    <xs:element name="catalogNumber" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:element name="Transaction" type="Transaction"/>
        <xs:complexType name="Transaction">
                <xs:sequence>
                    <xs:element name="date" type="xs:date"/>
                    <xs:element name="orderNumber" type="xs:string"/>
                    <xs:element ref="Account"/>
                    <xs:element ref="LineItem"/>
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

5 Write Transformations

This topic provides help in writing your own transformations. Subjects covered are:

Further hints and tips can be gleaned from a close study of the Transformation Templates provided with Enterprise Architect. Note also that writing transformations is very similar to writing code generation templates, so an understanding of the Code Template Framework can greatly assist in understanding transformations. See the *Code Templates* section of *SDK for Enterprise Architect.*

The Transformation Template editor provides the facilities of the *Common Code Editor*. For more information on the Common Code Editor, see the *Code Editors* topic in *Using Enterprise Architect - UML Modeling Tool*.

Transformation Templates are accessed from the **Settings | Transformation Templates** menu option.

5.1 Default Transformation Templates

In most transformations, there is a lot of information that is simply copied to the target model. In order to make writing new transformations simpler Enterprise Architect provides a default set of transformation templates. These templates perform a simple copy of the source model to the target model. This means that in order to write a new transformation you can modify the default templates to make the required changes.

Note:

When creating a new transformation you must modify at least one template before the transformation becomes available.

5.2 Intermediary Language

All transformations in Enterprise Architect work by generating a text form of the model to generate.

Any element is represented in this language by the type of element (for example, Class, Action, Method, Generalization or Tag) followed by the properties of the element and the elements that it is made from. The grammar for this resembles the following.

```
element:
    elementName { (elementProperty | element)* }

elementProperty:
    packageName
    stereotype
    propertyName = '' propertyValueSymbol* ''

packageName:
    name = '' propertyValueSymbol* '' ( . '' propertyValueSymbol* '' )*
stereotype:
    stereotype = '' propertyValueSymbol* '' (, '' propertyValueSymbol* '' )*
```

propertyValueSymbol:
\\
\"
Any character except " (U+0022), \ (U+005C)

- *elementName* is any one of the set of element types as described in Objects 36 and Connectors 40
- *propertyName* is any one of the set of properties as described in Objects 36 and Connectors 40.

Literal strings can be included in property values by escaping a quote character. For example:

default = "\"Some string value.\""

5.3 Objects

Objects are created in Enterprise Architect by generating text in the following form:

```
objectType
{
  objectProperties
}
```

where:

objectType is one of the following object types:

- *Action*
- *ActionPin*
- *Activity*
- *ActivityParameter*
- *ActivityPartition*
- *ActivityRegion*
- *Actor*
- *Association*
- *Change*
- *Class*
- *Collaboration*
- *CollaborationOccurrence*
- *Component*
- *DeploymentSpecification*
- *DiagramFrame*
- *Decision*
- *EntryPoint*
- *Event*
- *ExceptionHandler*
- *ExecutionEnvironment*
- *ExitPoint*
- *ExpansionNode*
- *ExpansionRegion*
- *ExposedInterface*
- *GUIElement*
- *InteractionFragment*
- *InteractionOccurrence*
- *InteractionState*
- *Interface*
- *InterruptibleActivityRegion*
- *Issue*
- *Iteration*
- *Object*
- *ObjectNode*
- *MessageEndpoint*

- *Node*
- *Package*
- *Parameter*
- *Part*
- *Port*
- *ProvidedInterface*
- *RequiredInterface*
- *Requirement*
- *Sequence*
- *State*
- *StateMachine*
- *StateNode*
- *Synchronization*
- *Table*
- *TimeLine*
- *Trigger*
- *UMLDiagram*
- *UseCase*.

objectProperties is zero, or one or more of the following properties:

- *Abstract*
- *Alias*
- *Arguments*
- *Author*
- *Cardinality*
- *Classifier*
- *Complexity*
- *Concurrency*
- *Filename*
- *Header*
- *Import*
- *IsActive*
- *IsLeaf*
- *IsRoot*
- *IsSpecification*
- *Keyword*
- *Language*
- *Multiplicity*
- *Name*
- *Notes*
- *Persistence*
- *Phase*
- *Scope*
- *Status*
- *Stereotype*
- *Version*
- *Visibility*.

and zero, or one or more of the following elements:

- *Attribute*

- *Classifier*
- *Parameter*
- *Operation*
- *Parent*
- *Tag*
- *XRef*
- Any object.

Notes:

- Some of the above only apply to certain object types.
- Every object created in a transformation should include an XRef element 43, as it enables Enterprise Architect to synchronize with the element and makes it possible to create a connector to that Class in a transformation.

Classes

A simple Class can be created as follows:

```
Class
{
    name = "Example"
}
```

It is then easy to add to this. The following example sets the language to C++, and adds a Tagged Value and an attribute:

```
Class
{
    name = "Example"
    language = "C++"
    Tag
    {
            name = "defaultCollectionClass"
            value = "List"
    }
    Attribute
    {
            name = "count"
            type = "int"
    }
}
```

Attributes

Attributes are created with the same structure as objects, and include the following properties:

- *Alias*
- *Classifier*
- *Collection*
- *Container*
- *Containment*
- *Constant*
- *Default*
- *Derived*
- *LowerBound*
- *Name*
- *Notes*
- *Ordered*
- *Scope*
- *Static*
- *Stereotype*
- *Type*

- *UpperBound*
- *Volatile*.

and the following elements:

- *Classifier*
- *Tag*
- *XRef*.

Operations

Operations are created with the same structure as objects, and include the following properties:

- *Abstract*
- *Alias*
- *Behavior*
- *Classifier*
- *Code*
- *Constant*
- *IsQuery*
- *Name*
- *Notes*
- *Pure*
- *ReturnArray*
- *Scope*
- *Static*
- *Stereotype*
- *Type*.

and the following elements:

- *Classifier*
- *Parameter*
- *Tag*
- *XRef*.

Parameters

Parameters are created with the same structure as objects, and include the tag element and the following properties:

- *Classifier*
- *Default*
- *Fixed*
- *Name*
- *Notes*
- *Kind*
- *Stereotype*.

Packages

Packages differ from other objects in the following ways:

- A reduced set of properties of *alias*, *author*, *name*, *namespaceRoot*, *notes*, *scope*, *stereotype* and *version*
- The extra property *namespaceRoot*
- Must have a name specified
- *Name* can be a qualified name; when a qualified name is specified the properties given are applied only to the final package
- Can contain other packages
- Can't contain attributes and operations.

Tables

Tables are a special sort of object, with the following differences from other object types:

- Can include columns and primary keys
- Can't include attributes.

Columns

Columns are similar to attributes, but have an *autonumber* element containing *Startnum* and increment, and the following added properties:

- *Length*
- *NotNull*
- *Precision*
- *PrimaryKey*
- *Scale*
- *Unique*.

Note:

In the column definition, you cannot assign a value to the **NotNull**, **PrimaryKey** or **Unique** properties.

5.4 Connectors

Creating connectors in a transformation can be complex, but the process has the same form as creating elements. The difference is that you must also specify each end.

The different connectors that can be created are as follows.

- Aggregation
- Assembly
- Association
- Collaboration
- ControlFlow
- Connector
- Delegate
- Dependency
- Deployment
- ForeignKey
- Generalization
- InformationFlow
- Instantiation
- Interface
- InterruptFlow
- Manifest
- Nesting
- NoteLink
- ObjectFlow
- Package
- Realization
- Sequence
- Transition
- UseCase
- Uses

Note:

- *ForeignKey* is a special case where not just a connector is created; you must also list the columns involved in the transformation. In addition, tags specified on the connector are actually created on the foreign key operation in the source Class, and a cascade property can be added; for example, *cascade="update","delete".*
- Each connector is transformed at both end objects, therefore the connector might appear twice in the transformation. This is nothing to be concerned about, but you should check carefully that the connector is generated exactly the same way, regardless of which end is on the current Class.

There are two different types of Class that you can use as a connector end: one created by a transformation, and one for which you already know the GUID.

Connect to a Class Created by a Transformation

The most common connection is to connect to a Class created by a transformation. To do this you must have three items of information:

- The original Class GUID
- The name of the transformation
- The name of the transformed Class.

This type of connector is created using the **TRANSFORM_REFERENCE** 43 function macro. When the element is in the current transformation, it can be safely omitted from the transformation. The simplest example of this is when you have created multiple Classes from a single Class in a transformation and want a connector between them. Consider this example from the EJB Entity transformation:

```
Dependency
{
 %TRANSFORM_REFERENCE("EJBRealizeHome",classGUID)%
 stereotype="EJBRealizeHome"
 Source
 {
  %TRANSFORM_REFERENCE("EJBEntityBean",classGUID)%
 }
 Target
 {
  %TRANSFORM_REFERENCE("EJBHomeInterface",classGUID)%
 }
}
```

There are three uses of the **TRANSFORM_REFERENCE** macro: one to identify this connector for synchronization purposes and the other two to identify the ends. All three use the same source GUID, because they all come from the one original Class. None of the three have to specify the transformation because the two references are referencing something in the current transformation. Each of them then only has to identify the transform name.

Of course it is also possible to create a connector from another connector.

You can create a connector template and list over all connectors connected to a Class from the Class level templates. You don't have to worry about only generating it once, because if you have created a **TRANSFORM_REFERENCE** for the connector then Enterprise Architect automatically synchronizes them. The following copies the source connector.

```
%connectorType%
{
 %TRANSFORM_CURRENT()%
 %TRANSFORM_REFERENCE("Connector",connectorGUID)%
 Source
 {
  %TRANSFORM_REFERENCE("Class",connectorSourceGUID)%
  %TRANSFORM_CURRENT("Source")%
 }
 Target
 {
  %TRANSFORM_REFERENCE("Class",connectorDestGUID)%
  %TRANSFORM_CURRENT("Target")%
 }
}
```

Connecting to a Class For Which You Know the GUID

The second type of Class that you can use as a connector end is one for which you know the current GUID. To do this, specify the GUID of the target Class in either the source or target end. The following example creates a dependency from a Class created in a transformation, to the Class it was transformed from.

```
Dependency
{
 %TRANSFORM_REFERENCE("SourceDependency",classGUID)%
 stereotype="transformedFrom"
 Source
 {
  %TRANSFORM_REFERENCE("Class",classGUID)%
 }
 Target
 {
  GUID=%qt%%classGUID%%qt%
 }
}
```

5.5 Copy Information

In many transformations there is a substantial amount of information to be copied. It would be tedious to type all of the common information into a template so that it is copied to the transformed Class. The alternative is to use the **TRANSFORM_CURRENT** function macro to do exactly this.

- **TRANSFORM_CURRENT(<*listOfExcludedItems*>)** - Generates an exact copy of all the properties of the current item, except for the items named in <*listOfExcludedItems*>.

Another form of this is available when transforming connectors that enables either end of the connector to be copied:

- **TRANSFORM_CURRENT(<*connectorEnd*>,<*listOfExcludedItems*>)** - Generates an exact copy of the connector end specified by <*connectorEnd*> except for the items named in <*listOfExcludedItems*>, where <*connectorEnd*> is either *Source* or *Target*.

5.6 Convert Types

Different target platforms almost certainly require different types, so you often require a method of converting between types. The following macro offers this.

- **CONVERT_TYPE(<destinationLanguage>, <originalType>) -** Converts <*originalType*> to the corresponding type in <*destinationLanguage*> using the datatypes and common types defined in the model, where <*originalType*> is assumed to be a platform independent common type.

A similar macro is available when transforming common datatypes to the datatypes for a specified database.

- **CONVERT_DB_TYPE(<destinationDatabase>, <originalType>)** - Converts <*originalType*> to the corresponding datatypes in <*destinationDatabase*>, which is defined in the model. The <*originalType*> refers to a platform independent common datatype.

5.7 Convert Names

Different target platforms use different naming conventions. As a result you might not want to copy the names of your elements directly into the transformed models. To facilitate this requirement, Enterprise Architect's transformation templates provide a *CONVERT_NAME* 42 function macro.

Another way in which you can transform a name is to remove a prefix from the original name, with the *REMOVE_PREFIX* 43 macro.

CONVERT_NAME(<originalName>, <originalFormat>, <targetFormat>)

This macro converts <*originalName*>, which is assumed to be in <*originalFormat*>, to <*targetFormat*>.

The supported formats are:

- Camel Case: New words start with a capital letter *except* for the first word, which begins with a lower case

letter; for example, *myVariableTable*

- Pascal Case: Same as Camel Case but the first letter of the first word is upper case; for example, *MyVariableTable*
- Spaced: Words are separated by spaces; the case of letters is ignored
- Underscored: Words are separated by underscores; the case of letters is ignored.

Note:

Acronyms are not supported when converting from Camel Case or Pascal Case.

The original format might also specify a list of delimiters to be used. For example a value of ' _' breaks words whenever either a space or underscore is found.

The target format might also use a format string that specifies the case for each word and a delimiter between them. It takes the following form:

<firstWord>(<delimiter>)<otherWords>

- *<firstWord>* controls the case of the first word (see below)
- *<delimiter>* is the string generated between words
- *<otherWords>* applies to all words after the first word.

<firstWord> and *<otherWords>* are both a sequence of two characters. The first character represents the case of the first letter of that word, and the second character represents the case of all subsequent letters. An upper case letter forces the output to upper case, a lower case letter forces the output to lower case, and any other character preserves the original case.

Example 1: To capitalize the first letter of each word and separate multiple words with a space:

"Ht( )Ht" to output "My Variable Table"

Example 2: To generate the equivalent of Camel Case, but reverse the roles of upper and lower case; that is, all characters are upper case except for the first character of each word *after* the first word:

"HT( )hT" to output "MY vARIABLE tABLE"

REMOVE_PREFIX(<originalName>, <prefixes>)

This macro removes any prefix found in *<prefixes>* from *<originalName>*. The prefixes are specified in a semi-colon separated list.

The macro is often used in conjunction with the *CONVERT_NAME* macro. For example, this code creates a *get property name* according to the options for Java.

```
$propertyName=%REMOVE_PREFIX(attName,genOptPropertyPrefix)%
%if genOptGenCapitalisedProperties=="T"%
$propertyName=%CONVERT_NAME($propertyName, "camel case", "pascal case")%
%endIf%
```

5.8 Cross References

Cross References are an important part of transformations. They are used to:

- Find the transformed Class to synchronize with
- Create connectors between transformed Classes
- Specify a classifier of a type
- Determine where to transform to for future transformations.

Each cross reference has three different parts:

- A *Namespace*, corresponding to the transformation that generated the element
- A *Name*, which is a unique reference to something that can be generated in the above transformation
- A *Source*, which is the GUID of the element that this element was created from.

When writing the templates for a transformation, it is easiest to create the cross references using the **TRANSFORM_REFERENCE** macro that is defined for this purpose. It has three optional parameters.

TRANSFORM_REFERENCE(<name>, <sourceGuid>, <namespace>)

Generates a reference that can be used in the ways described above. It resembles the following.

```
XRef{namespace="<namespace>" name="<name>" source="<sourceGuid>"
```

Where:

- If <*name*> is not specified it gets the name of the current template
- If <*sourceGUID*> is not specified it gets the GUID of the current Class
- If <*namespace*> is not specified it gets the name of the current transformation.

Note:

The only time that this should be specified is when creating a connector to a Class created in a different transformation.

A good example of the use of cross references is in the DDL 15 templates provided with Enterprise Architect. In the Class template a cross reference is created with the name table. Then up to two different connectors can be created, each of which must identify the two Classes it connects using cross references while having its own unique cross reference.

Specify Classifiers

Objects, attributes, operations and parameters can all reference another element in the model as their type. When this type is created from a transformation you must use a cross reference to specify it, using the **TRANSFORM_CLASSIFIER** macro.

TRANSFORM_CLASSIFIER(<name>, <sourceGuid>, <namespace>)

Generates a cross reference within a classifier element, where the parameters are identical to the **TRANSFORM_REFERENCE** macro but the name *Classifier* is generated instead of *XRef*.

If the target classifier already exists in the model before the transformation, a **TRANSFORM_CLASSIFIER** is inappropriate and instead the GUID can be given directly to a classifier attribute.

Note:

If a classifier is specified for any type it overrides the type specified.

Index

- B -

- C -

- D -

- E -

- I -

- J -

- M -

- N -

- O -

- P -

- R -

- T -

- W -

- X -

MDA Transformations User Guide

www.sparxsystems.com