# Model Transformation

Sparx Systems Enterprise Architect supports Model Driven Architecture (MDA) transformations of a single element or a whole Platform Independent Model (PIM) to one or more output Platform Specific Models (PSM) in, for example, Oracle, MySQL or SQLite.

CREATED WITH ENTERPRISE ARCHITECT

# Table of Contents

# Model Transformation

One of the great advantages of creating models is the ability to manipulate them to produce outputs, thus saving time and reducing the possibility of errors. Enterprise Architect implements Model Driven Architecture (MDA) transformations using a flexible and fully configurable template system. The templates act as instructions to a machine that takes a model as input and transforms it to a more resolved model as output. The input could be a large and complex model or a single element and one input model could be transformed to a variety of output models.

The transformations are commonly unidirectional and take a Platform Independent Model (PIM) and transform it to one or more Platform Specific Models (PSM). A good example of where this is useful is where a system must be implemented in a number of different relational database systems. A single platform independent conceptual model (the PIM) could be transformed to a number of platform specific models, say Oracle, MySQL and SQLite. As a further productivity boost, once the output models are produced they can also be converted to programming code, database definition language or schemas. Enterprise Architect automatically creates traceability that can be used to visualize how elements in the input model have been transformed to elements in the output model.

## Facilities

| Facility | Description |
|---|---|
| Transform Elements | Discover how to transform elements on a diagram or from a Browser window Package. |
| Built-in Transformations | Enterprise Architect provides a number of built-in transformations that support a wide range of target languages. Each is fully customizable to your specific needs. |
| Edit Transformation Templates | Learn how to adjust the transformation templates to produce transformations specific to your system. |
| Write Transformations | All the information you will need to create your own transformations. |

## Ready-built Transformations

The Enterprise Architect installer includes a number of basic built-in transformations, including:

- PIM to:
    - C#
    - C++
    - DDL table elements
    - EJB Entity Bean

- EJB Session Bean
- Java
- PHP
- VB.Net
- XSD

- Data Model to Entity Relationship diagram (ERD)

- Entity Relationship diagram (ERD) to Data Model

- Sequence diagram to Communication diagram

- Communication diagram to Sequence diagram

- Java model to JUnit test model

- .NET model to NUnit test model

- WSDL interface model to WSDL

Further transformations will become available over time, either built in or as downloadable modules from the Sparx Systems website.

## Customized Transformations

You can modify the built-in transformations or define your own, using Enterprise Architect's simple code generation template language. This involves little more than writing templates to create a simple intermediary source file; the system reads the source file and binds that to the new PSM.

## Transformation Dependencies

When you execute a transformation, the system creates internal bindings (Transformation Dependencies) between each PSM created and the original PIM. This is essential, providing the ability 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, 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.

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.

## Example of a Transformation

This diagram highlights how transformations work and how they can significantly boost your productivity.

**DDL**

- + Account
- + LineItem
- + Order
- + OrderStatus
- + ShoppingBasket
- + StockItem
- + Transaction

*(from Implementation Model (PSM))*

**Platform Independent model**

«create»

**Java Model**

- + Account
- + LineItem
- + Order
- + OrderStatus
- + ShoppingBasket
- + StockItem
- + Transaction

*(from Implementation Model (PSM))*

**Abstract Class Model (PIM)**

- + Account
- + LineItem
- + Order
- + OrderStatus
- + ShoppingBasket
- + StockItem
- + Transaction

«create»

**Multiple target domains generated from single PIM**

**C# Model**

- + Account
- + LineItem
- + Order
- + OrderStatus
- + ShoppingBasket
- + Transaction

*(from Implementation Model (PSM))*

«create»

«create»

**EJB Session**

- + Account
- + LineItem
- + Order
- + OrderStatus
- + ShoppingBasket
- + StockItem
- + Transaction

*(from Implementation Model (PSM))*

## Notes

- If you are using the Corporate, Unified or Ultimate edition, if security is enabled you must have 'Transform Package' access permission to perform an MDA Transformation on the elements of a Package

# Transform Elements

A model transformation is a user initiated function that starts the process of transforming one or more Platform Independent Model (PIM) elements into their corresponding Platform Specific Model (PSM) elements. This process takes place in accordance with the rules that have been codified in the Transformation Templates. The transformation can be initiated by selecting a Package in the Browser window or an element in a diagram.

## Access

| Ribbon | Design > Tools > Transform > Transform Selection |
|---|---|
| Keyboard Shortcuts | Ctrl+H        (transform selected elements) |
| | Ctrl+Shift+H    (transform selected Package) |

## Perform a Transformation

| Option | Action |
|---|---|
| Elements | Lists all of the individual elements selected in the diagram or held in the Package. Either: <br>• Click on an element to include just that element in the transformation <br>• Hold Ctrl and click on several separate elements to include them in the transformation, or <br>• Hold Shift and click on the first and last elements in a block to include those elements in the transformation |
| All | Click on this button to select all of the elements in the list to include them in the transformation. |
| None | Click on this button to deselect all of the elements in the list. |
| Include child packages | (If you have selected to transform elements in a Package.) <br>Select this checkbox to include (in the 'Elements' list and potentially in the transformation) elements from the child Packages of the selected Package. |
| Transformations | Select the checkbox against each type of transformation to perform. When you select a checkbox, the 'Browse Project' dialog displays; locate and select the target Package into which to generate the transformed elements. <br>If you want to change a selected target Package, click on the ... button to the right of the Package name and select the new Package from the dialog. |
| Generate Code on result | Select this checkbox to specify whether or not to automatically generate code for transformed Classes that target code languages. <br>If you select this option, the first time you transform to the Class the system prompts you to select a filename to generate code into; subsequent transformations |

| | |
|---|---|
| | automatically generate code to that filename. |
| Perform Transformations on result | Select the checkbox to automatically execute transformations previously done on the target Class or Classes. |
| Intermediary File | If you want to capture the intermediary language file (for example, to debug it), either type in the file path and name or click on the ⬚ button and locate and select the file. |
| Write Always | Select this checkbox to always write the intermediary file to disk. |
| Write Now | Click on this button to generate the intermediary file without performing the full transformation. |
| Do Transform | Click on this button to execute the transformation. When the transformation is complete, the 'Model Transformation' dialog closes. |
| Close | Click on this button to close the 'Model Transformation' dialog without performing the transformation. |

## Notes

- When the dialog displays, all elements are selected and all transformations previously performed from any of these Classes are checked
- This procedure does not apply to the Sequence diagram/Communication diagram transformation, or the Communication diagram/Sequence diagram transformation

# Chaining Transformations

Chaining transformations provides an extra degree of flexibility and power to performing transformations. For example, if two transformations have a common element; you might separate this element out into its own transformation, and then perform the original transformations from the common point. The separated transform could even produce a useful model itself.

You can chain transformations by selecting the 'Perform Transformations on result' checkbox in the 'Model Transformation' dialog, so that transformations that have already been performed on target Classes are executed automatically next time that Class is transformed to.

# Built-in Transformations

Enterprise Architect provides a rich set of built-in, commonly performed transformations. These will prove useful to a variety of disciplines from Domain Modeling to Code Engineering. The facility to transform models is a powerful productivity tool and it is expected that modelers will want to create their own custom transformations. The built-in transformations provide useful examples and are a valuable reference for the modeler.

## Built-in Transformations

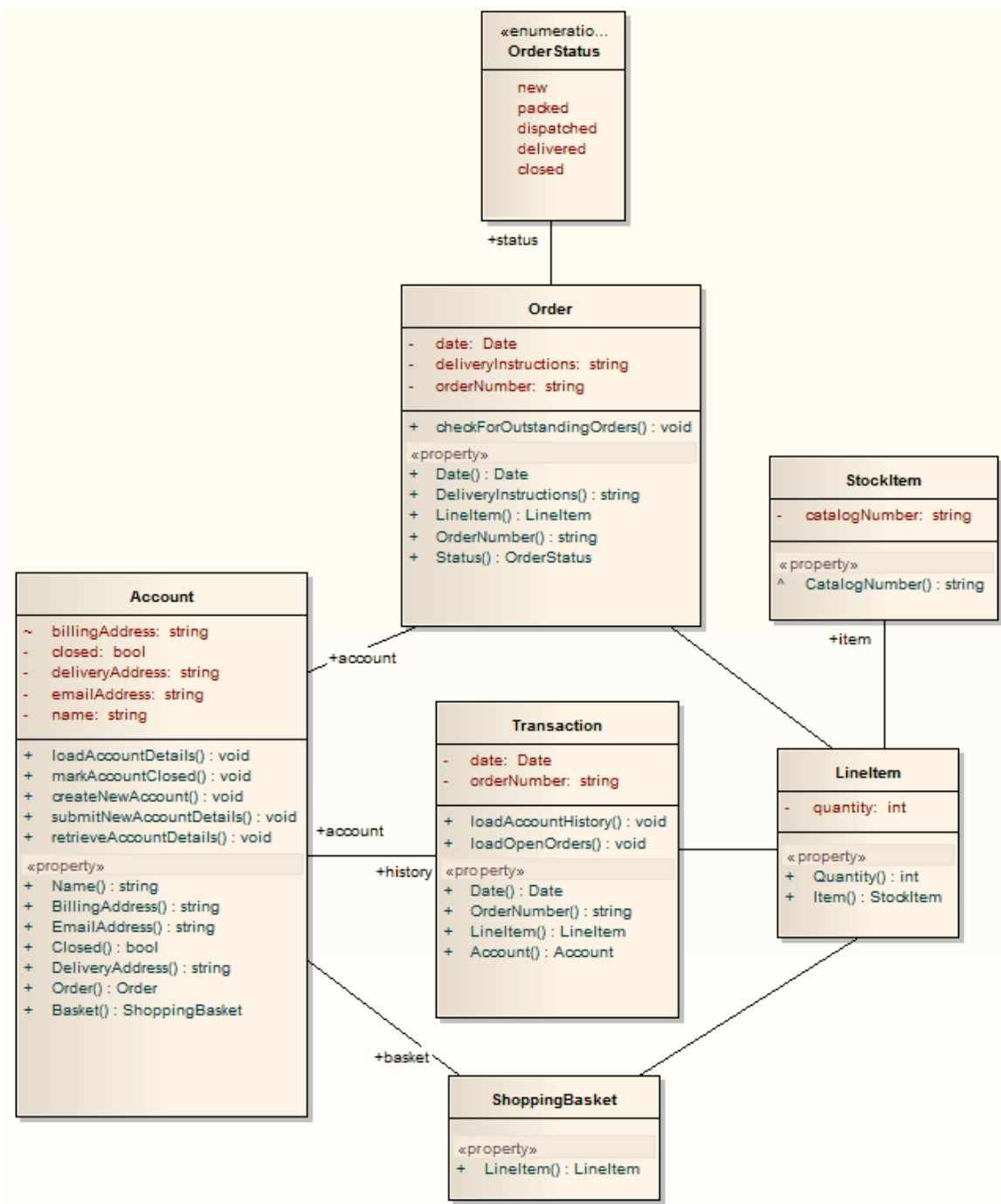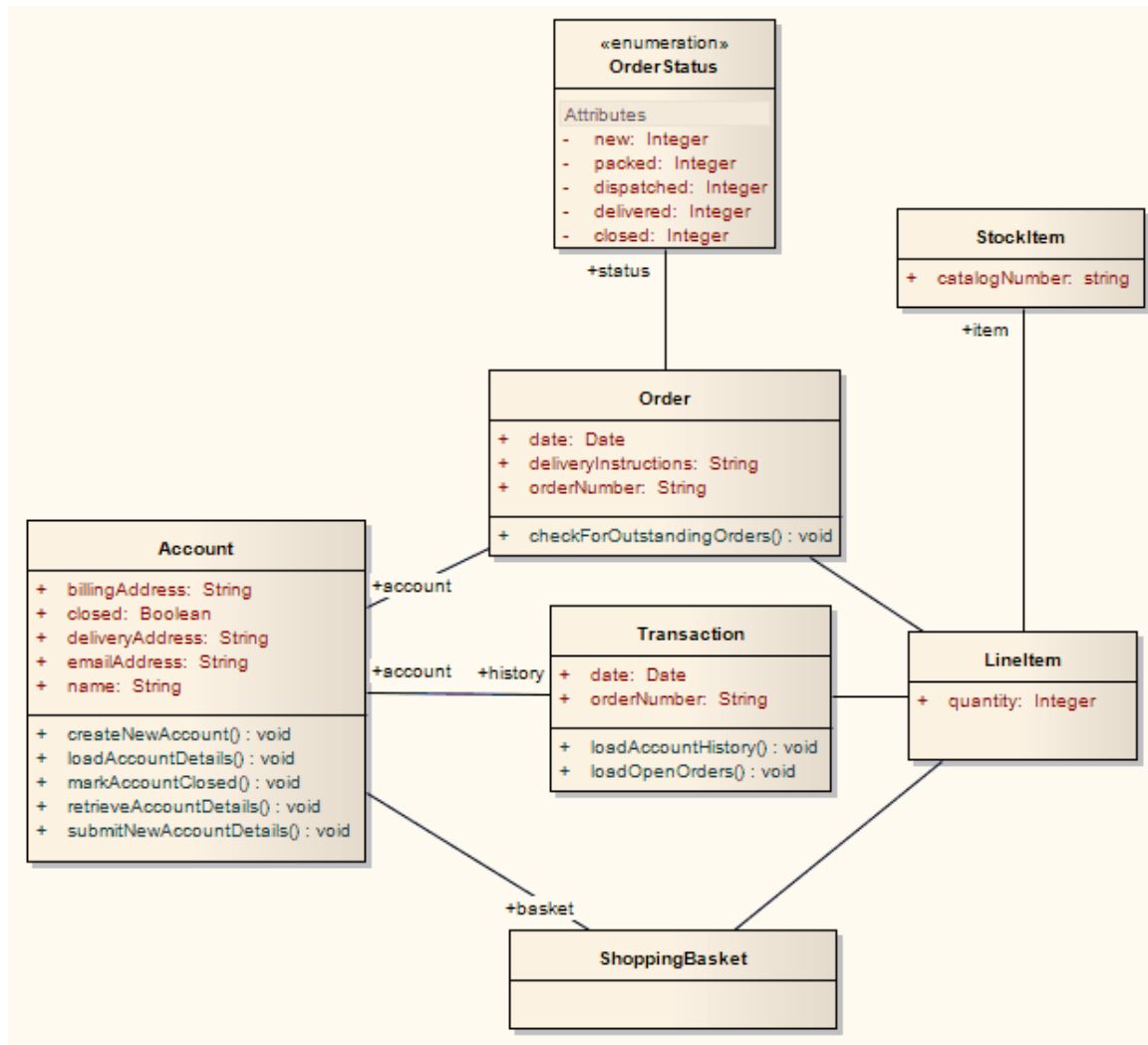| Transformation | Converts |
| --- | --- |
| C# | Platform-Independent Model (PIM) elements to language-specific C# Class elements. |
| C++ | PIM elements to language-specific C++ Class elements. |
| Data Definition Language | A logical model to a data model targeted at the default database type, ready for DDL generation. |
| Entity Relationship Diagram to Data Model | An ERD logical model to a data model targeted at the default database type, ready for DDL generation. |
| Data Model to Entity Relationship Diagram | A data model to an ERD logical model. |
| EJB Session Bean | A single Class element to the elements of an EJB session. |
| EJB Entity Bean | A single Class element to the elements of an EJB entity. |
| Java | PIM elements to language-specific Java Class elements. |
| JUnit | An existing Java Class element with public methods to a Class with a test method for each public method, plus the methods required to appropriately set up the tests. |
| NUnit | An existing .NET compatible Class with public methods to a Class with a test method for each public method, plus the methods required to appropriately set up the tests. |
| PHP | PIM elements to language-specific PHP Class elements. |
| Sequence/Communication Diagram | All elements and messages in an opened Sequence diagram into matching elements and messages in a Communication diagram, and vice versa. |
| VB.Net | PIM elements to language-specific VB.Net Class elements. |
| WSDL | A simple model to an expanded model of a WSDL interface, suitable for generation. |
| XSD | PIM elements to UML Profile for XML elements, as an intermediary step in creating an XML Schema. |

# C# Transformation

The C# transformation converts Platform-Independent Model (PIM) element types to C#-specific Class element types, and creates encapsulation according to the system options you have set for creating properties from C# attributes (on the 'C# Specifications' page of the 'Preferences' dialog).

## Example

The PIM elements



After transformation, become the PSM elements

**«enumeratio...**
**Order Status**

new
packed
dispatched
delivered
closed

+status

**Order**

- date: Date
- deliveryInstructions: string
- orderNumber: string

+ checkForOutstandingOrders() : void

«property»
+ Date() : Date
+ DeliveryInstructions() : string
+ LineItem() : LineItem
+ OrderNumber() : string
+ Status() : OrderStatus

**StockItem**

- catalogNumber: string

«property»
^ CatalogNumber() : string

+item

**Account**

~ billingAddress: string
- closed: bool
- deliveryAddress: string
- emailAddress: string
- name: string

+ loadAccountDetails() : void
+ markAccountClosed() : void
+ createNewAccount() : void
+ submitNewAccountDetails() : void
+ retrieveAccountDetails() : void

«property»
+ Name() : string
+ BillingAddress() : string
+ EmailAddress() : string
+ Closed() : bool
+ DeliveryAddress() : string
+ Order() : Order
+ Basket() : ShoppingBasket

+account

**Transaction**

- date: Date
- orderNumber: string

+ loadAccountHistory() : void
+ loadOpenOrders() : void

«property»
+ Date() : Date
+ OrderNumber() : string
+ LineItem() : LineItem
+ Account() : Account

+account

+history

**LineItem**

- quantity: int

«property»
+ Quantity() : int
+ Item() : StockItem

+basket

**ShoppingBasket**

«property»
+ LineItem() : LineItem
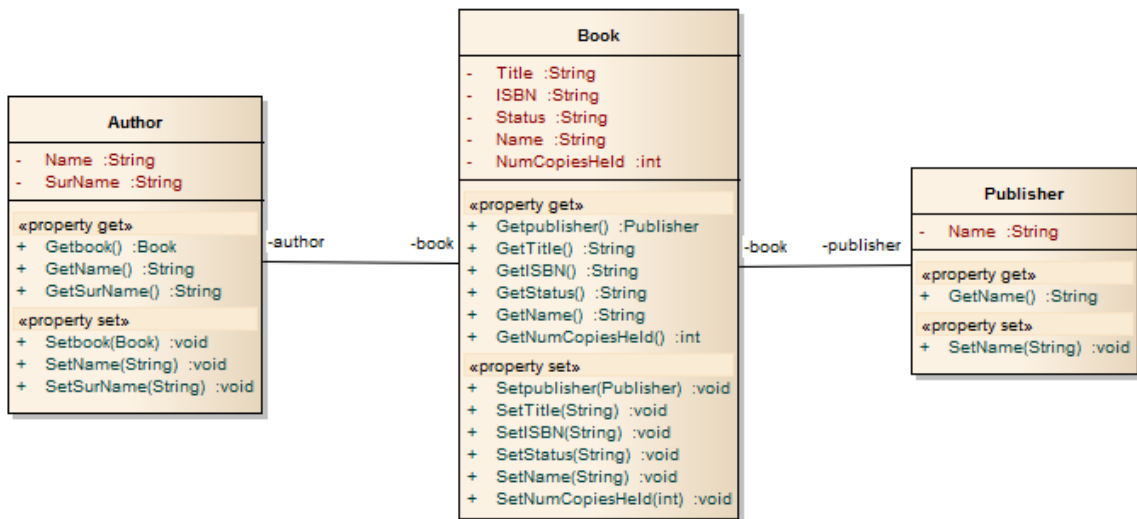
# C++ Transformation

The C++ transformation converts Platform-Independent Model (PIM) element types to C++ specific Class element types and creates encapsulation (producing the getters and setters) according to the options you have set for creating properties from C++ attributes (on the 'C++ Specifications' page of the 'Preferences' dialog). Note that the public attributes in the PIM are converted to private attributes in the PSM. All operations on an interface are transformed into pure virtual methods on an equivalent class.

## Example

The PIM elements
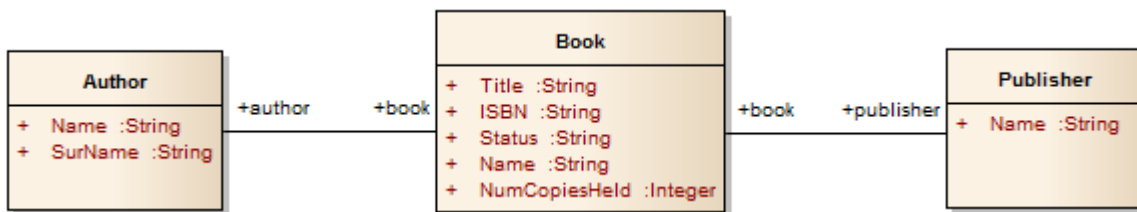
After transformation, become the PSM elements

# Data Model To ERD Transformation

The Data Model to Entity Relationship diagram (ERD) transformation creates an ERD logical model from a Data Model. It is the reverse of the ERD to Data Model transformation. This transformation uses and demonstrates support in the intermediary language for a number of database-specific concepts.

## Supported Concepts

| Concept | Effect |
|---|---|
| Entity | Mapped one-to-one onto Table elements. |
| Attribute | Mapped one-to-one onto Columns. |
| Primary Key | Derived from the PrimaryKey type of column. |

## Notes

- Sometimes you might want to limit the stretch of the diamond-shaped Relationship connectors; simply pick a Relationship connector, right-click to display the context menu, and select the 'Bend Line at Cursor' option

# DDL Transformation

The DDL transformation converts the logical model to a data model structured to conform to one of the supported DBMSs. The target database type is determined by which DBMS is set as the default database in the model. The data model can then be used to automatically generate DDL statements to run in one of the system-supported database products.

The DDL transformation uses and demonstrates support in the intermediary language for a number of database-specific concepts.

## Concepts

| Concept | Effect |
|---------|--------|
| Table | Mapped one-to-one onto Class elements. <br> 'Many-to-many' relationships are supported by the transformation, creating Join tables. |
| Column | Mapped one-to-one onto attributes. |
| Primary Key | Lists all the columns involved so that they exist in the Class, and creates a Primary Key Method for them. |
| Foreign Key | A special sort of connector, in which the Source and Target sections list all of the columns involved so that: <br> • The columns exist <br> • A matching Primary Key exists in the destination Class, and <br> • The transformation creates the appropriate Foreign Key |

## MDG Technology to customize default mappings

DDL transformations that target a new, user defined DBMS require an MDG Technology to map the PIM data types to the new target DBMS.

To do this, create an MDG Technology .xml file named 'UserDBMS Types.xml', replacing UserDBMS with the name of the added DBMS. Place the file in the EA\MDGTechnologies folder.  The contents of the MDG Technology file should have this structure:

```
<MDG.Technology version="1.0">
    <Documentation id="UserdataTypes" name="Userdata Types" version="1.0" notes="DB Type mapping for UserDBMS"/>
    <CodeModules>
        <CodeModule language="Userdata" notes="">
            <CodeOptions>
                <CodeOption name="DBTypeMapping-bigint">BIGINT</CodeOption>
                <CodeOption name="DBTypeMapping-blob">BLOB</CodeOption>
                <CodeOption name="DBTypeMapping-boolean">TINYINT</CodeOption>
```

```
            <CodeOption name="DBTypeMapping-text">CLOB</CodeOption>

            ...
        </CodeOptions>

    </CodeModule>

</CodeModules>

</MDG.Technology>
```
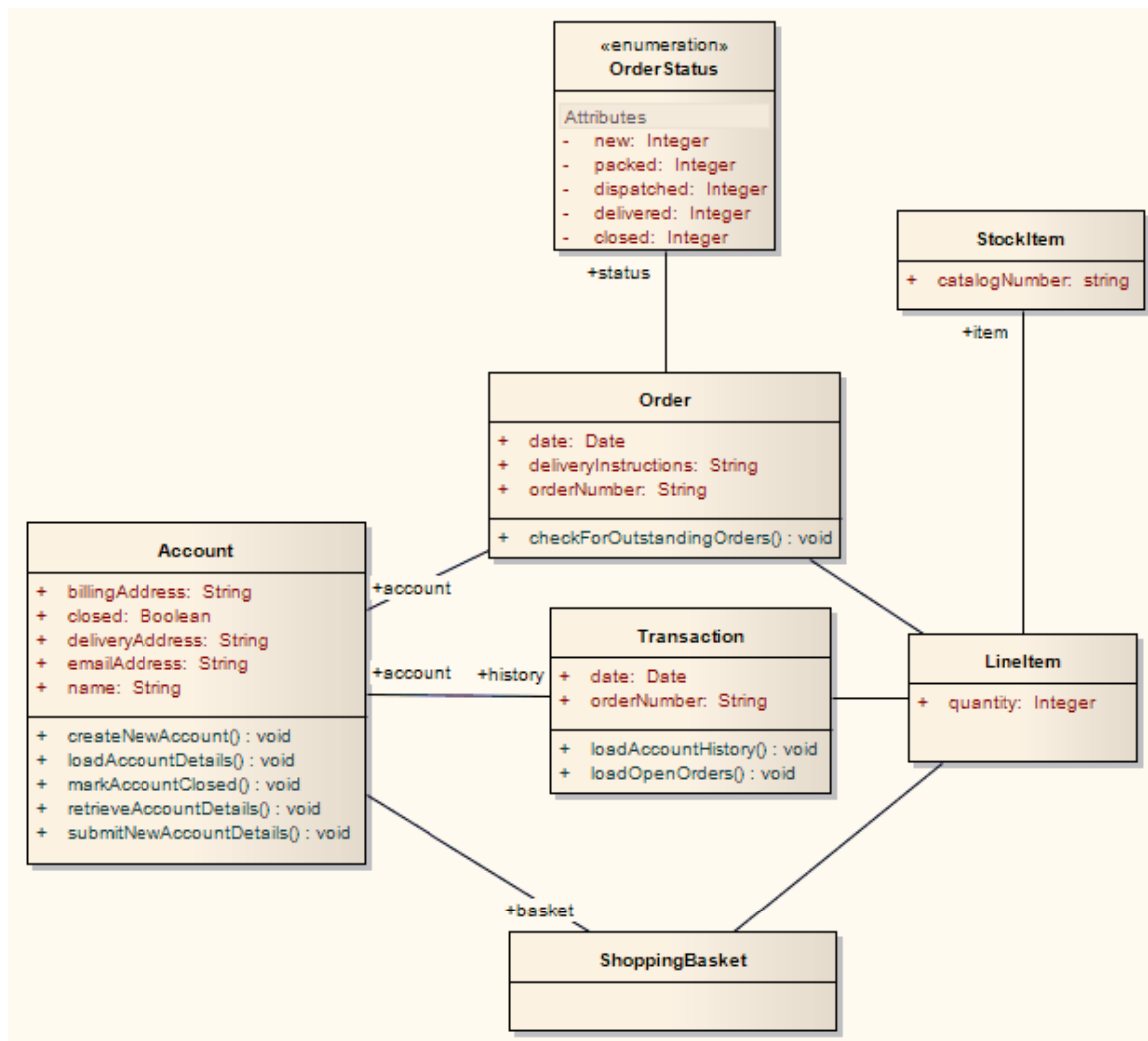
As an example, 'text' is a Common Type (as listed in the 'Database Datatypes' dialog) that maps to a new DBMS's 'CLOB' data type.

## Notes

- You can define DBMS-specific aspects not depicted in a Logical model, such as Stored Procedures, Triggers, Views and Check Constraints, after the transformation; see the Physical Data Model topic
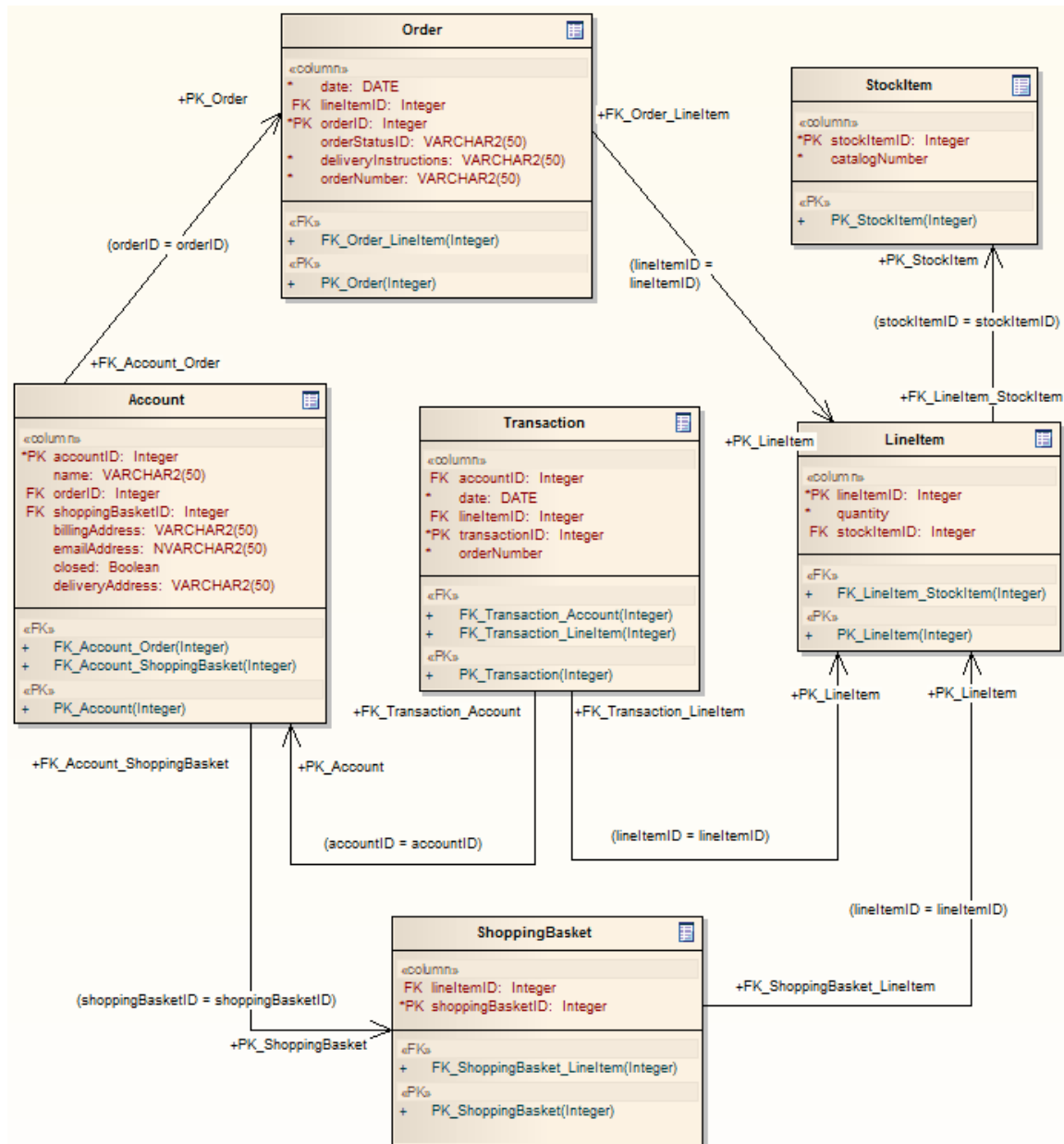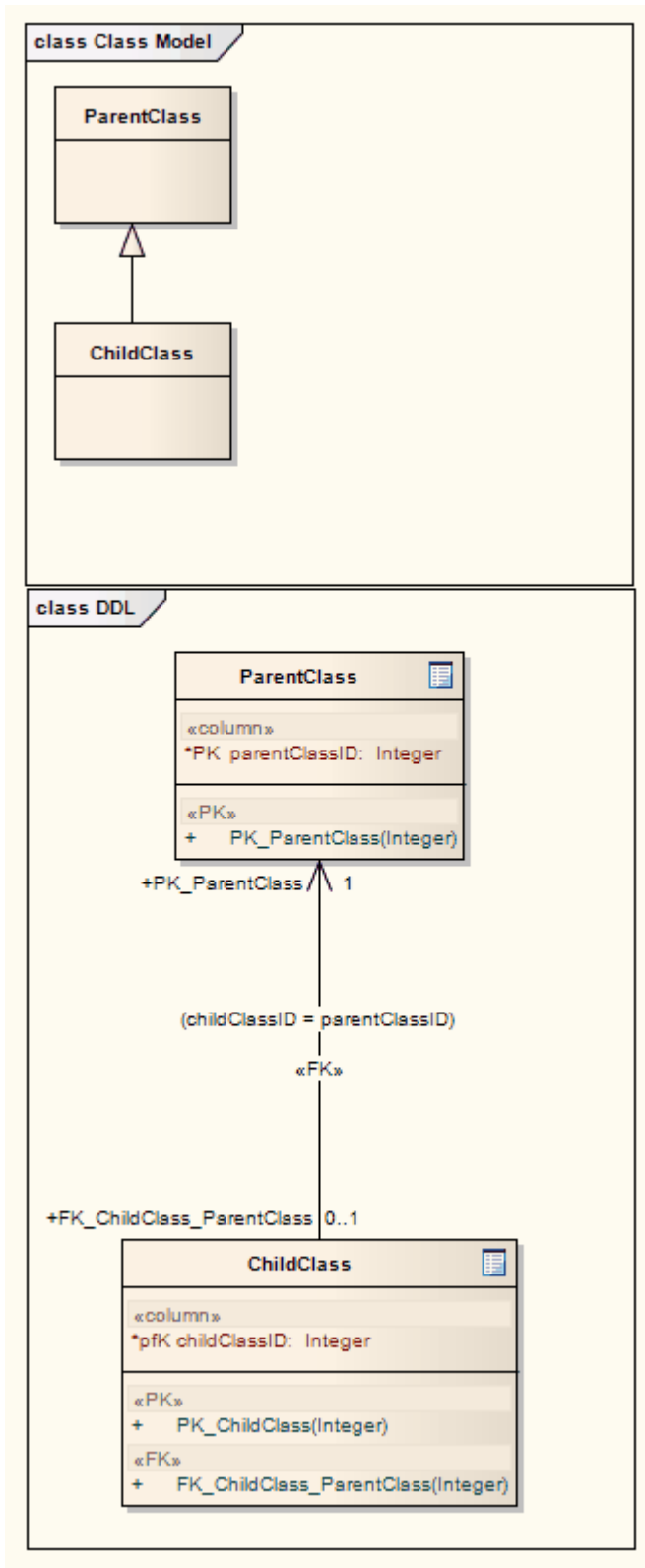
## Example

The PIM elements

After transformation, become the PSM elements



Generalizations are handled by providing the child element with a Foreign Key to the parent element, as shown. Copy-down inheritance is not supported.

class Class Model

ParentClass

ChildClass

class DDL

**ParentClass**

«column»
*PK parentClassID: Integer

«PK»
+        PK_ParentClass(Integer)

+PK_ParentClass     1

(childClassID = parentClassID)

«FK»

+FK_ChildClass_ParentClass 0..1

**ChildClass**

«column»
*pfK childClassID: Integer

«PK»
+        PK_ChildClass(Integer)
«FK»
+        FK_ChildClass_ParentClass(Integer)

# EJB Transformations

The EJB Session Bean and EJB Entity Bean transformations reduce the work required to generate the internals of Enterprise Java Beans. You can therefore focus on modeling at a higher level of abstraction.
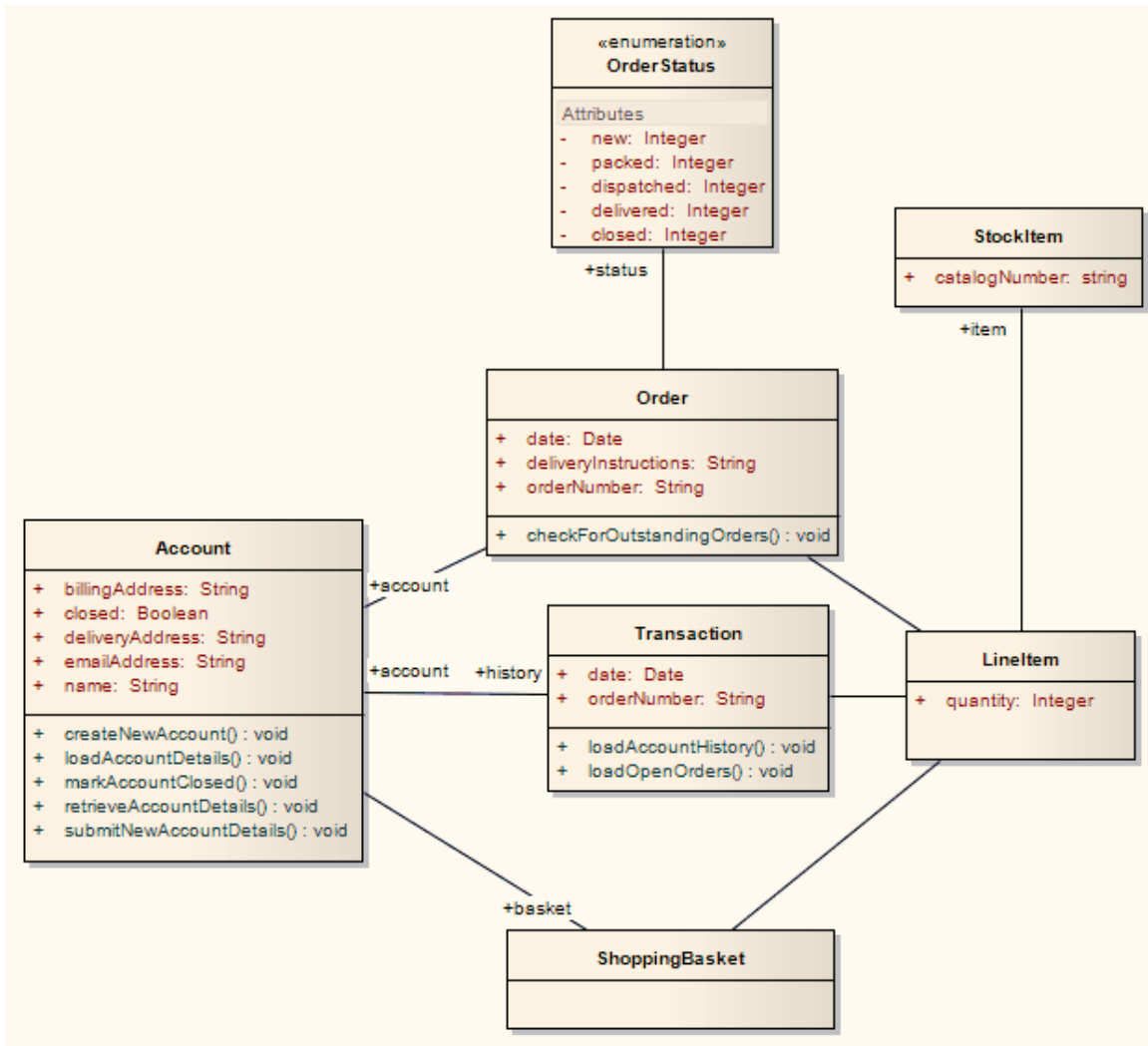
## Transformations

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

| Transformation | Detail |
|---|---|
| EJB Session Bean | This transformation converts a single Class element (containing the attributes, operations and references required for code generation by the javax.ejb.* Package) to<br>• An implementation Class element<br>• A home interface element<br>• A remote interface element |
| EJB Entity Bean | This transformation converts a single Class element (containing the attributes, operations and references required for code generation by the javax.ejb.* Package) to:<br>• An implementation Class element<br>• A home interface element<br>• A remote interface element<br>• A Primary Key element |

## Example

The PIM elements

After transformation generate a set of Entity Beans, where each one takes this form (for the Account Class):
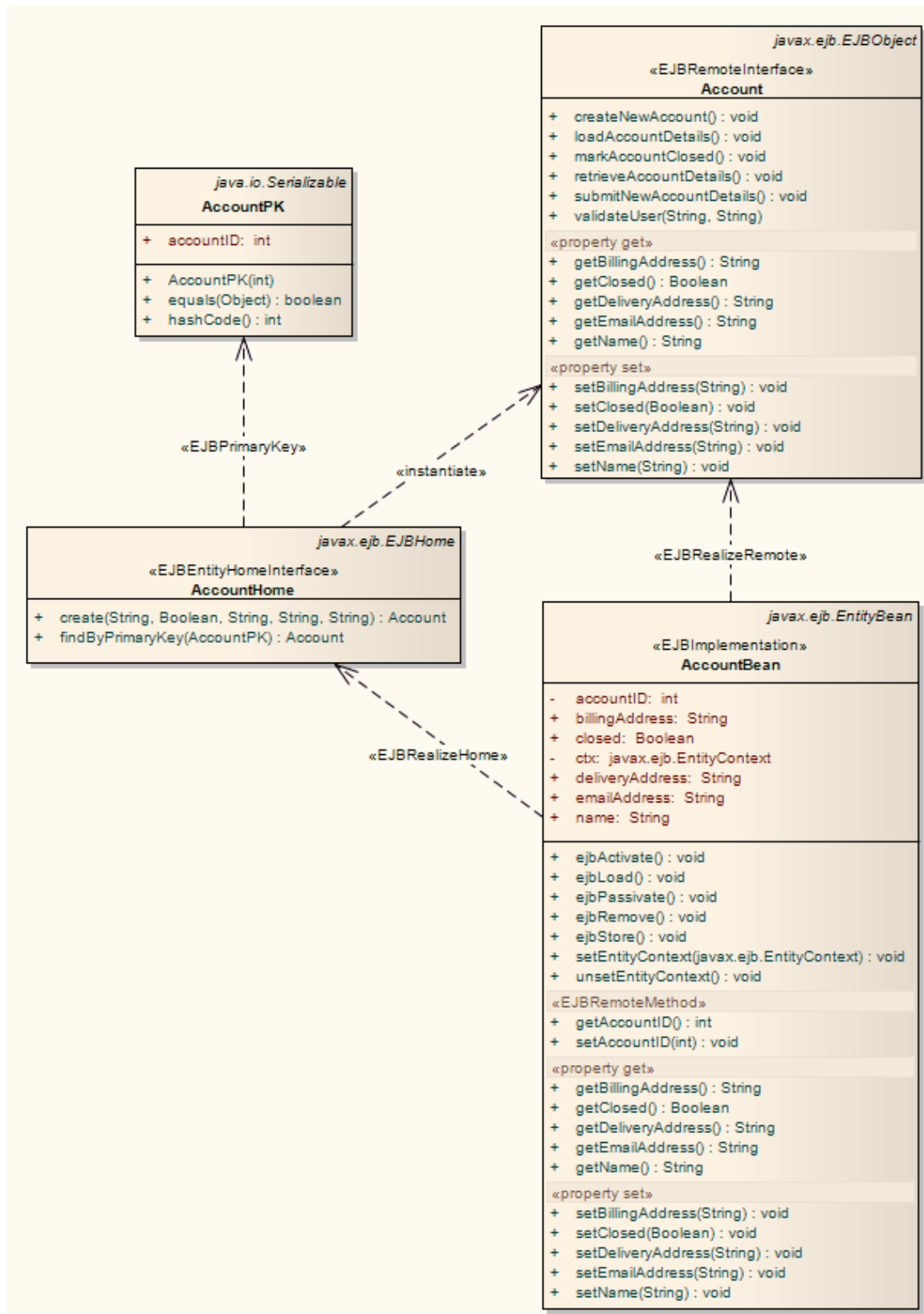
# ERD To Data Model Transformation

The Entity Relationship Diagram (ERD) to Data Model transformation converts an ERD logical model to a data model targeted at the default database type, ready for generating DDL statements to run in one of the system-supported database products. Before doing the transformation, you define the common data type for each attribute and select a database type as the default database. You can then automatically generate the Data Modeling diagram.

The transformation uses and demonstrates support in the intermediary language for a number of database-specific concepts.

## Concepts

| Concept | Definition |
|---|---|
| Table | Mapped one-to-one onto Class elements. |
| Column | Mapped one-to-one onto attributes. |
| Primary Key | Lists all the columns involved so that they exist in the Class, and creates a Primary Key Method for them. |
| Foreign Key | A special sort of connector, in which the Source and Target sections list all of the columns involved so that:<br>• The columns exist<br>• A matching Primary Key exists in the destination Class, and<br>• The transformation creates the appropriate Foreign Key |

## Generalization

ERD technology can handle Generalization, as shown. Note that the copy-down inheritance is currently supported with two levels only.

## Example

The ERD elements
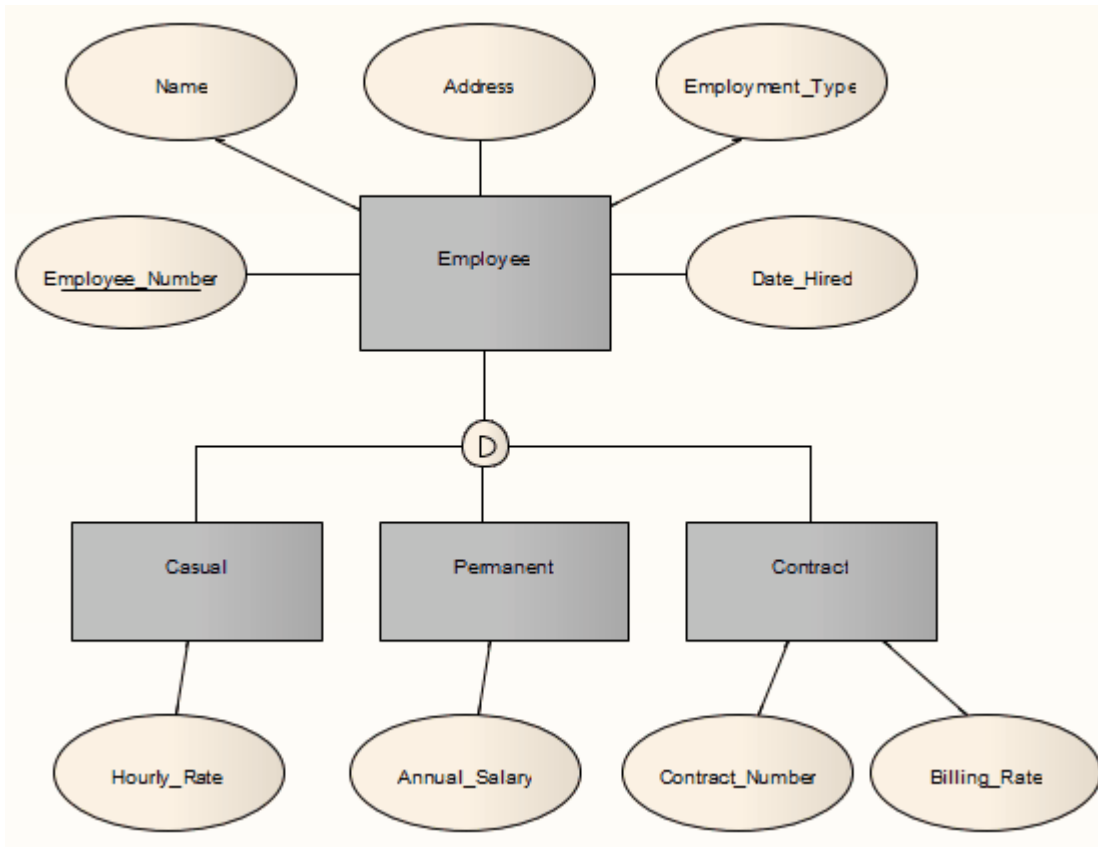
After transformation, become the Data Model elements
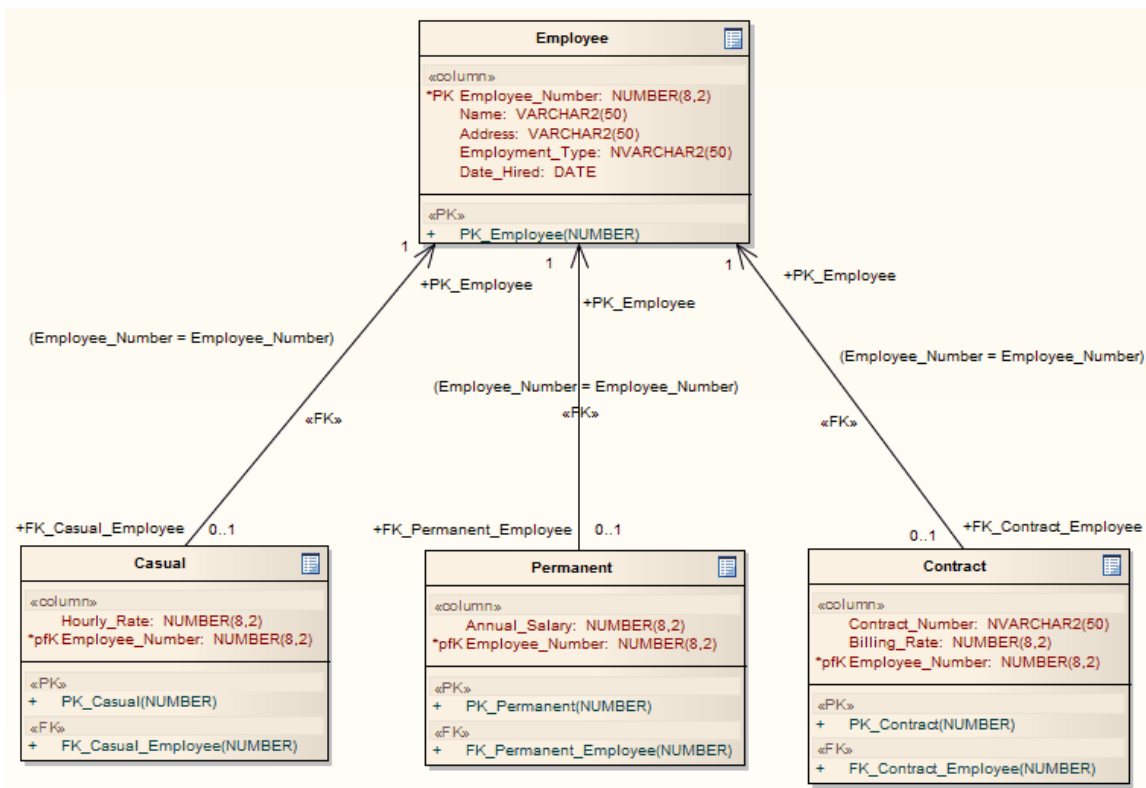


## Notes

- Sometimes you might go back to the ERD, make some changes and then need to do another transformation; in this case, to achieve better results, always delete the previous transformation Package before doing the next transformation

# Java Transformation

The Java transformation converts Platform-Independent Model (PIM) element types to Java-specific Class element types, and creates encapsulation (producing the getters and setters) according to the options you have set for creating properties from Java attributes (on the 'Java Specifications' page of the 'Preferences' dialog). Note that the public attributes in the PIM are converted to private attributes in the PSM. All operations in the interface are transformed into pure virtual methods.

## Example

The PIM elements
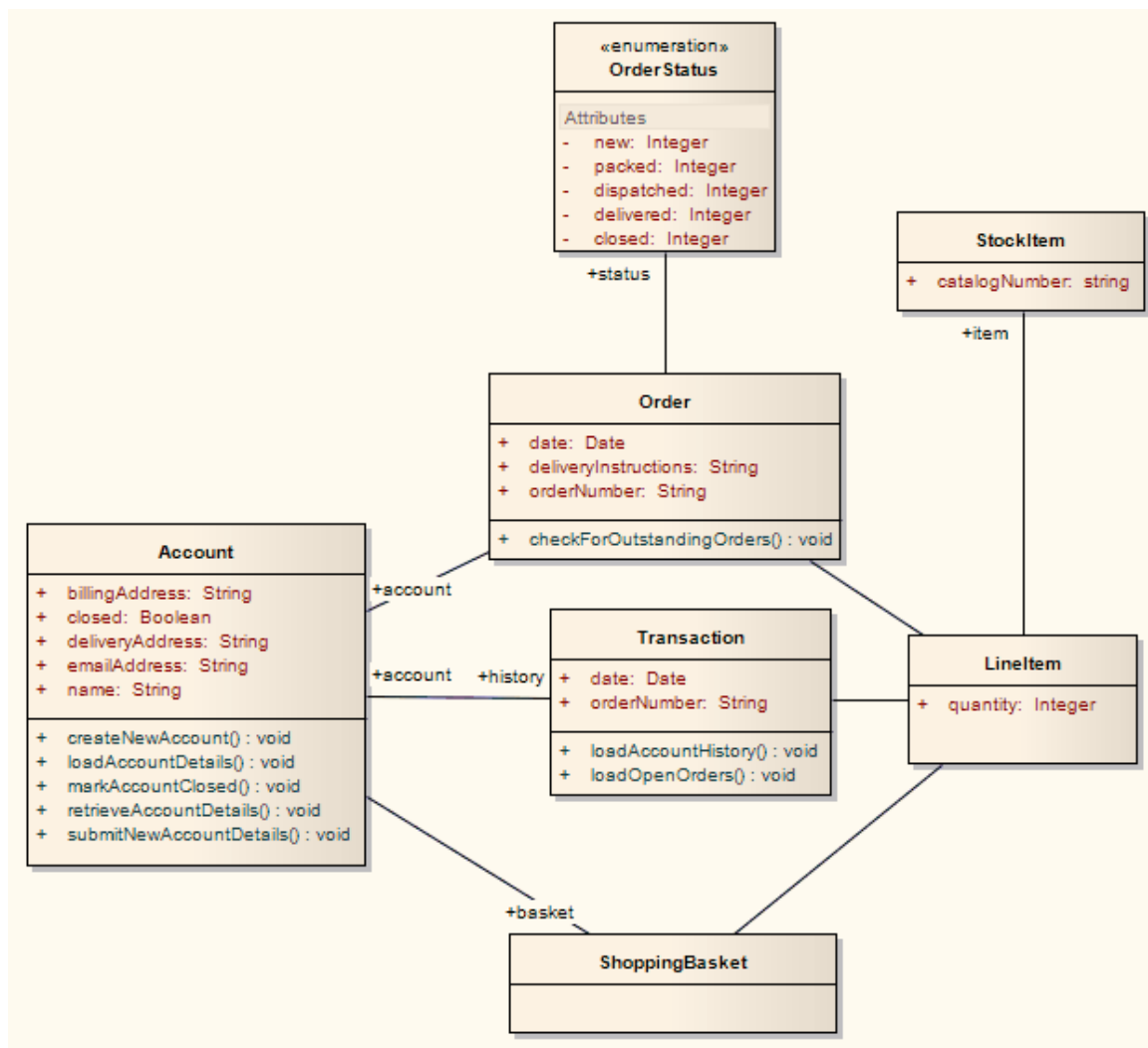


After transformation, become the PSM elements
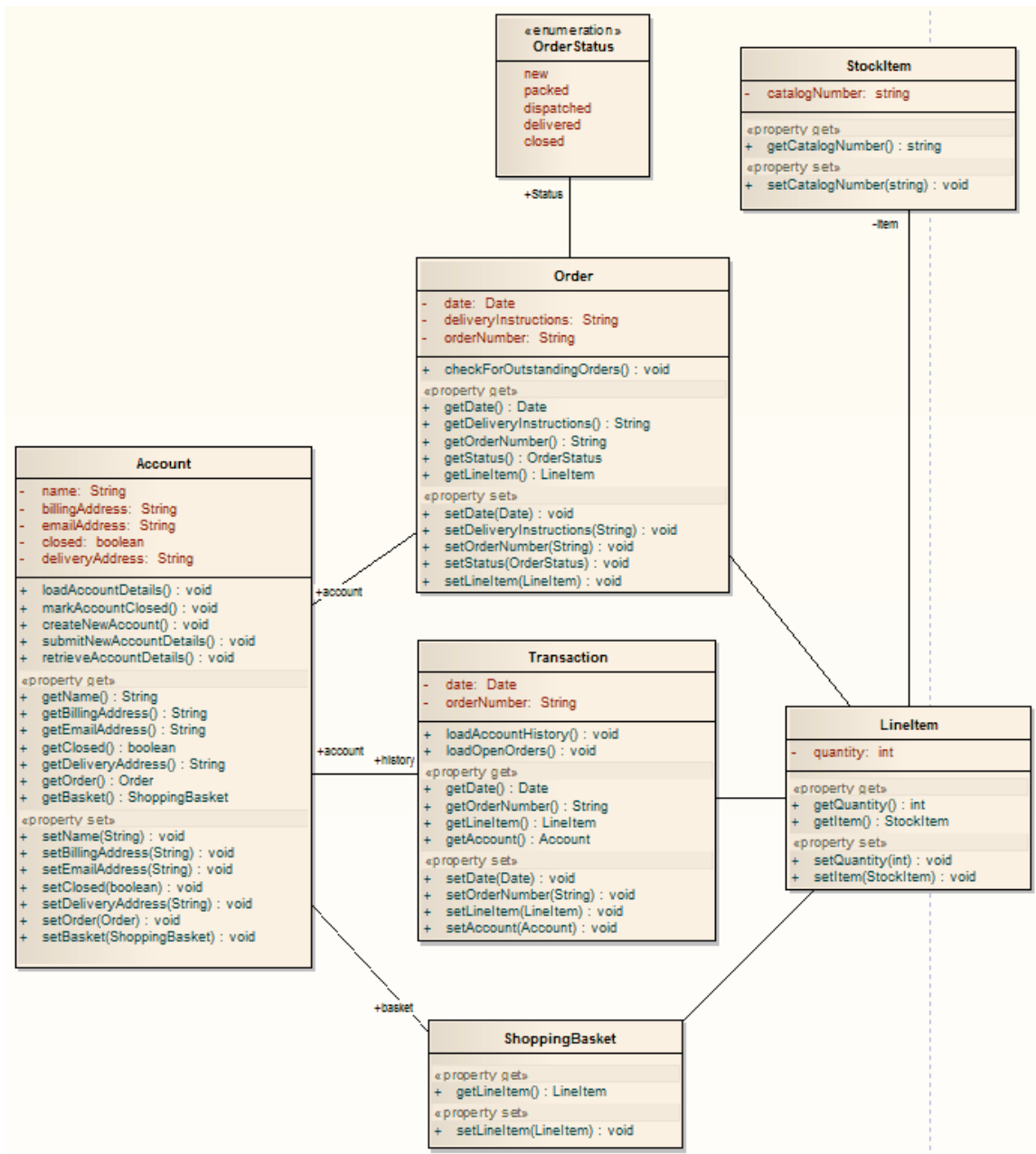
# JUnit Transformation

The JUnit transformation converts an existing Java Class with public methods into a Class with a test method for each public method. The resulting Class can then be generated and the tests filled out and run by JUnit.

## Example

The Java model elements (originally transformed from the PIM)



After transformation, become the PSM elements

**junit.framework.TestCase**
**AccountTest**

+ AccountTest(String)
+ main(String[]) : void
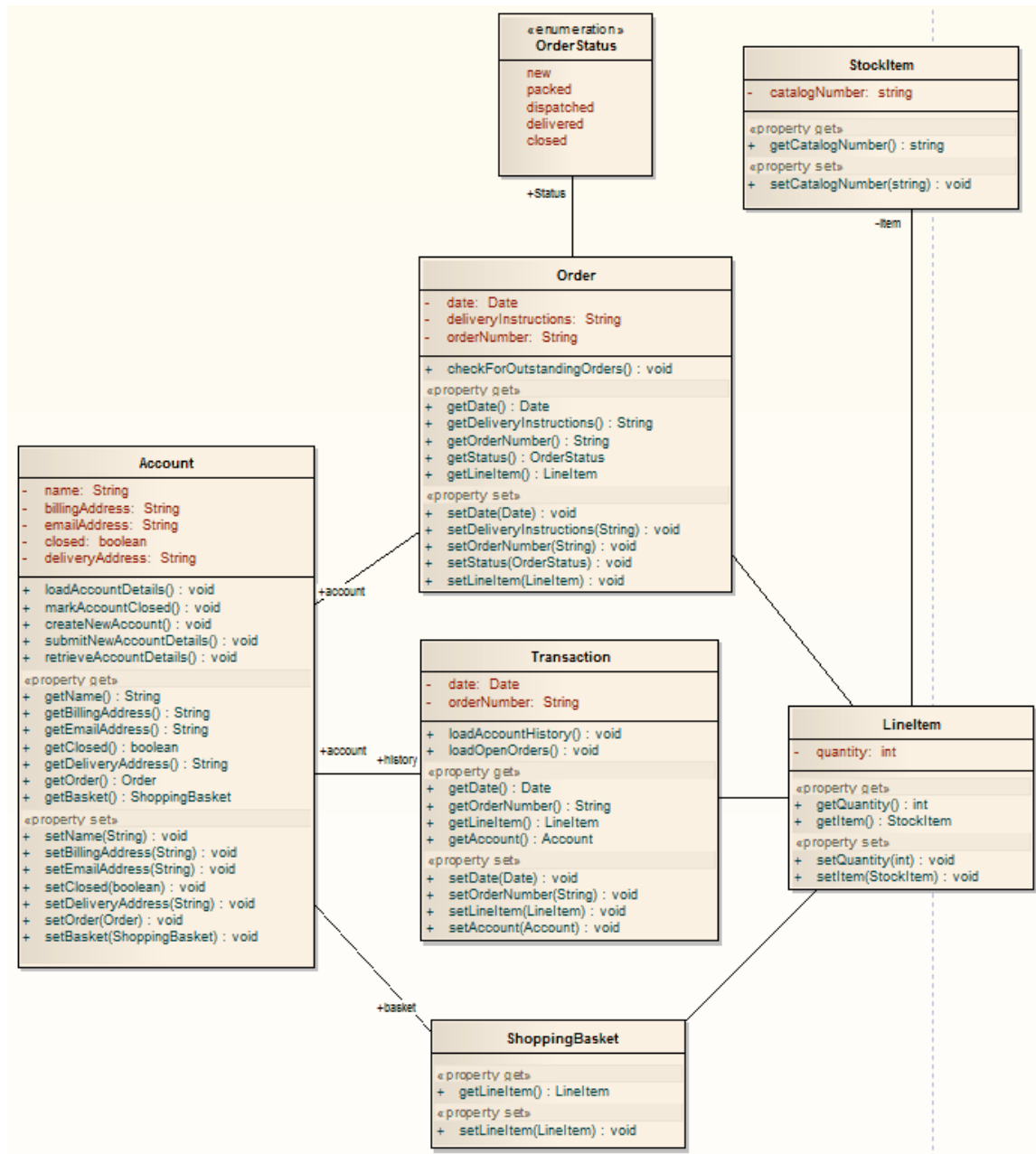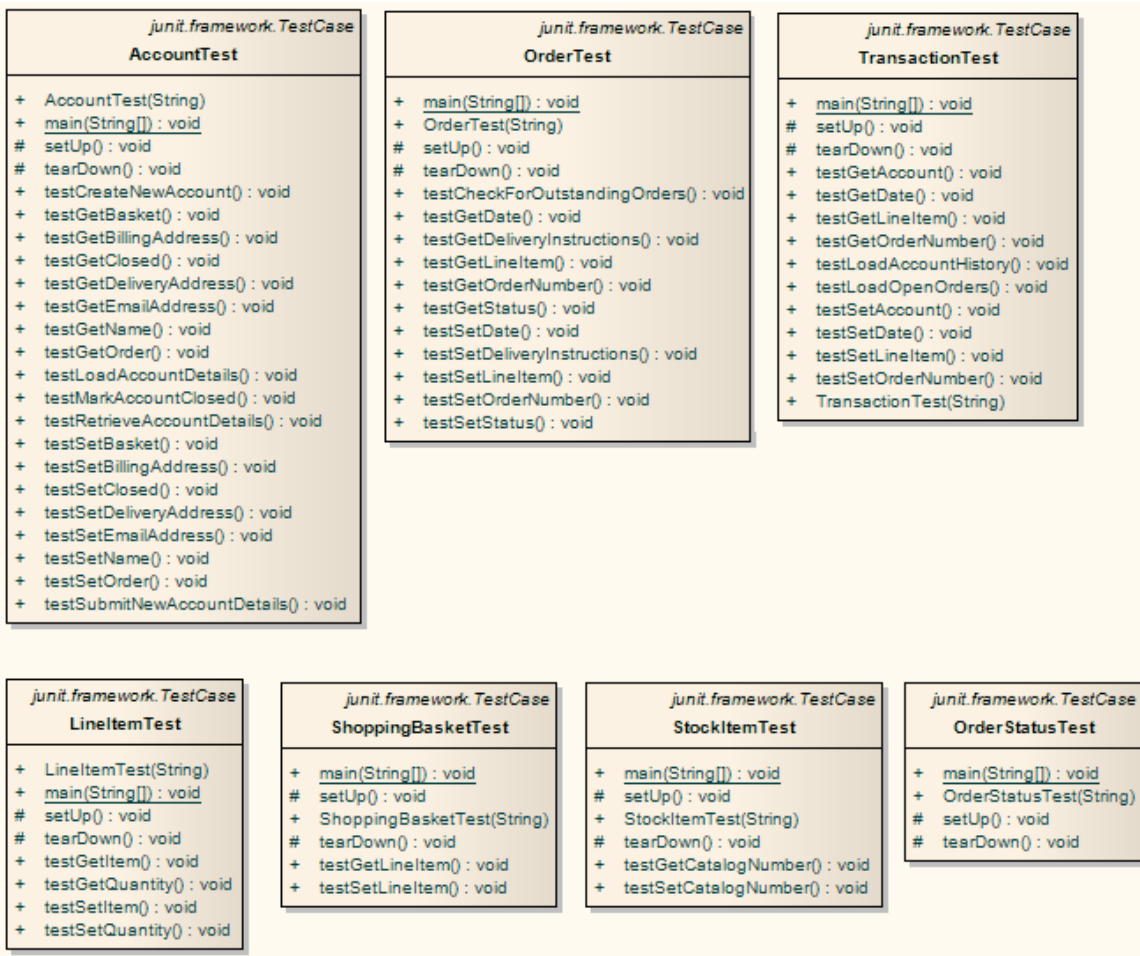# setUp() : void
# tearDown() : void
+ testCreateNewAccount() : void
+ testGetBasket() : void
+ testGetBillingAddress() : void
+ testGetClosed() : void
+ testGetDeliveryAddress() : void
+ testGetEmailAddress() : void
+ testGetName() : void
+ testGetOrder() : void
+ testLoadAccountDetails() : void
+ testMarkAccountClosed() : void
+ testRetrieveAccountDetails() : void
+ testSetBasket() : void
+ testSetBillingAddress() : void
+ testSetClosed() : void
+ testSetDeliveryAddress() : void
+ testSetEmailAddress() : void
+ testSetName() : void
+ testSetOrder() : void
+ testSubmitNewAccountDetails() : void

**junit.framework.TestCase**
**OrderTest**

+ main(String[]) : void
+ OrderTest(String)
# setUp() : void
# tearDown() : void
+ testCheckForOutstandingOrders() : void
+ testGetDate() : void
+ testGetDeliveryInstructions() : void
+ testGetLineItem() : void
+ testGetOrderNumber() : void
+ testGetStatus() : void
+ testSetDate() : void
+ testSetDeliveryInstructions() : void
+ testSetLineItem() : void
+ testSetOrderNumber() : void
+ testSetStatus() : void

**junit.framework.TestCase**
**TransactionTest**

+ main(String[]) : void
# setUp() : void
# tearDown() : void
+ testGetAccount() : void
+ testGetDate() : void
+ testGetLineItem() : void
+ testGetOrderNumber() : void
+ testLoadAccountHistory() : void
+ testLoadOpenOrders() : void
+ testSetAccount() : void
+ testSetDate() : void
+ testSetLineItem() : void
+ testSetOrderNumber() : void
+ TransactionTest(String)

**junit.framework.TestCase**
**LineItemTest**

+ LineItemTest(String)
+ main(String[]) : void
# setUp() : void
# tearDown() : void
+ testGetItem() : void
+ testGetQuantity() : void
+ testSetItem() : void
+ testSetQuantity() : void

**junit.framework.TestCase**
**ShoppingBasketTest**

+ main(String[]) : void
# setUp() : void
+ ShoppingBasketTest(String)
# tearDown() : void
+ testGetLineItem() : void
+ testSetLineItem() : void

**junit.framework.TestCase**
**StockItemTest**

+ main(String[]) : void
# setUp() : void
+ StockItemTest(String)
# tearDown() : void
+ testGetCatalogNumber() : void
+ testSetCatalogNumber() : void

**junit.framework.TestCase**
**OrderStatusTest**

+ main(String[]) : void
+ OrderStatusTest(String)
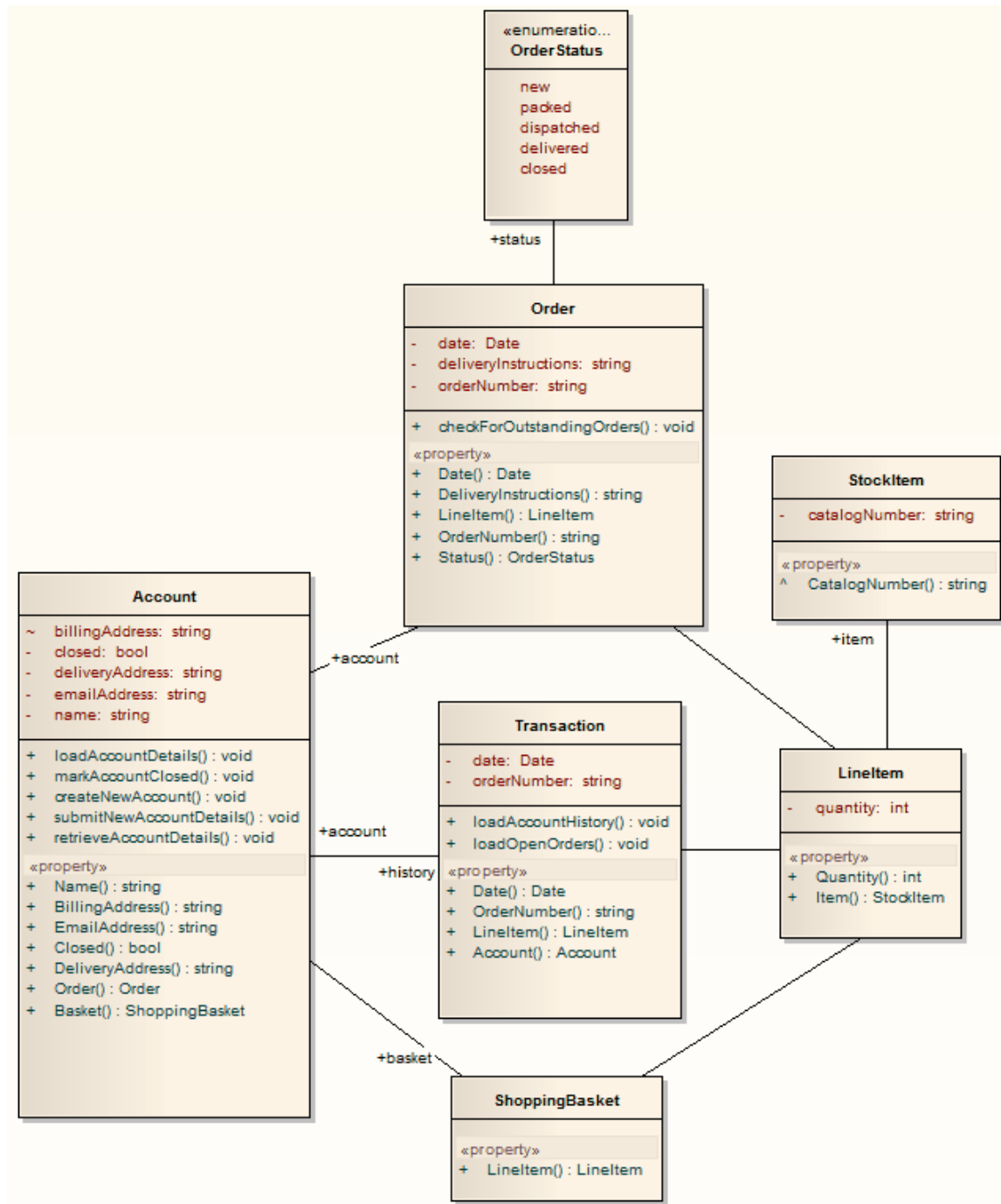# setUp() : void
# tearDown() : void

## Notes

- For each Class in the Java model, a corresponding Test Class has been created containing a test method for every public method in the source Class, plus the methods required to appropriately set up the tests; you fill in the details of each test
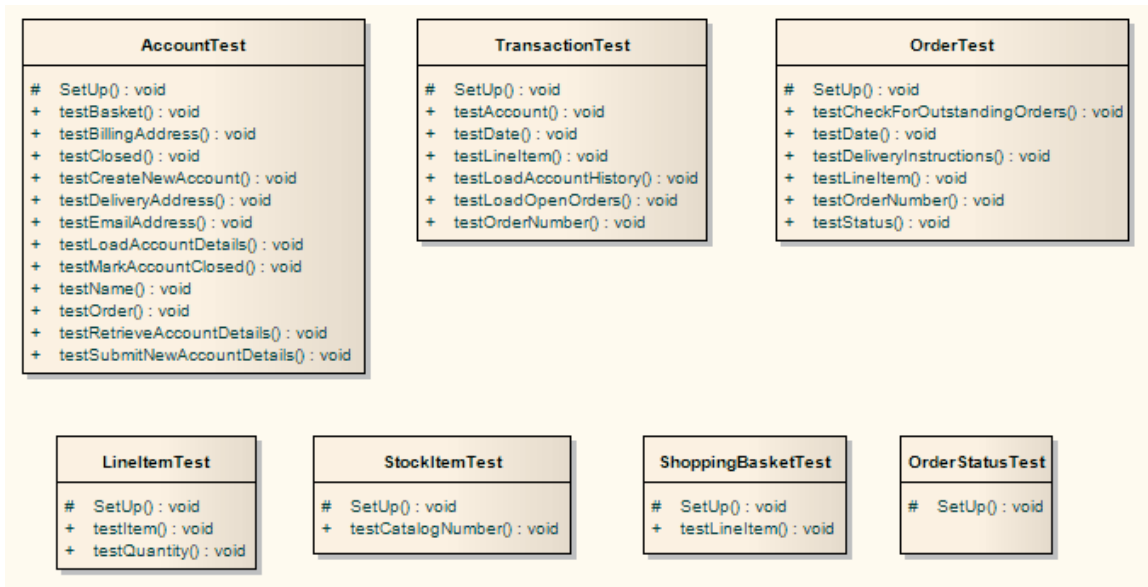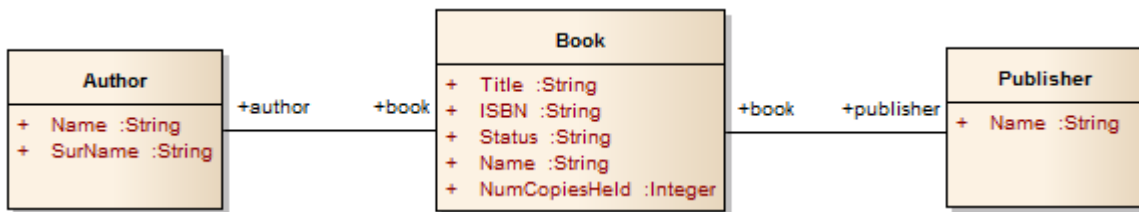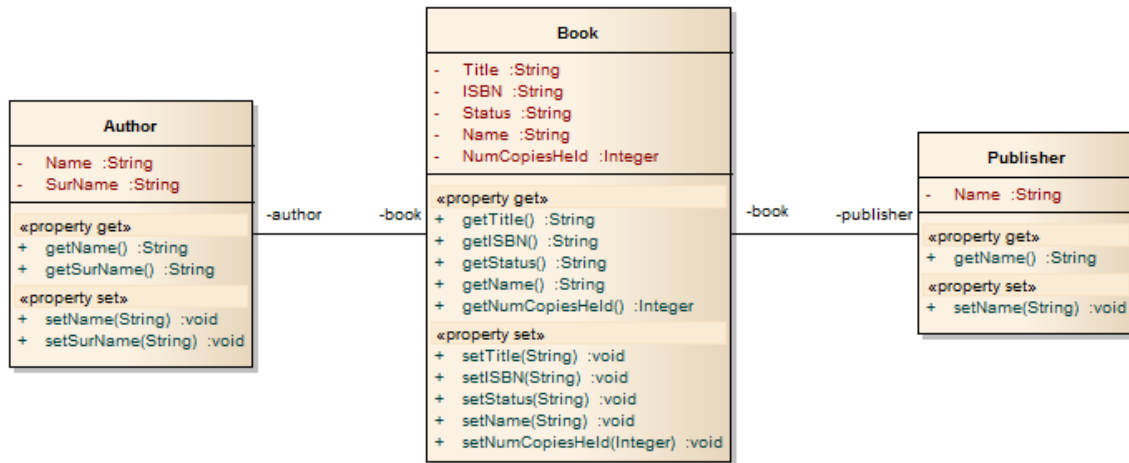
# NUnit Transformation

The NUnit transformation converts an existing .NET compatible Class with public methods into a Class with a test method for each public method. The resulting Class can then be generated and the tests filled out and run by NUnit.

## Example

The C# elements (originally transformed from the PIM)



After transformation, become the PSM elements

**AccountTest**

| # | SetUp() : void |
| + | testBasket() : void |
| + | testBillingAddress() : void |
| + | testClosed() : void |
| + | testCreateNewAccount() : void |
| + | testDeliveryAddress() : void |
| + | testEmailAddress() : void |
| + | testLoadAccountDetails() : void |
| + | testMarkAccountClosed() : void |
| + | testName() : void |
| + | testOrder() : void |
| + | testRetrieveAccountDetails() : void |
| + | testSubmitNewAccountDetails() : void |

**TransactionTest**

| # | SetUp() : void |
| + | testAccount() : void |
| + | testDate() : void |
| + | testLineItem() : void |
| + | testLoadAccountHistory() : void |
| + | testLoadOpenOrders() : void |
| + | testOrderNumber() : void |

**OrderTest**

| # | SetUp() : void |
| + | testCheckForOutstandingOrders() : void |
| + | testDate() : void |
| + | testDeliveryInstructions() : void |
| + | testLineItem() : void |
| + | testOrderNumber() : void |
| + | testStatus() : void |

**LineItemTest**

| # | SetUp() : void |
| + | testItem() : void |
| + | testQuantity() : void |

**StockItemTest**

| # | SetUp() : void |
| + | testCatalogNumber() : void |

**ShoppingBasketTest**

| # | SetUp() : void |
| + | testLineItem() : void |

**OrderStatusTest**

| # | SetUp() : void |

## Notes

- For each Class in the C# model, a corresponding Test Class has been created containing a test method for every public method in the source Class, plus the methods required to appropriately set up the tests; you fill in the details of each test

# PHP Transformation

The PHP transformation converts Platform-Independent Model (PIM) element types to language-specific PHP Class element types and creates encapsulation (producing the getters and setters) according to the options you have set for creating properties from PHP attributes (on the 'PHP Specifications' page of the 'Preferences' dialog). Note that the public attributes in the PIM are converted to private attributes in the PSM.

## Example

The PIM elements



After transformation, become the PSM elements

# Sequence/Communication Diagram Transformations

It is possible to transform a Sequence diagram into a Communication diagram, and to transform a Communication diagram into a Sequence diagram. In each case, every element or message in the source diagram type is mapped 1:1 to a matching element or message in the target diagram.

## Access

| Ribbon | Design > Tools > Transform > Transform Selection |
|---|---|
| Keyboard Shortcuts | Ctrl+Shift+H    (transform current Package)<br>Ctrl+H        (transform selected elements) |

## Perform a Transformation

| Field/Button | Action |
|---|---|
| Elements | Lists and highlights all the elements from the diagram, which will be included in the transformation. |
| Transformations | Select:<br>• The 'Communication' checkbox, if transforming a Sequence diagram into a Communication diagram, or<br>• The 'Sequence' checkbox, if transforming a Communication diagram into a Sequence diagram<br>In either case, the 'Browse Project' dialog displays. Browse for and select the target Package into which the target diagram will be created, then click on the OK button. |
| Do Transform | Click on this button to execute the transformation.<br>The target diagram is created and listed in the Browser window under the target Package with the name (depending on which transformation you have executed):<br><source diagram name> Communication or<br><source diagram name> Sequence |

## Notes

• The diagram being transformed must be open in the main diagram view for the 'Communication' or 'Sequence' options to appear in the Model Transformation dialog.

• Once you have selected the 'Communication' or 'Sequence' checkbox, these transforms ignore any other field setting in the dialog except for 'Target Package', and will perform a direct transformation of every element in the source diagram
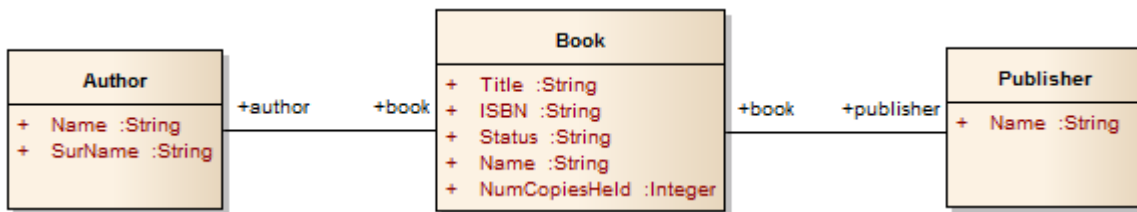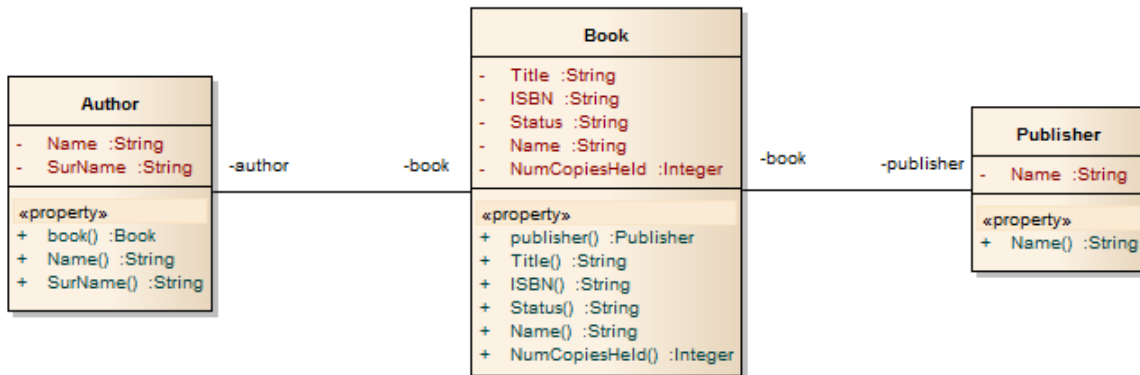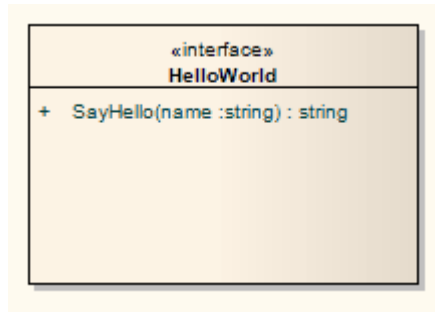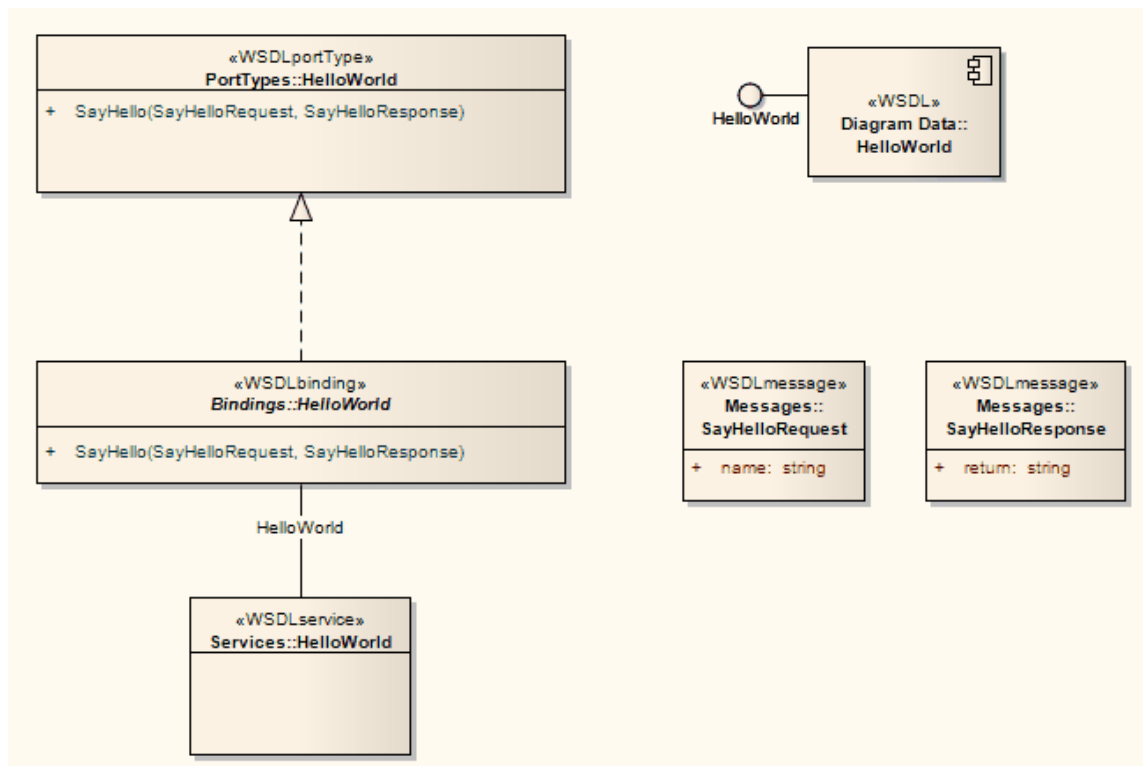
# VB.Net Transformation

The VB.Net transformation converts Platform-Independent Model (PIM) element types to language-specific VB.Net Class element types, and creates encapsulation according to the options you have set for creating properties from VB.Net attributes (on the 'VB.Net Specifications' page of the 'Preferences' dialog). Note that the public attributes in the PIM are converted to private attributes in the PSM.

## Example

The PIM elements



After transformation, become the PSM elements

# WSDL Transformation

The WSDL transformation converts a simple model into an expanded model of a WSDL interface that is suitable for generation. For example:



Transformation of this generates the corresponding WSDL Component, Service, Port Type, Binding and Messages:

- Classes are handled in the same way as in the XSD Transformation

- All 'in' parameters are transformed into WSDL Message Parts in the Request message

- The return value and all 'out' and 'return' parameters are transformed into WSDL Message Parts in the Response message

- All methods where a value is returned are transformed into Request-Response operations, and all methods not returning a value are transformed into OneWay operations

- The transformation does not handle the generation of Solicit-Response and Notification methods or faults
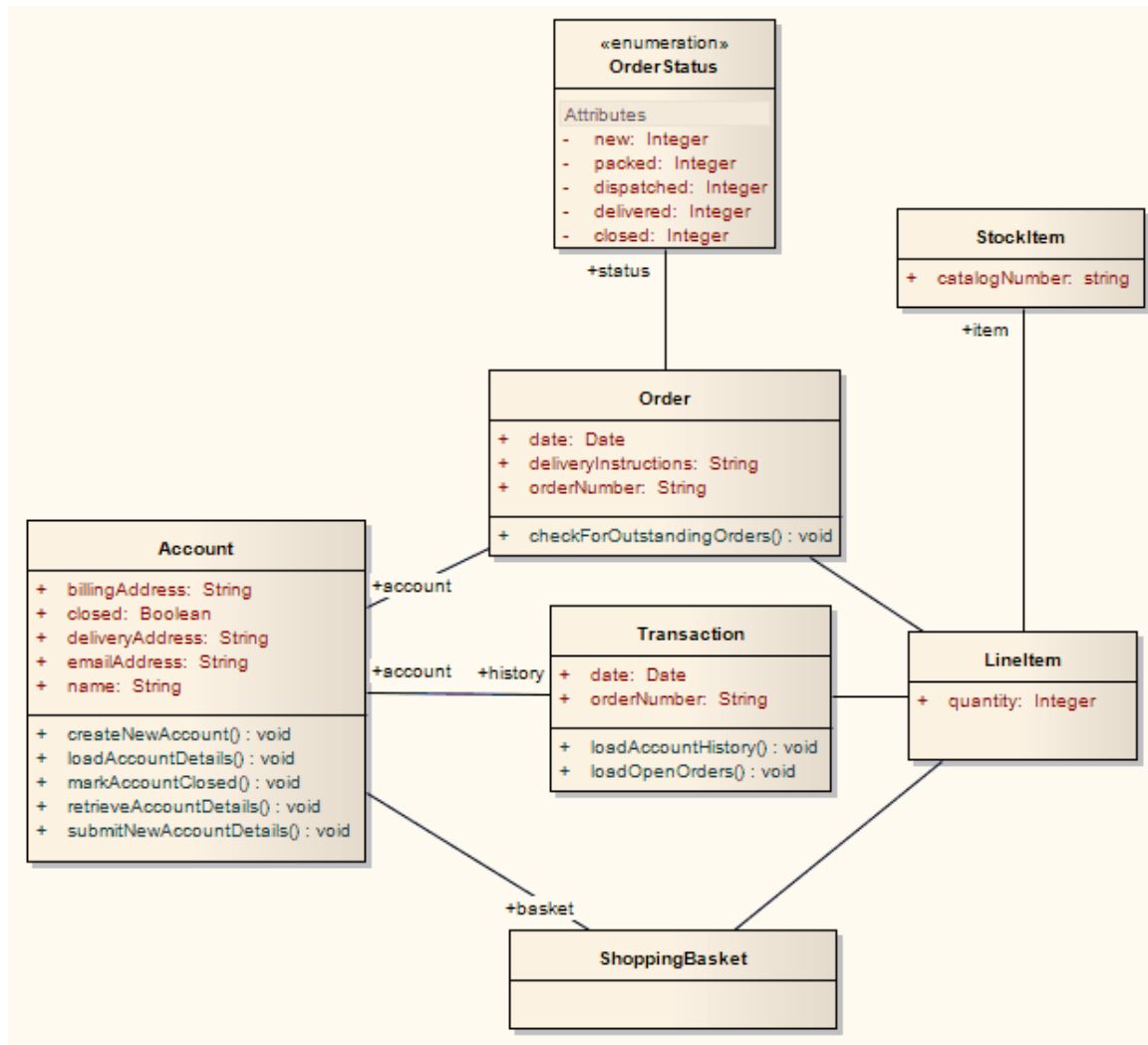


In the resulting Package you can then fill out the specifics using the WSDL editing capabilities of Enterprise Architect, and finally generate the Package using the WSDL generation tools.
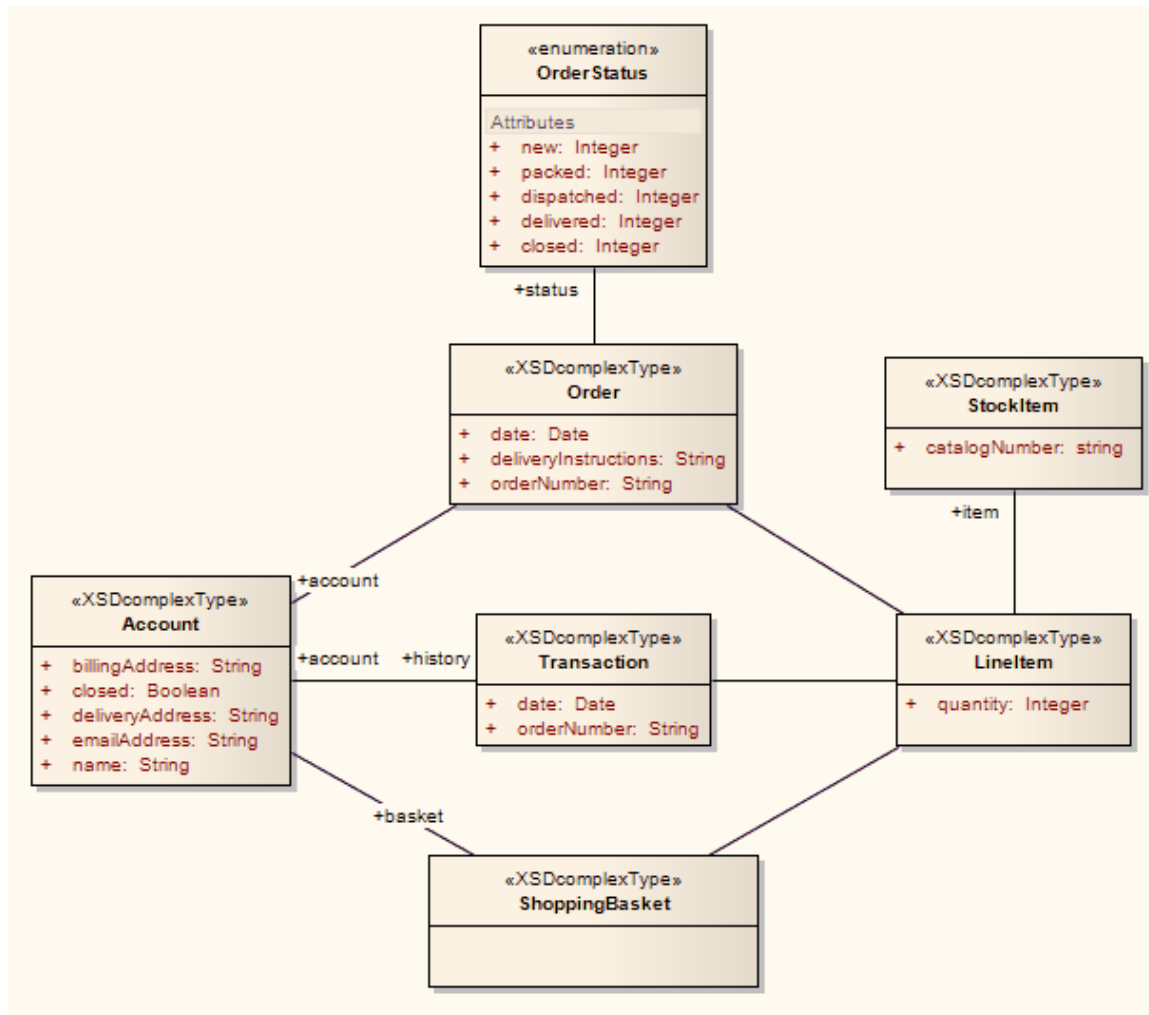
# XSD Transformation

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

## Example

The PIM elements



After transformation become the PSM elements

These in turn generate this 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>
```

# Edit Transformation Templates

A single transformation applies a number of transformation templates, each of which defines a type of object that is acted on within the transformation, and the actions that are performed on objects of that type. The system provides a range of built-in default templates, and each type of transformation uses a specific subset of these templates. Typically, the transformation type and the subset of templates are tailored to the target language. Some default templates within a set have no content; these are 'latent', and represent the potential for acting on an object that is not generally included in the transformation but is perfectly valid if you wanted to include it. An example of a latent template is the Linked Class Base template in the **C#** transformation.

You can tailor the transformation templates in various ways, including:

- Adjust the code in one or more of the templates in a default set

- Add code to a 'latent' default template

- Add a new custom template, based on one of the defaults but serving a different purpose that you define

- Add a new transformation type containing - initially - a basic set of default templates

- Add (or remove) a stereotyped override for a template

A stereotyped override directs the transformation to use the modified template only if the element and/or feature are of the specified stereotyped types. If the object or feature are not of that type, the transformation applies the base template.

## Access

| Ribbon | Design > Tools > Transform > Transform Templates |
|---|---|
| Keyboard Shortcuts | Ctrl+Alt+H |

## Edit Transformation Templates

| Option | Action |
|---|---|
| Language | Click on the drop-down arrow and select the name of the transformation. |
| New Transformation Type | Click on this button if you want to create a new transformation. |
| | A prompt displays for the name of the transformation; type in the name and click on the OK button. |
| | The 'Templates' list shows the default set of built-in templates, from which you can develop your transformation. Your custom transformation is not saved or available for use unless you add and/or edit one or more templates in the transformation. |
| Templates | Lists the transformation templates for the current transformation. |
| | Click on a template name to highlight it and display its content in the Template panel. The 'Modified' column indicates whether you have edited the template for this transformation. |
| Template | Displays the contents of the currently-selected template, and provides the editor facilities for modifying the template (right-click on the code text). |
| | |

| | |
|---|---|
| Stereotype Overrides | Lists the stereotyped templates, for the active base template. The 'Modified' column indicates whether you have modified a stereotyped template. |
| Add New Custom Template | Click on this button to create a custom template to add to the current transformation. A dialog displays, prompting you to specify: <br> • The object type (base template type) that this new template will respond to - click on the drop-down arrow and select the name (custom template types are not included in this list) <br> • The name of the new template - type in the appropriate text <br> Click on the OK button. The new template name is added to the list of templates, and it is opened in the template editor ready for you to add its code. |
| Add New Stereotyped Override | Click on this button to add a new stereotype override for the currently selected template. A dialog displays, prompting you to specify the: <br> • Base Class (stereotyped Class type - click on the drop-down arrow and click on the type in the list) and/or <br> • Feature (click on the drop-down arrow and click on the stereotyped feature in the list) <br> Click on the OK button. The override is added to the 'Stereotype Overrides' list. |
| Get Default Template | Click on this button to update the editor display with the default version of the current built-in template or to clear the content of the current custom template. If you have saved the changed template, re-instating the default version is a change, so the 'Modified' field still displays the word 'Yes'. |
| Save | Click on this button to save the new or edited current template. You cannot switch to another template without saving the current template, so this effectively saves the transition as well. |
| Delete | Click on this button to delete the current custom template or stereotype override, or the most recent changes to a built-in template (effectively returning it to the default, base content). You cannot delete a built-in template. You are prompted to confirm the deletion. |
| Help | Click on this button to display this Help topic. |

## Notes

• Transformation template editing is based very strongly on code generation template editing; for additional information on editing transformation templates see the *Code Template Editor* section and the *Editing Source Code* topic

# Write Transformations

Enterprise Architect provides a facility to create your own transformations; this can be useful to automate the process of generating more specific models from more general ones, reusing the transformation and preventing errors from being introduced as they might if the models were created by hand. The existing templates will provide a useful guide and reference to assist you when creating new templates.

Transformation templates are based on the Code Generation Template Framework, and an understanding of the way these templates work is critical to be able to adjust existing transformation templates or to create new ones. Therefore it is suggested that you read and understand the topics discussing Code Generation Templates prior to using the Transformation Template language.

## Access

| Ribbon | Design > Tools > Transform > Transform Templates |
|---|---|
| Keyboard Shortcuts | Ctrl+Alt+H |

## Factors concerning Transformation Templates

| Factor | Detail |
|---|---|
| Default Transformation Templates | Enterprise Architect provides a set of default transformation templates that you can use 'as is' or customize to your requirements. |
| General Syntax for the Intermediary Language | Transformations in Enterprise Architect generate an intermediary code form of the model being created in the transformation. You can review and edit this code. |
| Intermediary Language Debugging | You can also debug transformation scripts by checking the intermediary code generated from the Transform script. |
| Editing transformation templates and code | When writing transformations, you use the facilities of the common Code Editor. |
| Code Template Framework | You use the Code Template Framework to perform forward engineering of UML models. The Transformation Template Framework is derived from this. |
| Syntax for Creating Objects | To generate objects or elements in a transformation, you apply a specific syntax in the template script. |
| Syntax for Creating Connectors | To generate connectors (relationships) in a transformation, you also apply a specific syntax in the template script. |
| Transforming Duplicate Information | In many transformations there is a substantial amount of information to be copied. Rather than place this information in the template, you can use macros to read it from its source. |
| Transforming Template | In a transformation template, if you are transforming Template Binding connector |

| Parameter Substitutions | binding parameter substitutions, you can use the Template Parameter substitution macros. |
| --- | --- |
| Converting Types | You can apply various methods for converting data types to different target platform types. |
| Converting Names | You can apply various methods for converting names of elements to different target platform naming conventions. |
| Cross References | During a transformation, you can perform cross verification of transformed elements. |

## Notes

- Further hints and tips can be gleaned from a close study of the Transformation Templates provided with Enterprise Architect
- The Transformation Template editor applies the facilities of the common Code Editor

# Default Transformation Templates

Transformation templates provide the ability to represent the existing information in a model in a modified way. When creating a new transformation Enterprise Architect provides a default set of transformation templates that perform a direct copy of the source model to the target model. This allows you to think in terms of how the source model and target model are different. For each template you are able to prevent properties from being copied and add additional information until the appropriate target model is created.

You can list and examine the default templates in the Transformation Editor. The combination of default templates varies according to the language you are transforming.

## Access

| Ribbon | Design > Tools > Transform > Transform Templates |
|---|---|
| Keyboard Shortcuts | Ctrl+Alt+H |

## Notes

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

# Intermediary Language

All transformations in Enterprise Architect create an intermediary language form of the model to generate. You can access and edit the file containing this intermediary language code using an external editor. Each object is represented in this language by the object type (for example, Class, Action, Method, Generalization or Tag) followed by the object properties and the features that it is made from; the grammar of the object description resembles this:

    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
- propertyName is any one of the set of properties

Literal strings can be included in property values by 'escaping' a quote character:

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

# Intermediary Language Debugging

The script from an MDA template produces intermediate language text. However, on generating the model this script could return errors. When an error occurs, you can view and debug the generated text externally, preferably in an editor that prompts on updates to the file alterations.

## Access

| Ribbon | Design > Tools > Transform > Transform Selection |
|---|---|
| Keyboard Shortcuts | Ctrl+H          (transform selected elements)<br>Ctrl+Shift+H       (transform current Package) |

## Debug when errors are returned on generating altered code

| Step | Description |
|---|---|
| 1 | Select the Package to be transformed, and the 'Transform Package' option.<br>The 'Model Transformations' dialog displays. |
| 2 | In the 'Name' column, select the checkbox against the type of transformation being altered. |
| 3 | In the 'Intermediary File' field, click on the [...] button and set the file location into which to generate the code. |
| 4 | Select the 'Write Always' checkbox, and click on the Write Now button to generate the script.<br>This only generates the script, not the model. |
| 5 | If an error is returned specifying the line number of the problem, open the file in an external Code Editor (with Line Numbering) and locate the line number of problem. |
| 6 | Alter the template code to correct the error. |
| 7 | Click on the Do Transform button to check that the alteration has corrected the problem. |

## Example

For a MySQL database, the template code might resemble this:

$enumFieldName = "test"

Column

{

name= %qt%% CONVERT_NAME ($enumFieldName, "Pascal Case", "Camel Case")%%qt%

type= %qt%% CONVERT_TYPE (genOptDefaultDatabase, "Enum")%%qt%

}

This returns the output in the generated text file as:

Column

{

name = "test"

type = "ENUM"

}

If there is an error in the original transform, such as a spelling error - 'Colum' - clicking the Do Transform button returns an error message referring to the first line of intermediate code that includes the error 'Colum'.

# Objects

Objects are generated in a transformation as text in this form:

    objectType
    {
        objectProperties*
        XRef{xref}*
        Tag{tag}*
        Attribute{attributes}*
        Operation{operations}*
        Classifier{classifiers}*
        Parameter{parameters}*
    }

For example:

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

Every object created in a transformation should include an XRef syntax element (see the end of this topic), as it helps the system to synchronize with the object and makes it possible to create a connector to that Class in the transformation.

## Syntax elements in the code

| Element | Detail |
|---|---|
| objectType | objectType is one of these: <br> • Action <br> • ActionPin <br> • Activity |

|  | • ActivityParameter |
|  | • ActivityPartition |
|  | • ActivityRegion |
|  | • Actor |
|  | • Association |
|  | • Change |
|  | • Class |
|  | • Collaboration |
|  | • CollaborationUse |
|  | • 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<br>• Table<br>• TimeLine<br>• Trigger<br>• UMLDiagram<br>• UseCase |
| objectProperties | objectProperties is zero, or one instance of one or more of these:<br>• Abstract<br>• Alias<br>• Arguments<br>• Author<br>• Cardinality<br>• Classifier<br>• Complexity<br>• Concurrency<br>• Filename<br>• Header<br>• Import<br>• IsActive<br>• IsLeaf<br>• IsRoot<br>• IsSpecification<br>• Keyword<br>• Language<br>• Multiplicity<br>• Name<br>• Notes<br>• Persistence<br>• Phase<br>• Scope<br>• Status<br>• Stereotype<br>• Version<br>• Visibility |
| Attribute | Attribute has the same structure as objectType, and includes these properties:<br>• Alias<br>• Classifier<br>• Collection<br>• Container<br>• Containment<br>• Constant<br>• Default<br>• Derived |

| | |
|---|---|
| | • LowerBound<br>• Name<br>• Notes<br>• Ordered<br>• Scope<br>• Static<br>• Stereotype<br>• Type<br>• UpperBound<br>• Volatile<br><br>Attribute also includes these elements:<br>• Classifier<br>• Tag<br>• XRef |
| Operation | Operation has the same structure as objectType, and includes these properties:<br>• Abstract<br>• Alias<br>• Behavior<br>• Classifier<br>• Code<br>• Constant<br>• IsQuery<br>• Name<br>• Notes<br>• Pure<br>• ReturnArray<br>• Scope<br>• Static<br>• Stereotype<br>• Type<br><br>Operation also includes these elements:<br>• Classifier<br>• Parameter<br>• Tag<br>• XRef |
| Parameter | Parameter has the same structure as objectType, and includes the Tag element and these properties:<br>• Classifier<br>• Default<br>• Fixed<br>• Name |

|  |  |
| --- | --- |
|  | • Notes<br>• Kind<br>• Stereotype |
| Tag | Tag has these properties:<br>• Name<br>• Value |

## Special Cases

Certain types of object have variations of the object definition syntax.

| Object | Detail |
| --- | --- |
| Packages | Packages differ from other objects in these ways:<br>• They have a reduced set of properties: alias, author, name, namespaceRoot, notes, scope, stereotype and version<br>• The property namespaceRoot is only given to Packages<br>• A name must be specified for each Package<br>• The name property can be a qualified name; when a qualified name is specified, the properties given are applied only to the final Package<br>• Only Packages can contain other Packages<br>• Packages cannot contain attributes and operations |
| XRef | Cross references are defined using the transform statements. The properties include:<br>• Namespace<br>• Name<br>• Source<br>• Notes |
| Tables | Tables are a special type of object, with these differences from other object types:<br>• They can include columns and primary keys<br>• They cannot include attributes |
| Columns | Columns are similar to attributes, but have an autonumber element containing Startnum and its increment, and these added properties:<br>• Length<br>• NotNull<br>• Precision<br>• PrimaryKey<br>• Scale<br>• Unique<br>In the column definition, you cannot assign a value to the NotNull, PrimaryKey or Unique properties. |

# Connectors

The process of creating connectors in a transformation has the same form as for creating elements (objects). It is a little more complex, because you also define each end of the connector - the source and target.

Connectors are represented in the Intermediary language as:

```
ConnectorType
{
    connectorProperties*
    AssociationClass {associationClassProperties*}
    Source {sourceProperties*}
    Target {targetProperties*}
}
```

For example:

```
Association
{
    name="anAssociation"
    stereotype=""
    direction="Unspecified"
    Source
    {
        access="Private"
        navigability="Unspecified"
    }
    Target
    {
        access="Private"
        multiplicity="1..*"
    }
}
```

## Syntax elements in the code

| Element | Detail |
|---|---|
| ConnectorType | ConnectorType is one of these:<br>• Abstraction<br>• Aggregation<br>• Assembly<br>• Association<br>• Collaboration<br>• ControlFlow |

|  |  |
|---|---|
|  | • Connector<br>• Delegate<br>• Dependency<br>• Deployment<br>• ForeignKey<br>• Generalization<br>• InformationFlow<br>• Instantiation<br>• Interface<br>• InterruptFlow<br>• Manifest<br>• Nesting<br>• NoteLink<br>• ObjectFlow<br>• Package<br>• Realization<br>• Sequence<br>• Substitution<br>• TemplateBinding<br>• Transition<br>• Usage<br>• UseCase<br>• Uses |
| connectorProperties | connectorProperties is zero, or one instance of one or more of these:<br>• alias<br>• direction<br>• notes<br>• name<br>• stereotype<br>• tag<br>• XRef |
| associationClassProperties | associationClassProperties are one instance of these:<br>• Classifier<br>• XRef |
| sourceProperties<br>targetProperties | sourceProperties and targetProperties are each a reference to an element and zero, or one instance of one or more of these:<br>• aggregation<br>• alias<br>• allowduplicates<br>• changeable<br>• constraint<br>• containment<br>• navigability |

| | |
|---|---|
| | <ul><li>membertype</li><li>multiplicity</li><li>Notes</li><li>ordered</li><li>qualifier</li><li>role</li><li>scope</li><li>stereotype</li><li>tag</li></ul> |
| Element Reference | An element reference is either a guid that references an element that already exists before the transformation, or an XRef to reference an element that is created by a transformation.<ul><li>guid</li><li>XRef</li></ul> |

## Notes

- Each connector is transformed at both end objects, therefore the connector might appear twice in the transformation; this is not a problem, although you should check carefully that the connector is generated exactly the same way, regardless of which end is on the current Class

# Transform Connectors

When you transform a connector, you can use two different types of Class as the connector ends: either a Class created by a transformation, or an existing Class for which you already know the GUID.

## Connect to a Class Created by a Transformation

The most common connection is to a Class created by a transformation; to create this connection you use 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 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 you want a connector between them; consider this script from the EJB Entity transformation:

Dependency

{

%TRANSFORM_REFERENCE("EJBRealizeHome",classGUID)% stereotype="EJBRealizeHome"

Source

{

%TRANSFORM_REFERENCE("EJBEntityBean",classGUID)%

}

Target

{

%TRANSFORM_REFERENCE("EJBHomeInterface",classGUID)%

}

}

In this script there are three uses of the TRANSFORM_REFERENCE macro: one to identify the 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 to something within the current transformation - each of them then only has to identify the transform name.

It is also possible to create a connector from another connector. You can create a connector template and list all connectors connected to a Class from the Class level templates; you don't have to worry about only generating the connector once, because if you have created a TRANSFORM_REFERENCE for the connector then the system automatically synchronizes them.

This script 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 an existing element for which you know the current GUID. To create this connection, specify the GUID of the target Class in either the source or target end; this script 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%
}
}
```

## Notes

- Each connector is transformed at both end objects, therefore the connector might appear twice in the transformation; this is not a problem, although you should check carefully that the connector is generated exactly the same way, regardless of which end is on the current Class

# Transform Foreign Keys

Enterprise Architect supports the transformation into Foreign Keys of many different types of relationship defined between entities in a logical model.

Each Foreign Key in a Physical model is represented by the combination of a stereotyped connector and an operation in each of the involved Tables. Foreign Key transformations are achieved with the 'Connector' template in the DDL language. This template generates an intermediate dataset that is then interpreted by Enterprise Architect's transformation engine to create all the required physical entities and connectors.

By default, Enterprise Architect supports transformations of these connector types:

- Generalization - this kind of connector will create a Foreign Key with a multiplicity of 0..1 in the source and 1 in the destination

- Association Class - this kind of connector will create a 'join' table linking both the source and destination Tables

- Association/Aggregation - these kinds of connector use the multiplicity defined in the Logical model's relationship to join the source and destination Tables

All Foreign Key definitions will cause the addition of a new integer (or equivalent) column in both source and destination Tables, which will act as the Primary Key in the source Table and the Foreign Key column in the destination Table. The default names for the new columns will be the Table name with the suffix of 'ID' added, whilst the names of the Foreign Keys will be automatically generated using the FK DDL template.

# 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 and TRANSFORM_TAGS function macros.

## Use of Macros

| Objective | Detail |
|---|---|
| Copy Object | TRANSFORM_CURRENT (<listOfExcludedItems>)<br><br>The function generates an exact copy of all the properties of the current item, except for the items named in <listOfExcludedItems>. |
| Copy Connector | Another form of the function is available when transforming connectors to copy either end of the connector:<br>TRANSFORM_CURRENT (<connectorEnd>, <listOfExcludedItems>)<br><br>This generates an exact copy of the connector end specified by <connectorEnd> (either Source or Target) except for the items named in <listOfExcludedItems>. |
| Copy Tags | TRANSFORM_TAGS (<listOfExcludedItems>)<br><br>The function generates an exact copy of all the Tagged Values of the current item, except for the items named in <listOfExcludedItems>. |

# Convert Types

Different target platforms almost certainly require different data types, so you usually require a method of converting between types. This is offered by the macro:

CONVERT_TYPE (<destinationLanguage>, <originalType>)

This function 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>)

This function converts <originalType> to the corresponding datatypes in <destinationDatabase>, which is defined in the model; <originalType> refers to a platform independent common datatype.

# Convert Names

Different target platforms use different naming conventions, so you might not want to copy the names of your elements directly into the transformed models. To facilitate this requirement, the transformation templates provide a CONVERT_NAME function macro.

Another way in which you can transform a name is to remove a prefix from the original name, with the REMOVE_PREFIX 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: the first word begins with a lower-case letter but subsequent words start with an upper-case letter; for example, myVariableTable

- Pascal Case: the first letter of each 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

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 this form:

   <firstWord> (<delimiter>) <otherWords>

- <firstWord> controls the case of the first word

- <delimiter> is the string generated between words

- <otherWords> applies to all words after the first word

Both <firstWord> and <otherWords> are 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%

## Notes

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

# Cross References

Cross References are an important part of transformations. You can use them 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 transformation, and
- 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 generate the Cross References using the macro defined for this purpose:

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

The three parameters are optional. The macro generates a reference that resembles this:

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

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

The only time that a Cross Reference 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 transformation 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 macro:

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

This macro 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, TRANSFORM_CLASSIFIER is inappropriate, so instead the GUID can be given directly to a classifier attribute.

If a classifier is specified for any type, it overrides that type.

# Transform Template Parameter Substitution

If you want to provide access in a transformation template to data concerning the transformation of a Template Binding connector's binding parameter substitution in the model, you can use the Template Parameter substitution macros.

## Factors in the Transformation

| Factor | Detail |
|---|---|
| Intermediary Language | Template Parameter Substitutions are represented in the Intermediary language as:<br><br>TemplateParameterSubstitution<br>{<br>    Formal { FormalProperties }<br>    Actual { ActualProperties }<br>}<br><br>For example:<br>TemplateParameterSubstitution<br>{<br>    Formal<br>    {<br>        name=%qt%%parameterSubstitutionFormal%%qt%<br>    }<br>    Actual<br>    {<br>        name=%qt%%parameterSubstitutionActual%%qt%<br>        %TRANSFORM_CLASSIFIER("Class", parameterSubstitutionActualClassifier)%<br>    }<br>} |
| Formal Properties or Actual Properties | FormalProperties and ActualProperties are zero, or one instance of one of these properties:<br>• name<br>• classifier |
| Transform of Parameter Substitution Actual parameter | If the Actual parameter is assigned a String Expression, it will transform as Actual name. You can assign the Actual Classifier if you know the GUID:<br>TemplateParameterSubstitution<br>(<br>Formal<br>{<br>name=%qt%%parameterSubstitutionFormal%%qt%<br>}<br>Actual |

{

name=%qt%%parameterSubstitutionActual%%qt%

classifier=%qt%%parameterSubstitutionActualClassifier%%qt%

}

}

If you want the Actual parameter to be transformed so that its Classifier is assigned with an element that is transformed, then use TRANSFORM_CLASSIFIER or TRANSFORM_REFERENCE, as shown:

TemplateParameterSubstitution

{

Formal

{

name=%qt%%parameterSubstitutionFormal%%qt%

}

Actual

{

name=%qt%%parameterSubstitutionActual%%qt%

%TRANSFORM_CLASSIFIER("Class", parameterSubstitutionActualClassifier)%

}

}

Or

TemplateParameterSubstitution

{

Formal

{

name=%qt%%parameterSubstitutionFormal%%qt%

}

Actual

{

name=%qt%%parameterSubstitutionActual%%qt%

%TRANSFORM_REFERENCE("Class", parameterSubstitutionActualClassifier)%

}

}