



COGILITY
SOFTWARE

Modeling with Cogility Studio

Copyright © 2001-2011 Cogility Software Corporation

All Rights Reserved.

Modeling with Cogility Studio is copyrighted and all rights are reserved. Information in this document is subject to change without notice and does not represent a commitment on the part of Cogility Software Corporation. The document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Cogility Software Corporation.

Document version number 7.0

Cogility is a trademark of Cogility Software Corporation. Other brands and their products are trademarks of their respective holders and should be noted as such.

Cogility Software Corporation
111 N. Market St. #888
San Jose, CA 95113

support@cogility.com

Printed in the United States of America.

The software described in this book is furnished under a license agreement and may be used only in accordance with the terms of the agreement.



Modeling with Cogility Studio

Contents

Preface

Cogility Studio documentation	13
-------------------------------------	----

Chapter 1 Introduction to Cogility Modeler

Configuration management	16
Current CM context	16
Configuration map	16
Version	17
Logging into the current context	17
Exit Cogility Modeler	18
Authoring repository	19
Authoring repository configuration	19
Load Repository utility	20
Loading the repository	21
Model container	21
Creating a model container	22
Packages	22
Creating a package	22
Virtual Packages	23
Model artifacts	24
Deleting model artifacts	25
Configuration Parameters	25
Reloading configuration parameters into Cogility Modeler	25
Deployment	26
Execution	27

Chapter 2 Information model

Master information model (MIM)	29
Distinct information model (DIM)	30
Classes	33
Superclass	33
Abstract class	34
Attributes	35
Sequenceable attributes	36
POID attribute	37
Defining a class	37
Class comments	39
Database index	40
Operations	41
Special Operation to Initialize a Class Instance	44
Custom types	45
Determining where a custom type is used	45
Data object set (DOS)	46
Factories	48
Class diagrams	49
Associations	51
Association settings	53
Association classes	54

Editing an association class	55
Reflexive associations	57
Creating a reflexive association	57
Viewing association chains	57

Chapter 3 Legacy Information Model

Creating a LIM container	61
Discover Tables	62
Using a LIM in passive mode	65
LIMClasses	66
Adding operations to a LIMClass	67
SQL Statements	67
In-line SQL statements	67
Execute statements	68
Statement objects	69
Statement object SQL syntax	69
Select objects	70
Execute objects	71
SQL block objects	73
Using a LIM in active mode	75
Activating a LIM	76
ActivatedTables	76
Activated table actions	77
Creating actions for an activated table	78
Adding operations to an activated table	78
Activation Configuration Parameters	78

Chapter 4 Message modeling

Events	82
Standard event	84
Abstract event	84
Internal event	84
Creating a standard event	84
DOS event	85
Creating a DOS event	86
Time event	86
Time expression	87
Time value configuration parameter	88
Time Expression Override	88
Creating a time event	88
Transient class	89
Start object	90
Designating the start object	90
Conversions	91
Event conversions	91
M2E conversion	91
E2M conversion	93
DOS conversions	94
M2D conversion	94
D2M conversion	96
Conversion actions	98
Pre- and post-conversion actions	98
Default conversion	98
Business object locator	99
DOS populator	99
Applicability test	101

Defining an applicability test	101
Destinations	102
Creating a destination	102
Creating a destination in a conversion	102
Setting a destination in a conversion	102
Defining a destination	102
Messages	103
JMS messages	103
Web service messages	104
Creating a message	106
Creating a message in a conversion	107
Creating a message in a web service	107
Setting a message in a conversion	107
Setting a message in a web service	107
Defining a message	107
Message definitions	108
Exporting message definitions	108

Chapter 5 Behavior model

Transactions	111
State machine	112
State machine editor features	115
Creating a state machine	115
Defining behavior artifacts automatically	115
Defining behavior artifacts manually	116
Defining a state machine	116
States	117
Initial state	118
Simple state	118
Decision state	118
Sequential composite state	118
Creating a sequential composite state	119
Concurrent composite state	120
Creating a concurrent composite state	120
Replicate state	121
Creating a replicate state	121
Final state	122
Placing states in a state machine	122
Defining states	122
Setting states from within transitions	123
Editing states from within transitions	123
State actions	123
Creating state actions	124
Transformation unit	124
Transitions	125
Creating transitions between states	126
Creating a circular transition	126
Defining a transition	126
Event trigger	127
Creating a trigger event in a transition	128
Setting a trigger event in a transition	128
Guard condition	128
Creating a guard condition	129
Effect action	129
Creating an effect action	130
Start event	130
Applicability actions in a start event	130

Creating a start event	131
Adding a start event to a class or factory	131
Batch processing	132
Limitations	132
BusinessObjectsBatch	132
newBatchFor	132
State variable	134
Set the state variable	134
Get the state variable	134
Get the batch	134
Batch processing example	135
Model objects	135
Subscriber class	135
FindSubscriberM2E	136
FindSubscriberEvent	137
FindSubscriber behavior	137
Batch processing actions	137
CreateBatch	138
ProcessBatch	139
DeleteBatch	142
Running the batch processing example	142

Chapter 6 Transformation model

Opaque transformation map	146
Creating an opaque transformation map	148
Creating an opaque transformation map in the Data Transformation Editor	148
Defining an opaque transformation map	148
Transformation map	149
Creating a transformation map	152
Creating a transformation map in the Data Transformation Editor	152
Creating a transformation map in a classifier map	153
Adding a classifier map to a transformation map	153
Defining a transformation map	153
Classifier map	153
Creating a classifier map	154
Adding a class to a classifier map	154
Creating a classifier map in a transformation map	154
Creating a classifier map in the Data Transformation Editor	154
Adding a classifier map to a transformation map	155
Setting a classifier map in the Data Transformation Editor	155
Defining a classifier map	155
Feature map usage	156
Classifier feature map usage	157
Feature map	158
Creating a feature map	159
Creating a feature map in a feature map usage	159
Creating a feature map in the Data Transformation Editor	159
Setting a feature map in a feature map usage	159
Setting a feature map in the Data Transformation Editor	159
Defining a feature map	160
Classifier feature map	160
Creating a classifier feature map	162
Creating a classifier feature map in a classifier feature map usage	162
Creating a classifier feature map in the Data Transformation Editor	162
Setting a classifier feature map in a classifier feature map usage	162
Setting a classifier feature map in the Data Transformation Editor	162
Defining a classifier feature map	163

Transformation Chain	163
Rules for transformation chains	164
Transformation chain M2D	165
Transformation chain D2M	166
Serial transformation chains	168
Creating a serial transformation chain	168
Defining a serial transformation chain	169
Parallel transformation chains	169
Creating a parallel transformation chain	170
Defining a parallel transformation chain	170
Creating a transformation chain in an M2D conversion	170
Creating a transformation chain in a D2M conversion	171
Setting a transformation chain in an M2D conversion	171
Adding a transformation chain to a D2M conversion	171
Creating transformation units in a transformation chain	171
Adding transformation units to a transformation chain	172
Setting the order of transformation units	172
Data Transformation Editor	173
Panel function buttons	175
DIM/MIM Classes panel	175
DOS panel	176
Transforms panel	176
Panel Menus	177
DIM/MIM Classes panel	177
DOS panel	177
Transforms panel	178
Running the Data Transformation Editor	178
Viewing data transformations with the Data Transformation Editor	178
Creating a transformation with the Data Transformation Editor	179

Chapter 7 Custom Queries

Overview	181
Custom query attributes	181
Custom query conditions	182
Custom query arguments	183
Custom query relations	183
Other clauses in custom queries	184
Creating a custom query	185
Verify the generated SQL	186
Custom query example	186
Model objects	187
Running the sample model	187
XSD artifacts	188
Creating an XSD artifact	189
Imported XSD	189
Loading an XSD	190
XSD artifacts in action semantics	190
Read XML	190
Write XML	191
Associations for sub elements	192

Chapter 8 XSD Artifacts

Overview	195
Creating an XSD artifact	196
Importing an XSD	197
Loading an XSD	197

XSD artifacts in action semantics	198
Read XML	198
Write XML	199
Associations for sub elements	199

Chapter 9 Web Services Model

Modeling SOAP Inbound Web Services	202
Creating an inbound web service	202
Web service action	203
IWS parameters	203
Argument decoding and result encoding	203
XSD fragment	204
Result top level element	204
Omit nulls in result	204
Complex type support	205
IWS deployment	205
Creating an IWS deployment	206
Adding an IWS deployment to a handler chain	207
Defining an IWS deployment	207
Adding an inbound web service to a deployment (from the deployment)	207
Adding an inbound web service to a deployment (from the web service)	208
Updating deployment operations	208
Defining the IWS arguments and results messages	208
Inbound web service invocation	209
callIXIWS	210
Endpoint	211
Invocation example	211
Action script	211
Running the example	211
Outbound web services	212
Automatic WSDL import	212
Importing a WSDL from a URL	213
Importing a WSDL from a file	213
Testing an outbound web service	214
Outbound web service artifacts	214
Creating an outbound web service manually	214
Defining an outbound web service	214
OWS deployment	215
Adding an OWS deployment to a handler chain	215
OWS WSDL	215
OWS operation	217
Outbound web service invocation	219
callIOWS	219
DOM Document object	220
XmlLoad()	220
XmlGetString()	220
Outbound web service example	221
Model objects	221
Actions script	221
Example execution	223
Running the example with Cogility Web Service Exerciser	223
Running the example with Cogility Action Pad	224
Web service handlers	224
Creating a handler chain	224
Setting a handler chain	224
Adding a web service to a handler chain	225
Defining a handler chain	225

Handler objects	225
Creating a handler	226
Adding handlers to a chain	226
Defining a handler	226
Setting the order of handlers in handler chains	227
Request and response actions	227
Creating request and response actions	228
SOAP versus HTTP	228
Inbound web services using SOAP over HTTP	228
Inbound web services using HTTP	229
Interchanging HTTP and SOAP Web Services	230
Modeling HTTP operations inbound web services	230
REST versus RPC Architecture	230
"relativeURL" attribute for HTTP Web Services	230
Data Format for HTTP web services	231
Complex Data Payloads	231
OCL Syntax	233
Creating an inbound HTTP web service	233
Defining an HTTP inbound web service	234
HTTP Deployment	234
Creating an HTTP deployment	235

Chapter 10 HTML Views and Charts

HTML Views	237
Defining the HTML	238
Actions	239
var and HTML constructs	239
table construct	240
HTMLTable	240
Sorting table rows and columns	244
Cell Area Definition	245
Creating an HTML view	246
Adding HTML to an HTML View	246
Adding actions to an HTML View	247
Creating an HTML table	247
Creating an HTML table template	248
Creating an HTML table cell template	249
Chart View	250
Creating a chart view	251
Adding Actions to a Chart View	252
Setting chart titles programmatically	252
Adding categories to a chart	253
Drilling down in a chart view	253

Chapter 11 Deployment Model

Default Deployment Model	256
Editing the default deployment model	257
Deployments	257
Creating a deployment model	257
Database artifact	257
Creating the database artifact	258
Setting the database artifact	258
Defining the database artifact	258
Testing the database connection	259
Application server artifact	260
Creating an application server artifact	260

Setting the application server artifact	260
Defining a WebLogic application server artifact	260
Defining an Oracle 10g application server	262
Auxiliary Web Applications	264
Multiple Application Server Deployment	265
Topic to Queue Routing	266
Switching Between Target Deployment Environments	266
Run-time respiratory	267
Dropping the database tables	268
Deployment preparation	269
Inconsistencies	269
Non-additive model	270
Application server	270
Starting the application server	270
Actions after startup	270
Stopping the application server	271
Previewing a push	271
Pushing the model into execution	273
Actions after push	274

Appendix A Cogility Modeler Interface

Views	275
Tree view	276
Editor view	276
Messages view	277
Inconsistencies view	279
Push actions view	280
Search	281
Search For Name In Action Semantics	281
Search dialog	283
History	284
Menus and tools	284
File Menu	285
Selection Menu	285
UI menu	294
Help menu	295
Additional menus	295
Move selection	296
Select Object(s) dialog	297
Common editing functions	298
Tab and indent	300
Model assist	300
Association classes	303
Class attributes	304
Model classes	304
Java classes	305
Transient classes	305
Bound variables	305
Java actions	306
Keywords	306
LIM	306
Methods	307
Operations	307
Parameters	308
Publish	308
Roles	309
Variables	310

Web service	310
XSD element definition	311
XSD type definition	311
Keystrokes and mouse click responses	311
Model assist configuration parameters	312
Window size	312
Custom Java actions	312
Classes	312
Using model assist	313
Syntax assist	313
Using syntax assist	314
Find / Replace	314
Find String	315
Replace With	315
Find Next	315
Find From Start	315
Replace	315
Replace/Find	315
Find/Replace All	315
Find/Replace To End	315
Forward and Backward	315
Ignore Case	316
Process Backslashes As Escapes	316
Escape Characters	316
Escape Sequences for Character and String Literals	316
Using Find/Replace	317
History comments	317
History comment configuration parameters	317
Setting the history comment user name	319
Setting the history comment tracking reference	319
Adding history comments	319
Removing history comments	320
Java package alias	320
Defining a Java package alias	320
System view diagrams	321
Creating a system view diagram	321
Displaying a system view diagram	322
Model inputs and outputs	322
Adding model inputs	322
Adding model outputs	323
System view diagram elements	323
Adding elements to a system view diagram	323
Displaying inputs to an element	324
Displaying outputs from an element	324
Displaying derived outputs from an element	324
Displaying elements associated to an element	324
Selecting elements	324
Editing elements	325
Expanding nodes	325
Collapsing nodes	325
Setting the node image	325
Setting the node color	325
Setting the connection line color	326
Displaying element shadows	326
Reverting to default settings for elements	326
Removing elements from the diagram	326
Removing artifacts from the model	326
Display settings	327

Changing the display size	327
Setting the background color	327
Setting the background image	327
Resetting the background color to the default	327
Displaying the grid	328
Turning on snap to grid	328
Displaying page breaks	328
Print facilities	328
Defining the page setup	328
Defining the print scale	328
Specifying printing options	328
Creating a PDF document of the diagram	329
Class Diagrams	329
Class diagram elements	329
Selecting elements	330
Editing elements	330
Expanding nodes	330
Collapsing nodes	330
Displaying connections	331
Setting the node image	331
Setting the node color	331
Applying the shape style	332
Applying the default shape style	332
Setting the connection line color	332
Displaying element shadows	332
Reverting to default settings for elements	332
Removing elements from the diagram	333
Removing artifacts from the model	333
Display settings	334
Changing the diagram style	335
Changing the display size	335
Setting the background color	335
Setting the background image	336
Resetting the background color to the default	336
Displaying the grid	336
Turning on snap to grid	336
Displaying page breaks	336
Print facilities	336
Defining the page setup	336
Defining the print scale	337
Specifying printing options	337
Creating a PDF document of the diagram	337



Preface

Cogility Studio provides the tools to create a model for an enterprise information system, and deploy it as a J2EE application. The Cogility Studio documentation provides support for this endeavor.

Cogility Studio documentation

Cogility Studio comes with several volumes of documentation to help you.

- *Installing and Configuring Cogility Studio* describes the installation and configuration of your application server, database and Cogility Studio.
- *Getting Started with Cogility Studio* is a brief overview of Cogility Studio.
- *Modeling with Cogility Studio* tells you how to build a model-driven enterprise application using Cogility Modeler and associated tools.
- *Using Actions in Cogility Studio* provides a reference to modeling action semantics for use with Cogility Studio.
- *Change Management in Cogility Studio* describes the change management system for models and model artifacts.
- *Model Deployment & Execution in Cogility Studio* is a guide to application monitoring, maintenance and migration, and describes the utilities that you can use to test and monitor your model deployed as a enterprise application.

Several white papers on various topics are also available to further your understanding of enterprise application integration, business process management, model driven architecture and other related topics. See the Cogility website:

<http://www.cogility.com>.



Introduction to Cogility Modeler

Cogility Modeler is an integrated development environment (IDE) for the collaborative authoring of an enterprise system model. Like an IDE for developing code and code objects, it provides a single tool for working with many types of system artifacts. Unlike a typical IDE concerned only with individual entities, Cogility Modeler provides visibility over an entire system and semantic consistency checking between artifacts of a system. That system may include any or all business processes that span the enterprise, the messages between components of that system, and the data and operations of that system. All of these are described as artifacts of an enterprise model created in Cogility Modeler. The model may be developed collaboratively with contributions from people at both the high-level, conceptual end and the low-level, implementation end. The model becomes fully executable when deployed as a J2EE application.

An enterprise model consists of artifacts grouped in the following areas:

- **Information modeling**

The artifacts of the information model describe the business objects of the enterprise. A Customer, for example, is represented as a class artifact with such attributes as first name, last name, customer ID, address and so forth. The information model describes both the data in the *static model* and its behavior in the *dynamic model*.

See “[Information model](#)” on page 29.

- **Message modeling**

The message model is concerned with the communication between model artifacts and applications integrated by the model. Message artifacts represent JMS messages. Event artifacts define either stimulus or response from runtime behaviors. Conversion artifacts provide the mapping between the two.

See “[Message modeling](#)” on page 81.

- **Behavior modeling**

A subset of the information model, the dynamic model is also called the behavior model. Behavior artifacts describe business processes; a *state machine* is an example of a behavior artifact.

See “[Behavior model](#)” on page 109.

- **Transformation modeling**

At the heart of an integration model, transformation objects describe the way data from an application on one domain gets mapped to another application in a different domain. Transformation maps, classifier maps and feature maps are all examples of transformation artifacts.

See “[Transformation model](#)” on page 145.

- **Web service modeling**

Web service modeling artifacts provide for communication between the model’s J2EE application and other systems and applications. They also provide a way for any HTTP client to access application data in the model.

See “[Web Services Model](#)” on page 201.

Configuration management

Cogility Modeler features configuration management tools for maintaining model artifact versions from several authors. Before any work on a model may be performed, a base-line model common to all authors must be identified. That model is identified by a configuration management (CM) context.

Current CM context

The model you select during log in loads into the current CM context. The *current context* is what you see on the screen when you run Cogility Modeler. The model displayed is said to be in the current context. The authoring repository maintains the objects in the current context and it may hold objects not in the current context. That is, the authoring repository may have previous versions of objects and entire models. See “[Authoring repository](#)” on page 19.

If you are running Cogility Modeler with a repository that contains one or more models, or several model versions, you must select which of those you want to load into the current context when you log in. Once you are running Cogility Modeler you can always remove a model from the current context and add model objects or an entire model from the authoring repository to the current context. See “[Adding a model from the repository](#)” on page 32 of the guide, *Change Management in Cogility Studio*.

To run Cogility Modeler with a pre-existing model, you must specify the repository containing the appropriate CM context. That CM context consists of a configuration map and a configuration map version.

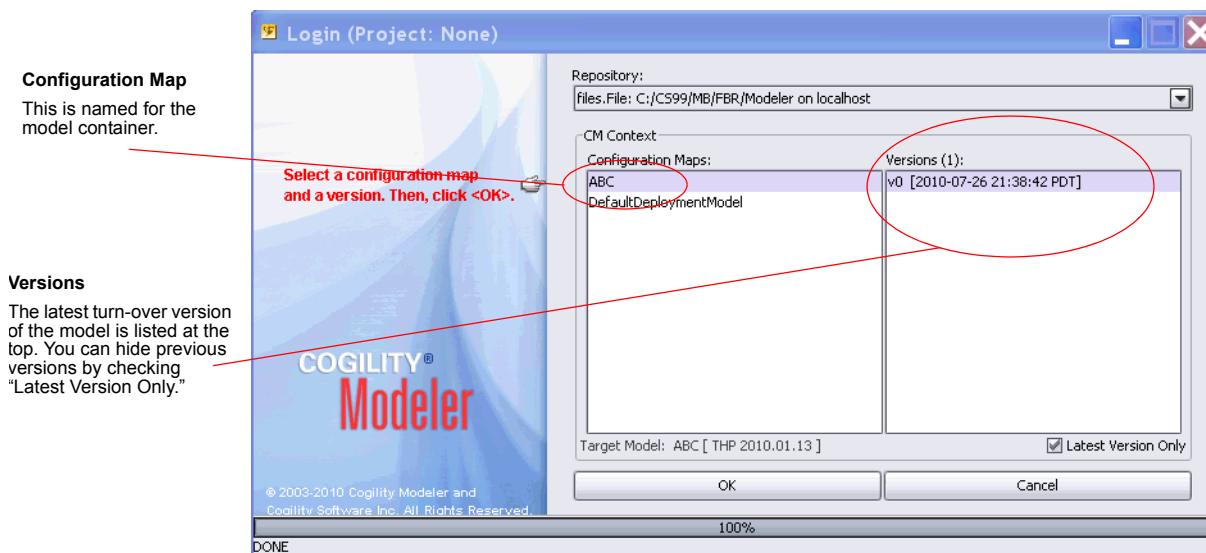


Figure 1-1: Cogility Modeler login screen

Configuration map

The configuration map identifies a particular model configuration consisting of a deployment model and an enterprise system model. The deployment model identifies how the system model will be

deployed: with a specific application server and database combination. See “[Configuration Parameters](#)” on page 25. The system model associated with that deployment model is shown in Configuration Map pane of the log-in screen, if the system model exists in the authoring repository. If the repository is empty, as when you log into Cogility Modeler for the first time, only the default deployment model loads into the current context.

When you create and turn over a system model in Cogility Modeler a configuration map named for that model and associated with the deployment model is also created. (See “[Turn-over version](#)” on page 23 of the guide, *Change Management in Cogility Studio*.) If you import a model from a file and that model is not already located in the current context or in the repository, a configuration map for it is not created until you place the model under change management explicitly.

Once you have created a system model, associated it with a deployment model and turned over the model, it will be available from the authoring repository in a CM context the next time you log in and select that repository.

For a given repository, each of the system models therein has a configuration map named for the system model and deployment model. The configuration map will appear in the configuration map pane along with the default deployment model. If you select only the default deployment model to load into the current context, no system model will be associated with it. The system model you then create and then turn over may be associated with the default deployment model and stored in the authoring repository under a new CM context. That new CM context will always be named for the system model, regardless of whether the system model is associated with the default deployment model or a named deployment model.

When you select a configuration map, the name of the system model appears next to Target Model below the Configuration Maps field.

Version

A system model may have an unlimited number of versions. Also, the artifacts within the model may have versions. That is, you can apply versioning to either a specific model artifact or to the model in its entirety. When you turn over a model you create a version of the entire model. Likewise, you may turn over specific objects to create versions of them. See “[Turn-over version](#)” on page 23 of the guide, *Change Management in Cogility Studio*.

In the figure above there are four versions of the Simple Model plus a fifth version, listed first in italics with the word *modified*, that has not been turned over. This is the version that was last loaded in the current context. Its author logged out of Cogility Modeler without first versioning or turning over the latest editions to the model. These editions were saved in the repository, however, and are always available as the version with this *modified* label. Once this model is versioned, it displays with a label like the other versions.

Also in the figure above, the checkbox labeled “Latest Version Only” is not checked, so all of the versions available are displayed. If this box is checked, only the latest version would be shown. In the example above, this is the *modified* version.

Logging into the current context

When you log into Cogility Modeler, you specify the repository from which to load a model. The model is loaded into the current context (see “[Current CM context](#)” on page 16).

To log in to Cogility Modeler:

1. From the Windows Start menu, select All Programs > Cogility Studio > Cogility Modeler > Cogility Modeler Login.

2. In the Cogility Modeler console window, at the command prompt, enter the letter corresponding to the target application server and press **Enter**.
If you have more than one application server configured on your system, you will be prompted to indicate which application server to use for this session with Cogility Modeler. Enter **W** for WebSphere, **L** for WebLogic, or **O** for OAS.
3. In the **Login** panel, under **Repository**, from the drop-down menu, select the repository that contains the model you want to load.
If you have only one repository configured, that repository is selected automatically. The repository is established in a file directory on your system, and the selections take the format, **files.File: <file directory> on localhost**. See "["Authoring repository" on page 19](#)".
If you see a connection error message that says, "unable to create a local session," you may have either attempted to run two instances of Cogility Modeler with the same repository (which is not allowed), or you have a lock file left over from the previous session. See "["Exit Cogility Modeler" on page 18](#)" for more information.
4. For **CM Context**, in the **Configuration Maps** field, select the configuration map and click **OK**.
The first time you run Cogility Modeler and any time you run it with an empty authoring repository, you do not have to select the CM Context or click OK; the **DefaultDeploymentModel** deployment container is loaded into the current context for you. This deployment model contains default deployments, each with specific configuration parameters for your model. You can create several deployment containers, each with different deployments that allow a model to be deployed on different application server and database configurations. See "["Configuration Parameters" on page 25](#)".

Exit Cogility Modeler

There are three ways to exit Cogility Modeler:

- From the File menu, select Exit.
- Click the close window icon  in the upper right corner.
- Shut down the Cogility Modeler process at the system level.

Each time you run Cogility Modeler for use with a particular repository, a session lock file for that repository is created. The existence of this file prevents more than one Cogility Modeler instance from working with the same repository. When Cogility Modeler exits, the lock file is deleted. During a working session (when Cogility Modeler is running) for a particular repository, the lock file is located with the repository files and named Session.slk. For example, if you installed Cogility Studio in D:/CS, the default full path to the repository is D:/CS/MB/FBR/Modeler. See "["Authoring repository" on page 19](#)" for more information about establishing the location of the repository.

Of the three ways to exit Cogility Modeler listed above, only the first guarantees that the lock file will be deleted upon exit. Using either the second or third way to exit may leave the lock file intact and prevent you from logging into Cogility Modeler to work with the locked repository. The following connection error appears when this is the case:

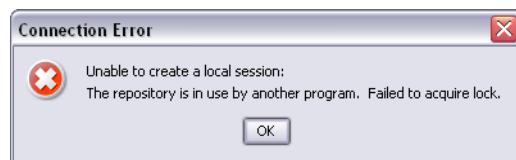


Figure 1-2: Session.lck file existence warning

To correct this condition, navigate to the location of the repository and delete the lock file. Then run Cogility Modeler (see "["Logging into the current context" on page 17](#)").

Authoring repository

There are two repositories that you work with in Cogility Studio: the **file-based authoring repository**, discussed here, where changes to your model are tracked as you work with the model, and the database run-time repository, where your model is deployed to for execution. See “[Run-time respiratory](#)” on page 267.

Each Cogility Studio installation can define and maintain an unlimited number of authoring repositories. Each is a location on the local system for a model under development. As the user of that installation, you configure Cogility Studio for each repository by editing your CustomConfigurations.txt file (see “[Authoring repository configuration](#)” on page 19), then run the Load Persistent Repository utility to create the repositories (see “[Load Repository utility](#)” on page 20). Because you may want to work with many different models, you can configure an authoring repository for each of them.

Cogility Studio’s configuration management facility operates on all objects defined in the local authoring repository. It maintains all version and branch information for the model in a specific repository. The model in an authoring repository may have many versions.

The files in the authoring repository are of the OOF format and may be read only by Cogility Studio. However, you may back up the files in a separate location and copy them back to the location of the authoring repository in the event that the repository becomes corrupted.

A model and all of its version information may be saved and imported or exported to other locations. See “[Model artifacts](#)” on page 24.

Authoring repository configuration

When you run the utility, Load Persistent Repository (see “[Load Repository utility](#)” on page 20), the utility reads all existing configuration files for repository specifications. The default settings for the authoring repository are found in the section, COGILITY MODELER REPOSITORY. They appear as follows:

```
#com.ohana.object.persistence.db.count=1
#com.ohana.object.persistence.db.connection=1
#com.ohana.object.persistence.db.class.1=files.File
#com.ohana.object.persistence.db.interface.1=FileRepository
#com.ohana.object.persistence.db.servername.1=localhost
#com.ohana.object.persistence.db.databasename.1=%OEMHOME%/FBR/Modeler
#com.ohana.object.persistence.db.username.1=
#com.ohana.object.persistence.db.password.1=
```

The first line specifies how many repositories to create. The second line specifies which (of several possible) repositories shall be selected when the autologin parameters are set. The subsequent lines describe the settings for each named repository.

The third line specifies the type of repository, in this case a file-based repository. This is currently the only type that Cogility Studio supports.

The fourth line specifies the interface that works with the repository; this setting is for internal use only.

The fifth line specifies the domain location of the repository. Currently Cogility Studio supports repositories located only on the local host (where Cogility Studio is located).

The sixth line specifies the file location of the repository. You may specify any valid directory on any valid drive. If the directory does not exist, Load Persistent Repository creates it when you select it.

Note that %OEMHOME% is the location of the repository directory (labeled /MB) under the location where you have installed Cogility Studio. For example, if you installed Cogility Studio in D:/CS, the default full path to the repository is D:/CS/MB/FBR/Modeler.

The last two lines, for the username and password should be left valueless. These settings are currently not used, and any values provided are ignored.

You can create as many authoring repositories as you need. You may want several authoring repositories if you have several models or several versions of a model that you want to keep separated. For example, if you wanted to create three repositories, the settings would appear as follows:

```
com.ohana.object.persistence.db.count=3
com.ohana.object.persistence.db.connection=1

com.ohana.object.persistence.db.class.1=files.File
com.ohana.object.persistence.db.interface.1=FileRepository
com.ohana.object.persistence.db.servername.1=localhost
com.ohana.object.persistence.db.databasename.1=%OEMHOME%/FBR/Modeler1
com.ohana.object.persistence.db.username.1=
com.ohana.object.persistence.db.password.1=

com.ohana.object.persistence.db.class.2=files.File
com.ohana.object.persistence.db.interface.2=FileRepository
com.ohana.object.persistence.db.servername.2=localhost
com.ohana.object.persistence.db.databasename.2=%OEMHOME%/FBR/Modeler2
com.ohana.object.persistence.db.username.2=
com.ohana.object.persistence.db.password.2=

com.ohana.object.persistence.db.class.3=files.File
com.ohana.object.persistence.db.interface.3=FileRepository
com.ohana.object.persistence.db.servername.3=localhost
com.ohana.object.persistence.db.databasename.3=%OEMHOME%/FBR/Modeler3
com.ohana.object.persistence.db.username.3=
com.ohana.object.persistence.db.password.3=
```

Note: The first line specifies how many authoring repositories are defined; in this example, three. When the Load Persistent Repository utility runs, it lets you select from a list of repositories to load. The length of the list is determined by this setting. If you specify two (2) repositories, only the first two configured repositories are listed, even if you have written a third group of settings in the customConfigurations.txt file.

Load Repository utility

The Load Persistent Repository utility creates an authoring repository. The utility initializes the repository with the meta-model schema that describes the modeling artifacts such as events, classes,

attributes, message conversions, and so on. You use these artifacts to create your model. A newly created repository contains a default deployment model.

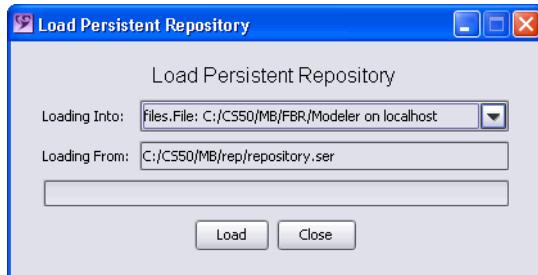


Figure 1-3: Load Persistent Repository tool

Loading the repository

Caution: The utility loads the meta-model schema from the repository.ser file installed with Cogility Studio. Each time you run the Load Model Repository utility, a new, empty repository is created. The old repository and any models and versions stored in the repository's location are deleted.

To create an authoring repository:

1. Configure your repositories.
See “[Authoring repository](#)” on page 19.
2. From the Start menu, select All Programs > Cogility Studio > Cogility Modeler > Load Persistent Repository.
3. From the drop-down list next to **Loading Into**, select the repository you wish to load into the current context.
4. Click **Load**.

Model container

The model container holds all model artifacts and appears at the top of the tree in Cogility Modeler. To create a system model, you start by adding the model container. You can then create other model artifacts. Both the fixed model object definitions and the dynamic data object definitions are mapped to tables of the run-time repository. The fixed model objects are those that describe the application you are modeling, for example, a Web Service named UpdateCustomerAddress, a behavior named orderProcess, or an Operation named getStatus. The dynamic data objects are those that represent your enterprise data, for example, a Customer or Order definition. Both the fixed and dynamic objects comprise the model’s schema. The model container holds the model’s schema.

The model you work with in Cogility Modeler is kept current, with a record of each change as it is made, in the authoring repository. See “[Authoring repository](#)” on page 19. The authoring repository may hold more than one model container, just as Cogility Modeler may display more than one model container. However, the best practice is to keep only one model container displayed in Cogility Modeler at a time to avoid confusion and the risk of corrupting your model. Usually you would have more than one model in your repository for the purposes of comparing different versions of a model. See “[Model comparison](#)” on page 25 of the guide, *Change Management in Cogility Studio*.

When you run Cogility Modeler with an empty repository, no model container is shown in the tree view because none exists in the repository. An empty repository is one that you have created as

described in “Load Repository utility” on page 20. To the empty repository you either add a new model container (see “Creating a model container” on page 22) or import an existing model container (described in “Model artifacts” on page 24).

Creating the model container is the first step toward defining your desired application. When you add a model container to a repository, the container is empty. Thereafter, you add all your model objects to this container.

Creating a model container

To creating a model container:

1. Click the **Add a Model** button  or from the **Selection menu**, select **Domain Modeling > Add a Model**.
2. In the editor view, under the **ModelDefinition** tab, in the **Name** field, enter the schema name.

Note: The name must begin with an alphabetic character and can contain any combination of alphanumeric characters or underscores. The name cannot contain spaces.

If the DeploymentModel object is selected, the **Add a Model** button will not be enabled. You must first unselect this object. To do so, hold down the Ctrl key and, with the mouse pointer, click on the model object again. Follow these steps to unselect any objects, leaving nothing selected in the tree view.

Packages

Packages allow you to group various model artifacts relating to a single functional area. You can group various modeling artifacts such as IWSs, IWS deployments, Events, Messages, M2E, E2M, and so on. The exceptions are those artifacts which define the information model. Class (MIM and DIM) artifacts are themselves containers and store their component parts such as attributes, operations, and behaviors. These component parts define the Class and therefore cannot be moved outside the Class container. Likewise, Classes either define the MIM or a specific DIM, and cannot be moved outside the parent.

Parts of the modeled business logic, such as state machines and operations on MIM classes, can also belong to a functional group. You may want to group one or more state machines and operations into a package. However, this is not possible because the state machines and operations physically reside in the MIM class, which resides in the MIM. You can use a virtual package within the package to include these items. See “Virtual Packages” on page 23 for more information.

By default, packages appear in the tree view alphabetically by name, like any other artifact, but you can sort them to the top of the tree. Also, they may be displayed with or without the package icon. See “UI menu” on page 294.

Creating a package

To create a package:

1. In Cogility Modeler’s tree view, select the model container, and click the package button .

Note: You may also create a Package within the MIM, a DIM, or another Package.

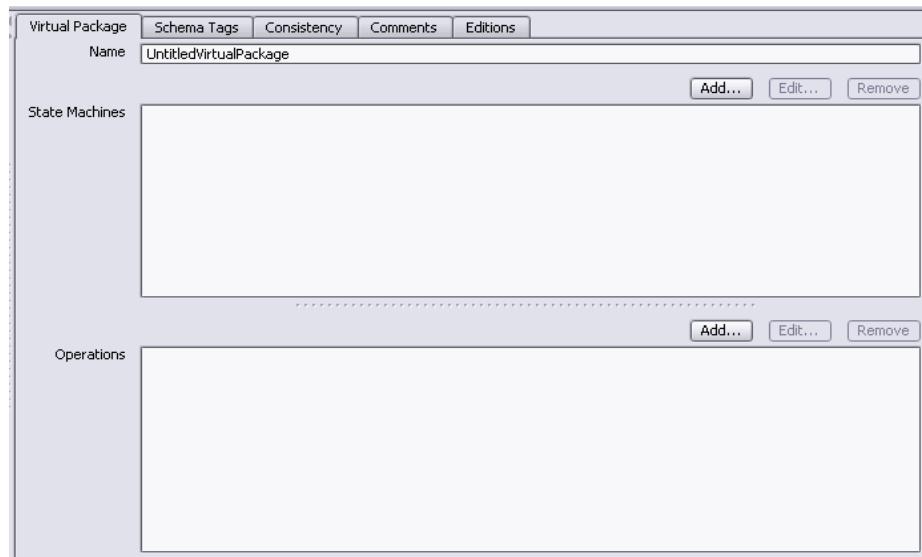
2. In the editor view, enter a name in the name field.

Virtual Packages

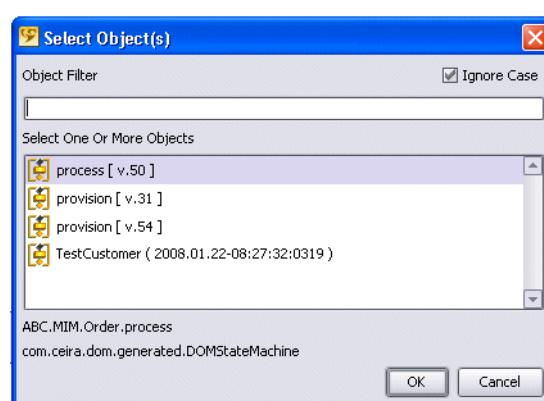
A Virtual Package has no functional behavior, but is an organizational tool. A Virtual Package provides pointers to class operations and behaviors which cannot be added to a Package.

To create a virtual package:

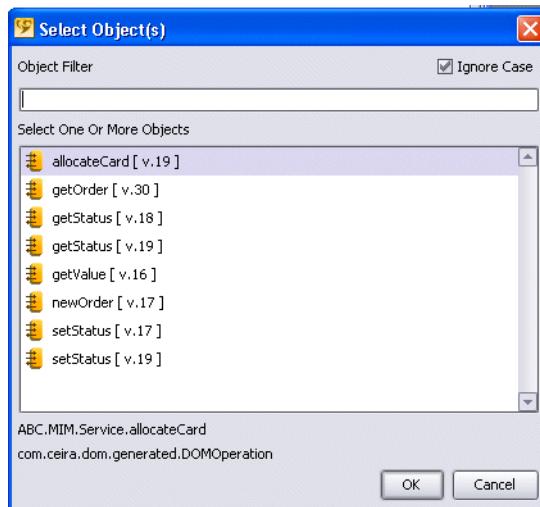
1. In Cogility Modeler's tree view, select the package under which to create the virtual package.
2. From the Selection menu, select **Domain Modeling > Add Virtual Package**. Or from the "Adds" menu dropdown  , click the Add Virtual Package button .



3. Enter a name for the Virtual Package in the **Name** field.
4. To add a state machine:
 - a. Click **Add** above the State Machines field.
 - b. Select the state machine from the list.



- c. Click OK.
5. To add an operation:
- Click Add above the Operations field.
 - Select the operation from the list.



- c. Click OK.

Note: Behaviors and/or Operations may have the same name, making it difficult to determine which object to select. When an object is selected, its fully qualified name is displayed at the bottom of the dialog allowing you to see if you have selected the correct object.

Model artifacts

When you create a model artifact, it appears in the tree view and its name appears in italics. Likewise, when you edit a model artifact, its name appears in italics. The italics indicate that the artifact is *mutable*. This means that the artifact in the current context is different from the last turned-over version of the artifact in the repository. Turning over the model or versioning the artifact creates an *immutable* version of that artifact in the current context and in the authoring repository. Immutable artifacts are those that appear in regular face type, not italicized. When you turn over the model, these artifacts will not be changed, nor will they be different from the artifacts of the same names in the previous turn-over version. See “[Turn-over version](#)” on page 23 of the guide, *Change Management in Cogility Studio*.

Model artifacts may be imported from a COG file set located in a file system. When you have completed a series of changes to your model and want to share it with other developers, you can export the model file set. For example you might want to check the model into your source control system. This is described in “[Model export](#)” on page 25 of the guide, *Change Management in Cogility Studio*.

Note: To avoid corrupting your model, do not import into the repository or current context model artifacts that are schematically different from the model already in the repository. A repository should contain only one model and all of its versions. Do not place two or more schematically different models in the same repository.

Deleting model artifacts

When you delete an artifact, you are deleting it from the current context only. You are not deleting it from your authoring repository. For information about recovering deleted artifacts and models, see “[Reverting elements](#)” on page 12 of the guide, *Change Management in Cogility Studio*.

Note that you cannot push a model or model artifact over an extant model when the model or artifacts you are pushing are missing elements in the extant model. Such a model is said to have an non-additive schema. Updating a model with a non-additive schema is described in “[Non-additive schema updates](#)” on page 32 of the guide, *Model Deployment & Execution in Cogility Studio*.

To delete an artifact:

1. In Cogility Modeler’s tree view, select the artifact.
 2. In the tool bar, click the Delete button .
- You can also open the **Selection** menu, and select **Edit > Delete Selection**, or right-click on the artifact and select **Edit > Delete Selection**.

Configuration Parameters

Configuration parameters can pertain to both the modeling environment (Modeler) and the execution environment (the application server). CogilityStudio defines configuration parameters as key-value pairs. Default values exist and are encoded into the product. However, users may wish to modify various configuration parameters for their unique environment. Those configuration parameters that are safe to modify are listed in the text file %OEMHOME%\Config\defaultConfigurations.txt. The subject of configuration in general is discussed in “[Configuration](#)” on page 13 of the guide, *Model Deployment & Execution in Cogility Studio*.

Reloading configuration parameters into Cogility Modeler

If it is necessary to modify configuration parameters during a Modeler session, then those changes must be reloaded into the cache in order for the modifications to take affect.

To reload configuration parameters to Cogility Modeler:

1. Edit your customConfiguration.txt file or run Cogility Project Configuration Editor, as needed.
2. From the **File** menu, select **Reload Configuration Parameters**.

Note: You can also reload configurations parameters for the application server and for the Action Pad.

Deployment

Once you have defined your application in terms of a model, you deploy it as an executable application on a J2EE application server for use with a specific run-time repository. You specify the deployment configuration as described in “[Deployment Model](#)” on page 255.

As shown in the figure below, deployment is the process of taking a model you have created in Cogility Modeler and pushing it into execution as a enterprise application. See “[Pushing the model into execution](#)” on page 273, for instructions on pushing an entire model into execution, assuming an entirely new deployment. However, different deployment options are available to accommodate different types of revision to a running enterprise application. See “[Push modes](#)” on page 27 of the guide, *Model Deployment & Execution in Cogility Studio*.

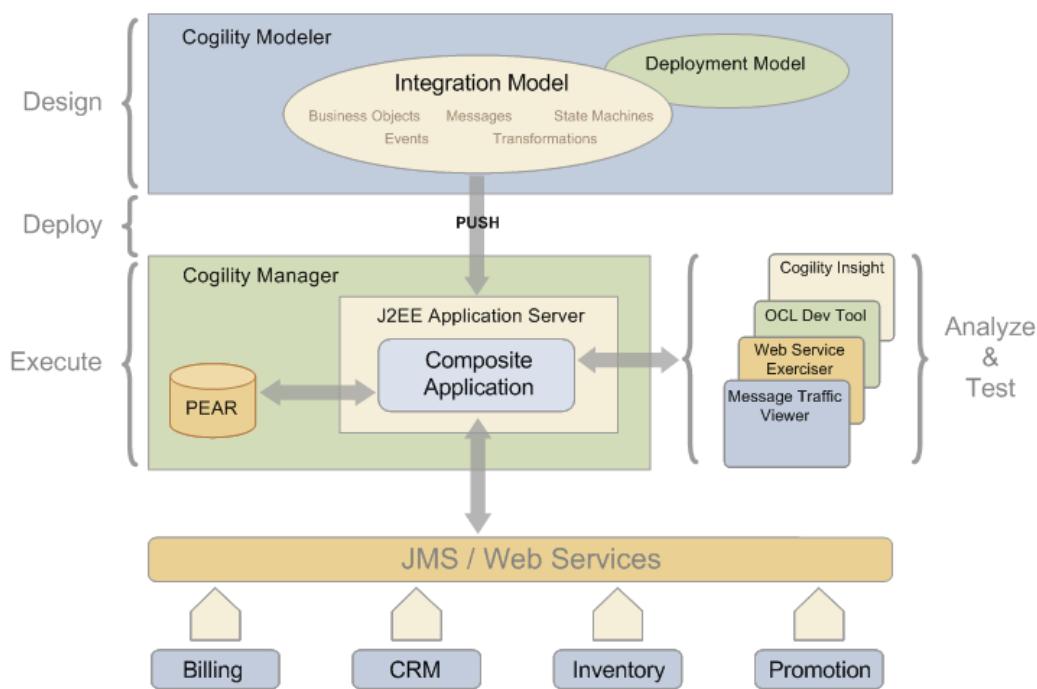


Figure 1-4: Deployment illustrated

During deployment (Push), Cogility Modeler will create and deploy to the target application server, an enterprise archive file (.ear). This archive will contain deployment descriptors, web archive for various web applications (e.g. Insight), enterprise java bean to serve as message listeners supporting JMS and web services, and source and third party code..

Pushing the model creates the tables in the run-time repository (see “[Run-time respiratory](#)” on page 267) for the model objects such as the M2E conversions, events, web services, and so on, which serve as the applicaiton metadata, and for the business objects that are created at run-time (such as Customers, Addresses and Products).

During push you can force the model’s action semantics to be recompiled, see “[Force action semantics recompilation](#)” on page 32 of the guide, *Model Deployment & Execution in Cogility Studio*. You can also force the push of artifacts to the application server. See “[Application server only push](#)” on page 29 of the guide, *Model Deployment & Execution in Cogility Studio*. Both of these options are applicable when migrating models to newer versions of Cogility Studio or deploying the model on several application servers.

Cogility Manager is the run-time engine in which the deployed composite application runs. It enables a model created in Cogility Modeler to run as an executable application on a specific application server with a specific database for the run-time repository. See “[Cogility Manager](#)” on page 11 of the guide, *Model Deployment & Execution in Cogility Studio*.

Execution

With your model fully deployed, you can run the several tools in Cogility Studio to test and monitor the enterprise application. You also have various options for maintaining your deployed enterprise application. These are described in “[Cogility Manager](#)” on page 11 of the guide, *Model Deployment & Execution in Cogility Studio*.



2

Information model

The information model describes the business objects that interact across the enterprise. An information model must include a master information model (MIM) and may include one or more distinct information models (DIMs). The master information model describes the information for the entire business enterprise, and the distinct information model describes the information for the individual domains.

Master information model (MIM)

The master information model describes the logical organization of the enterprise as a whole, in terms of its business objects and how they are associated with one another. Objects in the MIM may be hybrid representations of business objects from different domains. For example, a Customer object on an Order Entry system might have a customerID, and an Account object on a Billing system might have an accountID. The MIM would represent both objects as one MIMCustomer object (based on a UML class) that would have both a customerID and an accountID. In this way the MIM classes represent the common denominators of all integrated application objects.

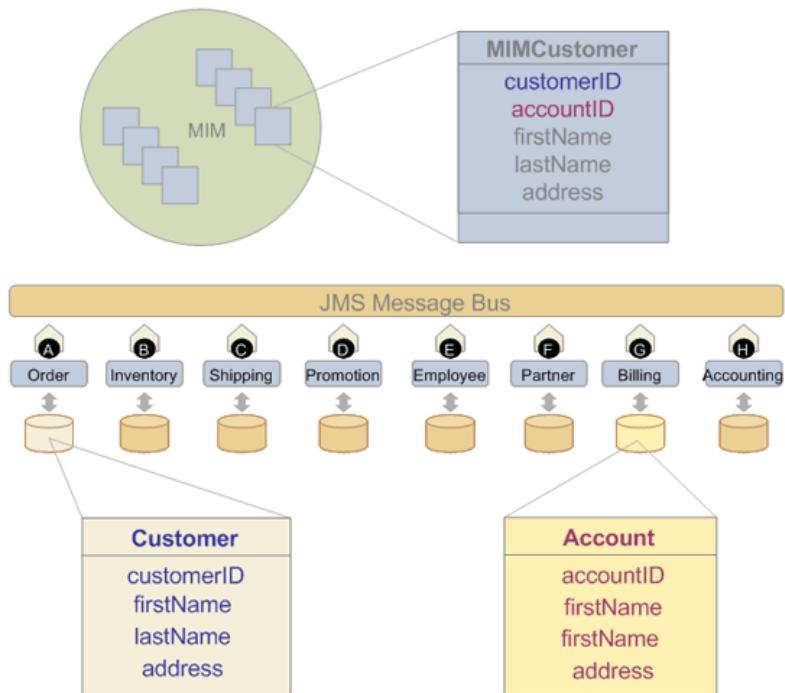


Figure 2-1: A MIM models the business objects common to the entire enterprise

An enterprise model must have one and only one master information model. It models the business objects that interact with each other and with application data across the enterprise. These objects are essential to the model's business processes and overall enterprise functionality. In a model of only one application, the master information model describes all of the business objects for the composite application.

In a model with several constituent applications, some of the objects of one application may have no need to interact with objects of the other applications. They play no part in integrating any two applications. The classes for these objects, therefore, are not modeled in the master information model. For those business objects that do have applicability across the enterprise, the master information model provides a representation of the business objects that is common to all of the integrated applications. That is, it represents all of the business objects in such a way that all of the integrated applications can use them. This is explained further under ["Distinct information model \(DIM\)" on page 30](#).

For those objects that interact across the enterprise, you describe their classes, attributes, operations, behavior and associations within the master information model.

To create a MIM:

1. In Cogility Modeler's tree view, select the model container.
2. Click the **Add a Master Information Model (MIM)** button  or from the **Selection** menu, select **Domain Modeling > Add Master Information Model (MIM)**.
3. In the MIM editor, under the **Information Model** tab, enter the name for the MIM.
4. Create the MIM classes.
 - a. With the new MIM selected in the tree view, click the **Add a MIM Class** button  or from the **Selection** menu, select **Domain Modeling > Add MIM Class**.
An untitled MIM class is displayed in the tree view, and the class editor is displayed in the content view.
5. Define the class. See ["Defining a class" on page 37](#).

Distinct information model (DIM)

An integration model may have one or more distinct information models. Each represents the business objects of an external application. The model needs a view of an external system's data so that it can communicate that data to other systems; the distinct information model provides this view.

If your enterprise model does not integrate applications on disparate domains, a distinct information model is not required, however. If you already have a homogenous set of business objects, that is, they can share data among them, you do not need a distinct information model; the master information model is all you need to represent your business objects.

In an integration model of two or more applications, where data must be shared between the applications and where that data may be represented differently on each application, the data must be *transformed* from one representation to another to achieve integration. When transformations are

required, your model must include a distinct information model for each application that sends or receives data to and from the other applications.

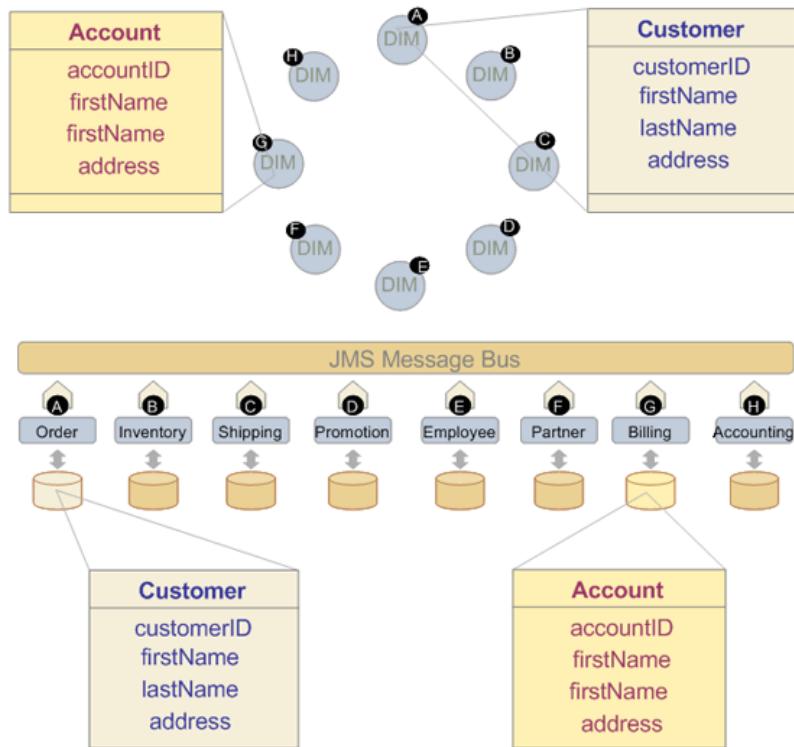


Figure 2-2: A DIM models the business objects in each of the integrated applications in a model

As the domain's application works with several classes, so must the DIM include representations of those classes that are used in the model. In the previous example, the Customer and Account classes were part of the Order Entry and Billing systems, respectively. Each of these classes is represented as a UML class in the Order Entry and Billing DIMs as well.

The DIM represents each distinct application in the model. While the DIM describes the domain application, the MIM serves as a centralized hub and an intermediary through which data may be communicated in its path from one application to another; the MIM, with its hybrid classes, can take data from and describe data to more than one application.

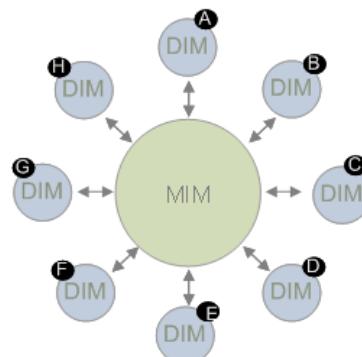


Figure 2-3: The MIM provides a hub for communication between integrated applications

Think of the MIM as the “inside” and the DIM as the “outside” of the information model.

For example, a date attribute on an Order Entry system might be represented in the MM/DD/YYYY format, while a date on the Billing system might be represented as DD.MM.YYYY. When a new customer submits an order, the information about that customer must be communicated to the billing system so that it may create an account for the customer. As illustrated below, the Customer.creationDate from the Order Entry system must be transformed into an Account.creationDate on the Billing system.

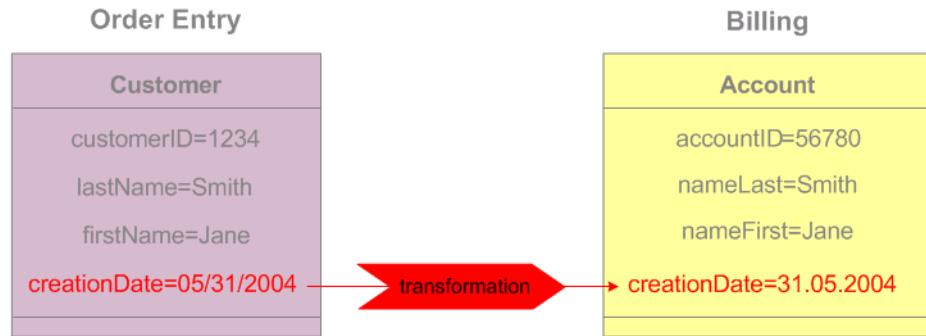


Figure 2-4: Classes participating in a transformation

In order to perform this transformation, your model needs, first, a representation of the data on each system, a distinct information model of each application. It also needs a homogenized representation of the data, one that represents the common denominators of the data, a master information model of all applications. Finally, it needs transformation objects to carry out the transformation.

As illustrated below, the distinct information models represent each application, the master information model represents both. To communicate data from one application to another, the data must be transformed into a master information model representation, then transformed into a distinct information model representation. In the illustration, the transformations are represented by red arrows.

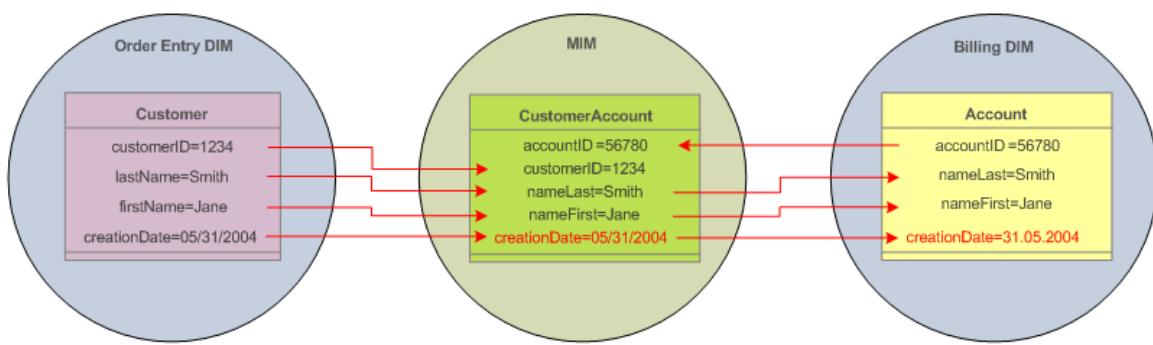


Figure 2-5: DIM classes and a MIM class participate in transformations

Notice that the master information model acts as an intermediary in a transformation. It can talk to both distinct information models; it is the “source of truth” about what business information participates in the enterprise. The distinct information models each represent the external applications. They may include business objects that do not participate in the enterprise integration, but they must at least include those business objects that do.

In an integration of several applications, the distinct information models are like the spokes of a wheel, and the master information model is the hub. This design reduces data redundancy and provides for greater scalability.

To create a DIM:

1. In Cogility Modeler's tree view, select the model container.
2. Click the **Add a Distinct Information Model (DIM)** button  or from the **Selection** menu, select **Domain Modeling > Add Distinct Information Model (DIM)**.
3. In the DIM editor, under the **Information Model** tab, in the **Name** field, enter a name.
4. Create the DIM classes.
 - a. Click the **Add a DIM Class** button  or from the **Selection** menu, select **Domain Modeling > Add DIM Class** from the menu bar.

An untitled DIM class is displayed in the tree view, and the class editor is displayed in the content view.
5. Define the class. See “[Defining a class](#)” on page 37.

Classes

Cogility Studio describes classes according to the standard UML class specification type. Classes describe business objects for your model. Within the MIM and within each DIM, you model each business object as a class. A class is a named model artifact. When the model is pushed during deployment, each class becomes a table in the run-time repository. During execution, this table is populated with class instances as these are created by the external applications. A class can be associated with specific behaviors and may have operations and attributes. [Figure 2-5 on page 32](#) illustrates DIM classes and a MIM class.

Classes are represented in Cogility Modeler's tree view with a class icon: . If the class has an associated behavior, the icon includes a B in the bottom left corner: . If the class has an operation, the class or subclass where that operation is defined with a method has an icon that includes an O in the bottom right corner: . Classes with both a behavior and an operation display with both a B and an O: .

Superclass

A class can inherit attributes and operations from its superclass. Class artifacts may also inherit associations and behaviors from a superclass. Any class can be used as a superclass.

For example, in the diagram below, the **GoldSLA** and **BronzeSLA** classes inherit the **ackReceipt()** and **processTicket()** operations from the superclass **ServiceLevelAgreement**.

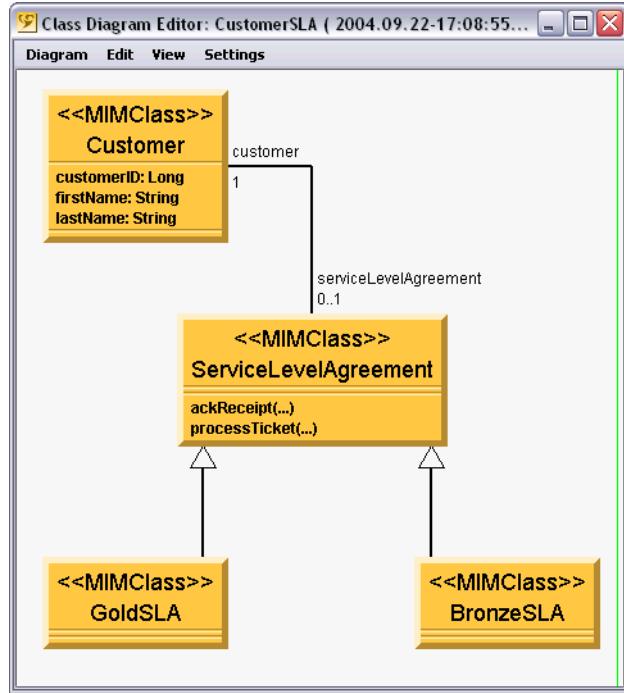


Figure 2-6: Classes inherit from a superclass

The default superclass for all classes is **Element**. If you are not selecting a superclass, **Element** will be shown in the Superclass field for all classes.

Abstract class

An abstract class is a class that cannot be instantiated; it is used as a superclass for other classes and serves as a template for classes that inherit from it. The inheriting classes can be instantiated.

To define a class as an abstract class, you check the **Abstract class** checkbox when creating the class.

Attributes

When you define a class, you can define one or more attributes to describe the details about the class. Each attribute has a name and data type. For example, in Figure 2-7, the MIM Service class has one attribute “type” of datatype String.

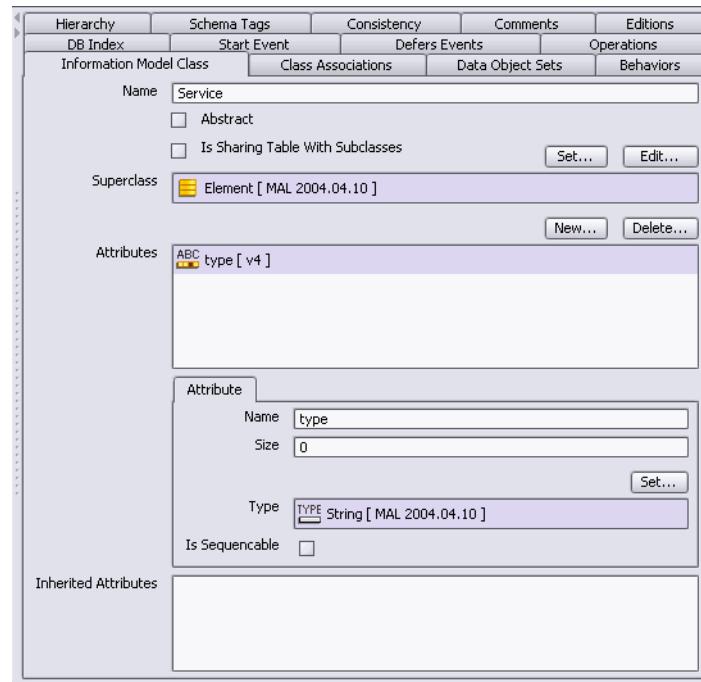


Figure 2-7: Attributes defined for a class

Attributes can also be inherited from a superclass, as shown in Figure 2-8. In this case the PremiumService class inherits the attribute “type” from the Service superclass.

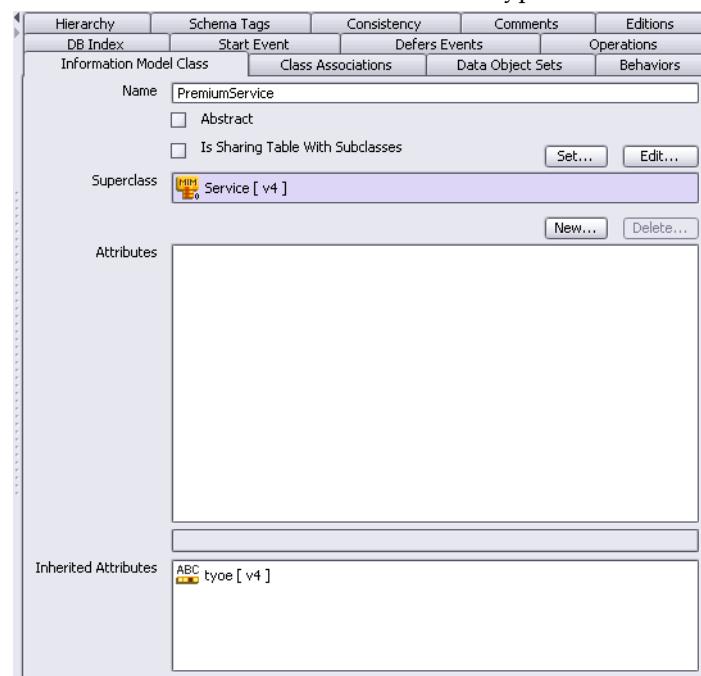


Figure 2-8: Inherited Attributes

The following are the primitive data types used in the model and action semantics. When you define an attribute's datatype, one of the following symbols is displayed in Cogility Modeler depending on its datatype:

	Byte Array		Integer
	Boolean		Long
	Double		String
	Float		Timestamp

The Timestamp data type is a Cogility-defined data type. One possible use for the Timestamp data type is when Cogility is used to populate a datetime field, but it is expected that another SQL-based application is going to digest that data - for example to create a report. A Cogility attribute of type Timestamp is mapped to a similar database native column type (usually timestamp or datetimestamp). When accessing this data, the database's library of SQL functions can be used to manipulate the datetime data. Internally, in Cogility, this datatype behaves much like a Long. Additional utility methods (for example to take a timestamp value and format it according to DDMMYY format) are available in the class `com.cogility.Timestamp`.

Sequenceable attributes

Checking the **isSequenceable** checkbox causes the value of this attribute to be generated in the run-time repository during execution. The run-time repository automatically assigns values to sequenceable attributes. Only Long type attributes can be sequenceable.

In your `CustomConfigurations.txt` file, you can set the initial number and the increment values. See “[Sequenceable attribute settings](#)” on page 22 of the guide, *Model Deployment & Execution in Cogility Studio*.

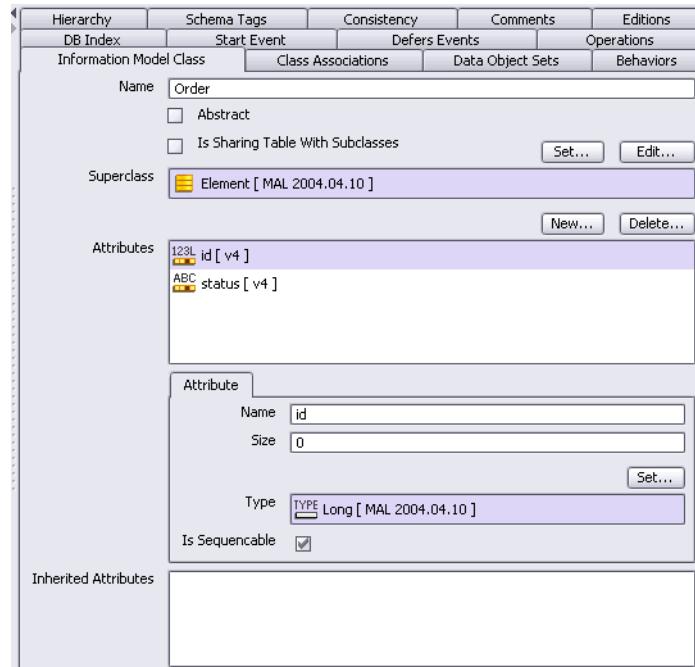


Figure 2-9: A sequenceable attribute definition

See “[Customizing installation configuration](#)” on page 13 of the guide, *Model Deployment & Execution in Cogility Studio* for more information about `CustomConfigurations.txt`.

POID attribute

All objects in Cogility Studio have a unique identifier. This identifier is a String called a persistent object identifier (POID), it is unique within the global boundaries of Cogility Studio.

You may find it useful to locate business objects by POID. This is described in “[LocateByIdentity](#)” on [page 69](#) of the guide, *Using Actions in Cogility Studio*. To access a business object by its POID, you must explicitly define a `poid` attribute on the business object’s class. The attribute must be named `poid` and it must be a String type. Such an attribute is shown in the figure below.

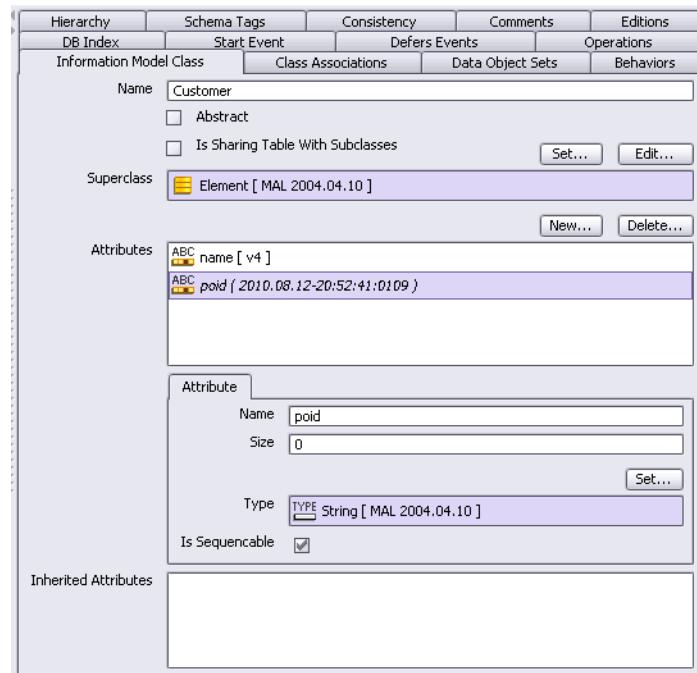


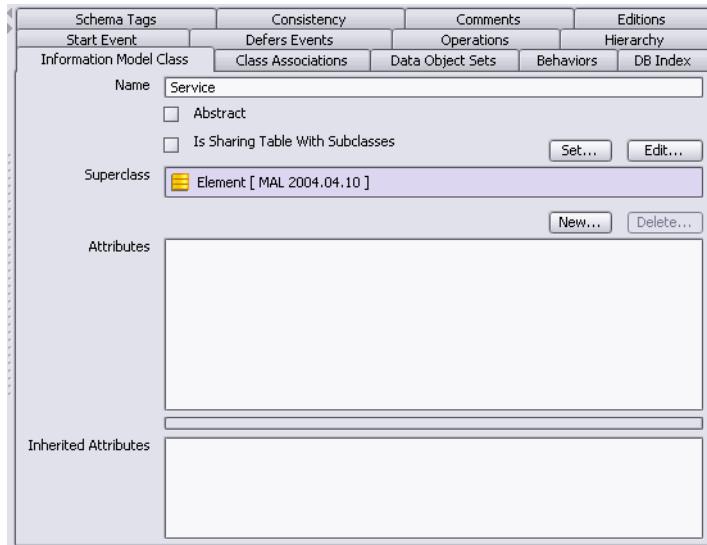
Figure 2-10: POID attribute

Defining a class

After adding a class, you must define the class using the class editor. The following instructions for defining a class apply to both MIM classes and DIM classes. You will not use all tabs in the class editor for every class. Use only those appropriate to the class you are defining.

To define a class:

1. In the class editor, under the **Information Model Class** tab, in the **Name** field, enter a name for the class.



The name must be unique within the context of the entire model.

2. If this class is to be used as an abstract class, check the **Abstract** checkbox. See “[Abstract class](#)” on page 34.
3. If this class shares tables with subclasses, check the **Is Sharing Table with Subclasses** checkbox.
4. If the class has a superclass, above the **Superclass** field, click **Set...**.

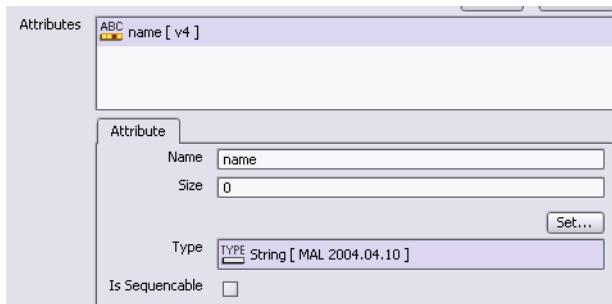
The Select Object(s) window displays a list of classes for use as superclasses. Use the Object Filter field to display a filtered list of classes. For example, Type “T” and the list is filtered to show only those classes beginning with the letter “T”.

- a. Select the class you want to use as the superclass and click **OK**.
- b. To make any changes to the superclass, click **Edit** and make the desired changes.

Note: Any changes you make to the superclass will affect each instance where the superclass is used. Be sure this is the desired effect, before you make edits to a superclass.

5. Above the **Attributes** field click **New...**.
 - a. In the dialog window that displays, for **Name**, enter a name.
 - b. For **Type**, select the type from the drop-down menu and click **OK**.

The attribute is listed in the Attributes field, and the Attribute tab is displayed.



6. If necessary, in the **size** field, set the **size** for String types.

This applies only to String types. Enter a character limit, as a positive integer. During run time, values with a greater number of characters are truncated to the limit you set. The default is 256 characters.

7. If the attribute is sequential, click the **Is Sequenceable** checkbox.

Checking the **Is Sequenceable** checkbox causes the value of this attribute to be generated in the run-time repository during execution. See “[Sequenceable attributes](#)” on page 36.

After you have defined the class and its attributes, you can use the additional tabs in the class editor to further define the class:

- Under the **Comments** tab, you can enter a comment about this class. See “[Class comments](#)” on page 39.
- Under the **Operations** tab, you can define class operations. See “[Operations](#)” on page 41.
- Under the **Data Object Sets** tab, you can define a DOS. See “[Data object set \(DOS\)](#)” on page 46.
- Under the **DB Index** tab, you can create a database index. See “[Database index](#)” on page 40.
- Under the **Behaviors** tab, you can define the class behaviors. See “[Behavior model](#)” on page 109.
- Under the **Schema Tag** tab, you can define a schema tag.
- Under the **Start Event** tab, define a start event for this class if the class includes a behavior. See “[Start event](#)” on page 130.
- Under the **Defers Events** tab, you can define a defered event for this class.
- Under the **Class Associations** tab, you can edit class associations. You will not be able to edit class associations until they have been created. See “[Associations](#)” on page 51.
- Under the **Hierarchy** tab, you can view the class hierarchy tree.
- Under the **Consistency** tab, you can view any consistency messages associated with the class.
- Under the **Editions** tab, you can view editions associated with the class.

Class comments

You can add comments to a class to provide helpful information on the purpose of the class, or for any other reason appropriate to your needs.

To add a comment to a class:

1. Select the **Comments** tab, then select **New**.
 2. Enter a name for the comment, and click **OK**.
- The comment name is listed in the Comment field, and the Comment tab is displayed.
3. Enter the text for the comment in the **Body** field.

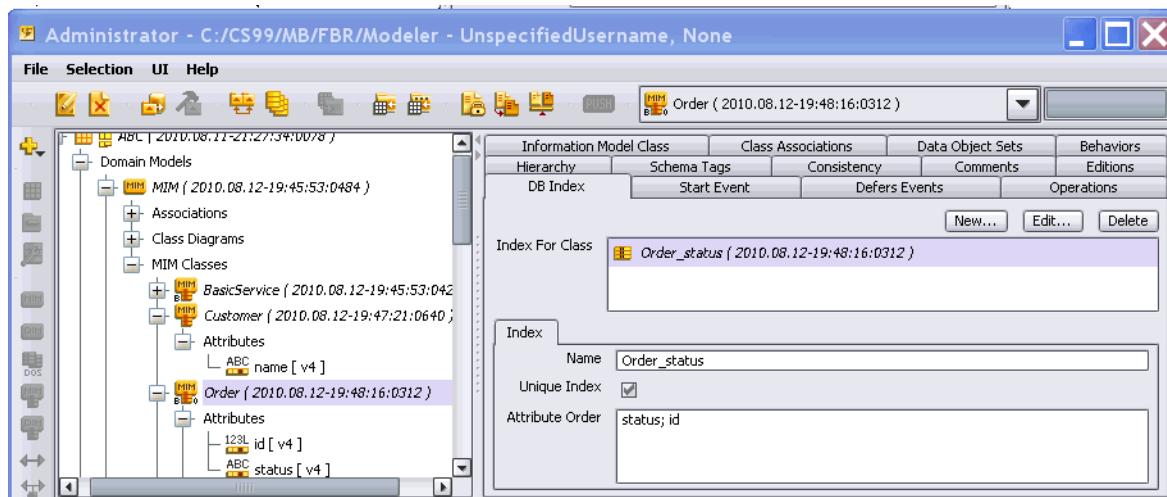
Database index

For your classes you can create database indexes that help speed up searches. You can make the index unique so that the values of the attributes describing the index may not be duplicated (by any two objects) when an object is created.

Although an index does speed up searches for those classes that have them, creating the objects of the class (writing to the database) is slower because of the class database index.

To create a database index:

1. In the class editor, in the **DB Index** tab, click **New**.
2. Enter a name for the index and click **OK**.
3. In the **Attribute Order** field, specify the attributes that are the keys for the index.
Separate multiple attributes with semicolons (;). The list may contain spaces. The attributes must exist for the class.



4. If the index is a unique index, check the **Unique Index** checkbox.

Operations

Operations define an interface to re-usable behavior. An operation has a name, zero or more named and typed input parameters and zero or one typed output parameters.

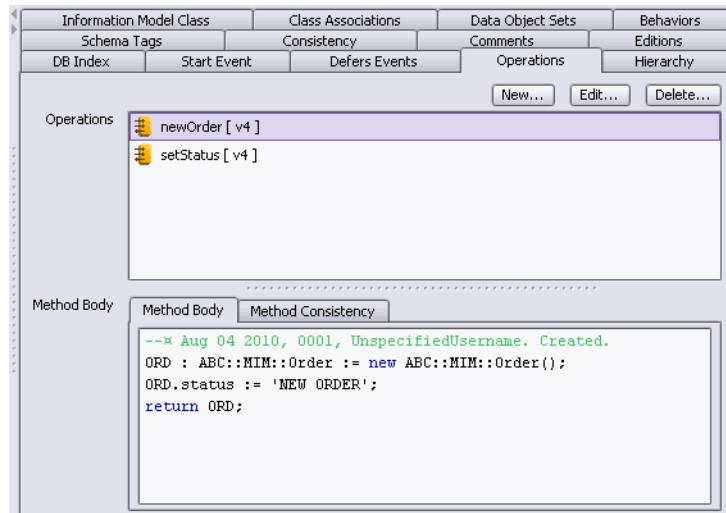


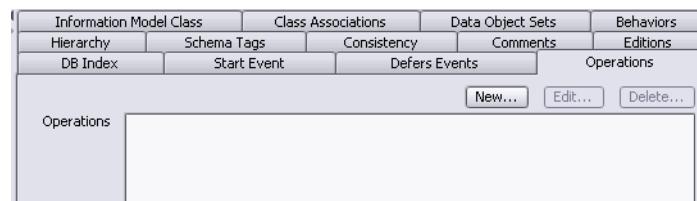
Figure 2-11: Operations defined for a class

Operations are one of several artifact types that may be pushed into execution separately from the model. See “[Single artifact push](#)” on page 30 of the guide, *Model Deployment & Execution in Cogility Studio*.

In addition to classes, operations can be defined for events and messages. The instructions for defining an operation for a class apply to operations for events and messages also.

To define an operation:

1. Open the editor for the class to which you want to add the operation.
2. In the **Operations** tab, click **New**.

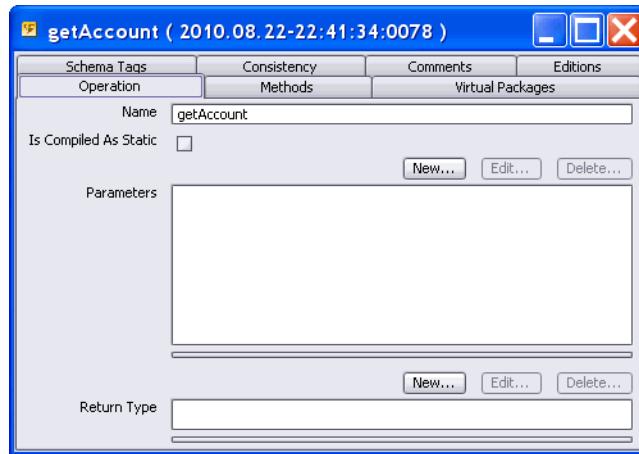


3. In the dialog window, enter the name for the operation and click **OK**.

The name displays in the content view, in the operations field, under the Operations tab. Two tabs also display below the operations field: MethodBody and MethodConsistency.

4. Select the operation and click **Edit**.

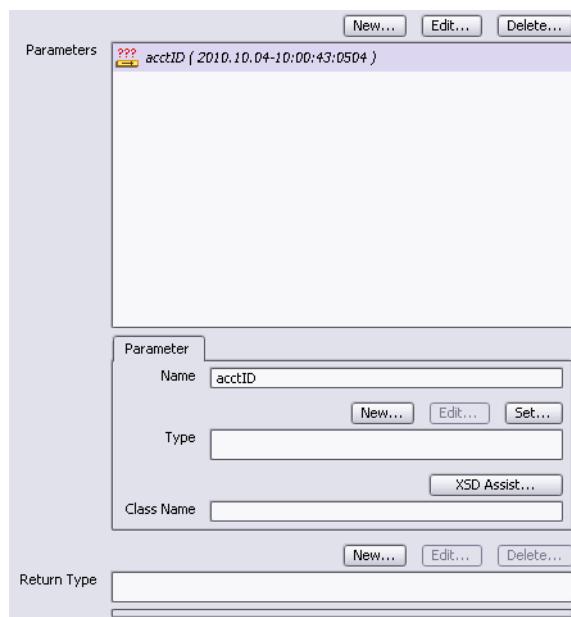
The Parameters window displays.



5. Above the **parameters** field, click **New** to add a parameter.

- In the dialog window, enter a name for the parameter and click **OK**.

The parameters field displays the name and a Parameter tab displays below the parameters field.



- Either click **Set** to set the parameter type from a list of types defined in the model, or skip to **step c** to specify a custom type or XSD type.

The types defined in Cogility Modeler include primitive types and user-defined class types, events and message types.

In the **Select Object(s)** dialog, select the type and click **OK**.

- To specify a custom or XSD type parameter, click **New**.

See “[Custom types](#)” on page 45 and Chapter 8, “[XSD Artifacts](#)”.

In the dialog box, enter a name for the parameter and click **OK**.

In the **Parameter** tab, in the **Class Name** field, enter the Java classpath to the custom type class.

You must have the custom type's class defined in the project class path. See “[Project Classpath Entries](#)” on page 62 of the guide, *Model Deployment & Execution in Cogility Studio*. You may ignore this field if you are using an XSD type.

6. If the operation returns a value, above the **returnType** field, click **New**.

- a. In the dialog window, type a name for the return type and click **OK**.

The name displays in the **returnType** field and a **Parameter** tab displays below the **returnType** field.

Note: Although Cogility Modeler requires that the **returnType** have a name, the name is not used.

- b. Either click **Set** to set the parameter type from a list of types defined in the model, or skip to **step c** to specify a custom type or XSD type.

The types defined in Cogility Modeler include primitive types and user-defined class types, events and message types.

- c. In the **Select Object(s)** dialog, select the type and click **OK**.

- d. To specify a custom or XSD type parameter, in the **Parameter** tab, click **New**.

See “[Custom types](#)” on page 45 and Chapter 8, “[XSD Artifacts](#)”.

- e. In the dialog box, enter a name for the parameter and click **OK**.

- f. In the **Parameter** tab, in the **Class Name** field, enter the Java classpath to the custom type class.

You must have the custom type's class defined in the project class path. See “[Project Classpath Entries](#)” on page 62 of the guide, *Model Deployment & Execution in Cogility Studio*. You may ignore this field if you are using an XSD type.

7. Check the **Is Compiled as Static** checkbox if you want this class to be compiled as a static class.

When checked, the operation's methods will not have "self" as a bound variable. The operation can still be called from a variable whose value is null or non-null, however, when defining the Method body, you can no longer reference the value of "self" (which, in a static method, would always be null).

You can also define "static" methods without checking/enabling "Is Compiled As Static". Just define the Operation and Method and be aware that "self" may be null.

All Operation/Methods have a bound variable named "oclTypeNameForSelf". Even though a "Static" method no longer has a bound variable named "self", "oclTypeNameForSelf" still contains the type name for the static/null "variable" that made the static call.

Static classes are commonly used to create classes that do not need to be instantiated.

8. Close the operation editor window and return to the class, message or event editor window.

9. In the **MethodBody** field, enter the actions that describes the logic of the operation.

Each operation consists of an action method that describes the logic of the operation. The artifact where the operation is defined must implement a method body. Subclasses of the artifact can implement their own method bodies to override the operation of the superclass. Methods in subclasses can call the methods in the hierarchy using the *self* and *super* keywords. Operations have one bound variable, **oclTypeNameForSelf**, which is accessible in the Method Body. For more information about action semantics and operations, see “[Method body](#)” on page 13 of the guide, *Using Actions in Cogility Studio*.

10. Click the **MethodConsistency** tab to compile the actions and check for errors.

Special Operation to Initialize a Class Instance

Two specially named operations allow you to initialize a new instance of a MIMClasses, DIMClasses, or Events, with default values. If you define an operation either named **initialize** or **postInitialize**, it will execute automatically when an instance is created using an action semantic “new” statement, for example:

```
new Model::MIM::Customer(firstName: 'John', lastName: 'Doe');
```

The **initialize** operation will execute before the assignment of values from the named parameter list in the “new” statement. The **postInitialize** operation will execute after the assignment of the values from the named parameter list in the “new” statement.

Generally, you should use **postInitialize** rather than **initialize**, because the attribute value assignments have not occurred when the **initialize** operation is called and therefore you cannot take advantage of them. Putting logic in 'postInitialize' would be functionally equivalent to **initialize**.

A third special operation, named **postAssociate**, may be defined for MIMClasses and DIMClasses and will execute automatically when object(s) are associated using the action semantic “new to” statement, for example:

```
new aCustomer to order(anOrder);
```

The association instance created can be either an Association or an AssociationClass. The **postAssociate** operation can be defined on the source class and/or the target class, and it will execute after the object(s) have been associated. If a **postAssociate** operation is defined on both object(s), then both operations will execute regardless of the order in which the association is created.

The postAssociate opeation has four or five additional bound variables.

- sourceObjec
- sourceRoleName
- targetObject
- targetRoleName
- linkObject (AssociationClass only)

You cannot define the special operations with either parameters or a return type.

The **postInitialize** and **postAssociate** functions allow you to perform a consistent set of actions in response to either the creation of a business object or the act of relating two business objects. There are multiple places in the model where action semantics can be used to create and/or relate business objects. These can include Inbound Web Services, State Machines, Messages and Operations on Business Objects. If there is a common set of actions that need to be performed each time an object is created or related to another object, it can be difficult to duplicate this logic in all the above named places in the model. Defining these actions in **postInitialize** and **postAssociate** solves this problem.

To make these special operations more obvious in Modeler, their icons are displayed in blue as opposed to the gold icons.

Custom types

You can use any type created in Java as either an argument to or a return from an operation. You must have the custom type's class defined in the project class path. See “[Project Classpath Entries](#)” on page [62](#) of the guide, *Model Deployment & Execution in Cogility Studio*.

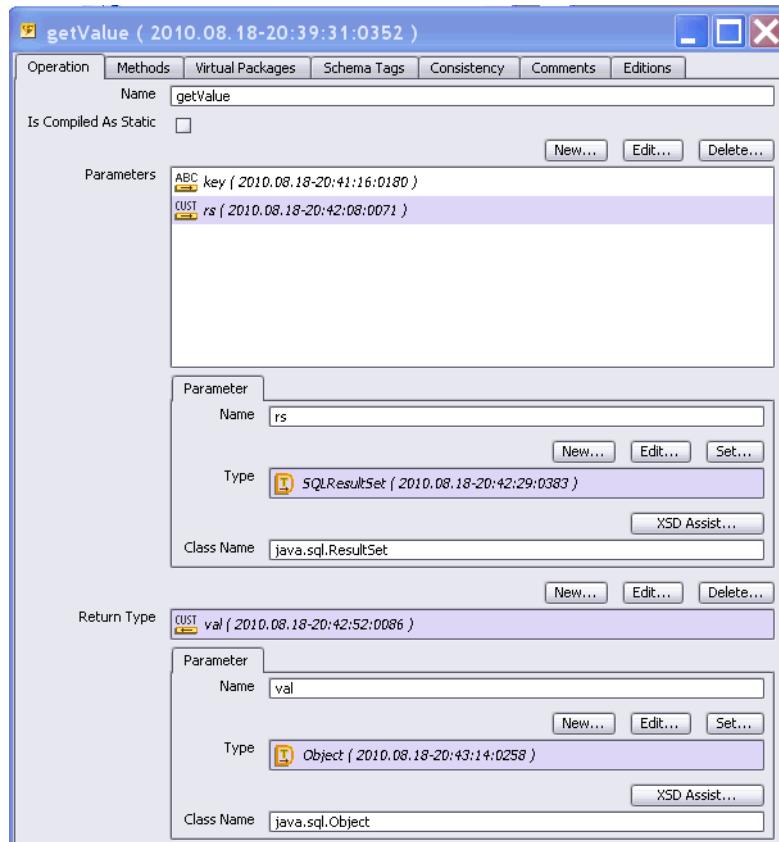
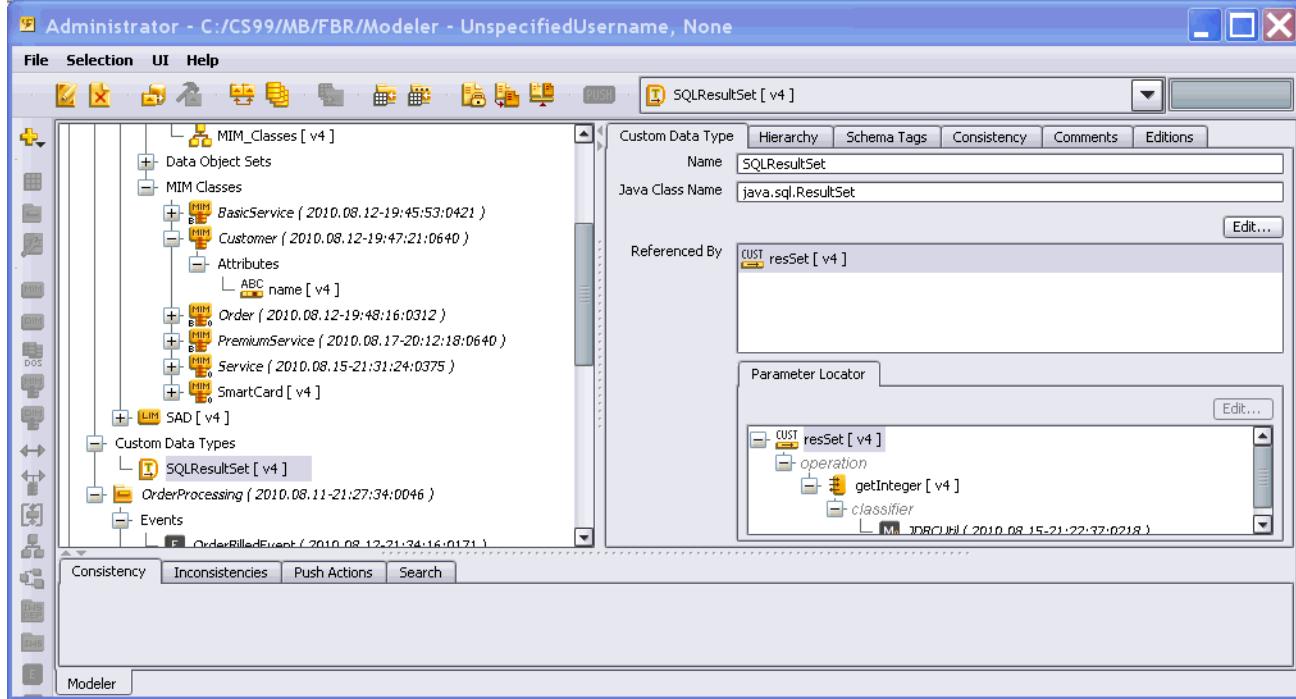


Figure 2-12: An operation that specifies a custom type for a parameter

Determining where a custom type is used

The tree view shows all custom data types for the model. You can determine where a custom data type is used by selecting and viewing the data type.

The Parameter Locator tab shows where the custom data type is used.



Data object set (DOS)

To use the DIM and MIM classes in a data transformation, the classes participating in the transformation must be included in a data object set (DOS). For more information on data transformations see “[Transformation model](#)” on page 145.

Only classes from a single DIM or from the MIM can be included in a single DOS. However, for each DIM and the MIM you can create multiple DOS. A DOS generally contains only those classes that participate in the transformation. For example, a Billing DIM includes the classes: Customer, Address and Account. Only the Customer and Address classes include data that needs to be transformed using a transformation, so only those two classes need to be included in a DOS. Another

transformation, may need only the Customer and Account classes. Another DOS would contain only those classes.

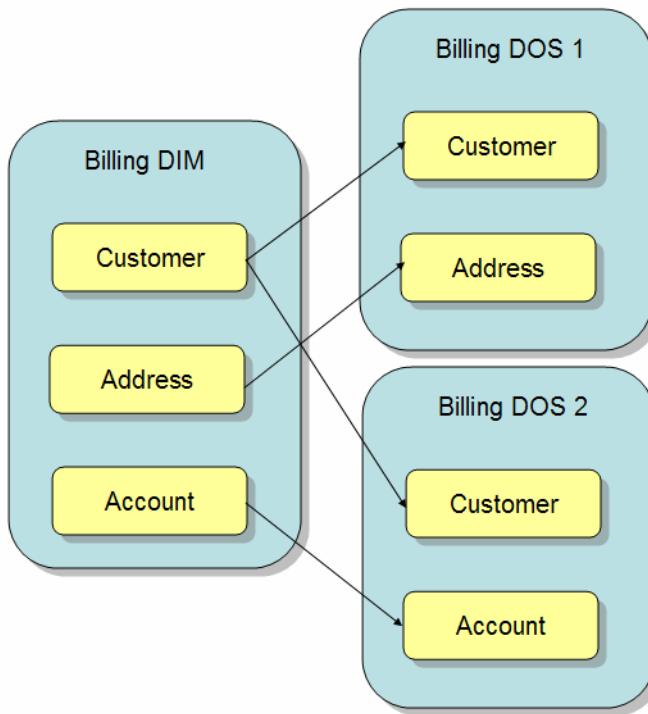


Figure 2-13: A DOS represents one or more classes

For each transformation, there is an input DOS and an output DOS. The input DOS contains the classes with the transformation input data and the output DOS contains the classes that receive the transformation output data. The only thing that distinguishes an input DOS from an output DOS is its function in a transformation. A DOS can be the input for one transformation and the output for a different transformation. But, it cannot be both the input and output for the same transformation.

You must create a DOS before it can be used in a transformation.

To create a DOS:

1. Create a DOS artifact using one of the following methods:
 - a. In Cogility Modeler's tree view, select the DIM or the MIM for which you wish to create a DOS.
 - b. Click the **Add a Data Object Set (DOS)** button  or from the **Selection** menu, select **Domain Modeling > Add a Data Object Set (DOS)**
Or
 - a. In Cogility Modeler's tree view, select the class for which you wish to create a DOS.
 - b. In the class editor, in the **Data Object Sets** tab, above the **DOS** field, click **New**.
2. In the dialog, enter a name for the DOS and click **OK**.
The new DOS name appears in the DOS list.
3. Select the new DOS name in the DOS list and click **Edit**.
4. In the DOS editor, above the **Classes** field, click **Add**.
5. Select the class or classes you wish to include and click **OK**.

You may hold down the Ctrl key while selecting to select multiple classes.

Factories

A factory creates an instance of a class. In many cases object creation may happen only when an event starts a business process that creates the new object. The trouble is, that event requires an object with the behavior (business process) that creates the new object. Clearly, the object to be created and the object with the creation behavior cannot be one and the same. To get around this chicken-and-egg problem, the event specifies a factory with the object creation behavior.

For example, the model in the figure below has a single MIM class, Customer. The model receives new customer information and generates an event to initialize the customer with a first name and a last name. The event needs an object that can execute this behavior. In this case, Customer cannot be that object, since an instance of Customer does not exist until the Customer is created. So the event specifies the customer creation behavior of a factory called CreateCustomer.

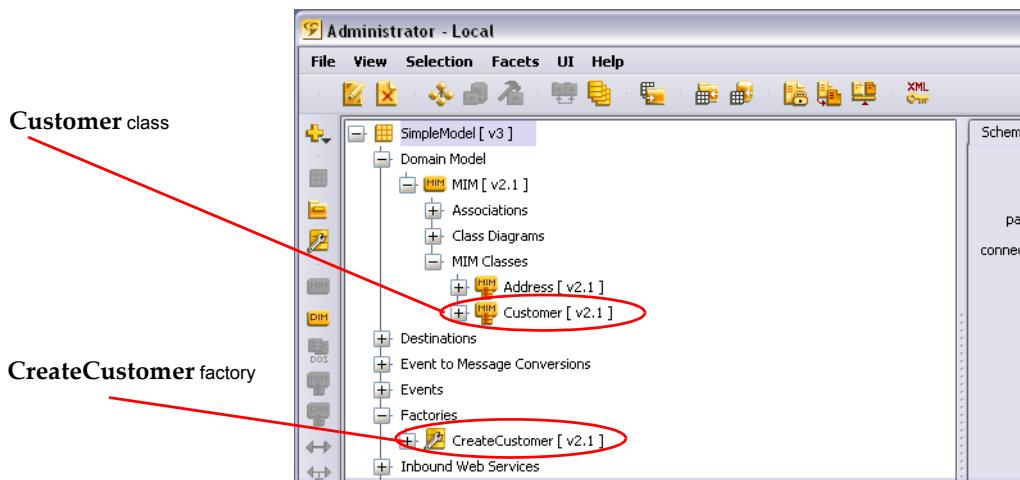


Figure 2-14: A factory creates an instance of a class

An instance of CreateCustomer is automatically created when the model is pushed into execution as a J2EE application. This instance has a behavior (state machine) for creating instances of the MIM class Customer.

The factory is a top-level, enterprise-wide artifact not related to either a MIM or a DIM. Only one instance of a factory exists. The one instance has behavior, and may have operations, but cannot have attributes or participate in associations.

Factories are represented in Cogility Modeler's tree view with a factory icon: . If the factory has an associated behavior, the icon includes a B in the top left corner: . If the factory has an operation, the icon has an O in the top right corner: . Factories with both a behavior and an operation display with both a B and an O: .

To create a factory:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the Add a Factory button , or from the Selection menu, select Domain Modeling > Add Factory.

An untitled factory displays in the tree view, and the class editor displays in the content In the factory editor, under the **Factory** tab, in the **Name** field, enter the name for the factory.

The name you enter must be unique within the model container.

3. In the **Operations** tab, define operations.

See “Operations” on page 41.

4. In the **Behaviors** tab, define the class behaviors.

See “Behavior model” on page 109.

5. In the **Start Event** tab, define a start event for the factory.

See “Start event” on page 130.

Class diagrams

A class diagram describes classes and their relationships according to the UML standard. You can use the class diagram to describe class hierarchies, define classes and associations between classes, including their source and target roles, and the multiplicity values of each role. You do not have to include all your model's classes on a diagram, and you can include a particular class on more than one diagram.

A class diagram may describe a hierarchy, as in the figure below.

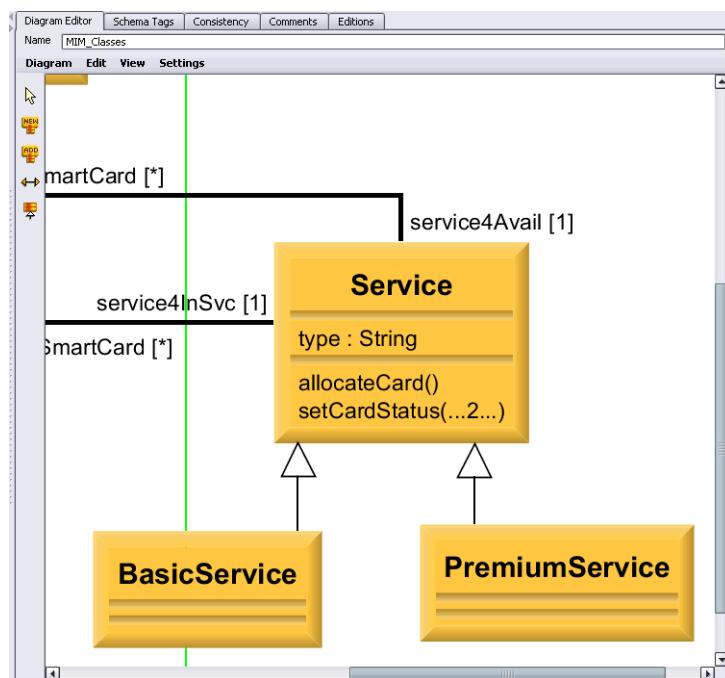


Figure 2-15: A class diagram describing a hierarchy

A class diagram can also describe associations between classes, as shown in the figure below.

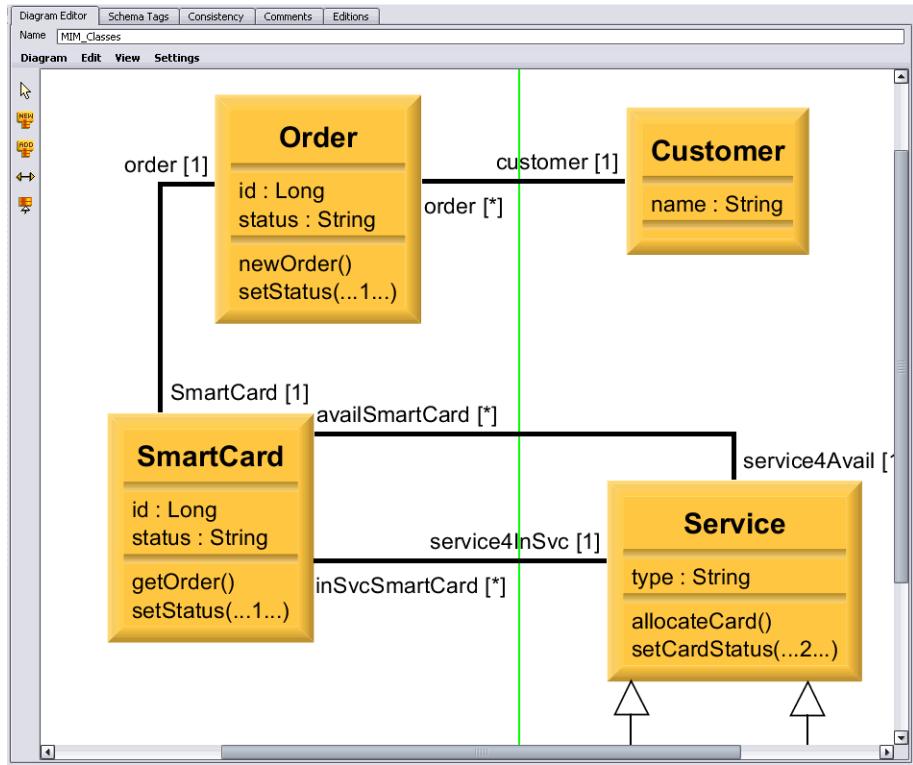


Figure 2-16: A class diagram describing a class association

To create a class diagram:

1. In Cogility Modeler's tree view, select the MIM or DIM within which you want to create the class diagram.

If your model does not yet have a MIM or a DIM, see “Master information model (MIM)” on page 29 or “Distinct information model (DIM)” on page 30 to add these artifacts.

2. Click the Add Class Diagram button

You can also choose, from the Selection menu, Domain Modeling > Add Class Diagram. The class diagram editor then appears in the content view.

3. In the diagram editor, in the Name field, enter the name for the new class diagram.

The name you enter must be unique within the MIM or DIM.

4. Create new classes or add existing classes to the diagram.

To create a new class:

- a. In the class diagram tool bar, click the Create New Class button

- b. Move the cursor into the diagram where you want to locate the new class and click.

An untitled class icon appears in the diagram editor window.

- c. Double-click the class icon to open the class editor. See “Defining a class” on page 37.

To add an existing class:

- a. In the class diagram tool bar, click the Add Existing Class button

- b. Move the cursor into the diagram where you want to locate the class and click again.

- c. From the Select Object(s) dialog, select the artifact(s) and click OK.

5. Create associations with other classes.

- a. In the class diagram tool bar, click the Create New Association button

- b.** In the class diagram, click the class with the source role of the association.
- c.** Drag the cursor to and click the class with the target role of the association.
- d.** In the **Set Association Class** dialog, select **Association** and click **OK**.

An association is displayed, as a line between the two classes, in the class diagram with the default names for the source role and the target role. These names are based on the name of their respective classes. Also, for each role, the default multiplicity * (many) is set by default. See “[Association settings](#)” on page 53 for information about changing these settings.

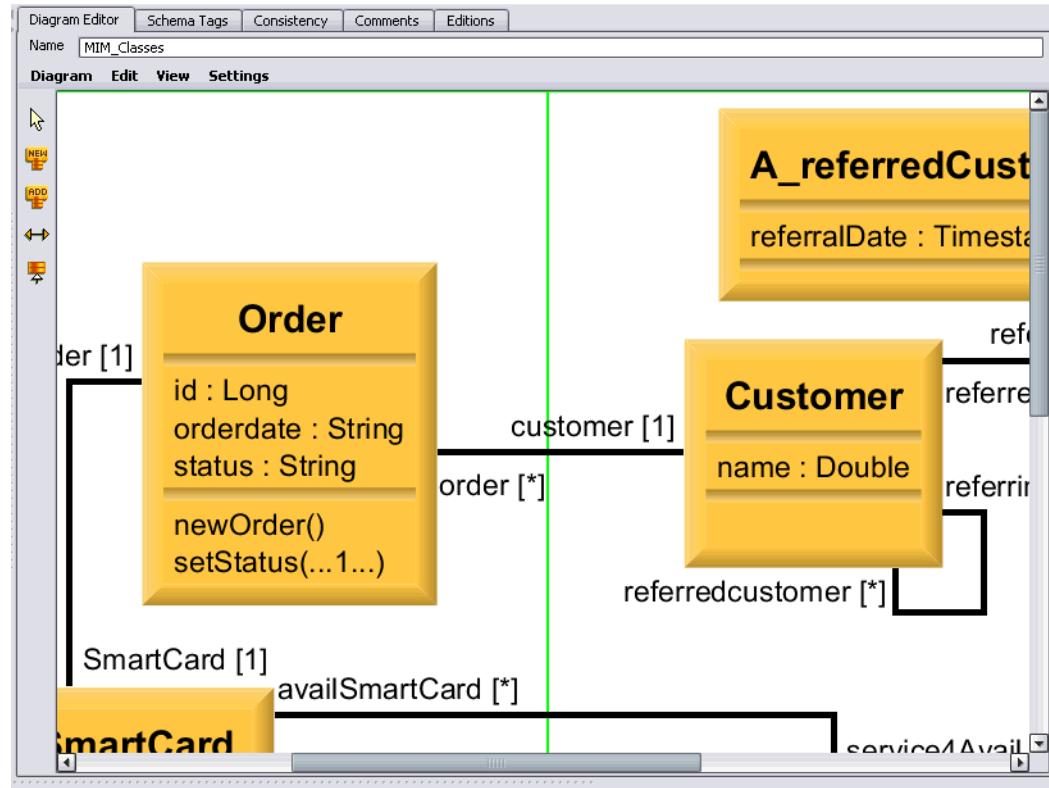
- e.** Define the association. See “[Association settings](#)” on page 53.
- 6.** Define superclass to class inheritance relationships.
 - a.** In the class diagram editor tool bar, click the **Define Superclass** button .
 - b.** Click the child class icon, then move the cursor to the parent class icon and click again. The child class inherits from the parent class or superclass. A class inheritance arrow is drawn from the child class to the parent class.

Associations

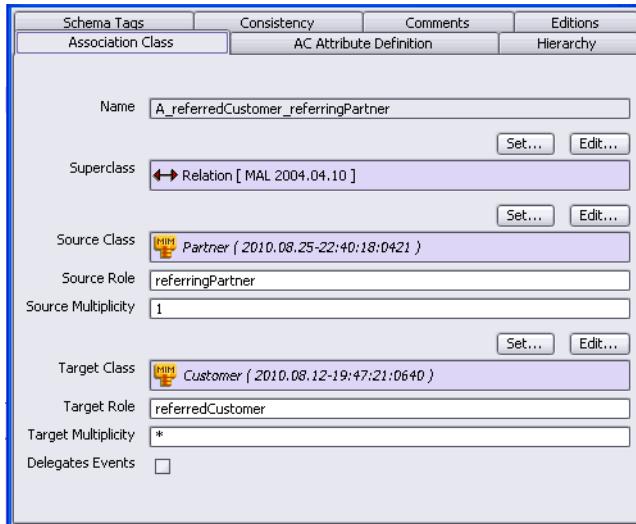
An association describes a relationship between two objects. Classes in Cogility Modeler describe their relationships to other classes with associations or association classes. An association class lets you define an association with attributes. See “[Association classes](#)” on page 54.

An association has two roles, a source role describes the role of the owning object in the relationship, and the target role describes the associated object’s role in the relationship. In the association illustrated below, the Customer has the role of orderOwner and the Order has the role of

ownedOrder. The multiplicity for each role describes how many objects may participate in the relationship. As shown below, one (1) Customer can have many (*) Orders.



The definition for the association is shown below.



A class may be associated with itself. Such a relationship, a reflexive association, describes an object's relationship with an object of the same class. See ["Reflexive associations" on page 57](#).

The multiplicity expression describes how many instances of a class may participate in the association. In Cogility Modeler, the following formats are valid for multiplicity expressions:

- A positive integer such as 1 or greater

- An asterisk * meaning no maximum

The expressions may be combined to describe a range, as in <expression>..<expression> (for example, 1..5, meaning 1 to 5 or 1..*, meaning 1 to no maximum).

To create an association:

1. In Cogility Modeler's tree view, select the object (the MIM or the DIM) which contains the classes you want to associate.
2. Click the **Add an Association** button  or from the **Selection** menu, select **Domain Modeling > Add Association**.
An association is displayed in the tree view and an editor is displayed in the content view.
3. Change the association settings as necessary for the association.
See "Association settings" on page 53.

Association settings

Associations are created with default names for the source and target roles and a default multiplicity of * (many) for each role. Follow these steps to change these settings.

To change association settings:

1. In the class diagram, double-click the association line, or in Cogility Modeler's tree view, select the association artifact.
The association editor displays.
2. If necessary, under the **Association Editor** tab, select the source and target classes.
When you use the buttons and menus to add the association, the Source Class and Target Class fields are blank and you must select a class for each. When you add the association using a class diagram the classes are already selected. You can use these steps to change the selected source and target class.
 - a. Above the **Source Class** field, click the **Set** button, select the target class from the list of classes and click **OK**.
 - b. Above the **Target Class** field, click the **Set** button, select the target class from the list of classes and click **OK**.
 If you are creating a reflexive association, select the same class that you selected for the Source Class. See "Reflexive associations" on page 57.
3. In the **Source Role** field, enter the a name for the source role.

Note: The name of an association or association class is derived from a concatenation of the source role name and the target role name. Therefore when you name a source role and a target role, the resulting combination must be unique within the context of the entire model.

4. In the **Source Multiplicity** field, enter the multiplicity expression, either 1 or *.
5. In the **Target Role** field, enter the name for the target role.
6. In the **Target Multiplicity** field, enter the multiplicity expression, either 1 or *.
7. Click the **Delegates events** checkbox if needed.

Association classes

Sometimes you need to describe an association with attributes; in such cases, the association may be described with an association class. Ordinary associations (described in [“Associations” on page 51](#)) do not provide for attribute definitions.

Association classes, like ordinary classes, messages and events, may inherit from a superclass. By default, association classes inherit from the Relation metaclass. Unlike ordinary classes, association classes may not have operations.

The association class icon appears in the diagram, connected to the association by a dashed red line.

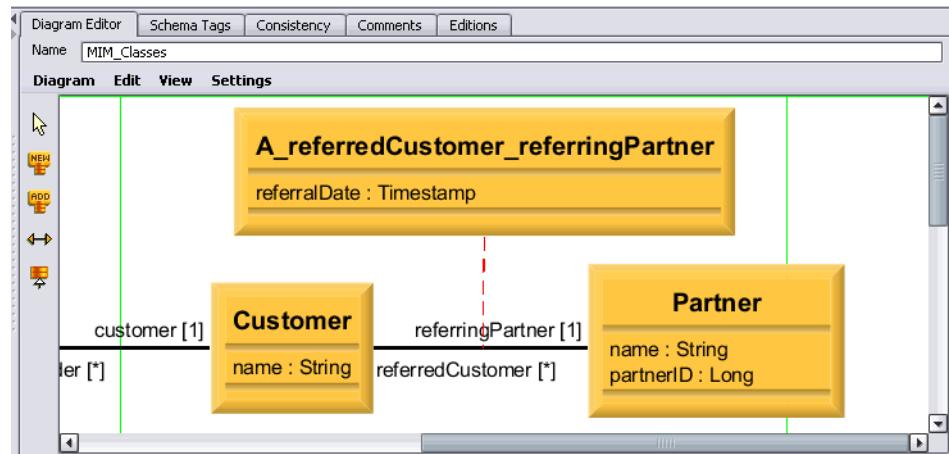


Figure 2-17: An association class in a class diagram

The definition for the association class is shown in [Figure 2-18](#).

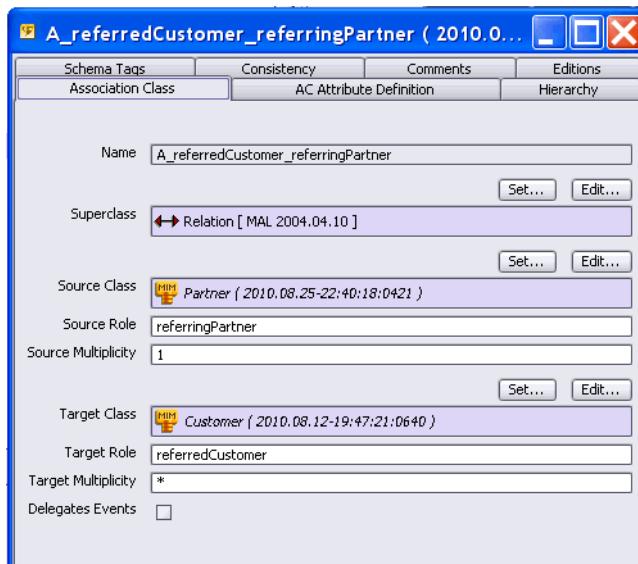


Figure 2-18: Association class definition

What makes the association class different from an ordinary association is that it may have attributes; in this example, the association class has a `referralDate`. This attribute does not properly belong to the `Customer` class because it is meaningless if the customer was not referred, and not all customers are referred. An association class lets you create a ternary (three-way) relationship between classes without creating an additional business object.

You can replace existing associations with association classes. Usually, you cannot replace an artifact with another artifact of the same name in Cogility Modeler. Doing so usually makes the model non-additive and it will not push into execution. Replacing association with association classes is allowed, however. See “[Non-additive schema updates](#)” on page 32 of the guide, *Model Deployment & Execution in Cogility Studio*.

In addition to using a class diagram (see “[To add an association class in a class diagram:](#)” on page 55), you can also define association classes using the action buttons or the menus in the Cogility Modeler window.

To create an association class:

1. In Cogility Modeler’s tree view, select the object (the MIM or the DIM) which contains the classes you want to associate with an association class.
2. Click the **Add an Association Class** button  or from the **Selection** menu, select **Domain Modeling > Add Association Class** from the menu.
An association class is displayed in the tree view and an editor is displayed in the content view.
3. Edit the association class.
See “[Editing an association class](#)” on page 55.

You can create an association with an association class once you have classes in your diagram, as described in “[Class diagrams](#)” on page 49.

To add an association class in a class diagram:

1. In the class diagram tool bar, click the Create New Association button .
2. In the class diagram, click the class with the source role of the association.
3. Drag the cursor to and click the class with the target role of the association.
4. In the **Set Association Class** dialog, select **AssociationClass** and click **OK**.
An association is displayed, as a line between the two classes, in the class diagram with the default names for the source role and the target role. These names are based on the name of their respective classes. Also, for each role, the default multiplicity * (many) is set by default. See “[Editing an association class](#)” on page 55 describes how to change these settings.
The association class for the association does not display automatically. You must add it to the diagram.
5. In the class diagram tool bar, click the **Add Existing Class** button .
6. In the **Select Object(s)** dialog, select the association class and click **OK**.
The name of an association class is derived from a concatenation of the source role and the target role. If you have not changed these roles, the name is based on the default names for the roles. The default role names are based on their respective class names.
7. Edit the association class.
See “[Editing an association class](#)” on page 55, next.

Editing an association class

With association classes, you can define attributes that participate in the association. Association classes are created with default names for the source and target roles and a default multiplicity of * (many) for each role. See “[To add an association class in a class diagram:](#)” on page 55. Follow these steps to change the default settings.

To edit an association class:

1. In the class diagram, double-click the association class icon, or in Cogility Modeler's tree view, select the association artifact.

2. If necessary, under the **Association Class Editor** tab, select a source and target class.

When you use the buttons and menus to add the association class, the source class and target class fields are blank and you must select a class for each end of the association. When you add the association class using a class diagram, the classes are already selected. However, you can use these steps to change the selected source and target class.

- a. Above the **Source Class** field, click the **Set** button.

A selection window displays containing the classes.

- b. Select the class you want as the source class and click **OK**.

- c. Above the **Target Class** field, click the **Set** button, select the target class and click **OK**.

If you are creating a reflexive association, select the same class that you selected for the Source Class. See “[Reflexive associations](#)” on page 57.

3. In the **Source Role** field, enter the role name for the source class (for example, referringPartner).

Note: The name of an association or association class is derived from a concatenation of the source role and the target role. Therefore when you choose a source role or target role, the combination must be unique within the domain (MIM or DIM).

4. In the **Source Multiplicity** field, enter the multiplicity expression, either 1 or *.

5. In the **Target Role** field, enter the role for the source class (for example, referredCustomer).

6. In the **Target Multiplicity** field, enter the multiplicity expression, either 1 or *.

7. If necessary, select a superclass for the association class.

- a. Above the **Superclass** field, click the **Set** button.

A list of classes displays. The default superclass is the Relation class. See “[Superclass](#)” on page 33.

- b. Select the parent class from the list and click **OK**.

8. Click the **AC Attribute Definition** tab, and above the **feature** field, click the **New** button.

- a. In the dialog window that appears, for **Name**, enter the name for the attribute.

- b. For **Type**, select the type and click **OK**.

The name you enter must be unique within the association class. The attribute is displayed in the feature field. Repeat this step for additional attributes.

9. If necessary, in the **maximumCharacters** field, enter the maximum number of characters.

This applies only to String types. Enter a character limit, as a positive integer, in the field. During run time, values with a greater number of characters are truncated to the limit you set. The default is 256 characters. Entering a zero specifies that no limit is imposed on the number of characters.

10. If necessary, check the **isSequenceable** box.

Checking the **isSequenceable** box causes the run time repository to automatically assign values to sequenceable attributes during execution. See “[Sequenceable attributes](#)” on page 36.



Reflexive associations

A reflexive association relates an instance of a class with another instance of the same class with either an association or an association class. You can use a class diagram or the buttons and menus in the Cogility Modeler to create a reflexive association or association class.

An example of a reflexive association is described in the following diagram. A Customer object may refer any number of Customers. Rather than illustrate the association with two Customer classes, the association is drawn as connecting the Customer class with itself.

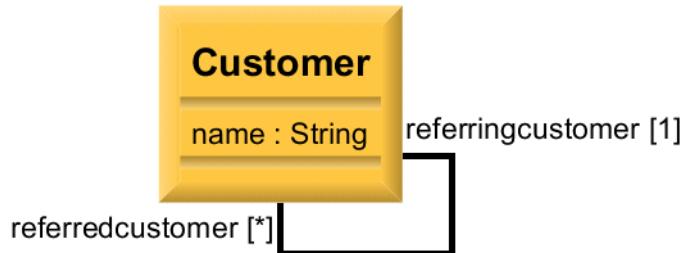


Figure 2-19: Reflexive association

Creating a reflexive association

To create a reflexive association or association class:

1. In the class diagram tool bar, click the Create New Association button .
2. In the class diagram, click the class with the reflexive association.
3. Click the same class.
4. In the **Set Association Class** dialog, select the type of association and click **OK**.
5. Define the association or association class information.

See “[Association settings](#)” on page 53 and “[Editing an association class](#)” on page 55.

Viewing association chains

In a large model, it may be difficult to see all the associations between classes. To make it easier to see all the associations between two classes, you can view the association using the Association Chain viewer (see [Figure 2-20](#).)

An Association Chain is a path from the source class to the target class through its roles, associations and intermediate classes. A chain can contain a single segment directly from the source class to the target class, or it can contain numerous segments through several intermediate classes.

In the Association Chain viewer, the chains are displayed in a tree with one branch per chain. Each chain starts with the source class and ends with the target class. Within a chain, each segment has five

parts (a Class, the Role for that Class, the Association between the two Classes, the Role for the other Class, and the other Class).

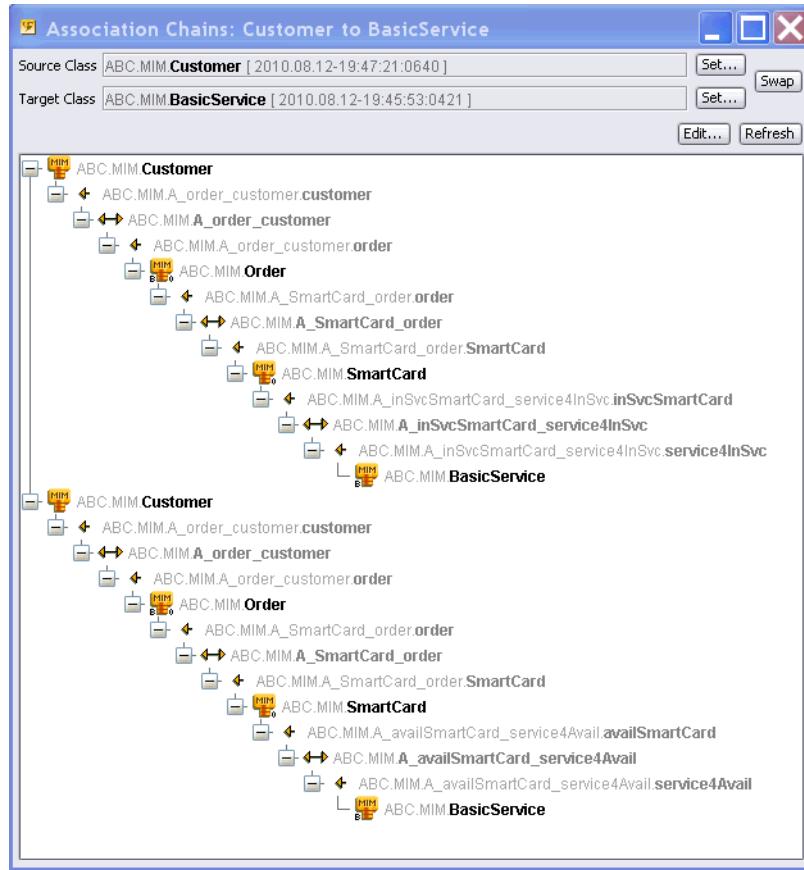


Figure 2-20: Association Chain Viewer

The tree field shows the Association Chains. Each branch of the tree represents a single chain. Each row within a branch represents an element in the chain (a DOMClass, a Role, or an Association). Each row can show the full XML key prefix and the version name for that element. To show the full name or version, right click on an element and select Show Full Name or Show Version Name from the menu. If they are shown, the full xml key prefix and version name are lighter in color to make the element's name easier to see, as shown in Figure 2-21.



Figure 2-21: Association chain showing full XML key prefix and version

The buttons allow you to do the following:

- The Set... buttons allow you to assign a different class to source or the target field. When you assign a different class to a field, the tree updates with the associations for the newly selected source or target.
- The Swap button swaps the source class and target class. The tree field also reflects the swap.



- The Refresh button regenerates the tree. The Association Chains viewer does not automatically update each time the model changes. Click Refresh to update the contents of the window to reflect any updates to the Model.
- The Edit... button opens a the appropriate Object Editor for the selected item in the tree.

To view an association chain:

1. Select one or two DOMClasses (DIMClasses or MIMClasses) from the Cogility Modeler tree view.

These classes are the Source Class and Target Class for the Association Chains. If only one DOMClass is selected, it is used as both the Source and Target Class. One of the classes you select appears in the Source field, the other appears in the Target field of the Association Chain viewer.

Note: If the source and target classes appear in an inverted order from what you wanted, you can use the Swap button to switch the source and target class.

2. From the Selection menu, select **Actions > Show Association Chains**.



Legacy Information Model

For communication with integrated applications, and the databases those applications access, most application will have an API that allows you to use either web services or Java Messaging Service (JMS) to access the database data. For those applications that do not have an API, you must access the database directly. Using a Legacy Information Model (LIM) is a way to access an external database directly. While it is generally preferable to use an API, the LIM provides an alternate method when an API is not available.

There are two modes in which you can access data using a LIM:

- Passive Mode - In passive mode, you request data from the database. There are two ways to request this data:
 - Using LIMClass objects. See “[LIMClasses](#)” on page 66.
 - Using SQL statements. See “[SQL Statements](#)” on page 67.
- Active mode - In active mode, activated tables are monitored for changes to the data. For each active table or column of a legacy database, you define actions in your LIM that direct the model to do something when the legacy data changes.
 - Using Activated Tables. See “[Using a LIM in active mode](#)” on page 75.

Creating a LIM container

A LIM container is the Modeler artifact which identifies a legacy database. A LIM container is defined by the connection information to the external database. This information includes the user ID, password, connect string, and schema name. If you need to include more than one legacy database in your model, create a LIM container for each database.

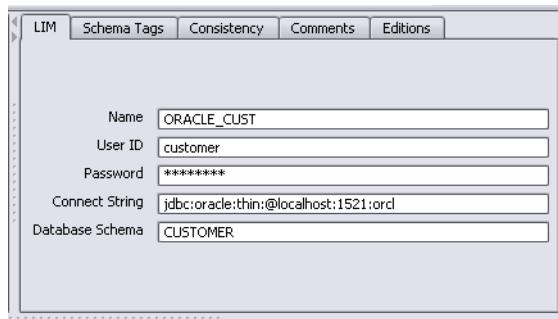


Figure 3-1: LIM definition

To create a LIM:

1. In Cogility Modeler’s tree view, select the model container or package.
2. Click the plus button and select **Add Legacy Information Model (LIM)**.

In the content view, the editor window for the LIM displays.

3. In the LIM editor, in the **LIM** tab, in the **Name** field, enter a name.
4. In the **UserID** field, enter a valid user ID for the external database.
5. In the **Password** field, enter the password corresponding to the user ID.
6. In the **Connect String** field, enter the connect string for the external database.
7. In the **Database Schema** field, enter the name of the schema in the database that you wish to use. This information is required if you will be using **Discover Tables**.

The connect string syntax varies according to the database. See your database documentation for more information.

After you define a LIM, you can verify the connection information by testing the database connection.

To test a LIM database connection:

1. In Cogility Modeler's tree view, select the LIM you wish to test.
2. Do one of the following:
 - From the **Selections** menu select **Actions > Check Database Connection**.
 - Right-click on the LIM and select **Actions > Check Database Connection** from the right-click menu.

You will receive either a success message or a failure message. If you receive a failure message, verify the connection parameters and retry.

If you are only using SQL statements to access the LIM data, you can define the SQL statements after creating the LIM container. For instructions on using SQL to access data, see “[SQL Statements](#)” on page 67.

If you are using LIMClasses or ActivatedTables, you must first use Discover Tables. See “[Discover Tables](#)” on page 62 for instructions.

Discover Tables

To use either Activated Tables or LIMClasses, you must first use Discover Tables. By using Discover Tables, the legacy database tables are made available for inclusion in the LIM as either ActivatedTables or LIMClasses.

The Discover Tables commands read a database schema for which you have created a LIM container and shows you the tables and columns contained in the schema. There are two Discover Tables dialogs:

- Discover Tables (Activated Tables)...
- Discover Tables (LIMClasses)...

The tables available in the legacy database appear in the Discover Tables dialogs as shown in Figure 3-2.

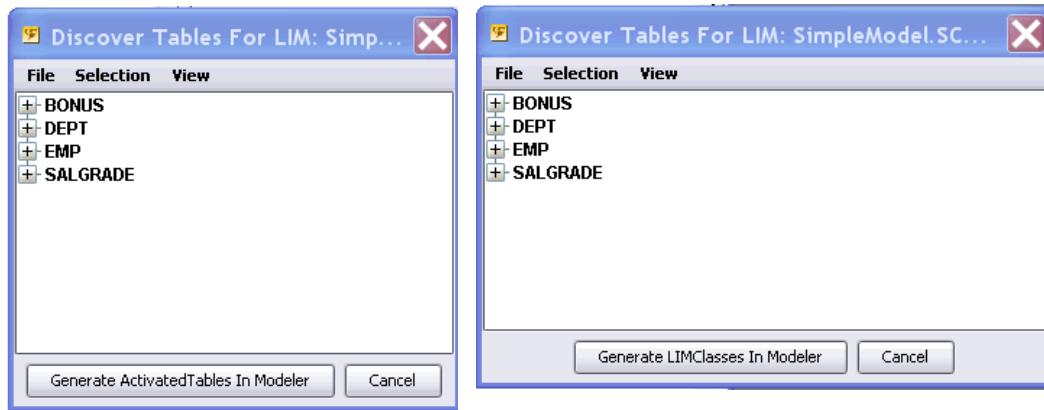


Figure 3-2: Discover Tables (ActivatedTables) and Discover Tables (LIMClasses) dialogs

To view the columns in a table, click the expand icon next to the table name. The table expands as shown in the figure below.

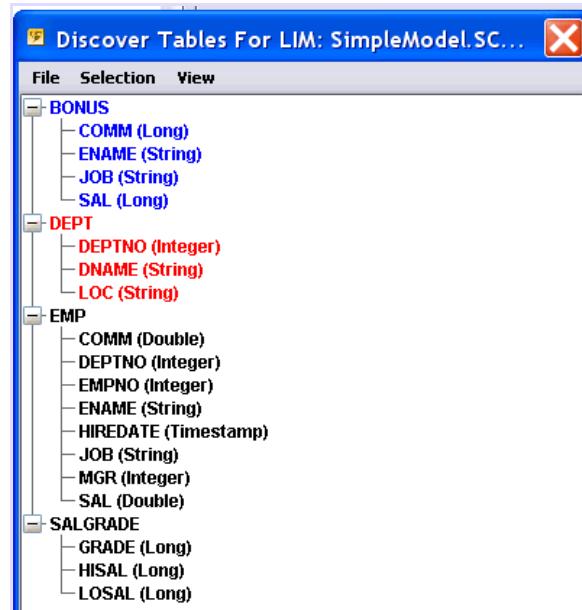


Figure 3-3: Expanded tables in the Discover Tables dialogs

Table and column entries appear in the following colors, depending on their state:

- Black represents Tables/Columns that were discovered in the LIM but have not been set as ActivatedTables or Added as LIMClasses in the Model.
- Blue represents Tables/Columns that have already been set as ActivatedTables or Added as LIMClasses in the Model.
- Red represents Tables/Columns that no longer exist in the database and that are set to deactivate ActivatedTables or Delete LIMClasses in the Model.
- Grey represents Tables/Columns that have already been set as ActivatedTables if you are in the LIMClasses dialog, or have been Added as LIMClasses if you are in the ActivateTables dialog.

A table and its columns can only exist in the Model as either an ActivatedTable or a LIMClass, not both.

The following icons are used during selection:

- icon will appear before Tables/Columns that will be activated or added to the LIM during generation.
- icon will appear before Tables/Columns that will be deactivated or deleted from the LIM during generation.
- icon appears before Tables/Columns that have entries in the LIM but no longer have entries in the legacy database. These entities will be removed from the LIM during generation.
- icon appears before Columns that exist in both the legacy database and Modeler database, however, the datatype is no longer the same. These entities will be removed from the LIM during generation. When a "!" icon appears, the tree will have two column entries with the same name. The "!" entry will have the Modeler datatype, the other will have the legacy datatype.

The options available in the "Selection" menu depend on whether you have selected ActivatedTables or Add LIMClasses as shown below:

ActivatedTables	Add LIMClasses	Description
Activate Selection(s)	Add Selection(s)	The selected tables will be flagged for activation or addition during generation.
Deactivate Selections(s)	Delete Selections(s)	The selected tables or columns will be flagged for deactivation or deletion during generation.
Revert To Original State(s)	Revert To Original State(s)	The selected entities will be unflagged and return to the state that they displayed immediately after the dialog appeared. This effectively removes any changes you have made during the current selection process.
Explain Selection...	Explain Selection...	For a single selection, a dialog will appear containing an explanation of the entity's state at the time of discovery and an explanation of what will occur during generation.

The options available in the View menu depend on whether you have selected ActivateTables or Add LIMClasses as shown below:

ActivatedTables	Add LIMClasses	Description
Show All	Show All	All Tables/Columns will appear in the tree (the default when the dialog opens).
Show Activated Only	Show Added Only	Only activated tables will appear in the tree. This includes Tables/Columns that were already activated in the Modeler database, plus Tables/Columns that have been flagged to be activated.

ActivatedTables	Add LIMClasses	Description
Show Deactivated Only	Show Deleted Only	Only 'deactivated' tables will appear in the tree. This includes Tables/Columns that will be dropped during 'generation'. Note: If a Column is being 'deactivated' its Table will appear in the tree, regardless of whether the Table is being 'deactivated'.
Show Non-Activated Only	Show Un-Added	Only Tables/Columns that have not been activated or added appear in the tree. This includes Tables/Columns that were discovered in the LIM database and that do not exist in the Modeler databases and that have not been flagged to be 'activated' or added. Note: If a Column is non-activated or not added, the Table appears in the tree, even if the table itself is already added or activated. Expand the table to see the affected column.
Show Deltas Only	Show Deltas Only	Only Tables/Columns that have been flagged for activation/deactivation or addition/deletion appear in the tree. Note: If a Column is 'flagged' its Table will appear in the tree, regardless of whether the Table is 'flagged'.
Expand All	Expand All	Causes all Table nodes in the tree to expand, revealing all Column nodes in the tree.
Collapse All	Collapse All	Causes all Table nodes in the tree to collapse, hiding all Column nodes in the tree.

The dialog also has one of the following Generate action buttons and a Cancel button:

- **Generate ActivatedTables in Modeler** - This initiates the generation process which creates or removes ActivatedTables from the Modeler database.
- **Generate LIMClasses in Modeler** - This initiates the generation process which adds or deletes LIMClasses from the Modeler database.
- **Cancel** - Dismisses the dialog without making any changes to the Modeler database.

Note: Closing the Discover dialog after making selections but before Generating the data in the Modeler cancels any selections made. This is the same as selecting the Cancel button.

During the Discovery process, Cogility Modeler uses JDBC to map the datatypes in the legacy database to the appropriate Modeler datatype. For example, a BLOB datatype in a legacy database maps to a byte array data type in Modeler; and data, time, and timestamp datatypes map to the Modeler timestamp datatype.

Using a LIM in passive mode

In passive mode, [SQL Statements](#) or [LIMClasses](#) are used to query the legacy database for data.

LIMClasses

In the Modeler, a LIMClass can be defined under a LIM, then, the LIMClass can be used in Action Semantics to access data from a table in that LIM's database. The 'locate', 'locateAll', and, 'locateAll orderedBy' statements can access the LIMClasses. For information on using these Action Semantics, refer to following sections of the *Using Action Semantics with Cogility Studio*: "["locate" on page 41](#)", and "["locateAll orderedBy" on page 41](#)".

Once you have a LIMClass instance, you can reference its attributes (columns) directly in Action Semantics using the format `<aLimTableInst.anAttrName>`. You can assign values to LIMClass attributes in memory, however, those values are NOT written back to the legacy database. You can create new instances of LIMClasses in memory, however, those instances are NOT be written to the legacy database.

To add LIMClasses:

1. In Cogility Modeler's tree view, select the LIM.
2. Do one of the following:
 - From the **Selections** menu, select **Legacy Modeling> Discover Tables (LIMClasses)....**
 - Right-click on the LIM and select **Legacy Modeling> Discover Tables (LIMClasses)...** from the right-click menu.
3. Select the tables and columns to import as LIMClasses from the discovery dialog.
 - a. To select a table and all its columns:
 - Highlight the table name.
 - From the **Selection** menu, select **Add Selection(s)**.
 - Click **Generate LIMClasses In Modeler**. The table is added to the LIM as a LIMClass.
 - b. To select a subset of columns from a table:
 - Click the expand icon  next to the table name.
 - Select the column names you wish to add.
To select multiple columns, hold the Shift key while selecting the column names.
 - From the **Selection** menu, select **Add Selection(s)**.
 - Click **Generate LIMClasses In Modeler**. The table and the selected columns are added to the LIM as a LIMClass.

To remove LIMClasses:

1. In Cogility Modeler's tree view, select the LIM.
2. Do one of the following:
 - From the **Selections** menu, select **Legacy Modeling> Discover Tables (LIMClasses)....**
 - Right-click on the LIM and select **Legacy Modeling> Discover Tables (LIMClasses)...** from the right-click menu.
3. Select the tables and columns to import as LIMClasses from the discovery dialog.
 - a. To select a table and all its columns:
 - Highlight the table name.
 - From the **Selection** menu, select **Delete Selection(s)**.
 - Click **Generate LIMClasses In Modeler**. The table is deleted from the LIM as a LIMClass.
 - b. To select a subset of columns from a table:

- Click the expand icon  next to the table name.
- Select the column names.
To select multiple columns, hold the Shift key while selecting the column names.
- From the **Selection** menu, select **Delete Selection(s)**.
- Click **Generate LIMClasses In Modeler**. The table and the selected columns are deleted from the LIM as a LIMClass.

Note: If you are using LIMClasses in Action Semantics, red flags will appear on the artifacts affected after the LIMClass is deleted. You are not required to fix the Action Semantic usages before deleting, however, you will need to fix the errors before pushing the model.

Adding operations to a LIMClass

To add operations to LIMClass:

1. In Cogility Modeler's tree view, select a LIMClass.
2. Select the Operations tab in the table editor and click **New**.
3. Enter a name for the new operation and click **OK**.
4. Select the new operation and click **Edit**.
5. Define the operation. For instructions on defining an operation, see “Operations” on page 41.

SQL Statements

SQL statements allow you to access the data in the LIM using SQL execute and select statements. Each **executeSQL** object and **selectSQL** object can only contain a single SQL statement. You can group multiple SQL statements into **SQLBlock** objects.

Note: Using SQL object with a LIM is different from using SQL objects and statements with the PEAR run-time repository, the data store for your model. To use SQL statements with the run-time repository, see “Queries” on page 99 of the guide, *Using Action Semantics with Cogility Studio*.

LIM objects include execute statements, select statements and SQL blocks that hold the actual SQL statements that query the external (legacy) database. You reference these statement objects in the actions of other model objects or separately from model execution with in-line statements. With either in-line SQL or statement objects, you can pass attribute values as parameters dynamically. See “In-line SQL statements” on page 67 and “Statement objects” on page 69.

Cogility does not provide syntax checking or validation for SQL statements.

In-line SQL statements

Once you have a LIM defined (See “Legacy Information Model” on page 61), you can write SQL statements that query or modify the legacy database by using either the **selectSQL()** or **executeSQL()** actions. These statements may be used to write in-line SQL, or they may be used in statement objects. See “Statement objects” on page 69.

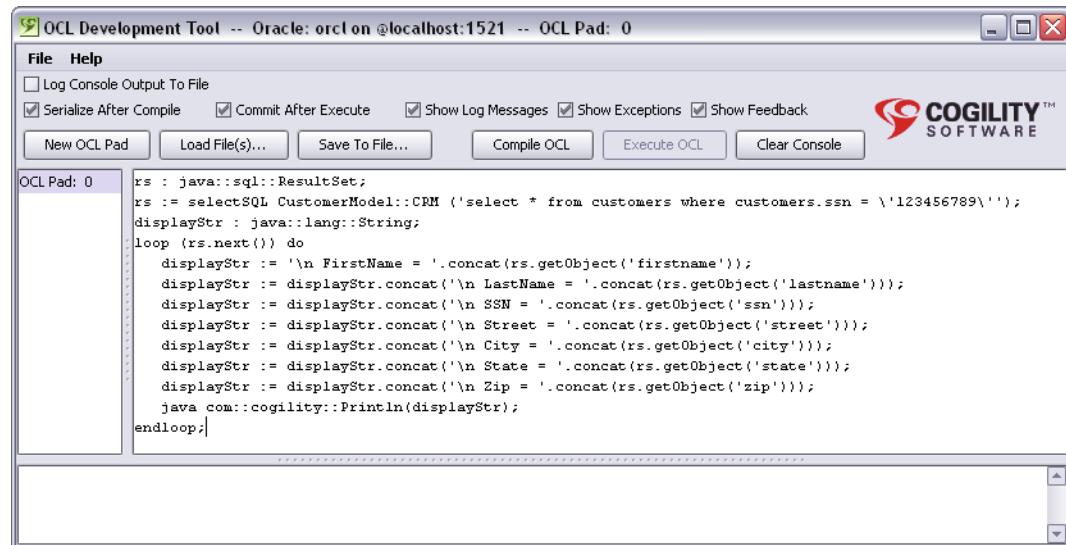
In-line statements are most useful in actions that you compile and execute separately from model execution; you can use the Cogility Action Pad to run these scripts.

There are two kinds of in-line SQL statements:

- “Select statements” on page 68
- “Execute statements” on page 68

Select statements

A select statement reads the legacy database for specific columns and rows. You submit a select statement using the `selectSQL()` keyword. The invocation of a select statement returns a `java::sql::ResultSet`. See “[selectSQL](#)” on page 52 of the guide, *Using Actions in Cogility Studio* for more information.



The screenshot shows the OCL Development Tool interface. The title bar reads "OCL Development Tool -- Oracle: orcl on @localhost:1521 -- OCL Pad: 0". The main window has several tabs at the top: File, Help, Log Console Output To File (unchecked), Serialize After Compile (checked), Commit After Execute (checked), Show Log Messages (unchecked), Show Exceptions (checked), Show Feedback (checked), New OCL Pad, Load File(s)..., Save To File..., Compile OCL, Execute OCL, and Clear Console. On the right side, there is a logo for COGILITY SOFTWARE. The central area is titled "OCL Pad: 0" and contains the following OCL code:

```
rs : java::sql::ResultSet;
rs := selectSQL CustomerModel::CRM ('select * from customers where customers.ssn = \'123456789\'');
displayStr : java::lang::String;
loop (rs.next()) do
    displayStr := '\n FirstName = '.concat(rs.getObject('firstname'));
    displayStr := displayStr.concat('\n LastName = '.concat(rs.getObject('lastname')));
    displayStr := displayStr.concat('\n SSN = '.concat(rs.getObject('ssn')));
    displayStr := displayStr.concat('\n Street = '.concat(rs.getObject('street')));
    displayStr := displayStr.concat('\n City = '.concat(rs.getObject('city')));
    displayStr := displayStr.concat('\n State = '.concat(rs.getObject('state')));
    displayStr := displayStr.concat('\n Zip = '.concat(rs.getObject('zip')));
    java com::cogility::Println(displayStr);
endloop;
```

Figure 3-4: In-line select statement

To use an in-line select statement:

1. Open an actions editor such as the Cogility Action Pad.

See “[Running Cogility Action Pad](#)” on page 80 of the guide *Model Deployment & Execution in Cogility Studio*.

2. Create the select statement.

See “[selectSQL](#)” on page 52 of the guide, *Using Actions in Cogility Studio* for more information.

3. Compile and execute the actions.

Execute statements

An execute statement performs an operation against the legacy database for specific columns and rows. The operation may be an INSERT, UPDATE, or DELETE in an execute statement passed as a parameter and enclosed in single quotes (''). The operation returns a `java.lang.Integer` that usually describes the number of rows edited. You submit an execute statement using the

`executeSQL()` keyword. See “[executeSQL](#)” on page 39 of the guide, *Using Actions in Cogility Studio* for more information.

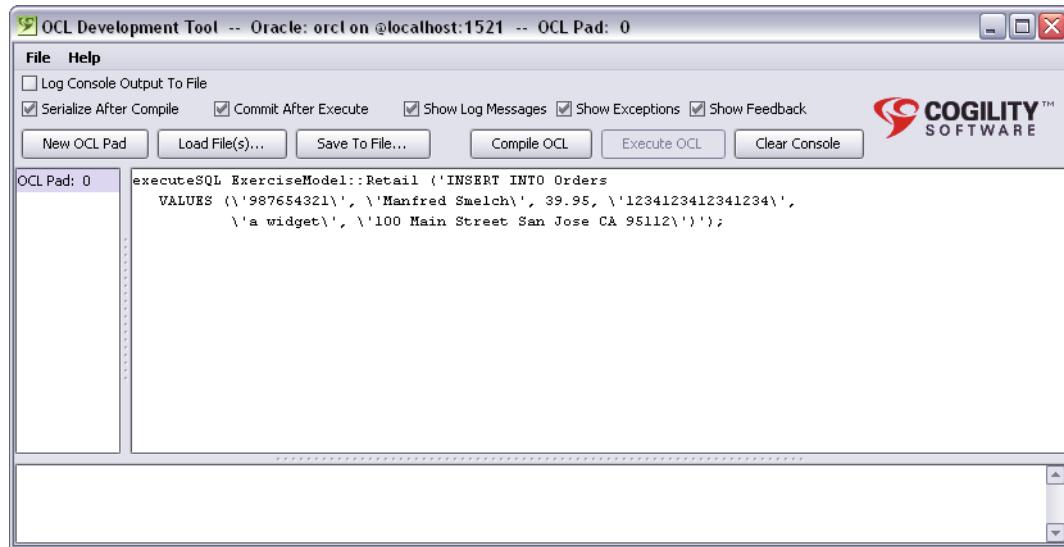


Figure 3-5: In-line execute statement

To use an in-line execute statement:

1. Open an actions editor such as the Cogility Action Pad.

See “[Running Cogility Action Pad](#)” on page 80 of the guide *Model Deployment & Execution in Cogility Studio*.

2. Create the operation statement.

See “[executeSQL](#)” on page 39 of the guide, *Using Actions in Cogility Studio* for more information.

3. Compile and execute the actions.

Statement objects

The statement objects of a LIM encapsulate the selection and execution SQL statements and provide the means to pass values to the SQL statements as parameters programmatically. You can reuse statement objects in your actions, passing to them the values for the parameters to the statements.

Statement object SQL syntax

You enter the SQL statement substituting the attribute values for parameter names identified by a question mark and included in brackets. For example, in the figure below, to specify the `ssn` attribute of the `customers` table, the select statement takes `?[ssn]` as a parameter. Also, as parameters are added, they appear in the Parameter Names field above.

Note: The SQL statement in Cogility Modeler omits the semicolon (`;`) normally appended at the end of the line.

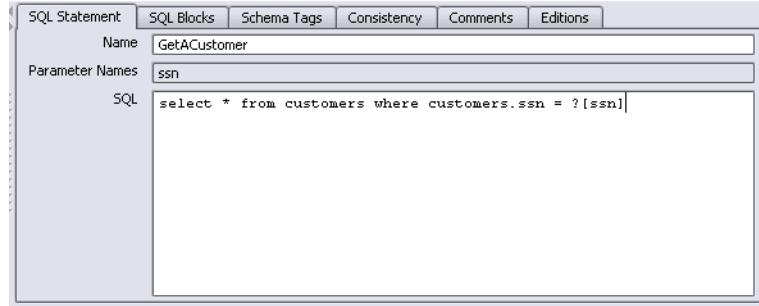


Figure 3-6: SQL syntax for a select statement object

In the figure below, to specify the `custID` attribute of the `account` table, the insert statement takes `?[custID]` as a parameter.

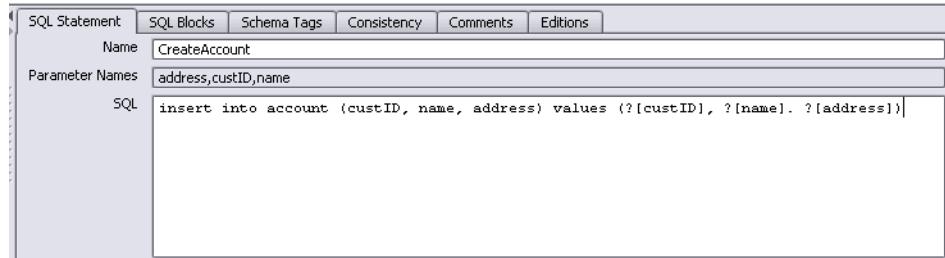


Figure 3-7: SQL syntax for an execute statement object

There are three kinds of statement objects:

- Select objects encapsulate an SQL select statement. See “[Select objects](#)” on page 70.
- Execute objects encapsulate an SQL execute statement. See “[Execute objects](#)” on page 71.
- SQL block objects let you combine select objects, execute objects or both. See “[SQL block objects](#)” on page 73.

Select objects

A select object encapsulates an SQL select statement. You add the object to your model’s LIM, then define the SQL, specifying parameter names.

In your actions, you invoke a select statement using the `selectSQL()` keyword, as described for in-line select statements in “[Select statements](#)” on page 68, except that instead of specifying the entire SQL statement and passing it as a parameter, you pass only the attributes and values as parameters. Use the following format when referring to an SQL select statement object in your actions:

```
selectSQL <model>::<LIM>::<select statement object> (
<param>: <value>,
<param>: <value>...);
```

See “[selectSQL](#)” on page 52 of the guide, *Using Actions in Cogility Studio* for more information about using in-line select statements.

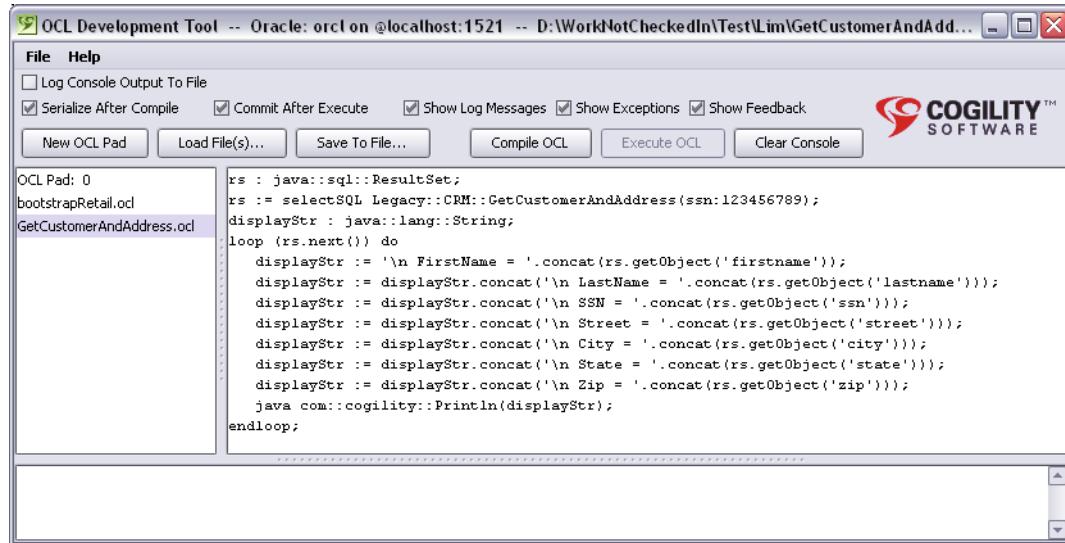


Figure 3-8: Using a select statement object

To create a select statement object:

1. In Cogility Modeler’s tree view, select the LIM.
2. Click the plus button and select **Add Select Statement**.
In the content view, the editor window for the select statement displays.
3. In the select statement editor, in the **SQL Statements** tab, in the **Name** field, enter a name.
4. In the **SQL** field, enter the SQL select statement.
See “[Statement object SQL syntax](#)” on page 69.

To use a select statement object:

1. Open an actions editor such as the Cogility Action Pad.

See “[Running Cogility Action Pad](#)” on page 80 of the guide *Model Deployment & Execution in Cogility Studio*.

2. Create the select statement.
3. Compile and execute the actions.

Execute objects

An execute object encapsulates an SQL INSERT, UPDATE or DELETE statement. You add the object to your model’s LIM, then define the SQL, specifying parameter names.

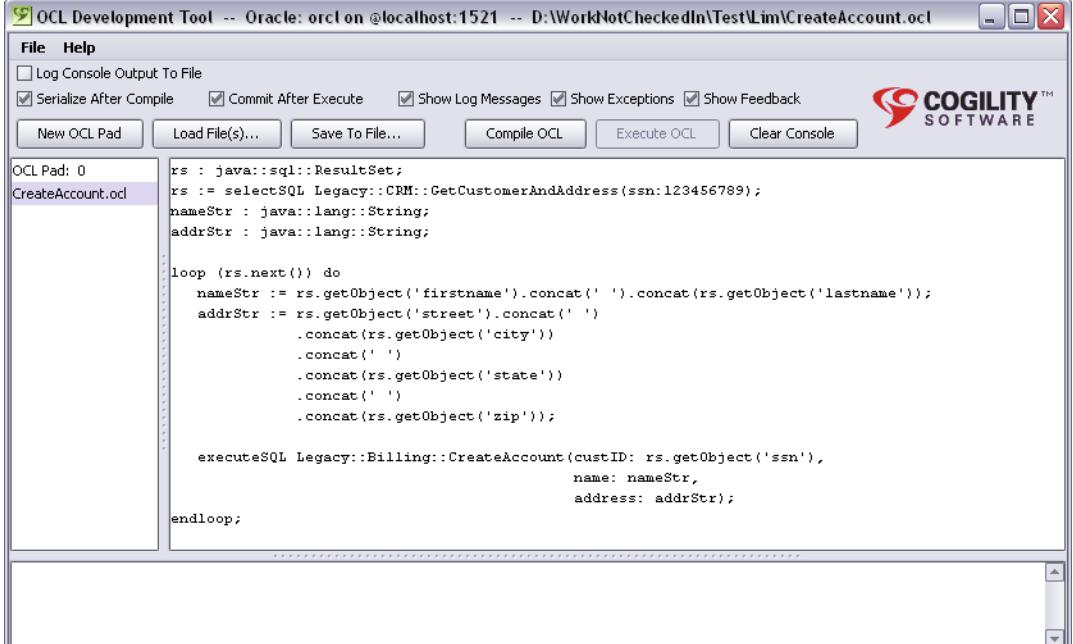
In your actions, you invoke an execute statement using the `selectSQL()` keyword, as described for in-line select statements in “[Execute statements](#)” on page 68, except that instead of specifying the entire SQL statement and passing it as a parameter, you pass only the attributes and values as parameters. Use the following format when referring to an SQL execute statement object in your actions:

```

executeSQL <model>::<LIM>::<execute statement object> (
<param>: <value>,
<param>: <value>...);
  
```

See “[executeSQL](#)” on page 39 of the guide, *Using Actions in Cogility Studio* for more information about using in-line execute statements.

In the example below, the execute statement, `executeSQL` refers to the `CreateAccount` execute object and passes the `custID`, `name` and `address` attribute values.



```
File Help
Log Console Output To File
Serialize After Compile Commit After Execute Show Log Messages Show Exceptions Show Feedback
New OCL Pad Load File(s)... Save To File... Compile OCL Execute OCL Clear Console
OCL Pad: 0
CreateAccount.ocl
rs : java::sql::ResultSet;
rs := selectSQL Legacy::CRM::GetCustomerAndAddress(ssn:123456789);
nameStr : java::lang::String;
addrStr : java::lang::String;

loop (rs.next()) do
    nameStr := rs.getObject('firstname').concat(' ').concat(rs.getObject('lastname'));
    addrStr := rs.getObject('street').concat(' ')
        .concat(rs.getObject('city'))
        .concat(' ')
        .concat(rs.getObject('state'))
        .concat(' ')
        .concat(rs.getObject('zip'));

    executeSQL Legacy::Billing::CreateAccount(custID: rs.getObject('ssn'),
                                                name: nameStr,
                                                address: addrStr);
endloop;
```

Figure 3-9: Using an execute statement object

To create an execute statement object:

1. In Cogility Modeler’s tree view, select the LIM.
2. Click the plus button  and select **Add Execute Statement**.
In the content view, the editor window for the select statement displays.
3. In the execute statement editor, in the **SQL Statements** tab, in the **Name** field, enter a name.
4. In the **SQL** field, enter the SQL INSERT, UPDATE or DELETE statement.
See “[Statement object SQL syntax](#)” on page 69.

To use an execute statement object:

1. Open an actions editor such as the Cogility Action Pad.
See “[Running Cogility Action Pad](#)” on page 80 of the guide *Model Deployment & Execution in Cogility Studio*.
2. Create the execute statement.
3. Compile and execute the actions.

SQL block objects

An SQL block object describes the execution order of a group of statement objects. You can combine SQL statement objects with SQL block objects and use them in your actions. The statement objects may be select objects, execute objects or both.

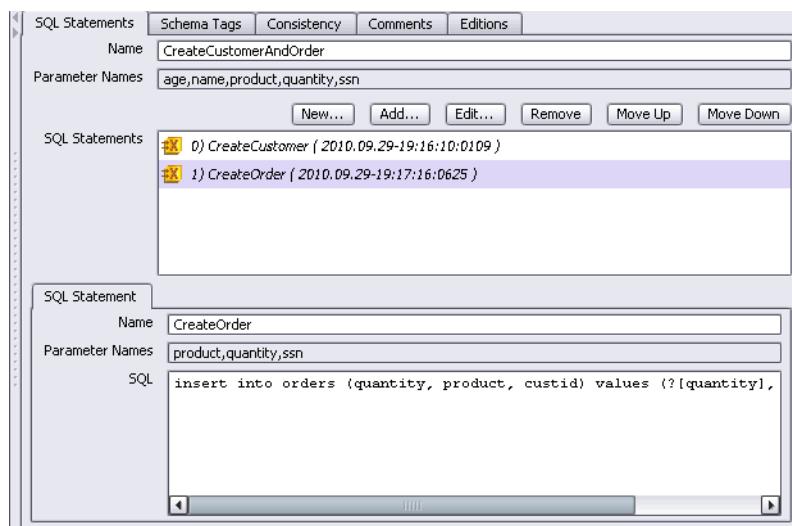


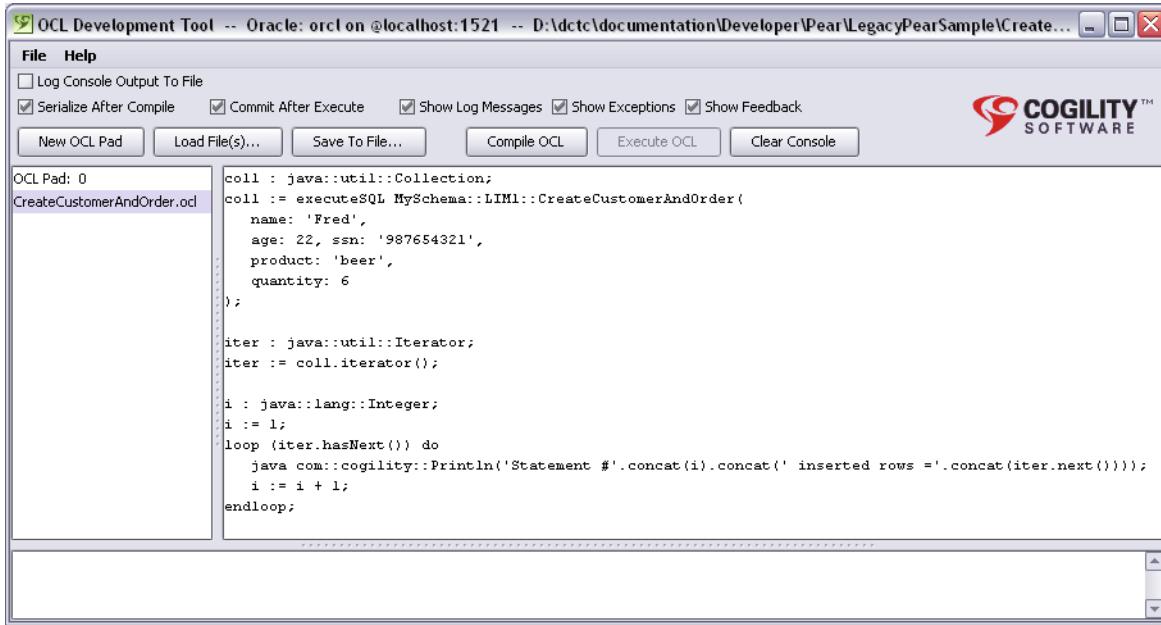
Figure 3-10: SQL block object definition

In your actions, you invoke an SQL block using the `executeSQL()` keyword, as described for in-line select statements in “[Execute statements](#)” on page 68, except that instead of specifying the entire SQL statement and passing it as a parameter, you pass only the attributes and values for the included statement objects as parameters. The parameters are listed in the Parameter Names field of the SQL block object.

Use the following format when executing a block object in your actions. Note that when used with an SQL block object, the `executeSQL()` operation returns a `java::util::Collection` through which you can then iterate.

```
coll : java::util::Collection;
coll := executeSQL <model>:<LIM>:<sql block object> (
<param>: <value>,
<param>: <value>...);
```

In the example below, the execute statement, `executeSQL` refers to the `CreateCustomerAndOrder` block object and passes the `age`, `name`, `product`, `quantity` and `ssn` attribute values.



The screenshot shows the OCL Development Tool interface. The title bar reads "OCL Development Tool -- Oracle: orcl on @localhost:1521 -- D:\dctc\documentation\Developer\Pear\LegacyPearSample\Create...". The menu bar includes "File" and "Help". Below the menu are several checkboxes: "Log Console Output To File", "Serialize After Compile" (checked), "Commit After Execute", "Show Log Messages", "Show Exceptions", and "Show Feedback". Buttons for "New OCL Pad", "Load File(s)...", "Save To File...", "Compile OCL", "Execute OCL", and "Clear Console" are also present. The main area is titled "OCL Pad: 0" and contains the following Java code:

```
coll : java::util::Collection;
coll := executeSQL MySchema::LIM1::CreateCustomerAndOrder(
    name: 'Fred',
    age: 22, ssn: '987654321',
    product: 'beer',
    quantity: 6
);

iter : java::util::Iterator;
iter := coll.iterator();

i : java::lang::Integer;
i := 1;
loop (iter.hasNext()) do
    java com:cogility::Println('Statement #'.concat(i).concat(' inserted rows ='.concat(iter.next())));
    i := i + 1;
endloop;
```

Creating a block object

To create an SQL block object:

1. In Cogility Modeler's tree view, select the LIM.
2. Click the plus button  and select **Add SQL Block**.
In the content view, the editor window for the SQL block displays.
3. In the block object editor, in the **SQL Statements** tab, in the **Name** field, enter a name.
4. Create or add the statement objects.
See “[Creating statement objects in a block object](#)” on page 74 and “[Adding statement objects to a block object](#)” on page 75.
5. Change the order of the statement objects, if necessary.
When you create statement objects from within the SQL block object or add them to the list under the SQL Statements field, they appear at the end of the list. See “[Ordering statement objects in a block object](#)” on page 75.

Creating statement objects in a block object

To create statement objects in a block object:

1. In the block object editor, in the **SQL Statements** tab, above the **SQL Statements** field, click **New**.
2. In the **Select Type** dialog, select the statement object type and click **OK**.

The statement object is added to the list of statement objects in the SQL Statements field. It appears after any other objects in the list. Also, as statement objects are added to the list, their parameters appear in the Parameter Names field above. To change the order of the statement objects, see “[Ordering statement objects in a block object](#)” on page 75. Also, the statement object’s editor appears in the SQL Statement tab at the bottom of the content view.

3. With the statement object selected, click **Edit** to define the statement object.

See “[Select objects](#)” on page 70 and “[Execute statements](#)” on page 68.

You can also remove statement objects from the list by clicking Remove. Doing so removes the statement object from the SQL block object, but does not delete it from the model.

Adding statement objects to a block object

To add an existing statement object:

1. In the block object editor, in the **SQL Statements** tab, above the **SQL Statements** field, click **Add**.
2. In the **Select Object(s)** dialog, choose the statement object and click **OK**.

You can also remove statement objects from the list by clicking Remove. Doing so removes the statement object from the SQL block object, but does not delete it from the model.

Ordering statement objects in a block object

To change the order of the statements in the list:

1. In the block object editor, in the **SQL Statements** tab, select the statement object.
2. Click **Move Up** or **Move Down** to change the object’s location in the list.

Using a block object

To use an SQL block object:

1. Open an actions editor such as Cogility Action Pad.

See “[Running Cogility Action Pad](#)” on page 80 of the guide *Model Deployment & Execution in Cogility Studio*.

2. Create the execute statement.
3. Compile and execute the actions.

Using a LIM in active mode

An activated LIM table lets you monitor specific columns or specific tables for changes to the data therein. For each active table or column of a legacy database, you describe actions in your LIM that direct the model to do something when the legacy data changes.

When a table is activated, an ActivatedTable is added to the model. When the model is pushed, a “trigger” is added to the table in the Legacy database and two tables are added to the target schema. Whenever a row is inserted, deleted or updated in the legacy database table, the “trigger” fires and writes records to these target tables. The pushed model application can be configured to poll the legacy database to read these tables and determine if a change has occurred in the database. If a change has occurred, then the appropriated associated action is executed. You can define separate actions for insert, delete, or update actions in the database.

When an ActivatedTable is deactivated, the ActivatedTable is removed from the model. When the model is pushed, the “trigger” is removed from the Legacy table. The Legacy table and its rows are unaffected.

Operations and Methods can be defined for LIM ActivatedTables. In addition, you can use an ActivatedTable type as an Operation argument type or an Operation return type.

Activating LIM tables creates a polling service which identifies changes to the external database, captures what has changed, and allows your model to appropriately respond to those changes. Activated tables track the insertion of new records and the update and deletion of existing records. Action fields are available to code specific behavior to the different changes.

Activating a LIM

To activate a LIM:

1. Run the **Discover Tables (Activated Tables)**... command.
See “[Discover Tables](#)” on page 62.
2. Specify the tables for activation.
See “[ActivatedTables](#)” on page 76.
3. Describe the actions for activated tables.
See “[Activated table actions](#)” on page 77.

ActivatedTables

After you have run the Discover tables command (see “[Discover Tables](#)” on page 62), you can mark specific tables or columns for change monitoring. After selecting the tables and columns to activate, you select the Generate action to place the ActivatedTables in the model. As changes to the target database occur during run time, your pushed model monitors specific tables or columns for those changes. You write actions to handle the database changes in the model as described in “[Activated table actions](#)” on page 77.

When Cogility Modeler activates a legacy database table or column, it installs in that column a trigger that reports the updates. Cogility Modeler also creates two new tables in the database that keep track of the changes. Note that Cogility Modeler does not modify the existing data, and the legacy database will continue to work with its API as usual.

The red inconsistency flag appears until you add actions to the activated table to specify what happens in the model to updated table data. See “[Activated table actions](#)” on page 77.

Deactivating a table or column removes it from consideration for updates during run time. By deactivating a table or column, you are removing from it the triggers that monitor the table or column for updates. The tables marked for deactivation appear with a red X.

To activate tables:

1. In the **Discover Tables** window, right-click the tables or columns you wish to activate and select **Activate Selection(s)**.
2. (Optional) preview the activation.
 - a. Right-click one of the items and select **Explain Selection**.
A dialog appears with information on the item and what action will be taken when you select Generate Modeler Data.
 - b. Click **OK** to close the dialog.
3. Click **Generate ActivatedTables in Modeler**.
4. In the information dialog, click **OK**.
The activated tables appear in Cogility Modeler.

To deactivate a LIM table:

1. In Cogility Modeler’s tree view, select the LIM.
2. From the **Selection** menu, choose **Legacy Model > Discover Tables**.

3. Right-click the table or columns that you wish to deactivate and select **Deactivate Selection(s)**.
4. (Optional) preview the deactivation.
 - a. Right-click one of the items and select **Explain Selection**.
A dialog appears with information on the item and what action will be taken when you select Generate Modeler Data.
 - b. Click **OK** to close the dialog.
5. Click **Generate ActivatedTables in Modeler**.
6. Click **OK**.
Previously activated tables are removed from Cogility Modeler.

Activated table actions

An activated table requires that some action be performed when the data in that table changes. Once you have activated a table, you must enter actions in at least one of the actions tabs. The actions may communicate the updated data to the model (in a MIM or DIM table), it may publish a message with the data or perform any other activity you can describe with actions. You access the data with the `row`, `oldRow`, or `newRow` bound variables. Activated table actions may be described in any of three tabs.

The **Insert Action** tab describes the action to take when a new record is entered in the table or column. The `row` bound variable provides access to the data of that record.

The **Update Action** tab describes the action to take when an existing record is updated. Use the `oldRow` bound variable to refer to the record before the update, and the `newRow` bound variable to refer to the record after the update.

The **Delete Action** tab describes the action to take when an existing record is deleted from the table or column. The `row` bound variable provides access to the data of that record.

The `row`, `oldRow` or `newRow` bound variables are available through code assist, as shown in the picture below.

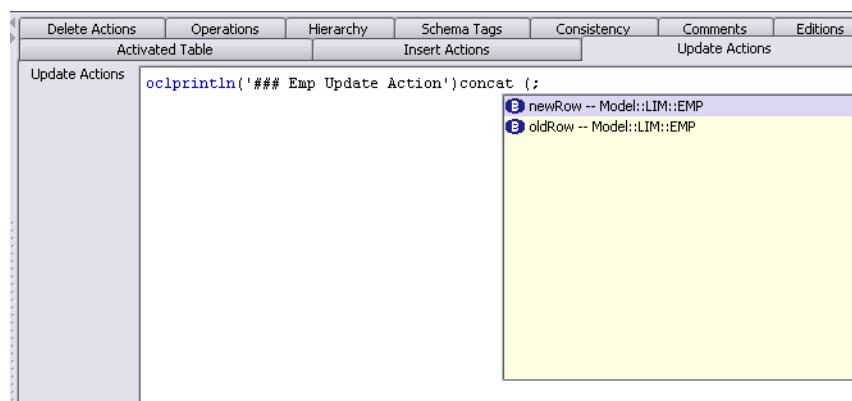


Figure 3-11: Activated table update actions code assist

In your `ActivateTable` or `ActivateColumn` actions you may use **In-line SQL statements** or **Statement objects** that refer to the same legacy database. Note that you are performing activities on the legacy database that are triggered by changes to that database, and the transaction processing may not behave predictably or reliably. It is probably not wise to make changes to tables that are related to tables whose updates trigger the change actions. However, your LIM might monitor a table for

changes, then read the related tables and report changes to those. For example, if a Customer record changes, your actions could read the related Address table for updates as well.

Also, your actions may use [In-line SQL statements](#) or [Statement objects](#) that refer to a different legacy database, updating the records on it when changes to the triggering legacy database take place.

When a record is added, updated or deleted, this action triggers an update to the monitoring tables created on the legacy database by Cogility Modeler. When the actions for an activated table successfully execute, the monitoring tables are cleared.

Creating actions for an activated table

To create actions for an activated table:

1. In Cogility Modeler's tree view, select the LIM ActivatedTable.
2. In the activated table editor, select the **Insert Actions**, **Update Actions**, or **Delete Actions** tab.
3. Enter actions in the actions field.

The actions compile when you click out of the field. You can check the consistency of the actions in the Consistency tab.

Adding operations to an activated table

To add operations to an activated table:

1. In Cogility Modeler's tree view, select a LIMActivatedTable.
2. Select the Operations tab in the table editor and click **New**.
3. Enter a name for the new operation and click **OK**.
4. Select the new operation and click **Edit**.
5. Define the operation. For instructions on defining an operation, see "[Operations](#)" on page [41](#).

Activation Configuration Parameters

There is a service associated with LIM monitoring which must be turned on in order to use ActivatedTables. The following configuration parameter must be set to true to turn on monitoring. It is found in the Monitor Services section of the **defaultConfigurations.txt** file.

`com.cogility.monitorservice.2.enabled`

Description: Enables or disables the LIM monitoring service.

Legal values: true - turns on the service. False turns off the service.

Default: False

Note: You can also turn the LIM monitor service on in Insight. However, if you turn it on in Insight and the app service is restarted, you must also restart the service. If you are using ActivatedTables set the parameter in the configurations file.

The following run-time configuration parameters are associated to the function of LIM activation. They are found in the LIM Activation Parameters section of the **defaultConfigurations.txt** file.

`com.cogility.lim.pollinterval`

Description: Poll Interval (in millies) controls how often the master and details tables are polled.

Legal values: any positive integer.

Default: 5000

`com.cogility.lim.maxWorkerThreadCount`

Description: Controls how many threads to spawn off to process all the new events. Additional events will be queued up and wait in line to get processed.

Legal values: any positive integer.

Default: 5

`com.cogility.lim.maxCridsToRead`

Description: Controls approximately how many CR_IDs to read from the master/detail table in an activated LIM. Note this is approximate as there may be gaps in CR_ID.

Legal values: 0 or any positive integer.

Default: 0 (all available rows in the master/detail table)



Message modeling

Cogility Studio includes several artifacts that model the messages between an integrated application and the model, as well as messages within the model between the various modeling artifacts.

Message modeling artifacts include the following:

- Message or destination artifacts that describe the data communicated between the model and the integrated applications. These artifacts, based on the DOM Message type, may represent JMS messages or web service messages. See “[Messages](#)” on page 103.
- Conversion artifacts that convert the message data into event attributes (during inbound communication to the model) and conversion objects that convert event data into message attributes (during outbound communication from the model to the external application). See “[Conversions](#)” on page 91
- Event artifacts that spawn business processes. See “[Events](#)” on page 82.

These artifacts are illustrated in [Figure 4-1 on page 82](#) below. During run time, the external application sends a JMS message to the model (deployed as a J2EE composite application). This inbound communication is shown with the left to right arrow. A message to event (M2E) conversion object takes that message data and converts it to an event object. The event object attributes hold business object information that may be used in a business process. The illustration below shows

how the model's inbound communication takes a message, converts it into an event that starts a business process to create a Customer on the master information model.

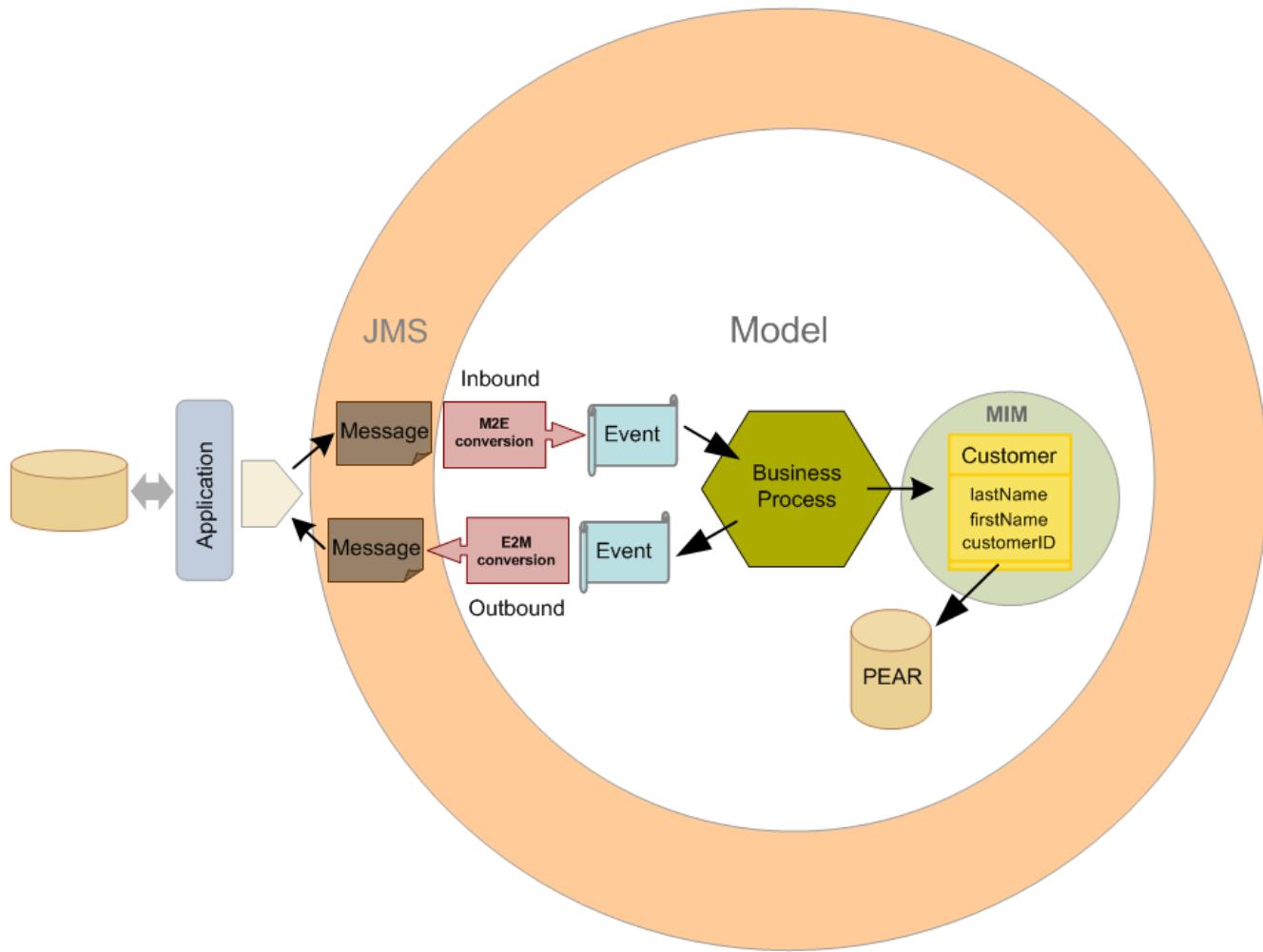


Figure 4-1: Inbound and outbound communication

Outbound communication, illustrated by the arrows pointing right to left, takes information from a business process in the form of an event, converts it with an event to message (E2M) conversion into a message that the model then sends to the external application.

Because each run-time object of the process, message, conversion and event, is modeled as an artifact, it is reusable. Furthermore, the business process that uses the message data is isolated from the message. This allows you to change either the message data or the business process that uses it independently.

Before you define a message model, you should first create your initial information and behavior models.

Events

Events handle asynchronous notification of business activity for the model. Incoming JMS messages and web services can generate events. A synchronous web service can trigger a long-lived,

asynchronous process using an event. Events can also be generated by actions. See “[event](#)” on page [23](#) of the guide *Using OCLPlus with Cogility Studio*.

Events may serve as stimuli for state machines, web services and other artifacts. They can both start state machines and effect transitions between states. The class or factory for the state machine must designate a start event; the state machine cannot execute without it. See “[Behavior model](#)” on page [109](#) for more information about state machines. An internal event may be published to or from a web service (see “[Internal event](#)” on page [84](#)).

An event may indicate the occurrence of some activity on a source system (either external or internal to the model). That system sends a JMS message to the model which is converted to an event. A business process (state machine) receives the inbound event and generates an outbound event as illustrated in the picture below. The outbound event may be published to JMS or used internally.

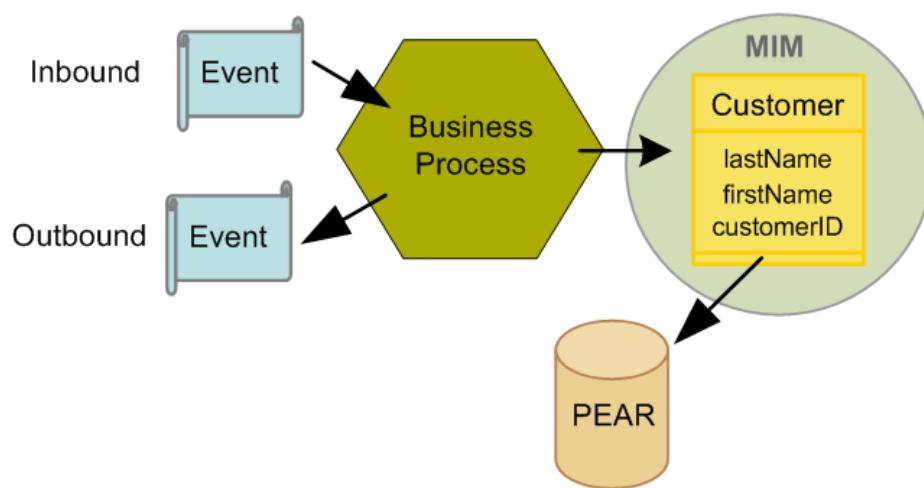


Figure 4-2: Events in business processes

For example, an integrated application would send new customer information to the model in a JMS message. The message is converted to the customer creation inbound event which starts a business process that creates the new Customer object in the run-time repository (PEAR). The business process that creates the new Customer also generates an outbound event with a customer ID to be communicated to the external application.

Events are defined by attributes that have a key, type, and value. However, events are transient, not persistent objects the way class instances are. While you may have a Customer object with lastName Smith and firstName John, you would not have a persistent event object with the same attributes. Instead, event instances are created at run time, their attribute values are populated by the conversions or business processes that generate them, and they are deleted when they have completed processing.

There are three types of events:

- Standard events define a set of data with event attributes for use in a state machine. See “[Standard event](#)” on page [84](#).
- DOS events specify a data object set to be used in a transformation embedded in a state machine. See “[DOS event](#)” on page [85](#).
- Time events provide a timed delay in a transition from one state to another in a state machine. See “[Time event](#)” on page [86](#).

Standard event

A standard event defines a set of data with event attributes. The attributes have a key (name), type (of the usual primitives) and a value. See “[Attributes](#)” on page 35. For inbound communication to the model, the event attributes are populated according to the specification of the associated M2E conversion (see “[Conversions](#)” on page 91), which gets the data from the inbound JMS message. For outbound conversion, the event attribute values populate the outbound JMS message according to the specified E2M conversion. Events may also be used internally, without an associated conversion. See “[Internal event](#)” on page 84.

When an event gets its data from or provides data for a data object set (DOS) for a transformation embedded in a state machine, use a DOS event. See “[DOS event](#)” on page 85.

Events are represented in Cogility Modeler’s tree view with an event icon:  . If the event has an associated operation, the event class or subclass where that operation is defined with a method has an icon that includes an O in the bottom right corner:  .

Abstract event

As with other artifacts that may have attributes and operations, an event may be abstract. An abstract event describes a superclass for other event objects and is not necessarily associated with a conversion. Usually you do not use the abstract event in the data path, but rather an event that designates the abstract event as its superclass. Events that designate a superclass inherit the event attributes from that superclass. See “[Superclass](#)” on page 33.

Internal event

An internal event is one that may be published to or from a model artifact other than a conversion. By default, an event is assumed to be part of a message to event (M2E) conversion or an event to message (E2M) conversion, utilizing the Java Messaging Service for external communication to and from the model. Such events are inconsistent with the modeling rules if they are not associated with a conversion, message and destination. To use an event in a web service or other model artifact and avoid an inconsistency flag, the event must be designated as internal.

Internal events must, however, be associated with either a state machine or a transition. That is, an internal event must either be the start event for a state machine or the trigger for a state machine transition. For example, a web service generates an event that starts a business process (such as a customer creation), or a business process triggers an event that launches some other business process within the model.

Creating a standard event

To create a standard event:

1. In the tree view, select the model container or package.
2. Click the **Add an Event** button , or from the **Selection menu**, choose **Domain Modeling > Add Event**.

Under the Events in the tree view, an untitled event appears and the event editor displays in the content view.

3. Under the **Event** tab, in the **Name** field, enter the name for the event.
The name you enter must be unique within the model container.
4. If necessary, designate this event’s superclass or specify this event as a superclass for other events. See “[Superclass](#)” on page 33.
5. If this is an abstract event, check the **Abstract** checkbox. See “[Abstract event](#)” on page 84.

6. If this is an internal event, check the **Is Internal Event** checkbox. See “[Internal event](#)” on [page 84](#).

7. Above the **Attributes** field click **New**.

- a. In the dialog window that displays, for **Name**, enter a name.
- b. For **Type**, select the type from the drop-down menu and click **OK**.

The attribute is listed in the Attributes field, and the Attribute tab is displayed.

8. If necessary, in the **size** field, set the size for String types.

This applies only to String types. Enter a character limit, as a positive integer. During run time, values with a greater number of characters are truncated to the limit you set. The default is 256 characters.

9. If you wish to add a comment to the attribute, select the **Comments** tab, then select **New**.

10. Enter a name for the comment, and click **OK**.

The comment name is listed in the Comment field, and the Comment tab is displayed.

11. Enter the text for the comment in the **Body** field.

12. Specify the event conversion artifacts.

See “[Event conversions](#)” on [page 91](#).

13. In the **Start Object** tab, specify the start object.

See “[Start object](#)” on [page 90](#).

14. In the **Operations** tab, define the event operation.

For events, the operation may be inherited from event superclass artifacts.

See “[Operations](#)” on [page 41](#)

DOS event

A DOS event is similar to a standard event in that both communicate data for use in a business process. Whereas the data in a standard event is held in event attributes, the data for a DOS event is held in a data object set for use in a transformation. Anytime a state machine includes a transformation, it must use a DOS event rather than a standard event to deliver the data. The transformation embedded in the state machine can be any of the transformation types: opaque

transformation map, transformation map, or transformation chain. See “Transformation model” on page 145 for more about transformations.

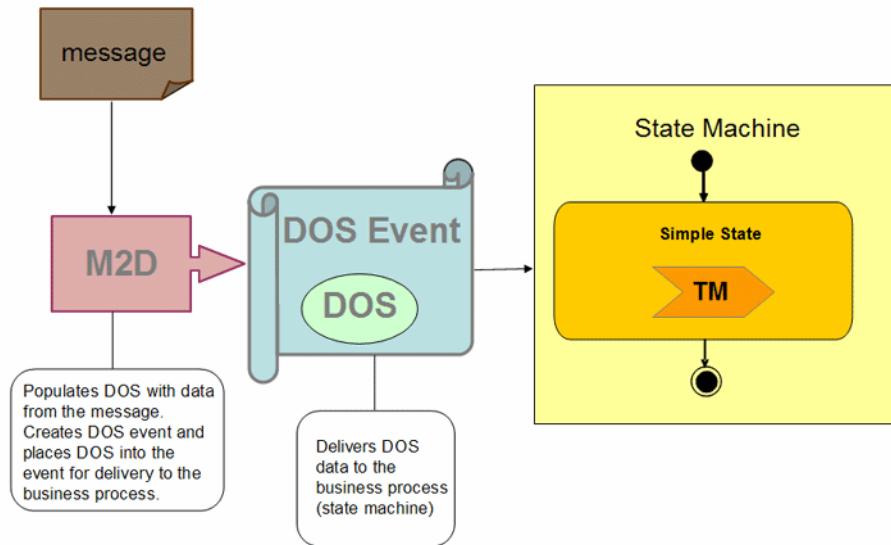


Figure 4-3: DOS event

The DOS event definition specifies the M2D conversion that owns the event, the start object that contains the state machine that uses the data for a transformation, and the started behavior.

DOS events are represented in Cogility Modeler’s tree view with a DOS event icon: . If the DOS event has an associated operation, the icon includes an O in the bottom right corner: .

Creating a DOS event

To create a DOS event:

1. In the tree view, select the model container or package.
2. Click the **Add a DOS Event** button , or from the **Selection menu**, choose **Domain Modeling > Add DOS Event**.
Under the DOS Events in the tree view, an untitled event appears and the event editor displays in the content view.
3. In the DOS event editor, under the **DOS Event** tab, in the **Name** field, enter the name for the DOS event.
The name you enter must be unique within the model container.
4. Specify the owning conversion.
See “DOS conversions” on page 94.
5. Specify the start object.
See “Start object” on page 90.
6. Define the DOS event operations.
See “Operations” on page 41.

Time event

Time events are used exclusively in the transitions of state machines. See “Transitions” on page 125. A time event triggers a transition in a state machine after a specified period of time. For example,

many commercial web sites cancel an order if the customer does not click “purchase” or otherwise commit the order within a specified period of time.

In [Figure 4-4](#), the TimeOut transition is triggered by the threeMinutesElapsed event.

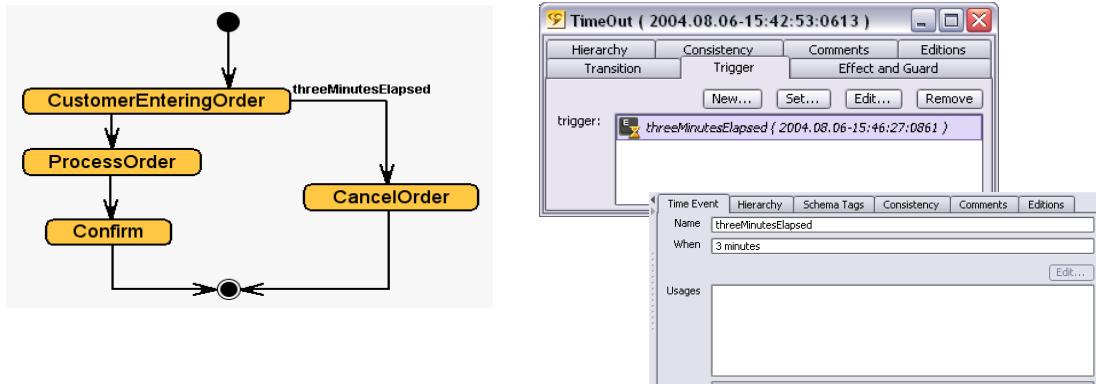


Figure 4-4: Time event

Time expression

A time event includes a time expression field (labeled "When") that identifies a time relative to when the state was started. For more information about using time events, see [“Time event” on page 86](#) of the guide, *Modeling with Cogility Studio*.

The time expression includes two elements:

- A time value that specifies how many units must pass before the transition occurs, expressed as one of the following:
 - A number (either an integer or a double)
 - A reference to a configuration parameter that specifies the time value. See [“Time value configuration parameter” on page 88](#).
- A time unit indicator. Valid time unit indicators are:
 - second** or **seconds**
 - minute** or **minutes**
 - hour** or **hours**
 - day** or **days**
 - week** or **weeks**
 - month** or **months**.

In [Figure 4-5](#), a time expression uses a numeric value to cause a transition once the specified number of minutes have elapsed.



Figure 4-5: Time expression with a numeric value

In [Figure 4-6](#), a time expression uses a configuration parameter reference to cause a transition once the configured number of seconds have elapsed.

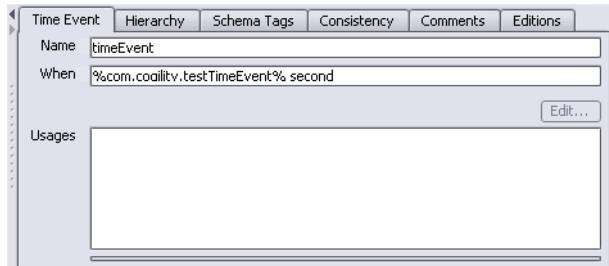


Figure 4-6: Time expression with a configuration reference

Time value configuration parameter

You can set the time value in a configuration parameter. Doing so allows you to change the value for the time expression, without having to change the model and re-push it onto the application server. You simply change the value in the configuration parameter, then either restart the application server or refresh the configuration parameters while the application server is still running. See [“Reload configuration message” on page 38](#).

To set the numeric value in a configuration parameter, you edit your `customConfigurations.txt` file and add a line for the parameter, for example:

```
com.cogility.testTimeEvent=20
```

The value set for the time value parameter specifies the number of units that must elapse before a time event triggers a transition in the state machine.

If you add a parameter for a time value but do not set the value or set an invalid value, a default value of 1 is used. An inconsistency message appears in the command window:

```
Time Expression has a Configuration Parameter, %com.cogility.timeEvent2%,  
that has a value of 'ugh' which is not a valid number. If invalid at  
runtime, a default of 1 will be used.
```

You must set value for the parameter then re-load your configuration parameters or restart Cogility Modeler to register the parameter.

Time Expression Override

For States that transition due to a Time event, an expression can be defined to override the value specified in the Time event, regardless if it was defined with a numeric value or a configuration parameter.

A Time Expression Override is defined on the State object that is transitioning. The **Time Event Actions** field is used to specify the override expression. The expression returns a Long value which denotes Seconds, not the time unit defined in the Time event. If the override expression evaluates to a non-numeric value, the State object will become inconsistent (in Modeler), with a message indicating that the expression evaluated to an invalid return type.

If no override expression exists, it is ignored at run-time.

Creating a time event

While you can create a time event apart from a state machine transition, as described here, you generally set a time event from within a transition. See [“Event trigger” on page 127](#).

To create a time event:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **Add a Time Event** button , or select **Selection > Domain Modeling > Add Time Event** from the menu.
3. In the **Name** field, enter a name for the time event.
The name must be unique within the model container.
4. In the **When** field, enter the time expression in the following format:

<time_value> <time_unit_identifier>

The time_value is either:

- A number (either an integer or a double)
- A configuration parameter reference in the format:

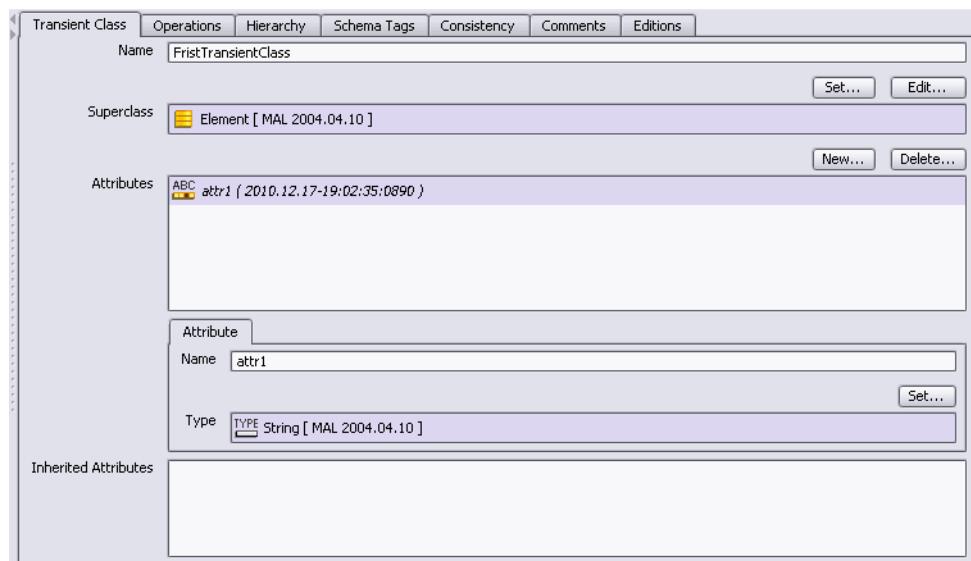
%configuration_parameter_reference%

Transient class

Transient classes are similar to Messages. However, transient classes have attributes whose types allow more than the basic, primitive types allowed in messages. Transient class attribute types can be any of the primitive types. In addition, they can be Collection, List, Map, Object or Set. Transient classes can be used in OCL, and, they can have operations and inheritance (superclasses/subclasses).

To create a transient class:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **Add a Transient Class** button , or select **Selection > Domain Modeling > Add Transient Class** from the menu.



3. Enter a name in the **Name** field.
4. If the class has a superclass, above the **Superclass** field, click **Set...**.

The Select Object(s) window displays a list of classes for use as superclasses. Use the Object Filter field to display a filtered list of classes. For example, Type "T" and the list is filtered to show only those classes beginning with the letter "T".

- a. Select the class you want to use as the superclass and click **OK**.
- b. To make any changes to the superclass, click **Edit** and make the desired changes.

Note: Any changes you make to the superclass will affect each instance where the superclass is used. Be sure this is the desired effect, before you make edits to a superclass.

5. Above the **Attributes** field click **New**.

- a. In the dialog window that displays, for **Name**, enter a name.
- b. For **Type**, select the type from the drop-down menu and click **OK**.

The attribute is listed in the Attributes field, and the Attribute tab is displayed.

6. In the **Operations** tab, define the transient class operation.

See "[Operations](#)" on page 41

Start object

An event that serves as the start event to a state machine must designate that state machine's class or factory as the event's start object. You can also designate the start event from within the class or factory; see "[Start event](#)" on page 130.

A single event may be reused by many different classes or factories as a start event. For each use of the event, you must designate the started behavior of that class or factory. A class or factory may have many behaviors, but only one may be associated with the start event.

Designating the start object

To designate a start object:

1. In the tree view, select the event artifact that will serve as the start event.

See "[Standard event](#)" on page 84.

2. In the event editor, in the **Start Object** tab, above the **startObject** field, click **Add**.

The Select Object window displays with the classes currently defined for the model.

3. Select the start object and click **OK**.

The selected class is displayed in the **startObject** field.

4. In the **startObject** field, select the class.

The StartedBehavior tab appears at the bottom of the content view. There may be more than one behavior for a class, so you have to designate the behavior which pertains to this start event.

5. Above the **startedBehavior** field, click **Set**.

The available behaviors for the selection display.

6. Select the started behavior and click **OK**.

The selected behavior is displayed in the **startedBehavior** field. You can display the state machine diagram for editing by selecting the behavior and clicking **Edit Diagram**.



Conversions

Conversions are logical handlers for cases where an event needs to be published and for cases where a message has been received by the model. They specify the action taken when either scenario occurs. Conversions link inbound messages to the model and the model to outbound messages. There are two types of conversions:

- Event conversions convert message data to and from event data. See “Event conversions” on page 91.
- DOS conversions convert message data to and from a data object set. See “DOS conversions” on page 94.

Event conversions

Event conversions enable the system to respond to a stimulus (either an event or a message) in a predictable manner. It does not matter what generated the stimulus. It can be external, or it can be internal. Thus, one portion of the system can asynchronously stimulate a different portion of the system in order to handle an occurring situation.

Event conversions for inbound communication use an M2E conversion that takes message data and converts it for use with an event. The event uses the converted information either to initiate a state machine or to effect a transition from one state to another in the state machine. See “Events” on page 82. For outbound communication, an E2M conversion takes event data and places it into a JMS message for an integrated application or for an internal object. A simple event conversion is illustrated below.

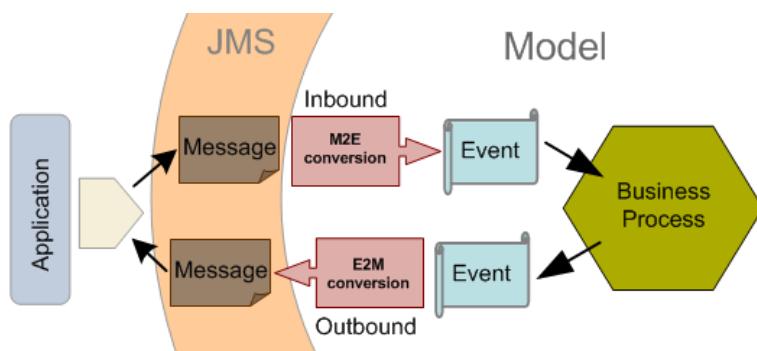


Figure 4-7: Inbound and outbound event conversion

There are two types of event conversion:

- An event to message (E2M) conversion converts an event to a JMS based message that is processed by a separate J2EE processing thread. See “E2M conversion” on page 93.
- A message to event (M2E) conversion fields incoming JMS messages, parses their content into UML events and dispatches each event to the appropriate entity within the system designed to process it. See “M2E conversion” on page 91.

M2E conversion

An M2E conversion fields incoming JMS messages, parses their content into UML events and dispatches each event to the appropriate entity within the system that is designed to process it. You can add an M2E conversion either at the model’s top level or from within the artifacts (messages or events) associated with the conversion.

An M2E conversion describes the message destination (or topic) to which it subscribes, its associated event and the business object with the information for that event. To specify the actual conversion

you may choose between a default conversion or specific attribute conversion. You may also define other information such as an applicability test, pre- and post-conversion logic.

Creating an M2E conversion

To create an M2E conversion:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **Add an M2E** button , or from the **Selection** menu, choose **Conversions > Add M2E Conversion**.
Under the Message to Event Conversions in the tree view, an untitled M2E conversion appears and the M2E editor is displayed in the content view.
3. Define the conversion.
See “Defining an M2E Conversion” on page 92.

Creating an M2E conversion in an event

To create an M2E conversion in an event:

1. In the event editor, in the **Conversions** tab, click **New**.
2. In the **Select Type** dialog, select the **MessageToEventConversion** and click **OK**.
3. In the dialog, enter a name for the conversion and click **OK**.
4. To define the conversion, click **Edit**.
See “Defining an M2E Conversion” on page 92.

Adding an M2E conversion in an event

To set an M2E conversion in an event:

1. In the event editor, in the **Conversions** tab, click **Add**.
2. In the **Select Object(s)** dialog, select the conversion and click **OK**.
3. To define the conversion, click **Edit**.
See “Defining an M2E Conversion” on page 92.

Defining an M2E Conversion

To define an M2E conversion:

1. In the conversion editor, in the **M2E** tab, in the **Name** field, enter a name.
The name must be unique within the model container.
2. If necessary, define an applicability test.
See “Applicability test” on page 101.
3. Set an existing event or add a new event.
 - a. On the **M2E** tab, above the **Event** field, click **New** or **Set**.
 - b. Define the event. See “Creating a standard event” on page 84.
4. In the **Conversion Actions** tab, describe the conversion actions.
See “Conversion actions” on page 98.
5. In the **Destination** tab, create or set the message destination.
See “Destinations” on page 102.

E2M conversion

An event to message (E2M) conversion is an outbound conversion that takes event attributes and converts them to message attributes which are then published to JMS. An E2M conversion converts an event to a JMS message that is processed by a separate J2EE processing thread. The E2M conversion specifies an event and a message, and maps event attributes to message attributes. The E2M conversion publishes to a JMS message given a message destination and rules for producing an instance of the message type based on the event type. It also maps data from a state machine action to a JMS message or another behavior.

Creating an E2M conversion

To create an E2M conversion:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **Add an E2M** button , or from the **Selection** menu, choose **Conversions > Add E2M Conversion**.
An E2M conversion appears in the tree view under Event to Message Conversions, and the E2M editor displays in the content view.
3. Define the conversion.
See “Defining an E2M conversion” on page 93.

Creating an E2M conversion in an event, message or destination

To create an E2M conversion in an event, message or destination:

1. In the editor, in the **Conversions** tab, click **New**.
2. In the **Select Type** dialog, select **EventToMessageConversion** and click **OK**.
3. In the dialog, enter a name for the conversion and click **OK**.
4. To define the conversion, click **Edit**.

See “Defining an E2M conversion” on page 93.

Adding an E2M conversion to an event, message or destination

1. In the editor, in the **Conversions** tab, click **Add**.
2. In the **Select Object(s)** dialog, select the conversion and click **OK**.
3. To define the conversion, click **Edit**.

See “Defining an E2M conversion” on page 93.

Defining an E2M conversion

To define an E2M conversion:

1. In the conversion editor, in the **E2M** tab, in the **Name** field, enter a name.
The name must be unique within the model container.
2. If necessary, define an applicability test.
See “Applicability test” on page 101.
3. Set an existing event or add a new event.
 - a. On the **E2M** tab, above the **Event** field, click **New** or **Set**.
 - b. Define the event. See “Creating a standard event” on page 84.
4. Set the message.

See “[Messages](#)” on page 103.

5. In the **Conversion Actions** tab, describe the conversion actions.

See “[Conversion actions](#)” on page 98.

6. In the **Destination** tab, create or set the message destination.

See “[Destinations](#)” on page 102.

DOS conversions

DOS conversion takes place between the message and a DOS in a data path that includes data transformation. DOS conversions for inbound communication use an M2D conversion that takes message data and populates a DOS. For outbound communication, a D2M conversion takes DOS data and places it into a message.

There are two types of DOS conversions:

- Message to DOS (M2D) conversions that take message data and convert to a data object set. See “[M2D conversion](#)” on page 94.
- DOS to Message (D2M) conversions that takes a data object set and converts it to message data. See “[D2M conversion](#)” on page 96

M2D conversion

The M2D conversion takes data from a message and places it for use in a data object set. There are two cases where you need to define an M2D conversion:

- To convert message data to a DOS for use in a DOS event. A DOS event provides data to a business process that includes a transformation embedded in a simple state. See “[DOS event](#)” on page 85.
- To convert message data to a DOS for use in a transformation. See “[Transformation model](#)” on page 145.

Creating an M2D conversion

To add an M2D conversion:

1. In Cogility Modeler’s tree view, select the model container or package.
2. Click the **Add an M2D** button , or from the **Selection** menu, choose **Conversions > Add M2D Conversion**.

Under the Message to DOS Conversions in the tree view, an untitled M2D conversion appears and the M2D editor displays in the content view.

3. Define the conversion.

For a DOS event, see “[Defining an M2D conversion for a DOS event](#)” on page 95.

For a transformation chain, see “[Defining an M2D conversion for a transformation chain](#)” on page 96.

Creating an M2D conversion in a DOS event or a destination

To create an M2D in a DOS event or destination:

1. In the editor, in the **Conversions** tab, click **New**.
2. In the **Select Type** dialog, select **MessageToDOSConversion** and click **OK**.
3. In the dialog, enter a name for the conversion and click **OK**.
4. To define the conversion, click **Edit**.

See “Defining an M2D conversion for a DOS event” on page 95.

Creating an M2D conversion in a transformation chain

To create an M2D in a transformation chain:

1. In the transformation chain editor, in the **Triggers** tab, above the **Triggering M2D** field, click **New**.
2. In the dialog, enter a name for the conversion and click **OK**.
3. To define the conversion, click **Edit**.

See “Defining an M2D conversion for a transformation chain” on page 96.

Adding an M2D conversion in a DOS event or a destination

To set an M2D in a DOS event or destination:

1. In the editor, in the **Conversions** tab, click **Add**.
2. In the **Select Object(s)** dialog, select the conversion and click **OK**.
3. To define the conversion, click **Edit**.

See “Defining an M2D conversion for a DOS event” on page 95.

Adding an M2D conversion to a transformation chain

To add an M2D to a transformation chain:

1. In the transformation chain editor, in the **Triggers** tab, above the **Triggering M2D** field, click **Add**.
2. In the **Select Object(s)** dialog, select the conversion and click **OK**.
3. To define the conversion, click **Edit**.

See “Defining an M2D conversion for a transformation chain” on page 96.

Defining an M2D conversion for a DOS event

To define an M2D conversion for a DOS event:

1. In the conversion editor, in the **M2D** tab, in the **Name** field, enter a name.
The name must be unique within the model container.
2. If necessary, define an applicability test.
See “Applicability test” on page 101.
3. Specify the data object set.
See “Data object set (DOS)” on page 46.
4. Set an existing DOS event or create a new DOS event.
 - a. On the **M2D** tab, above the **DOS Event** field, click **New** or **Set**.
 - b. If this is a new DOS event, enter a name for the DOS event and click **OK**.
The DOS Event field displays the event.
 - c. If you are setting a DOS event, in the **Select Object(s)** dialog, select the event and click **OK**.
 - d. Click **Edit** to edit the DOS event.
See “DOS event” on page 85.
5. In the **Destination** tab, create or set the message destination.
See “Destinations” on page 102.

6. In the **Conversion Actions** tab, describe the conversion actions.

See “[Conversion actions](#)” on page 98.

Defining an M2D conversion for a transformation chain

To define an M2D conversion for a transformation chain:

1. In the conversion editor, in the **M2D** tab, in the **Name** field, enter a name.
The name must be unique within the model container.
2. If necessary, define an applicability test.
See “[Applicability test](#)” on page 101.
3. Specify the data object set.
See “[Data object set \(DOS\)](#)” on page 46.
4. In the **Conversion Actions** tab, describe the conversion actions.
See “[Conversion actions](#)” on page 98.
5. In the **M2D Triggered Chain** tab, set or create a transformation chain.
See “[Transformation Chain](#)” on page 163.

D2M conversion

The output from a transformation populates a DOS. To move the data from the DOS to a message, you assign a DOS to message (D2M) conversion to the transformation chain or the simple state where the transformation is used.

Creating a D2M conversion

To create a D2M conversion:

1. In Cogility Modeler’s tree view, select the model container or package.
2. Click the **Add an D2M** button , or from the **Selection** menu, choose **Conversions > Add D2M Conversion**.
In the tree view, under the DOS to Message Conversions, an untitled D2M conversion appears and the D2M editor displays in the content view.
3. Define the D2M conversion.
See “[Defining a D2M conversion](#)” on page 97.

Creating a D2M conversion in a simple state

To create a D2M conversion in a simple state:

1. In the simple state editor, in the **Transformation** tab, above the **D2M Conversion** field, click **New**.
2. In the dialog, enter a name for the conversion and click **OK**.
3. To define the conversion, click **Edit**.
See “[Defining a D2M conversion](#)” on page 97.

Creating a D2M conversion in a transformation chain

To create a D2M conversion in a transformation chain:

1. In the transformation chain editor, in the **Triggers** tab, above the **Triggered D2M** field, click **New**.

2. In the dialog, enter a name for the conversion and click **OK**.
3. To define the conversion, click **Edit**.
See “Defining a D2M conversion” on page 97.

Creating a D2M conversion in a destination

To create a D2M conversion in a destination:

1. In the destination editor, in the **Destination** tab, above the **Conversion** field, click **New**.
2. In the dialog, enter a name for the conversion and click **OK**.
3. To define the conversion, click **Edit**.
See “Defining a D2M conversion” on page 97.

Adding a D2M conversion in a transformation chain

To set a D2M conversion in a transformation chain:

1. In the transformation chain editor, in the **Triggers** tab, above the **Triggered D2M** field, click **Add**.
2. In the **Select Object(s)** dialog, select the conversion and click **OK**.
3. To define the conversion, click **Edit**.
See “Defining a D2M conversion” on page 97.

Adding a D2M conversion in a simple state

To set a D2M conversion in a simple state:

1. In the simple state editor, in the **Triggers** tab, above the **Triggered D2M** field, click **Add**.
2. In the **Select Object(s)** dialog, select the conversion and click **OK**.
3. To define the conversion, click **Edit**.
See “Defining a D2M conversion” on page 97.

Adding a D2M conversion in a destination

To set a D2M conversion in a destination:

1. In the destination editor, in the **Conversions** tab, click **Add**.
2. In the **Select Object(s)** dialog, select the conversion and click **OK**.
3. To define the conversion, click **Edit**.
See “Defining a D2M conversion” on page 97.

Defining a D2M conversion

To define a D2M conversion:

1. In the conversion editor, in the **D2M** tab, in the **Name** field, enter a name.
The name must be unique within the model container.
2. If necessary, define an applicability test.
See “Applicability test” on page 101.
3. Specify the data object set.
See “Data object set (DOS)” on page 46.

4. Set the message.
See “[Messages](#)” on page 103.
5. In the **Conversion Actions** tab, describe the conversion actions.
See “[Conversion actions](#)” on page 98.
6. In the **Destination** tab, create or set the message destination.
See “[Destinations](#)” on page 102.
7. In the **D2M Triggered By** tab, set or create the transformation chain.
See “[Transformation Chain](#)” on page 163. Note the Simple State field lets you edit or remove a simple state designated to work with this conversion, but you cannot create a or add a simple state here. You must create the simple state within a state machine first. See “[State machine](#)” on page 112.

Conversion actions

Sometimes you do not have control over how an integrated system’s messages are formatted; it is incumbent upon you to describe the conversion so that your model’s event can use the message data. If the message format allows a direct conversion from message to event or event to message, you specify a [Default conversion](#). If the conversion cannot be a default conversion, you can define [Pre- and post-conversion actions](#) to assist with the conversion. Note that you can specify a default conversion for an Event conversion (see “[Event conversions](#)” on page 91), but for DOS conversions (see “[DOS conversions](#)” on page 94) the conversion is always a default conversion.

Also, in M2E and M2D conversions, you must specify a [Business object locator](#) to specify the target business object. In M2D conversions, you must also specify a [DOS populator](#) that fills the attributes in the target DOS with the message attribute values.

Pre- and post-conversion actions

For added flexibility in converting data in JMS messages, you can define pre- and post-conversion actions. These pre- and post-conversion actions are available in any conversion artifact.

Defining pre- or post-conversion actions

To define pre- or post-conversion logic:

1. In the conversion editor, under the **M2E**, **E2M**, **M2D** or **D2M** tab.
2. In the **Pre Conversion** or **Post Conversion** field, enter the appropriate actions.
See the guide, *Using Actions in Cogility Studio* for assistance.

Default conversion

Usually, an M2E or E2M conversion (see “[Event conversions](#)” on page 91) may be specified as a default conversion. DOS conversions are always default conversions (see “[DOS conversions](#)” on page 94). When you specify a default conversion, the information in the message is assumed to map directly to the event. That is, the message format should consist of key/value pairs that are consistent with the event attributes. All event conversions are initially set to be default conversions.

Removing the default conversion specification

To remove the default conversion specification:

1. In the event conversion editor, click the **Conversions Actions** tab.
2. Uncheck the **Default Conversion** check box.

Business object locator

An inbound message contains data for a specific business object. The business object locator in an M2E or M2D conversion specifies the business object. That business object's class behavior is associated with the event specified by the conversion. The conversion must locate the business object at run time so that the event can run the behavior of its class.

Recall that events are transient objects, as explained under “[Events](#)” on page 82. The event attributes are passed through to the business process associated with the persistent business object. The business object locator finds that business object based on the attribute values in the message and the conversion.

For the creation of a new business object, the business object locator specifies the factory object that creates the business objects (as in the case of a new Customer creation, shown in the picture below). Otherwise, if the object does not exist, the business object locator returns null and the conversion fails. For more about factories, see “[Factories](#)” on page 48.

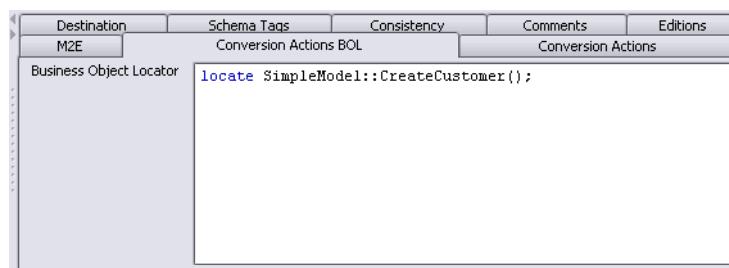


Figure 4-8: Business object locator

See the guide, *Using Actions in Cogility Studio* for the various ways you can locate instances.

Creating a business object locator

To create a business object locator:

1. In the M2E or M2D conversion editor, click the **Conversion Actions BOL** tab.
2. In the **Business Object Locator** field, enter the actions that locate the business object that receives the event.

DOS populator

For M2D conversions associated with events, you must specify a DOS populator along with the business object locator (see “[Business object locator](#)” on page 99). The DOS populator describes the message attributes to the DOS used in an event. In the example below, the business object locator locates the CreateCustomer() factory to create a new Customer. The DOS populator assigns the new

Customer attribute values to the DOS. For more information on defining a DOS populator, see the guide *Using Actions in Cogility Studio*.

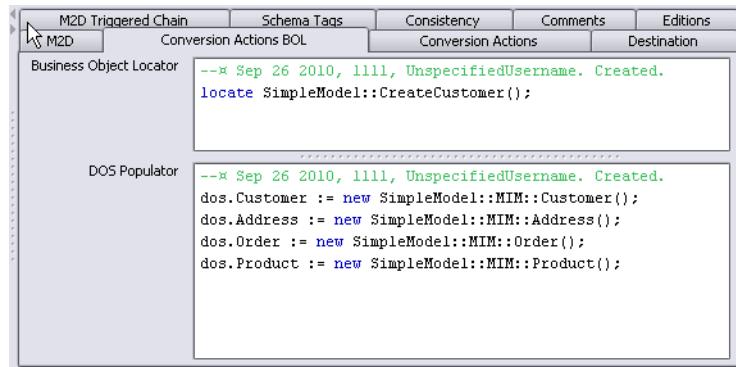


Figure 4-9: DOS populator

You also need a DOS populator for an M2D conversion associated with a transformation chain, though you do not have to specify the business object locator. For either an opaque transformation map or a transformation map, you must also specify a DOS.

In the DOS populator for a transformation, the actions use two reserved keywords, `input` and `output` as known variables to specify the input DOS and the output DOS. The DOS populator locates an object of the type specified in the input DOS, and populates the output DOS with an object of the same type using the data from the transformation. The example below shows a completed opaque transformation actions definition:

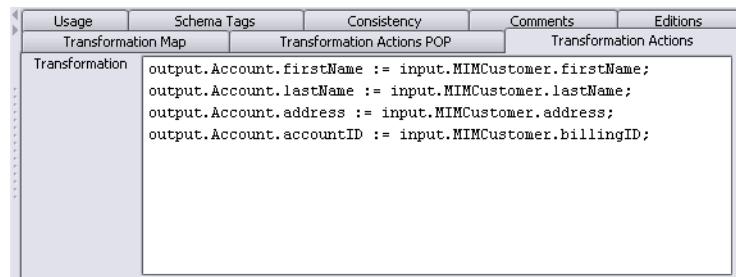


Figure 4-10: Opaque transformation map definition

The DOS Populator declares a variable, `acct`, of the Account type in the BillingDIM. The action then looks for an instance of an Account with an accountID that matches the billingID of the input DOS MIMCustomer instance and assigns it to the `acct` variable. If an existing instance cannot be found, a new instance is created. The value of the variable `acct` is then assigned to the output DOS Account class. The transformation actions assign the output DOS Account class features (attributes) values to the input DOS MIMCustomer class features (attributes). For more information see “[Transformation model](#)” on page 145.

Defining a DOS populator

To define a DOS populator:

1. In the M2D conversion editor, click the **Conversion Actions BOL** tab, or in the transformation editor, click the **Transformation Actions** tab.
2. In the **DOS Populator** field or **Transformations** field, enter the required actions.

Applicability test

A message may contain data bound for different events and data object sets. Likewise data from events and data object sets may be defined for different messages. You use an applicability test to determine which messages from which events and DOS, and which events and DOS from which messages. Each conversion's applicability test describes the target for specified data according to the logic of an action.

Just as a single event can generate many types of system notification messages (albeit on the same destination), a single inbound message can cause multiple trigger events to be generated, if this is warranted. The applicability tests govern and control this process as dictated by the business logic.

The applicability tests are evaluated during the model's execution as a composite application. Any conversion with a test that evaluates to true is invoked. That is, if the conversion matches the destination and passes its applicability test, it is used to create the event. If an applicability test is not specified (null), it evaluates to true by default.

In the example below, if the incoming message specifies that the customerLevel attribute is Gold the OE2MIM_M2E conversion's applicability test evaluates to true and the OE_Event is generated.

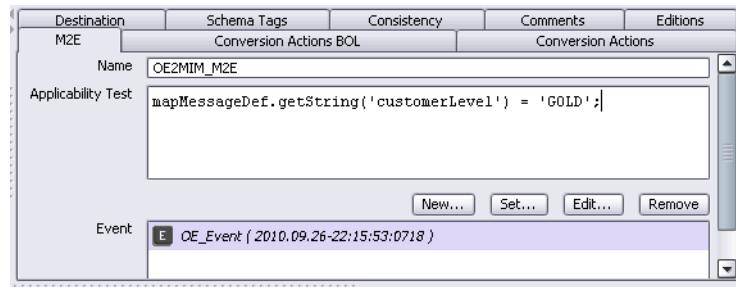


Figure 4-11: Applicability test

The `jmsMapMessage` bound variable retrieves values from messages. See the guide, *Using Actions in Cogility Studio*.

The `eventDef` bound variable refers to the event object. See the guide, *Using Actions in Cogility Studio*.

Defining an applicability test

To define an applicability test:

1. In the conversion editor, click the **M2E**, **E2M**, **M2D** or **D2M** tab.
2. In the **Applicability Test** field, enter the actions to describe the test.
3. In the **Event** field, add an event.
 - a. To add a new event:
 - Above the Event field, click New.
 - Add a name for the event in the dialog box./.../.
 - Click OK.
 - b. To add an existing event:
 - Above the Event field, click Set.
 - Select the Event from the list.
 - Click OK.

Destinations

The destination object designates the JMS message destination to which a conversion subscribes or publishes. For the JMS publish-and-subscribe method utilized in Cogility Studio, this is the topic destination type.

Note: If your model has an E2M and an M2E conversion connected to the same destination, it can send messages to itself and often does.

Creating a destination

To create a destination:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **Plus** button  and select **Add Destination**, or from the **Selection** menu, choose **Conversions > Add Destination**.
Under Destinations in the tree view, an untitled destination displays, in the content view the Destination Editor is displayed.
3. Define the destination.
See “Defining a destination” on page 102.

Creating a destination in a conversion

To create a destination in a conversion:

1. In the conversion editor, under the **Destination** tab, above the **Destination** field, click **New**.
2. Enter the name for the destination and click **OK**.
The name must be unique within the model container.
3. Define the destination.
See “Defining a destination” on page 102.

Setting a destination in a conversion

To set a destination in a conversion:

1. In the conversion editor, under the **Destination** tab, above the **Destination** field, click **Set**.
A list of the available destinations displays.
2. Select the destination and click **OK**.
3. Define the destination, if necessary.
See “Defining a destination” on page 102.

Defining a destination

To define a destination:

1. In the destination editor, in the **Destination** tab, in the **Name** field, enter a name for the destination.
The name must be unique within the model container.
2. Specify the owning conversion.
See “Conversions” on page 91. If you are defining the destination from within a conversion, this field fills automatically.

Messages

Message artifacts in Cogility Modeler carry information to, from and within the model for both web services and Java Messaging Service (JMS). For more information about web services, see “[Web Services Model](#)” on page 201. This section describes messages and destinations (or topics) that identify the message. Message artifacts may be used as superclass objects for other message artifacts. The subclass messages inherit the message attributes of the superclass.

Messages are represented in Cogility Modeler’s tree view with a message icon:  . If the message has an associated operation, the message class or subclass where that operation is defined with a method has an icon that includes an O in the bottom right corner: .

JMS messages

Most message traffic occurs within the model. Messages carry information used in asynchronous processes that branch from synchronous processes.

A message contains several parts: a message header, message properties, and a message payload. The message header contains one or more destinations. For the JMS publish-and-subscribe method utilized in Cogility Studio, this destination type is known as a topic. When the external system publishes a message, it publishes the message to the destination.

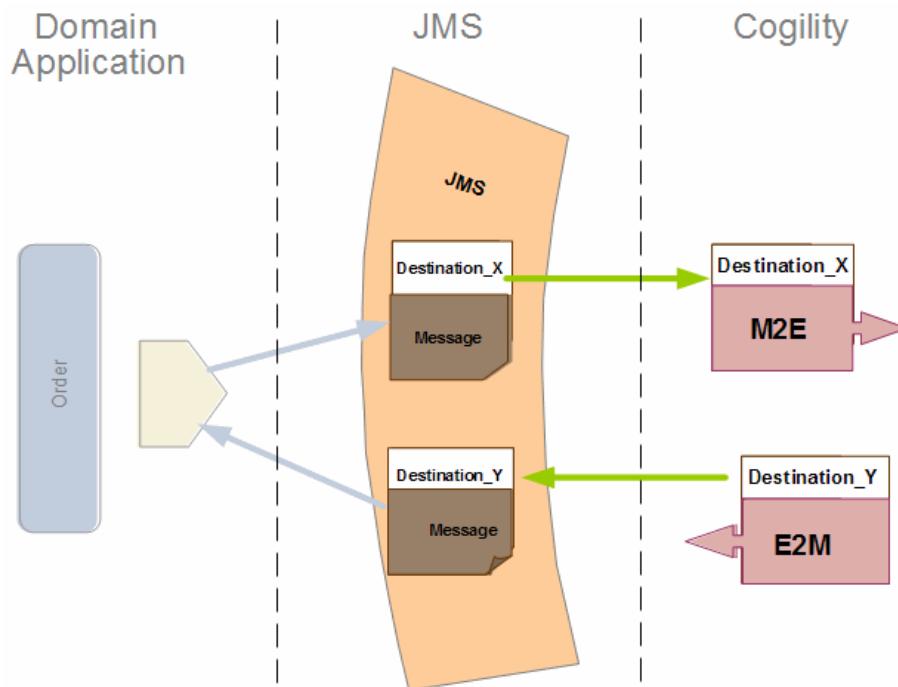


Figure 4-12: Message transmission between an external application and the model

When you define an inbound M2E conversion, you specify a destination as part of the definition. The destination must match the destination of the specified incoming message. When the model is pushed, message-driven beans are created which listen for JMS messages published on the destination specified in the conversion. During execution, when a message is received the information in the message is passed along to the event associated with the message by the M2E. For more information about M2E conversions, see “[M2E conversion](#)” on page 91.

When you define an outbound E2M conversion, you select or define a destination, a message payload and an attribute conversion from the event to the message. As with the M2E conversion, the

destination must match the destination subscribed to by the message recipient. When the model is pushed, a message is assembled using this information and published to the destination designated by the topic. The external system that subscribes to this destination receives the message and its information. For more information about E2M conversions, see “[E2M conversion](#)” on page 93.

Message objects in Cogility Studio model JMS messages for publication. You must define a message object for an E2M or D2M conversion if the recipient of the information requires it in a format different from that of the generating event. In such a case, specific conversion logic must be added to the conversion. If the recipient of the message is within the model, then the form may be left as “default” in almost all cases (unless you are simulating multiple message sends in different forms from the same physical event). See “[Conversions](#)” on page 91.

Web service messages

When the model receives external data via an inbound web service, the web service must complete its synchronous process by responding to the caller. The model decodes the SOAP request from the caller places it in an argument message artifact. From the model, a result message artifact encodes a SOAP response.

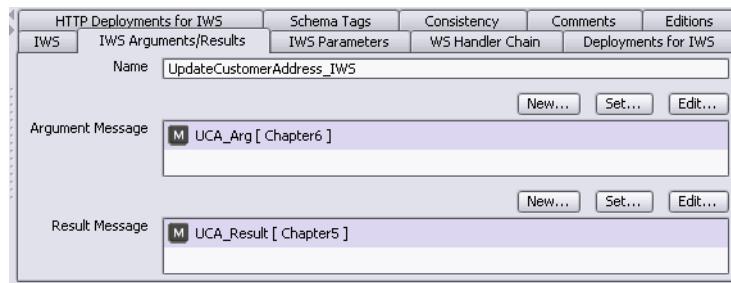


Figure 4-13: Web service message arguments

The argument message defines what the web service expects in terms of message attributes. These represent elements of the XML document and may contain data to be processed by the inbound web service.

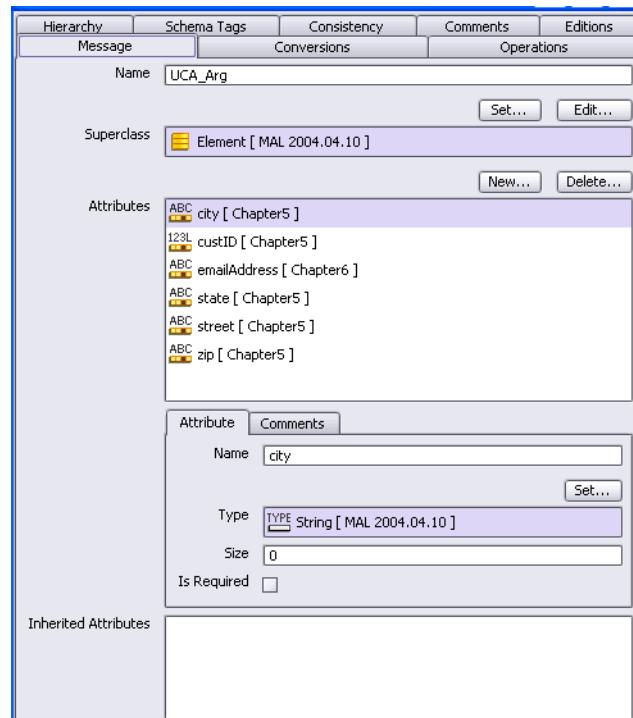


Figure 4-14: Web service argument parameter definition

In the example above, the web service expects city, emailAddress, state, street, and zip; all of these are String types as signified by the icon next to each of the attribute names. It also expects a custID which is a long type.

Because the argument message reuses the Cogility Modeler message template, it includes a Message Conversions tab. But web services do not make use of the JMS messaging layer and do not require message conversions. You can ignore this tab.

The result message carries information to the requestor. The message payload attributes represent elements or attributes of the XML document, depending on the decoding style, and may contain data to be returned to the process which invoked the inbound web service.

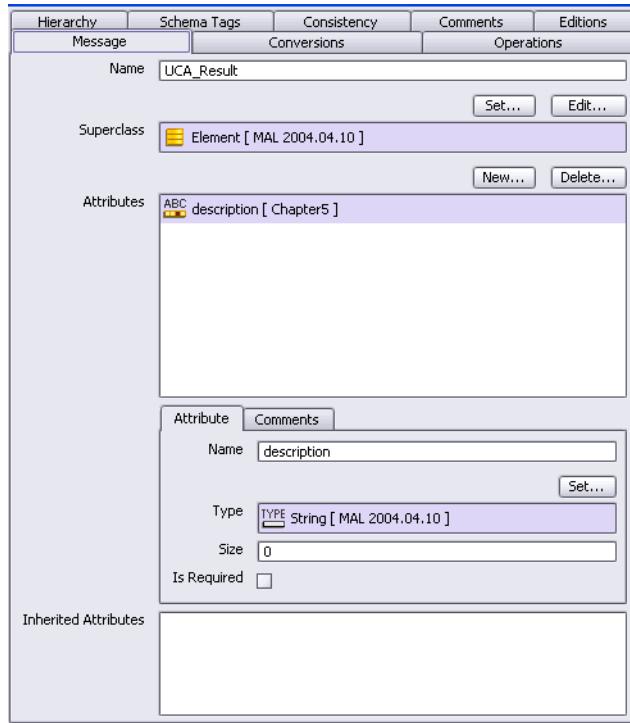


Figure 4-15: Web service result message definition

In the example above the returned attribute, description, holds a String for a message transmitted to the user. The web service logic (see “[Web service action](#)” on page 203) defines the value of the description attribute.

Because the argument message reuses the Cogility Modeler message template, it includes a Message Conversions tab, as illustrated above. But web services do not make use of the JMS messaging layer and do not require message conversions. You can ignore this tab.

You can also create superclasses for messages (see “[Messages](#)” on page 103). However, when you select a message for use as superclass, the selection dialog will list all of the message artifacts defined for the model, including those message objects used in web services. When naming messages, use a naming scheme that identifies message superclasses and messages for different purposes.

Creating a message

To create a message:

1. In Cogility Modeler’s tree view, select the model container or package.
2. Click the **Add a Message** button M, or from the **Selection** menu, choose **Conversions > Add Message**.

In the tree view, under Messages, an untitled message displays, and the content view displays its message editor window.

3. Define the message.

See “[Defining a message](#)” on page 107.

Creating a message in a conversion

To create a message in a conversion:

1. In the conversion editor, in the E2M or D2M tab, above the **Message** field, click **New**.
2. In the dialog, enter a name for the message and click **OK**.
3. To define the message, click **Edit**.

See “[Defining a message](#)” on page 107.

Creating a message in a web service

To create a message in a web service:

1. In the web service editor, in the **Web Service** tab, above either the **Argument Message** or **Result Message** field, click **New**.
2. In the dialog, enter a name for the message and click **OK**.
3. To define the message, click **Edit**.

See “[Defining a message](#)” on page 107.

Setting a message in a conversion

To set a message in a conversion:

1. In the conversion editor, in the E2M or D2M tab, above the **Message** field, click **Set**.
2. In the **Select Object(s)** dialog, select the message and click **OK**.
3. To define the message, click **Edit**.

See “[Defining a message](#)” on page 107.

Setting a message in a web service

To set a message in a web service:

1. In the web service editor, in the **Web Service** tab, above either the **Argument Message** or **Result Message** field, click **Set**.
2. In the **Select Object(s)** dialog, select the message and click **OK**.
3. To define the message, click **Edit**.

See “[Defining a message](#)” on page 107.

Defining a message

To define a message:

1. In the **Message** tab of the message editor, enter the name for the message in the **Name** field.
2. If necessary, specify a superclass for this message.
If you specify a superclass, the inherited attributes are shown in the **Inherited Attributes** field. See “[Superclass](#)” on page 33.
3. Above the **Attributes** field click **New**.
 - a. In the dialog window that displays, for **Name**, enter a name.
 - b. For **Type**, select the type from the drop-down menu and click **OK**.The attribute is listed in the **Attributes** field, and the **Attribute** tab is displayed.

4. If necessary, in the **size** field, set the size for String types.

This applies only to String types. Enter a character limit, as a positive integer. During run time, values with a greater number of characters are truncated to the limit you set. The default is 256 characters.

5. If the attribute is required, click the **Is Required** checkbox.

6. If you wish to add a comment to the attribute, select the **Comments** tab, then select **New**.

7. Enter a name for the comment, and click **OK**.

The comment name is listed in the Comment field, and the Comment tab is displayed.

8. Enter the text for the comment in the **Body** field.

9. In the **Operations** tab, define the message operation.

For messages, the operation may be inherited from message superclass artifacts.

See “[Operations](#)” on page 41

10. In the **Conversion** tab, specify the message conversion artifacts.

See “[Conversions](#)” on page 91.

Message definitions

To use JMS messages to communicate to and from your model, the messages must be available to the application sending the messages. Cogility Modeler makes your model’s messages available in a .msg file.

To test your model without an external application, you use Cogility Message Traffic Viewer to publish messages on the destinations for which your composite application is listening. These are listed in Cogility Modeler’s tree view under Destinations. Cogility Message Traffic Viewer reads from the .msg file to publish the messages. See “[Message Traffic Viewer](#)” on page 129 of the guide, *Model Deployment & Execution in Cogility Studio*.

Exporting message definitions

To export message definitions:

1. In Cogility Modeler’s tree view, select the model container, right-click and select **Actions > Output MessageEditor Messages**.
2. In the navigation dialog that comes up, choose a location for the output file and click **Output in Directory**.
3. Note the location where the file is generated and click **OK**.



5

Behavior model

A business process is a collection of related, structured activities or tasks that serve a particular goal for a particular customer or customers. For example, when a customer places an order at a web site, the order must be communicated to several systems, including a billing system, an inventory system and an order fulfillment system, each of which is responsible for completing a portion of the overall process with its own sub-process. The end product of this business process is the delivery of an item from inventory to the customer.

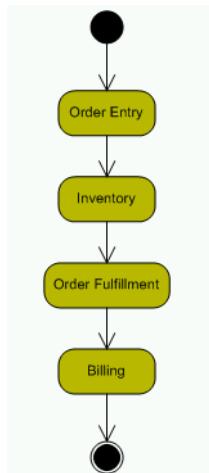


Figure 5-1: Example business process

A UML (Unified Modeling Language) state diagram describing this business process might look like the one illustrated above. This diagram describes the states of an object in a process.

When a customer creates an order, its corresponding Order object is in the Order Entry state. During that state, the customer's credit card might be validated before the Order can progress to the Inventory state. Once sufficient inventory is allocated, the Order moves to the Order Fulfillment state. Once the order has shipped, the Order proceeds to the Billing state, the last state in the process.

UML provides a standard for describing a business process and the systems involved in that process, just like it provides a standard for describing how a computer program runs. For years software engineers and systems architects have used UML to "model" elaborate object oriented systems. UML is the *lingua franca* of systems engineering. With Cogility Studio, you can use UML for business process and systems integration as well.

In Cogility Studio, the UML state activity diagram that models a business process actually describes a fully interactive *controller* of a business process called a state machine. A state machine functions on two levels: first it describes a business process, second it controls a business process.

A state activity diagram is the starting point for creating a state machine wherein a business analyst or someone with a high level view might draw the process as a state diagram in Cogility Modeler. Later, an architect or engineer responsible for implementing the model might continue working with the state diagram, adding the logic that controls the business process, thus making it a state machine.

In Cogility Studio, business processes describe the states of (or the behavior for) an enterprise object. An enterprise object encapsulates data effective in several applications over several domains. The “[Example business process](#)” on page 109 describes an Order object as it moves from Order Entry to Billing. Enterprise objects are modeled in the master information model as MIM classes.

The MIM class, MIMOrder from the example above would have attributes used by each of the Order Entry, Inventory, Order Fulfillment and Billing systems. As such it is a bridge to each of the distinct Order, Requisition, Shipment and Invoice classes on each system. The MIMOrder class holds the information communicated to all integrated systems. Notice in the figure below that the MIMOrder class has a CustomerName and ID attributes for all systems.

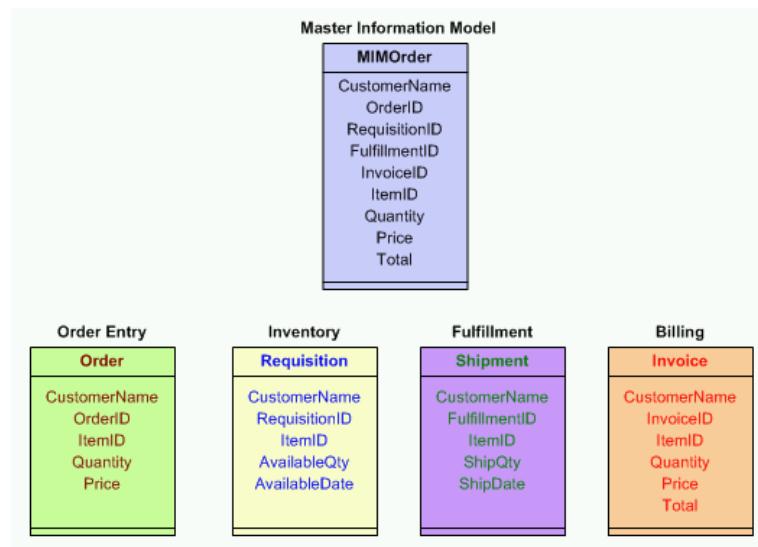


Figure 5-2: The Master Information Model class

The MIM class describes the data that is accessed during the business process. The MIM class' behavior describes how the data is used in each of the systems. The state machine models this behavior.

In the “[Example business process](#)” on page 109, the state machine follows the behavior of an Order class. Each of the systems involved in processing the order is shown as a state. Each state contains logic that describes what to do with an Order. Between each state, shown as arrows, are transitions. These, also, contain logic controlling the process flow.

A MIM class may have more than one behavior and multiple state machines. For example, the MIMOrder class might also have a state machine that communicates order information to a Promotions system and then to a Data Warehouse system. This behavior would happen in parallel with the previous behavior.

Factory classes (described in “[Factories](#)” on page 48) may also have behaviors, and one or more state machines. An example of this is in “[Batch processing](#)” on page 132.

Transactions

All database changes are grouped within Transactions whose boundaries are implied by the structure of the model. Transactions may take place within a state or during a transition. Both types of transactions are atomic (that is, uninterruptible). If an error occurs during the transaction, the entire transaction must be rolled back and restarted. If the transaction involves batch processing, the entire batch of records must be resubmitted. See “[Batch processing](#)” on page [132](#) for more information.

State transactions occur within the state’s do activity. The entry action of the very first state is performed with the transaction in which the message that starts the new state machine run-time instance is received. At the end of the entry action, there an internal message is published to create the do action message, which ensures that the do action happens in a separate transaction. Thus, for an eventless transition, it is possible that the do action, the exit action , the transition action and the next state’s entry action all happen in the same transaction. But every do action always starts a new transaction.

Transition transactions include the exit action of a state, the effect action of the transition, and the entry action of the next state. The boundaries of this type of transaction are the begining of the exit action and the ending of the next entry action. Note that the transition gets its own transaction only if it is triggered by an event. An untriggered transition is done as part of the do activity transaction which preceded it.

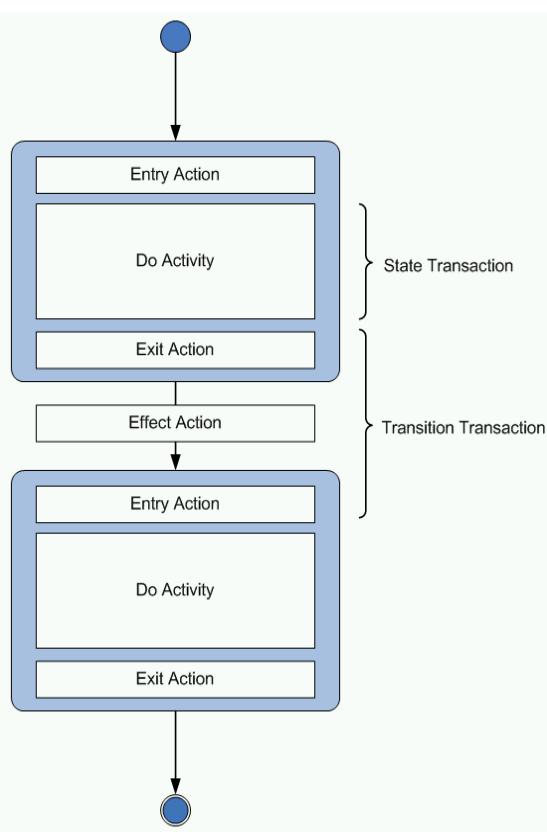


Figure 5-3: State and transition transactions

Cogility Manager operates by default in automatic transaction mode. Every JMS message that enters the system creates a database transaction. The JMS message-induced transaction may also be shared. For example, if the message is a trigger, then the message's transaction is also used to do the

transition directed by the trigger. If the transition errors out, the message's transaction is rolled back, and the message is not considered successfully delivered. Also, every web service request that enters the system creates a database transaction.

Cogility Studio uses transacted JMS messages for all message traffic. The receipt of a JMS message occurs in the context of a Transaction which has guaranteed delivery. If an error occurs and the Transaction is rolled back, the message will be re-delivered. The receipt of a JMS message (or call of a Cogility IWS) starts a paired Database and JMS transaction, which are committed or rolled back together. No matter what publish requests may be done by the model, they are all thrown away if the underlying JMS session is rolled back. As noted previously, untriggered transitions don't get a separate transaction. When a do activity completes, an untriggered transition is taken (if available) at the end of the do activity's transaction. If the transition fails, the do activity (and receipt of the do activity request event) is rolled back.

Because state machines can be started only by a JMS message received by Cogility, they are automatically transactional. Since every event-based state transition must be triggered by a message, these types of state transitions are transactional as well. The messages are based on Java Message-driven beans (MDBs). MDBs are instantiated into pools to which events are distributed. The receipt of a message by an (already instantiated) MDB starts the transaction. The successful completion of processing following the receipt of a message commits a transaction to the database. One MDB typically fields many JMS messages after being instantiated, and there is a transaction for each fielded message. Cogility Studio internally sends a message to notify an object to perform the do activity that is required by its transition to a new state. Likewise, the object's do activity completes with another notification message to transition to the next state. Therefore, the transaction boundary between a state transaction and a transition transaction is determined by this message notification.

These boundaries place some limitation on using transactions within a state. JMS messages are sent as the transaction commits. This may be when the do activity completes (if waiting on a trigger). The state machine remains in the given state until the trigger is received. Once a transaction successfully commits, it may not be rolled back. But, for example, if an exception occurs in an EntryAction of the subsequent state, even though the previous state is "left," the transaction will be rolled back and the previous state returned to active status.

You can also execute a transaction with a web service operation. A web service transaction is synchronous, one that performs some operation and returns some value in one round trip. However, while IWSes are "synchronous," calling an OWS or callXIWS, may kick off changes elsewhere that cannot be rolled back if some later error causes the local Transaction to be rolled back, because OWSes and XIWS calls work in the context of their own Transactions. For more information about using web service operations in state machines, see "[Inbound web service invocation](#)" on page 209 or "[Outbound web service invocation](#)" on page 219.

State machine

A state machine both describes a business process and controls a business process for an object. The object's class definition in Cogility Modeler includes the class behavior modeled as a state machine artifact. The state machine describes each state of the object as it proceeds through a business process. These states include actions that describe the object's interaction with integrated systems via model artifacts such as events and messages, as described in "[Message modeling](#)" on page 81. The states may also describe data transformations with transformation artifacts, as described in "[Transformation model](#)" on page 145. The state machine also describes the transitions between states, and these transitions may also be controlled by events and actions.

Before any actual controller logic is added to the state machine, it is simply a UML state activity diagram. It may serve as a high-level blueprint to guide further development. The diagram's components (states and transitions) can hold comments with instructions to developers. The

developers then augment the diagram, adding the logic that controls the business process, thus making it a state machine. Several features distinguish a state machine from a state activity diagram. These include actions, embedded transformations, event triggers and guard conditions. These artifacts give the state machine the ability to interact with the integrated systems, and control the logic of a business process.

An event triggers a state machine. The event triggered a message describing the event, then the enterprise model converted the message to an event object and started the business process in the state machine.

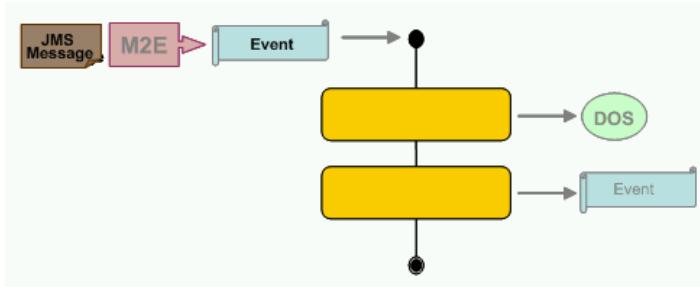


Figure 5-4: An event triggers a state machine

The state machine may require some object data for transformation during the business process. In this case, a DOM event encapsulates the object data and triggers the state machine. The state machine may produce an event, a DOS or both as outputs.

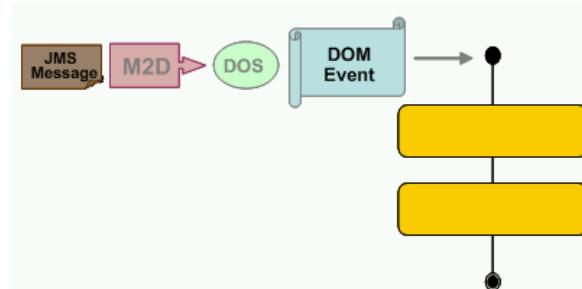
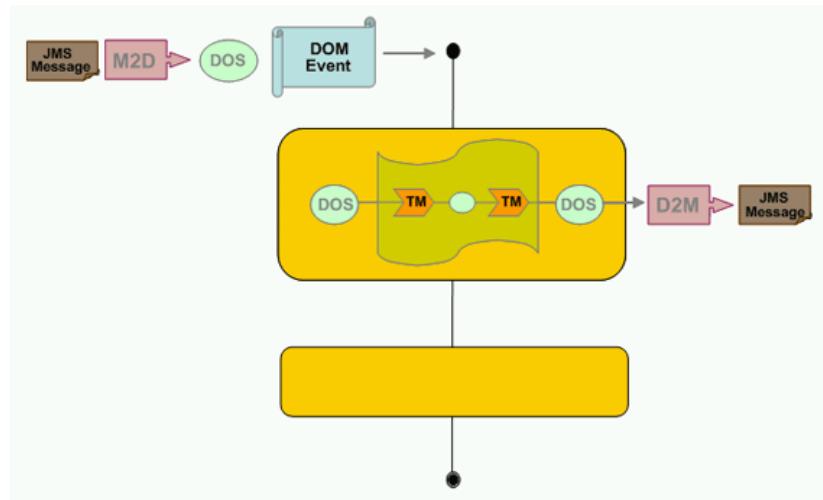


Figure 5-5: A DOM event triggers a state machine

The DOS event gets its data from a DOS. Notice that the conversion object in this case is an M2D, not an M2E as in the first example. The DOS is needed to perform the transformations embedded in the state machine.

When a business process includes data transformations, the transformation objects are said to be embedded in the state machine.



During the business process, embedded transformations may produce object data for conversion and communication to the domain applications.

State machines are located in the tree view under the behaviors branch of a class.

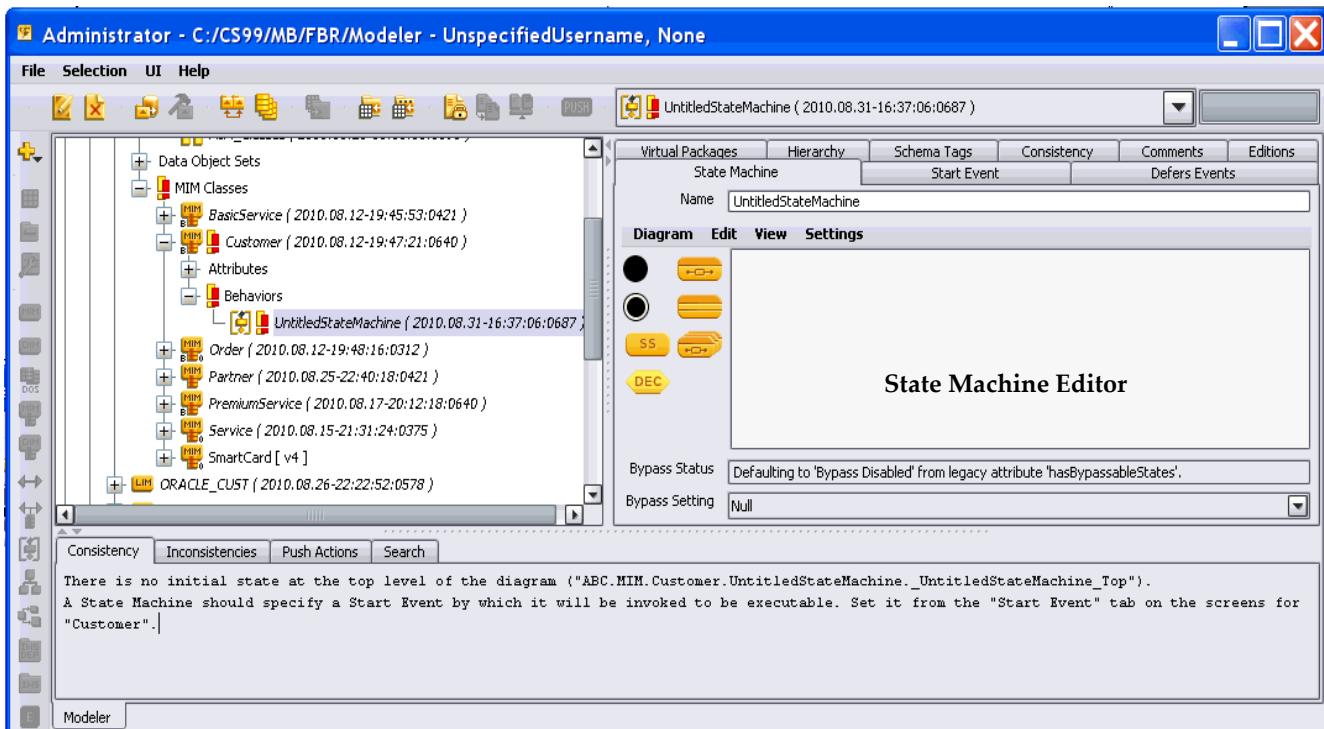


Figure 5-6: The state machine editor

State machines are one of several artifact types that may be pushed into execution separately from the model. See “[Single artifact push](#)” on page 30 of the guide, *Model Deployment & Execution in Cogility Studio*.

State machine editor features

While editing state machines, you can use several features to make design the activity diagram easier. The Edit menu allows you to:

- Select All - this option allows you to select all items in the activity diagram
- Select All Nodes - this option allows you to select only the nodes in the activity diagram
- Select All Connections - this option allows you to select only the connections in the activity diagram

The View menu allows you to change the size of the font used in the activity diagram.

The Diagram menu provides the following functions (you can also access these functions by right clicking on the diagram background):

- Print functions:
 - Page setup - opens the page setup dialog box so that you can define the print options for the diagram.
 - Print - opens the print dialog box so that you can print the diagram based on the print options you selected page setup dialog.
- Set New Start Event - Opens the Start Event Editor. See “[Creating a start event](#)” on page 131.
- Export SVG - Exports the behavior definition to a .svg file.

If you right click on a state machine object, you can:

- Open the state editor.
- Delete the node.
- Connect the node to itself - creating a transition. Click on the new transition to open the transition editor. See “[Creating a circular transition](#)” on page 126.

Creating a state machine

To create a state machine:

1. In Cogility Modeler’s tree view, select the class for which you want to add a state machine.
2. Click the Add a Behavior button .

A new state machine artifact appears in the tree view under the MIM class, and the state machine editor appears in the content view. The new behavior shows in the tree view with the default name *UntitledStateMachine*.

Once you have added a behavior, you can define the behavior artifacts that are needed to invoke/start that behavior by publishing/receiving a message. There are two ways you can define do this:

- You can add all the required artifacts for the state machine automatically. For instructions, see “[Defining behavior artifacts automatically](#)” on page 115.
- You can add each of the required artifacts for the state machine manually. For instructions, see “[Defining behavior artifacts manually](#)” on page 116.

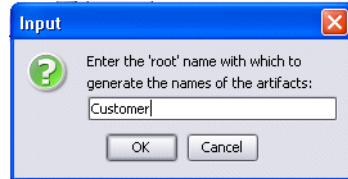
Defining behavior artifacts automatically

When defining behavior artifacts automatically, you are prompted to select the "root" name that will be used to create the names of the elements that are created. The default is to use the name of the StateMachine, but you can enter another name. The "root" name is used to create a Destination, Event, Message, M2E, and E2M with default names of root_Topic, root_Event, root_Message, root_M2E and rootE2M, respectively. The Destination and Event are connected to the M2E and E2M.

The Message is connected to the E2M. And, the Event is connected as the "Start Event" for the StateMachine.

To define behavior artifacts automatically:

1. In Cogility Modeler's tree view, select the newly added behavior for which you want to add behavior artifacts.
2. Right click on the behavior and select **Actions>Create Behavior Invocation Artifacts** from the menu.
3. Enter a root name in the Root Name dialog and click **OK**.



The root name is used to create the names of each of the behavior elements: Destination, Event, Message, E2M, and M2E.

4. Click **OK** in the creation confirmation dialog.



In this example the root name is: Customer.

5. After the artifacts are created, you can define each artifact. See the appropriate section:
 - “Destinations” on page 102
 - “Events” on page 82
 - “Messages” on page 103
 - “E2M conversion” on page 93
 - “M2E conversion” on page 91

Defining behavior artifacts manually

Defining a state machine

To define a state machine:

1. In the state machine editor, under the **State Machine** tab, in the **Name** field, enter a name to describe the behavior.
2. Specify a start event for the class, and assign the start event to this behavior.
See “[Start event](#)” on page 130.
3. Place the states in the state machine.
See “[States](#)” on page 117.
4. Create transitions between the states.
See “[Transitions](#)” on page 125.

States

A state activity diagram consists of states connected by transitions. A state describes an object at a particular point in a process. State diagrams in Cogility Modeler have six types of states, described as follows:

- **Initial state** - see “Initial state” on page 118.
- **Simple state** - see “Simple state” on page 118.
- **Decision state** - see “Decision state” on page 118.
- **Sequential composite state** - see “Sequential composite state” on page 118.
- **Concurrent composite state** - see “Concurrent composite state” on page 120.
- **Replicate state** - see “Replicate state” on page 121.
- **Final state** - see “Final state” on page 122.

The state types are illustrated below.

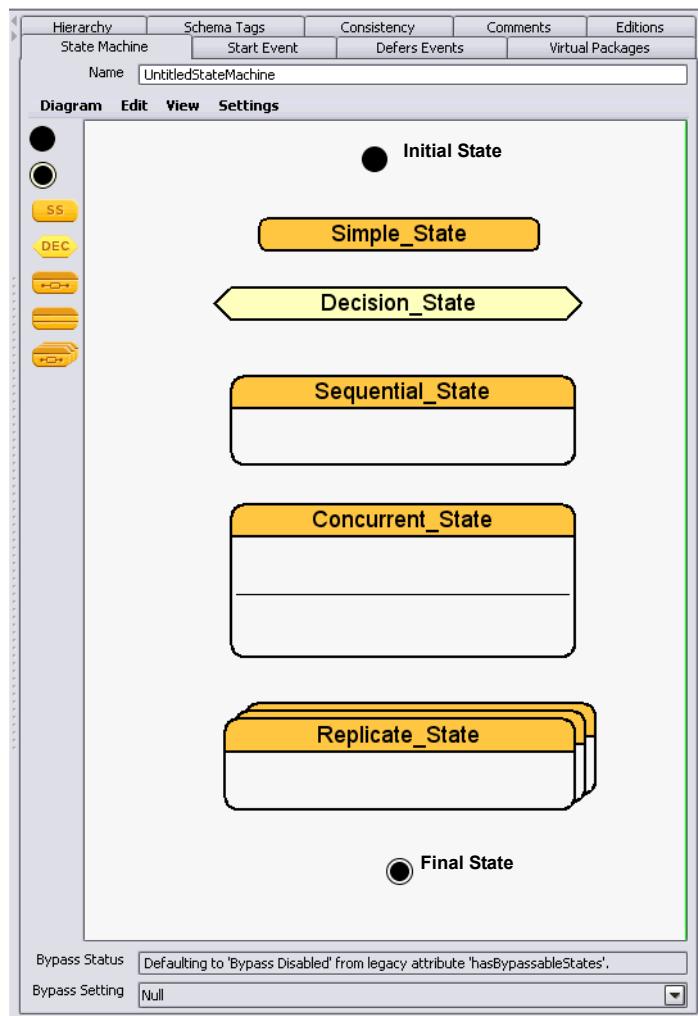


Figure 5-7: State types

Initial state

Every state activity diagram begins with an initial state, indicated by a solid circle. It serves only to mark the beginning of the state machine and has no behavior logic. It must have one transition leading to another state; it may not have any triggers or guard conditions on this transition. It may not have transitions leading into it from another state.

Simple state

The simplest of states that describe an object at a specific point in its life cycle. It may contain behavior logic, data transformations and D2M conversions.

Decision state

Decision states are the same as simple states, with one exception as noted below and their primary purpose is to provide a 'visual cue' to the 'intention' of that state, not to be functionally different from simple states. The difference is that there must be at least two outgoing transitions from a Decision State, otherwise there is no opportunity to 'decide' between at least two possible options. The 'intention' of Decision State is that a decision is being made in that state and that is visually indicated for better readability of the State Machine.

Sequential composite state

Like the state machine itself, a sequential composite state starts with an initial state and ends with a final state. Between the two there may be one or more of a simple state, or another composite state (sequential or concurrent).

A sequential composite state does not complete until its internal process reaches its final state. If it has an eventless (non-triggered) transition, that transition is taken when the final state completes. A sequential composite state that lacks a final state will cause the entire state machine to hang at that location.

A triggered transition may be taken earlier, if the triggering event arrives before the internal state machine completes. This can result in the internal state machine being summarily terminated (per UML state machine semantic rules).

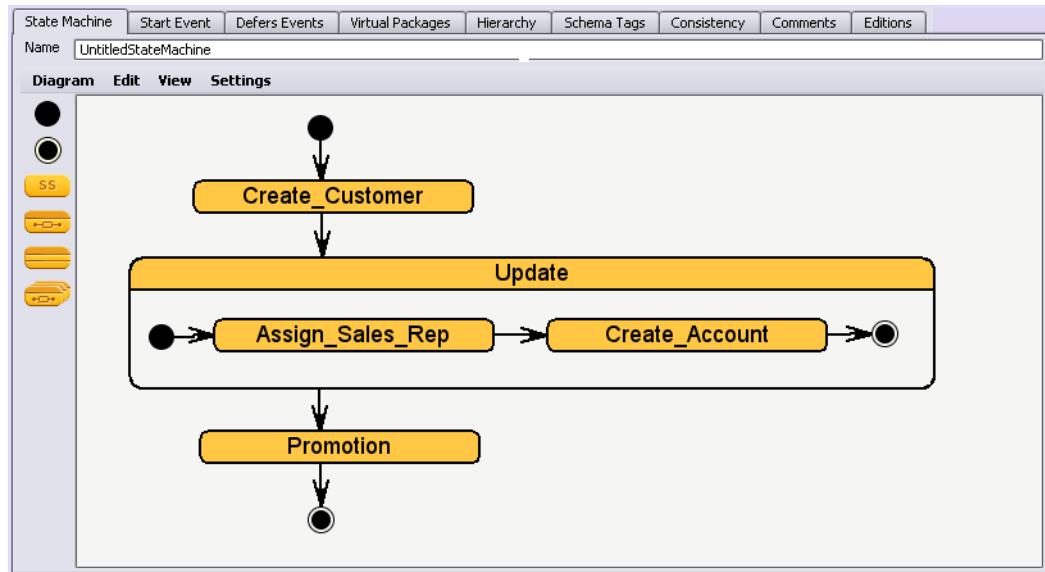


Figure 5-8: Sequential composite state

Creating a sequential composite state

To create a sequential composite state:

1. From the state machine editor's icon bar, drag and drop the **sequential composite state** icon  into the state machine editor.
2. Define the sequential composite state, as needed.
See “Defining states” on page 122.
1. From the state machine editor's icon bar, drag and drop the **initial state** icon  into the sequential composite state icon.
2. Resize the composite state icon to accommodate the nested states.
When adding a state to a concurrent or composite state, the concurrent or composite state must be large enough to accommodate the state icon you are adding, otherwise the addition fails.
 - a. Place the cursor over the state icon's border.
A double-ended arrow appears.
 - b. Click on the border and drag it outward to expand the icon.
3. From the icon bar, select, drag and drop a state icon into the sequential composite state.
For example, the simple state icon .
4. Add any other state icons and name them, as necessary.
5. From the icon bar, select the **final state** icon , drag and drop it in the **sequential composite state**.
6. Double-click each state to define it.
See “Defining states” on page 122.
7. Create the transitions between the states.
See “Creating transitions between states” on page 126.

Concurrent composite state

A concurrent composite state contains other states that execute in parallel. The concurrent state may contain behavior logic, but not data transformations nor D2M conversions. It may contain simple states that do contain these, however. A concurrent composite state contains one or more states that execute in parallel. It may be located in the state machine proper or nested within another composite state. Each region within a concurrent composite state must have an initial and a final state along with the other states that execute in that region.

A Concurrent Composite does not complete until one of the following happens:

- Every one of its concurrent regions (each of which is actually a sequential composite state) reaches its final state. Every concurrent region must have a final state. A concurrent region that lacks a final state will cause the entire state machine to hang at that location.
- A transition from inside the concurrent composite to outside its boundary is executed. In this case, incomplete concurrent regions are summarily terminated.

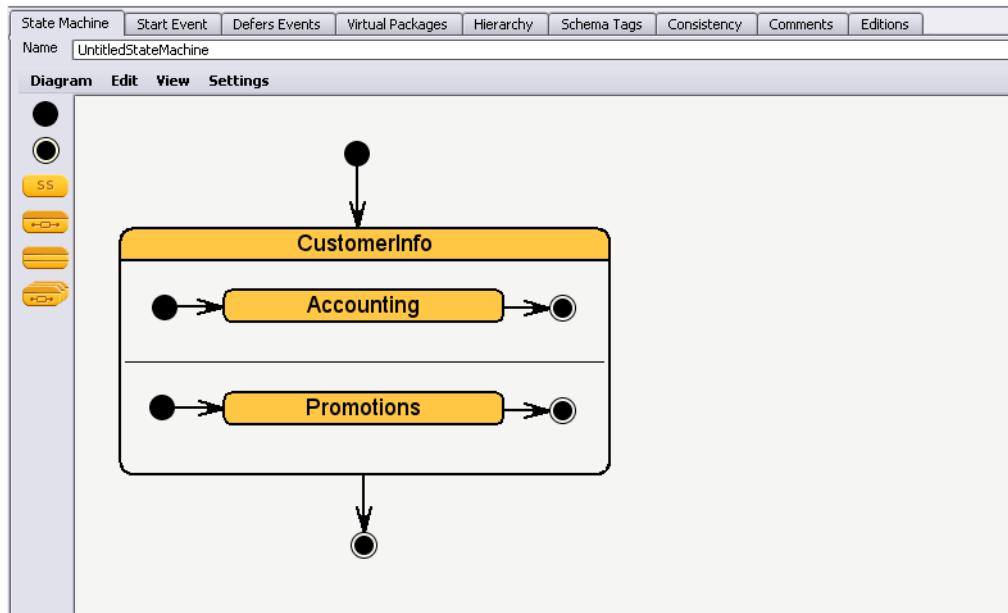


Figure 5-9: Concurrent composite state

Creating a concurrent composite state

To create a concurrent composite state:

1. From the state machine editor's icon bar, drag and drop the **concurrent composite state** icon

2. Define the concurrent composite state, as needed.

See “[Defining states](#)” on page 122.

3. Resize the concurrent composite state icon to accommodate the nested states.

When adding a state to a concurrent or composite state, the concurrent or composite state must be large enough to accommodate the state icon you are adding, otherwise the addition fails.

a. Place the cursor over the state icon’s border.

A double-ended arrow appears.

b. Click on the border and drag it outward to expand the icon.

4. From the icon bar, select, drag and drop a state icon into the sequential composite state.
For example, the simple state icon .
5. Add any other state icons and name them, as necessary.
6. From the icon bar, select the **final state** icon , drag and drop it in the **sequential composite** state.
7. Double-click each state to define it.
See “Defining states” on page 122.
8. Create the transitions between the states.
See “Creating transitions between states” on page 126.
9. Select the concurrent composite state’s label, right-click and select **Add concurrent region**.
A new region appears in the concurrent composite state. The concurrent composite state icon includes two concurrent regions by default.
10. Repeat steps 3 through 8.
11. Repeat the previous steps for any additional concurrent regions.

Replicate state

A replicate state synchronizes multiple runtime instances of the same state in a state machine. A replicate state is a subclass of the composite state, and works similarly to a composite state, except more than one runtime instance may be spawned. This is useful when you don’t know at run-time how many objects will be processed, and the processing threads must be dynamically generated. A replicate state has a Replicates Expression field where action semantics can specify how many replicate states are to be spawned. The Replicates expression can return the number of replicates, or a Collection of Objects (one per Replicate), or an Iterator over a Collection of Objects (one per Replicate).

As with a composite state, other states, transitions and guards can be inserted into the replicate state. For the action semantics of these states, transitions and guards, a bound variable refers to the replicate state. Its name is the same as the name of the replicate state followed by “_ID” (that is, if you leave your replicate named “Replicate_1234512345”, the bound variable will be named `Replicate_1234512345_ID`). Action semantics in the states of the replicate state can reference this run-time replicate state ID bound variable and thereby assign separate pieces of the process to separate threads.

There is an additional bound variable with an object from the collection or iterator, if any. This bound variable’s name is `<Replicate'sName>_OBJ`.

If the “Replicates Expression” evaluates to an Integer, that number determines the number of Replicates that will be created. For each Replicate, `<Replicate'sName>_ID` is assigned a number from 1 to `NumberOfReplicates`, and, `<Replicate'sName>_OBJ` is assigned a value of null.

If the “Replicates Expression” evaluates to a collection or iterator, one Replicate is created for each object in the collection or iterator. For each Replicate, `<Replicate'sName>_ID` is assigned a unique integer value (starting from 1), and `<Replicate'sName>_OBJ` is assigned a value from the collection or iterator. The Object must be a BusinessObject, or, it must be serializable.

Creating a replicate state

To create a replicate state:

1. From the state machine editor’s icon bar, drag and drop the **replicate state** icon  into the state machine editor.
2. Define the replicate state, as needed.

See “Defining states” on page 122.

3. In the **Number of Replicates Action** tab, define the number of run-time instances for the replicate state.

The action semantics in this field must return the number of replicate states. The return statement may report a constant, as in `return 3;`, or report a variable populated at run-time, as in `return replicates;` where `replicates` is the variable that holds the number of replicate states.

4. Resize the state icon to accommodate any nested states.

When adding a state to a replicate state, the replicate state must be large enough to accommodate the state icon you are adding, otherwise the addition fails.

- a. Place the cursor over the state icon’s border.

A double-ended arrow appears.

- b. Click on the border and drag it outward to expand the icon.

5. Add any nested states, if needed.

See “Placing states in a state machine” on page 122.

6. Double-click each state to define it.

See “Defining states” on page 122. Note that the action semantics for these states may refer to the run-time instance of the replicate state in which they are contained by the variable named for the replicate state followed by `_ID`. For example, `Replicate_1234512345_ID`. The value for the variable is an integer that identifies the specific run-time instance of the replicate state.

7. Create the transitions between the states.

See “Creating transitions between states” on page 126.

Final state

The final state, indicated by a solid circle surrounded by a circle outline, marks the end of a state machine. The final state may include behavior logic, but not data translations nor D2M conversions.

Placing states in a state machine

To add a state to a state activity diagram:

1. From the state machine editor’s icon bar, select the icon for the state, drag and drop it into the editor.

Usually, you start by adding the initial state icon

When adding a state to a replicate, concurrent or composite state, the replicate, concurrent or composite state must be large enough to accommodate the state icon you are adding, otherwise the addition fails. Enlarge your replicate, concurrent or composite state before adding other states.

2. Double-click the state icon to bring up the state’s editor window.

See “Defining states” on page 122.

Defining states

To define a state:

1. In the state editor, in the **State** tab, in the **Name** field, enter a name.

It is always a good practice to name a state, even the initial state, so that you can identify it during debugging.

2. In the **Actions** tab, describe the processing logic for this state.
See “[State actions](#)” on page 123
3. In the **Transformation** tab, describe a transformation, if needed.
See “[Transformation unit](#)” on page 124.
4. In the **Transformation** tab, describe the D2M conversion, if needed.
See “[D2M conversion](#)” on page 96.

Setting states from within transitions

To set the target or source state of a transition:

1. In the transition editor, in the **Transition** tab, above either the **Source** or **Target** state field, click **Set**.
The source state is that from which the transition begins. The target state is where the transition ends. Valid selections include only those states defined in the same state machine.
2. In the **Select Object(s)** dialog, select the state and click **OK**.

Editing states from within transitions

To edit the target or source state of a transition:

1. In the transition editor, in the **Transition** tab, above either the **Source** or **Target** state field, click **Edit**.
The source state is that from which the transition begins. The target state is where the transition ends. Valid selections include only those states defined in the same state machine.
2. Define the state.
See “[Defining states](#)” on page 122.

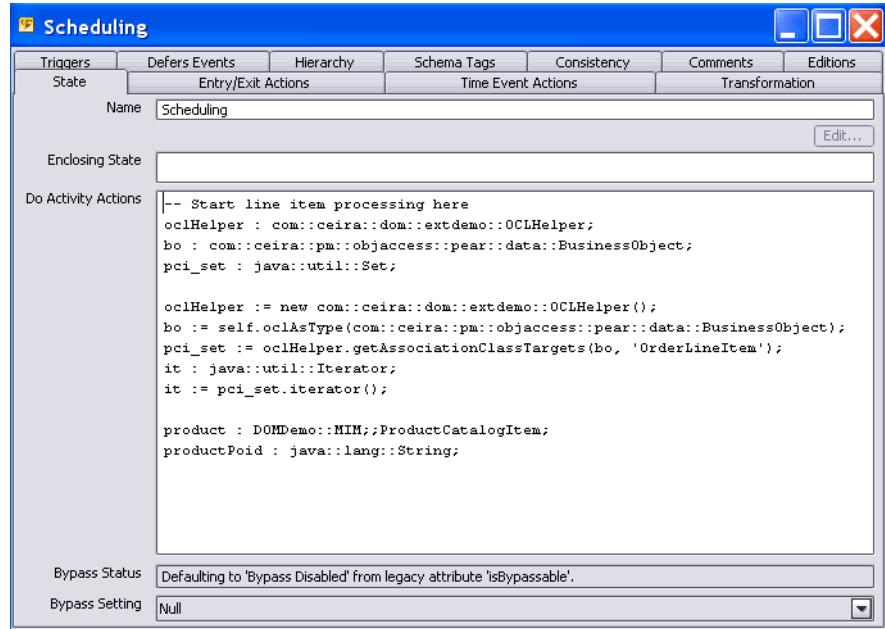
State actions

Except for the initial state, all of the states may include behavior logic that dictates what happens during the state. This logic is usually described in actions in any of three fields:

- **Entry Actions** that describe what happens when the object enters the state.
- **Do Activity Actions** that describe what happens during the state.
- **Exit Actions** that describe what happens when the object leaves the state.

In these fields you can enter comments to describe the logic, as well as actions that invoke web services, generate events, and perform other processing tasks. See *Using Actions in Cogility Studio*. How you use these fields may depend upon the transaction boundaries within which you want the logic to be executed. See “[Transactions](#)” on page 111 for more information.

Actions, as shown below, provide the run-time processing logic. See *Using Actions in Cogility Studio* for more information about actions.



Comments are preceded on each line with a double dash (--). To avoid registering an inconsistency, use a double dash at the start of each line of your comments.

Creating state actions

State actions can be defined on two tabs, depending on the type of action to be defined. Exit/entry actions are defined on the **Entry/Exit Actions** tab. Do Activity actions are defined on the **State** tab.

To create do activity state actions:

1. In the state editor, click the **State** tab.
2. In the **Do Activity Actions** field, enter actions that describe the logic for the state.
3. Click the **Consistency** tab to check for errors.

As soon as you click any other tab or out of the editor window the actions are compiled. Any compilation errors appear in the **Consistency** tab.

To create entry/exit state actions:

1. In the state editor, click the **Entry/Exit Actions** tab.
2. In the **Entry Actions** field, enter actions that describe the entry action for the state.
3. In the **Exit Actions** field, enter actions that describe the exit action for the state.
4. Click the **Consistency** tab to check for errors.

As soon as you click any other tab or out of the editor window the actions are compiled. Any compilation errors appear in the **Consistency** tab.

Transformation unit

You can embed a transformation unit in a simple state. Doing so makes a state machine part of a data path between two data object sets. The transformation unit may be a serial transformation chain, a

parallel transformation chain, a transformation map or an opaque transformation map. For more information about creating transformation units see “[Transformation model](#)” on page 145.

In example below, the UpdateCustomerIDs simple state invokes the parallel transformation chain, FanOut_PTC.

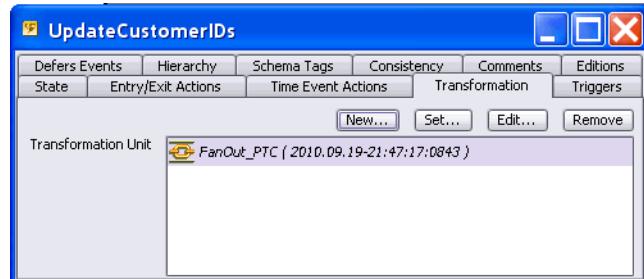


Figure 5-10: A transformation in a simple state

To create a transformation unit in a simple state:

1. In the simple state editor, select the **Transformation** tab.
2. To create a new transformation, click **New**.
 - a. Select the transformation unit type and click **OK**.
 - b. In the dialog box, enter a name for the transformation unit and click **OK**.
3. To select an existing transformation unit, click **Set**.
 - a. Select the transformation unit from the list.
 - b. Click **OK**.
4. To define the transformation unit, click **Edit**.

See “[Transformation model](#)” on page 145 for information on defining the transformation.

Transitions

A transition moves an object from one state to another. Transitions are unidirectional (an object cannot retrace a transition backwards) and indicated as arrows in a state activity diagram. Just as a state may define behavior logic, a transition may define transition logic.

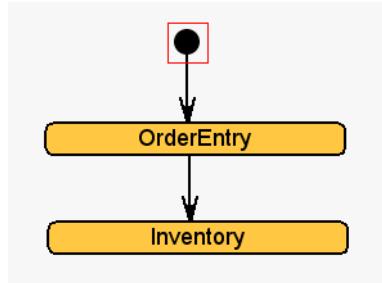


Figure 5-11: A transition describes an object between states

Transitions may also be circular, that is, they may refer the object back to the original state. The transition logic would specify a condition that causes this return.



Figure 5-12: A transition may refer the object to the original state

An object's transition from one state to another may be triggered by an event (see “Event trigger” on page 127) or it may be controlled by a guard condition (see “Guard condition” on page 128). Transitions may include logic that dictates what happens during the transition (see “Effect action” on page 129).

Creating transitions between states

To create a transition from one state to another:

1. In the state activity diagram, select the state icon to transition from, hold down the **Ctrl** key, and drag the cursor to the state icon for the destination state.
2. Double-click the transition arrow to define the transition.

See “Defining a transition” on page 126.

Note: If the routing lines on the state activity diagram become to cluttered, you can select the transition routing lines (hold down the shift key while selecting multiple lines) and right-click for the routing menu. Select **Clear routing**. The routing lines will be automatically redrawn to the most direct routing.

Creating a circular transition

To create a circular transition:

1. In the state activity diagram, select the object to transition from, right-click and select **Connect to Self**.
2. Double-click the transition arrow to define the transition.

See “Defining a transition” on page 126.

Defining a transition

To define a transition:

1. In the transition editor, in the **Transition** tab, in the **Name** field, enter a name.
It is always a good practice to name a transition so that you can identify it during debugging.
2. In the **Transition** tab, set or edit the source and target states.
See “States” on page 117. Usually you set the states when you define the state machine. See “Defining behavior artifacts manually” on page 116. But you can change the source and target states though this tab as well as edit them. Under normal usage, you would not set the source or target states. A transition automatically gets these values assigned to it when you graphically draw the transition between two states. The Set function comes in handy when you have to move a transition, that is, preserve it instead of deleting it and creating a new

one. You could have useful data already set on the transition that you do not want to lose; for example, an effect or guard.

3. In the **Trigger** tab, define the event trigger, if needed.
See “Event trigger” on page 127.
4. In the **Effect and Guard** tab, define the guard condition.
See “Guard condition” on page 128.
5. In the **Effect and Guard** tab, define the effect action.
See “Effect action” on page 129.

Event trigger

Events can effect transitions between states as well as start state machines (see “Start event” on page 130). A transition may be triggered by an event or it may be eventless. If it is eventless, the transition is taken when the source state’s doActivityAction is completed. If it is triggered, the transition is taken when the source state’s exitAction is active and the matching event has been received by the runtime behavior.

The event originates as a message from an integrated application or an internal object. See “Events” on page 82. Trigger events may be of any event type: event, DOS event, or time event. The procedure for specifying these in the transition is the same for all.

A standard event describes a stimulus from an integrated application or internal object. See “Standard event” on page 84.

A DOS event encapsulates object data (see “DOS event” on page 85). The object data may be used in the effect action for the transition. See “Effect action” on page 129.

A time event triggers a transition in a state machine after a specified period of time. See “Time event” on page 86.

In the example illustrated below, the BillingAccountRegistryResponse event is set in the transition. When the Billing system sends a message indicating it has received and updated the new customer information, that message gets converted to an event that triggers the transition of the MIM Customer object out of the Billing state.

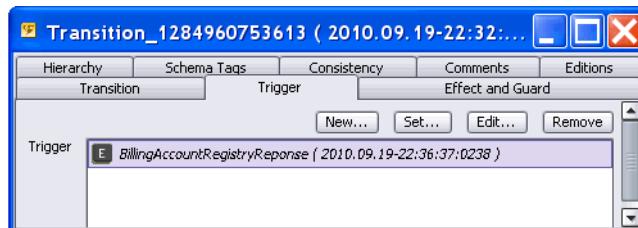


Figure 5-13: A transition's event trigger

The event name also appears in the state machine diagram next to the transition to denote the event trigger.

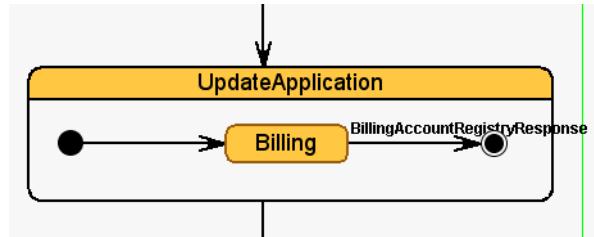


Figure 5-14: An event trigger labels a transition

Creating a trigger event in a transition

To create a trigger event in a transition:

1. In the transition editor, in the **Trigger** tab, click **New**.
2. In the **Select Type** dialog, select the event type and click **OK**.
3. In the dialog, enter a name for the event, and click **OK**.
4. To define the event, click **Edit**.
For standard events, see “Standard event” on page 84.
For DOS events, see “DOS event” on page 85.
For time events, see “Time event” on page 86.

Setting a trigger event in a transition

To set a trigger event in a transition:

1. In the transition editor, in the **Trigger** tab, click **Set**.
2. In the **Select Object(s)** dialog, select the event and click **OK**.
3. To define the event, click **Edit**.
For standard events, see “Standard event” on page 84.
For DOS events, see “DOS event” on page 85.
For time events, see “Time event” on page 86.

Guard condition

A transition may have only one guard condition. If the condition evaluates to ‘true’ the transition may take place. If a state has multiple transitions whose guard conditions evaluate to ‘true,’ only one of the transitions is taken (depending on UML rules for transition precedence).

You define a guard condition with actions. For more information about actions, see *Using Actions in Cogility Studio*.

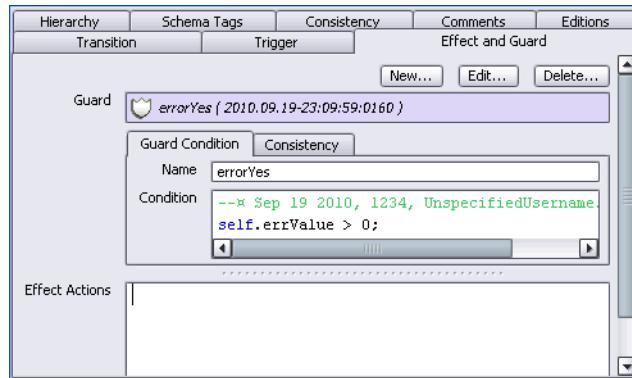


Figure 5-15: Guard condition

The guard condition for each of the possible transitions out of a state is shown in the state activity diagram.

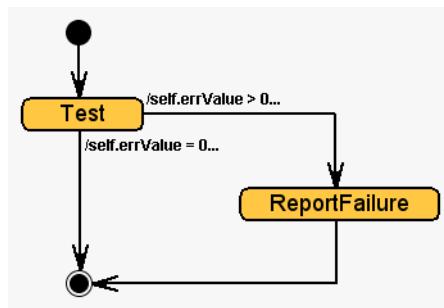


Figure 5-16: Guard conditions label transitions

Creating a guard condition

To create a guard condition:

1. In the transition editor, click the **Effect and Guard** tab.
 2. For **Guard**, click **New**.
 3. In the **Name** field, enter a name for the guard condition.
 4. In the **Condition** field, enter the action that describes the condition.
- See [Using Actions in Cogility Studio](#) for more information about actions.*

Effect action

A transition may have one or more effect actions that specify one or more operations on the object data or perform some other task with actions. An effect action takes place during the transition. The effect action may make use of the class data held in a DOS. Such an action requires that the transition's event trigger be defined with as DOS event. See “[Event trigger](#)” on page 127.

Compare an effect action to an exit action: an effect action is specific to the transition whereas an exit action occurs no matter which transition is taken. See “[Transactions](#)” on page 111.

In the following example, the effect action is to assign the event's `accountID` to the `billingID` in the object. See *Using Actions in Cogility Studio* for more information about actions.

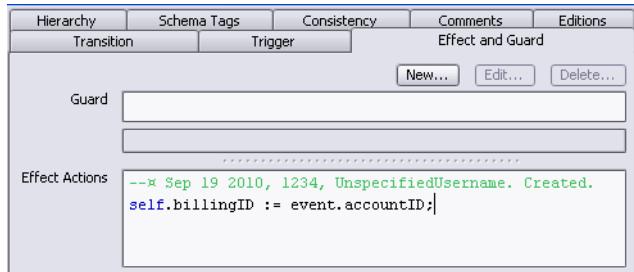


Figure 5-17: Effect action

Creating an effect action

To create an effect action in a transition:

1. In the transition editor, click the **Effect and Guard** tab.
2. In the **Effect Actions** field enter the action.

As soon as you click out of the Expression field, the action is compiled. Any errors appear in the Consistency tab. See *Using Actions in Cogility Studio* for more information about actions.

Start event

Events serve as stimuli for state machines (behaviors). They can both start state machines and effect transitions between states (see “Event trigger” on page 127). The class or factory with the state machine must designate a start event; the state machine cannot execute without it. A start event may be either a standard event or a DOS event. For more information about events, see “Events” on page 82.

Because classes may have more than one start event and more than one behavior, each start event must be associated with a specific behavior. You must create this behavior before associating it with the start event. See “State machine” on page 112.

A start event can be defined in either a class or factory, or in a state machine. A start event which has already been defined can be added to an existing class or factory.

Applicability actions in a start event

Start Events and Trigger Events behave differently. Trigger Events transition a State Machine from one state to another. Start Events, on the other hand, are responsible for starting a State Machine on a business object.

When a Trigger Event is delivered to a specific State Machine, running on a Business Object, “guard actions” specified on the transition act as a final gate for determining the event’s applicability. This gives you programmatic control over the applicability, based on desired criteria. If an Event is deemed to not be applicable, it is ignored.

Applicability actions provide a similar functionality for Start Events. These actions control whether a Start Event, that is delivered to a Business Object, actually starts a new State Machine or not. A Start Event that is deemed to not be applicable, is ignored.

There are two situations where this can apply:

- When you wish to avoid starting a State Machine. This might occur where you want to start a business process for only some combination of attribute values in the incoming event and attribute values of business objects in the database.
- Where you might expect multiple occurrences of the same event. The applicability test actions allow you to start a new State Machine when the first of these events is delivered to a business object but avoids doing so, for additional occurrences of the event.

Creating a start event

To create a start event:

1. Do one of the following:
 - a. In the class or factory editor, in the **Start Event** tab, click **New**.
 - b. In the state machine editor, in the Diagram menu, select **Set New Start Event...**. You can also right click in the state machine editor to access **Set New Start Event...** option.
2. In the **Select Type** dialog, select the event type and click **OK**.
3. In the dialog, enter a name for the event and click **OK**.
4. To define the event, click **Edit**.

For standard events, see “[Creating a standard event](#)” on page 84.
 For DOS events, see “[Creating a DOS event](#)” on page 86.
5. In the **Start Event** field, select the start event.
6. Under the **Started Behavior** tab, click **Set**.
7. Select the behavior and click **OK**.
 - a. To edit the state activity diagram for the started behavior, click **Edit**. See “[State machine](#)” on page 112.
8. Select the **Applicability Actions** tab to define applicability actions for the start event. The Applicability Action evaluates to true or false and indicates whether a received Start Event is applicable for a State Machine:
 - If true, the State Machine is started.
 - If false, the Start Event is ignored.

If Applications Actions is undefined, it defaults to true. Within Applicability Actions there are two bound variables; 'startEvent' is the Start Event and 'self' is the Class or Factory for the State Machine.

Adding a start event to a class or factory

To add a start event to a class or factory:

1. In the class or factory editor, in the **Start Event** tab, click **Add**.
2. In the **Select Object(s)** dialog, select the event and click **OK**.
3. To define the event, click **Edit**.

For standard events, see “[Creating a standard event](#)” on page 84.
 For DOS events, see “[Creating a DOS event](#)” on page 86.
4. In the **Start Event** field, select the start event.
5. Under the **Started Behavior** tab, click **Set**.
6. Select the behavior and click **OK**.
7. To edit the state activity diagram, click **Edit Diagram**.

See “State machine” on page 112.

Batch processing

Sometimes during a business process you want to work with a large number of records, too many to process in one transaction. Doing so would cause database performance problems, and any one record in error would require the entire group of records to be reprocessed. Breaking the records into several batches and processing each in its own transaction alleviates these problems. You can use state machines to work with batches in this fashion.

State machines begin and commit transactions with the database automatically, thereby defining the transaction boundaries. As described in “Transactions” on page 111, a state transaction is bounded within the state’s do activity. When a large group of records is broken into small batches, each batch may be processed in its own state transaction. For any errors encountered, the transaction may be rolled back and restarted.

For an example of a model that includes a batch processing scenario, see “Batch processing example” on page 135.

Limitations

Running batch processes with a state machine is inherently bound by the limitations imposed by the J2EE application server. The state machine, executes within Cogility Manager on the J2EE application server. The application server imposes a time limit on the message-driven beans that perform database transactions. The default time limit for these transactions is usually two minutes, though the environment may be configured for longer time limits.

However, running a batch process within the application server process is not the ideal approach. A batch process is inherently a big job that processes potentially a very large number of database records, even when broken into small batches. A transaction for a batch may not complete within the two minute time limit.

A better approach is to execute a batch processing script in its own process, apart from the application server, using Cogility Action Pad.

BusinessObjectsBatch

In your state machine’s do activity action or in a script action, you must declare a batch of type `com::cogility::poal::BusinessObjectsBatch`. The batch object creates and maintains sub-batches for each group of business objects you want to work with from the greater batch of records. Each sub-batch has its own transaction.

newBatchFor

The `newBatchFor()` method instantiates a batch object of the `BusinessObjectsBatch` class, sets the search attributes for the records in the batch, the sort string for sorting the records of the batch, the number of objects in a sub-batch, and the threshold qualifiers that determine how many transactions are allowed to fail before disqualifying the batch. The signature is as follows:

```
public static BusinessObjectsBatch newBatchFor(
    String type,
    Map searchAttributes,
    String sortString,
    int batchSize,
```

```

        int percentThreshold,
        int absoluteThreshold)
throws OAException
{
    return (new BusinessObjectsBatch(
        type,
        searchAttributes,
        sortString,
        batchSize,
        percentThreshold,
        absoluteThreshold)
    );
}

```

The parameters are described as follows:

Parameter	Type	Description
<code>type</code>	String	The type for the objects in the batch, usually cast with single quotes. For example, a batch of Subscriber objects would be set as ' <code>MyModel.MIM.Subscriber</code> '.
<code>searchAttributes</code>	Map	The search attributes are contained in a Java Map object that associates key/value pairs. You must call the Map object's <code>put()</code> method to assign the key/value pair for the search.
<code>sortString</code>	String	The string that describes the attribute by which you wish to sort the collection of objects in the batch.
<code>batchSize</code>	Integer	The number of objects in the batch that you wish to handle in a single transaction as a sub-batch. If you have a batch of 1000 objects and specify a <code>batchSize</code> of 5, you have 200 sub-batches. The sub-batches are actually separate instances of the batch object, they are maintained internally by the batch object. Making this number larger means that there are fewer sub-batch executions to handle. But it also means that an error and transaction rollback (which rolls back the entire current sub-batch) might undo a larger number of desired changes. It also means that more objects are read into memory during processing of the sub-batch.
<code>percentThreshold</code>	Integer	The percent of sub-batch transactions that may fail before halting processing of any other sub-batches.
<code>absoluteThreshold</code>	Integer	The number of sub-batch transactions that may fail before halting processing of any other sub-batches. The threshold value that returns the lowest number of sub-batches determines which of <code>percentThreshold</code> or <code>absoluteThreshold</code> is used.

For example, in the following batch object creation, a batch of Subscriber objects is created, a Map object is set, the sort key, '`age`' is set, the `batchSize` is set to 5, and the percent and absolute thresholds are set to 10 and 4, respectively.

```

batch := (null.oclAsType(com::cogility::poal::BusinessObjectsBatch))
        .newBatchFor('MyModel.MIM.Subscriber', map, 'age', 5, 10,
4);

```

For an example of this, see “[CreateBatch](#)” on page 138.

State variable

For processing a batch in a state machine, you save the ID of the batch object in a state variable located on the state machine’s parent object so that the batch object is visible to each state in the process. A known variable called `parent` describes the container of the state in which the expression is being written, and this is where the state variable is held.

The entire state machine is enclosed in a top-level sequential composite state, not shown on the state activity diagram, which is the context for the entire diagram. This `parent` makes a convenient place to store data, because it is visible to many different states. However, if the state activity diagram includes composite states, the states under the composite state do not have the same value for `parent`. Their `parent` is defined on the composite state.

By virtue of its being in the parent object of the state machine, the state variable is available throughout the state machine. You use the `SetExtendedStateData` and `GetExtendedStateData` methods to set and get the state variable.

Set the state variable

To set the variable with the batch, you call the `SetExtendedStateData()` static method. For example, the following Java action calls the method which assigns the batch object’s persistent object ID (POID) to the String `batchPoid` and locates the variable on the parent.

```
java com::ceira::oem::SetExtendedStateData(parent, 'batchPoid',
batch.getIdString());
```

Notice that the `BusinessObjectsBatch.getIdString()` method is called to retrieve the batch’s POID. This serves as the value for the key, `'batchPoid'`. Since extended state data is stored in the database, only simple data can be stored. You can store the ID of a business object, for example, but not a business object itself.

Get the state variable

When you are ready to work with the batch, you retrieve it with the `GetExtendedStateData()` method. You must maintain the type casting on this object in referring to it by its POID, which is a String. In the following action example, a Java String to hold the state variable is first declared, then the variable is set to the batch’s POID.

```
s : java::lang::String;
s := java com::ceira::oem::GetExtendedStateData(
    parent, 'batchPoid').oclAsType(java::lang::String);
```

In this example the `oclAsType()` method casts the `batchPoid` to a Java String. The `GetExtendedStateData()` method retrieves the variable, and the action assigns it to the String `s`.

For more information see the guide, [Using Actions in Cogility Studio](#).

Get the batch

Once you have retrieved the state variable, described above, you may retrieve the batch object. In the following example, a `BusinessObjectBatch` is first declared, then assigned the batch object via the `LocateByIdentity()` method, which locates the batch object by its POID string.

```
b : com::cogility::poal::BusinessObjectsBatch;
b := java com::ceira::oem::LocateByIdentity(s).oclAsType(
```

```
com::cogility::poal::BusinessObjectsBatch);
```

Once again the oclAsType() method is called to cast the object as a BusinessObjectsBatch. Now that you have the batch object, you can work with it.

For more information about these methods, see the guide, *Using Actions in Cogility Studio*.

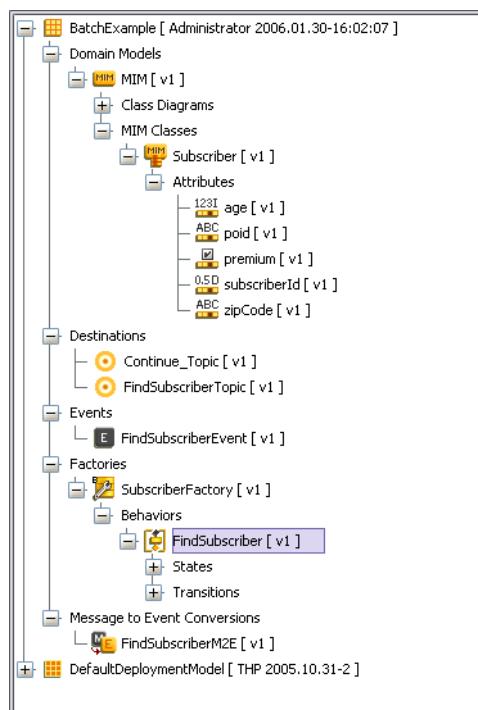
Batch processing example

The following example presents the basics of batch processing in a state machine. You can use it to build your own batch processing scenario. The batch processing example model is available as a file that you can load into Cogility Modeler. The model's objects and actions are also described in the following sections. The example model is located as %DCHOME%\Examples\Behavior\BatchProcess\BatchExample_v1.cog.

See “[Message definitions](#)” on page 108 for loading instructions.

Model objects

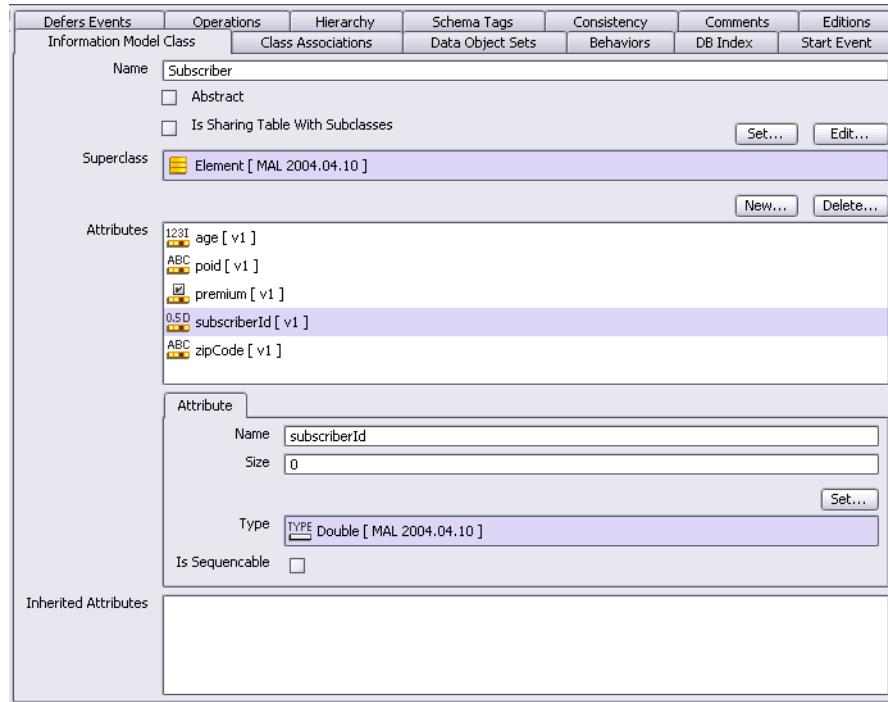
This simple model consists of a Subscriber MIM class and a factory class. The factory class includes the behavior to create a batch object. The behavior instantiates a subset of Subscriber objects for a range of object IDs sorted by age. The model container also includes a message to event conversion for the findSubscriber event that serves as the start event for the behavior. These objects are illustrated in the following figure.



Subscriber class

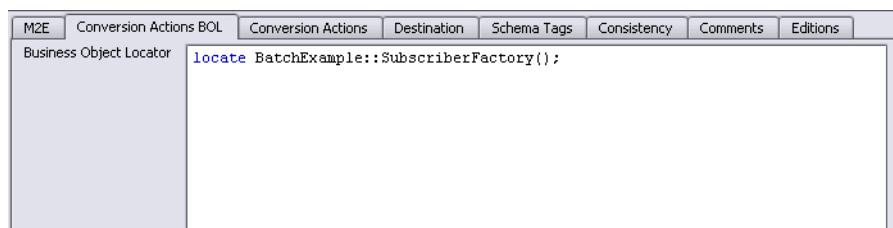
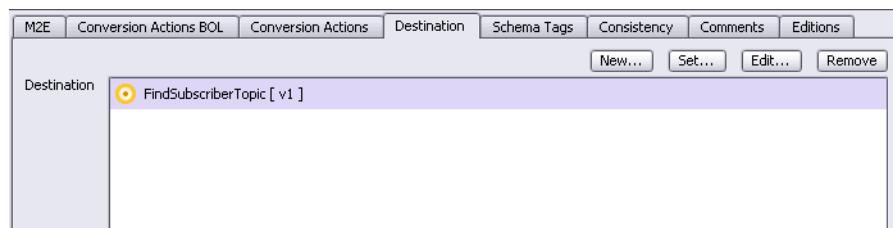
The subscriber is described by the following attributes:

- **age** - the subscriber's age as an integer.
- **poid** - the persistent object ID; a unique identifier String.
See “[POID attribute](#)” on page 37.
- **premium** - a Boolean for whether or not the subscriber is a premium subscriber.
- **subscriberId** - a double type for a unique subscriber identification number.
- **zipcode** - the subscriber's zip code, expressed as a String.



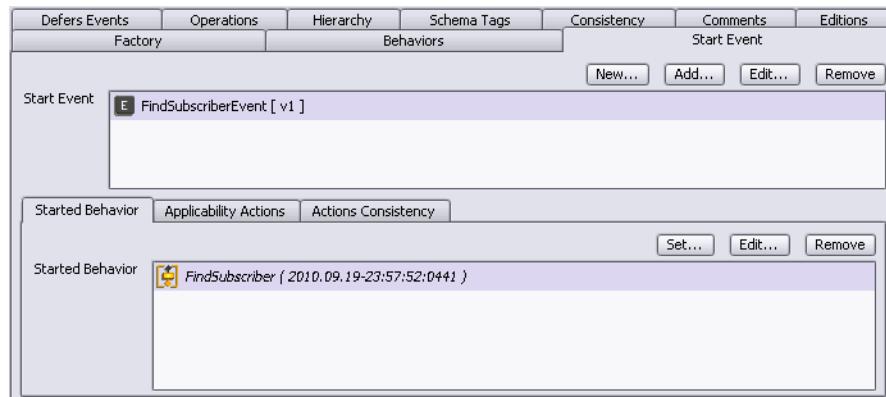
FindSubscriberM2E

The FindSubscriberM2E listens for the FindSubscriberTopic message, locates the SubscriberFactory and generates the FindSubscriberEvent.



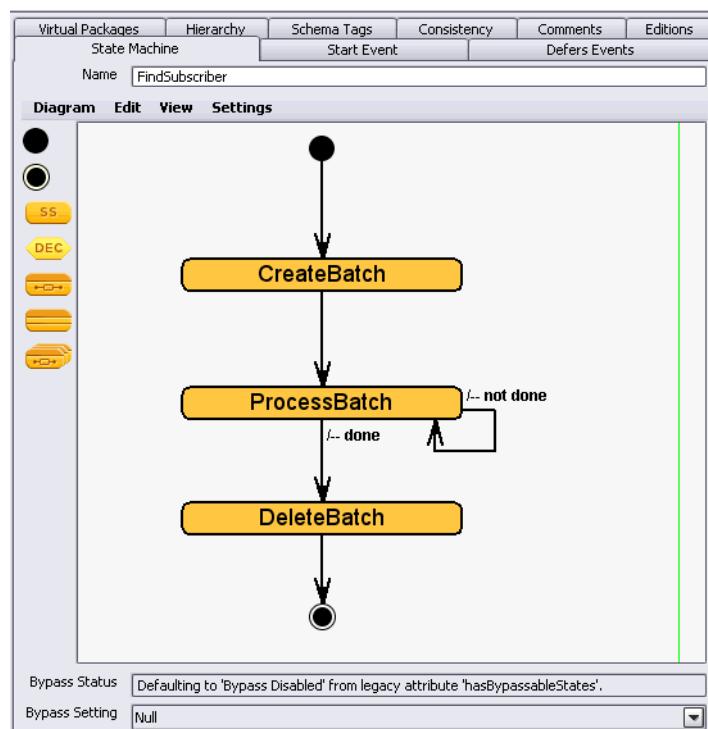
FindSubscriberEvent

The FindSubscriberEvent serves as the start event for the SubscriberFactory class and the FindSubscriber behavior.



FindSubscriber behavior

The FindSubscriber state machine describes the process that creates the Subscriber objects within a batch object. It has two states, the first to create the batch, the second to instantiate the Subscriber objects and place them in sub batches.



Batch processing actions

Within each state of the FindSubscriber state machine, the do activity actions dictate the logic for creating and working with a batch of Subscribers. Each Subscriber object passes through the

FindSubscriber state machine; it is placed in a sub-batch and processed. Processing in this example is limited to setting the value of a Subscriber object's variable. If this processing does not generate an exception, the sub-batch is committed to the database. A separate transaction is committed for each sub-batch. If the processing of any Subscriber objects in a sub-batch throws an exception, the sub-batch transaction is rolled back.

CreateBatch

In the CreateBatch state of the FindSubscriber state machine, in the do activity action, the batch object is created and stored on the `parent` state variable.

```
batch : com::cogility::poal::BusinessObjectsBatch;
map : java::util::HashMap;
map := new java::util::HashMap();
directSQL : com::ceira::DirectSQLString;
directSQL := new com::ceira::DirectSQLString('BETWEEN 1000 and 1500');
map.put('subscriberId', directSQL);

batch :=
(null.oclAsType(com::cogility::poal::BusinessObjectsBatch)).newBatchFor(
'BatchExample.MIM.Subscriber', map, '', 5);

----- save id of the batch object in extended state data
java com::ceira::oem::SetExtendedStateData(parent, 'batchPoid',
batch.getIdString());

java com::cogility::Println('BATCH ID='.concat(batch.getIdString()));

----- set a variable for the batch count
cnt : java::lang::Integer;
cnt := 1;
java com::ceira::oem::SetExtendedStateData(parent, 'count', cnt);
```

Instances of the BusinessObjectsBatch class, called sub-batches, are created to hold a set of objects; each of these sub-batches will be committed to the database in a separate transaction. These sub-batches are maintained by the batch object identified by `batchPoid`. When all sub-batches are committed, you delete the batch object. To create the batch, the `newBatchFor()` method is called (this is described in ["newBatchFor" on page 132](#)). It requires the following parameters:

- The type for the objects in the batch, in this case, `MyDOMSchema.MIM.Subscriber` objects.
- Search attributes described in a Java.util.HashMap object. In this example, the map variable holds the name/value pair that describes the search attributes. You must call the Map object's `put()` method to set the pair. Here the name of the search attribute is '`subscriberId`' and the value is held in a `com::ceira::DirectSQLString` which supplies a search criteria '`'BETWEEN 1000 and 1500'`'. The sub-batches are composed of Subscriber objects with a `subscriberId` between 1000 and 1500.
- The sort string that describes the attribute by which you wish to sort the collection of objects in the sub-batch, in this case, the attribute is not specified (''), so the sub-batch will not be sorted.
- The sub-batch size, `5`. This integer describes the number of objects held in a sub-batch. The batch is processed in sub batches to avoid the problem of having to reprocess the entire batch in the event of a system failure. Sub-batches are actually separate instances of the `BusinessObjectsBatch` batch object and they are maintained internally by the batch object.
- The threshold percent, not specified. This integer describes the percent of sub-batch transactions that may fail before halting processing of any other sub-batches.

- The absolute threshold, not specified. This integer describes the number of sub-batch transactions that may fail before halting processing of any other sub-batches.

Once the batch is created with the newBatchFor() method, its persistent object ID (POID) is stored in a state variable. A bound variable called `parent` describes the top level object of the state machine, and this is where the state variable is held. By virtue of its being in the parent object of the state machine, the state variable is available throughout the state machine. You use the constructor methods for the SetExtendedStateData and GetExtendedStateData classes to set and get the state variable.

In this example, to set the state variable, you call `setExtendedStateData()`, locating the variable on the `parent`, getting the batch object's POID and and assigning it to the String '`batchPoid`'.

In this example, for monitoring the processing of sub-batches, a variable to hold the batch count is established. The batch count is initialized on the `parent` by passing the '`count`' String and assigning to it the value held in the local variable, `cnt`.

For more information see “[Set the state variable](#)” on page 134.

ProcessBatch

In the ProcessBatch state of the FindSubscriber state machine, in the do activity action, the sub-batch is retrieved and each of its Subscriber objects are processed.

`GetExtendedStateData()` is called to retrieve the POID string `s` of the batch. Using the `batchPoid` identifier stored in `s`, `LocateByIdentity()` finds the batch object and assigns it to the variable, `b`.

```
s : java::lang::String;
s := java com::ceira::oem::GetExtendedStateData
    (parent, 'batchPoid' ).oclAsType(java::lang::String);
b : com::cogility::poal::BusinessObjectsBatch;
b := java com::ceira::oem::LocateByIdentity(s)
    .oclAsType(com::cogility::poal::BusinessObjectsBatch)
```

To keep track of the sub-batch count, the `count` variable is recalled from the `parent` object.

```
cnt : java::lang::Integer;
cnt := java com::ceira::oem::GetExtendedStateData(parent, 'count')
    .oclAsType(java::lang::Integer);
```

Each pass through the ProcessBatch state will require a `sub` variable to hold the Subscriber object being processed. For each sub-batch in process, a message describing the sub-batch will be printed to standard output.

```
sub : BatchExample::MIM::Subscriber;
java com::cogility::Println('Attempting Batch #: '.concat(cnt));
```

The nextBatch action retrieves the next sub-batch and places it in a `curBatch` List, then iterates through the Subscriber objects in the List. It tests each Subscriber object's `age` attribute for a value greater than 21, and if the value is greater than 21, sets the `premium` attribute to true. It prints to the console the `subscriberId` and `age`. Also, those Subscriber objects that don't have a value for the `age` attribute will throw an exception, which will be handled in the `catch` block. If none of the Subscriber objects in the sub-batch cause an exception, the sub-batch will be committed to the database when the do activity action completes with `endtry`; See “[Transactions](#)” on page 111 for more information about transaction boundaries.

```
try
    curBatch : java::util::List;
```

```
curBatch := b.nextBatch();
cbIter : java::util::Iterator;
cbIter := curBatch.iterator();
loop cbIter.hasNext() do
    sub := cbIter.next().oclAsType(BatchExample::MIM::Subscriber);
    if sub.age > 21 then
        java
com::ceira::Println(sub.subscriberId.concat('\t').concat(sub.age));
        sub.premium := true;
    endif;
endloop;
```

If any of the Subscriber objects causes an exception, the catch block rolls back the transaction (see “[oclRollback](#)” on page 49 of the guide, *Using Actions in Cogility Studio*) including all database modifications for the current sub-batch.

```
catch (java::lang::Throwable ae)
java com::cogility::Println('##### Exception was caught - rolling back
transaction.');
    oclRollback;
```

At this point the batch object refers to a sub-batch which has Subscriber objects in error. When the transaction for that sub-batch was rolled back, the batch object itself, which holds its own state information as a persistent database object, has also been rolled back. To retrieve the batch object in its rolled-back state, you must explicitly locate the batch object by its POID with [LocateByIdentity](#) again before you can work with the sub-batch in error. Note that the sub-batch in error is no longer the *current* sub-batch (which is no longer defined), but the *next* sub-batch, according to the batch object’s state information.

```
b := java com::ceira::oem::LocateByIdentity(s)
    .oclAsType(com::cogility::poal::BusinessObjectsBatch);
```

The sub-batch is set aside for re-processing with [setAsideNextBatchAsErrored](#). When the sub-batch is set aside, each object in the sub-batch is placed in its own sub-batch (each new sub-batch has a batch size of 1). This is how the objects in error are isolated; the new sub-batches are re-processed after the initial sub-batches (in this example, those with a batch size of 5). After all of the initial sub-batches and all of the new sub-batches are processed, there will be set aside a number of new sub-batches each with an object in error. Notice that each Subscriber object in error has been tested twice; once in its original sub-batch along with the other Subscriber objects, then a second time in its own sub-batch.

```
    b.setAsideNextBatchAsErrored();
endtry;
```

The Done and NotDone transitions control the sub-batch processing. Each time a sub-batch passes through the ProcessBatch state, the guard on the NotDone transition checks the batch for more sub-batches. As long as [return b.hasMoreInstances\(\)](#) evaluates to true, the guard condition evaluates to true, allowing the transition to take place and the effect action to be executed.

```
s : java::lang::String;
s := java com::ceira::oem::GetExtendedStateData(parent, 'batchPoid' )
    .oclAsType(java::lang::String);
b : com::cogility::poal::BusinessObjectsBatch;
b := java com::ceira::oem::LocateByIdentity(s)
    .oclAsType(com::cogility::poal::BusinessObjectsBatch);
return b.hasMoreInstances();
```

The effect action of the NotDone transition increments the sub-batch counter; it executes when the NotDone guard condition evaluates to true.

```
cnt : java::lang::Integer;
cnt := java com::ceira::oem::GetExtendedStateData(parent, 'count')
        .oclAsType(java::lang::Integer);
cnt := cnt + 1;
java com::ceira::oem::SetExtendedStateData(parent, 'count', cnt);
```

As specified, the target of the NotDone transition is the ProcessBatch state again, and process flow returns to the start of the ProcessBatch state's do activity, thus opening another transaction to the database for the next sub-batch. When all of the sub-batches have either been committed or set aside, the NotDone guard condition evaluates to false because `b.hasMoreInstances()` evaluates to false and the transition will not take place.

Conversely, the Done transition's guard condition will evaluate to true. Once again, `b.hasMoreInstances()` is called and evaluates to false. The `not` keyword negates this result so that the condition can evaluate to true.

```
s : java::lang::String;
s := java com::ceira::oem::GetExtendedStateData(parent, 'batchPoid' )
        .oclAsType(java::lang::String);

b : com::cogility::poal::BusinessObjectsBatch;
b := java com::ceira::oem::LocateByIdentity(s)
        .oclAsType(com::cogility::poal::BusinessObjectsBatch);

return not b.hasMoreInstances();
```

Now the process can deal with each of those sub-batches that contains the single Subscriber object in error. Again the action must declare a variable to hold the batch object, then locate it by the `batchPoid` on the parent. It also needs a variable to hold the Subscriber object.

```
s : java::lang::String;
s := java com::ceira::oem::GetExtendedStateData(parent, 'batchPoid' )
        .oclAsType(java::lang::String);
b : com::cogility::poal::BusinessObjectsBatch;
b := java com::ceira::oem::LocateByIdentity(s)
        .oclAsType(com::cogility::poal::BusinessObjectsBatch);

sub : BatchExample::MIM::Subscriber;

if (b.hasErroredInstances()) then
    erroredList : java::util::List;
    iter : java::util::Iterator;
    erroredList := b.getErroredInstances();
    java com::ceira::Println('There are '.concat(erroredList.size()
        .concat(' errored instances')));
    iter := erroredList.iterator();
    loop (iter.hasNext()) do
        sub := iter.next().oclAsType(BatchExample::MIM::Subscriber);
        java com::ceira::Println('Could not process subscriber:
            '.concat(sub.subscriberId));
    endloop;
endif;
```

The action creates a List of errored instances and retrieves them with `b.getErroredInstances`. It then prints the number of errored instances, iterates through the list, and prints the `subscriberId` of each errored instance. Process flow continues to the DeleteBatch state.

DeleteBatch

The final state of the FindSubscriber state machine, DeleteBatch, retrieves the batch object and deletes it.

```
s : java::lang::String;
s := java com::ceira::oem::GetExtendedStateData(parent, 'batchPoid')
    .oclAsType(java::lang::String);

b : com::cogility::poal::BusinessObjectsBatch;
b := java com::ceira::oem::LocateByIdentity(s)
    .oclAsType(com::cogility::poal::BusinessObjectsBatch);

b.delete();
```

Running the batch processing example

You can run this sample model with data created by an included action script.

To run the batch processing sample model:

1. In Cogility Modeler, import the model,
`%DCHOME%\Examples\Behavior\BatchProcess\BatchExample_v1.cog`.
See “[Message definitions](#)” on page 108 for loading instructions.
2. Turn over and push the model.
See “[Pushing the model into execution](#)” on page 273.
3. Run Cogility Action Pad.
See “[Running Cogility Action Pad](#)” on page 80 of the guide, *Model Deployment & Execution in Cogility Studio*. Cogility Action Pad opens in a new window.
4. In Cogility Action Pad, open the script file,
`%DCHOME%\Examples\Behavior\BatchProcess\CreateBatchData.ocl`
See “[Opening FilePads](#)” on page 82 of the guide, *Model Deployment & Execution in Cogility Studio*.
5. Run the script.
See “[Running action semantics in an action pad](#)” on page 90 of the guide, *Model Deployment & Execution in Cogility Studio*.
6. If the application server is not running, start it.
See “[Starting the application server](#)” on page 270.
7. Run Cogility Insight.
See “[Running Cogility Insight](#)” on page 100 of the guide, *Model Deployment & Execution in Cogility Studio*.
 - a. Under Roles, click **Business Objects** and **View Data**.
 - b. From the drop-down menu, select **MyDOMSchema.MIM.Subscriber** and click the **Next** button .
 - c. Click the **Next** button again to show the data produced.

Notice that the data includes 100 Subscriber records. Four of the records lack an age value, and these will generate exceptions during the batch processing.

8. Run Cogility Message Traffic Viewer.

See “[Running Cogility Message Traffic Viewer](#)” on page 129 of the guide, *Model Deployment & Execution in Cogility Studio*.

- a.** Click **Load**, navigate to the %DCHOME%\Examples\BatchProcess directory, select **BatchExample.dat** and click **Open**.

- b.** With **FindSubscriberTopic** selected, click **Re-Publish On Subject**.

The FindSubscriber state machine executes, and its messages are printed to the application server’s standard output.

9. Return to Cogility Insight.

- a.** Under **Roles**, click **Business Objects** and **View Data**.

- b.** From the drop-down menu, select **MyDOMSchema.MIM.Subscriber** and click the **Next** button .

- c.** Click the **Next** button again to show the data produced.

10. In Cogility Action Pad, open the script,

%DCHOME%\Examples\Behavior\BatchProcess\DeleteBatchData.ocl.

11. Run the script.

See “[Running action semantics in an action pad](#)” on page 90 of the guide, *Model Deployment & Execution in Cogility Studio*.

12. Close the Cogility Action Pad window.



6

Transformation model

A transformation is required when one or more domains are represented in a model, when each of these domains needs to share information with the other domains and when the domains represent the data differently. Transforming data for use between two or more domains requires an intermediary, a hub that can represent a given set of data to each of the domains. In Cogility Modeler, you represent each domain with its own distinct information model (see “[Distinct information model \(DIM\)](#)” on page 30), and the intermediary hub with the master information model (see “[Master information model \(MIM\)](#)” on page 29).

To complete a transformation from one domain to the others, the data is first transformed between the domain’s DIM and the MIM, then between the MIM and the DIMs for each of the other domains. A transformation can be as simple as a copy of data from one field to another, or the transformation can contain complex logic that manipulates the data as it is moved.

For example, A model includes an Order Entry DIM with data from an order entry system, and a MIM that holds the enterprise-wide business process model. In the Order Entry DIM and the MIM there is a Customer class. Although both Customer classes hold similar information, the data formats as well as the number of features (or attributes) in each class are different. The order entry information from the Order Entry DIM customer class needs to be transformed for use in the MIM.

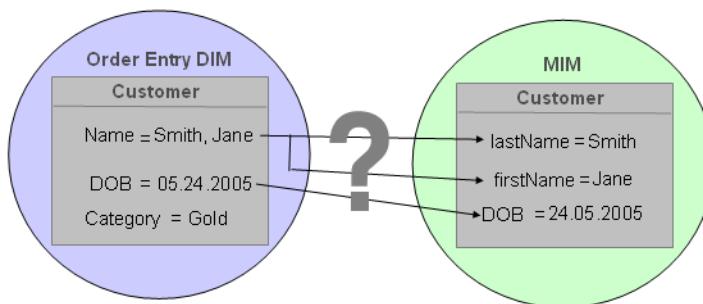


Figure 6-1: Disparities between data formats creates the need for transformations

In this example, the Name attribute of the Customer class on the Order Entry DIM is a single string containing the last name and first name, separated by a space and a comma. Its counterpart, the Customer class on the MIM, has two attributes for the name, a lastName and a firstName. The data transformation from the DIM to the MIM must parse the Name attribute and populate the lastName and firstName attributes. In the reverse, where the data is transformed from the MIM to the DIM, the lastName and firstName attribute values must be concatenated and the resulting string used to populate the Name attribute.

Likewise, the DIM Customer DOB attribute holds a date in a MM.DD.YYY format, while the MIM Customer DOB has the format, DD.MM.YYYY. The value for the DIM Customer DOB must be transformed for the MIM Customer DOB format.

To move data from a DIM to the MIM transformations use a data object set (DOS) to hold the DIM and MIM data during the transformation process, as shown in the figure below. For additional information on creating and using a DOS, see “[Data object set \(DOS\)](#)” on page 46.

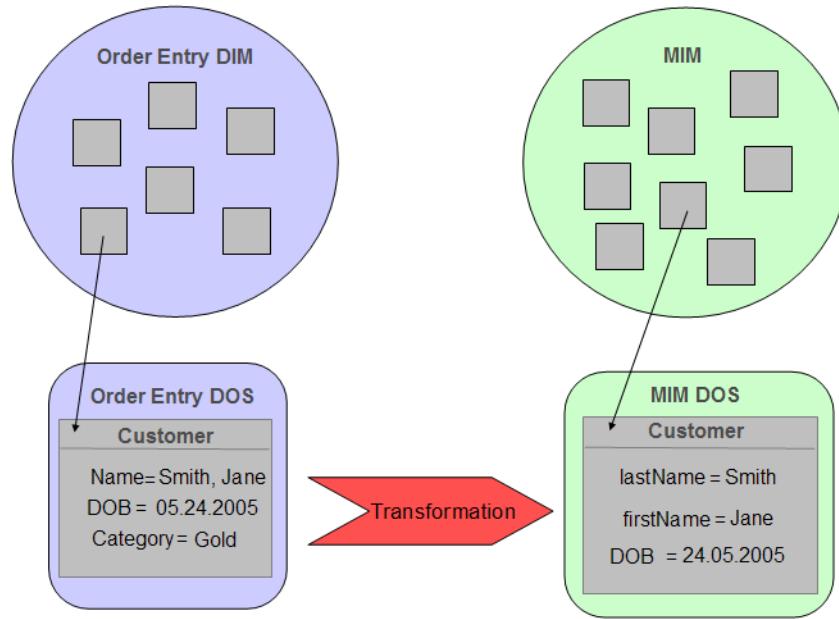


Figure 6-2: A transformation takes data from a DOS

Cogility Modeler’s data transformation artifacts make it possible to perform everything from simple field copy transformations to complex manipulation of data from one source to another. The following points apply to all transformations:

- Every transformation can have only one input DOS (Data Object Set) and one output DOS. The DOS describes the class data of an object participating in a transformation.
- The input DOS and the output DOS cannot be the same DOS.
- A transformation cannot execute on its own: it must be assigned to a transformation chain or a simple state in a state machine.

Transformation artifacts may be used in state machines (see “[State machine](#)” on page 112), or several transformation artifacts may be linked together in transformation chains (see “[Transformation Chain](#)” on page 163). Two top level transformation artifacts provide data transformation functionality:

- An opaque transformation map is an atomic artifact that defines transformations using actions. The actions use fully qualified names to refer to the objects participating in the transformation, thereby limiting reusability. The opaque transformation map is generally used for simple transformations. See “[Opaque transformation map](#)” on page 146.
- A transformation map contains a hierarchical order of reusable transformation artifacts that provide visibility and traceability. The lowest level artifacts contain actions to define the data transformations. See “[Transformation map](#)” on page 149.

Opaque transformation map

An opaque transformation map is a model artifact that defines the data mapping between the class features (attributes) in one input DOS and the class features in one output DOS. Unlike a transformation map (see “[Transformation map](#)” on page 149), the opaque transformation map may

not be further divided into other transformation artifacts. Instead, it is an atomic artifact that defines a transformation using actions. Because the instructions are coded in actions, an opaque transformation is generally used for simple transformations and usually is not reused. The actions refer to the objects participating in the transformation with fully qualified names, thereby limiting reusability.

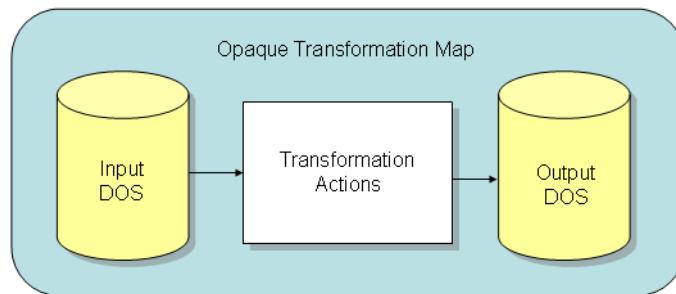


Figure 6-3: Opaque transformation map illustrated

An opaque transformation map cannot execute on its own: it must be assigned to a transformation chain (see “[Transformation Chain](#)” on page 163) or a simple state in a state machine (see “[State machine](#)” on page 112).

An opaque transformation requires an input DOS, an output DOS, a DOS populator for the output DOS and the transformation actions. In both the DOS populator and the transformation, the actions use two reserved keywords, `input` and `output` as known variables to specify the input DOS and the output DOS. As you write the transformation actions you must use the fully-qualified names to identify the features (attributes) that are being transformed. For example, the following actions declare that the specified output DOS, Account class, `firstName` feature (attribute) is set to the value of the input DOS, MIMCustomer class, `firstName` feature (attribute):

```
output.Account.firstName := input.MIMCustomer.firstName
```

[Figure 6-4](#) and [Figure 6-5](#) on page 148 show a completed opaque transformation map definition.

The DOS populator in [Figure 6-4](#) declares a variable, `acct`, of the Account type in the BillingDIM. The action then looks for an instance of an Account with an `accountID` that matches the `billingID` of the input DOS MIMCustomer instance and assigns it to the `acct` variable. If an existing instance cannot be found, a new instance is created. The value of the variable `acct` is then assigned to the output DOS Account class.

	Transformation Actions	Usage	Schema Tags	Consistency	Comments	Editions
	Transformation Map					
Dos Populator	<pre> acct : DOMCE::BillingDIM::Account; acct := locate DOMCE::BillingDIM::Account(accountID:input.MIMCustomer.billingID); if acct = null then acct := new DOMCE::BillingDIM::Account(accountID:input.MIMCustomer.billingID); endif; </pre>					

Figure 6-4: Opaque transformation map DOS populator

The transformation actions assign the output DOS Account class features (attributes) values to the input DOS MIMCustomer class features (attributes).

The transformation actions assign the output DOS Account class features (attributes) values to the input DOS MIMCustomer class features (attributes).

The transformation actions assign the output DOS Account class features (attributes) values to the input DOS MIMCustomer class features (attributes).

Usage	Schema Tags	Consistency	Comments	Editions
Transformation Map	Transformation Actions POP	Transformation Actions		
Transformation	<pre>output.Account.firstName := input.MIMCustomer.firstName; output.Account.lastName := input.MIMCustomer.lastName; output.Account.address := input.MIMCustomer.address; output.Account.accountID := input.MIMCustomer.billingID;</pre>			

Figure 6-5: Opaque transformation map actions definition

Creating an opaque transformation map

To create an opaque transformation:

1. In Cogility Modeler's tree view, select the model container or package, click the **Add** button  and select **Add an Opaque Transformation Map**.
An untitled opaque transformation map displays in the tree view, and the opaque transformation editor displays in the content view.
2. Define the opaque transformation map.
See “[Defining an opaque transformation map](#)” on page 148.

Creating an opaque transformation map in the Data Transformation Editor

To create an opaque transformation map:

1. In the Data Transformation Editor, select a DOS from the right **DOS** panel and one from the left **DOS** panel.
2. In the **Transforms** panel, click the **create a new transform** button .
3. Select **Create a new OpaqueTransformationMap**.
4. In the **Transformation Direction** dialog, select the **>> Left to Right >>** or **<=> Right to Left <=>** button to determine the direction of the transformation (input to output).
Once you have selected a direction, a directional arrow  appears below the **Transforms** panel so you can see the direction of flow when you are viewing a transformation. The arrow pertains to the transform selected in the **Transforms** panel.
5. Click the edit icon  to define the opaque transformation map.
See “[Defining an opaque transformation map](#)” on page 148.

Defining an opaque transformation map

To define an opaque transformation map:

1. In the opaque transformation editor, under the **Opaque Transformation** tab, in the **Name** field, enter a name for the transformation.
The name you enter must be unique within the model container.
2. Set the input and output DOS.
See “[Data object set \(DOS\)](#)” on page 46 for more information.

3. In the **Transformation Actions** POP tab, in the **DOS Populator** field, define the DOS populator.
See “[DOS populator](#)” on page 99.
4. In the **Transformation** field, enter the transformation actions.

Transformation map

A transformation map defines the data transformation mapping from an input DOS to an output DOS. A transformation map contains a hierarchical order of reusable transformation artifacts that provide visibility and traceability. The lowest level artifacts contain actions that define the data transformation instructions. These artifacts are reusable in other transformations and provide much greater visibility into the transformation than the atomic opaque transformation map (see “[Opaque transformation map](#)” on page 146).

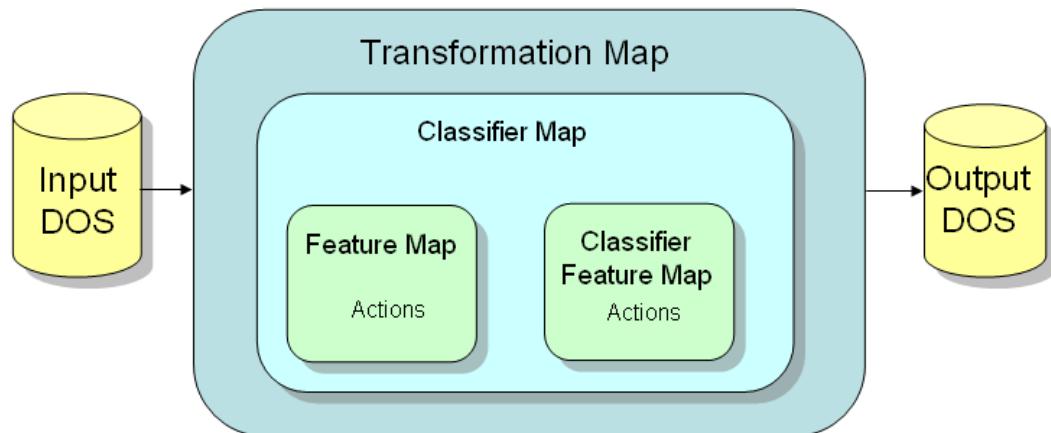


Figure 6-6: Transformation map illustrated

A transformation map hierarchy generally contains the following components:

- **Transformation map** - A transformation map is a container for other transformation artifacts, its definition specifies the input DOS and output DOS for the transformation. It also uses actions to locate the object specified in the output DOS to be updated as a result of the transformation action.
- **Classifier map** - A classifier map specifies the input classes and the output classes that participate in the transformation. The transformation map may include several classifier maps.
- **Feature map** - A feature map specifies transformation instructions at the feature level and specifies the feature type (string, double, etc.). A classifier map may include several feature maps, each defined with a feature map usage. A feature map is reusable for any feature of the applicable type. It can be reused in multiple classifier maps and can be used multiple times within the same classifier map for different features.
- **Classifier feature map** - A classifier feature map allows you to transform a classifier (a class or classes) to a feature, or to transform a feature to a classifier. A classifier map may include several classifier feature maps, each defined with a classifier feature map usage. A classifier feature map can be reused although its reuse is more limited because it specifies either specific features or specific classes.

The figure below shows the hierarchy of a transformation map.

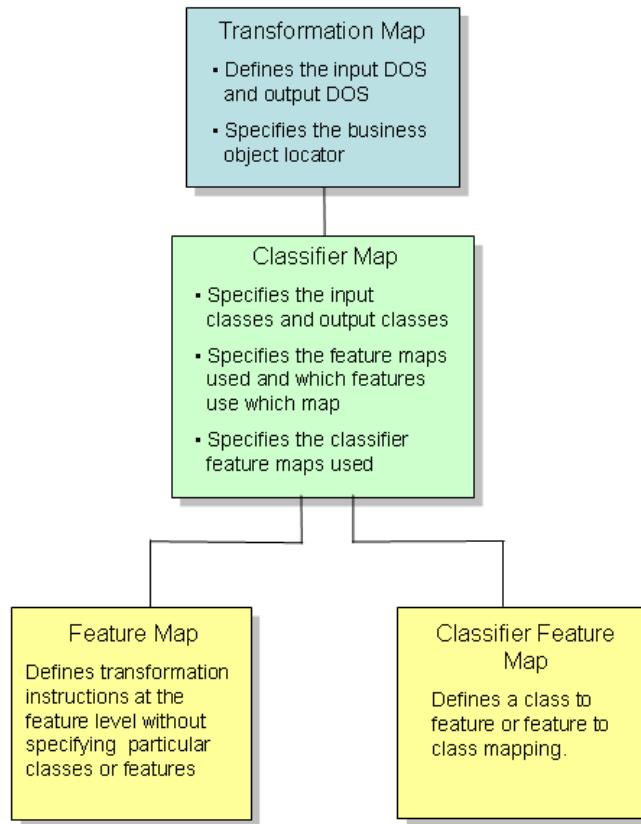


Figure 6-7: Transformation map hierarchy

At its most simple, a transformation map implementation can consist solely of the transformation map definition and actions that define the transformation instructions. In effect, this would be the same as using a opaque transformation map, and it is recommended that an opaque transformation map be used in such situations. See “[Opaque transformation map](#)” on page 146.

The next most simple implementation of a transformation map is a transformation map with a classifier map that is not further broken down. The classifier map can include actions that specify transformations between classes. Doing so reduces granularity and reuseability for that classifier map.

A full transformation map implementation uses one or more classifier maps that specify feature maps or classifier feature maps. Using feature maps and classifier features maps increases the granularity of the classifier map and provides increased visibility and traceability into the details of the transformation generally. The feature maps and classifier feature maps may be reused in other classifier maps and in other transformations.

The complexity and atomicity of a transformation are determined by the level at which the transformation actions are defined. Proceeding from left to right, simple to complex, and least atomic to most atomic, three implementations of a transformation map are illustrated below.

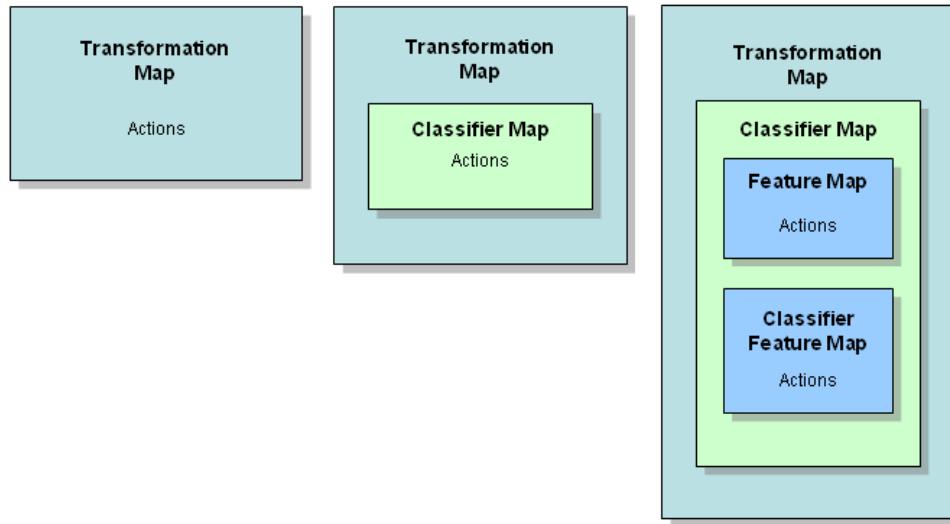


Figure 6-8: Three levels of transformation map complexity

A transformation map may use a combination of classifier maps, feature maps, classifier feature maps and actions to define a transformation. In the figure below, the transformation map shows the various levels of atomicity and complexity. It includes three classes in the input DOS and four classes in the output DOS. Class1 has two features that map directly to the two features of ClassA using a classifier map and two feature maps. Class2 must be mapped to ClassB and ClassC; two of features of Class2 must be mapped to a single classC using a classifier feature map. Notice that the classifier map Cmap2 references both Class2 and ClassC while the classifier feature map CFmap references

Class2, its feature5 and feature6 and featureC1. The last class, Class3 maps directly to ClassD using actions; this is the simplest, least atomic implementation of a classifier map.

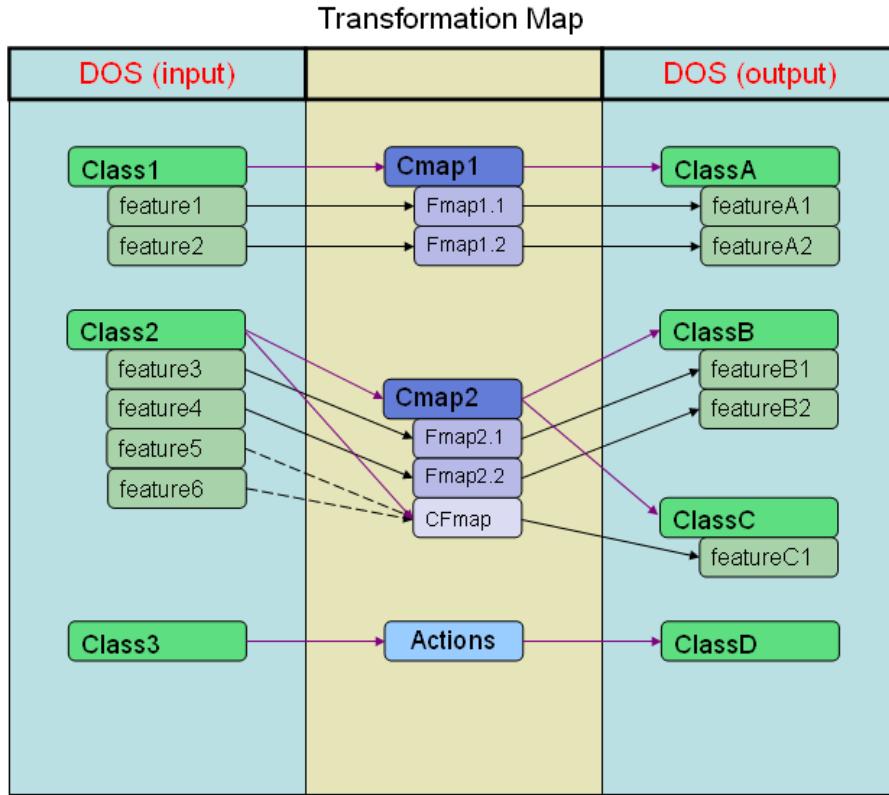


Figure 6-9: Transformation map showing various levels of atomicity.

Creating a transformation map

To create a transformation map:

1. In Cogility Modeler's tree view, select the model container or package, click the **add** button  and select **Add TransformationMap**.
An untitled transformation map displays in the tree view, and the transformation map editor displays in the content view.
2. Define the transformation map.
See “Defining a transformation map” on page 153.

Creating a transformation map in the Data Transformation Editor

To define a transformation map:

1. In the Data Transformation Editor, select a DOS from the left **DOS** panel and the right **DOS** panel.
2. In the **Transforms** pane, click the **create a new transform** button .
3. Select **Create a new TransformationMap**.
4. In the **Transformation Direction** dialog, select the **>> Left to Right >>** or **<< Right to Left <<** button to determine the direction of the transformation (input to output).

Once you have selected a direction, a directional arrow  appears below the Transforms panel so you can see the direction of flow when you are viewing a transformation. The arrow pertains to the transform selected in the Transforms panel.

5. Click the edit icon  to define the transformation map.

See “[Defining a transformation map](#)” on page 153.

Creating a transformation map in a classifier map

To create a transformation map in a classifier map:

1. In the classifier map editor, in the **Classifier Map** tab, above the **Transformation Maps** field, click **New**.
2. In the dialog, enter a name for the transformation map and click **OK**.
3. To define the transformation map, click **Edit**.

See “[Defining a transformation map](#)” on page 153.

Adding a classifier map to a transformation map

To add a classifier map to a transformation map in the classifier map:

1. In the classifier map editor, in the **Classifier Map** tab, above the **Transformation Maps** field, click **Add**.
2. In the **Select Object(s)** dialog, select the transformation map and click **OK**.
3. To define the transformation map, click **Edit**.

See “[Defining a transformation map](#)” on page 153.

Defining a transformation map

To define a transformation map:

1. In the content view, under the **Classifier Map Definition** tab, in the **Name** field, enter a name for your map.
The name is updated in the tree view also.
2. Set the input DOS and the output DOS.
See “[Data object set \(DOS\)](#)” on page 46 and “[Data object set \(DOS\)](#)” on page 46.
3. Click the **Transformation Actions** tab, and in the **DOS Populator** field, define the DOS populator.
See “[DOS populator](#)” on page 99.
4. In the **Pre Transformation or Post Transformation** field, enter any pre-transformation and post-transformation actions. These are optional.
5. In the **Classifier Maps** tab, create or add the classifier maps.
See “[Classifier map](#)” on page 153 and .

Classifier map

The classifier map specifies the input classes and output classes used in a transformation map. The classes available for input and output are determined by the input DOS and output DOS specified in the transformation map. When you define a classifier map, you specify the feature maps and/or classifier feature maps used by the classifier map. You can also specify a transformation directly in

the classifier map using actions, or you can specify a combination of feature maps, classifier feature maps, and actions.

Creating a classifier map

To create a classifier map:

1. In Cogility Modeler's tree view, select the model container or package, click the **add** button  and select **Add ClassifierMap**.

An untitled classifier map displays in the tree view, and the classifier map editor displays in the content view.

2. Define the classifier map.

See “[Defining a classifier map](#)” on page 155.

Adding a class to a classifier map

To add classes to a classifier map:

1. In the classifier map editor, in the **Classifier Map** tab, above the **Input Classes** field, click **Add**.

2. Select the classes you want to use for input and click **OK**.

Since a transformation can only have one input DOS, and a DOS can only contain classes from a single DIM, or from the MIM, be sure the classes you select exist in the same DIM, or in the MIM. The input DOS used by a classifier map is specified in the owning transformation map.

3. Above the **Output Classes** field, click **Add**.

4. Select the classes you want to use for input and click **OK**.

Since a transformation can only have one output DOS, and a DOS can only contain classes from a single DIM, or from the MIM, be sure the classes you select exist in the same DIM, or in the MIM. The output DOS used by a classifier map is specified in the owning transformation map.

5. To edit the classes in either field, select the class and click **Edit**. The class editor opens.

See “[Defining a class](#)” on page 37 for information on using the class editor.

Creating a classifier map in a transformation map

To create a classifier map in a transformation map:

1. In the transformation map editor, in the **Classifier Maps** tab, click **New**.

2. In the dialog, enter a name for the classifier map and click **OK**.

3. To define the classifier map, click **Edit**.

See “[Defining a classifier map](#)” on page 155.

Creating a classifier map in the Data Transformation Editor

To create a classifier map:

1. In the Data Transformation Editor, in the **Transforms** panel, select the transformation map.

2. In the left **DOS** panel, open a DOS tree and select one or more classes. Do the same in the right **DOS** panel.

3. In the **Transforms** pane, click the **create a new transform** button .

4. Select **Create a new ClassifierMap**.

5. Click the edit icon  to define the classifier map.
See “Defining a classifier map” on page 155.

Adding a classifier map to a transformation map

To add a classifier map to a transformation map:

1. In the transformation map editor, in the **Classifier Maps** tab, click **Add**.
2. In the **Select Object(s)** dialog, select the classifier map and click **OK**.
3. To define the classifier map, click **Edit**.
See “Defining a classifier map” on page 155.

Setting a classifier map in the Data Transformation Editor

To reuse a classifier map:

1. In the Data Transformation Editor, in the **Transforms** panel, select the transformation map.
2. Open a DOS tree and select one or more classes from the left **DOS** panel. Do the same in the right **DOS** panel.
3. In the **Transforms** panel, click the **reuse an existing transform** button .
4. Select **Reuse an existing ClassifierMap**.
5. In the **Select Object(s)** dialog, select the classifier map and click **OK**.
If there are no existing classifier maps that match your input and output criteria, a message appears to that effect. Click **OK** to close the message. The classifier map appears in the transformation map tree in the **Transforms** panel.
6. Click the edit icon  to define the classifier map.
See “Defining a classifier map” on page 155.

Defining a classifier map

To define a classifier map:

1. In the **Classifier Map** tab, in the **Name** field, enter a name for the classifier map.
2. If necessary, create or add a transformation map.
See “Transformation map” on page 149. If you are creating this classifier map from within a transformation map, that transformation map appears in the field.
3. Above the **Input Classes** field, click **Add**.
4. In the **Select Object(s)** dialog, select the classes and click **OK**.
You may select multiple classes by holding down the Ctrl key while selecting. The classes available are those specified in the input DOS for the transformation map that owns this classifier map. If you have created the classifier map independent of any transformation map (and therefore any particular DOS), the input classes available include all of the classes in the model. You must select only classes from the same DIM or MIM; these must be in the same DOS and associated with the owning transformation map before the classifier map will work properly. The best practice is to define the transformation map with input and output DOS and associate it with this classifier map before selecting the input classes. See step 2, above.
5. Above the **Output Classes** field, click **Add**.
6. In the **Select Object(s)** dialog, select the classes and click **OK**.

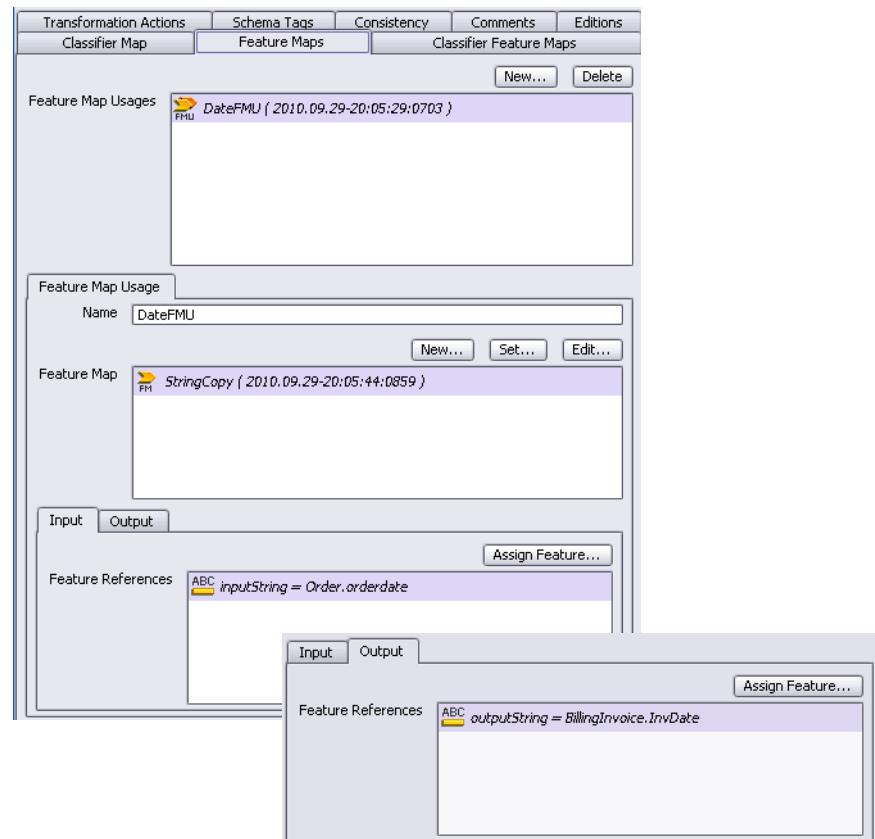
As with the input classes in step 4, above, the best practice is to define the transformation map with input and output DOS and associate it with this classifier map before selecting the output classes.

7. In the **Feature Maps** tab, define the feature map usages.
See “[Feature map usage](#)” on page 156 and “[Defining a feature map usage](#)” on page 157.
8. In the **Classifier Feature Maps** tab, define the classifier feature map usages.
See “[Classifier feature map usage](#)” on page 157.
9. In the **Transformation Actions** tab, in the **Pre Transformation or Post Transformation** field, define transformation actions as needed.
See the guide, *Using Actions in Cogility Studio* for help with actions.

Feature map usage

For a designated input feature reference and output feature reference, a feature map usage describes which feature map performs the transformation between the two references. The feature map usage assigns the input class feature to the feature map’s input feature reference, and the output class feature to the feature map’s output feature reference. Because this assignment happens in the feature map usage, not the feature map, different class features can use the same feature map.

The figure below shows a completed feature map usage definition. The DateFMU feature map usage assigns the Order.orderDate to the input feature reference and the BillingInvoice.InvDate to the output feature reference; the StringCopy feature map specifies the input feature reference to be transformed to the output feature reference.



Feature map usages are not reusable objects. A feature map usage belongs to a specific classifier map that has specific input classes and output classes. The feature map usages are inherently bound to these class features in the classifier map.

Defining a feature map usage

To define a feature map usage:

1. In the classifier map editor, in the **Feature Maps** tab, above the **Feature Map Usages** field, click **New**.
2. In the dialog, enter a name for the feature map usage and click **OK**.
The Feature Map Usage Detail tab appears.
3. Above the **Feature Map** field, create or set a feature map.
See “[Feature map](#)” on page 158. You must have defined a feature map with input and output feature references before continuing with these steps.
4. In the **Input** tab, click **Assign Feature**.
5. In the **Select Object(s)** dialog, select the input feature and click **OK**.
The dialog displays a list of features from the input class of the type specified by the input feature reference in the feature map. The input classes must be specified in the classifier map, and there must be features of the specified type in that class before the dialog can display the appropriate selection. See “[Defining a classifier map](#)” on page 155.
6. In the **Output** tab, click **Assign Feature**.
7. In the **Select Object(s)** dialog, select the output feature and click **OK**.
As with the input feature in step 5, above, the classifier map must define a feature of the type specified in the output feature reference of the feature map.

Classifier feature map usage

For a designated input classifier or feature reference and a designated output classifier or feature reference, a classifier feature map usage describes which classifier feature map performs the transformation between the two references. The classifier feature map usage assigns the input classifier or feature to the classifier feature map’s input classifier or feature reference, and the output classifier or feature to the feature map’s output classifier or feature reference.

When you define a classifier feature map usage for a classifier map, you specify how to map the classifier or feature reference to either an output feature (if the classifier is the input) or an input feature (if the classifier is the output).

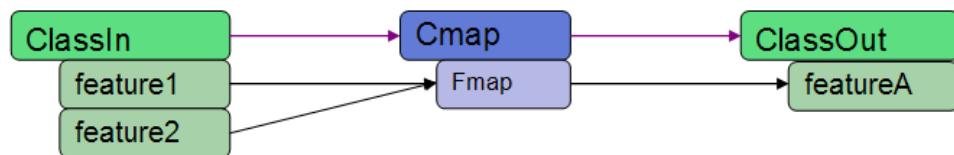
Defining a classifier feature map usage

To define a classifier feature map usage:

1. In the classifier map editor, in the **Classifier Feature Maps** tab, above the **Classifier Feature Map Usages** field, click **New**.
2. In the dialog, enter a name for the classifier feature map usage and click **OK**.
The Usage Definition tab appears.
3. Above the **Classifier Feature Map** field, create or set a classifier feature map.
See “[Classifier feature map](#)” on page 160.

Feature map

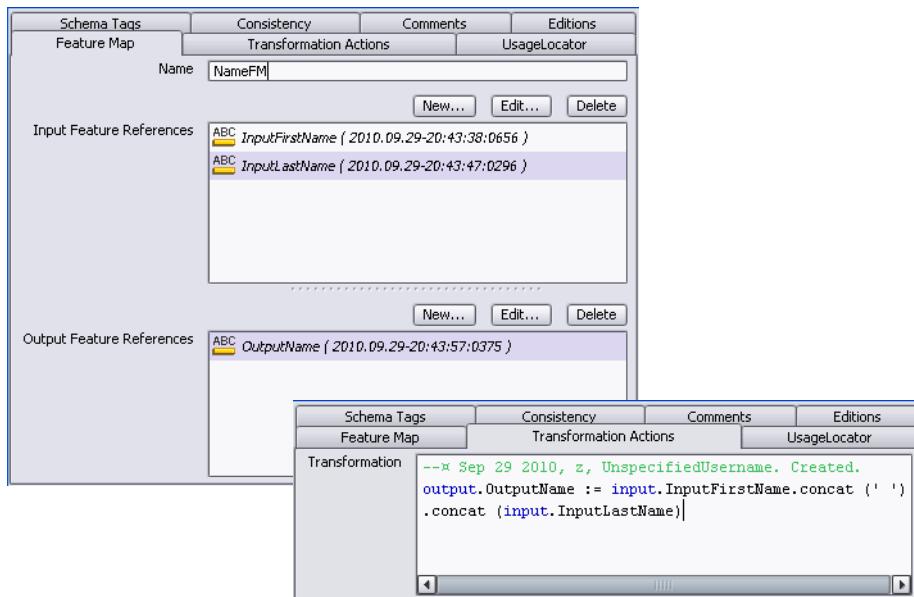
A feature map describes a data transformation between two class features. For example, two features 1 and 2, both of type String, need to be concatenated to form a single string value. A feature map would take two string values as input and produce a single string value as an output. A transformation action concatenates the strings and assigns the output value. This feature map may be reused any time two string type features are concatenated and output as a single value.



A feature map uses actions to perform the transformation. The feature map specifies an input feature reference and an output feature reference, each defined with a name and a type. The transformation action specifies how the input feature reference is transformed and then populates the output feature reference. The transformation action for a feature map can be as simple as a string copy, a concatenation or it can consist of many lines of code. Actions use the reserved keywords `input` and `output` to refer to the input feature reference and the output feature reference, respectively.

The feature map is not bound to a particular class feature and may be reused any time that the input feature is of the type specified by the feature reference and the output feature is of the type specified by the output feature reference. When the feature map is used in a classifier map, a feature map usage assigns the class feature to the feature reference for both input and output. See “[Feature map usage](#)” on page 156.

The example below shows a feature map definition for a concatenation transformation that specifies two input feature references and one output feature reference. The transformation action describes the concatenation. This feature map can be used any time two string input fields are concatenated and output as one string.



Creating a feature map

To create a feature map:

1. In Cogility Modeler's tree view, select the model container or package, click the **add** button  and select **Add FeatureMap**.

An untitled feature map displays in the tree view, and the feature map editor displays in the content view.

2. Define the feature map.

See “Defining a feature map” on page 160.

Creating a feature map in a feature map usage

To create a feature map in a feature map usage:

1. In the classifier map editor, in the **Feature Maps** tab.
2. In the **Feature Map Usage** tab, click **New**.
3. In the dialog, enter a name for the feature map and click **OK**.
4. To define the feature map, click **Edit**.

See “Defining a feature map” on page 160.

Creating a feature map in the Data Transformation Editor

To create a feature map:

1. In the Data Transformation Editor, in the **Transforms** panel, select the classifier map.
2. Open a DOS class and select one or more features from the left **DOS** panel. Do the same in the right **DOS** panel.
3. In the **Transforms** panel, click the **create a new transform** button .
4. Select **Create a new FeatureMap**.
5. Click the edit icon  to define the feature map.

See “Defining a feature map” on page 160.

Setting a feature map in a feature map usage

To set a feature map in a feature map usage:

1. In the classifier map editor, in the **Feature Maps** tab.
2. In the **Feature Map Usage** tab, click **Set**.
3. In the dialog, select the feature map and click **OK**.
4. To define the feature map, click **Edit**.

See “Defining a feature map” on page 160.

Setting a feature map in the Data Transformation Editor

To reuse a feature map:

1. In the Data Transformation Editor, in the **Transforms** panel, select the classifier map.
2. Open a DOS class and select one or more features from the left **DOS** panel. Do the same in the right **DOS** panel.
3. In the **Transforms** panel, click the **reuse an existing transform** button .
4. Select **Reuse an existing FeatureMap**.

5. Select the feature map you want to reuse and click **OK**.
6. Click the edit icon  to define the feature map.
See “[Defining a feature map](#)” on page 160.

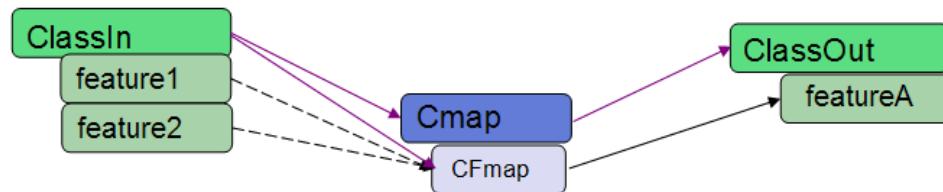
Defining a feature map

To define a feature map:

1. In the feature map editor, under the **Feature Map** tab, in the **Name** field, enter a name.
The name you enter must be unique within the model container.
2. Above the **Input Feature References** field, click **New**.
 - a. In the **NewFeatureReference** dialog, in the **Name** field, enter a name for the feature reference.
 - b. From the **Type** drop-down list, select the data type for the feature reference and click **OK**.
3. Above the **Output Feature References** field, click **New**.
 - a. In the **NewFeatureReference** dialog, in the **Name** field, enter a name for the feature reference.
 - b. From the **Type** drop-down list, select the data type for the feature reference.
 - c. Click **OK**.
4. In the **Transformation Actions** tab, in the **Transformation** field, define the transformation action.
See the guide, *Using Actions in Cogility Studio* for help with actions.

Classifier feature map

A classifier feature map transforms a classifier (a class or classes) to a feature, or it transforms a feature to a classifier. For example, a class with two features may be transformed into a class with one feature. This is similar to the example described in “[Feature map](#)” on page 158, except that the output is for a specific, atomic class rather than a feature for any class.



The figure above illustrates this example. Notice that using a classifier feature map limits its usability to specific output or input class types.

When you define a classifier feature map, you indicate whether the classifiers are the input to the map. If the classifiers are the input, then the transformation is a class to feature transformation. If you do not specify that the classifiers are input, then the transformation is a feature to class transformation by default. Although a classifier feature map is also reusable, its reusability is more limited than the feature map, since the classifiers are specified in the definition.

The example below shows a classifier feature map definition for a concatenation transformation that specifies a class as the input and one output feature reference. The transformation actions specify which input class features are concatenated and produced as a single output value. The reserved

keywords `input` and `output` refer to the input feature reference and the output feature reference, respectively.

This classifier feature map can be used whenever this specific input class is transformed into a feature of the type specified by the feature reference (a String). When the classifier feature map is used in a classifier map, a classifier feature map usage assigns the output feature reference value to an output feature. See “[Classifier feature map usage](#)” on page 157.

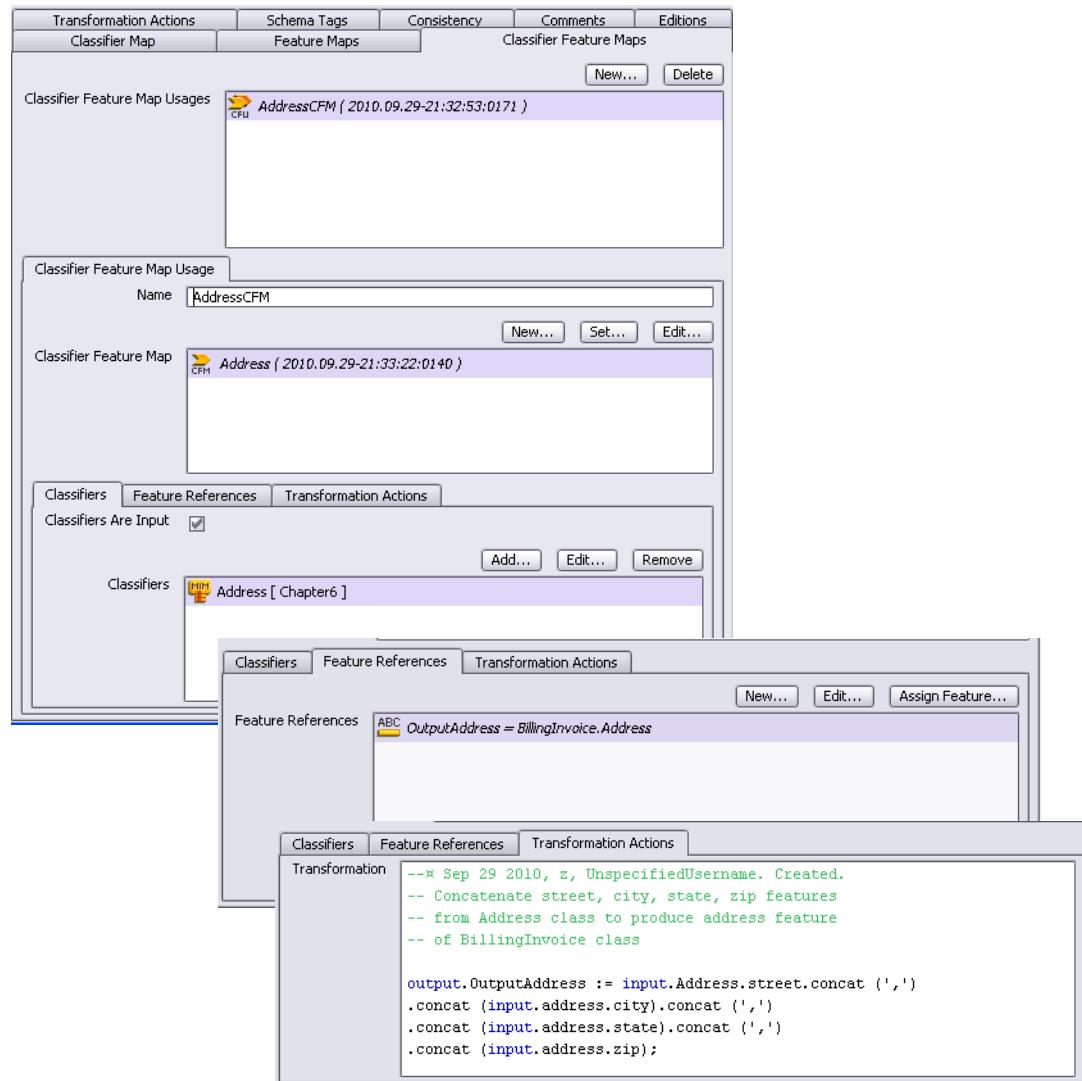


Figure 6-10: Classifier feature map defined

To reuse an existing classifier map, the classifiers and transformation actions defined in the classifier feature map must be the same as the classifiers and actions required for the designated usage. Deviating from either of these specifications in the classifier feature map could cause errors if the classifier feature map is used for different classifiers and when the transformation actions are incompatible. The feature reference assignment is specific to the usage, so this cannot be specified differently for each classifier feature map usage.

Creating a classifier feature map

To create a classifier feature map:

1. In Cogility Modeler's tree view, select the model container or package, click the **add** button  and select **Add ClassifierFeatureMap**.
An untitled classifier feature map displays in the tree view, and the classifier feature map editor displays in the content view.
2. Define the classifier feature map.
See “[Defining a classifier feature map](#)” on page 163.

Creating a classifier feature map in a classifier feature map usage

To create a classifier feature map in a classifier feature map usage:

1. In the classifier map editor, in the **Classifier Feature Maps** tab.
2. In the **Classifier Feature Map Usage** tab, click **New**.
3. In the dialog, enter a name for the classifier feature map and click **OK**.
4. To define the classifier feature map, click **Edit**.

See “[Defining a classifier feature map](#)” on page 163.

Creating a classifier feature map in the Data Transformation Editor

To create a classifier feature map:

1. In the Data Transformation Editor, in the **Transforms** panel, select the classifier map.
2. Open a DOS class and select a feature or a class from the left **DOS** panel. Do the same in the right **DOS** panel. If you select a class in one panel, you must select a feature in the other panel.
3. In the **Transforms** panel, click the **create a new transform** button .
4. Select **Create a new ClassifierFeatureMap**.
5. Click the edit icon  to define the classifier feature map.

See “[Defining a classifier feature map](#)” on page 163.

Setting a classifier feature map in a classifier feature map usage

To set a classifier feature map in a classifier feature map usage:

1. In the classifier map editor, in the **Classifier Feature Maps** tab.
2. In the **Classifier Feature Map Usage** tab, click **Set**.
3. In the dialog, select the classifier feature map and click **OK**.
4. To define the classifier feature map, click **Edit**.

See “[Defining a classifier feature map](#)” on page 163.

Setting a classifier feature map in the Data Transformation Editor

To reuse a classifier feature map:

1. In the Data Transformation Editor, in the **Transforms** panel, select the classifier map.
2. Open a DOS class and select a feature or a class from the left **DOS** panel. Do the same in the right **DOS** panel. If you select a class in one panel, you must select a feature in the other panel.

3. In the **Transforms** pane, click the **reuse an existing transform** button .
 4. Select **Reuse an existing ClassifierFeatureMap**.
 5. Select the classifier feature map you want to reuse and click **OK**.
 6. Click the edit icon  to define the classifier feature map.
- See “Defining a classifier feature map” on page 163.

Defining a classifier feature map

To define a classifier feature map:

1. In the classifier feature map editor, under the **Classifier Feature Map** tab, in the **Name** field, enter a name.
The name you enter must be unique within the model container.
 2. If you are transforming a class into a feature, check the **Classifiers Are Input** checkbox.
If you are transforming features into a class, leave the **Classifiers Are Input** checkbox unchecked.
 3. Above the **Classifiers** field, click **Add**.
 4. In the **Select Object(s)** dialog, select the classes and click **OK**.
 5. Above the **Feature References** field, click **New**.
 - a. In the **NewFeatureReference** dialog, in the **Name** field, enter a name for the feature reference.
 - b. From the **Type** drop-down list, select the data type for the feature reference, and click **OK**.
 6. In the **Transformation Actions** tab, in the **Transformation** field, define the transformation action.
- See the guide, *Using Actions in Cogility Studio* for help with actions.

Transformation Chain

The opaque transformation maps, transformation maps and their component artifacts, collectively referred to as transformation units, may be linked together in transformation chains. This is one of two ways to execute a transformation. The other is by embedding transformation units, including a transformation chain, in a simple state of state machine. See “Transformation unit” on page 124.

Transformation chains let you describe a reusable data path between data object sets. Different transformation chains can use the same transformation units, and the different transformation chains can be designed for different data paths: from a DIM to a MIM or from a MIM to a DIM. There are two types of transformation chains:

- **Serial transformation chain** - contains one or more transformation units that are processed in sequential order. Each transformation unit in the chain is dependant on the preceding transformation unit.
- **Parallel transformation chain** - contains one or more transformation units that are processed at the same time. No transformation unit is dependant on any other transformation unit in the transformation chain.

A complete serial or parallel transformation chain definition includes:

- An M2D that specifies the incoming message to DOS conversion. A M2D is not necessary if the transformation chain is embedded in a state machine. See “Transformation chain M2D” on page 165.
- One or more transformation units.

- One or more D2Ms that specify the outgoing DOS to message conversions. [“Transformation chain D2M” on page 166](#).

Rules for transformation chains

When adding transformation units to a transformation chain the following rules are enforced by Cogility Modeler, and transformation chains or transformation units that do not comply will be shown as inconsistent. The model will not push into execution until all inconsistencies are resolved.

- The output DOS of each transformation unit in a serial transformation chain must be the input DOS for the next transformation unit.
- You can use each DOS only once in the chain. For example, the output DOS (A in the figure below) of a transformation unit cannot be the same DOS as the input DOS of a previous transformation unit. This applies to both serial and parallel transformation chains.

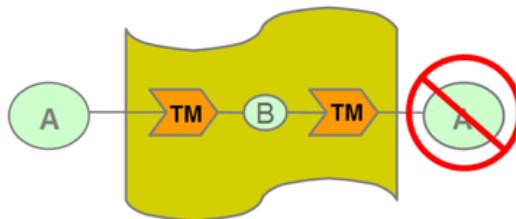


Figure 6-11: The input DOS may not be an output DOS in a transformation chain

- In a parallel transformation chain, each output DOS must be a different DOS. Multiple transformation units cannot use the same output DOS (B in the figure below).

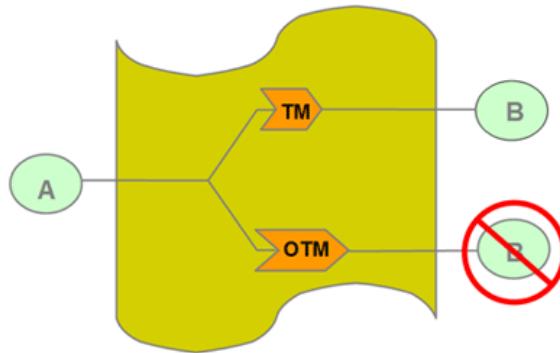


Figure 6-12: In a parallel transformation chain, an output DOS may be used only once

- The input DOS for a single parallel transformation chain, or for the first parallel transformation chain in a set of nested transformation chains, must be the input DOS for each transformation unit in the parallel transformation chain.
- In a serial transformation chain, if a parallel transformation chain is followed by another parallel transformation chain, the output DOS(s) of the first parallel transformation chain must

be the input DOS(s) of the following parallel transformation chain. In this case, the second parallel transformation chain can have multiple DOS inputs.

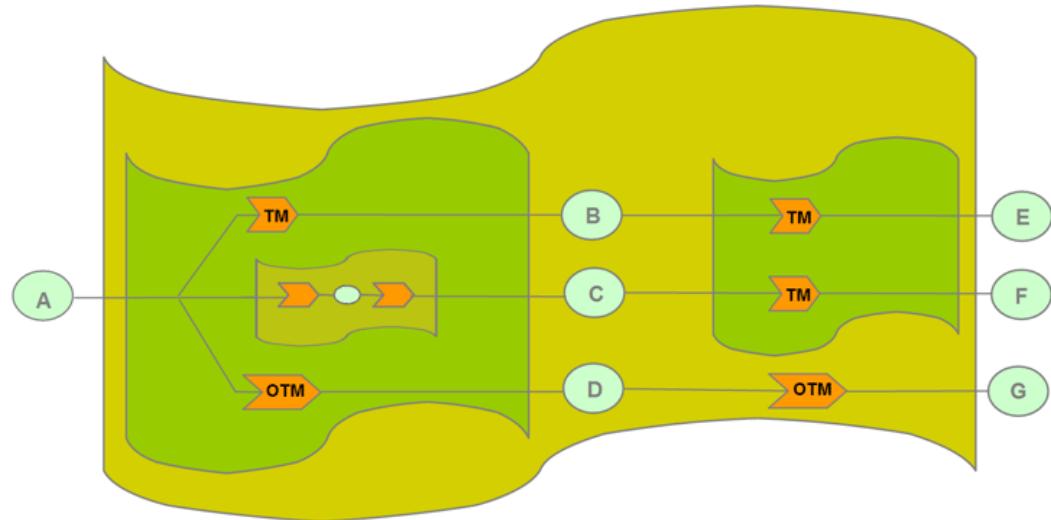


Figure 6-13: The output DOS of one parallel transformation chain must be the inputs for a succeeding parallel chain

Transformation chain M2D

An M2D conversion can be used to trigger a transformation chain execution. See “[M2D conversion](#)” on page 94. An M2D must be assigned to a transformation chain if either of the following conditions apply:

- The transformation chain is not used in a state machine.
If an M2D is assigned to a transformation chain used in a state machine, the M2D is not relevant, the transformation executes when the simple state executes.
- The transformation chain is not nested in another transformation chain.

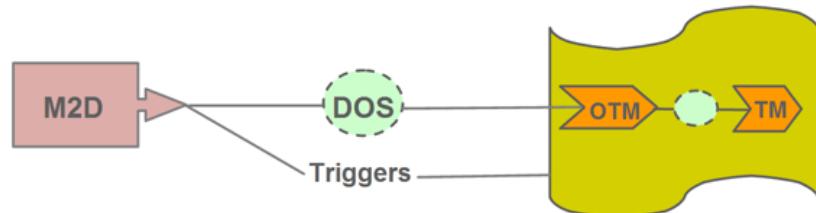


Figure 6-14: An M2D triggers a transformation chain not embedded in a state machine

To avoid processing conflicts, you should not assign an M2D to a nested transformation chain. The M2D should be assigned to the outermost transformation chain.

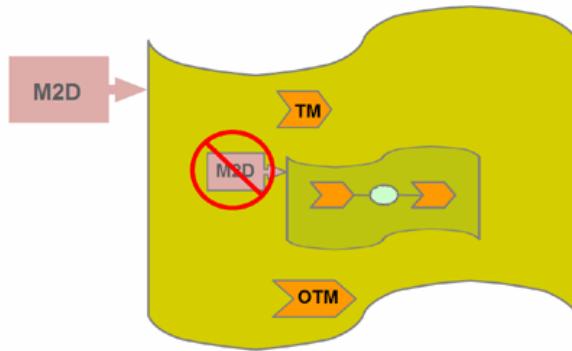


Figure 6-15: The M2D must be assigned to the outermost transformation chain

If your business model includes circumstances where different incoming messages could require the same transformation processing, you can assign multiple M2Ds to a single transformation chain.

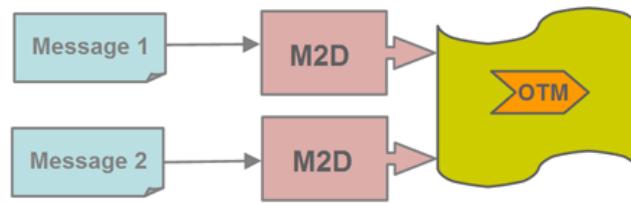


Figure 6-16: Multiple M2Ds may be assigned to a transformation chain

A single M2D can only trigger one transformation chain. If you need to trigger multiple transformation chains with the same M2D, you should nest the triggered transformation chains in a parallel transformation chain and assign the M2D to the outer parallel transformation chain.

Transformation chain D2M

The output from a transformation chain generally populates a DOS. To move the data from the DOS to the receiving application, you assign a D2M conversion to the transformation chain. For additional information and instructions on D2M conversions, see “[D2M conversion](#)” on page 96.

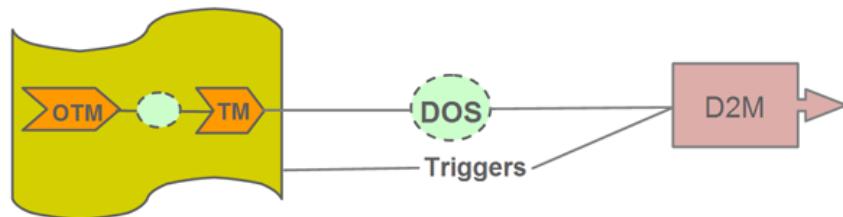


Figure 6-17: The transformation chain triggers a D2M conversion

A transformation chain can trigger multiple D2M conversions as long as each D2M uses the same DOS.

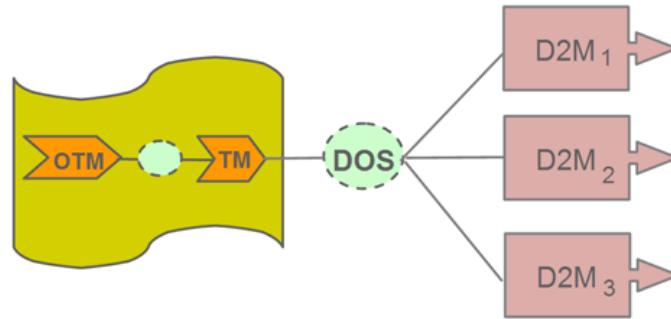


Figure 6-18: The transformation chain may trigger multiple D2Ms

In a parallel transformation chain, each transformation unit can trigger one or more D2M conversions. Each D2M must be specific to the output DOS of the transformation unit.

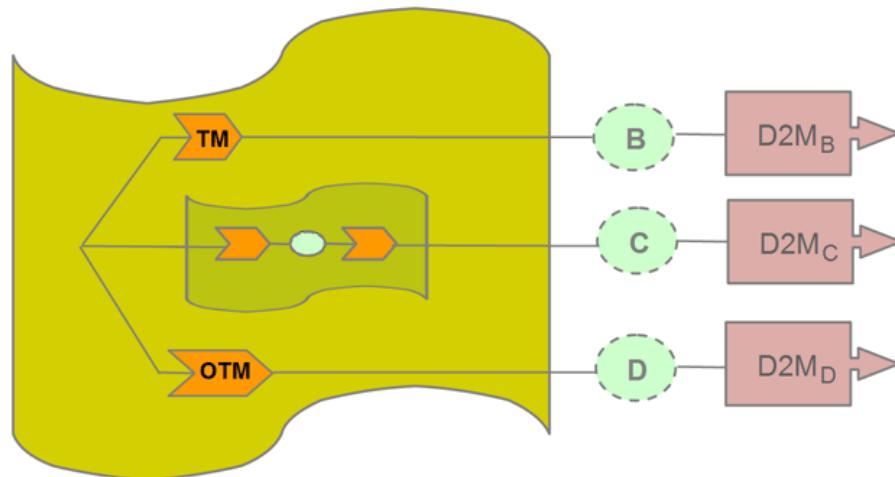


Figure 6-19: For multiple output DOS, each DOS must be assigned to a separate triggered D2M

You can assign a D2M on nested transformation chains. The D2M on a nested transformation chain does not execute until the entire transformation chain has successfully executed.

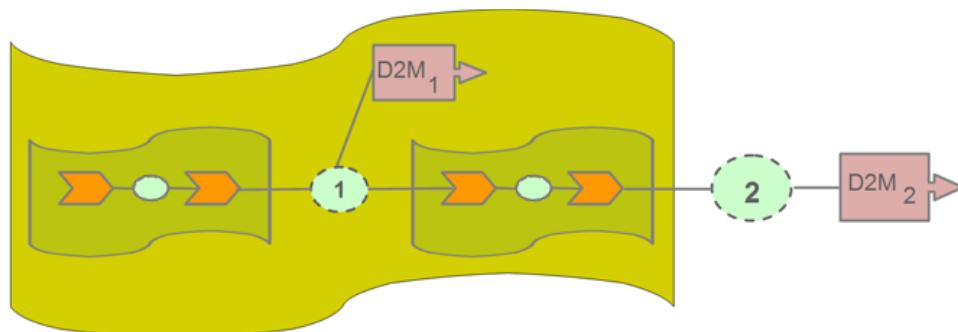


Figure 6-20: A D2M may also be triggered from within the transformation chain

Serial transformation chains

A serial transformation chain contains one or more transformation units that are processed in sequential order. Each transformation unit in the chain is dependant on the preceding transformation unit. The most basic serial transformation chain consists of one transformation unit. The figure below shows a basic serial transformation chain that includes one transformation map.

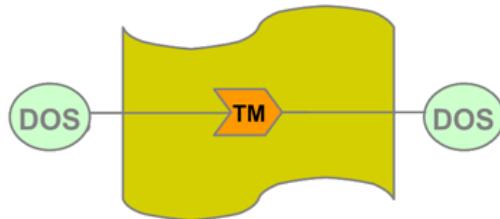


Figure 6-21: A simple serial transformation chain illustrated

In a serial transformation chain, the output DOS of one transformation unit is the input DOS for the next transformation unit as shown below. In this example, the chain consists of an opaque transformation map and a transformation map. The intermediary DOS (shown with the broken lines) is not specified in the transformation chain definition. It is implicit in the transformation unit definitions.

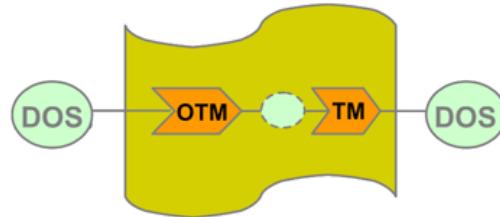


Figure 6-22: A serial transformation chain with an intermediary DOS

A serial transformation chain can also include other serial or parallel transformation chains as shown below.

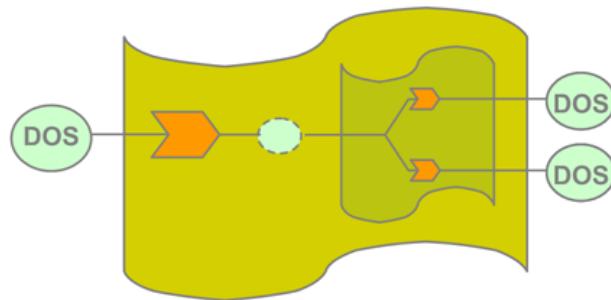


Figure 6-23: Transformation chains may be nested in a serial transformation chain

Creating a serial transformation chain

To create a serial transformation chain:

1. In Cogility Modeler's tree view, select the model container or package, click the **add** button  and select **Add Serial Transformation Chain**.

An untitled serial transformation chain displays in the tree view, and the serial transformation chain editor displays in the content view.

2. Define the serial transformation chain.

See “[Defining a serial transformation chain](#)” on page 169.

Defining a serial transformation chain

To define a serial transformation chain:

1. In the serial transformation chain editor, in the **Serial Transformation Chain** tab, in the **Name** field, enter a name.

The name you enter must be unique within the model container.

2. Create or add the transformation units.

See “[Creating transformation units in a transformation chain](#)” on page 171 and “[Adding transformation units to a transformation chain](#)” on page 172

3. Set the order of the transformation units.

See “[Setting the order of transformation units](#)” on page 172.

4. Define any pre- or post-transformation actions.

See the guide, *Using Actions in Cogility Studio* for help with actions.

5. In the **Triggers** tab, above the **Triggering M2D** field, create or add a triggering M2D conversion.

See “[M2D conversion](#)” on page 94, “[Creating an M2D conversion in a transformation chain](#)” on page 95 and “[Adding an M2D conversion to a transformation chain](#)” on page 95.

6. In the **Triggers** tab, above the **Triggered D2M** field, create or add a triggered D2M conversion.

See “[D2M conversion](#)” on page 96, “[Creating a D2M conversion in a transformation chain](#)” on page 96 and “[Adding a D2M conversion in a transformation chain](#)” on page 97.

Parallel transformation chains

The most basic parallel transformation chain consists of two transformation units that are processed concurrently.

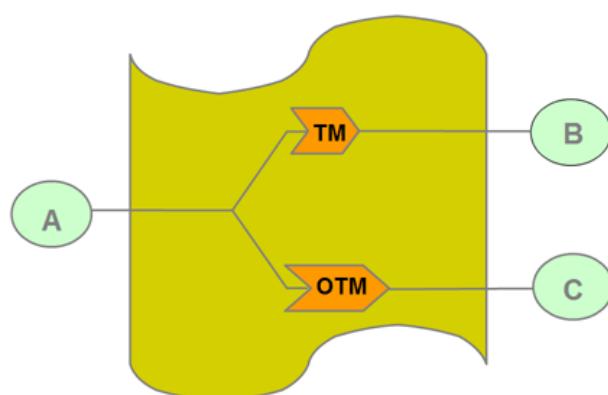


Figure 6-24: A simple parallel transformation chain illustrated

Unlike a serial transformation chain, a parallel transformation chain has no specified or implied sequence. Each transformation unit is processed concurrently. You can, however, include a serial transformation chain in a parallel transformation chain. Any intermediary DOS included within the

serial transformation chain cannot be a DOS used as input or output to the parallel transformation chain.

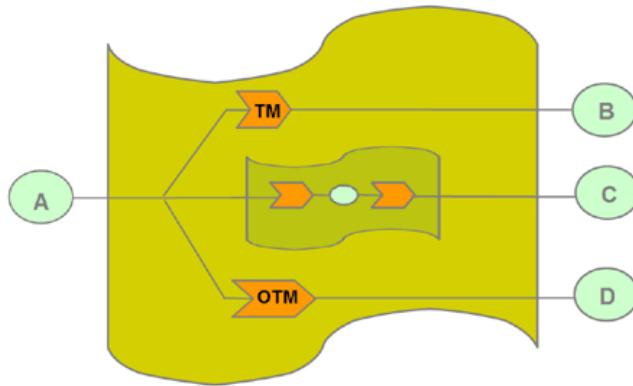


Figure 6-25: Transformation chains may be nested within a parallel transformation chain

Creating a parallel transformation chain

To create a parallel transformation chain:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **add** button  and select **Add a Parallel Transformation Chain**.
An untitled serial transformation chain displays in the tree view, and the serial transformation chain editor displays in the content view.
3. Define the parallel transformation chain.
See “Defining a parallel transformation chain” on page 170.

Defining a parallel transformation chain

To define a parallel transformation chain:

1. In the parallel transformation chain editor, in the **Parallel Transformation Chain** tab, in the **Name** field, enter a name.
The name you enter must be unique within the model container.
2. Create or add the transformation units.
See “Creating transformation units in a transformation chain” on page 171 and “Adding transformation units to a transformation chain” on page 172
3. In the **Triggers** tab, above the **Triggering M2D** field, create or add a triggering M2D conversion.
See “M2D conversion” on page 94.
4. In the **Triggers** tab, above the **Triggered D2M** field, create or add a triggered D2M conversion.
See “D2M conversion” on page 96.

Creating a transformation chain in an M2D conversion

To create a transformation chain in an M2D conversion:

1. In the M2D conversion editor, in the **M2D Triggered Chain** tab, click **New**.
2. In the selection dialog, select either a serial or parallel transformation chain and click **OK**.

3. In the dialog, enter a name for the transformation chain, and click **OK**.
4. To define the transformation chain, click **Edit**.
See “Defining a serial transformation chain” on page 169 and “Defining a parallel transformation chain” on page 170.

Creating a transformation chain in a D2M conversion

To create a transformation chain in a D2M conversion:

1. In the D2M conversion editor, in the **D2M Triggered By** tab, above the **Transformation Chain** field, click **New**.
2. In the selection dialog, select either a serial or parallel transformation chain and click **OK**.
3. In the dialog, enter a name for the transformation chain, and click **OK**.
4. To define the transformation chain, click **Edit**.
See “Defining a serial transformation chain” on page 169 and “Defining a parallel transformation chain” on page 170.

Setting a transformation chain in an M2D conversion

To set a transformation chain in an M2D conversion:

1. In the M2D conversion editor, in the **M2D Triggered Chain** tab, click **Set**.
2. In the selection dialog, select the transformation chain and click **OK**.
3. To define the transformation chain, click **Edit**.
See “Defining a serial transformation chain” on page 169 and “Defining a parallel transformation chain” on page 170.

Adding a transformation chain to a D2M conversion

To add a transformation chain to a D2M conversion:

1. In the D2M conversion editor, in the **D2M Triggered By** tab, above the **Transformation Chain** field, click **Add**.
2. In the selection dialog, select the transformation chain and click **OK**.
3. To define the transformation chain, click **Edit**.
See “Defining a serial transformation chain” on page 169 and “Defining a parallel transformation chain” on page 170.

Creating transformation units in a transformation chain

To create a transformation unit in a transformation chain:

1. In the transformation chain editor, in the **Serial or Parallel Transformation Chain** tab, above the **Transformation Units** field, click **New**.
2. In the selection dialog, select the transformation unit type and click **OK**.
3. In the dialog, enter a name for the transformation unit and click **OK**.
4. To define the transformation unit, click **Edit**.
See “Defining an opaque transformation map” on page 148, “Defining a transformation map” on page 153, “Defining a serial transformation chain” on page 169 and “Defining a parallel transformation chain” on page 170.

5. If you are adding units to a serial transformation chain, set the order of execution.
See “[Setting the order of transformation units](#)” on page 172.

Adding transformation units to a transformation chain

To add a transformation unit to a transformation chain:

1. In the transformation chain editor, in the **Serial or Parallel Transformation Chain** tab, above the **Transformation Units** field, click **Add**.
2. In the selection dialog, select the transformation units and click **OK**.
To select multiple units, hold down the Ctrl key while selecting.
3. If you are adding units to a serial transformation chain, set the order of execution.
See “[Setting the order of transformation units](#)” on page 172.
4. To define the transformation unit, click **Edit**.
See “[Defining an opaque transformation map](#)” on page 148, “[Defining a transformation map](#)” on page 153, “[Defining a serial transformation chain](#)” on page 169 and “[Defining a parallel transformation chain](#)” on page 170.

Setting the order of transformation units

To set the order of the transformation units for a serial transformation chain:

1. Select a transformation unit.
2. Use the **Move Up** and **Move Down** buttons to place the transformation unit in the correct order.
3. Repeat these steps until all the transformation units are in the correct order.

Data Transformation Editor

The Data Transformation Editor provides a visual approach to defining a transformation and allows you to see all the transformation artifacts in a structured view. The data transformation editor interface consists of a window with five panels as shown below.

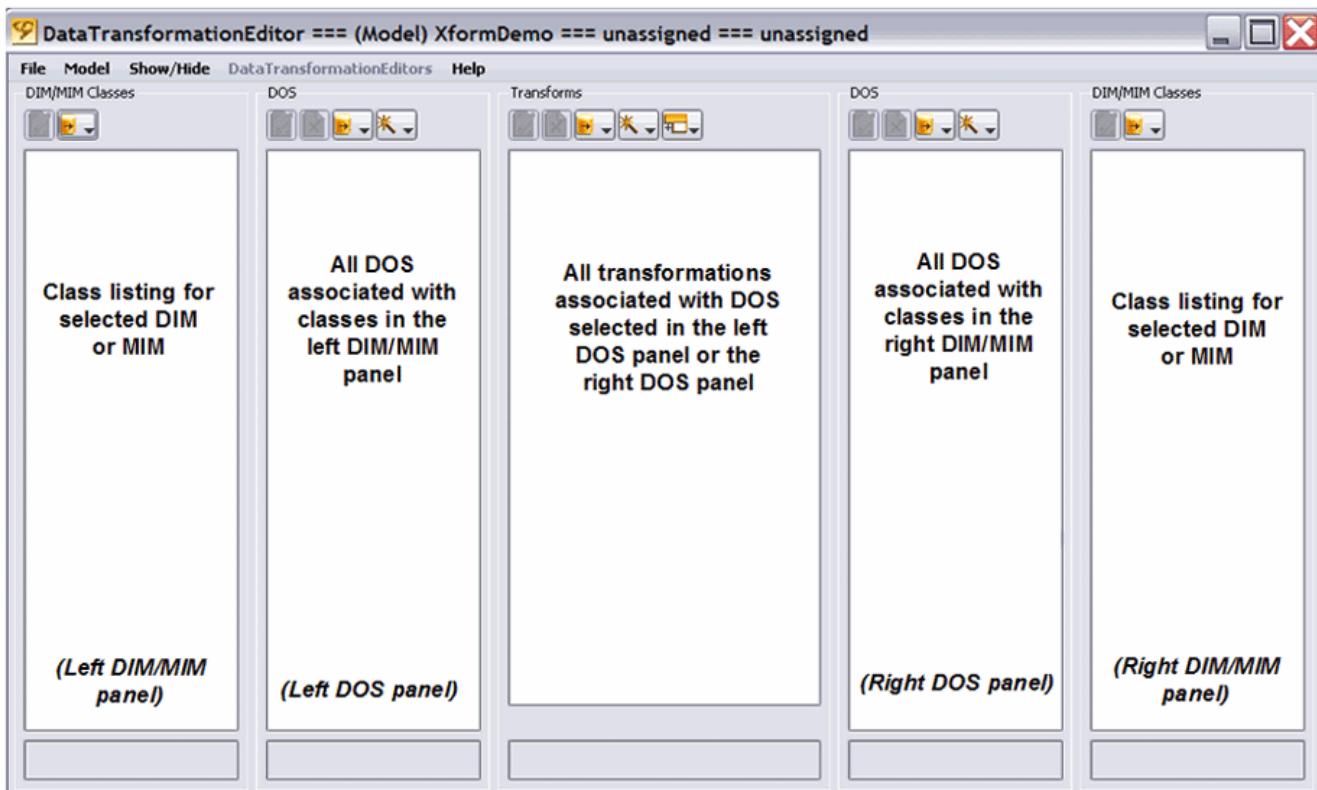


Figure 6-26: Data Transformation Editor

The leftmost and rightmost panels are the DIM/MIM Classes panels. These two panels list the classes for a specified DIM or the MIM. You select which DIM is used, or if the MIM is used, for each of the panels. A specific DIM can only be displayed in one panel at a time. The MIM can only be displayed in one panel at a time. The classes displayed in these panels are available for your transformation. There is no inherent “direction,” left to right or right to left for input and output in the editor display. The direction of the transformation (input to output) is specified during the transformation definition.

The next panels, moving toward the center from the left and the right, are the DOS panels. When you select a DIM or MIM in the DIM/MIM Classes panel, the associated DOS for that DIM or MIM are shown in the DOS panel.

The center panel is the Transforms panel. This panel shows existing transformations. If you select a DOS, the panel will display the transformations associated with that DOS. You also have the option of loading an existing transformation. If you load an existing transformation, the DOS and DIM/MIM Classes panels are updated to reflect the items associated with the loaded transformation. If the associated DOS have additional transformations, those are also displayed.

At the bottom of each panel is a consistency message field. Any consistency errors or problems related to a selected item in that panel are shown in this field.

You can open multiple instances of the Data Transformation Editor to work on multiple transformations.

In the example below, the MIMCustomer2OE_TM transformation map is loaded. The OECustomerDOS is highlighted in the left DOS panel as is the MIMCustomerDOS in the left DOS panel. The arrow just below the Transforms list shows the direction of the transformation, in this case right to left. The MIMCustomerDOS is the input DOS and the OECustomerDOS is the output DOS.

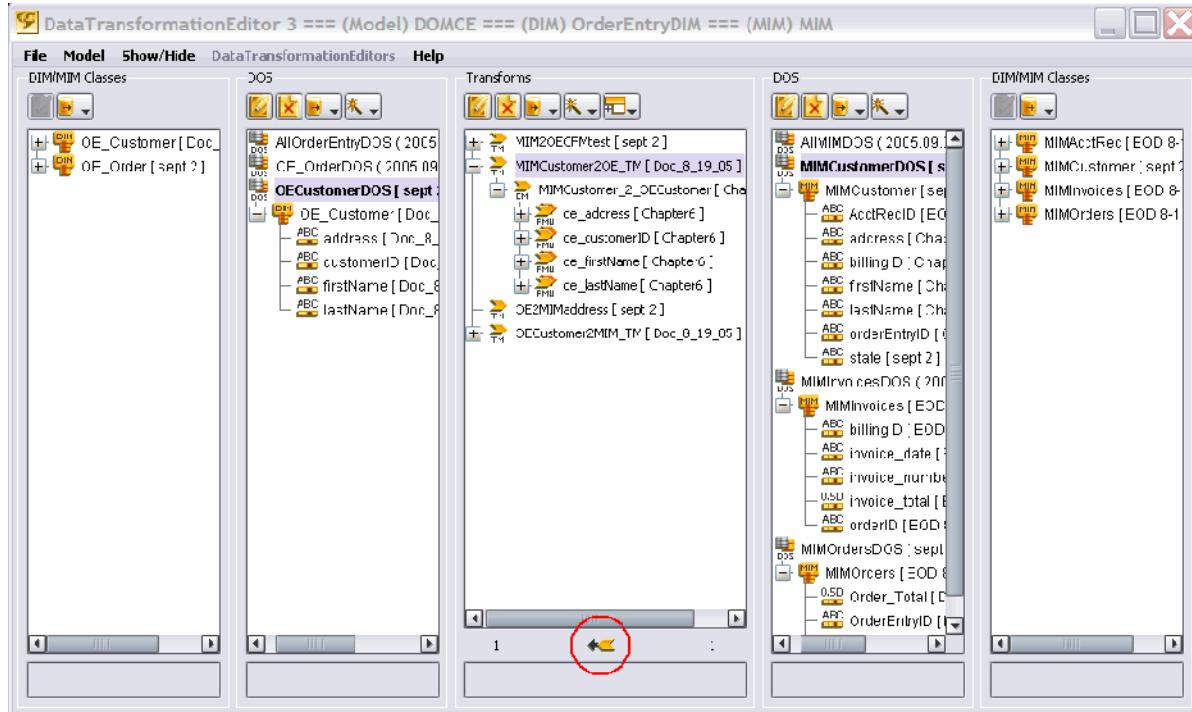


Figure 6-27: The direction of the transformation is indicated by the arrow (circled)

Clicking the classifier map for the transformation map allows you to see the classes used in the transformation. Continuing with the same example above, clicking the MIMCustomer_2_OECustomer classifier map shows that the MIMCustomer class is the input class.

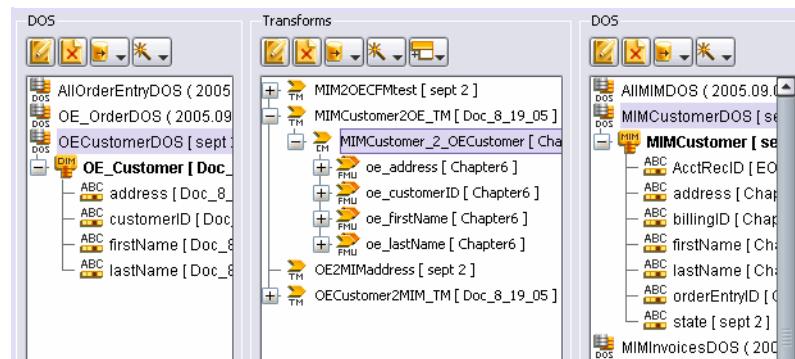


Figure 6-28: Transformation components are described in expandable trees

Listed below the classifier map are the feature map usages. If you click on a feature map usage, you can see which features are being transformed. In addition, if you open the feature map usage branch, you can see the feature map that is being used for the feature. In this case the oe_address feature map

usage shows that the StringCopy feature map is being used to copy the MIMCustomer address attribute to the OE_Customer address attribute.

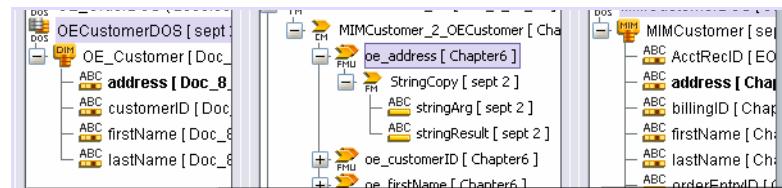


Figure 6-29: The transformation may be broken down to its lowest level containers

Panel function buttons

Each panel in the Data Transformation Editor interface provides a set of buttons to perform various functions as shown below:

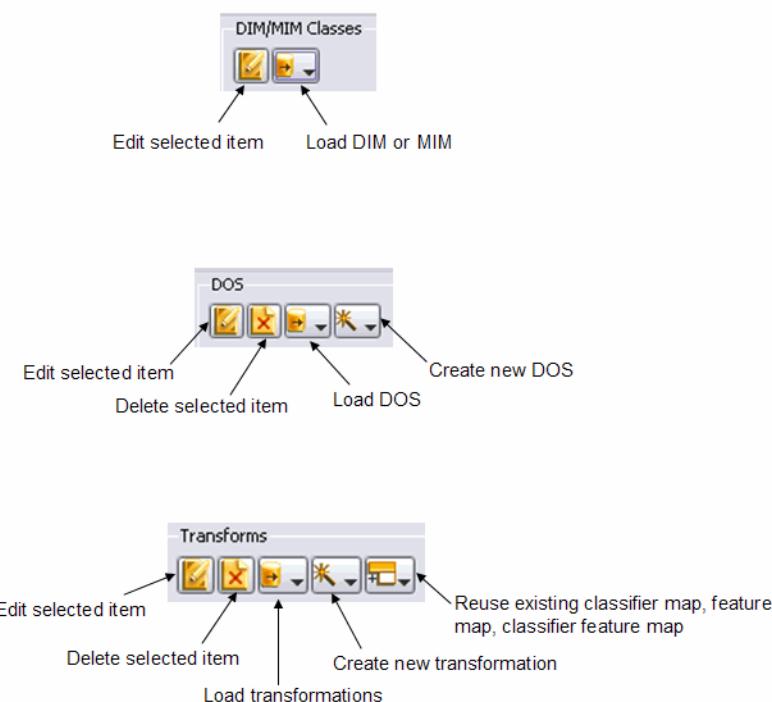


Figure 6-30: Data Transformation Editor buttons

The following sections describe the functionality of the panel buttons. Refer to the figure above for the button icons.

DIM/MIM Classes panel

Edit selected item - Opens the appropriate editor to edit a DIM or MIM class selected in the list.

Load DIM or MIM- Allows you to select from one of the following:

Load tree with existing DIM - Allows you to select the DIM to load.

Load tree with the only MIM:<MIM name>- Loads the MIM.

DOS panel

Edit selected item - Opens the appropriate editor to edit a DOS selected in the list.

Delete selected item - Allows you to delete a DOS selected in the list. A confirmation dialog allows you to verify the delete action before the item is deleted.

Load DOS - Allows you to select the following:

Load tree with existing DOS - Allows you to load a different DOS. If the DOS you select is not associated with the DIM or MIM already appearing the DIM/MIM panel, a dialog appears stating that the DIM or MIM classes panel needs to be updated to reflect your new choice and asks if you wish to continue. All DOS for the associated DIM or MIM are then displayed in the DOS panel, not just the DOS you selected.

Create new DOS - Opens the appropriate editor to create a new DOS. The available classes for the DOS are based on the DIM or MIM selected for the DIM/MIM Classes panel.

Transforms panel

Edit selected item - Opens the appropriate editor to edit a transformation selected in the list.

Delete selected item - Allows you to delete a transformation selected in the list. A confirmation dialog allows you to verify the delete action before the item is deleted.

Load transformations - Allows you to chose from one of the following:

Load tree with existing OpaqueTransformationMap - allows you to load an existing opaque transformation map.

Load tree with existing TransformationMap - allows you to load an existing transformation map.

If the transformation you select is not associated with a DOS already displayed in the DOS panel, a dialog appears stating that the DIM or MIM classes panel needs to be updated to reflect your new choice and asks if you wish to continue (this also updates the DOS panel). All transformations (both opaque transformation maps and transformation maps) for the associated DIM or MIM are then displayed in the Transformation panel, not just the transformation you selected.

Create a new transformation - Allows you to select one of the following:

Create a new OpaqueTransformationMap - enabled when you select a DOS from the right DOS panel and the left DOS panel. A dialog opens asking you to select the direction of the transformation (left to right or right to left). The Opaque Transformation Map editor opens with the input DOS and output DOS set based on your selections.

Create a new TransformationMap - enabled when you select a DOS from the right DOS panel and the left DOS panel. A dialog opens asking you to select the direction of the transformation (left to right or right to left). The Transformation Map editor opens with the input DOS and output DOS set based on your selections.

Create a new ClassifierMap - enabled when you select a transformation and one or more classes from the right and left DOS panels (the classes must be members of the DOS that is input to or output from the transformation map). The Classifier Map editor opens with the input classes and output classes set based on your selections.

Create a new FeatureMap - enabled when you select a classifier map and one or more attributes from the right and left DOS panels (the attributes must be members of the classes that are input to or output from the classifier map). The Feature Map editor opens with the input feature and output feature references set based on your selections.

Create a new ClassifierFeatureMap - enabled when you select a classifier map and one or more classes are selected in one DOS panel and one or more attributes are selected in the



other DOS panel (the classes and attributes must be members of the classes selected for the classifier map). The Classifier Feature Map editor opens with the classifiers and feature references set based on your selections.

Panel Menus

In addition to the menu buttons, each panel has a set of functions available through the right mouse button menu:

DIM/MIM Classes panel

Edit Selection - Opens the appropriate editor to edit a DIM or MIM class selected in the list.

Add Class(es) to DOS - Adds the selected class(es) to the DOS selected in the DOS panel.

Add Class(es) to new DOS - Adds the selected class(es) to a new DOS and opens the DOS editor.

Mark as ‘Not to Be Transformed’ - Allows you to specify that the selected class will not be used in a transformation. Removes the Not in Transformation inconsistency for the class.

Unmark as ‘Not to Be Transformed’ - Removes the Not to be Transformed specification from the selected class.

Load tree - Provides the same functionality as the load tree button. See “[Panel function buttons](#)” on page 175.

Show/Hide - Allows you to show or hide certain elements:

Show Version IDs - Select this option to show the version names in the object label.

Hide Version IDs - Select this option to hide the version names in the object label.

Show Consistency Pane - Select this option to show the consistency panel (and messages) below the panel.

Hide Consistency Pane - Select this option to hide the consistency panel.

Show Connection Info - Select this option to show the connection information. Indicates whether a selected class has a connection (through a transformation) to the DIM/MIM in the other DIM/MIM panel. If there is no connection, a purple flag  appears in the tree, and a message appears in the consistency panel.

Hide Connection Info - Select this option to hide the connection information. When you hide the connection information, you also cannot see the connection messages in the consistency panel.

DOS panel

Edit Selection - Opens the appropriate editor to edit the DOS or the class selected in the list.

Delete Selection - Deletes the DOS selected in the list. You can select delete only if you have selected a DOS in the list. The action is not enabled for the classes or attributes.

Rename Selection - Allows you to rename the DOS selected in the list. You can select rename only if you have selected a DOS in the list. The action is not enabled for the classes or attributes.

Create a new DOS - Opens the appropriate editor to create a new DOS. The available classes for the DOS are based on the DIM or MIM selected for the DIM/MIM Classes panel.

Remove Class(es) from DOS - Removes the selected class from the DOS.

Load tree with existing DOS - Provides the same functionality as the load tree button. See “[Panel function buttons](#)” on page 175.

Show/Hide - Provides the same show/hide options for version ID and the consistency panel as on the DIM/MIM Classes panel. See “[DIM/MIM Classes panel](#)” on page 177.

Transforms panel

Edit Selection - Opens the appropriate editor to edit the transformation map, classifier map, feature map, or classifier feature map selected in the list. It is not enabled for feature map usages or classifier feature map usages.

Delete Selection - Deletes the transformaton selected in the list.

Rename Selection - Allows you to rename the transformation selected in the list. You can select rename only if you have selected a DOS in the list. The action is not enabled for the classes or attributes.

Create - Provides the same functionality as the create a transformation button. See “[Panel function buttons](#)” on page 175.

Reuse - Provides the same functionality as the reuse a transformation button. See “[Panel function buttons](#)” on page 175.

Load tree - Provides the same functionality as the reuse a transformation button. See “[Panel function buttons](#)” on page 175.

Show/Hide - Provides the same show/hide options for version ID and the consistency panel as on the DIM/MIM Classes panel. See “[DIM/MIM Classes panel](#)” on page 177.

Running the Data Transformation Editor

To run the Data Transformation Editor:

1. In Cogility Modeler’s tree view, select the model container or package.
2. From the **Selection** menu, select **Open Data TransformationEditor**.

A blank Data Transformation Editor appears. See “[Viewing data transformations with the Data Transformation Editor](#)” on page 178 or “[Creating a transformation with the Data Transformation Editor](#)” on page 179 for instructions on using the Data Transformation Editor.

Viewing data transformations with the Data Transformation Editor

To view an existing data transformation:

1. Run the Data Transformation Editor.
See “[Running the Data Transformation Editor](#)” on page 178.
2. In the Data Transformation Editor, in the **Transforms** panel, click the **Load transformation** button .
3. From the drop-down list, select **Load tree with existing Transformation Map** or **Load tree with existing Opaque Transformation Map**.
4. In the dialog, select the transformation you wish to view and click **OK**.

The transformation you requested as well as all other transformations that use the same right and left DOS are shown in the Transforms panel. The DOS panels and the DIM/MIM panels are updated with the associated artifacts. All DIM/MIM classes and all DOS are

shown for the DIM/MIMs associated with the transformation. The DOS participating in the selected transformation are highlighted.

5. To edit a transformation unit, select it and click the **edit** button  above the transformation unit's column.

Creating a transformation with the Data Transformation Editor

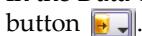
To create a new transformation:

1. Run the Data Transformation Editor.

See “[Running the Data Transformation Editor](#)” on page 178.

2. Select the participating DIM/MIM.

a. In the Data Transformation Editor, in the left **DIM/MIM Classes** panel, click the **load**



b. From the drop-down list, select **Load tree with existing DIM** or **Load tree with the only MIM**.

c. If you selected to load a DIM, in the dialog, select the DIM and click **OK**.

If you selected to load the MIM, it loads automatically.

You can click the plus sign next to the DIM or MIM to expand the tree and view the classes contained therein.

Note: When you select the MIM or a DIM, its associated DOS are displayed in the adjacent DOS pane. Click on the plus signs in the panel to expand the DOS tree.

d. Repeat these steps to make a selection in the right DIM/MIM panel.

3. Select the participating DOSs.

a. Select an input DOS from the right panel and an output DOS from the left panel. You can only select one DOS from each panel (a transformation can have only one input DOS and only one output DOS). If there is no available DOS that contains the classes you need to access for your transformation, you can also create a new DOS.

- In the **DOS** panel where you need to add a DOS, click the **create new DOS** button .
- Click the edit icon  to define the DOS. See “[Data object set \(DOS\)](#)” on page 46.

4. Create the transformation map or opaque transformation map.

See “[Creating an opaque transformation map in the Data Transformation Editor](#)” on page 148 and “[Creating a transformation map in the Data Transformation Editor](#)” on page 152.



Custom Queries

Overview

Within your model you may use actions to retrieve information from Cogility Studio's run-time repository. For simple queries involving one class and a few attributes, you can use direct SQL statements with the `locate` and `locateAll orderedBy` functions. This is described in ["Direct SQL" on page 99](#) of the guide, *Using Actions in Cogility Studio*.

For more complex queries involving several classes, several attributes and multiple associations you can use custom queries. You develop these with model artifacts described in this section and specify the SQL as described in ["Custom queries" on page 101](#) of the guide, *Using Actions in Cogility Studio*.

Custom queries are one of several artifact types that may be pushed into execution separately from the model. See ["Single artifact push" on page 30](#) of the guide, *Model Deployment & Execution in Cogility Studio*.

Custom query attributes

A query's attributes describe the information it returns. For custom queries, the SELECT and FROM statements of SQL are represented by the query's attribute expression. A custom query attribute specifies the target of the statement SELECT column X FROM table Y.

For the example, the query shown in the following figures returns the number of orders for the state in which the orders originate. The query attribute expression is composed of ordinary SQL and some shorthand for describing the column names. In the following example, the #C is shorthand for class, meaning the class table, and the #A is shorthand for attribute, meaning the attribute of the table. Each

shorthand notation is followed by the name of the table or attribute in brackets []. In the Expression field below, the expression identifies the lastName attribute of the Employee class.

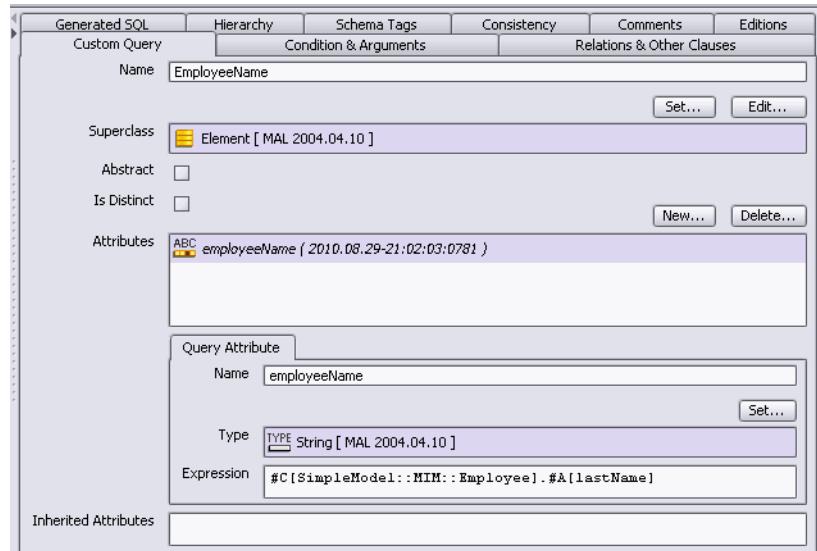


Figure 7-1: Custom query attribute definition

You can use model assist in the Expression field to help you find the class, association class, role, and so forth. See “[Model assist](#)” on page 300.

The expression may also represent a computed value. The expression below describes the sum of the product orders based on the quantity attribute of the A_product_order association class (#AC).

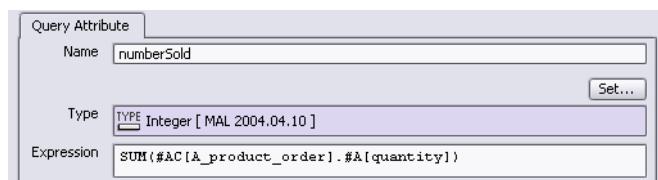


Figure 7-2: Custom query attribute definition

For a complete description of the Cogility custom query shorthand, see the guide, *Using Actions in Cogility Studio*.

Custom query conditions

A query must have a condition in order to be consistent with the rules of modeling in Cogility Studio. If you do not specify a condition, the query displays an inconsistency warning. The condition is described with the WHERE clause in SQL.

If you do not want to specify a condition, and you leave the Condition field blank, the query object will show an inconsistency. To avoid the inconsistency, enter an expression in the field that evaluates to true, such as `1=1`. Note that not specifying a condition and entering an expression that evaluates to true causes the query to retrieve the entire database at run time.

You can specify SQL conditions using operators such as greater than (>), less than (<), IN and so forth. The arguments specify the values for the SELECT statement of the query, and are represented by the argument model artifact.

In the following example, the condition specifies that the query return Orders with an orderDate that is greater than the dateSince submitted by the caller.

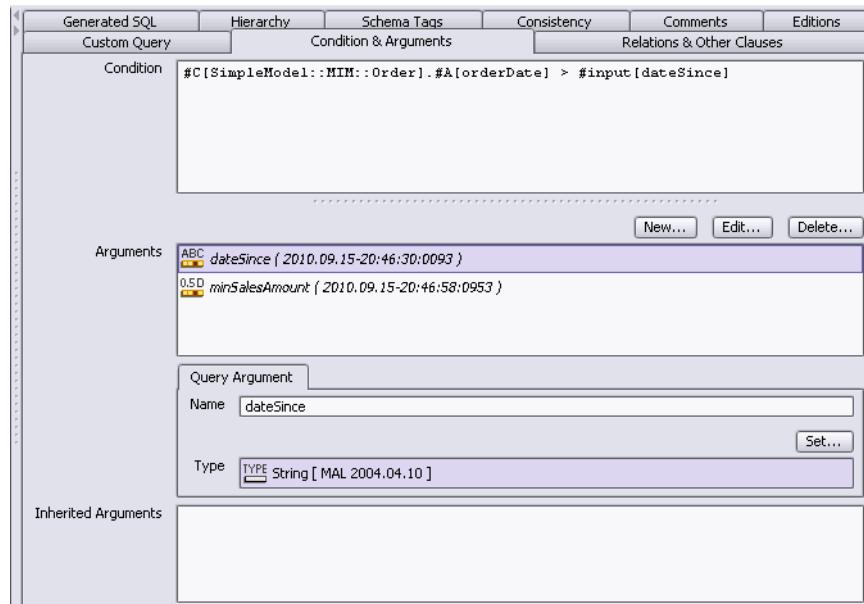


Figure 7-3: Custom query condition definition

Custom query arguments

When you call the query in actions, you pass arguments that specify the bounds of the query condition.

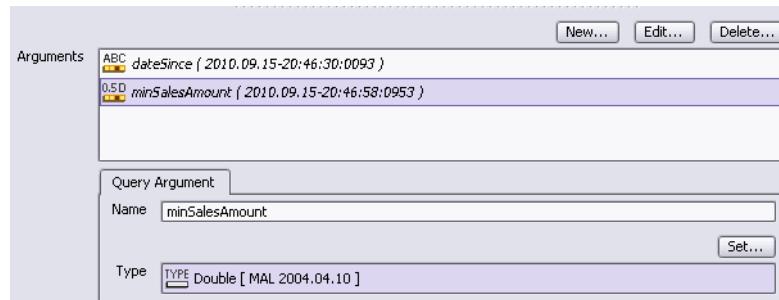


Figure 7-4: Custom query argument definition

Custom query relations

The associations between your model's classes are referenced as relations in the SQL join statement. You represent these associations with relation expressions.

For example, consider the associations between the classes as described in the diagram below.

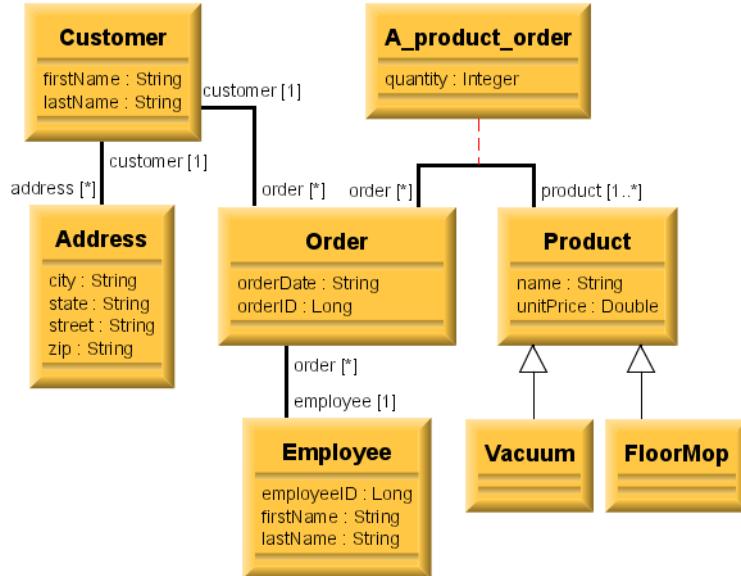


Figure 7-5: Associations illustrated for a custom query

These associations are expressed as relations with the following expressions:

	Generated SQL	Hierarchy	Schema Tags	Consistency	Comments	Editions
Custom Query			Condition & Arguments			Relations & Other Clauses
Relations	<pre>#C[SimpleModel::MIM::Address].#R[customer] #C[SimpleModel::MIM::Customer].#R[order] #C[SimpleModel::MIM::Order].#R[employee] #C[SimpleModel::MIM::Order].#R[product@SimpleModel::MIM::Vacuum]</pre>					
Other Clauses						<pre>GROUP BY #C[SimpleModel::MIM::Employee].#A[lastName], #C[SimpleModel::MIM::Address].#A[state], #C[SimpleModel::MIM::Vacuum].#A[name], #C[SimpleModel::MIM::Vacuum].#A[unitPrice] HAVING max(#C[SimpleModel::MIM::Vacuum].#A[unitPrice]) * SUM(#AC[SimpleModel::MIM::A_product_order].#A[quantity]) > #input[minSalesAmount] ORDER BY #C[SimpleModel::MIM::Employee].#A[lastName]</pre>

Figure 7-6: Relations expressions in a custom query

The default join action for a relationship declaration is an inner join. You can also explicitly specify an outer join. To specify an outer join, you use the `*` shorthand symbol. For the syntax to specify an outer join, see the guide, *Using Actions in Cogility Studio*.

Other clauses in custom queries

The SQL join clause includes the GROUP BY, HAVING and ORDER BY statements. These further qualify the information returned in the query.

For example, the following expressions group the query results by Employee lastName, Customer Address State, Product Name, and Product unitPrice for those results having a minimum sales amount and order the results by Employee lastName.

```

Other Clauses
GROUP BY #C[SimpleModel::MIM::Employee].#A[lastName],
#C[SimpleModel::MIM::Address].#A[state],
#C[SimpleModel::MIM::Vacuum].#A[name],
#C[SimpleModel::MIM::Vacuum].#A[unitPrice]

HAVING max(#C[SimpleModel::MIM::Vacuum].#A[unitPrice]) *
SUM(#AC[SimpleModel::MIM::A_product_order].#A[quantity]) > #input[minSalesAmount]

ORDER BY #C[SimpleModel::MIM::Employee].#A[lastName]

```

Figure 7-7: Other clauses in a custom query

Creating a custom query

To create a custom query:

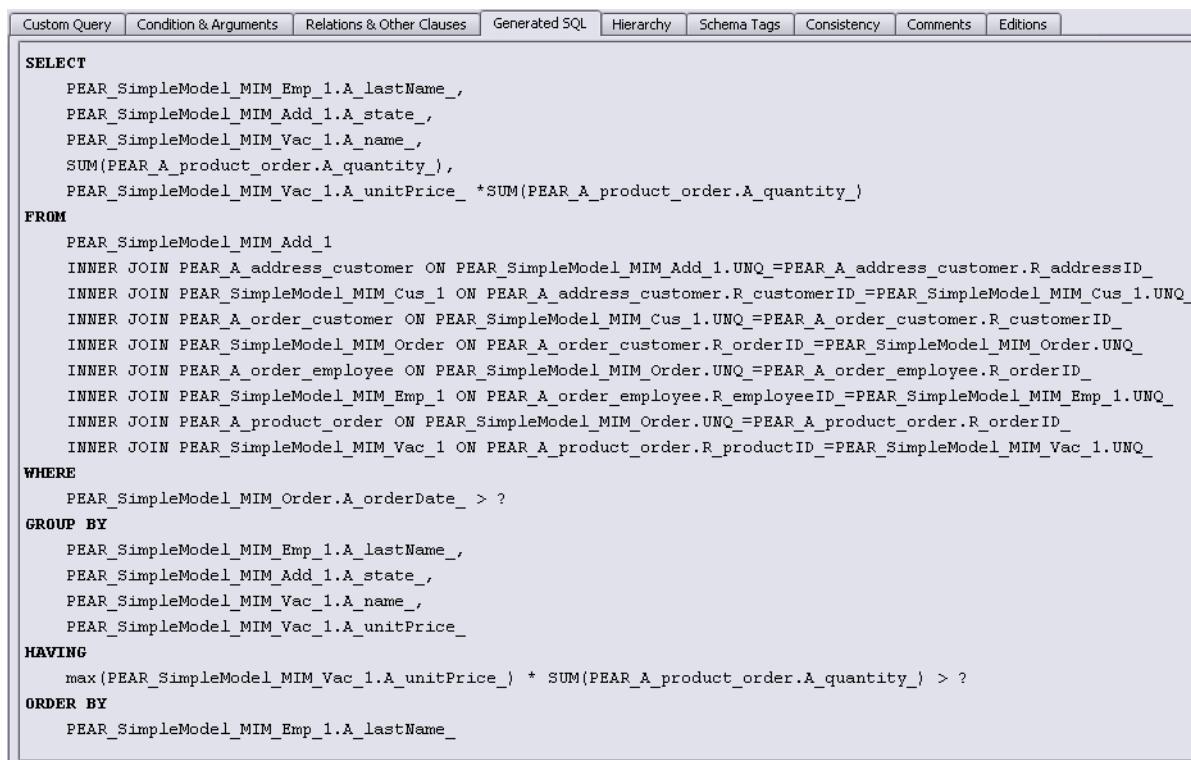
1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **Plus** button and select **Add Custom Query**.
3. In the custom query editor, in the **Custom Query** tab, for **Name**, enter a name.
4. In the Superclass field, set the superclass. See “[Superclass](#)” on page 33.
5. In the Attributes tab, create the attributes.
 - a. In the custom query editor, in the **Custom Query** tab, for **Attributes**, click **New**.
 - b. In the dialog, in the **Name** field enter a name.
 - c. For **Type**, select a type from the drop down list and click **OK**.
 - d. In the **Custom Query Attribute** tab, for **Expression**, enter the expression that describes the query attribute.
 Specify the SQL as described in “[Custom queries](#)” on page 101 of the guide, *Using Actions in Cogility Studio*.
6. Create the custom query condition and argument:
 - a. Click the **Conditions and Arguments** tab.
 - b. In the **Conditions** field, enter a condition.
 - c. Above the **Arguments** field, click **New**.
 - d. In the dialog, in the **Name** field enter a name.
 - e. For **Type**, from the drop down list select a type and click **OK**.
 Specify the SQL as described in “[Custom queries](#)” on page 101 of the guide, *Using Actions in Cogility Studio*.
7. Define the relations and other clauses used in the custom query:
 - a. Click the **Relations & Other Clauses** tab.
 - b. In the **Relations** field, enter the relation expressions.
 Specify the SQL as described in “[Custom queries](#)” on page 101 of the guide, *Using Actions in Cogility Studio*.
- c. In the **Other Clauses** field, enter the query expressions that specify the clauses.

Specify the SQL as described in “[Custom queries](#)” on page 101 of the guide, *Using Actions in Cogility Studio*.

Verify the generated SQL

As you write your query, Cogility Modeler attempts to verify the query statements against the tables in the run-time repository. If you created new classes or associations in the current context but have not pushed them onto the run-time repository, a query that uses these unpushed model objects cannot be verified. Once you complete the query and push the finished model, the query verifies and Cogility Modeler can display the generated SQL.

Cogility Modeler generates a final SQL query from custom query objects you created.



The screenshot shows the Cogility Modeler interface with the 'Generated SQL' tab selected. The window title is 'Generated SQL'. Below the title bar are tabs for 'Custom Query', 'Condition & Arguments', 'Relations & Other Clauses', 'Generated SQL' (which is highlighted), 'Hierarchy', 'Schema Tags', 'Consistency', 'Comments', and 'Editions'. The main area contains the generated SQL code:

```
SELECT
    PEAR_SimpleModel_MIM_Emp_1.A_lastName_,
    PEAR_SimpleModel_MIM_Add_1.A_state_,
    PEAR_SimpleModel_MIM_Vac_1.A_name_,
    SUM(PEAR_A_product_order.A_quantity_),
    PEAR_SimpleModel_MIM_Vac_1.A_unitPrice_ * SUM(PEAR_A_product_order.A_quantity_)

FROM
    PEAR_SimpleModel_MIM_Add_1
    INNER JOIN PEAR_A_address_customer ON PEAR_SimpleModel_MIM_Add_1.UNQ_=PEAR_A_address_customer.R_addressID_
    INNER JOIN PEAR_SimpleModel_MIM_Cus_1 ON PEAR_A_address_customer.R_customerID_=PEAR_SimpleModel_MIM_Cus_1.UNQ_
    INNER JOIN PEAR_A_order_customer ON PEAR_SimpleModel_MIM_Cus_1.UNQ_=PEAR_A_order_customer.R_customerID_
    INNER JOIN PEAR_SimpleModel_MIM_Order ON PEAR_A_order_customer.R_orderID_=PEAR_SimpleModel_MIM_Order.UNQ_
    INNER JOIN PEAR_A_order_employee ON PEAR_SimpleModel_MIM_Order.UNQ_=PEAR_A_order_employee.R_orderID_
    INNER JOIN PEAR_SimpleModel_MIM_Emp_1 ON PEAR_A_order_employee.R_employeeID_=PEAR_SimpleModel_MIM_Emp_1.UNQ_
    INNER JOIN PEAR_A_product_order ON PEAR_SimpleModel_MIM_Order.UNQ_=PEAR_A_product_order.R_orderID_
    INNER JOIN PEAR_SimpleModel_MIM_Vac_1 ON PEAR_A_product_order.R_productID_=PEAR_SimpleModel_MIM_Vac_1.UNQ_

WHERE
    PEAR_SimpleModel_MIM_Order.A_orderDate_ > ?

GROUP BY
    PEAR_SimpleModel_MIM_Emp_1.A_lastName_,
    PEAR_SimpleModel_MIM_Add_1.A_state_,
    PEAR_SimpleModel_MIM_Vac_1.A_name_,
    PEAR_SimpleModel_MIM_Vac_1.A_unitPrice_

HAVING
    max(PEAR_SimpleModel_MIM_Vac_1.A_unitPrice_) * SUM(PEAR_A_product_order.A_quantity_) > ?

ORDER BY
    PEAR_SimpleModel_MIM_Emp_1.A_lastName_
```

Figure 7-8: Generated SQL of a custom query

To verify the generated SQL:

1. In the tree view, select the custom query.
2. In the custom query editor, click the **Generated SQL** tab.

Custom query example

An example model that illustrates a custom query is available for you to load and run with the accompanying action scripts. Those scripts are described in “[Example](#)” on page 104 of the guide, *Using Actions in Cogility Studio*. This section describes the model and the class diagram that describes the relationships between classes.

The example model is located as %DCHOME%\Examples\Queries\QuerySample.cog.



Model objects

The custom query is included in the SimpleModel sample. The query reports employee sales data of products per state.

The sample model consists of several classes associated with several relationships that the action scripts traverse in executing their queries. For more information about associations and association classes, see “[Associations](#)” on page 51.

See the sections under “[Overview](#)” on page 181 for a complete description of this custom query.

Running the sample model

In this section, you push the model and use Cogility Action Pad to run the custom queries.

To run the example model:

1. Load the example model.
2. Turn over and push the model.

See “[Pushing the model into execution](#)” on page 273.

3. Run Cogility Action Pad.

See “[Running Cogility Action Pad](#)” on page 80 of the guide, *Model Deployment & Execution in Cogility Studio*.

4. Open and run the file %DCHOME%\Examples\Queries\DataSetUp.ocl.

See “[Opening FilePads](#)” on page 82 and “[Running action semantics in an action pad](#)” on page 90 of the guide, *Model Deployment & Execution in Cogility Studio*.

5. Open and run the file, %DCHOME%\Examples\Queries\RunSalesQuery.ocl.

This script performs a query on the run-time data. To review the script see “[Example](#)” on page 104 of the guide, *Using Actions in Cogility Studio*.

The output generated is shown below.

```
Console

>>> [Verify ]: Action's syntax verified in 511 ms.
>>> [Serialize]: Serialization/de-serialization succeeded in 20 ms. Size=13394 bytes.

>>> [Message]: Factory lookup returned a(n): com.ibm.ws.sib.api.jms.impl.JmsTopicConnectionFactoryImpl
>>> [Println]: Size of Coll = 13
>>> [Println]: Employee      State       ProductName     ItemsSold     SaleAmount
>>> [Println]: ======  ======  ======  ======  ======
>>> [Println]: Black        CA          Hoover         5            7995.0
>>> [Println]: Black        CA          Oreck          2            2598.0
>>> [Println]: Black        NY          DustDevil     2            318.0
>>> [Println]: Black        NY          Hoover        12           19188.0
>>> [Println]: Brown       CA          Hoover        1             1599.0
>>> [Println]: Brown       CA          Kirby          2             499.98
>>> [Println]: Brown       NY          DustDevil     8             1272.0
>>> [Println]: Brown       NY          Kirby          20            4999.8
>>> [Println]: Green       CA          DustDevil     5             795.0
>>> [Println]: Green       CA          Kirby          10            2499.9
>>> [Println]: Green       CA          Oreck          4             5196.0
>>> [Println]: Green       NY          Kirby          12            2999.88
>>> [Println]: Green       NY          Oreck          3             3897.0

>>> [Running]: Running of Actions succeeded in 561 ms.
>>> [Commit ]: Action's results committed.
```

Figure 7-9: Output from a custom query

XSD artifacts

XML Schema Definition (XSD) artifacts model complex types (those composed of subtypes) defined in XML. Simple types such as strings, integers, longs, and so forth, readily map to the argument and result attributes of web services or the arguments and returns of operations. If the web service or operation receives or returns a complex type, it may be defined with an XSD artifact. For example, if an external web service expects the complex type as an argument, your outbound web service artifact must describe that complex type. If the external web service returns a complex type, the type is described in an XML string that must then be parsed and worked with in the model. Likewise, if you have an inbound web service that describes a complex type in its WSDL, the inbound web service artifact must be able to work with that complex type programmatically.

If you have an inbound web service that specifies a complex type for the argument or result, you can create an XSD artifact for that type and use it in your web service's action semantics. You can create the XSD artifact by importing the XSD from a URL or writing your own XSD. Once you have defined an XSD artifact, you have access to it through your model's action semantics, and you can use model assist to retrieve the type, its attributes and associations. See “[Inbound web services using SOAP over HTTP](#)” on page 228. Likewise, you can define operations that take an XSD type as an argument. See “[Operations](#)” on page 41.

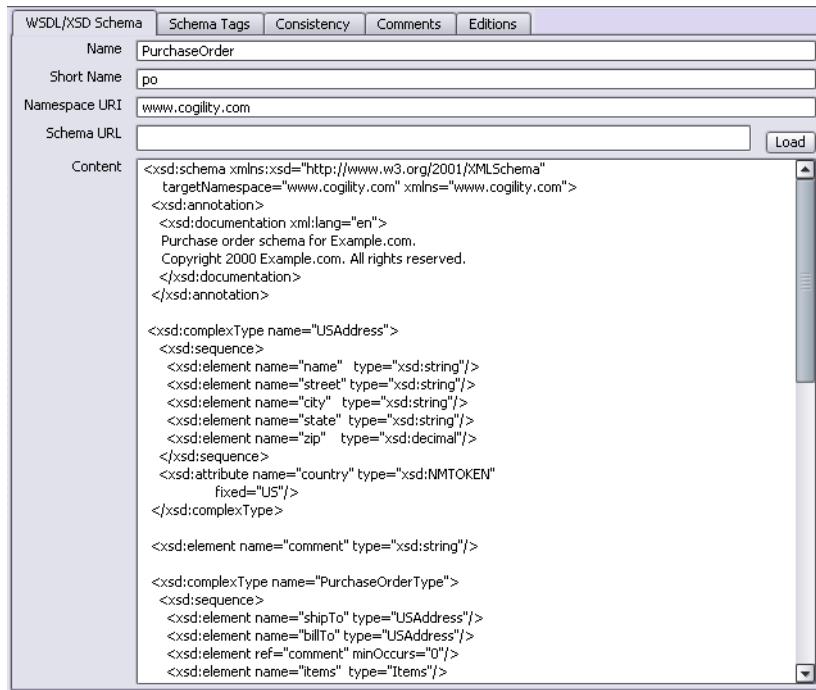


Figure 7-10: An XSD artifact

The XSD artifact has the following fields:

- **Name** - the name of the artifact.
- **Short Name** - the namespace identifier for the schema. When you refer to the objects in the schema in your action semantics, you identify them by the name in this field. For example, the PurchaseOrderType defined in the above schema is accessed in action semantics as follows. Its short name, `po` identifies the namespace for the schema.

```
pot : po::PurchaseOrderType;
```

The statement above defines a variable (`pot`) for the PurchaseOrderType. See “[XSD artifacts in action semantics](#)” on page 190.

- **Namespace URI** - the universal resource identifier that describes the namespace associated with the schema.
- **Schema URL** - the location from where the XSD or its WSDL are imported.
- **Content** - the XSD or WSDL definitions contained in this model object. If the XSD is imported from a WSDL, the entire WSDL will appear in the field. You can also define your own XSD by typing it here.

Creating an XSD artifact

To create an XSD artifact:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the plus button  and select **Add WSDL/XSD Schema**.
3. In the **Name** field, enter a name for the artifact.
4. If you are loading an XSD from a URL, see "[Imported XSD](#)" on page 189.
5. In the **Short Name** field, enter a short identifier unique within the model container.
This is only needed if the content is not imported. The import process attempts to create short name mappings for each namespace encountered in the imported content.
6. If you are creating the XSD from scratch, enter the definition in the **Content** field.

Imported XSD

You can import an XSD from a URL or import a WSDL containing an XSD from a URL. If you import the WSDL, the XSD contained therein will be parsed out and made available programmatically through the XSD artifact while the rest of the WSDL will remain for reference.

To use the XSD artifacts in your action semantics, you must refer to the fully qualified XSD types. (See "[XSD artifacts in action semantics](#)" on page 190.) The qualifiers include the namespace that identifies the XSD type. For an imported XSD of one schema object with one associated namespace, the namespace identifier will be used to populate the Short Name field in the same XSD artifact. For imported WSDLs, since a given WSDL may have many XSD fragments and each of these may have a different namespace, each namespace will have its own empty XSD artifact and the namespace identifier will be used to populate the Short Name of that artifact. Internally, the XSD elements parsed from the WSDL will be associated with those namespace XSD artifacts. Also, when the XSD types in a WSDL refer to the default namespace, a separate XSD artifact will be created for that namespace.

For example, when the usaddressverification WSDL is imported from webservicex.net, a separate XSD artifact named "tns" also gets created. Both of the XSD types in the WSDL reside under this namespace, but because it is the default namespace, Cogility Modeler creates a separate artifact named for that namespace. You can see that both XSD types are referenced with respect to the

namespace, as in `tns::Address`. Cogility Modeler makes the association between the namespace and the XSD type explicit, as shown below when the element is accessed using model assist.

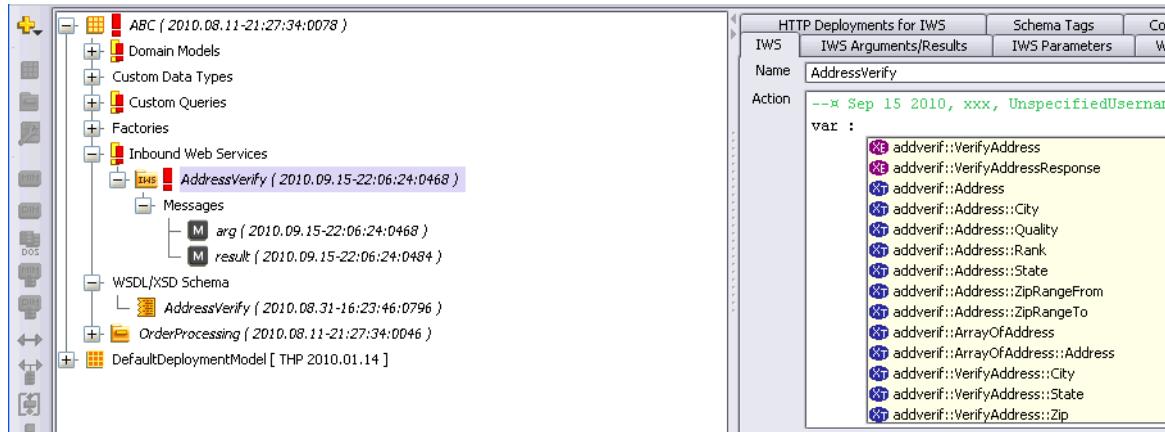


Figure 7-11: The USAddress and VerifyAddress XSD elements imported into Cogility Modeler

Loading an XSD

To load an XSD into the XSD artifact:

1. Create an XSD artifact, following steps 1 through 3 of “[Creating an XSD artifact](#)” on page 189.
 2. In the XSD artifact, in the **Schema URL** field, enter the URL for the XSD.
 3. Click the **Load** button.
If the XSD or WSDL containing it loads properly, a dialog indicating the content loaded successfully appears.
 4. In the dialog, click **OK**.
- The XSD or WSDL text populates the Content field. The Short Name and Namespace URI fields are populated, either on the same XSD artifact or within a separate XSD artifact, one for each namespace, in the tree view.

XSD artifacts in action semantics

Working with XSD types in your action semantics involves either reading the XML string returned from a web service and then creating from that string a run-time object of the type defined in the model by the XSD artifact, or writing an XML string for a model-defined XSD type with the class information and run-time data.

Read XML

The `CreateInstanceOfXSDType` Java action creates an instance of an XSD type definition given an XML string and a fully qualified XSD model artifact. It creates the instance of the specified XSD type from the XML string. This Java action is described in “[CreateInstanceOfXSDType](#)” on page 62 of the guide, *Using Actions in Cogility Studio*.

For example, an inbound web service that reads an XML string holding the definition of a object will expect that string as an argument. The web service’s action semantics can refer to that string from the `input` bound variable, create the object instance and assign it to a variable. First, the action semantics must declare the variable of the XSD type, fully qualified with the short name that defines its namespace. The following lines create an instance of the `PurchaseOrder` type described in Figure 7-10 on page 188.

```

purchOrd : po::PurchaseOrderType;
purchOrd := java
com::cogility::CreateInstanceOfXSDType(input.singleString,
'po::PurchaseOrderType');

```

From there, the elements that compose the complex type may be parsed out and assigned to model classes, as in the following.

```

billTo : po::USAAddress;
billTo := purchOrd.billTo;
billAddr.street := billTo.street;
billAddr.city := billTo.city;
billAddr.state := billTo.state;
billAddr.country := billTo.country;
billAddr.zip := billTo.zip;

```

In the above, the USAAddress is an element of the PurchaseOrderType. Notice that you can access the attributes very simply with dot notation: `purchOrd.billTo`. Likewise, the attributes of the USAAddress are accessed with dot notation: `billTo.street`. See also, “[Associations for sub elements](#)” on page 192.

Write XML

The Java method, `toXML()` composes an XML string from an instance of an XSD type. Likewise, the `toXMLFragment()` method composes the XML string without header information so that the string may be inserted in a larger block of XML. Both of these methods are available from model assist for any model-defined XSD type, as illustrated below.

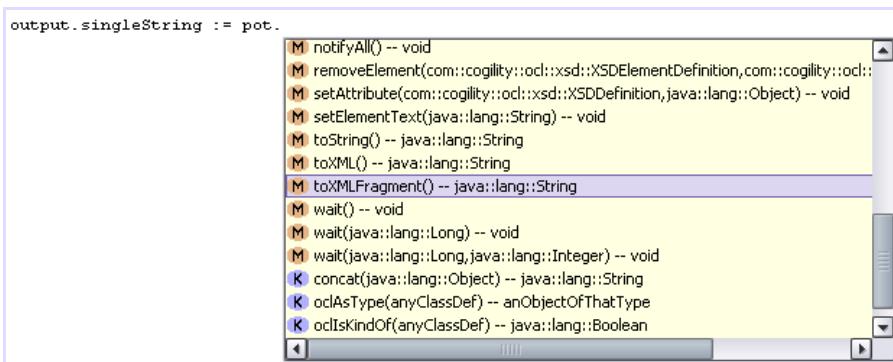


Figure 7-12: Java methods `toXML` and `toXMLFragment` in model assist

These methods take a fully-qualified XSD type as an argument and return a String. That String may be assigned to the result message of a web service, accessed by the `output` bound variable, as shown below. In this example, `singleString` is the attribute of the result message that holds the String; `pot` is the variable that holds the PurchaseOrderType XSD type defined in the model.

```
output.singleString := pot.toXML();
```

Before you produce the String, however, you have to compose the XSD type from model class instances. First, declare a variable of the XSD type, as shown below.

```

pot : po::PurchaseOrderType;
pot := new po::PurchaseOrderType();

```

Then you compose the XSD type from the model objects, traversing the associations that define the entity. See [Associations for sub elements](#), next.

Associations for sub elements

Instead of using JAXB or XPath to assemble the XSD type from your model objects, you can use the standard action semantics facilities for traversing associations. Then you can assign the values from these associations to the elements in the XSD type. With the XSD type thus defined, you convert it to an XML string as discussed in “[Write XML](#)” on page 191.

In the following example, an inbound web service’s action semantics need to write an XML string for an XSD type that describes an Order and all of its associated objects such as Customer, Product, Address and so forth. The inbound web service argument takes a String for the `orderId` accessible from the `input` bound variable, and locates the object.

```
ord : Model::MIM::Order;  
ord := locate Model::MIM::Order(orderId: input.orderId);
```

As described in “[Write XML](#)” on page 191, you then declare a variable of the XSD type, as shown below.

```
pot : po::PurchaseOrderType;  
pot := new po::PurchaseOrderType();
```

Where the association has only one target, you can use `WalkAssoc` to get the association (see “[WalkAssoc](#)” on page 75 of the guide *Using Actions in Cogility Studio*) and assign the value of the associated object attribute using dot notation. The action semantics below get the Customer (via the `customer` role) of the Order and assign it to the `cust` variable all in one line. Then the `customerId` attribute value is assigned to the `custID` of the `PurchaseOrderType`.

```
cust : Model::MIM::Customer := java com::cogility::WalkAssoc(ord,  
'customer');  
pot.custID := cust.customerId;
```

For associations that may have multiple targets, the action semantics need to create the associations between business objects explicitly. Consider the association described in the following class diagram.

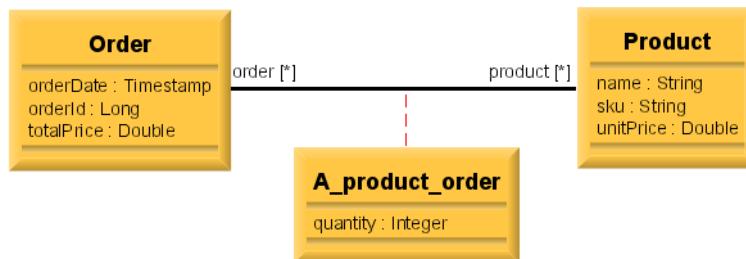


Figure 7-13: A multiple target association

For each of the Products associated with the Order, the action semantics must create an item (element) of the `Items` element and assign the product attribute values to the item attribute values. This is accomplished by iterating over the Collection of Products for the Order and creating the associations using dot notation.

```
ord.product -> iterate (prod : Model::MIM::Product |  
itm : po::Items::item;  
itm := new po::Items::item();  
itm.productName := prod.name;  
itm.USPrice := prod.unitPrice;  
itm.partNum := prod.sku;
```

Because the quantity of a particular Product is held in the association class attribute, the action semantics use the `oclLinkObject` keyword to retrieve that association class. See “`oclLinkObject to`” on page 47 of the guide, *Using Actions in Cogility Studio*.

```
objLink : Model::MIM::A_product_order;  
objLink := oclLinkObject ord to product(prod);  
itm.quantity := objLink.quantity;
```

The association between the item (element) and the Items element is created using the `new` and `to` keywords. See the guide, *Using Actions in Cogility Studio*. These action semantics are used to create the associations where the relationship between the source and target is multiple, as with many `items` to the `Item` collection.

```
new items to item(itm);  
);
```

Finally, the collection of items are assigned to the PurchaseOrderType.

```
pot.items := items;
```




Overview

XML Schema Definition (XSD) artifacts model complex types (those composed of subtypes) defined in XML. Simple types such as strings, integers, longs, and so forth, readily map to the argument and result attributes of web services or the arguments and returns of operations. If the web service or operation receives or returns a complex type, it may be defined with an XSD artifact. For example, if an external web service expects the complex type as an argument, your outbound web service artifact must describe that complex type. If the external web service returns a complex type, the type is described in an XML string that must then be parsed and worked with in the model. Likewise, if you have an inbound web service that describes a complex type in its WSDL, the inbound web service artifact must be able to work with that complex type programmatically.

If you have an inbound web service that specifies a complex type for the argument or result, you can create an XSD artifact for that type and use it in your web service's action semantics. You can create the XSD artifact by importing the XSD from a URL or writing your own XSD. Once you have defined an XSD artifact, you have access to it through your model's action semantics, and you can use model assist to retrieve the type, its attributes and associations. See ["Inbound web services using SOAP"](#)

over HTTP" on page 228. Likewise, you can define operations that take an XSD type as an argument. See "Operations" on page 41.

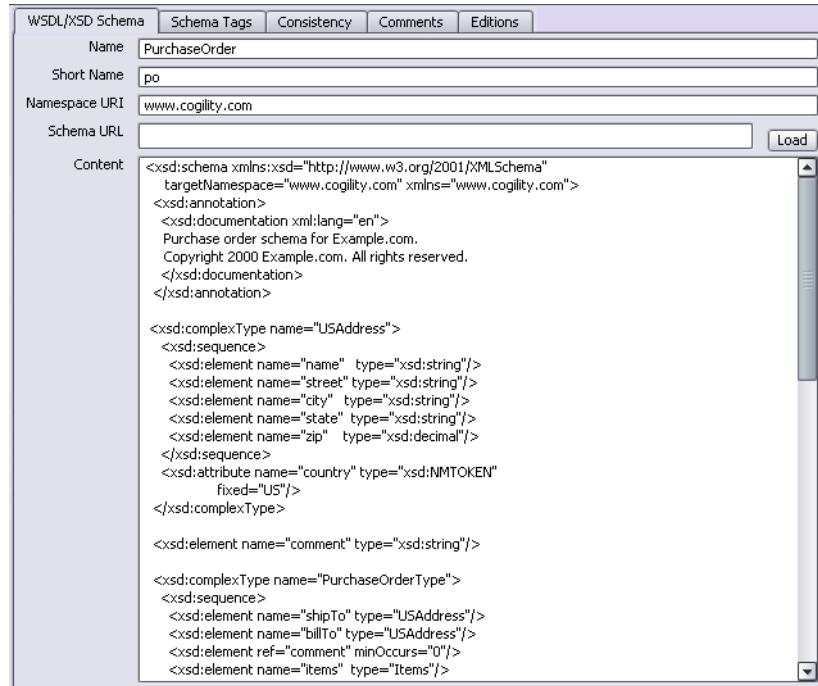


Figure 8-1: An XSD artifact

The XSD artifact has the following fields:

- **Name** - the name of the artifact.
- **Short Name** - the namespace identifier for the schema. When you refer to the objects in the schema in your action semantics, you identify them by the name in this field. For example, the PurchaseOrderType defined in the above schema is accessed in action semantics as follows. Its short name, **po** identifies the namespace for the schema.

`pot : po::PurchaseOrderType;`

The statement above defines a variable (**pot**) for the PurchaseOrderType. See "XSD artifacts in action semantics" on page 198.

- **Namespace URI** - the universal resource identifier that describes the namespace associated with the schema.
- **Schema URL** - the location from where the XSD or its WSDL are imported.
- **Content** - the XSD or WSDL definitions contained in this model object. If the XSD is imported from a WSDL, the entire WSDL will appear in the field. You can also define your own XSD by typing it here.

Creating an XSD artifact

To create an XSD artifact:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the plus button and select **Add WSDL/XSD Schema**.
3. In the **Name** field, enter a name for the artifact.

4. If you are loading an XSD from a URL, see “[Importing an XSD](#)” on page 197.
5. In the **Short Name** field, enter a short identifier unique within the model container. This is only needed if the content is not imported. The import process attempts to create short name mappings for each namespace encountered in the imported content.
6. If you are creating the XSD from scratch, enter the definition in the **Content** field.

Importing an XSD

You can import an XSD from a URL or import a WSDL containing an XSD from a URL. If you import the WSDL, the XSD contained therein will be parsed out and made available programmatically through the XSD artifact while the rest of the WSDL will remain for reference.

To use the XSD artifacts in your action semantics, you must refer to the fully qualified XSD types. (See “[XSD artifacts in action semantics](#)” on page 198.) The qualifiers include the namespace that identifies the XSD type. For an imported XSD of one schema object with one associated namespace, the namespace identifier will be used to populate the Short Name field in the same XSD artifact. For imported WSDLs, since a given WSDL may have many XSD fragments and each of these may have a different namespace, each namespace will have its own empty XSD artifact and the namespace identifier will be used to populate the Short Name of that artifact. Internally, the XSD elements parsed from the WSDL will be associated with those namespace XSD artifacts. Also, when the XSD types in a WSDL refer to the default namespace, a separate XSD artifact will be created for that namespace.

For example, when the usaddressverification WSDL is imported from webservicex.net, a separate XSD artifact named “tns” also gets created. Both of the XSD types in the WSDL reside under this namespace, but because it is the default namespace, Cogility Modeler creates a separate artifact named for that namespace. You can see that both XSD types are referenced with respect to the namespace, as in `tns::Address`. Cogility Modeler makes the association between the namespace and the XSD type explicit, as shown below when the element is accessed using model assist.

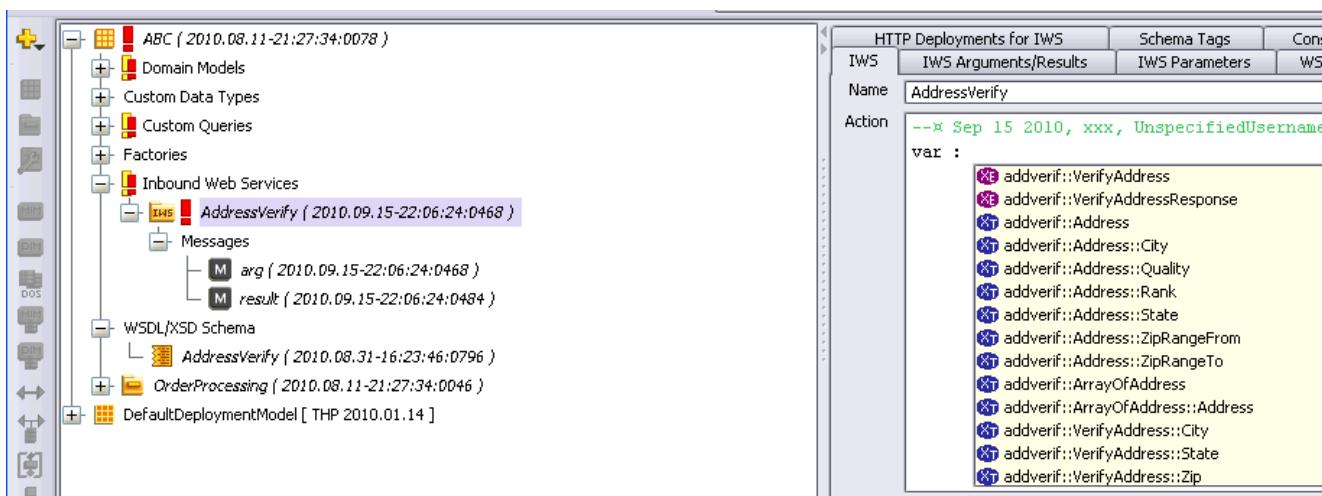


Figure 8-2: The USAddress and VerifyAddress XSD elements imported into Cogility Modeler

Loading an XSD

To load an XSD into the XSD artifact:

1. Create an XSD artifact, following steps 1 through 3 of “[Creating an XSD artifact](#)” on page 196.

2. In the XSD artifact, in the **Schema URL** field, enter the URL for the XSD.

3. Click the **Load** button.

If the XSD or WSDL containing it loads properly, a dialog indicating the content loaded successfully appears.

4. In the dialog, click **OK**.

The XSD or WSDL text populates the Content field. The Short Name and Namespace URI fields are populated, either on the same XSD artifact or within a separate XSD artifact, one for each namespace, in the tree view.

XSD artifacts in action semantics

Working with XSD types in your action semantics involves either reading the XML string returned from a web service and then creating from that string a run-time object of the type defined in the model by the XSD artifact, or writing an XML string for a model-defined XSD type with the class information and run-time data.

Read XML

The `CreateInstanceOfXSDType` Java action creates an instance of an XSD type definition given an XML string and a fully qualified XSD model artifact. It creates the instance of the specified XSD type from the XML string. This Java action is described in “[CreateInstanceOfXSDType](#)” on page 62 of the guide, *Using Action Semantics in Cogility Studio*.

For example, an inbound web service that reads an XML string holding the definition of a object will expect that string as an argument. The web service’s action semantics can refer to that string from the `input` bound variable, create the object instance and assign it to a variable. First, the action semantics must declare the variable of the XSD type, fully qualified with the short name that defines its namespace. The following lines create an instance of the `PurchaseOrder` type described in [Figure 8-1 on page 196](#).

```
purchOrd : po::PurchaseOrderType;  
  
purchOrd := java  
com::cogility::CreateInstanceOfXSDType(input.singleString,  
'po::PurchaseOrderType');
```

From there, the elements that compose the complex type may be parsed out and assigned to model classes, as in the following.

```
billTo : po::USAddress;  
billTo := purchOrd.billTo;  
billAddr.street := billTo.street;  
billAddr.city := billTo.city;  
billAddr.state := billTo.state;  
billAddr.country := billTo.country;  
billAddr.zip := billTo.zip;
```

In the above, the `USAddress` is an element of the `PurchaseOrderType`. Notice that you can access the attributes very simply with dot notation: `purchOrd.billTo`. Likewise, the attributes of the `USAddress` are accessed with dot notation: `billTo.street`. See also, “[Associations for sub elements](#)” on page 199.

Write XML

The Java method, `toXML()` composes an XML string from an instance of an XSD type. Likewise, the `toXMLFragment()` method composes the XML string without header information so that the string may be inserted in a larger block of XML. Both of these methods are available from model assist for any model-defined XSD type, as illustrated below.

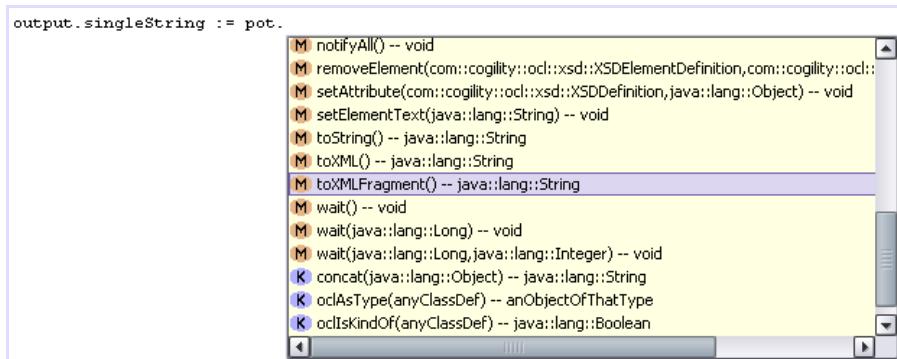


Figure 8-3: Java methods `toXML` and `toXMLFragment` in model assist

These methods take a fully-qualified XSD type as an argument and return a String. That String may be assigned to the result message of a web service, accessed by the `output` bound variable, as shown below. In this example, `singleString` is the attribute of the result message that holds the String; `pot` is the variable that holds the `PurchaseOrderType` XSD type defined in the model.

```
output.singleString := pot.toXML();
```

Before you produce the String, however, you have to compose the XSD type from model class instances. First, declare a variable of the XSD type, as shown below.

```
pot : po::PurchaseOrderType;
pot := new po::PurchaseOrderType();
```

Then you compose the XSD type from the model objects, traversing the associations that define the entity. See [Associations for sub elements](#), next.

Associations for sub elements

Instead of using JAXB or XPath to assemble the XSD type from your model objects, you can use the standard action semantics facilities for traversing associations. Then you can assign the values from these associations to the elements in the XSD type. With the XSD type thus defined, you convert it to an XML string as discussed in ["Write XML" on page 199](#).

In the following example, an inbound web service's action semantics need to write an XML string for an XSD type that describes an Order and all of its associated objects such as Customer, Product, Address and so forth. The inbound web service argument takes a String for the `orderId` accessible from the `input` bound variable, and locates the object.

```
ord : Model::MIM::Order;
ord := locate Model::MIM::Order(orderId: input.orderId);
```

As described in ["Write XML" on page 199](#), you then declare a variable of the XSD type, as shown below.

```
pot : po::PurchaseOrderType;
pot := new po::PurchaseOrderType();
```

Where the association has only one target, you can use WalkAssoc to get the association (see “WalkAssoc” on page 75 of the guide *Using Action Semantics in Cogility Studio*) and assign the value of the associated object attribute using dot notation. The action semantics below get the Customer (via the customer role) of the Order and assign it to the `cust` variable all in one line. Then the `customerId` attribute value is assigned to the `custID` of the PurchaseOrderType.

```
cust : Model::MIM::Customer := java com::cogility::WalkAssoc(ord,
'customer');
pot.custID := cust.customerId;
```

For associations that may have multiple targets, the action semantics need to create the associations between business objects explicitly. Consider the association described in the following class diagram.

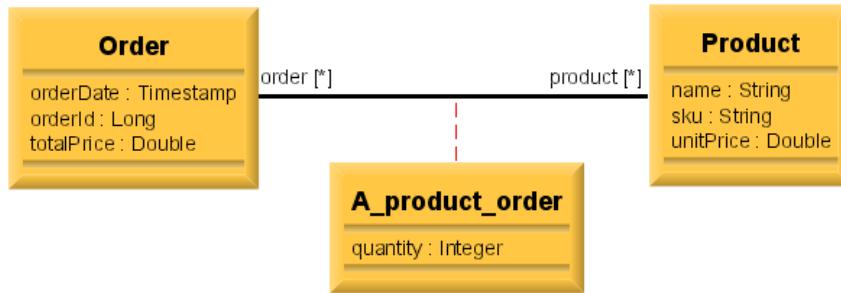


Figure 8-4: A multiple target association

For each of the Products associated with the Order, the action semantics must create an item (element) of the Items element and assign the product attribute values to the item attribute values. This is accomplished by iterating over the Collection of Products for the Order and creating the associations using dot notation.

```
ord.product -> iterate (prod : Model::MIM::Product |
itm : po::Items::item;
itm := new po::Items::item();
itm.productName := prod.name;
itm.USPrice := prod.unitPrice;
itm.partNum := prod.sku;
```

Because the quantity of a particular Product is held in the association class attribute, the action semantics use the `oclLinkObject` keyword to retrieve that association class. See “`oclLinkObject to`” on page 47 of the guide, *Using Action Semantics in Cogility Studio*.

```
objLink : Model::MIM::A_product_order;
objLink := oclLinkObject ord to product(prod);
itm.quantity := objLink.quantity;
```

The association between the item (element) and the Items element is created using the `new` and `to` keywords. See the guide, *Using Action Semantics in Cogility Studio*. These action semantics are used to create the associations where the relationship between the source and target is multiple, as with many `items` to the `Item` collection.

```
new items to item(itm);
);
```

Finally, the collection of items are assigned to the PurchaseOrderType.

```
pot.items := items;
```



Web Services Model

A web service is a programmable component that provides access to application functionality over the internet. With Cogility Studio you can model inbound web services to provide platform independent access to your composite application. There are two types of inbound web services you can define in Cogility Studio: a web service that uses the SOAP protocol or a web service that uses HTTP as the transport mechanism. Your composite application can also access external systems by modeling outbound web services.

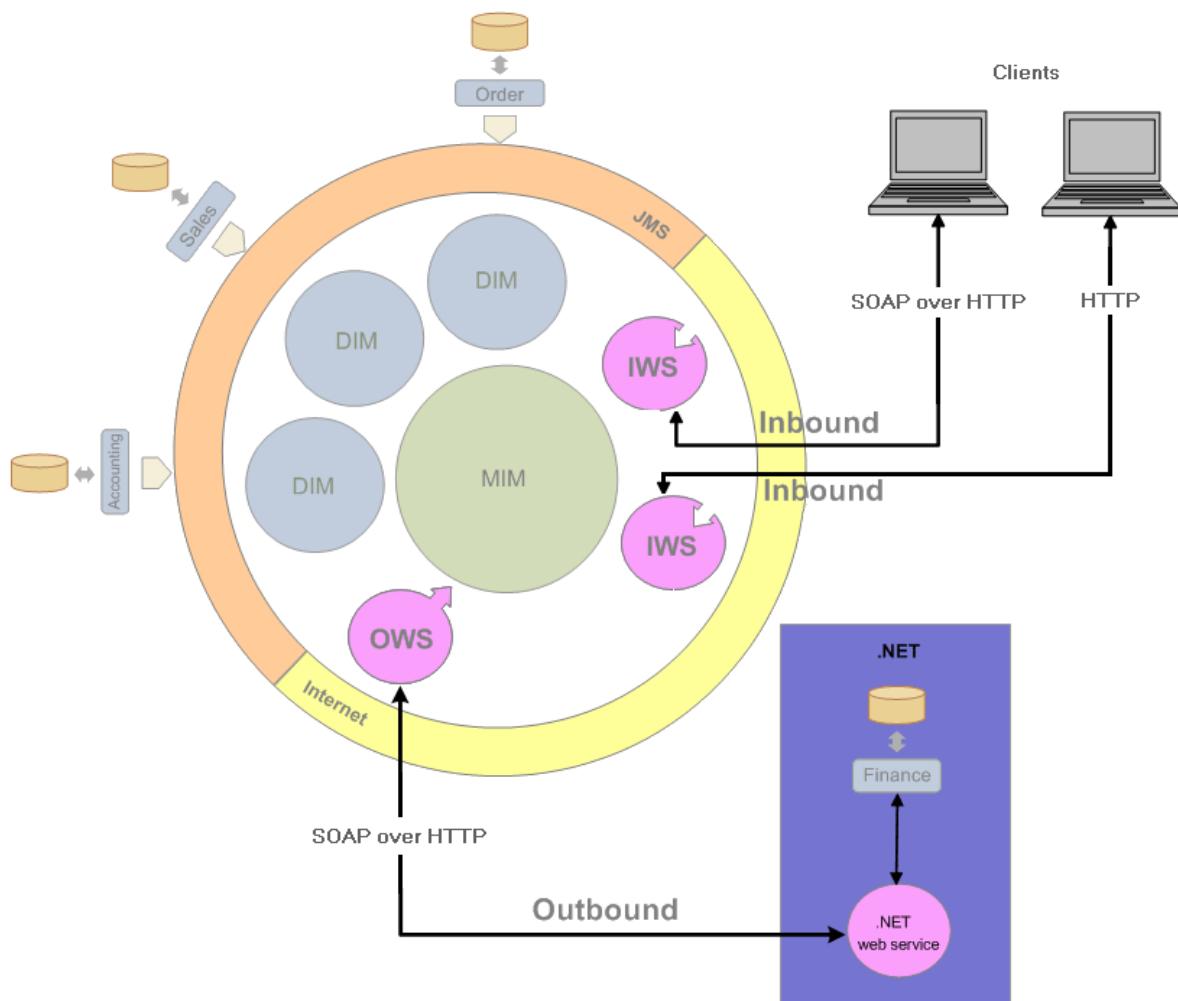


Figure 9-1: Inbound and outbound web services

Inbound and outbound basically refer to web services from the perspective of the executing composite application. Inbound web services are deployed by the model and hosted on the application server onto which the model is pushed. Outbound web services invoke external or third party web services.

Other applications call into the composite application to access all the inbound web services of a model. Inbound web services serve as input points for various applications or front ends to communicate with the model using a standard interface. For example, in the figure above, one client accesses an inbound web service using SOAP. Another client accesses an inbound web service using HTTP operations..

Outbound web services allow the model to access web services that are deployed by external applications. In the figure above, an outbound web service invokes a web service for a Finance application deployed on a .NET platform.

Modeling SOAP Inbound Web Services

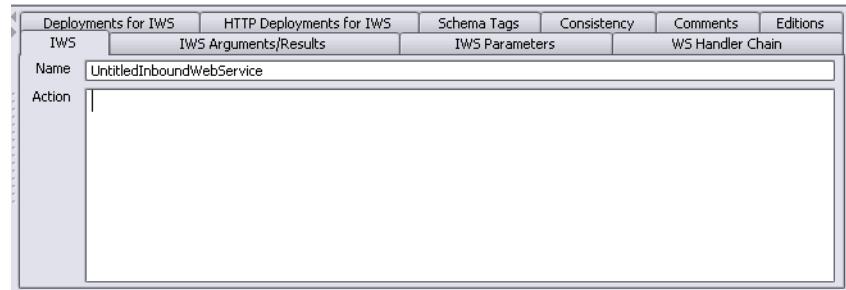
The inbound web service editor contains several tabs that you use to define and inbound web service.

Creating an inbound web service

To create an inbound web service:

1. In Cogility Modeler's tree view, select the model container or package.
2. Click the **Add an Inbound Web Service** button .

An inbound web service editor appears in the content view.



3. In the web service editor, under the **Inbound Web Service** tab, for **Name**, enter a name.
4. In the **Action** field, define the action for the inbound web service.
See “[Web service action](#)” on page 203.
5. On the **IWS Arguments/Results** tab, define the argument and results messages.
 - a. In the **Argument Message** field, define an argument message.
See “[Messages](#)” on page 103.
 - b. In the **Result Message** field, define a result message.
See “[Messages](#)” on page 103.
6. In the **IWS Parameters** tab, define the parameters.
See “[IWS parameters](#)” on page 203.
7. In the **Deployment for IWS** tab, define the deployment.
See “[IWS deployment](#)” on page 205.
8. In the **WS Handler Chain** tab, define the web service handlers.

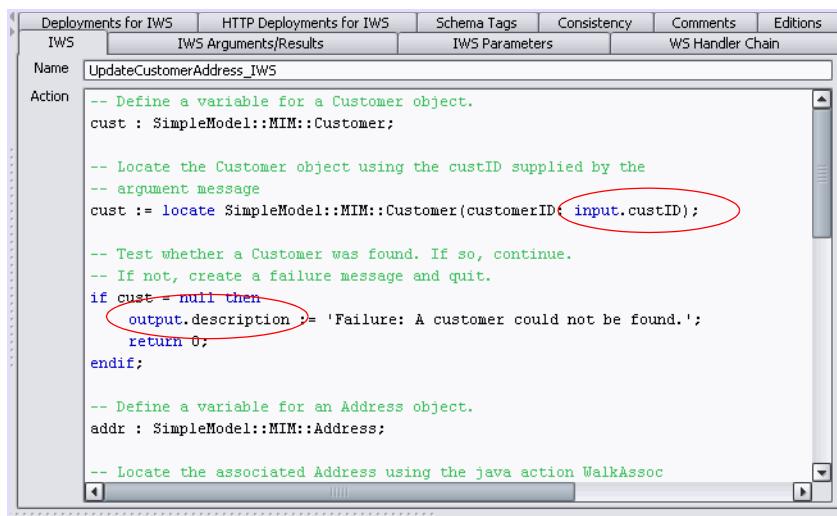
See “[Web service handlers](#)” on page 224.

Web service action

The web service action uses the argument message attribute data and populates the result message attribute data. Web service actions may include the following:

- Database lookups/writes
- Control structures
- Basic exception handling
- Invocation of specialized Java actions

Web service actions use the `input` and `output` reserved keywords to refer to the argument and result message objects, respectively. In the example below, the web service checks the argument message attributes such as `input.custID`, and it returns an error message in the result message as `output.description` where necessary.



```

Deployments for IWS          HTTP Deployments for IWS          Schema Tags          Consistency          Comments          Editions
IWS                         IWS Arguments/Results          IWS Parameters          WS Handler Chain

Name: UpdateCustomerAddress_IWS
Action:
-- Define a variable for a Customer object.
cust : SimpleModel::MIM::Customer;

-- Locate the Customer object using the custID supplied by the
-- argument message
cust := locate SimpleModel::MIM::Customer(customerID: input.custID);

-- Test whether a Customer was found. If so, continue.
-- If not, create a failure message and quit.
if cust = null then
    output.description = 'Failure: A customer could not be found.';
    return 0;
endif;

-- Define a variable for an Address object.
addr : SimpleModel::MIM::Address;

-- Locate the associated Address using the java action WalkAssoc

```

Figure 9-2: The input and output reserved keywords in a web service action

For more information about actions, see the guide, *Using Actions in Cogility Studio*.

IWS parameters

The IWS Parameters tab describes the decoding and encoding methodology for the web service, provides for complex type support, lets you define the top level element and omit nulls in the result.

Argument decoding and result encoding

The Arg Decoding Style field describes how the web service argument XML string should be decoded, and the Result Encoding Style field describes how the web service result XML string should be encoded. If the Arg Decode Style or Result Encoding Style is set to XML String in Sole Parameter, an XSD fragment is required. Cogility Modeler will flag the inbound web service with an inconsistency if the condition is not satisfied and the error message will present an example XSD fragment.

XSD fragment

For both the argument message and the result message, you can document the message's complex types in an XSD fragment. The XSD fragment is for documentation purposes only; it does not provide information for use during model execution. Your inbound web service's actions cannot use the XSD fragment programmatically.

For example, in the figure below, the XSD fragment describes a complex type for the GetCustomerInfo_result output message. The complex type is composed of the Customer elements firstName and lastName and a nested complex type Address associated with the Customer. It also includes the customerID argument attribute.

The web service in this example takes a customerID in an argument message and returns the customer information: the Customer.firstName, Customer.lastName and the associated Address object with its street, city, state and zip members together with the supplied customerID.

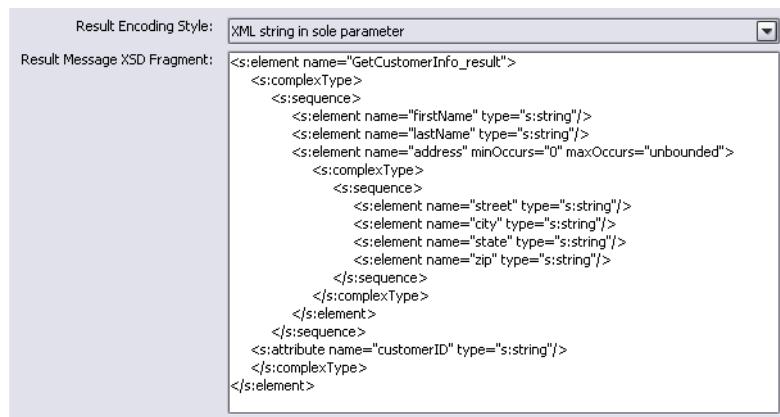


Figure 9-3: XSD fragment for a complex type

Each element of the XSD fragment must be prepended with an **s:** to designate it as a member of the namespace for XSD fragments. Using the namespace is mandatory and prevents conflicts with other XML elements in the WSDL. There is only one namespace for all XSD fragments, designated with an **s:**.

Result top level element

This is the name of the top-level entity expected in the SOAP message. The value is the name given to the tag that describes the result attributes in the result XML. The inbound web service expects the list of response attributes to be described within the tag pair named in this field.

For example, the result message for the example from the figure above might include the following XML. The **<params>** tag is the top level element.

```
<params>
  <attr1>"someAttributeString"</attr1>
  <attr2>"anotherAttributeString"</attr2>
  <attr3>"yetAnotherAttributeString"</attr3>
</params>
```

Omit nulls in result

When you check the box next to Omit Null in Result, the result XML does not include those attributes with null values. For example, if the **<attr2>** value from above were null, the result XML string would read as follows:

```
<params>
    <attr1>"someAttributeString"</attr1>
    <attr3>"yetAnotherAttributeString"</attr3>
</params>
```

Complex type support

For simple types such as strings, integers, longs, and so forth, the inbound web service readily maps the sent and returned objects to its argument or result. Those types are identified in the XML string returned and received. If the inbound web service receives or returns a complex type, it can only treat the complex type data in a single XML string. If you want to create an inbound web service that specifies a complex type for the argument or result, you can create an XML Schema Definition (XSD) artifact for that type and use it in your web service's action semantics. See [Chapter 8, "XSD Artifacts"](#).

IWS deployment

You may deploy the inbound web service in more than one deployment artifact. Usually, the deployment artifact deploys several web services and is created separately from any particular web service. You may also create a new deployment artifact from within an inbound web service. However, if you create a deployment from within an inbound web service, that deployment is associated *under* the web service. Usually, the best practice is to create the deployment object at the level of the model container, then add the web service to that deployment. See ["Creating an IWS deployment" on page 206](#).

A deployment object lets you associate several inbound web service *operations*. Though each web service is modeled separately, when it is grouped with other web services in a deployment, it is referred to as an operation. For example you might have several web service operations to update customer information, and they might be deployed within the same deployment artifact. You might create another deployment object to include web services that update account information. Either of these deployments could include one or more of the same web service operations.

When you select the web service operation in the IWS for Deployment field, its deployment details are displayed. If you change any of these values in the web service, the changes to the deployed

inbound web services are communicated automatically to the deployment and the fields are updated instantly. You can also update the deployment. See “[Updating deployment operations](#)” on page 208.

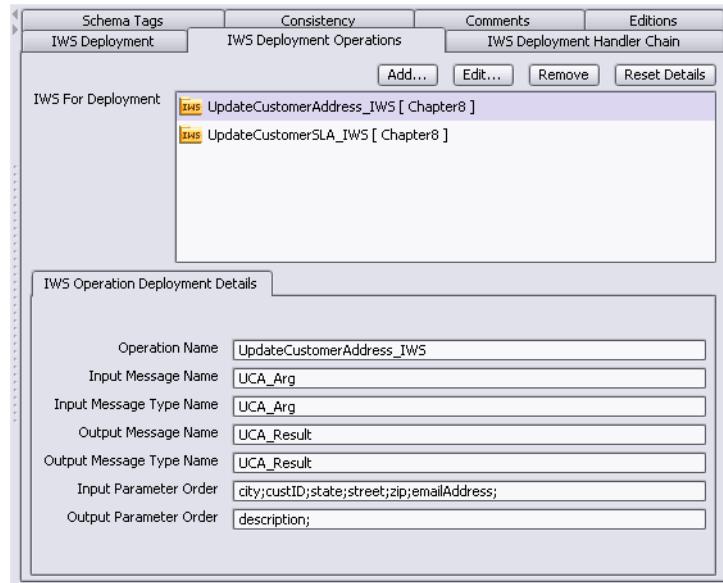


Figure 9-4: Deployment details for a specific inbound web service

The deployment object provides an inbound web service with both its URL and its location on the application server when deployed.

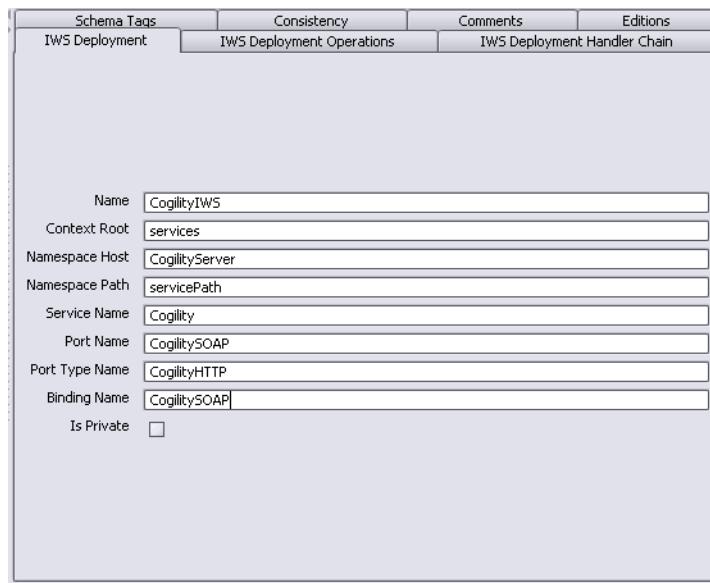


Figure 9-5: IWS deployment definition

Creating an IWS deployment

To create an inbound web service deployment:

1. In Cogility Modeler’s tree view, select the model container or package and click the Inbound Web Service Deployment button .
2. Define the deployment.

See “Defining an IWS deployment” on page 207

Adding an IWS deployment to a handler chain

To add an IWS deployment to a handler chain:

1. In the handler chain editor, in the **Handler Chain Usage** tab, above the **IWS Deployment** field, click **Add**.
2. In the dialog, select the IWS deployment and click **OK**.
3. To define the IWS deployment, click **Edit**.

See “Defining an IWS deployment” on page 207.

Defining an IWS deployment

To define an IWS deployment:

1. In the IWS deployment editor, under the **IWS Deployment** tab, in the **Name** field, enter a name.
2. For **contextRoot**, enter the path at which you access the web service on the application server.
3. For **namespaceHost**, enter the resource name, such as www.cogility.com.
4. For **namespacePath**, enter the path to the deployment’s name space.
The name space path provides for a unique definition of the web service’s XML elements.
5. For **serviceName**, enter a unique name.
6. For **portName**, enter an identifier for SOAP messages.
7. For **portTypeName**, enter the port type name.
Cogility supports only SOAP port types.
8. For **bindingName**, enter an identifier for SOAP messages.
Usually, this matches the portName, above.
9. Check the **Is Private** checkbox if this deployment should be marked as private.
When checked/enabled, Is Private indicates that this Deployment is NOT to be included in the published WSDL.
10. In the **IWS Deployment Operations** tab, add the inbound web service operations.
See “Adding an inbound web service to a deployment (from the deployment)” on page 207.
11. In the **IWS Deployment Handler Chain** tab, define the handler chain for the deployment.
See “Web service handlers” on page 224.

Adding an inbound web service to a deployment (from the deployment)

The best practice for deploying an inbound web service is to first create the web service (see “Creating an inbound web service” on page 202), then add it to an existing deployment object (see “Creating an IWS deployment” on page 206).

To add an inbound web service to a deployment:

1. In the tree view, select the deployment object.
2. In the deployment editor, click the **IWS Deployment Operations** tab.
3. Click **Add**, select the inbound web service operation, and click **OK**.

Adding an inbound web service to a deployment (from the web service)

From within the defined inbound web service, you can add the web service to an existing deployment. This one of two recommended methods for deploying the web service. The other is to add the inbound web service operation from within the deployment object. See “[Adding an inbound web service to a deployment \(from the deployment\)](#)” on page 207.

To add a web service operation to a deployment:

1. In the web service editor, click the **Deployment for IWS** tab, click **Add**.
2. In the dialog, select a deployment and click **OK**.

Updating deployment operations

The IWS Operation Deployment Details tab lists the values generated on the web service operation for its name, its argument and result message names and types, and the attributes (as parameters) for each message. Usually the changes to the deployed inbound web services are communicated automatically to the deployment and the fields are updated instantly. In the unlikely event that you need to update the deployment manually, you can use the **Reset Details** button.

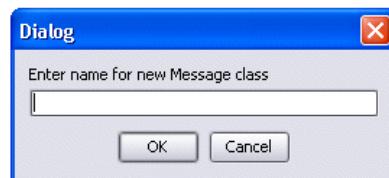
To update a deployment for changes to a web service operation:

1. In the tree view, select the deployment object.
2. Click the **IWS Deployment Operations** tab.
3. Select the web service operation and click **Reset Details**.

Defining the IWS arguments and results messages

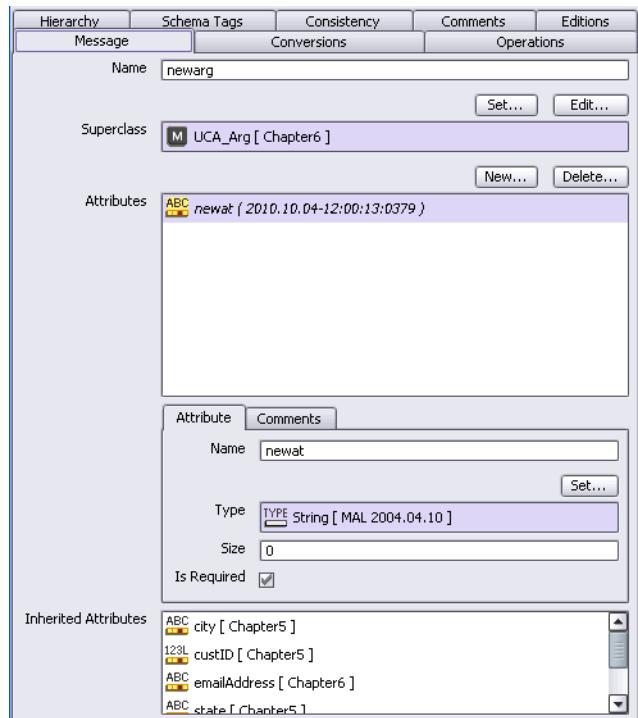
To define the IWS arguments messages:

1. Open the IWS Arguments/Results tab.
2. Click **New** above the Argument messages field.
3. Enter a name for the argument message and click **OK**.



4. Select the new argument, and click **Edit**.
5. Click **Set**.
6. Select the object type from the list and click **OK**.
7. Click **New** above the **Attributes** field.
8. Define the attribute:
 - a. Enter a name for the attribute in the **Name** field.
 - b. Select the type from the **Type** drop-down list.
 - c. Click **OK**.

If the attribute is required, check the **Is Required** checkbox.



If the object type selected has attributes, the inherited attributes are shown in the **Inherited Attributes** field.

To define the IWS results messages:

1. Open the IWS Arguments/Results tab.
2. Click **New** above the **Results messages** field..
3. Enter a name for the result message and click **OK**.
4. Select the new argument, and click **Edit**.
5. Click **Set**.
6. Select the object type from the list and click **OK**.
7. Click **New** above the **Attributes** field.
8. Define the attribute:
 - a. Enter a name for the attribute in the **Name** field.
 - b. Select the type from the **Type** drop-down list.
 - c. Click **OK**.

If the attribute is required, check the **Is Required** checkbox.

Inbound web service invocation

The primary reason for creating webservices is to host them on the J2EE application server so that applications external to Cogility can call them. It is also possible to call an inbound webservice from another part of the model if desired or from an Action Pad test script.

If there is a need to invoke an inbound web service from another inbound webservice or state machine, from a database perspective , the invoked web service will execute in its own database transaction.

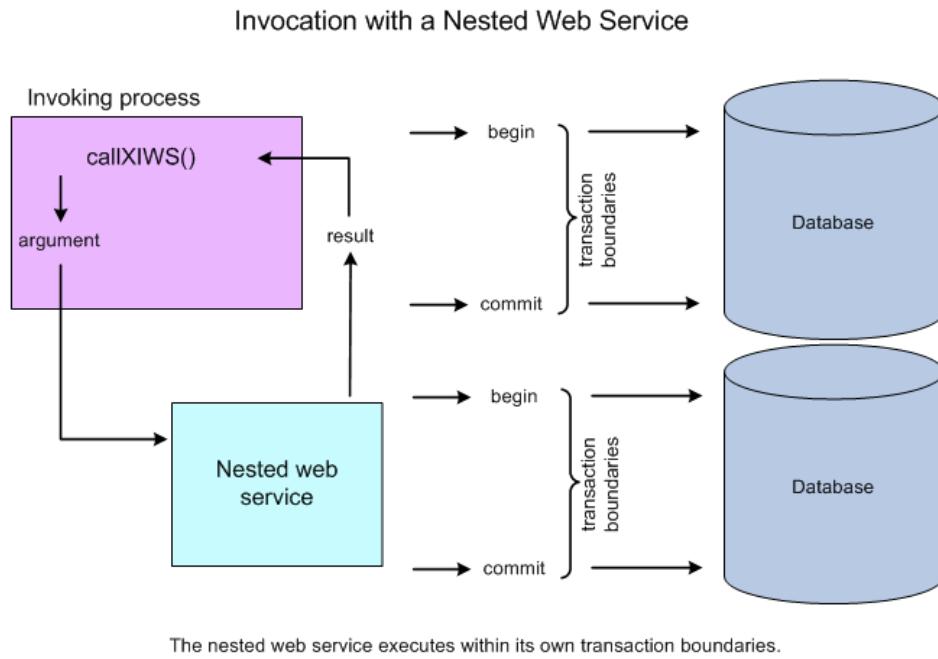


Figure 9-6: Invocation of an inbound web service

callXIWS

To accomplish invocation, use the callXIWS keyword in the actions of a state machine or in another inbound web service. You can use this keyword when you need to do automated testing of several inbound web services, running them from a script in Cogility Action Pad. Whenever you need to invoke inbound web services programmatically, and call them as if from outside the deployed model application, use this method. See “[callXIWS at](#)” on page 35 of the guide, *Using OCLPlus with Cogility Studio*. Here is the syntax:

```
callXIWS <web service name> (<argument msg>);
```

The callXIWS keyword takes two parameters, described below.

Parameter	Type	Description
web service name	String	The String that describes the inbound web service, usually cast with single quotes, and fully-qualified starting with the model container object, followed by the deployment object, and finally the name of the web service. For example, in the WS_Test model container of the CustomerCare deployment, the RegisterCustomer inbound web service is represented as follows: <code>'WS_Test.CustomerCare.RegisterCustomer'</code> .

Parameter	Type	Description
argument message	Message	<p>The argument message for the web service, of the DOM Message type.</p> <p>For this parameter, you create an argument message object, passing in the list of parameters for the argument message. See the example under “Action script” on page 211.</p>

Endpoint

The endpoint for your model’s inbound web services is set with a configuration parameter in the %DCHOME%\Config\Configuration-WebService.txt file. The entry reads as follows:

```
# what to use for the endpoint for Cogility IWS deployments.
# default = IBM WAS (http://localhost:9080)
# e.g., WebLogic = http://localhost:7001

com.ceira.webservicegui.inbound.endpoint.start=http://localhost:7001
```

If no endpoint is specified, the default is the localhost. In the Configuration-WebService.txt file the parameter is set for the BEA WebLogic localhost. If you want to change the specification, copy the parameter to your customConfigurations.txt file and make the change.

Invocation example

You can invoke the inbound web service using an action script, as if you were testing it from the perspective of your systems users as part of a large, automated test program. You run this invocation scenario using the Cogility Action Pad.

Action script

To invoke the web service and pass the argument message, you run the InvokeIWS.ocl script located in the %DCHOME%\Examples\WebService\Invoke directory. That script consists of the following actions. It first declares the `arg` and `res` message objects. To the `arg` message `inValue` attribute, it assigns the string, ‘External Invocation Example’. Then it calls the web service IWS1 with the `callXIWS` keyword passing the `arg` message. The script then prints out the processed `outValue`.

```
arg : Schema:::IWS1:::arg1;
res : Schema:::IWS1:::res1;

arg := new Schema:::IWS1:::arg1(inValue: 'External Invocation Example');
res := callXIWS Schema.IWS1 (arg);

java com:::ceira:::Println(res.outValue);
```

Running the example

In this section, you push the model and use an action semantics script with Cogility Action Pad to invoke the web service.

To run the example:

1. Turn over and push the model .

“[Pushing the model into execution](#)” on page 273.

2. Start the application server.
See “[Application server](#)” on page 270.
3. Run Cogility Action Pad.
See “[Running Cogility Action Pad](#)” on page 80 of the guide, *Model Deployment & Execution in Cogility Studio*.
4. Open and run the file, %DCHOME\Examples\WebService\Invoke\InvokeIWS.ocl.
See “[Opening FilePads](#)” on page 82 and “[Running action semantics in an action pad](#)” on page 90 of the guide, *Model Deployment & Execution in Cogility Studio*.
The final `Println` statement appears in the window.

Outbound web services

An outbound web service invokes a third party or external web service. You can call the outbound web service from within a state machine or from within an inbound web service using actions.

The outbound web service is deployed along with the integration model. The web service it invokes can be running on any platform, and is identified by its web service definition (WSDL) file. The outbound web service artifact invokes the external web service by sending an input SOAP request, encoded from an input argument message artifact. The outbound web service artifact gets its return value from the external web service’s response message, decoded from an output SOAP response.

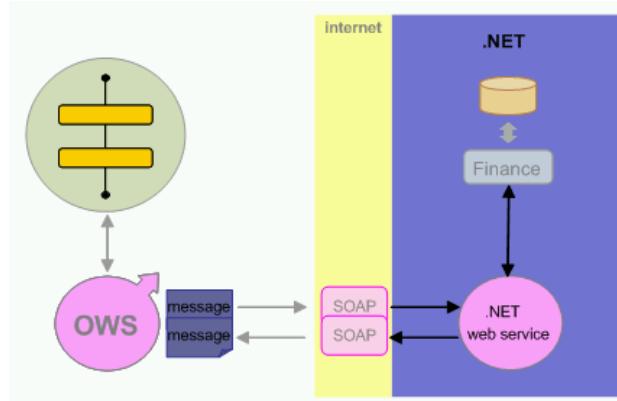


Figure 9-7: Outbound web service

A WSDL file defines the web service called by the outbound web service. Each outbound web service represents one web service operation. If the external web service’s WSDL file defines several operations, there must be an outbound web service for each operation.

Cogility Modeler provides the ability to import a WSDL file and thereby create outbound web services for each of its operations automatically. When you use this method, all of the model objects for the outbound web service are created for you. See “[Automatic WSDL import](#)” on page 212.

You may also create each of the model objects manually. Doing so is described in “[Outbound web service artifacts](#)” on page 214 and thereafter.

Automatic WSDL import

When you import a WSDL file, all of the outbound web service objects needed to call the web service are created automatically.

You may import a WSDL file at any point in your model's tree view, under any package, including the model container or a package named "Outbound web services." See "["Packages" on page 22](#)". When you import the WSDL, it is located in its own package, named according to the web service's name. In the figure below, the WSDL is imported under the model container, and a StockQuote package is created.

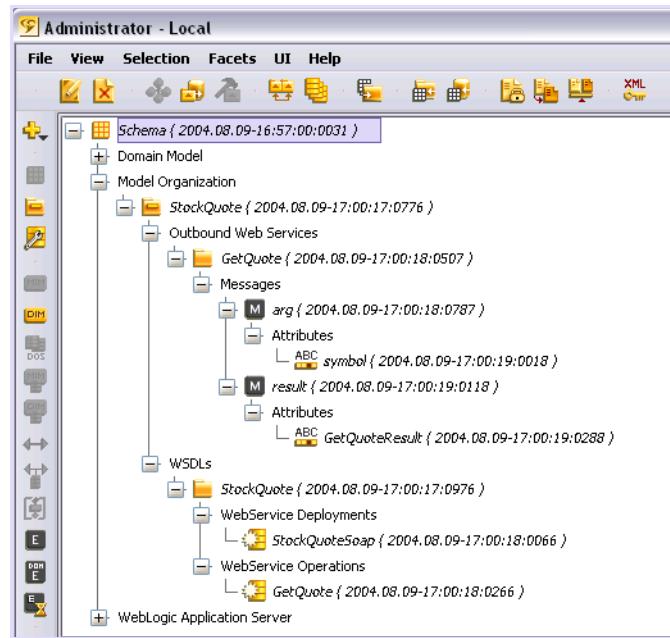


Figure 9-8: Result of an automatic WSDL import

Importing a WSDL from a URL

To import a WSDL from a URL:

1. In the tree view, select the package object under which the WSDL resides.
2. Right-click, and under **Actions**, select **Import WSDL from URL**.
3. Enter the URL for the WSDL file and click **OK**.

The WSDL and the outbound web service is created under a package named for the web service.

4. Test the outbound web service.

See "["Testing an outbound web service" on page 214](#)".

Importing a WSDL from a file

To import a WSDL from a file:

1. In the tree view, select the package object under which the WSDL resides.
2. Right-click, and under **Actions**, select **Import WSDL from File**.
3. Navigate to the WSDL file and click **Open**.
4. Test the outbound web service.

See "["Testing an outbound web service" on page 214](#)".

Testing an outbound web service

You can test an outbound web service before your model is deployed as a J2EE application.

To test an outbound web service:

1. In the tree view, select the outbound web service.
2. Right-click, and under **Actions**, select **Try Outbound Web Service**.
The Exercise OWS console appears with a field labeled according to the name of the argument message attribute.
3. In the input field for each argument parameter, enter a value for the argument message attribute.
4. Under the input field, click **Execute OWS**.
The outbound web service executes and the result string appears under the Returned MSG tab in the output window. You can also review the SOAP argument, SOAP result, and the console output by clicking the corresponding tabs.

Outbound web service artifacts

You may have to manually create an outbound web service if the WSDL file does not import correctly or if you have other considerations. This and the following sections describe how to create and work with outbound web service artifacts.

The outbound web service is defined by its model artifact name that the action of the calling process (either a state machine or an inbound web service) uses to call the outbound web service. An outbound web service represents one operation and identifies one deployment. The definition includes both an argument and a result message.

Creating an outbound web service manually

To create an outbound web service manually:

1. In Cogility Modeler's tree view, select the model container or package, click the  Plus button and select **Add Outbound Web Service**.
2. Define the outbound web service.
See “Defining an outbound web service” on page 214.

Defining an outbound web service

To define an outbound web service:

1. In the outbound web service editor, under the **Web Service Definition** tab, for **Name**, enter a name.
2. Create the argument message.
See “Messages” on page 103.
3. Create the result message.
See “Messages” on page 103.
4. Create the deployment artifact.
See “OWS deployment” on page 215.
5. Create or specify the WSDL object.
See “OWS WSDL” on page 215.
6. Specify the operation.

See “[OWS operation](#)” on page 217.

OWS deployment

The OWS deployment artifact identifies the location at which the web service exists. The deployment is defined by the *endpoint*, the URL for the web service. A single deployment artifact may be associated with multiple outbound web services. Each deployment pertains to a unique service, defined in the WSDL file. An outbound web service is associated with only one deployment artifact.

Within the deployment, you must specify the call order of the argument message attributes. These are passed to the web service in the order you specify with a list separated by commas. You get these attributes from the WSDL file.

The deployment must also specify the web service operation to invoke. You know this from the WSDL file, as well.

Defining an OWS deployment

To define a deployment for an outbound web service:

1. With the outbound web service selected in the tree view, in the content view click the **OWS Deployment** tab.
2. In the **Call Order** field, enter the argument message attributes to be submitted in order, separated by commas.
3. For **Deployment**, click **New**, and enter the name of the deployment for the outbound web service. You can also click **Set**, select the deployment, and click **OK** if you want to use an existing deployment.
The Deployment Definition tab appears.
4. For **Endpoint**, enter the URL for the web service.
5. Specify or create the WSDL object. See “[OWS WSDL](#)” on page 215, below.

Adding an OWS deployment to a handler chain

To add an OWS deployment to a handler chain:

1. In the handler chain editor, in the **Handler Chain Usage** tab, above the **OWS Deployment** field, click **Add**.
2. In the dialog, select the OWS deployment and click **OK**.
3. Click **Edit** to define the OWS deployment.

See “[Defining an OWS deployment](#)” on page 215.

OWS WSDL

A web service definition language (WSDL) file describes a web service to its users. The external web service’s WSDL file may define several operations, and you must create an outbound web service for each operation. Your model needs a WSDL object for the web service’s WSDL file.

You can import a WSDL file directly into your model automatically. Doing so creates all of the objects needed for the outbound web service. See “[Automatic WSDL import](#)” on page 212.

Definition

The WSDL object is defined by the following:

- **WS Namespace** - identifies a grouping of elements within the WSDL document. This is the XML Namespace or "target namespace" from the WSDL document.
- **WS Port** - a single endpoint defined as a combination of a binding and a network address.
- **WS Service** - identifies a set of related ports.



Figure 9-9: WSDL definition for an outbound web service

You can create a WSDL object either at the top of the Cogility Modeler tree or under an outbound web service. If you are using more than one operation from the WSDL file, it makes sense to define its object at the top of the tree. Thereafter, you can associate outbound web services with those operations.

Creating a top level WSDL for outbound web services

When a WSDL contains several operations and you want to model each as an outbound web service, create the WSDL object at the top of the Cogility Modeler tree.

To create a top level WSDL:

1. select the model container or package, click the plus button and select **Add Web Service WSDL**.
 - a. For **Name**, enter a unique name to identify the model object.
 - b. For **WS Service**, enter the service name that identifies one or more ports for the web service.
This value is identified by the `<service name>` tag in the WSDL file.
 - c. For **WS Port**, enter the port type for the web service.
This value is identified by the `<port type>` tag in the WSDL file.
 - d. For **WS Namespace**, enter the location of the web service's namespace.
This value is identified by the `targetNamespace` parameter of the `<definition>` tag in the WSDL file.
- The next two steps are optional. You may want to specify the deployments and operations within the outbound web services themselves rather than through the WSDL object.
2. Click the **WSDL Deployment** tab.
 - a. For **Deployment**, click **New**.
 - b. Enter a name for the deployment and click **OK**.
You can specify several deployments for the WSDL.
3. Click the **WSDL Operation** tab.
 - a. For **Operation**, click **New**.

b. Select an operation and click **OK**.

You can specify several operations from the WSDL. For each operation you must specify the top level element that identifies it, the encoding and decoding for its argument and result messages, and the encoding style. See “[OWS operation](#)” on page 217.

Creating a WSDL object within an outbound web service

When you create the WSDL object within the context of a particular web service when the WSDL contains several operations to be modeled as outbound web services, the WSDL object appears in the Cogility Modeler tree under that outbound web service where it was created and not under the other outbound web services. When the WSDL contains only one operation, and thus only applies to one outbound web service, this is not an issue and you can create the WSDL object under the outbound web service.

To create a WSDL object within an outbound web service:

1. With the web service selected in the tree view, click the **OWS Deployment** tab.
2. In the **Call Order** field, enter the argument message attributes to be submitted in order, separated by commas.
3. Select the **Deployment** that you entered in steps under “[OWS deployment](#)” on page 215.
The Deployment Definition tab appears. If you are using an existing WSDL click **Set**, select the WSDL and click **OK**. You can ignore the remaining steps. Otherwise continue with the next step.
4. To create a new WSDL object, click **New**.
 - a. Enter a unique name to identify the model object and click **OK**.
 - b. In the **WSDL** field, select the WSDL object.
 - c. For **WS Service**, enter the service name that identifies one or more ports for the web service.
This value is identified by the `<service name>` tag in the WSDL file.
 - d. For **WS Port**, enter the port type for the web service.
This value is identified by `<port type>` tag in the WSDL file.
 - e. For **WS Namespace**, enter the location of the web service’s namespace.
This value is identified by the `targetNamespace` parameter of the `<definition>` tag in the WSDL file.

The resulting screen might look like the following.
5. Click the **WSDL Operation** tab.
If you have already created the operation for the WSDL, when you select it under this tab the fields are populated.
 - a. For **Operation**, click **New**.
 - b. Select an operation and click **OK**.

You can specify several operations from the WSDL. For each operation you must specify the top level element that identifies it, the encoding and decoding for its argument and result messages, and the encoding style. See “[OWS operation](#)” on page 217.

OWS operation

An operation is an action supported by the web service. An outbound web service must specify only one operation object. The operation takes the attributes of the argument message as parameters and returns values in the attributes of the response message.

An operation may be specified by multiple outbound web services. That is, the same operation might be used by more than one outbound web service object in the model. However, an operation may be

associated with only one WSDL object. The WSDL is its source, and conflicts result if an operation is associated with more than one WSDL.

Definition

The operation is defined by the following:

- The **encoding style** of the arguments used to call the web service.
- The **top level element** expected by the web service.
- The **decoding style** of the result returned from the web service.
- How the SOAP envelope is filled out.

The argument presentation must match what the web service expects. Likewise, your operation object must know how to decode the response. The operation itself is identified by the top level element. You must also know how to format the SOAP envelope. You can ascertain these from the web service's WSDL file.

Encoding style

This is the method used to encode an XML string inserted into the SOAP request. Options include the following:

- XML string in sole parameter
 - The entire SOAP body is contained in the single parameter of the argument message.
- Parameters as sub-elements
 - This is the standard encoding style.
- Parameters as attributes
- Parameters as either attributes or sub-elements

For outbound web services, only the first two are useful encoding styles. Most web services can be modeled with sub-element parameters. Web services with complex types have parameters that do not map easily to and from message attributes and require single XML string encoding. See “[Complex type support](#)” on page 219.

Top level element

This represents the beginning and ending tags that encapsulate all information pertaining to the parameters of the operation. It describes which operation to invoke. You may specify only one top level element. You must specify a top level element if one of the following encoding styles is used:

- Parameters as attributes
- Parameters as sub-elements
- Parameters as attributes or sub-elements

When sub-element encoding is used, the top level element is the operation name under the SOAP body.

Note: Some .Net web services return an element name which is not mentioned in the WSDL file, such as the operation name concatenated to the word, “Result.”

Decoding element

This is the method used for decoding an argument message out of a SOAP request. Options include the following:

- XML string in sole parameter
The entire SOAPMessage is returned in the single parameter of the result message.
- Parameters as sub-elements
- Parameters as attributes
- Parameters as either attributes or sub-elements

For outbound web services, only the first two are useful decoding styles. Most web services can be modeled with sub-element parameters. Web services with complex types have parameters that do not map easily to and from message attributes and require single XML string decoding. See “[Complex type support](#)” on page 219.

SOAP Envelope

You must specify whether the envelope is a “literal” SOAP call or an “RPC-style” SOAP call.

Creating an Operation

To create the outbound web service operation:

1. In the tree view with the outbound web service selected, in the content view click the **OWS Deployment** tab.
2. For **Operation**, click **Set**.
3. Select the operation, and click **OK**.
4. With the operation selected, click **Edit**.

The operation editor appears in its own window with the Operation Definition tab displays.

- a. Select the **Arg Encoding Style** from the drop-down menu.
- b. Enter the **Arg Top Level Element**.
- c. Select the **Result Decoding Style** from the drop-down menu.
- d. For **Use Literal Encoding**, check the box if the SOAP envelope is to be literal encoded.

Complex type support

When you import a web service’s WSDL file to create an outbound web service, if that WSDL specifies a complex type for the argument or result, you can create an XML Schema Definition (XSD) artifact that gives you programmatic access to that complex type through your model’s action semantics. See [Chapter 8, “XSD Artifacts”](#).

Outbound web service invocation

You can call an outbound web service from within a state machine or from within an inbound web service using actions with the `callOWS` keyword.

`callOWS`

Invokes a modeled outbound web service from another modeled object (for example, an inbound web service or a state machine). This action takes two arguments: the fully-qualified web service name `<web service name>` and the argument message `<argument msg>`. Returns an outbound web service result object. See “[“callOWS at”](#) on page 34 of the guide, *Using Actions in Cogility Studio*. Here is the syntax:

```
callOWS <web service name> (<argument msg>);
```

The WebServiceAction() method takes two parameters, described below.

Parameter	Type	Description
webServiceName	String	The String that describes the outbound web service object, usually cast with single quotes, starting with the model container, followed by the deployment object, and finally the name of the web service object. For example, in the WS_Test model container of the outbound web service is represented as follows: ' <code>WS_Test.GetQuote_OWS</code> '.
arg	Message	<p>The argument message for the web service, of the DOM Message type.</p> <p>For this parameter, you create an argument message object, passing in the list of parameters for the argument message. See the example under "Outbound web service example" on page 221</p>

DOM Document object

In order for the invoking object to work with the web service's response string attributes, the attributes must be contained in a DOM Document object. This is an object type based on the World Wide Web Consortium (W3C) Document Object Model (DOM) specification. Visit <http://www.w3.org> for more information.

XmlLoad()

The invoking object calls the XmlLoad() method and passes to it the variable that holds the XML response string. The oclAsType() method casts the string as a Document.

```
doc : org:::w3c::dom:::Document;
doc := java
com:::ceira:::XmlLoad(xmlResponse).oclAsType(org:::w3c::dom:::Document);
```

XmlGetString()

Following XmlLoad(), above, the invoking object calls XmlGetString() and passes in the document object and the path to the XML attribute to retrieve. The text() method retrieves the text at that location. The individual attributes are then assigned to the Strings declared earlier.

XMLGetString() uses the XPath API (visit <http://www.w3.org> for more information) for picking content out of DOM entities (parsed from XML strings).

```
name := java com:::ceira:::XmlGetString(doc, '/StockQuotes/Stock/Name/
text());
symbol := java com:::ceira:::XmlGetString(doc, '/StockQuotes/Stock/Symbol/
text());
last := java com:::ceira:::XmlGetString(doc, '/StockQuotes/Stock/Last/
text());
change := java com:::ceira:::XmlGetString(doc, '/StockQuotes/Stock/Change/
text());
```

See the "[Outbound web service example](#)" on page [221](#), next.

Outbound web service example

Cogility Studio includes an example model with an outbound web service that invokes a web service called StockQuote. This web service takes a stock ticker symbol and returns the stock price and other information.

To demonstrate the concept of external invocation, Cogility Modeler includes a sample model that calls the outbound StockQuote web service. You can import the model container into Cogility Modeler. The example model is located at
%DCHOME\Examples\WebService\Invoke\InvokeOWS.xml.

Model objects

The model consists of one outbound web service named GetQuote, a WSDL named GetQuote_WSDL, and a deployment named GetQuote_Deployment. Also, the model has two messages, GetQuoteSoapIn_Msg and GetQuoteSoapOut_Msg. By convention, you cannot push a model that does not include a class, so an UntitledMIMClass is also created, though it has no functional purpose. The exploded tree view of the sample model is shown below.

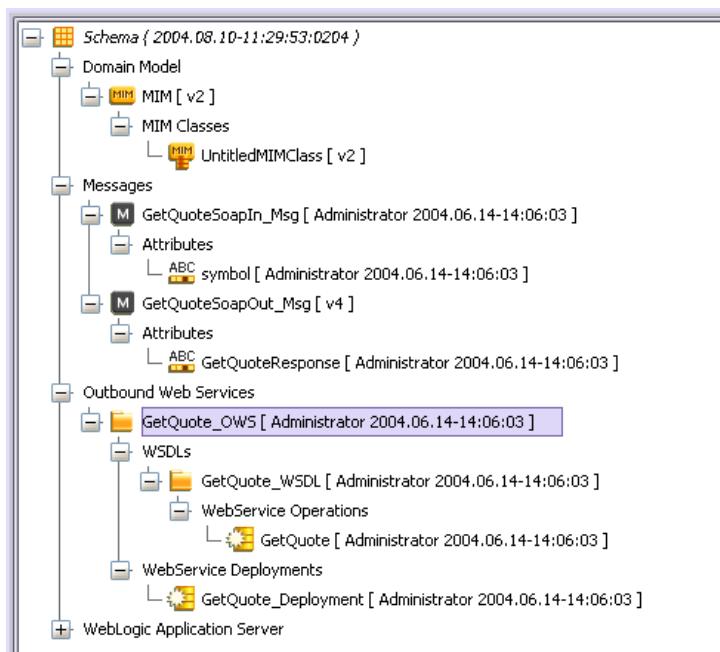


Figure 9-10: Outbound web service invocation example model objects

Actions script

Also included with the example model is an actions script that calls the outbound web service with the WebServiceAction() method. This script, named **InvokeOWS.ocl**, is located in the **%DCHOME\Examples\WebServices\Invoke** directory along with the sample model.

The first four lines declare and initialize variables for the argument and result messages, **GetQuoteSoapIn_Msg** and **GetQuoteSoapOut_Msg**, respectively. The next section sets up some variables for error handling.

```
inMsg : Schema::StockQuote::GetQuote::arg
outMsg : Schema::StockQuote::GetQuote::result;
```

```
inMsg := new Schema::StockQuote::GetQuote::arg();
inMsg.symbol := '<symbol>INTC</symbol>';

success : java :: lang :: Boolean;
success := true;
exceptionCondition : java :: lang :: Boolean;
exceptionCondition := false;
exceptionString : java :: lang :: String;
exceptionString := 'An error has occurred';
```

Note: The values `y`, `yes`, `1` and `t` are also accepted for Boolean true in web service calls.

The `WebServiceAction()` method executes within a try/catch block. The return value of the call is assigned to the `outMsg` result message variable.

```
try
outMsg := callOWS Schema.StockQuote.GetQuote (inMsg);
catch(java :: lang :: Throwable ex)
    exceptionString := ex.toString();
    success := false;
    exceptionCondition := true;
endtry;
```

If the outbond web service is successful, the next block sets up some variables to hold the response attributes from the web service. It then assigns the response XML string to the `xmlResponse` variable with the `GetQuoteResult()` method.

```
if success then
    name : java :: lang :: String; symbol : java :: lang :: String;
    change : java :: lang :: String; last : java :: lang :: String;

    xmlResponse : java :: lang :: String;
    xmlResponse := outMsg.GetQuoteResult;
```

In order for the calling process to work with the response string attributes, the XML string must be contained in a DOM Document object. The calling process uses the `XmlLoad()` method and passes to it the `xmlResponse` variable that holds the XML string. The `oclAsType()` method casts the string as a Document object.

```
doc : org::w3c::dom::Document;
doc := java com :: ceira :: XmlLoad(xmlResponse).oclAsType(org::w3c::dom::Document);
```

Now the calling process can call `XmlGetString()` and pass in the document object and the path to the XML attribute to retrieve. The `text()` method retrieves the text at that location. The individual attributes are then assigned to the Strings declared earlier.

```
name := java com :: ceira :: XmlGetString(doc, '/StockQuotes/Stock/
Name/text());
symbol := java com :: ceira :: XmlGetString(doc, '/StockQuotes/Stock/
Symbol/text());
last := java com :: ceira :: XmlGetString(doc, '/StockQuotes/Stock/
Last/text());
change := java com :: ceira :: XmlGetString(doc, '/StockQuotes/Stock/
Change/text());
```

Finally, the variables are concatenated and printed to the console. If an exception was encountered with the web service's response, an `exceptionString` is printed instead.

```
java com::ceira::Println('\n\n\nName:\t\t'.concat(name)
    .concat('\nSymbol:\t').concat(symbol).concat('\nLast:\t\t')
    .concat(last).concat('\nChange:\t').concat(change))

endif;

if exceptionCondition then
    java com ::ceira :: Println(exceptionString);
endif;
```

Example execution

You can run this outbound web service in three ways:

- Testing it from Cogility Modeler before the model is pushed
This is a convenient way to debug an outbound web service before deploying the model application. See “[Testing an outbound web service](#)” on page 214.
- Running it with Cogility Web Service Exerciser
This utility lets you import an outbound web service and test it, even before you have built a model around the outbound web service. See “[Running the example with Cogility Web Service Exerciser](#)” on page 223.
- Running it with Cogility Action Pad
This tool lets you simulate calling the outbound web service as if from another process like a state machine or inbound web service. See “[Running the example with Cogility Action Pad](#)” on page 224.

Running the example with Cogility Web Service Exerciser

For this example, you load the outbound web service from the model and run it in Cogility Web Service Exerciser.

To run the example with Cogility Web Service Exerciser:

1. Turn over and push the model.
See “[Pushing the model into execution](#)” on page 273.
2. Start the application server.
See “[Application server](#)” on page 270.
3. Run Cogility Web Service Exerciser.
See “[Running Cogility Web Service Exerciser](#)” on page 137 of the guide, *Model Deployment & Execution in Cogility Studio*.
 - a. Import the outbound web service, GetQuote.
See “[Loading an outbound web service](#)” on page 139 of the guide, *Model Deployment & Execution in Cogility Studio*. All of the outbound web services included with the current model are loaded into the console. Because GetQuote is the only outbound web service in the model, it appears highlighted in the console, and its attributes and parameters appear in the fields and lists.
 - b. In the field next to the **Parameters** drop-down list, enter a stock ticker symbol such as GE.
 - c. Click **Run**.

Cogility Web Service Exerciser invokes the web service that returns a result message and displays in the console window.

Running the example with Cogility Action Pad

Cogility Action Pad is a multi-functional utility for writing and testing actions. Here you use it to call an outbound web service with the script described in “[Actions script](#)” on page 221.

To run the example with Cogility Action Pad:

1. Turn over and push the model .
“[Pushing the model into execution](#)” on page 273.
2. Start the application server.
See “[Application server](#)” on page 270.
3. Run Cogility Action Pad.
See “[Running Cogility Action Pad](#)” on page 80 of the guide, *Model Deployment & Execution in Cogility Studio*.
4. Open and run the file, %DCHOME\Examples\WebService\Invoke\InvokeOWS.odl
See “[Opening FilePads](#)” on page 82 and “[Running action semantics in an action pad](#)” on page 90 of the guide, *Model Deployment & Execution in Cogility Studio*.
The final `Println` statement appears in the window.

Web service handlers

Web service handlers are Java classes that manipulate a request, response, or error message. Specifying handlers is optional for web service operations, and is usually used in more advanced situations. Cogility supports the modeling of externally supplied handlers written in Java, and their application to individual web services or deployments of several web services. A Cogility handler model object identifies, initializes, and activates a JAX-RPC handler at runtime.

You implement web service handlers as handler chains in your inbound or outbound web services or in a deployment of several web services. A handler chain may be a series of handlers performed on either the request or response message. The handler chain may consist of zero or more handler objects. You can also use a handler chain that does not include any handler objects, but specify the handler logic with its own request and response actions.

Creating a handler chain

To create a handler chain for your deployment or web service:

1. In the deployment or web service editor, click the **Handler Chain** tab.
2. Click **New** to create a new handler chain.
3. In the dialog, enter a name for the handler chain and click **OK**.
4. Click **Edit** to define the handler chain.

See “[Defining a handler chain](#)” on page 225.

Setting a handler chain

To create a handler chain for your deployment or web service:

1. In the deployment or web service editor, click the **Handler Chain** tab.



2. Click **Set** to set an existing handler chain.
3. In the dialog, select the handler chain and click **OK**.
4. Click **Edit** to define the handler chain.
See “[Defining a handler chain](#)” on page 225.

Adding a web service to a handler chain

To add a web service to a handler chain:

1. In the handler chain editor, in the **Handler Chain Usage** tab, above the **Web Service** field, click **Add**.
2. In the dialog, select the web service (IWS or OWS) and click **OK**.
3. Click **Edit** to define the web service.
See “[Creating an inbound web service](#)” on page 202. See “[Defining an outbound web service](#)” on page 214.

Defining a handler chain

To define a handler chain for your deployment or web service:

1. In the handler chain editor, in the **Handler Chain** tab, in the **Name** field, enter a name.
2. In the **Handler** field, define the handler objects.
See “[Handler objects](#)” on page 225.
3. In the **Request Actions** tab, define the request actions, if needed.
See “[Request and response actions](#)” on page 227.
4. In the **Response Actions** tab, define the response actions, if needed.
See “[Request and response actions](#)” on page 227.
5. In the **Handler Chain Usage** tab, add deployments and web services as needed.
See “[IWS deployment](#)” on page 205. See “[OWS deployment](#)” on page 215.

Handler objects

Zero or more handler objects may comprise a handler chain. You can reuse these objects in different handler chains for different deployments and web services. Each handler, like the handler chain itself, can define request and response actions.

An example handler definition of a handler is shown below. It includes a handler class specification and configuration XML. The handler class is an implementation of the Java javax.xml.rpc.Handler interface and must be included in the current project configuration (see “[Project Configuration Editor](#)” on page 51 of the guide, *Model Deployment & Execution in Cogility Studio*). The configuration

XML, shown in the Config XML field, is a simple XML string that configures the handler class by specifying the key-value pairs passed in as initialization data.

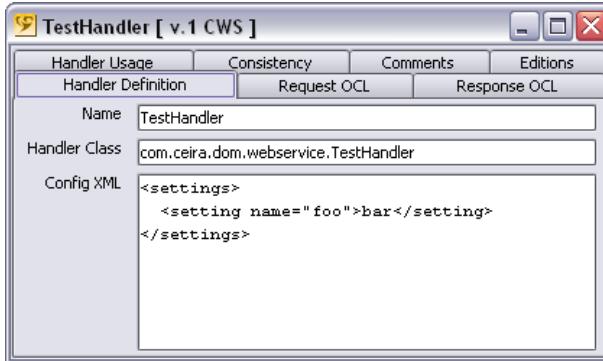


Figure 9-11: Web service handler object definition

Following standard JAX-RPC semantics, the handlers in a handler chain are executed in order for a request, and in reverse order for a response. For example, consider a web service with a handler chain consisting of three handlers, "A", "B", and "C" listed in that order. When applied to a typical request/response, the handlers are called in order (A,B,C) for the request, then in reverse order (C,B,A) for the response.

Creating a handler

To create a handler in a handler chain:

1. In the handler chain editor, in the **Handler Chain** tab, click **New**.
2. In the dialog, enter a name for the handler and click **OK**.
3. To define the handler, click **Edit**.

See “[Defining a handler](#)” on page 226.

Adding handlers to a chain

To add a handler to a handler chain:

1. In the handler chain editor, in the **Handler Chain** tab, click **Add**.
2. In the dialog, select the handler and click **OK**.
3. To define the handler, click **Edit**.

See “[Defining a handler](#)” on page 226.

Defining a handler

To define a handler chain for your deployment or web service:

1. In the handler editor, in the **Handler** tab, in the **Name** field, enter a name.
2. In the **Handler** tab, in the **Handler Class** field, specify the fully qualified Java class name of your existing handler class.

Use this field to specify an existing Java handler class. The Java class must be included in the current project configuration (see “[Project Configuration Editor](#)” on page 51 of the guide, *Model Deployment & Execution in Cogility Studio*), and must implement the javax.xml.rpc.Handler interface.

Note that if your handler chain uses handler objects, the handler object must reference a class as described above. If your handler chain does not include handlers that reference

classes that implement the Handler interface, you can specify the handler logic in the handler chain's request or response action fields. See “[Request and response actions](#)” on page 227.

3. In the **Config XML** field, enter the configuration XML.

This is a simple XML string that configures the handler class by specifying the key-value pairs passed in as initialization data.

4. In the **Request Actions** tab, define the request actions, if needed.

See “[Request and response actions](#)” on page 227.

5. In the **Response Actions** tab, define the response actions, if needed.

See “[Request and response actions](#)” on page 227.

6. In the **Handler Chain Usage** tab, add deployments and web services as needed.

See “[IWS deployment](#)” on page 205. See “[OWS deployment](#)” on page 215.

Setting the order of handlers in handler chains

To set the order of a handler in a handler chain:

1. In the handler chain editor, in the **Handler Chain** tab, select the handler object.
2. To move the handler up in the call order, click **Move Up**.
3. To move the handler down in the call order, click **Move Down**.

The handlers in a handler chain are executed in order for a request, and in reverse order for a response. For example, consider a web service with a handler chain consisting of three handlers, “A”, “B”, and “C” listed in that order. When applied to a typical request/response, the handlers are called in order (A,B,C) for the request, then in reverse order (C,B,A) for the response.

en applied to a typical request/response, the handlers are called in order (A,B,C) for the request, then in reverse order (C,B,A) for the response.

Request and response actions

Your handler or handler chain can specify pre- and post- request logic, written with actions. In a handler chain, the pre-request actions are executed before the list of handlers is invoked to process a request or response message; the post-request actions are executed after the handler chain is invoked to process a request or response message. The actions have access to the message context supplied to the owning handler chain instance. The purpose of these actions is to act as a hook for arbitrary message access or modification, outside of the handlers or the handler chain.

The message context is the standard JAX-RPC means by which handlers access and manipulate web service requests, responses, and faults. The SOAP MessageContext encapsulates both a SOAP Message and a data structure for storing arbitrary data. As a hook for custom message modification, Cogility supports access to the message context before and after a handler chain is executed, and before and after each handler in the handler chain is executed. The message context is accessible in the actions—on both handler and handler chain model objects—that execute before and after requests or responses. Arbitrary changes to the SOAP Message header fields and/or body fields can thereby be performed from the actions in the handler chain and handlers.

You can use actions in the request and response action fields instead of the logic supplied by individual handler objects (see “[Handler objects](#)” on page 225). The handlers of a handler chain must implement the javax.xml.rpc.Handler interface, but the handler chain itself does not have this requirement. If you do not wish to implement existing handlers from Java classes, you can define handler logic in the handler chain's request and response action fields.

Creating request and response actions

To create request or response actions:

1. In the handler chain editor, click either the **Request Action** or **Response Action** tab.
2. In either the **Pre** or **Post** fields, enter the actions for your handler chain.

See the guide, *Using Actions in Cogility Studio*.

SOAP versus HTTP

The choice of whether to use SOAP or HTTP web services (or a combination of both) on a specific project depends on the details of the integration. There is no single right answer. Cogility allows both types of web services in a model. The needs of the project, in terms of complexity required, can help determine which type of web service to use for a particular task.

Using SOAP for web service calls generally involves a high level of coding complexity. Using HTTP web service eliminates SOAP, thereby providing less coding complexity to make web service calls. There is also a significant reduction in the amount of data that needs to be transported.

SOAP is a more extensive specification with support for features needed to solve issues that can be encountered in complex integration projects. For example, SOAP allows additional information in 'headers' and the ability to insert 'handler chains' into the communications. WSDL provides a standard way to define SOAP Web Services and for tools to consume them. With HTTP Web Services these things need to be defined on a per-project basis.

Often, many of the complexities of SOAP are not needed and HTTP offer an easier way to achieve integration. For example, with WSDL and UDDI, you get not only a description of the XML data payload but of the service operations that use this data and a way to search the registry to find services that match a certain criterion. With an HTTP approach, the XML schema would still be described in a standardized format (.xsd files), however, there might be additional documents (project specific) that describe the service operations and the endpoints that host these services. An increasingly sophisticated set of AJAX based toolkits are now widely available. These tools typically provide a rich GUI/user interaction that is web browser based and they are adept at HTTP level communications.

HTTP web services offer a lighter weight alternative way of communicating between systems. Instead of SOAP over HTTP, communications are done over HTTP directly using the HTTP GET, PUT, POST and DELETE methods.

Inbound web services using SOAP over HTTP

An inbound web service provides access to an enterprise application via a standard interface such as HTTP. You can invoke an inbound web service from within another web service or from within a state machine. The web service is deployed along with the integration model in its J2EE application. Following deployment, when the inbound web service receives a SOAP request it converts the

request into an argument message and executes the web service's actions; it then converts a result message into a SOAP response for the requestor.

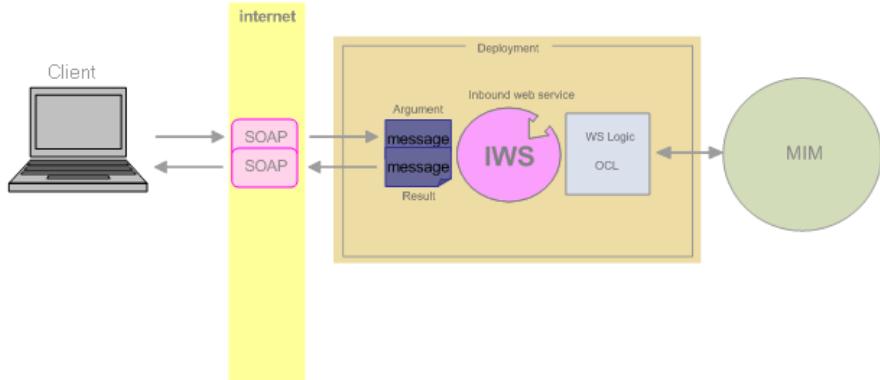
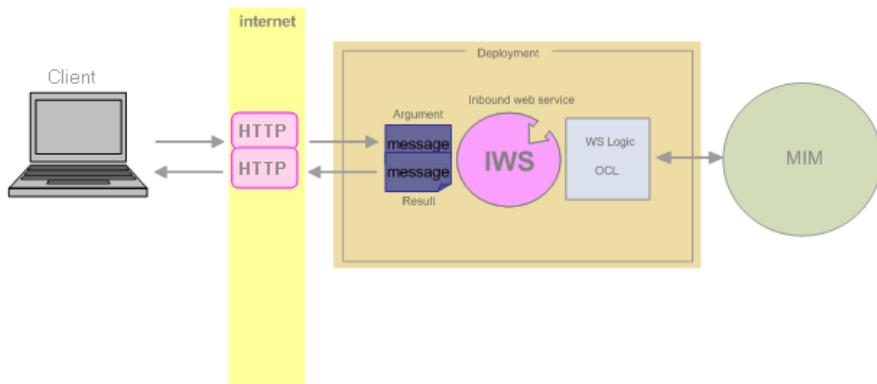


Figure 9-12: Inbound web service

Inbound web services using HTTP

HTTP web services send and receive data from remote servers using HTTP operations directly. To get data from the server, you use an HTTP GET; to send new data to the server, you use HTTP POST. You can modify existing data and delete data, using HTTP PUT and HTTP DELETE. The “verbs” built into the HTTP protocol (GET, POST, PUT, and DELETE) map directly to application-level operations for receiving, sending, modifying, and deleting data.

You can invoke an HTTP inbound web service from within another web service or from within a state machine. The web service is deployed along with the integration model in its J2EE application. Following deployment, when the inbound web service receives a HTTP request it converts the request into an argument message and executes the web service's actions; it then converts a result message into a HTTP response for the requestor.



HTTP inbound web services contain four action fields for which OCL can be specified. This corresponds to the four actions that can be specified for HTTP communications: GET, PUT, POST and DELETE.

The OCL in HTTP inbound web services is written in terms of input argument and output result attributes. The OCL does not contain protocol specific details. The deployment of these web services to the J2EE container is automatically handled by the Push process.

Interchanging HTTP and SOAP Web Services

A particular SOAP web service can be deployed to multiple SOAP endpoints. Similarly, an HTTP web service can be deployed to more than one HTTP deployment. Additionally, SOAP web services can be deployed to both SOAP and HTTP deployments. This allows existing SOAP web service functionality to be deployed via the HTTP protocol. When a SOAP web service is connected to an HTTP deployment, a new HTTP web service exists. The single OCL action of the SOAP web service maps to the POST action of the equivalent HTTP web service.

The XML data payload itself remains unchanged when moving from SOAP to HTTP.

Modeling HTTP operations inbound web services

REST versus RPC Architecture

In simple terms, RPC (Remote Procedure Call) style can be thought of as a few end-points with many operations on each one. REST on the other hand has many end-points (URLs) and the same 4 operations (GET, PUT, POST and DELETE) on each one.

If you need to architect a solution that is more 'Resource Oriented' and less 'Remote Procedure Call', it is possible that up to four OCL action fields may need to be specified. For example, a resource (Customer or Loan) can be retrieved (GET), modified (PUT), deleted (DELETE) or created (POST). In many instances, however, there may only be one OCL action. For example, the single action that needs to be taken when a Procedure (Web Service) is invoked such as 'CreateCustomer'.

Even though HTTP Web Services are often used to implement REST style architectures, they can just as easily be used with an RPC style. Conversely REST or REST-RPC hybrid approaches are possible with SOAP Web Services.

"relativeURL" attribute for HTTP Web Services

The following example illustrates how web services might be arranged using the "REST" style. This example creates, updates, deletes and queries Customers that exist in the database.

For RPC style, you might have a few web service operations, all deployed to one Endpoint and performing a specific action.

```
http://<host>:<port>/<Endpoint>/CreateCustomer(custDetails)  
http://<host>:<port>/<Endpoint>/GetCustomer(id)  
http://<host>:<port>/<Endpoint>/ModifyCustomer(custDetails)  
http://<host>:<port>/<Endpoint>/DeleteCustomer(id)  
http://<host>:<port>/<Endpoint>/GetInactiveCustomers()
```

For REST you may choose to expose one for each customer in the database.

```
http://<host>:<port>/<EndpointRoot>/Customers/1234  
http://<host>:<port>/<EndpointRoot>/Customers/5678
```

You would continue on with one url for each customer in the database.

GET retrieves the specified Customer

PUT updates details on the specified Customer

DELETE deletes the specified Customer

It would not be feasible to have a separate web service for each Customer endpoint. To solve this using one web service, HTTP Web Services can have an optional argument called 'relativeURL'. What appears to the web service client as multiple URLs are mapped to the same single web service (called 'Customers'). The last part of the URL is stripped out by Cogility and passed in to the web service as an input parameter named 'relativeURL'.

Additional input arguments can always be specified even if 'relativeURL' is used. For example for the 'Customers' web service, you also need inputs for 'customerName', 'address' and so on.

Creating a new Customer can be accomplished through the same Web Service - a client would do a POST operation and pass in 'customerName', 'address' etc. Alternatively another web service, called 'CustomerRegistry' for example, can be created for operations like 'CreateCustomer' and 'GetInactiveCustomers'.

Data Format for HTTP web services

The data payload for an HTTP web service can be XML or JSON. The reason for the JSON data format support is that HTTP Web Services are often paired with browser based AJAX clients and these clients often prefer to deal with JSON which is JavaScript's native format. JSON can provide a reduction in the amount of wire data, compared to XML and it is much faster on the browser side when compared to XML data parsing. However, XML comes with a standardized way of expressing its data structure (schema) and JSON currently does not.

Complex Data Payloads

The incoming arguments and outgoing return values of an HTTP web service can be simple data values or can be complex nested data structures. During the execution of the web service, Cogility automatically manages the top level attributes, but if there are nested data sub-structures in any of the attributes, the OCL must manage that.

This is true for both the XML and JSON data formats.

XML Data Format

This section provides an example of the XML data format. For this example, assume that you have a web service called 'CustomerInfo' that has three return values 'name', 'age' and 'id'. The OCL in the web service can set these three attributes to the correct values and Cogility encodes the XML string, which might look as follows.

(<BoldText> = web service name, <i>italic text

```
<CustomerInfo>
  <name>Manfred Smelch</name>
  <age>32 </age>
  <id>12345</id>
</CustomerInfo>
```

Now, assume that there is another attribute called 'addresses' which has a nested sub-structure underneath it, as follows:

```
<CustomerInfo>
  <name>Manfred Smelch</name>
```

```
<age>32 </age>
<id>12345</id>
<addresses>
  <billing>
    <street>123 Main Street</street>
    <city>Santa Clara</city>
    <state>CA</state>
    <zip>95050</zip>
  </billing>
  <delivery>
    <sameAsBilling>false</sameAsBilling>
    <street>Downtown Warehouse</street>
    <city>Santa Clara</city>
    <state>CA</state>
    <zip>95050</zip>
  </delivery>
</addresses>
</CustomerInfo>
```

In this case, the OCL in the web service is responsible for creating the string:

```
"<billing>
  <street>123 Main Street</street>
  <city>Santa Clara</city>
  <state>CA</state>
  <zip>95050</zip>
</billing>
<delivery>
  <sameAsBilling>false</sameAsBilling>
  <street>Downtown Warehouse</street>
  <city>Santa Clara</city>
  <state>CA</state>
  <zip>95035</zip>
</delivery>"
```

This string becomes the value of the attribute 'addresses'.

JSON Data Format

For creating and parsing JSON encoded strings, Cogility includes utilities from json.org and these can be used as appropriate inside the OCL actions of the web service. Please refer to www.json.org for full details.

The following code snippet shows one example of how these utilities can be used.

```
JSONObject billingAddr = new JSONObject();
  billingAddr.put("street", "123 Main Street");
  billingAddr.put("city", "Santa Clara");
  billingAddr.put("state", "CA");
  billingAddr.put("zip", "95050");

  JSONObject deliveryAddr = new JSONObject();
  deliveryAddr.put("sameAsBilling", false);
  deliveryAddr.put("street", "Downtown Warehouse");
  deliveryAddr.put("city", "Santa Clara");
  deliveryAddr.put("state", "CA");
  deliveryAddr.put("zip", "95050");
```

```
JSONObject topJsonObj = new JSONObject();
topJsonObj.put("billing", billingAddr);
topJsonObj.put("delivery", deliveryAddr);
```

This produces the JSON coded string that represents the nested data structure for the attribute 'addresses' of the web service 'CustomerInfo' described in the previous section.

```
{"delivery": {
    "street": "Downtown Warehouse",
    "sameAsBilling": false,
    "state": "CA",
    "zip": "95050",
    "city": "Santa Clara"},

"billing": {
    "street": "123 Main Street",
    "state": "CA",
    "zip": "95050",
    "city": "Santa Clara"}
}
```

Similarly, utilities exist in the json.org package for converting String into JSON Objects. For example:

```
topLevelObject = XML.toJSONObject(jsonEncodedString);
topLevelObject.getJSONObject("someKey");
```

OCL Syntax

Like SOAP Web Services, HTTP Web Services are primarily invoked by front-end clients but they can also be invoked through OCL. This OCL can be in the Model or can be part of Action Pad test scripts.

```
callHTTPPIWS GET <WebServiceName>(<parameters>);
callHTTPPIWS PUT <WebServiceName>(<parameters>);
callHTTPPIWS POST <WebServiceName>(<parameters>);
callHTTPPIWS DELETE <WebServiceName>(<parameters>);
callHTTPPIWS GET <WebServiceName>(<parameters>) at (endpointURL);
callHTTPPIWS PUT <WebServiceName>(<parameters>) at (endpointURL);
callHTTPPIWS POST <WebServiceName>(<parameters>) at (endpointURL);
callHTTPPIWS DELETE <WebServiceName>(<params>) at (endpointURL);
```

When the 'at' part of the syntax is omitted, the web service is assumed to be deployed on a server on localhost.

Creating an inbound HTTP web service

To create an inbound HTTP web service:

1. In Cogility Modeler's tree view, select the model container or package.
2. Select Domain Modeling from the menu list.
3. Click the **Add HTTP Inbound Web Service** button  .
An HTTP inbound web service editor appears in the content view.
4. Define the web service.
See “[Creating an inbound web service](#)” on page 202.

Defining an HTTP inbound web service

To define an HTTP inbound web service:

1. Open the HTTP IWS tab.
2. Enter a name in the Name field.
3. Enter the appropriate GET, POST, PUT, and DELETE actions for the web service.

The screenshot shows the 'HTTP IWS' tab in the Cogility Studio interface. The tab has several tabs at the top: 'HTTP IWS' (which is selected), 'IWS Arguments/Results', 'HTTP Deployments for IWS', 'Consistency', 'Comments', and 'Editions'. Below these tabs, there are four sections, each representing an HTTP method:

- GET Action:** Handles a GET request for a note. It checks if a note ID is provided in the URL. If not, it returns an error message. If a note ID is provided, it retrieves the note from the database and returns its details.
- POST Action:** Handles a POST request to create a new note. It checks if the URL is valid. If not, it creates a new lender and note object, sets their names, amounts, and deposit dates, and saves them to the database.
- PUT Action:** Handles a PUT request to update an existing note. It checks if a note ID is provided in the URL. If not, it returns an error message. If a note ID is provided, it updates the note's details in the database.
- DELETE Action:** Handles a DELETE request to delete a note. It checks if a note ID is provided in the URL. If not, it returns an error message. If a note ID is provided, it deletes the note from the database.

The OCL code for each action is as follows:

```
--D Nov 06 2007, 1234, hdesai. Created.  
LL : java::lang::Long;  
noteId : java::lang::Long := LL.parseLong(input.relativeURL);  
NOTE : MyModel::MIM::LoanNote := locate MyModel::MIM::LoanNote(id: noteId);  
if (NOTE = null) then  
    output.operationResult := 'Could not locate a Loan Note with id '.concat(noteId);  
else  
    output.id := noteId;  
    output.duration := NOTE.durationToNow..
```

```
--D Nov 06 2007, 1234, hdesai. Created.  
if ((input.relativeURL = null) or (input.relativeURL.equalsIgnoreCase(''))) then  
    LENDER : MyModel::MIM::Lender := locate MyModel::MIM::Lender(name: input.lenderName);  
    if (LENDER = null) then  
        LENDER := new MyModel::MIM::Lender(name: input.lenderName);  
    endif;  
    NOTE : MyModel::MIM::LoanNote := new MyModel::MIM::LoanNote(amount: input.amount, du...  
    SYS : java::lang::System;  
    timenow : java::lang::Long := SYS.currentTimeMillis();  
    NOTE.depositDate := new com:cogility::Timestamp(timenow);  
    NOTE.status := 'ACTIVE';  
    new LENDER to loanNote(NOTE);  
    output.id := NOTE.id;
```

```
--D Nov 06 2007, 1234, hdesai. Created.  
LL : java::lang::Long;  
noteId : java::lang::Long := LL.parseLong(input.relativeURL);  
NOTE : MyModel::MIM::LoanNote := locate MyModel::MIM::LoanNote(id: noteId);  
if (NOTE = null) then
```

```
--D Nov 06 2007, 1234, hdesai. Created.  
LL : java::lang::Long;  
noteId : java::lang::Long := LL.parseLong(input.relativeURL);  
NOTE : MyModel::MIM::LoanNote := locate MyModel::MIM::LoanNote(id: noteId);  
if (NOTE = null) then
```

Any errors in the OCL syntax are shown on the Consistency tab. Be sure to check this tab for errors.

HTTP Deployment

These are similar to the SOAP style Deployments in that their primary function is to group multiple web services at one endpoint. HTTP Deployments have only two attributes.

- **Name:** The name is part of the endpoint (URL). For example, an HTTP Inbound Web Service named 'foo' and connected to the HTTP Deployment 'Accounting' could be accessed via the URL `http://<host_or_ip>:<port>/Accounting/foo`.
- **Data Format Type:** The 'XML' and 'JSON' data formats are supported. This attribute controls how the web service return values are encoded.

For example, a web service called 'GetAddress', that returns values for 'street', 'city' and 'zip' returns the following string when coded as XML:

```
<GetAddress>
  <street>123 Main Street</street>
  <city>Anytown</city>
  <zip>90210</zip>
</GetAddress>
```

When the data format is set to JSON, it returns:

```
{GetAddress:{street:5 Main Street, city:Anytown, zip:90210}}
```

Creating an HTTP deployment

To create an HTTP deployment:

1. In Cogility Modeler's tree view, select the model container or package.
 2. Select Domain Modeling from the right-click menu list, or click the plus button .
 3. Click the **Add HTTP Deployment** button .
- An HTTP deployment editor appears in the content view.
4. Open the HTTP deployment tab.
 - a. Enter a name in the **Name** field.
 - b. Select the data format type from the drop-down list.
 - c. Check the **Is Private** checkbox, if desired.
 - When checked/enabled, Is Private indicates that this deployment is NOT to be included in the published WSDL.
 5. Open the IWS Deployment Operations tab.
 - a. Click **Add**.
 - b. Select the object to deploy from the list and click **OK**.

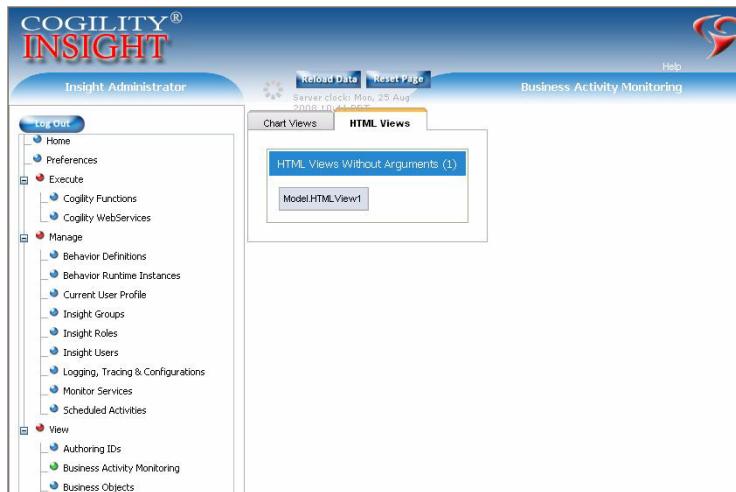


HTML Views and Charts

HTML Views

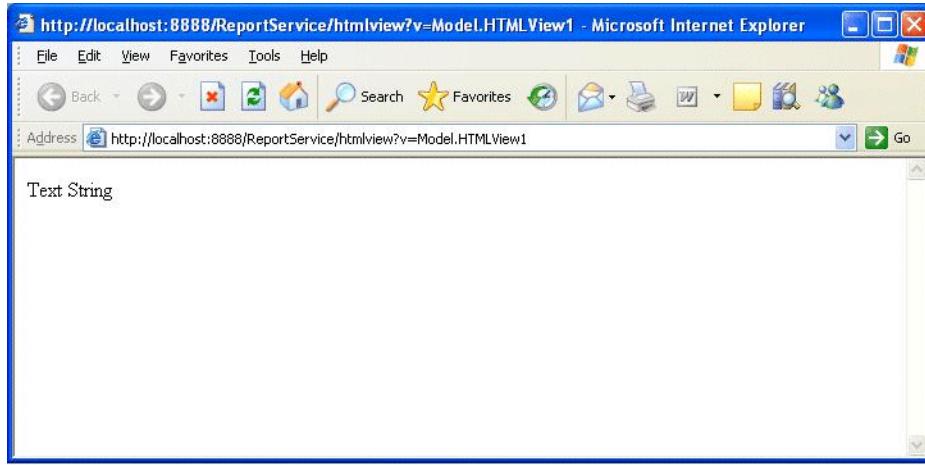
An HTML view and its associated objects define an HTML web application deployed by Cogility Modeler to the application server. The contents or the HTML view get populated upon request, at runtime. The contents of the HTML view can be either static or dynamic, and can be invoked with or without arguments.

The deployed web application is accessible through Insight. Under the View menu, select Business Activity Monitoring. On the adjacent page, two tabs will display one for Chart Views and one for HTML Views. Click on the HTML Views tab and a list of deployed HTML Views is displayed.



Clicking on the desired HTML View will open the associated web application. The web application is deployed to the application server and to a particular URL. The web application runs independent of Insight and as long as the application server is running, the web application can be launched in a web browser, given the URL.

Below is a simple, "static" HTML View which just displays a text string.



An HTML View and its associated objects define an HTML web application deployed by Modeler to the application server, whose contents can be either static or dynamic, and invoked with or without arguments.

The URL address of an HTML view has the following format:

```
http://<appServer IP address or Hostname>:<Port Number>/ReportService/  
htmlview?v=<Full XML Key of HTML view>
```

For example, the sample above has a URL of:

```
http://localhost:8888/ReportService/htmlview?v=ModelHTMLView1
```

Defining the HTML

An HTMLView has an 'HTML' field in which you can define the master template for a view and an 'Actions' field in which you can define actions that can populate special constructs within the HTML. An HTMLView can have HTMLTables that can be populated by the actions, and, can have HTMLTableCellTemplates that can be used to override the default styling for an individual table cell.

The three constructs which can be included in the HTML field are var, HTML, and table. "var" represents a variable value, "HTML" represents an HTML string, and "table" represents an HTMLTable associated to the HTMLView being modeled.

These constructs are added to the HTML of the HTMLView using the following syntax:

- `# [var varNameString]`
- `# [html varNameString]`
- `# [table tableNameString]`

These constructs can be embedded in the HTML and are replaced by values assigned to by the Actions when the HTML view is generated, as follows:

- `# [var varNameString]` - these values are escaped. For example, a varNameString value of "" is inserted as ""
- `# [html varNameString]` - these values are inserted as is. For example, a varNameString value of "" is inserted as "".

External sources can be graphics or java script files. By default, Modeler expects external sources to be placed in:

```
%DCHOME%\MB\resources\htmlviews
```

Files in sub-directories can be referenced by adding the sub-directory name to the path of the source which is being utilized in the HTML.

The default root folder can be specified by using the following configuration parameter:

```
com.cogility.deployment.htmlview.resource.dir=
```

Note: There is no validation performed in the HTML field and/or the HTML generated at runtime.

A simple example of 'HTML' follows:

```
<body>
<p>#[var MyTitle]</p>
#[html MyImgDuJourHTML]
#[table MyTable]
</body>
```

Actions

All Action Semantics which support HTMLViews and Tables are entered into the Actions field of the HTMLView object.

An HTMLView has an 'Actions' attribute in which you can enter actions that populates the #[var variables, the #[html variables, and, the #[table data. Within the 'Actions' attribute, 'self' is the bound variable representing the HTMLView itself.

In HTMLView actions, #[var variable values and #[html variable values can be assigned in either of the following ways:

```
self.setVariableValue( varNameString, valueObject );
self.varNameString.setValue( valueObject );
```

In HTMLView actions, an associated HTMLTable can be referenced in either of the following ways:

```
self.getTable( tableNameString );
self.tableNameString
```

In HTMLView actions, an associated HTMLTableCellTemplate can be referenced in either of the following ways:

```
self.getCellTemplate( tableCellTemplateNameString );
self.tableCellTemplateNameString
```

var and HTML constructs

The "var" and "html" constructs reference a variable name and values can be set using the following expressions:

```
self.setVariableValue( varNameString, valueObject );  
self.varNameString.setValue( valueObject );
```

For example, assume a variable named RawHTML. The syntax in the HTML field would be entered as follows:

```
# [html RawHTML]
```

The variable value can be set using either of these Action Semantics expressions:

```
self.setVariableValue('RawHTML', '<b>This is a bold statement</b>');  
self.RawHTML.setValue('<b>This is a bold statement</b>');
```

Note: Variable names are displayed in code assist (second expression) if they exist in the HTML field.

The "var" and "html" constructs can be used in the following fields:

- HTMLView: "HTML"
- HTMLTable: "Additional Table Tags" and "Table Tag Adornments"
- HTMLTableTemplate: "Additional Table Tags" and "Table Tag Adornments"
- HTMLTableCellTemplate: "Pre Value HTML", "Post Value HTML" and "Td Tag Adornments"

table construct

The "table" construct references HTMLTable objects that are associated to a target HTMLView. The table can be referenced using either of the following Action Semantics expressions:

```
self.getTable( tableNameString );*  
self.tableNameString
```

Note: To use the getTable method, the expression must be assigned to an HTMLTable type variable. For example:

```
custTable : com::cogility::chart::runtime::HTMLTable;  
custTable := self.getTable('CustomerTable');
```

HTMLTable

An HTMLTable and its associated objects define the structure of table and how to populate and style its contents.

There is a fair amount of flexibility when defining a table which is to be displayed in an HTMLView. Tables can be statically defined if the number of rows and columns is known and will not change. Tables can be dynamically defined if the structure of the table is not known due to the nature of changing data. Tables can also have a mixed definition, for example, a fixed set of columns, but an unknown number of rows. Modeling artifacts define the structures (columns and rows) that are known, and in those instances when the final structure is not known, action semantics can be written to define columns and rows at runtime.

An HTMLTable and its structure can also be styled using modeling artifacts that provide the means to incorporate valid HTML tags and attributes. Styling can also be dynamic. Action semantics can be written to specify which model artifacts should be used to style either a single table cell or a region of cells.

An HTMLTable can predefine some of the structure of a <table> in HTML. It can predefine columns, rows and styling templates for rectangular areas in the resulting HTMLTable.

If there are a known number of specific columns, HTMLTableColumns can be added to the HTMLTable. These HTMLTableColumns define the colNameString by which they can be accessed in the HTMLView's actions, and, they define the actual display string that can appear at the top of a column.

If there are a known number of specific rows, HTMLTableRows can be added to the HTMLTable. These HTMLTableRows define the rowNameString by which they can be access in the HTMLView's actions, and, they define the actual display string that can appear at the left of a row.

If special styling is wanted for rectangular areas within the resulting HTMLTable, an HTMLTableTemplate can be defined. An HTMLTableTemplate can define adornments that will be inserted in the <table> tag, and, can define styling for rectangular areas with HTMLTableCellAreas.

An HTMLTableCellArea defines a rectangular region (using a somewhat simple syntax) and specifies an HTMLTableCellTemplate that applies to that region. The syntax is described below.

Examples of cellAreaExprs:

0,0	-> the top left cell
\$\$	-> the bottom right cell
,	-> the entire table
0,*	-> the left column of cells (column 0 and all rows)
*,1..\$	-> all but the top row
0..-\$1,1..-\$1	-> all but the top row, bottom row and rightmost column
T,*	-> the entire column of row titles (if being shown)
*T	-> the entire row of column titles (if being shown)

An HTMLTableCellTemplate can define adornments that will be inserted in the <td> tag, and, can define raw HTML that will be inserted before and after the values that will appear within a cell.

For javascript manipulations and CSS styling, HTML tag attributes such as "id" and "name" can be assigned to a specific HTML element. For example:

```
<table id="mytable1" name="mytable1">...</table>
<table id="mytable2" name="mytable2">...</table>
```

"Table Tag Adornments" and "Additional Table Tags" fields can be used in HTMLTables as well as the HTMLTableTemplate. The HTMLTable values appear in the HTML for that table only, and the HTMLTableTemplate values appear in the HTML for each table that uses that template. The "Table Tag Adornments" value, if any, is inserted after the 'table' keyword in a "<table>" tag, and, after the "Table Tag Adornments" value in the HTMLTableTemplate, if any. The "Additional Table Tags" value, if any, is added after the closing of the "<table>" tag and before the first "<tr>" tag, and, after the "Additional Table Tags" value in the HTMLTableTemplate, if any. The resulting HTML has the following structure:

```
<table adornmentsFromTemplateIfAny adornmentsFromTableIfAny>
<additionalTagsFromTemplateIfAny>
<additionalTagsFromTableIfAny>
```

```
<tr>...</tr>  
<tr>...</tr>
```

From the HTMLView, an associated HTMLTableCellTemplate can be referenced as follows:

```
self.getCellTemplate( tableCellTemplateNameString );*  
self.tableCellTemplateNameString
```

Note: To use the getTableCellTemplate method, the expression must be assigned to an HTMLTable type variable. For example:

```
desiredTemplate :  
com::cogility::chart::runtime::HTMLTableCellTemplate;  
desiredTemplate := self. getTableCellTemplate ('Times10ptBold');
```

In order to use the overrideTableCellTemplate, the target HTMLTableCellTemplate object must be associated to the HTMLView.

```
</table>
```

From the HTMLTable, you can:

Note: The expressions can be prefixed by `self.<htmlTableName>`, or by a variable of type `com.cogility.chart.runtime.HTMLTable`. The examples show `self.<htmlTableName>`.

- Set table cell values:

```
self.<htmlTableName>.setValue(colNumInteger, rowNumInteger,  
cellValueObject);  
  
self.<htmlTableName>.setValue(colNumInteger, rowNumInteger,  
cellValueObject,overrideTableCellTemplate);  
  
self.<htmlTableName>.setValue(colNumInteger, rowNameString,  
cellValueObject);  
  
self.<htmlTableName>.setValue(colNumInteger, rowNameString,  
cellValueObject, overrideTableCellTemplate);  
  
self.<htmlTableName>.setValue(colNameString, rowNumInteger,  
cellValueObject, overrideTableCellTemplate);  
  
self.<htmlTableName>.setValue(colNameString, rowNameString,  
cellValueObject);  
  
self.<htmlTableName>.setValue(colNameString, rowNameString,  
cellValueObject, overrideTableCellTemplate);
```

Integer values refer to the position of the column or the row in the sequence as it appears in the Modeler UI. The initial position is zero (0). String values refer to the name of the TableColumn or TableRow object.

- Add table column/row titles



```

    self.<htmlTableName>.addNewTableColumn(tableName,
tableNameDisplayString <or NULL>);

    self.<htmlTableName>.addNewTableColumn(tableName,
tableNameDisplayString <or NULL>, overrideTableCellTemplate);

    self.<htmlTableName>.addNewTableRow(tableName,
tableNameDisplayString <or NULL>);

    self.<htmlTableName>.addNewTableRow(tableName,
tableNameDisplayString <or NULL>, overrideTableCellTemplate);

■ Enable/disable the showing of column/row titles in the rendered table

    self.<htmlTableName>.setShowTableColumnTitles( whetherToShowBoolean );

    self.<htmlTableName>.setShowTableRowTitles( whetherToShowBoolean );

■ Reference predefined columns by name

    self.<htmlTableName>.columnNameString

```

From the `HTMLTableColumn`, you can:

Note: The expressions can be prefixed by `self.<htmlTableName>.<htmlTableColumnName>`. or by a variable of type `com.cogility.chart.runtime.HTMLTableColumn`. The examples show `self.<htmlTableName>.<htmlTableColumnName>`.

- Set table cell values:

```

    self.<htmlTableName>.<htmlTableColumnName>.setValue( rowNumInteger,
cellValueObject );

    self.<htmlTableName>.<htmlTableColumnName>.setValue( rowNumInteger,
cellValueObject, overrideTableCellTemplate );

    self.<htmlTableName>.<htmlTableColumnName>.setValue( rowNameString,
cellValueObject );

    self.<htmlTableName>.<htmlTableColumnName>.setValue( rowNameString,
cellValueObject, overrideTableCellTemplate );

```

- Reference predefined rows by name

```
    self.<htmlTableName>.<htmlTableColumnName>.rowNameString
```

From the `HTMLTableRow`, you can:

Note: The expressions can be prefixed by `self.<htmlTableName>.<htmlTableColumnName>.<htmlTableRowName>`. or by a variable of type `com.cogility.chart.runtime.HTMLTableRow`. The examples show `self.<htmlTableName>.<htmlTableColumnName>.<htmlTableRowName>`.

- Set table cell values

```

    self.<htmlTableName>.
<htmlTableColumnName>.<htmlTableRowName>.setValue( cellValueObject );

```

```
self.<htmlTableName>.  
<htmlTableColumnName>.<htmlTableRowName>.setValue( cellValueObject,  
overrideTableCellTemplate );
```

Sorting table rows and columns

You can sort the rows of a table by the values within a single column, or vice versa.

To sort the rows of a table by the values in a single column of its data, use the following syntaxes:

```
self.<htmlTableName>.<htmlTableColumnName>.sortByValuesInColumn(   
isAscending )  
  
self.<htmlTableName>.sortByValuesInColumn( aColumnName, isAscending )  
  
self.<htmlTableName>.sortByValuesInColumn( aColumnName, isAscending )  
)
```

To sort the columns of a table by the values in a single row of its data, use the following syntaxes:

```
self.<htmlTableName>.anyColumnName.myRowName.sortByValuesInRow(   
isAscending )  
  
self.<htmlTableName>.sortByValuesInRow( aRowName, isAscending )  
  
self.<htmlTableName>.sortByValuesInRow( aRowIndex, isAscending )
```

The following rules apply when sorting rows/columns:

- If a column/row value is unassigned, it will be sorted to the end of the list, regardless of sort order.
- If a column/row string value is null, it will be sorted to the end of the non-null entries, regardless of sort order.
- If all of the values in the column/row are numbers, null, or, undefined, they will be sorted numerically. Otherwise, they will be sorted alphabetically. If you want your numbers sorted alphabetically, convert them to strings before calling the 'setValue' methods.
- If sorting by a column of values, the row titles and labels (if any) will be re-ordered, too. Similarly, if sorting by a row of values, the column titles and labels (if any) will be re-ordered, too.
- If a value is 'set' and has an overrideHTMLTableCellTemplate, that overrideHTMLTableCellTemplate will remain with the 'set' value, regardless of where it sorts in the list. Values whose HTMLTableCellTemplate is determined by the HTMLTableTemplate and an HTMLTableCellArea, may have their HTMLTableCellTemplate change as a result of being re-ordered.
- Sorting is stable (i.e., if two values are equal, they will remain in their original order). This means that you can perform multiple sorts to order table entries based on multiple columns (for example, sortByValuesInColumn('FirstName',true) followed by sortByValuesInColumn('LastName',true) to sort something such as the names in a phone book).

Note: The sort occurs at the time that the 'sortByValuesIn...' method is called with whatever values are in the table at that time. If you set more values after sorting, they will be unaffected by the sort and will remain where you set them.

Cell Area Definition

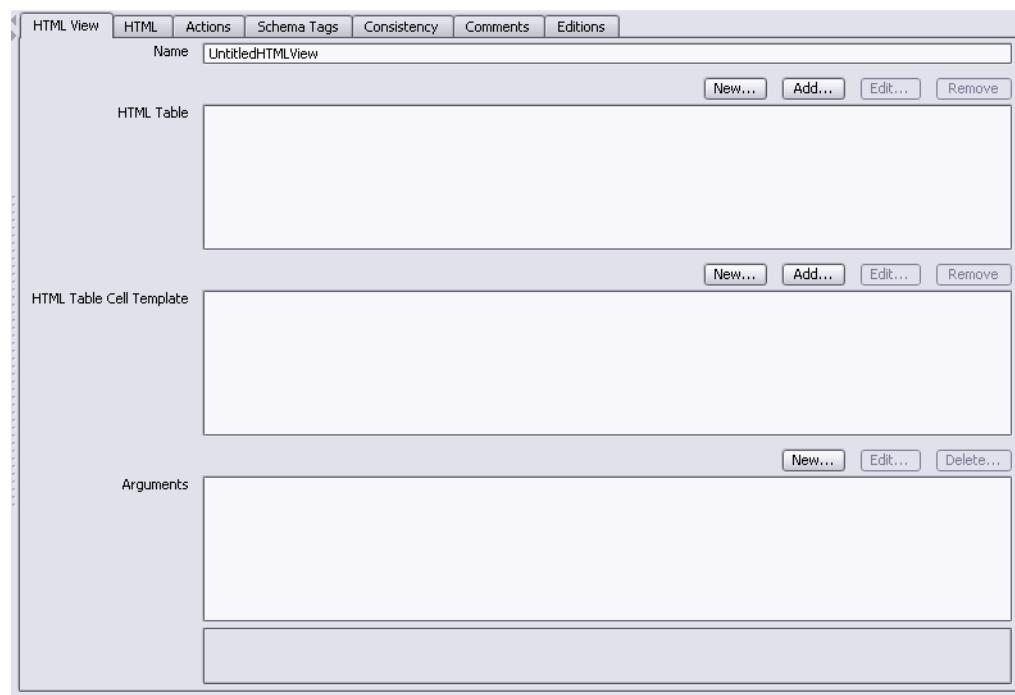
The HTMLTableCellArea can define a rectangular region by specifying a range of columns and a range of rows that bounds that region.

The syntax for defining rectangular regions is as follows:

```
cellAreaExpr -> colExpr comma rowExpr  
colExpr -> rangeExpr  
rowExpr -> rangeExpr  
rangeExpr -> singleColRow | rangeOfColsRows | titleSign | star  
singleColRow -> number | dollarSign | dollarSignExpr  
rangeOrColsRows -> singleColRow dotDot singleColRow  
dollarSignExpr -> dollarSign minusSign number  
number -> [0-9]+  
comma -> ,  
dollarSign -> $      (indicates that last column/row in the table)  
dotDot -> ..  
minusSign -> -  
titleSign -> T      (indicates the 'title' column/row in the table)  
star -> *      (indicates all columns/rows in the table)
```

Creating an HTML view

Figure 10-1: Create HTML view page



To create an HTML view:

1. In Cogility Modeler's tree view, select the model container or package.
2. From the Selection menu, select **Chart and HTML Views > Add HTML View**.
3. Enter a name for the new HTML view.
(Steps 4 through 6 are optional.)
4. Add an HTML table. See "[Creating an HTML table](#)" on page 247.
5. Add an HTML Table Cell Template. See "[Creating an HTML table cell template](#)" on page 249.
HTML Table Cell Templates are added only if they will be accessed and used by the Actions in the HTMLView.
6. Add the arguments to the HTML view:
 - a. Click New.
 - b. Enter a name.
 - c. Select the datatype.
 - d. Click OK.

Adding HTML to an HTML View

At a minimum, the HTMLView object requires some valid HTML. Modeler does not validate the HTML syntax. You must validate all HTML syntax.

To add HTML to an HTMLView:

1. Select the HTML tab.
2. Enter the HTML. See "[Defining the HTML](#)" on page 238

Adding actions to an HTML View

Action semantics are used to assign values to the special constructs, assign values to table cells, dynamically create table rows and columns, dynamically toggle table row and column titles, and dynamically override styling applied by `HTMLTableCellTemplate` objects.

To add Actions to an HTML View:

1. Select the Actions tab.
2. Enter the action semantics. See “[Creating an HTML view](#)” on page 246.

Creating an HTML table

An HTML Table Column defines a column in a table. Table columns can be defined statically in the HTML Table or dynamically using Action Semantics in the HTML View.

An HTML Table Row defines a row in a table. Table rows can be defined statically in the HTML Table or dynamically using Action Semantics in the HTML View.

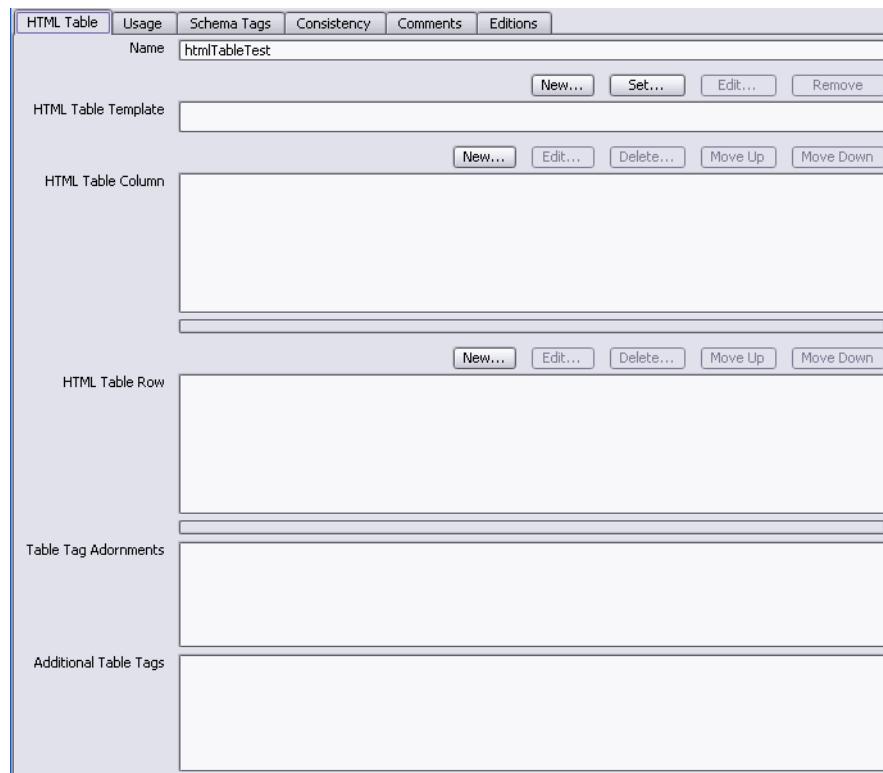


Figure 10-2: Create HTML Table Page

To create an HTML table:

1. In Cogility Modeler’s tree view, select the model container or package.
2. From the Selection menu, select **Chart and HTML Views > Add HTML Table**.
3. Enter a name for the new HTML table.
4. (Optional) Add an HTML Table Template.

To add a new HTML Table Template:

- a. Click **New** above the **HTML Table Template** field.

- b.** Enter a name and click OK.
- c.** Click Edit.
- d.** Enter information for the HTML Table Template. See “[Creating an HTML table template](#)” on page 248.

To select an existing HTML Table Template:

- a.** Click Set.
- b.** Select an HTML Table Template from the object list and click OK.
- 5.** (Optional) Add an HTML Table column.
 - a.** Click New above the **HTML Table Column** field.
 - b.** Enter a name and click OK.
- 6.** (Optional) Add an HTML Table row.
 - a.** Click New above the **HTML Table Row** field.
 - b.** Enter a name and click OK.
- 7.** (Optional) Enter Table Tag Adornments.
- 8.** (Optional) Enter Additional Table Tags.

Creating an HTML table template

An HTML Table Template defines the default styling to be applied to some or all cells in an HTML Table.

Table Tag Adornments are attributes that are valid for the <table> tag. Additional Table Tags are valid element tags that are defined within the open/close Table tags.

Note: Table Tag Adornments and Additional Table Tags can be defined for a table on the HTML Table, the HTML Table Template, or both. If values are defined on both, then both values will be written to the HTML source, with the values defined on the HTML Table appearing in the source as the first value, and thus used in the display.

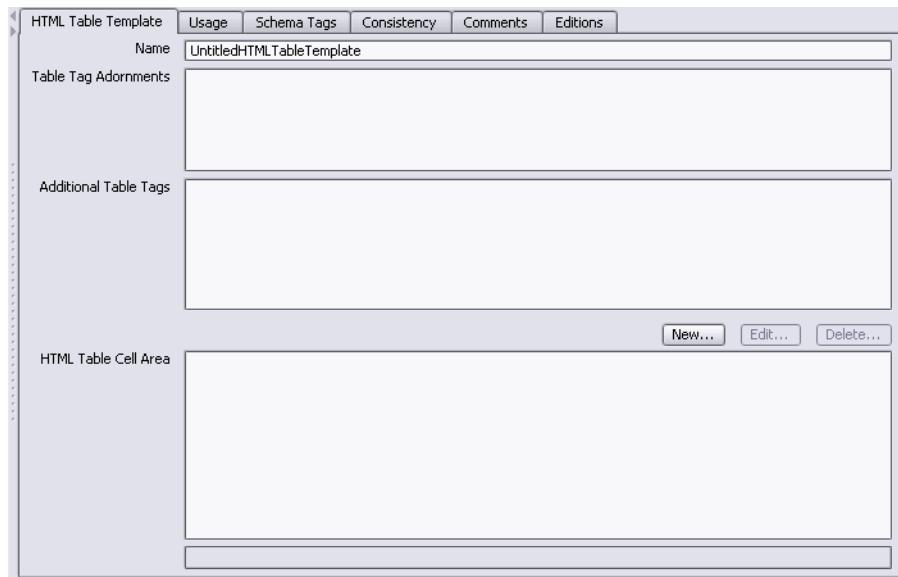


Figure 10-3: Create HTML Table Template PageTo create an HTML table template:

1. In Cogility Modeler’s tree view, select the model container or package.

2. From the Selection menu, select **Chart and HTML Views > Add HTML Table Template**.

3. Enter a name for the new HTML table template.

4. (Optional) Enter Table Tag Adornments.

5. (Optional) Enter Additional Table Tags.

6. (Optional) Define the HTML Table Cell Area.

a. Click New above the HTML Table Cell Area field.

b. Enter a name and click OK.

c. Enter the cell area expression. See “[Cell Area Definition](#)” on page 245.

d. Specify an HTML Table Cell Template:

To create a new HTML Table Cell Template:

- Click New.
- Enter a name and click OK.
- Click Edit.
- Enter information for the HTML Table Cell Template. See “[Creating an HTML table cell template](#)” on page 249.

To select an existing HTML Table Cell Template:

- Click Set.
- Select an HTML Table Cell Template from the object list and click OK.

Creating an HTML table cell template

The HTML Table Cell Template defines, in terms of HTML, the styling of an area of cells in a table.

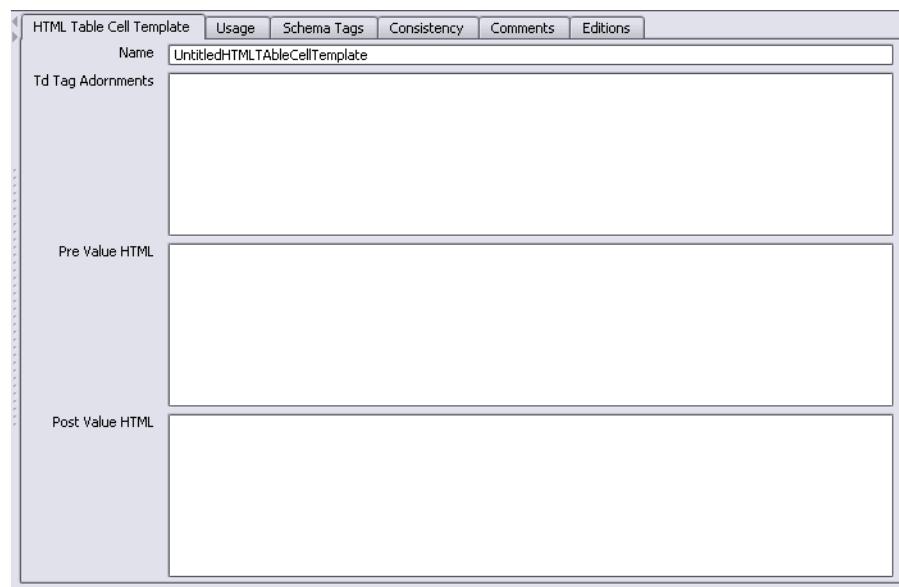


Figure 10-4: Create HTML Table Cell Template Page

To create an HTML Table Cell Template:

1. In Cogility Modeler’s tree view, select the model container or package.

2. From the Selection menu, select **Chart and HTML Views > Add HTML Table Cell Template**.

3. Enter a name for the new HTML table cell template.

4. (Optional) Enter TD Tag Adornments.

5. (Optional) Enter Pre-value HTML.
6. (Optional) Enter Post-value HTML.

Chart View

A ChartView has a 'Chart Type' enumeration that allows you to specify the chart type. The chart types are:

- Pie 2D
- Pie 3D
- Line
- Point
- Bar
- Clustered Bar
- Stacked Bar
- Area
- Stacked Area

For the BarChart, the 'Orientation' enumeration allows the bar to be displayed horizontally or vertically. For all Chart Types, the Chart View has a 'Chart Title' field that is displayed above the chart. For BarCharts, the ChartView has an 'X Axis Title' and 'Y Axis Title' that are displayed along the X and Y axes.

A ChartView can also have a ChartProperties. A ChartProperties contains a sequence of comma separated RGB color values (values are from 0 to 255), one color per line. These colors will be used, in the order entered, to represent the Category 1 slices/bars.

A ChartView has a "Drill Down" capability which allows you to navigate from one Chart to another Chart.

A ChartView can have arguments. The arguments have names and data types. Before the chart is displayed, you are prompted to provide values for each argument.

Creating a chart view

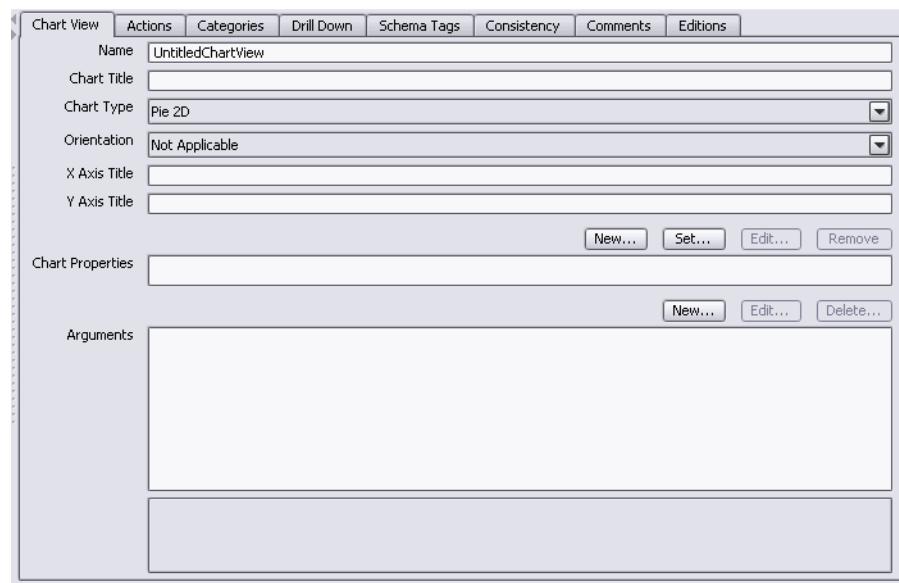


Figure 10-5: Chart View PageTo create a new chart view:

1. In Cogility Modeler's tree view, select the model container or package.
2. From the Selection menu, select **Chart and HTML Views > Add a Chart View**.
3. Enter a name for the new Chart. This name is the chart name within the Cogility model.
4. Enter a title for the Chart if desired. This is the title for that chart that appears with the chart.
5. Select the chart type from the Chart Type drop-down list.
6. Complete the Orientation and X/Y axis names as appropriate for the chart type as shown in the table below:

Chart Type	Orientation	X/Y axis name
Pie 2D	Not applicable	Not applicable
Pie 3D	Not applicable	Not applicable
Line	Not applicable	Valid
Point	Not applicable	Valid
Bar	Valid	Valid
Clustered Bar	Valid	Valid
Stacked Bar	Not applicable	Valid
Area	Not applicable	Valid

7. (Optional) Enter the chart properties:
 - a. To define new chart properties:
 - Click New.
 - Enter a name for the chart properties and click OK.
 - Click Edit.

- Add the RGB color definitions. The color definitions are a sequence of comma separated RGB color definition values (from 0 to 255), one color per line.
 - b. To use an existing chart property definition:
 - Click Set.
 - Select the chart property from the object list and click OK.
8. (Optional) Add the arguments to the chart view:
- a. Click New.
 - b. Enter a name.
 - c. Select the datatype.
 - d. Click OK.

Adding Actions to a Chart View

After creating the Chart View, you must define the actions that will populate the chart. Actions for the chart views are defined using Cogility Action Semantics. These actions are used to populate the bars/slices of a Chart. Within the actions, 'self' represents the ChartView itself.

Following are the 'dynamic' methods that are available for 'self'. 'Dynamic' methods are available in 'hybrid' and 'static' charts, also.

- Valid for PieCharts and BarCharts:
 - `self.addNewChartCategory1(categoryName, displayString);`
- Valid for BarCharts only:
 - `self.addNewChartCategory2(categoryName, displayString);`
 - `self.setValue(category1Num , category2Num , numericCategoryValue);`
 - `self.setValue(category1Num , category2Name, numericCategoryValue);`
 - `self.setValue(category1Name, category2Num , numericCategoryValue);`
 - `self.setValue(category1Name, category2Name, numericCategoryValue);`
- Valid for PieCharts only:
 - `self.setValue(category1Num , numericCategoryValue);`
 - `self.setValue(category1Name, numericCategoryValue);`

For 'static' and 'hybrid' ChartViews, a dot notation can be used to reference the categories:

- Valid for BarCharts only:
 - `self.<category1Name>.<category2Name>.setValue(numericCategoryValue);`
 - `self.<category1Name>.setValue(category2Num , numericCategoryValue);`
 - `self.<category1Name>.setValue(category2Name, numericCategoryValue);`
- Valid for PieCharts only:
 - `self.<category1Name>.setValue(numericCategoryValue);`

To add actions to a chart view:

1. Select the Actions tab.
2. Enter the action semantics for the desired actions.

Setting chart titles programmatically

Using Action Semantics, you can set the chart and axes titles of a ChartView programmatically in the **Actions** field of the ChartView definition. You can set the string value of the title you want, or, you can set the value to null to remove a title. Any values that are defined in the ChartView "Chart Title",

"X Axis Title" and/or "Y Axis Title" fields are overwritten at runtime using the Action Semantics methods.

The methods for setting the ChartView titles are:

- `self.setChartTitle(newChartTitle);`
- `self.setXAxisTitle(newXAxisTitle);`
- `self.setYAxisTitle(newYAxisTitle);`

For more information on using Action Semantics, see *Using Action Semantics with Cogility Studio*.

Adding categories to a chart

A ChartView also has Categories. Pie Charts have Category 1s only, which represent slices of the pie. All other Charts have Category 1s and Category 2s where Category 1s represents a single bar within a repeating group of bars, and Category 2s represents the repeating groups.

To add categories to a chart view:

1. Select the Categories tab.
2. (Optional) Click New above the Chart: Category 1 text box to add a Category 1.
3. Add a name for the Category 1 and click OK.
4. (Optional, if applicable) Click New above the Chart: Category 2 text box to add a Category 2.
5. Add a name for the Category 2 and click OK.

Drilling down in a chart view

Defining a drill down for a Chart View allows you to define the navigation from one chart to another. You define the Drilled Down From chart (indicating a chart that precedes the current chart in a chart navigation) and a Drilled Down To chart (indicating a chart that follows the current chart in a chart navigation). Defining a Drill Down is optional.

To define a drill down:

1. Select the Drill Down tab.
2. To define a Drilled Down To chart:
 - a. Click Add (or New) above the Drills Down To text box.
 - b. Select a chart view from the object list and click OK.
3. To define a Drilled Down From chart:
 - a. Click Add (or New) above the Drills Down From text box.
 - b. Select a chart view from the object list and click OK.



Deployment Model

Once you have created an integration model in Cogility Modeler, you deploy it as an application on a J2EE application server for use with a specific run-time repository. You specify the deployment configuration as described in “[Deployment Model](#)” on page 255.

As shown in the figure below, deployment is the process of taking a model you have created in Cogility Modeler and pushing it into execution as a enterprise application. See “[Pushing the model into execution](#)” on page 273, for instructions on pushing an entire model into execution, assuming an entirely new deployment. However, different deployment options are available to accommodate different types of revision to a running enterprise application. See “[Push modes](#)” on page 27 of the guide, *Model Deployment & Execution in Cogility Studio*.

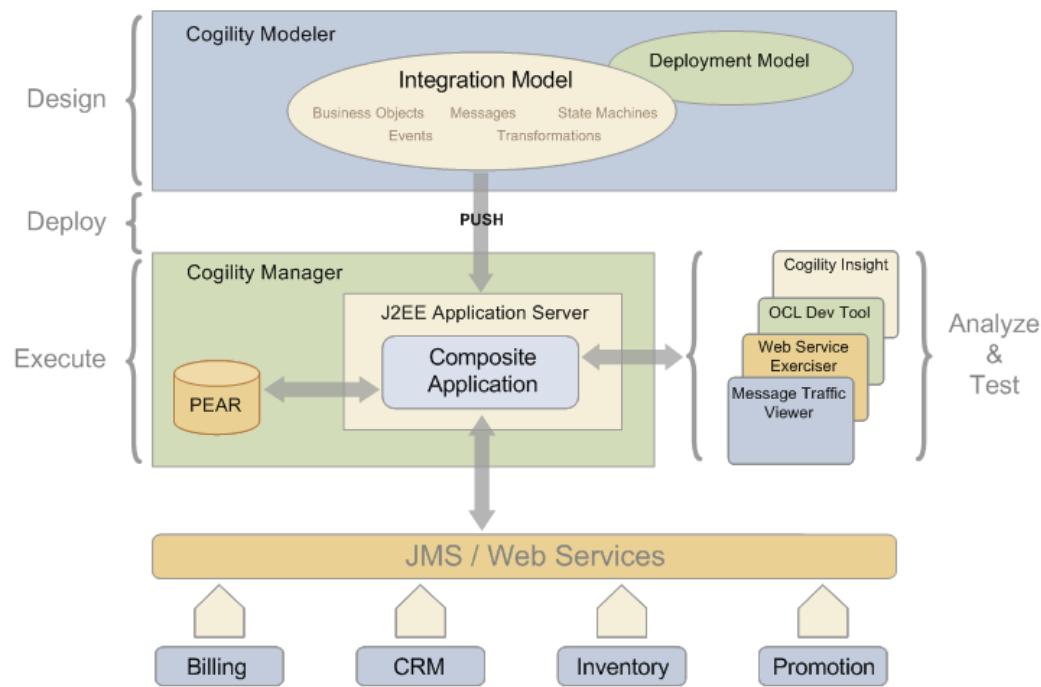


Figure 11-1: Deployment illustrated

During deployment, Cogility Modeler creates the Java beans, deployment descriptors, routines and other Java run-time utilities. You do not have to write any Java code to complete the deployment because deploying a model not only creates the Java application, but the run-time environment that manages interactions between the composite application, the specific application server and the database for the run-time repository (PEAR).

Pushing the model creates the tables on the run-time repository (see “Run-time respiratory” on page 267) for the model objects (such as the M2E conversions, events, web services, and so on), and for the business objects that are created at run-time (such as Customers, Addresses and Products); it also creates the Java beans, deployment descriptors, routines and libraries that manage the communication between the composite application, the J2EE application server and the run-time repository.

During push you can force the model’s action semantics to be recompiled, see “Force action semantics recompilation” on page 32 of the guide, *Model Deployment & Execution in Cogility Studio*. You can also force the push of artifacts to the application server. See “Application server only push” on page 29 of the guide, *Model Deployment & Execution in Cogility Studio*. Both of these options are applicable when migrating models to newer versions of Cogility Studio or deploying the model on several application servers.

Cogility Manager is the run-time environment in which the deployed composite application runs. It enables a model created in Cogility Modeler to run as a composite application on a specific application server with a specific database for the run-time repository. See “Cogility Manager” on page 11 of the guide, *Model Deployment & Execution in Cogility Studio*.

Cogility Manager includes the libraries and routines that run the composite application on a specific J2EE application server which provides system security and enables system messaging and database transactions. See the guide, *Installing and Configuring Cogility Studio* for more information about supported application servers.

Deployment creates a message-driven bean (MDB) for each M2E or E2M conversion, based on each conversion’s destination. This includes the XML deployment descriptors and the Java files that specify the MDB execution behavior.

Default Deployment Model

You can deploy the system model to any number of different application server and database configurations specified in the deployment model. The default deployment model consists of artifacts for each type of application server and database supported by Cogility Studio. An application server artifact and a database artifact are paired in a deployment. The deployment model includes a deployment for each supported application server and database combination. You edit these for your particular application and database installations. You can also create your own deployments and application server artifacts and database artifacts. The deployment model may have any number of deployments. When you push a system model into execution (see “Pushing the model into execution” on page 273), you select the deployment for the target machine’s configuration from the deployment model.

When you log in to Cogility Modeler for the first time or with the authoring repository empty, the default deployment model is loaded automatically into the current context. See “Logging into the current context” on page 17. You cannot delete the default deployment model from the current context. You may rename the default deployment model and you can create a new deployment model. The deployment model, its deployments, application server artifacts and database artifacts can be versioned; the deployment model can be exported and imported in the same manner as a system model.

Each deployment model contains the following:

- Application Server artifacts - the application server artifact defines the application server to which an application will be deployed
- Database artifacts - the database artifact defines a connection to a run-time repository.

- Deployment artifacts - the deployment artifact defines the association between an application server artifact and a database artifact.

Editing the default deployment model

To edit the default deployment model:

1. In Cogility Modeler's tree view, select the **DefaultDeploymentModel**.
2. In the **Name** field, enter a new name, if desired.
3. Click the plus sign next to the **DefaultDeploymentModel** container.
4. Define the deployments.

["Creating a deployment model" on page 257.](#)

Deployments

A deployment artifact associates an application server artifact with a database artifact. Several deployments may use the same application server artifact or database artifact. Editing an artifact in a particular deployment will change that artifact where it is used in other deployments. You should create an artifact for each discrete installation of an application server or a database. See ["Application server artifact" on page 260](#) and ["Database artifact" on page 257](#). You can create new application server artifacts and database artifacts within the deployment, and these will be added to the list of artifacts available to other deployments.

Creating a deployment model

To create a deployment model:

1. In Cogility Modeler's tree view, select the system model container.
Usually, this is the topmost model object in the tree view.
2. Hold down the Ctrl key and, with the mouse pointer, click on the system model object again.
This effectively "de-selects" the model object, leaving nothing selected in the tree view.
3. Click the plus button  and select **Add Deployment Model**.
An untitled deployment model appears in the tree view and the editor view.
4. Enter a name for the deployment in the Name field.
5. Click the plus button  and select **Add Deployment**.
An untitled deployment appears in the tree view and the editor view.
6. In the **Name** field, enter a name.
7. Create or set the application server artifact.
See ["Application server artifact" on page 260](#).
8. Create or set the database server artifact.
See ["Database artifact" on page 257](#).

Database artifact

A database artifact defines a connection to the run-time repository. The run-time repository is a database that holds the deployed system model's schema and the run-time objects that are created

during the model application's execution. The connection to the run-time repository consists of a username and password and the connection string that specifies the database's location.

Cogility Studio supports the IBM DB2, MySQL and Oracle 10g and Oracle 11g databases. The default deployment model includes one database artifact for the Oracle 10g database that is configured for a default installation. If your deployment requires a different database, you can change this artifact or create a new artifact for the database type.

You should create a database artifact for each discrete database installation. Several deployments may use the same database. But there may be different installations of the same database type, and each must have a distinct database artifact. For example you may have Oracle 10g installed both locally and on a remote machine. When you deploy the model application to the local machine, you use the database artifact for that installation. The artifact's connect string reads, `jdbc:oracle:thin:@localhost:1521:orcl`. If you are deploying the model application remotely, you must use the deployment including the database for that remote machine. Its connect string might read, `jdbc:oracle:thin:@remotehost:1521:orcl`. Because each installation has a different connect string, specifically a different host name, each installation must have its own database artifact.

Creating the database artifact

To create a database artifact:

1. In the deployment model, select the deployment.
2. Above the **Database** field, click **New**.
3. In the dialog, enter a name for the database artifact and click **OK**.
4. Define the database artifact.

See “[Defining the database artifact](#)” on page 258.

Setting the database artifact

To set the database artifact for a deployment:

1. In the deployment model, select the deployment.
2. Above the **Database** field, click **Set**.
3. In the dialog, select the database artifact and click **OK**.
4. Define the database artifact, if necessary.

See “[Defining the database artifact](#)” on page 258.

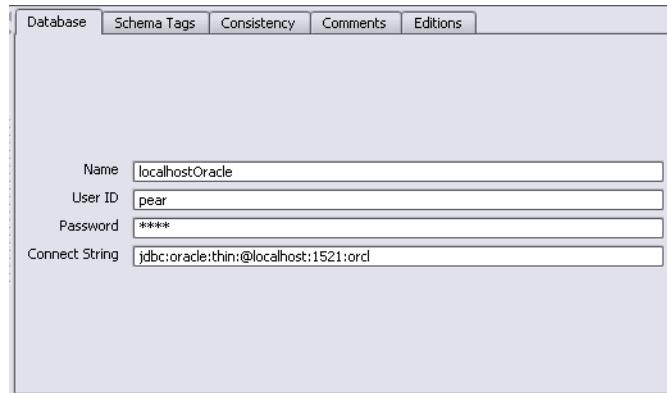
Defining the database artifact

You can define a database artifact either through a deployment or by editing the artifact in the tree view. Changing the artifact in either location will cause the changes to be propagated to all other deployments where the artifact is used.

Note that if you change the default database artifact for a database other than Oracle 9i, the icon for the artifact will change to match the database type as specified in the connect string.

To define a database artifact:

1. In the tree view, select the database artifact or the deployment containing the database artifact.



2. In the **Database** tab, in the **Name** field, enter a name for the artifact.
3. In the **User ID** field, enter a user ID.
4. In the **Password** field, enter a password.

You created the User ID and Password values during installation. See the guide, *Installing and Configuring Cogility Studio*. The default value for both the user ID and password is **pear**.

5. In the **Connect String** field, enter the database connect string.

This is the connection string to the database used for the run-time repository. The default connection string to the Oracle database is provided with the default server name (localhost) and the default database name or SID (orcl). See your database server documentation for more information.

The following connect strings are valid for the specified database type, database driver, and supporting jar file;

Database	Database type	Database driver	Supporting jar file
Oracle 9i	jdbc:oracle:thin:@targethost:1521:orcl	oracle.jdbc.driver.OracleDriver	OracleJDBC_9i_jdk14.jar
Oracle 10g	jdbc:oracle:thin:@targethost:1521:orcl	oracle.jdbc.driver.OracleDriver	OracleJDBC_10g_jdk14.jar
DB2	jdbc:db2://targethost:50000/mydb	com.ibm.db2.jcc.DB2Driver	db2jcc_81.jar
SQL Server	jdbc:jtds:sqlserver://targethost:1433	net.sourceforge.jtds.jdbc.Driver	jtds102_sqlserver.jar
MySQL	jdbc:mysql://targethost:16:3306/pear	org.gjt.mm.mysql.Driver	MySqlJDBC.jar

Testing the database connection

Before you can push the model, the model must have a valid database connection string.

To test the database connection:

1. In Cogility Modeler's tree view, select the model container and right-click to display the menus.
2. Under **Actions**, select **Check Database Connection**.

Application server artifact

An application server artifact describes the J2EE application server to which the model application is deployed. Cogility Studio supports the Oracle WebLogic and Oracle application servers and provides a default artifact for each type in the default deployment model. You can either edit these default artifacts or create new ones.

You should create an application server artifact for each discrete application server installation. Several deployments may use the same application server. But there may be different installations of the same application server type, and each must have a distinct application server artifact. For example you may have WebLogic installed both locally and on a remote machine. When you deploy the system model application to the local machine, you use the application server artifact for that installation. The artifact's Target Host field specifies the name of the machine where the application server resides, in this case, the local host. When you deploy to a remote machine, you must use a different application server artifact, one with a Target Host field that specifies the machine name for that remote machine. See [Chapter 8, "Remote Deployment"](#). of the guide, *Model Deployment & Execution in Cogility Studio*.

Creating an application server artifact

To create an application server artifact:

1. In the deployment model, select the deployment.
2. Above the **Application Server Type** field, click **New**.
3. In the dialog, select the application server type and click **OK**.
4. In the dialog, enter a name for the artifact and click **OK**.
5. Define the application server artifact.

See “[Defining a WebLogic application server artifact](#)” on page 260.

See “[Defining an Oracle 10g application server](#)” on page 262.

See “[Multiple Application Server Deployment](#)” on page 265.

Setting the application server artifact

To set the application server artifact for a deployment:

1. In the deployment model, select the deployment.
2. Above the **Application Server Type** field, click **Set**.
3. In the dialog, select the application server artifact and click **OK**.
4. Define the application server artifact, if necessary.

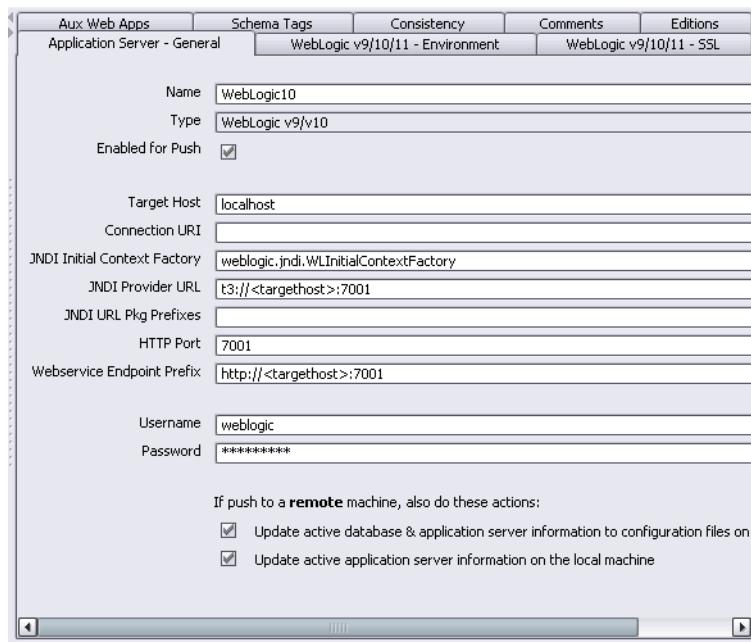
See “[Defining a WebLogic application server artifact](#)” on page 260.

See “[Defining an Oracle 10g application server](#)” on page 262.

Defining a WebLogic application server artifact

Note: You are required to update the `SET_WL_ENV` file, located in `%DCHOME%\MB\scripts\subroutines\SET_WL_ENV.bat`; and on Linux in `$DCHOME/MB/bin/subroutines/set_WL_env.sh`, with the location of the WebLogic server installation (`WL_HOME`).

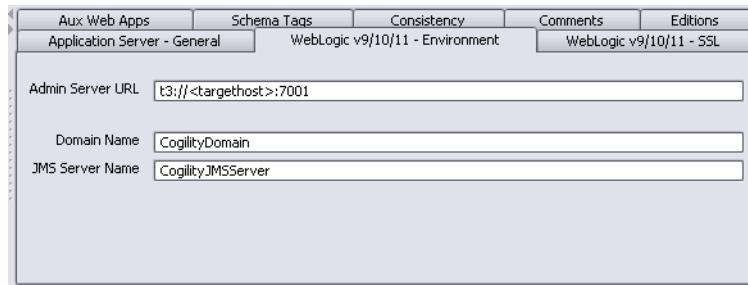
To define a WebLogic application server artifact:



1. In the tree view, select the application server artifact or the deployment containing the application server artifact.
2. In the **Application Server - General** tab, in the **Name** field, enter a name for the artifact.
3. In the **Target Host** field, enter the name of the target host.
The default is localhost, an alias for the local system, regardless of name. You usually do not need to change this setting. You may also enter an IP address in this field.
4. In the Connection URI field, no value is required. This field is only applicable for OAS.
5. In the **JNDI InitialContext Factory** field, enter the JNDI initial context factory name.
The default is provided in the default deployment model.
6. In the **JNDI Provider URL** field, enter the JNDI provider URL.
The default is provided in the default deployment model.
7. In the **JNDI URL Package Prefixes** field, enter the JNDI URL package prefixes. This field is left blank by default.
8. In the **HTTP Port** field, enter the HTTP port used by WebLogic.
The default is 7001 for an unsecured connection and 7002 for a secure connection (using SSL), if you have installed WebLogic according to the instructions in the guide, *Installing and Configuring Cogility Studio*.
9. In the **Webservice Endpoint Prefix** field, enter the prefix for all Cogility web service URLs.
The default is provided in the default deployment model.
10. In the **Username** field, enter the username to connect to WebLogic. This value is defined during the configuration of WebLogic for Cogility. The default value is set to "weblogic".
11. In the **Password** field, enter the password to connect to WebLogic. This value is defined during the configuration of WebLogic for Cogility. The default value is set to "cogility". If no value is specified, the object will be inconsistent.
12. Check the two checkboxes as follows:
 - For Remote Push, there are two checkboxes to consider.

- The first box should be checked if the target application server is remote. This will copy the appropriate configuration files to DCHOME/MB/Config/Files on that target machine. If the first box is checked, the remote database and application server configuration files will be overwritten by the ones from the push machine. As such, it is imperative that Target Hostname must also be DNS-resolvable by the target server.
- The second box should be checked only if you intend to run Cogility Operational Support tools such as Insight, the Web Service Exerciser and such, against the remote application server. Otherwise, the second can remain unchecked.

13. In the **WebLogic v9/10/11 - Environment** tab, in the **Admin Server URL** field, enter the URL to connect to WebLogic administration server. The default is provided in the default deployment model.



14. In the **Domain Name** field, enter the name of WebLogic domain where the Cogility model is to be deployed. The default is CogilityDomain.
15. In the **JMS Server Name** field, enter the name of JMS server to be used by Cogility. The default is CogilityJMSServer.
16. In the **WebLogic v9/10/11 - SSL** tab, in the SSL Trust Key Store field, select from one of the following:
- a. None - if you are not using Secure Socket Layer.
 - b. JavaStandardTrust - if you are using JDK standard trust certificate authorities.
 - c. CustomTrust - if you want to specify your own trust key store.
17. If you are using a custom key store, in the SSL Trust Key Store Filename field, enter the path to the trust key store file.
18. If you are using a custom key store, in the SSL Trust Key Store Pass Phrase field, enter the password to access the custom trust key store.
19. If you are using a custom key store, in the SSL Trust Key Store Type field, enter the key store type.

Defining an Oracle 10g application server

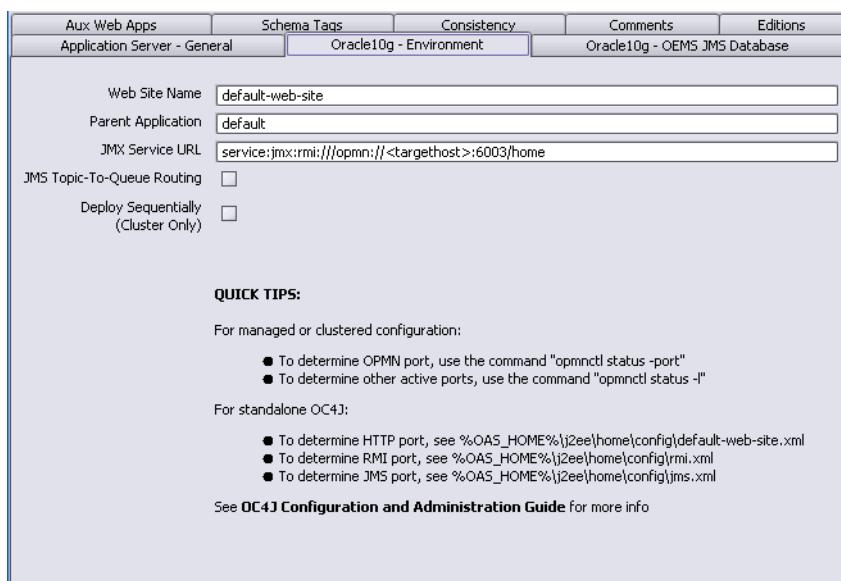
Note: You are required to update the SET_OAS_ENV file, locate in Windows - DCHOME\MB\Scripts\subroutines\SET_OAS_ENV.bat or Linux - DCHOME/MB/bin/subroutines/SET_OAS_ENV.bat , with the location of the Oracle Application Server installation (**OAS_HOME**).

To define an Oracle 10g application server artifact:

Aux Web Apps	Schema Tags	Consistency	Comments	Editions
Application Server - General	Oracle10g - Environment	Oracle10g - OEMS JMS Database		
<p>Name: Oracle10g_clustered Type: OracleAS 10g <input checked="" type="checkbox"/> Enabled for Push</p> <p>Target Host: localhost Connection URI: deployer:cluster:opmn://<targethost>:6003/default_group JNDI Initial Context Factory: oracle.j2ee.rmi.RMIInitialContextFactory JNDI Provider URL: ormi://<targethost>:12401 JNDI URL Pkg Prefixes: HTTP Port: 7777 Webservice Endpoint Prefix: http://<targethost>:7777</p> <p>Username: oc4jadmin Password: <input type="password"/></p> <p>If push to a remote machine, also do these actions: <input checked="" type="checkbox"/> Update active database & application server information to con <input checked="" type="checkbox"/> Update active application server information on the local machin</p>				

1. In the tree view, select the application server artifact or the deployment containing the application server artifact.
2. In the **Application Server - General** tab, in the **Name** field, enter a name for the artifact.
3. In the **Target Host** field, enter the host name or IP address of the target WebSphere Server.
4. In the **Connection URI** field, enter the connection URI for the type of OAS:
 - Standalone - the default value is set to deployer:oc4j:<targethost>:23791
 - Managed - the default value is set to deployer:oc4j:opmn://<targethost>:6003/home
 - Clustered - the default value is set to deployer:cluster:opmn://<targethost>:6003/default_group
5. In the **JNDI Initial Context Factory** field, enter the JNDI initial context factory name. The default is provided in the default deployment model.
6. In the **JNDI Provider URL** field, enter the JNDI provider URL. The default is provided in the default deployment model.
7. In the **JNDI URL Package Prefixes** field, enter the JNDI URL package prefixes. This field is left blank by default.
8. In the **HTTP Port** field, enter the HTTP port used by OAS
 - Standalone - the default port number is set to 8888
 - Managed and Clustered - the default port number is set to 7777
9. In the **Webservice Endpoint Prefix** field, enter the prefix for all Cogility web service URLs. The default is provided in the default deployment model.
 - Standalone - the default port number is set to 8888
 - Managed and Clustered - the default port number is set to 7777
10. In the **Username** field, enter the username to connect to OAS. This value is defined during the installation of OAS. By default this value is set to oc4jadmin.

11. In the **Password** field, enter the password to connect to OAS. This value is defined during the installation of OAS. By default this value is set to oc4jadmin.
12. Check the two checkboxes as follows:
 - For Remote Push, there are two checkboxes.
 - The first box should be checked if the target application server is remote. This will copy the appropriate configuration files to DCHOME/MB/Config/Files on that target machine.
 - The second box should be checked only if you intend to run Cogility Operational Support tools e.g. Insight, the Web Service Exerciser.. against the remote application server. Otherwise, the second can remain unchecked.
13. In the **Oracle 10g - Environment** tab, in the **Web Site Name** field, enter the name of the web site the Cogility model is to be deployed. The default value is set to default-web-site.



14. In the **Parent Application** field, enter the name of the parent application to which the Cogility model is to be deployed. The default value is set to default.
15. In the **JMX Service URL** field, enter the URL to connect to the JMX MBean Server. The URL should take the form of one of the following:
 - Standalone - service:jmx:rmi://<targethost>:23791
 - Managed and Clustered - service:jmx:rmi://<targethost>:6003/home
16. The **JMS Topic-To-Queue Routing** checkbox enables high JMS concurrency, but can be resource intensive, especially for OEMS JMS Database (unsupported in r5.0). This is not recommended for Cogility models that have many Destination objects. The default value is unchecked.
17. The **Deploy Sequentially** checkbox specifies that each OAS instance in the cluster is to be deployed in sequential order. The default value is unchecked.

Auxiliary Web Applications

Non-Cogility web applications that need to be deployed with Cogility.ear because of their dependency on Cogility java classes can be specified in a deployment model. The web application must be in the form of a .war file.

To specify an auxiliary web application:

1. Select the deployment model to which you need to add the web application in the tree view.
2. Select the **Aux Web Apps** tab.
3. Click **New**.
4. Add a name in the name dialog box, and click **OK**.
5. Select the new web application, and click **Edit**.
6. Enter the required information in the edit dialog:
 - Web App Context Root**
 - Local War File Path** (you can select **Browse** to locate the file)

Note: The file path for an Aux Web App can accept embedded standard Cogility macros such as %DCHOME%, %OEMHOME%, and such. This allows you to specify, for example "%OEMHOME%\mywebapp.war" as a file path for an auxiliary web application. If another modeler needs to use this deployment, and as long as the modeler has the same web application relative to that modeler's %OEMHOME%, no consistency error occurs.

7. If you wish to deploy this web app to additional application servers, you can add additional application server targets:
 - a. Click **Add** above the Target Appserver field.
 - b. Select the target application server.
 - c. Click **OK**.

Multiple Application Server Deployment

Cogility Modeler supports deployment to multiple application server instances in a single push operation. Each application server instance must deploy to the same database.

To create a multiple application server deployment, follow the instructions for creating a regular deployment model (["Creating a deployment model" on page 257](#)). After defining the first application server instance, select **New** and add the additional application server instances.

The following rules apply when specifying and using a multiple application server deployment model:

- The list of application server instances is ordered. The deployments will process through the applications servers in exactly the order in which they are specified in the deployment model. You have the option to skip a particular application server during a push. You can use this option to selectively push a model to only those application servers which failed in a prior push. To do this, edit the deployment model before pushing the model.
To disable an application server, select the application server in the deployment model and click **Enable/Disable**. Disabled application servers are greyed out in the list.
To enable the application server again, select it and click **Enable/Disable**.
- The information for the last application server that is pushed successfully has its information stored as the "active" server in deploy_active_appserver.txt. The application server in this file is the one seen as the "active" server in all Cogility tools which connect to the application server. If you need to switch "active" servers, you can do this in the Project Configuration Editor tool. See ["Running Cogility Project Configuration Editor" on page 51](#) in *Model Deployment & Execution in Cogility Studio*.
- The database transaction for a push is committed as long as the model is pushed successfully to at least one application server in a multiple application server deployment.

- If the push is not successful to any of the application servers, you have the option to still commit the database changes. To do this, select the **Always commit transaction on successful database push regardless of application server push status** checkbox when defining the deployment.
- You have the option to abort the push to successive application servers if the push to the current application server fails. To do this, select the **If an application server instance push fails, do not continue with the remaining instances** checkbox when defining the deployment.

Topic to Queue Routing

Unlike other application servers, Oracle application server does not support multiple threads of the same MDB listening to a JMS topic. It does, however, for JMS queues. This may create a bottleneck issue when too many messages are published against a JMS topic as these messages cannot be processed concurrently by the same MDB. One way to resolve this problem is to use Oracle JMS Router where messages arriving at a topic will be routed to its corresponding queue that has multiple threads listening. In practice, this so-called router has many issues under heavy JMS loads - from messages either not getting delivered or simply disappearing (seen with file-persistence JMS) to the requirement of configuring the router to have visibility to Cogility application's jar files (seen with database-persistence JMS).

Cogility provides its own JMS routing service, independent of Oracle JMS Router. For each JMS topic, Cogility creates a corresponding queue. The MDB listening to the topic just simply passes on received messages to the multi-threaded MDB listening to the queue. You have the option to use this service with JMS destinations defined in your own models.

Do not indiscriminately select the JMS routing service unless the JMS topics defined in a model require high concurrency in processing. This is because it may significantly consume more resources, particularly with database-persistence JMS. The number of JMS destinations will be automatically doubled, demanding more Oracle database sessions and processes.

Switching Between Target Deployment Environments

Cogility deploys a modeled application to a target application server and a target database. Configuration information pertaining to both the application server and the database is defined in the deployment model. When a model is pushed, configuration information pertaining to both the application server and the database is written to the Cogility Configuration Facility. This provides sufficient information to the Cogility tools to access the application server and the database.

An enterprise may require multiple environments to support both development and production operations. This can require pushing the same model into multiple environments.

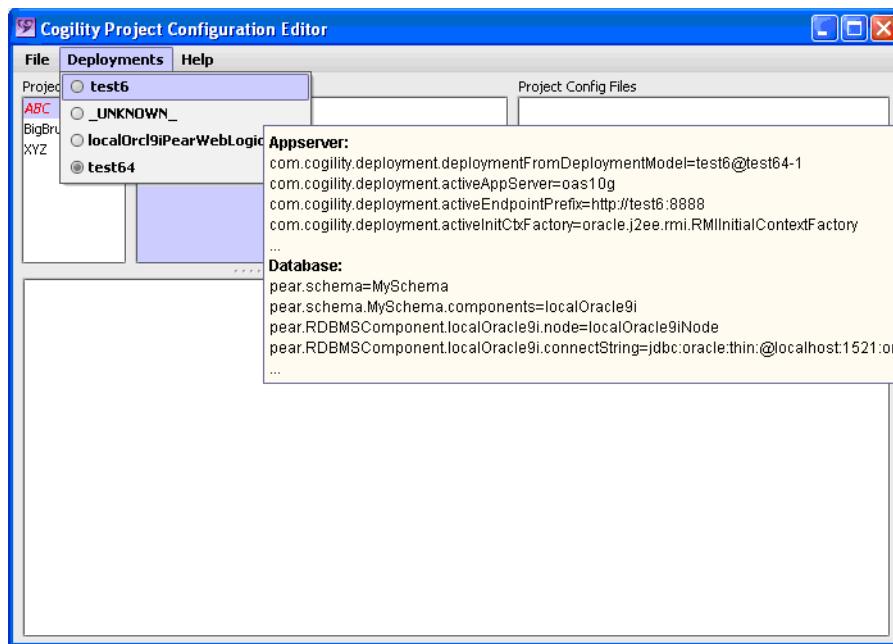
The deployment model supports the definition of multiple environments thus allowing you to push a model from a single source computer to multiple target environments. However, with every subsequent push, the application server and database configuration information is over-written. When a model is pushed, not only do the active database and application server configuration files get updated, but a second set of files gets saved based on the name of Deployment object defined in the Modeler. These files contain the same information that gets written to the configuration files, but they are not over-written unless the model is pushed using the same Deployment object. Thus, if you define multiple Deployment objects and have pushed each of them, you will have configuration information pertaining to each object.

Although there can be multiple deployment environments, only one deployment environment can be active. The active deployment environment is usually the one that last got pushed from the Modeler. To make another environment active, use the Project Configuration Editor to switch. The tool overwrites the active database and application server config files with the information from the second set of saved files.

The Project Configuration Editor can be used to toggle between different environments. It copies the deployment files into the Cogility Configuration Facility. The files must be copied prior to launching the desired Cogility tool, for example Action Pad. For more information on the Project Configuration Editor, see [Chapter 7, “Project Configuration Editor”](#) in the *Model Deployment & Execution in Cogility Studio*.

The Project Configuration Editor uses the Deployments menu to display known deployment information and to allow you to toggle to a different deployment. When multiple deployments exist, they are displayed in a list with an associated radio button. The radio button that is checked is the active deployment. Only one deployment can be checked at any one time.

Moving the cursor over a deployment will display a tool tip with the deployment's configuration information, allowing you to see its connection parameters.



Deployment information can be exported directly from Cogility Modeler by selecting either a single Deployment object or the DefaultDeploymentModel and using the menu **Selection > Import/Export > Export Deployment Configuration Information**.

To remove selections from the Deployments list, you must delete them manually.

Run-time respiratory

There are two repositories that you work with in Cogility Studio: the database run-time repository, where your model resides during execution, discussed here, and the file-based authoring repository, where changes to your model are tracked as you work with the model. See [“Authoring repository” on page 19](#).

You push your integration model on to a J2EE application server as a composite application that accesses the run-time repository during execution. The run-time repository is located on a database.

You push the model's artifacts and data into the run-time repository. You may see the run-time repository referred to as the PEAR repository. PEAR stands for Persistent Element Attributes and Relationships. See the guide, *Installing and Configuring Cogility Studio*.

The run-time repository may contain only one unique model schema at a time. Cogility Modeler cannot let you push a different model over a model already in the run-time repository. You can, and must, push a model to which you have added artifacts over its previous version. Pushing additions to a model is allowed. However, you cannot push a new version of a model from which you have deleted model artifacts.

During deployment, tables are generated on the run-time repository for the model objects (such as the M2E conversions, events, web services, and so on), and for the business objects that are created at run-time (such as Customers, Addresses and Products). The database schema for the run-time repository is divided between the fixed schema and the dynamic schema. The model objects describe the *fixed schema*, so called because their definitions do not change during execution. The business objects describe the *dynamic schema*, where objects may be created, edited, and deleted during execution.

Deployment extends the Persistent Entities, Attributes, and Relationships (PEAR) repository into the relational database and creates the tables necessary to run the application. PEAR is an object abstraction layer on top of relational databases and directory services developed by Cogility. It provides schema-based persistent data storage, which is modeled in terms of UML entities, attributes and relationships. The schema can be partitioned over multiple, independent data stores supporting a distributed heterogeneous environment.

PEAR is an object location and persistence facility. It provides a unified, simple, message-enabled interface to data in multiple new or legacy data stores. PEAR transforms enterprise data into information by removing the barriers to data storage and access, such as vertical data silos, complex and hidden schema definitions, network locations, connect strings, access authorization, SQL, LDAP, and other physical repository protocols. PEAR supports data concurrency and allows multiple threads over a single PEAR connection as well as multiple PEAR connections to a single repository. In addition, PEAR offers transactional support for relational database management systems. Pessimistic locking is used to ensure data integrity when multiple threads or connections to the same database occur at the same time.

PEAR provides a Java API that allows full meta-schema traversal and a simple interface for accessing data. This high level API allows a client program to focus on the data that it wishes to operate. The API handles data store connectivity and data query concerns. Because all physical repository information is managed centrally, PEAR does not require the client program to specify where the data resides. A client program only needs to know that it wants to create an entity and relate it to another, all other details are managed by PEAR.

During the development cycle, you may need to clear the database tables of the run-time repository. This may be necessary for various reasons:

- There have been schema changes that are incompatible with the existing database content.
- A developer may wish to start with a “clean slate,” getting rid of all existing data quickly.
- You may need to push a model based on a different schema over a model already in the database.

Before you can do any of these actions, you must drop the existing tables from the run-time repository. See “[Dropping the database tables](#)” on page 268.

Dropping the database tables

Note that clearing the database tables will remove both model data and run-time data.

To clear the database tables:

1. Back up your database if you intend to use it again.
2. Navigate to the %DCHOME%\MB\Scripts\DBTools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.
3. At the warning prompt, enter **Y** to continue.
The program will indicate when it is done dropping the tables.
4. Close the **Universal PEAR Table Dropper** command window.
The database is now empty, and you may load a new model into the run-time repository.

Deployment preparation

Prior to deployment, ensure that the model or artifact has the necessary requirements to deploy properly: consistency, immutability, a valid application server configuration, and a valid connection with the database. When you deploy (or push) your model or artifact into execution, the deployment process verifies the following. Deployment fails if any of these is not true:

- The model or artifact is immutable.
Immutable artifacts are those that have been turned over or versioned. See “[Turn-over version](#)” on page 23 and “[User modifications](#)” on page 9 of the guide, *Change Management in Cogility Studio*.
- The model or artifact is consistent.
There should be no red flags on your model or artifact when you deploy. See “[Inconsistencies](#)” on page 269 below.
- A valid connection to the run-time repository exists.
See “[Deployment Model](#)” on page 255.
- The connection parameters can be saved.
The connection parameters are written to the file, %DCHOME%\Config\Files\pear-params.txt, for accessed by Cogility applications.
- The model or artifact is additive to the model in the execution database.
See “[Non-additive model](#)” on page 270 below.
- The model can be deployed on the application server.
The application server is properly installed and the configuration variables are properly set. See the guide, *Installing and Configuring Cogility Studio*. Depending on which application server to which you are deploying, you may or may not have to run the application server before deployment. See “[Application server](#)” on page 270.

Inconsistencies

All of the model artifacts in Cogility Modeler must conform to the rules for their use as application objects. For example, an M2E conversion object must be associated with an event. Artifacts that do not conform are said to be inconsistent with the rules of Cogility Modeler. Inconsistent artifacts are identified by either a yellow flag or a red flag in Cogility Modeler’s tree view.

A yellow flag  warns that the artifact is incomplete. Such artifacts may not work in an executable model, but they may have other purposes. For example, by definition, a state machine must be associated with a start event. But if the state machine is incomplete, lacking an associated start event, it may be used as a state activity diagram.

A red flag  indicates that the model artifact is inoperative. Such artifacts may cause the model to fail during execution. For example, an M2E conversion must be associated with an event. Also, all compiler errors for actions semantics included with an artifact make the artifact inoperative.

Although Cogility Modeler lets you push in to execution a model with inconsistencies, the best practice is to resolve them; you should never attempt to run a model with red flag inconsistencies.

Non-additive model

Deployment fails if the model or artifact is not additive to the extant model or artifact in the run-time repository. An additive model or artifact has all extant artifacts, and it may include new artifacts.

If the deploying model or artifact includes deleted or modified information that could negatively impact instances already in the database, deployment fails. For example, if you delete a class from the model, your deployment fails with the following message in the push actions view (see “[Push actions view](#)” on page 280):

```
[Error]: #CannotSubtractSchema: Existing class DOM.MIM.Order is missing in the proposed metadata
```

Likewise, you receive an error message if you change an attribute’s data type, reduce an attribute’s size or reduce the multiplicity of any associations. For example:

```
[Error]: #AttrsMismatch: Class: DOM.MIM.Order Attribute: orderId - Old type = long New type = string
```

Changes to business object class attribute names will produce this type of error if there are run-time instances of the attributes in the repository when you try to re-push the changed attribute names. However, you may change an attribute name or type, or size of a String type, for uninstantiated attributes.

Before you push a new model into execution, remove any extant model from the run-time repository. See “[Dropping the database tables](#)” on page 268.

Application server

You run Cogility Studio with a Oracle WebLogic, IBM WebSphere, or Oracle application server. See the guide, *Installing and Configuring Cogility Studio* for more information. The application server must be running before you deploy your model.

Cogility Studio includes batch scripts for starting and stopping each of the supported application servers. These are mapped to selections in the Start menu.

Starting the application server

To start the application server:

1. For Oracle WebLogic: from the Start menu, select All Programs > Cogility Studio > Application Server > WebLogic > Start WebLogic.
2. For IBM WebSphere: from the Start menu, select All Programs > Cogility Studio > Application Server > WebSphere > Start WebSphere.
3. For Oracle Application Server from the Start menu, select All Programs > Cogility Studio > Application Server > OAS > Start OAS.

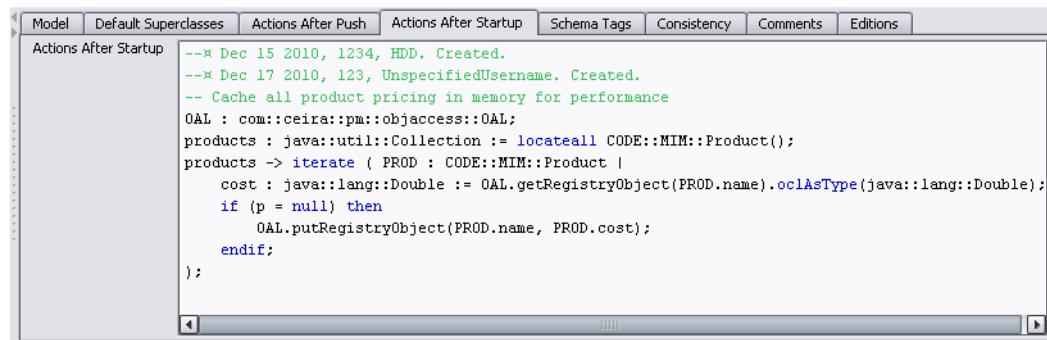
Actions after startup

Actions After Startup are executed in the application server when the Cogility Modeler starts. This can occur when the application server is started by the startup scripts, or when the application server is restarted after application server artifacts are pushed by **Actions > Push...** or by **Actions > Push to AppServer....**

The message *Running 'Actions After Startup'* is logged before the Actions and *Finished Actions After Startup* is logged after they have completed. If an error occurs during execution, a stacktrace is output to the log and the transaction around the actions is rolled back. You can check the log to tell whether the Actions executed successfully.

To define Actions After Startup:

1. In Cogility Modeler's tree view, select the model container.
2. Select the Actions After Startup tab in the editor view.
3. Enter the Action Semantics for the desired actions in the **Actions after Startup** field.



```
--> Dec 15 2010, 1234, HDD. Created.
--> Dec 17 2010, 123, UnspecifiedUsername. Created.
-- Cache all product pricing in memory for performance
OAL : com:cceira:pm:objaccess::OAL;
products : java::util::Collection := locateall CODE::MIM::Product();
products -> iterate ( PROD : CODE::MIM::Product |
    cost : java::lang::Double := OAL.getRegistryObject(PROD.name).oclAsType(java::lang::Double);
    if (p = null) then
        OAL.putRegistryObject(PROD.name, PROD.cost);
    endif;
);

```

Stopping the application server

To stop the application server:

1. For BEA WebLogic: from the **Start** menu, select **All Programs > Cogility Studio > Application Server > WebLogic > Stop WebLogic**.
2. For IBM WebSphere: from the **Start** menu, select **All Programs > Cogility Studio > Application Server > WebSphere > Stop WebSphere**.
3. For Oracle Application Server: from the **Start** menu, select **All Programs > Cogility Studio > Application Server > OAS > Stop OAS**.

Previewing a push

To preview the objects that will be pushed:

1. In Cogility Modeler's tree view, select the model container.
2. From the **Selection** menu, select **Actions > Preview - Push to Database...**.
3. In the **Select Deployment to be Used** dialog, select the deployment.
4. Under **Push Options**, select a push option. This is optional.

The push options are:

Use Map of Table names.

If you select this option, you must select the file with the table name map from the file browser dialog and select **OK**.

Force Push of Recompiled Action Semantics

If you are previewing a normal push, you should not have to check this box. Only under special circumstances should you need to do this. See the following section in the guide, *Model Deployment & Execution in Cogility Studio*, "[Force action semantics recompilation](#)" on page 32 for information about recompiled action semantics

5. After you make your selections, click **Use Deployment**.

Once the preview is generated, you can see the “Push Actions” tab for details on the push. Below is an example of a push preview:

```
[Info]: ----- Begin Deployment Parameters -----
[Info]: --- 
[Info]: --- Source Of Parameters = Deployment: WebLogic10_localhostOracle [ COG ]
[Info]: --- Connection Info From = RDBMSComponent: localhostOracle [ COG ]
[Info]: ---     userID          = pear
[Info]: ---     connectString    = jdbc:oracle:thin:@localhost:1521:orcl
[Info]: --- Application Server = WebLogic v9/v10: WebLogic10 [ COG ]
[Info]: --- 
[Info]: ----- End Deployment Parameters -----
[Info]: 
[Info]: ===== BEGIN PUSH DETAILS ===== 2009-07-26 22:11:17 =====
[Info]: 
[Info]: Creating Factory singleton: SimpleModel.CreateCustomer
[Info]: Creating definition for Operation:
SimpleModel.MIM.ServiceLevelAgreement.processTicket
[Info]: Creating definition for Operation: SimpleModel.MIM.ServiceLevelAgreement.ackReceipt
[Info]: Creating MIM: SimpleModel.MIM
[Info]: Creating Event: SimpleModel.TroubleTicket_Event
[Info]: Creating Event: SimpleModel.CreateCustomer_Event
[Info]: Creating Event: SimpleModel.CreationAck_Event
[Info]: Creating Message: SimpleModel.UpdateCustomerAddress_IWS.UCA_Arg
[Info]: Creating Message: SimpleModel.OWS.ValidateEmail.IsValidEmail.arg
[Info]: Creating Message: SimpleModel.UpdateCustomerAddress_IWS.UCA_Result
[Info]: Creating Message: SimpleModel.UpdateCustomerSLA_IWS.UCSLA_Result
[Info]: Creating Message: SimpleModel.OWS.ValidateEmail.IsValidEmail.result
[Info]: Creating Message: SimpleModel.UpdateCustomerSLA_IWS.UCSLA_Arg
[Info]: Creating CustomQuery: SimpleModel.SalesQuery
[Info]: Creating OutboundWebService: SimpleModel.OWS.ValidateEmail.IsValidEmail
[Info]: Creating definition for InboundWebService: SimpleModel.UpdateCustomerSLA_IWS
[Info]: Creating definition for InboundWebService: SimpleModel.UpdateCustomerAddress_IWS
[Info]: Creating IWSDeployment: SimpleModel.CogilityIWS
[Info]: Pushing IWSDeployment associations for: SimpleModel.CogilityIWS
[Info]: Pushing IWS Deployment Operation for: SimpleModel.UpdateCustomerSLA_IWS
[Info]: Pushing IWS Deployment Operation for: SimpleModel.UpdateCustomerAddress_IWS
[Info]: Creating M2E: SimpleModel.CreateCustomer_M2E
[Info]: Creating M2E: SimpleModel.TroubleTicket_M2E
[Info]: Creating E2M: SimpleModel.CreationAck_E2M
[Info]: Creating InboundWebService: SimpleModel.UpdateCustomerSLA_IWS
[Info]: Creating InboundWebService: SimpleModel.UpdateCustomerAddress_IWS
[Info]: Pushing new Behavior: SimpleModel.CreateCustomer.CreateCustomer_SM
[Info]:     [_CreateCustomer_SM_Top [ Chapter2 ]] Pushing STATE
[Info]:     [Stop_1216856677500 [ Chapter2 ]] Pushing STATE
[Info]:     [CreateCustomer_State] Pushing STATE
[Info]:     [Start_1216856500296 [ Chapter2 ]] Pushing STATE
[Info]:     [Transition_1216856827718 [ Chapter2 ]] Pushing TRANSITION
[Info]:     [Transition_1216856829921 [ Chapter2 ]] Pushing TRANSITION
[Info]:     [CreateCustomer_State] Pushing DoAction
[Info]: Pushing new Behavior: SimpleModel.MIM.ServiceLevelAgreement.ProcessTroubleTicket
[Info]:     [_ProcessTroubleTicket_Top [ Chapter7 ]] Pushing STATE
[Info]:     [ProcessTicketPerSLA] Pushing STATE
[Info]:     [AcknowledgeReceipt] Pushing STATE
[Info]:     [Stop_1217284737843 [ Chapter7 ]] Pushing STATE
[Info]:     [Start_1217284689484 [ Chapter7 ]] Pushing STATE
[Info]:     [Transition_1217284742843 [ Chapter7 ]] Pushing TRANSITION
```

```
[Info]:      [Transition_1217284740968 [ Chapter7 ]] Pushing TRANSITION
[Info]:      [Transition_1217284744593 [ Chapter7 ]] Pushing TRANSITION
[Info]:      [ProcessTicketPerSLA] Pushing DoAction
[Info]:      [AcknowledgeReceipt] Pushing DoAction
[Info]: Creating Method: SimpleModel.MIM.GoldSLA.MethodFor_processTicket
[Info]: Creating Method: SimpleModel.MIM.ServiceLevelAgreement.MethodFor_processTicket
[Info]: Creating Method: SimpleModel.MIM.BronzeSLA.MethodFor_processTicket
[Info]: Creating Method: SimpleModel.MIM.ServiceLevelAgreement.MethodFor_ackReceipt
[Info]:
[Info]: ===== END PUSH DETAILS ===== 2009-07-26 22:11:23 =====
[Info]:
[Info]: ----- Begin Deployment Parameters -----
[Info]: ---
[Info]: --- Source Of Parameters = Deployment: WebLogic10_localhostOracle [ COG ]
[Info]: --- Connection Info From = RDBMSComponent: localhostOracle [ COG ]
[Info]: ---     userID          = pear
[Info]: ---     connectString    = jdbc:oracle:thin:@localhost:1521:orcl
[Info]: --- Application Server = WebLogic v9/v10: WebLogic10 [ COG ]
[Info]: ---
[Info]: ----- End Deployment Parameters -----
```

Pushing the model into execution

To push the model into execution:

1. In Cogility Modeler's tree view, select the model container.
2. Turn over the model.

Turning over a model creates a new version of the model in the change management system. Before you can push a model into execution, you must turn it over. For more information, see “Turn-over version” on page 23 of the guide, *Change Management in Cogility Studio*.

3. If the application server is not running, start it.
See “Application server” on page 270.
4. With the model container selected, click the **Push** button .

5. In the **Deployments** dialog, select the deployment and click **Use Deployment**.

See “Deployment Model” on page 255. If you selected the DefaultDeploymentModel when you logged into Cogility Modeler (see “Logging into the current context” on page 17), upon pushing your model, you can choose from several default deployments. By default, a repository is loaded with a DefaultDeploymentModel, which defines multiple deployments. You can modify this model or import one that had been previously exported. The deployment model is saved as part of the model definition when you turn over your model.

During push, the push actions view lists the actions performed as objects are created in the execution environment. See “Push actions view” on page 280.

6. Under **Push Options**, check the box to force a push of either the **Recompiled Action Semantics** or **AppServer Artifacts**.

During a normal push, which includes the initial push and subsequent pushes, you should not have to check either box. Only under special circumstances should you need to do this. See the following sections in the guide, *Model Deployment & Execution in Cogility Studio*:

- “Force action semantics recompilation” on page 32 for information about recompiled action semantics
- “Application server only push” on page 29 for information about the application server artifact push

7. If the push is successful, a notice to that affect appears. Click **OK**.

The Cogility Modeler console window indicates that the model successfully deployed to the application server.

8. If the push is unsuccessful, you see an error message.

The error messages are also displayed in the push actions view at the bottom of the Cogility Modeler window. Some changes to the model's action semantics or changes to the configuration parameters prior to pushing the model may require that the application server be restarted. The dialog notifies you of this.

Actions after push

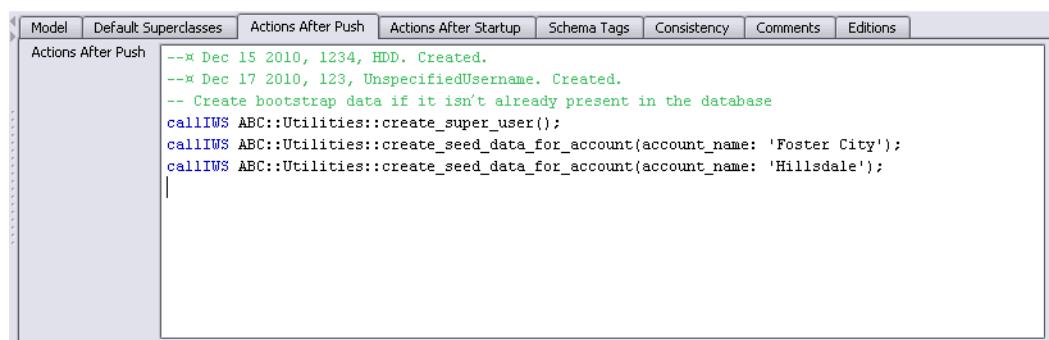
A Push operation pushes the model (which is the definition of the application) into the database. It is not capable of making any data changes. Cogility modeler allows you to define data modifications that need to occur after the model is pushed to the database, using **Actions After Push**.

For example, the model may contain the class and association definitions for Users and Roles. The **Actions After Push** can be used to create actual instances of users and roles. This removes the need to manually run a separate script after the push is done in order to create this root data for the system, by allowing this logic to be encapsulated in the Push process.

Actions After Push are executed by the Modeler after **Actions > Push to Database...** or after the database portion of **Actions > Push...**, but before pushing AppServer artifacts, if any. If an error occurs while executing the after push actions, the push will be interrupted and rolled back.

To define Actions After Push:

1. In Cogility Modeler's tree view, select the model container.
2. Select the Actions After Push tab in the editor view.
3. Enter the Action Semantics for the desired actions in the **Actions after Push** field.



```
--> Dec 15 2010, 1234, HDD. Created.  
--> Dec 17 2010, 123, UnspecifiedUsername. Created.  
-- Create bootstrap data if it isn't already present in the database  
callIWS ABC::Utilities::create_super_user();  
callIWS ABC::Utilities::create_seed_data_for_account(account_name: 'Foster City');  
callIWS ABC::Utilities::create_seed_data_for_account(account_name: 'Hillsdale');
```



Cogility Modeler Interface

This appendix describes the Cogility Modeler user interface and those features common to all model object editor interfaces. The various model objects may have unique functionality and object specific interfaces. These are described in the other volumes of the Cogility Studio documentation. See the “Preface” on page 13 for a complete list of these volumes.

Views

A model loaded into the current context is shown in the picture below. The model objects are represented by three primary views: a tree view, a editor view and a messages view.

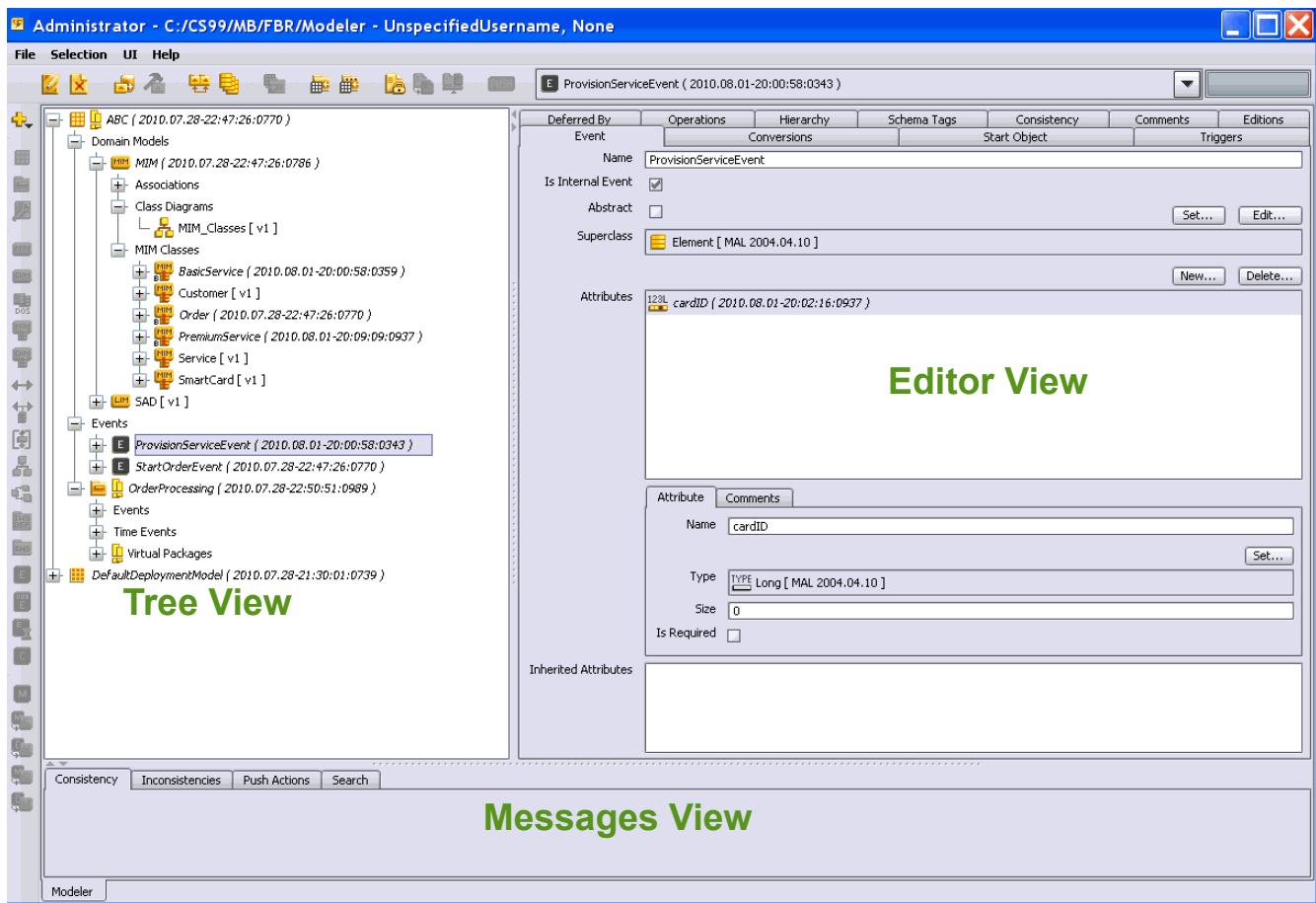


Figure A-1: The views of Cogility Modeler



Tree view

The objects of a model are listed under the model container in the tree view. As shown in the picture above, this is the leftmost pane of the Cogility Modeler window. To expand a branch, click the plus sign +. To collapse the branch, click the minus sign -. Double-clicking a model object causes a new editor window to open for it. Selecting a model object displays its editor window in the editor view.

Editor view

The contents of the object selected in the tree view are displayed in the editor view, the rightmost pane in the Cogility Modeler window. You can also open an object's editor view in a separate editor window by double clicking the object in the tree view. Different objects may have different contents. Refer to the separate volumes of the documentation for information about specific model object contents. These documents are listed in the “Preface” on page 13. There are several view elements common to all object contents, described in the following sections. In the editor view, click on the tab for the Hierarch, Editions, Comments and Consistency information.

Hierarchy

Model objects which are subject to inheritance and which have attributes are described in the hierarchy where they appear. For example, the hierarchy for the GoldSLA class shown below shows that GoldSLA has a parent class called ServiceLevelAgreement.

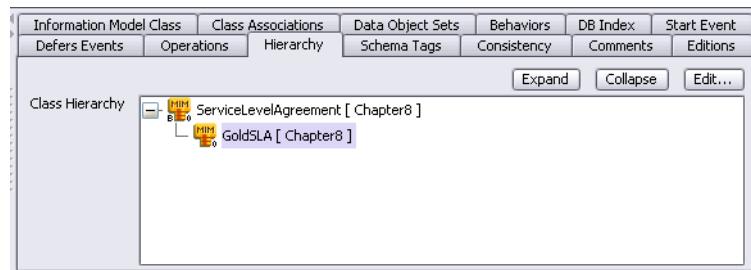


Figure A-2: Hierarchy tab

Editions

As you work with a model and make changes to model objects, each time you turn over a model object, you create a version of it in the repository described as an edition. The Editions tab displays a list of object editions. Selecting an edition displays its version information.

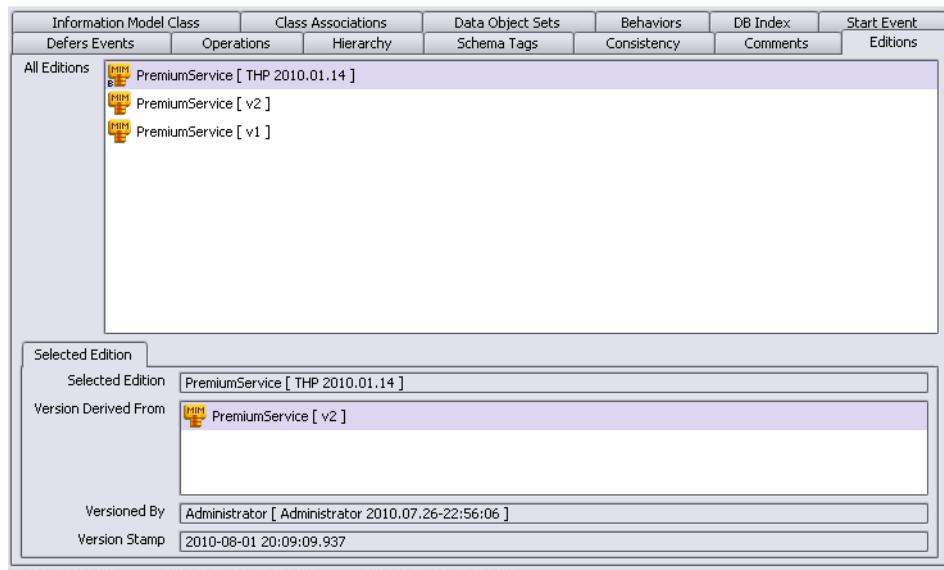


Figure A-3: Editions tab

Consistency

The editor view of each model object includes a tab with a consistency view. It displays the same information as the consistency view itself. See [Messages view](#), next.

Messages view

The messages view, the lowermost pane of Cogility Modeler, contains tabs for consistency messages, Push Action messages, and Searches.

Consistency Tab

The Consistency tab contains information about objects that are inconsistent with the rules of Cogility Modeler. When an object is not consistent, its entry in the tree view is shown with a red or yellow exclamation mark next to it, and the explanation is in shown on the consistency tab.

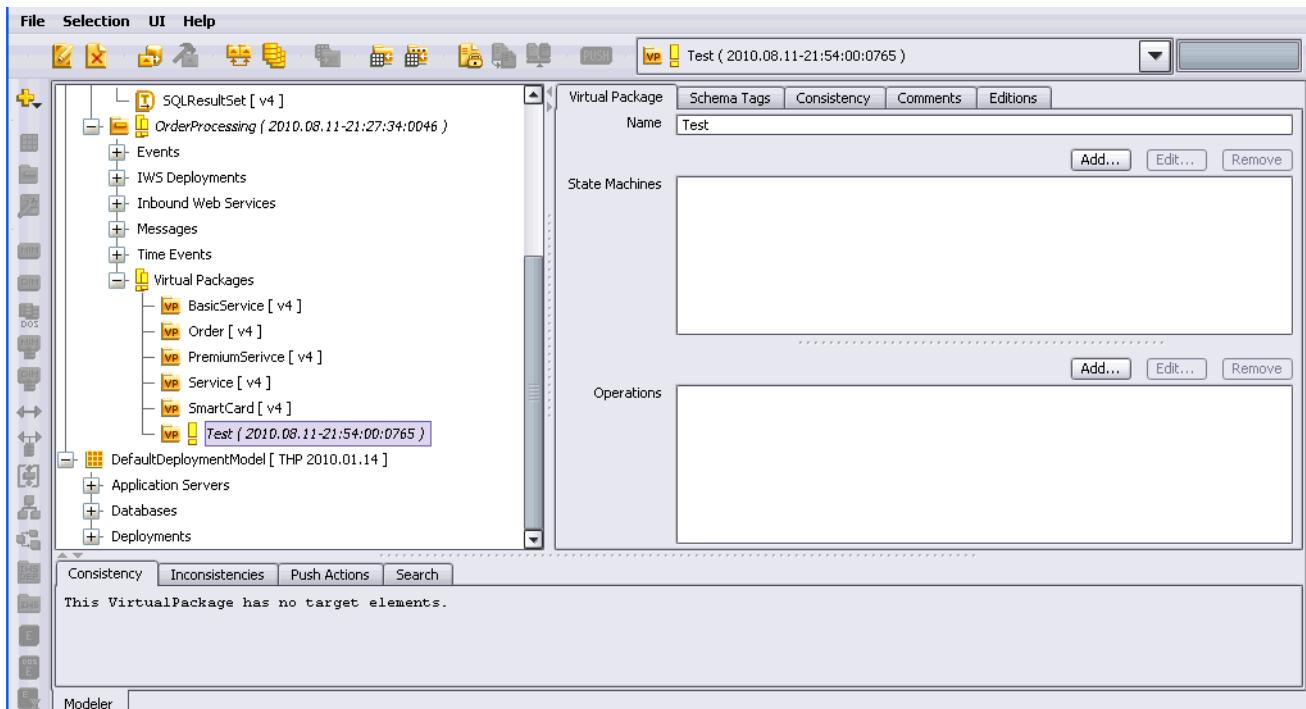


Figure A-4: Consistency Message

Consistency checking ensures that the model, from a structural standpoint, is complete. Consistency errors flagged with a red exclamation mark are critical and must be addressed. A model cannot be pushed if it contains a critical consistency error. Consistency errors flagged with a yellow exclamation mark represent a warning and can be addressed at the author's discretion. The combination of both yellow and red indicates that the object with the consistency error is contained within another object or group of objects.

If the error occurs within some action semantics, the consistency tab describes the point in the action semantics where the error occurs. Action semantic errors can relate to such problems as missing or

incorrect variables, attributes not found, or other syntax errors. The error message on the Consistency tab indicates the type of problem found.

```

Consistency Inconsistencies Push Actions Search

#### Known Variables:
# boPoid -> JavaClassDefinition(java.lang.String)
# parent -> _provision_Top [ v4 ]
# self -> PremiumService ( 2010.08.17-20:12:18:0640 )
# smPoid -> JavaClassDefinition(java.lang.String)
# startEvent -> JavaClassDefinition(java.lang.Object)
# state -> WaitForFirstGame
####

#### Syntax error at line 2
Action Semantics error/warning in "doActivityActions":
    #### Syntax error at line 2

Line 1: --R Aug 08 2010, 002, UnspecifiedUsername. Created.
Line 2: self.setCardStatus(cardID: startEvent.cardID, statusVal: 'FOOTBALL CHANNELS ON');
           ^
#### Syntax Error
    #### [PFEN] Attribute 'cardID' not found in type 'java::lang::Object'.
    # There are 0 known Attributes for 'java::lang::Object'.
    ####
        Attribute 'cardID' not found in type 'java::lang::Object'.

Line 3: CARD : ABC::MIM::SmartCard := locate ABC::MIM::SmartCard(id: startEvent.cardID);
Line 4: ORD : ABC::MIM::Order := CARD.getOrder();
Line 5: publish ORD do ABC::OrderProcessing::ServiceProvisionedEvent();
Line 6:

#### Known Variables:
# boPoid -> JavaClassDefinition(java.lang.String)
# parent -> _provision_Top [ v4 ]
# self -> PremiumService ( 2010.08.17-20:12:18:0640 )
# smPoid -> JavaClassDefinition(java.lang.String)
# startEvent -> JavaClassDefinition(java.lang.Object)
# state -> FootballChannelsOn
####

#### Syntax error at line 2
A State Machine should specify a Start Event by which it will be invoked to be executable. Set it from the "Start Event" t
Modeler

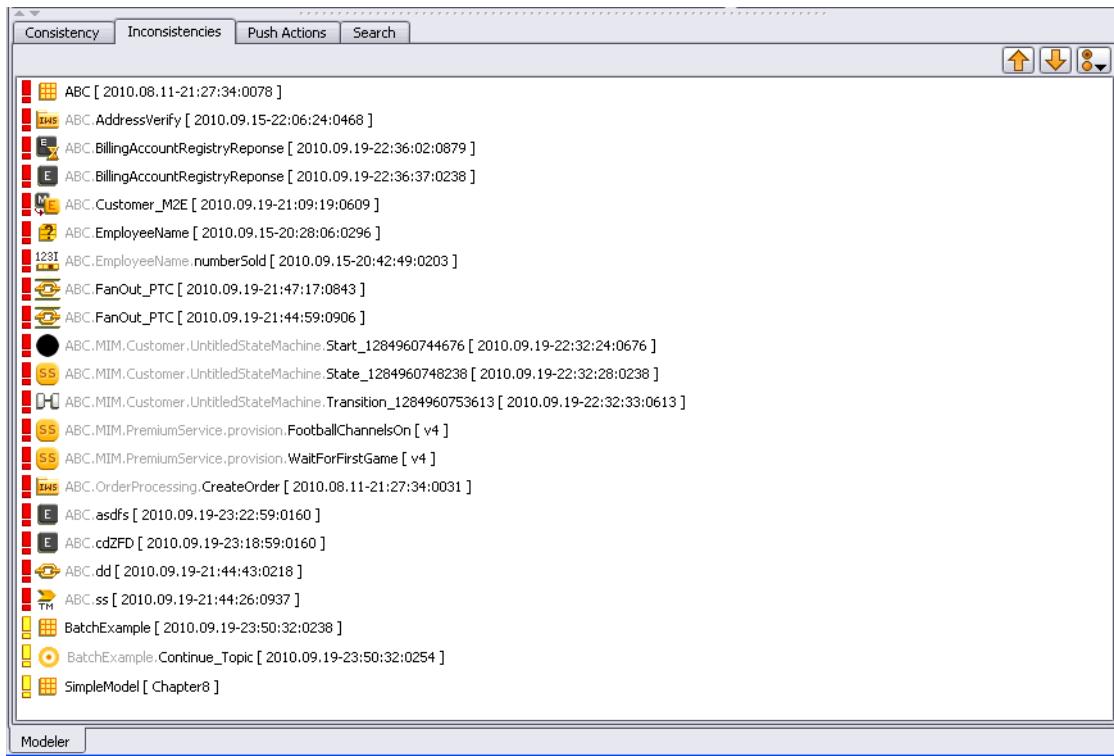
```

Figure A-5: Consistency message sample

Whenever you enter some action semantics then click out of the entry field, Cogility Modeler automatically compiles the action semantics. Any compiler inconsistencies are reported in the consistency view.

Inconsistencies view

Located on the Inconsistencies tab, the Inconsistencies view shows a list of all inconsistencies within the Modeler tree. You can click on an entry in the list to access the related object in the Modeler tree.



The Inconsistencies view shows all inconsistencies within the model. The view updates each time background consistency checking completes a cycle.

When you click on an entry in the "Inconsistencies" list, Modeler navigates to that object in the Modeler tree.

The Inconsistency pane includes the following navigation buttons:

- The up-arrow button displays the previous inconsistency error in the list.
- The down-arrow button displays the next inconsistency error in the list.
- The Options button allow you to select display options for the Inconsistencies view. The options are:
 - Show XML key (default)
 - Hide XML key
 - Sort by XML key (default)
 - Sort by Name
 - Show Yellow flags (default)
 - Hide Yellow flags

Push actions view

Located on the Push Actions tab, the push actions view reports errors that occur when a model is pushed into execution. Push failure can occur for a number of reasons. For example, if the model

being pushed is in conflict with the model already existing in the PEAR database repository, it cannot push successfully. First, Cogility Modeler displays a warning message, as follows.



Figure A-6: PEAR error message

The details of the conflict between the model pushed and the extant model are described in the push action view. In the example below, the model being pushed is a prior version of the extant model and is missing several classes.

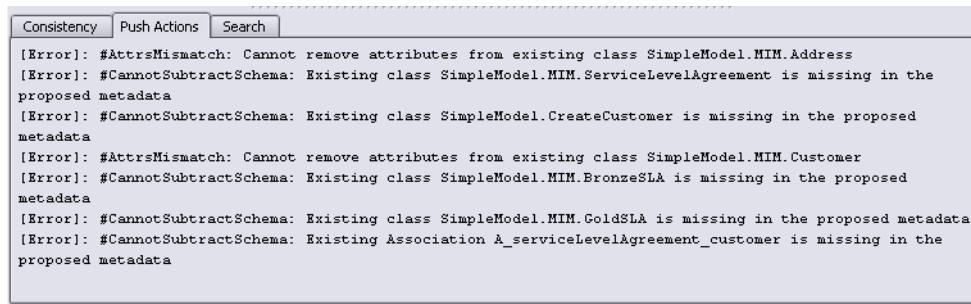


Figure A-7: Push actions tab

Search

Cogility Modeler features the capability to search a model for specific strings. You can Search by doing one of the following:

- Clicking the Search button on the Search tab. See “[Search dialog](#)” on page 283.
- Selecting the Search sub-menus in the Selection menu.

On the Selection>Search menu, you have two options:

- Search in this Hierarchy Only...
This search option opens the Search dialog. See “[Search dialog](#)” on page 283.
- Search For Name In Action Semantics
This search option performs a search for the selected item. See “[Search For Name In Action Semantics](#)” on page 281.

Search For Name In Action Semantics

The Search For Name In Action Semantics menu choice is available when you select a single element in the model tree. It searches for the full XML key reference in any Action Semantics. The results appear in the Search tab.

[Figure A-8 on page 282](#) shows that MIM has been selected as the search element. The search looks for the full XML key, in this case ABC::MIM. Each instance of the search result is shown in the Search tab.

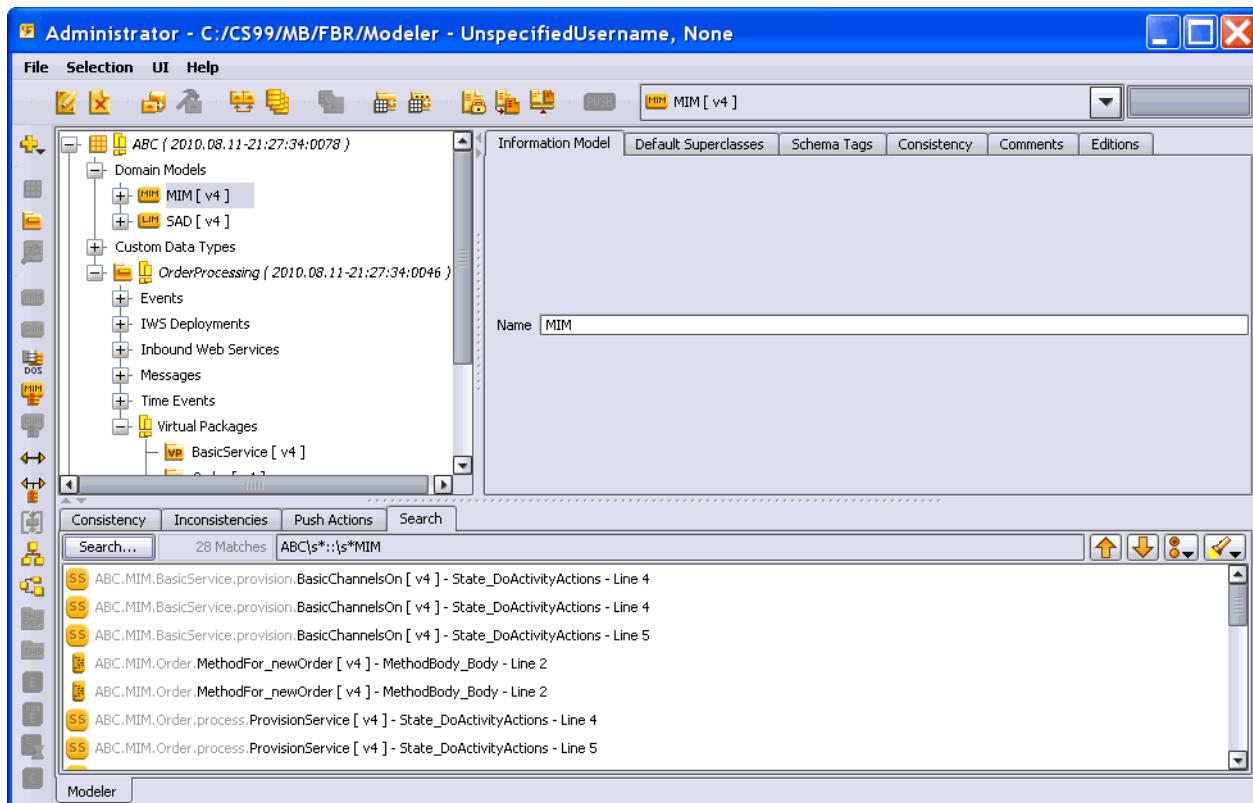


Figure A-8: MIM selected as item for search by name

To view a result, click on the result item in the results list. The location of the search result item is highlighted in the model tree and in the Action Semantics, as shown in [Figure A-9](#).

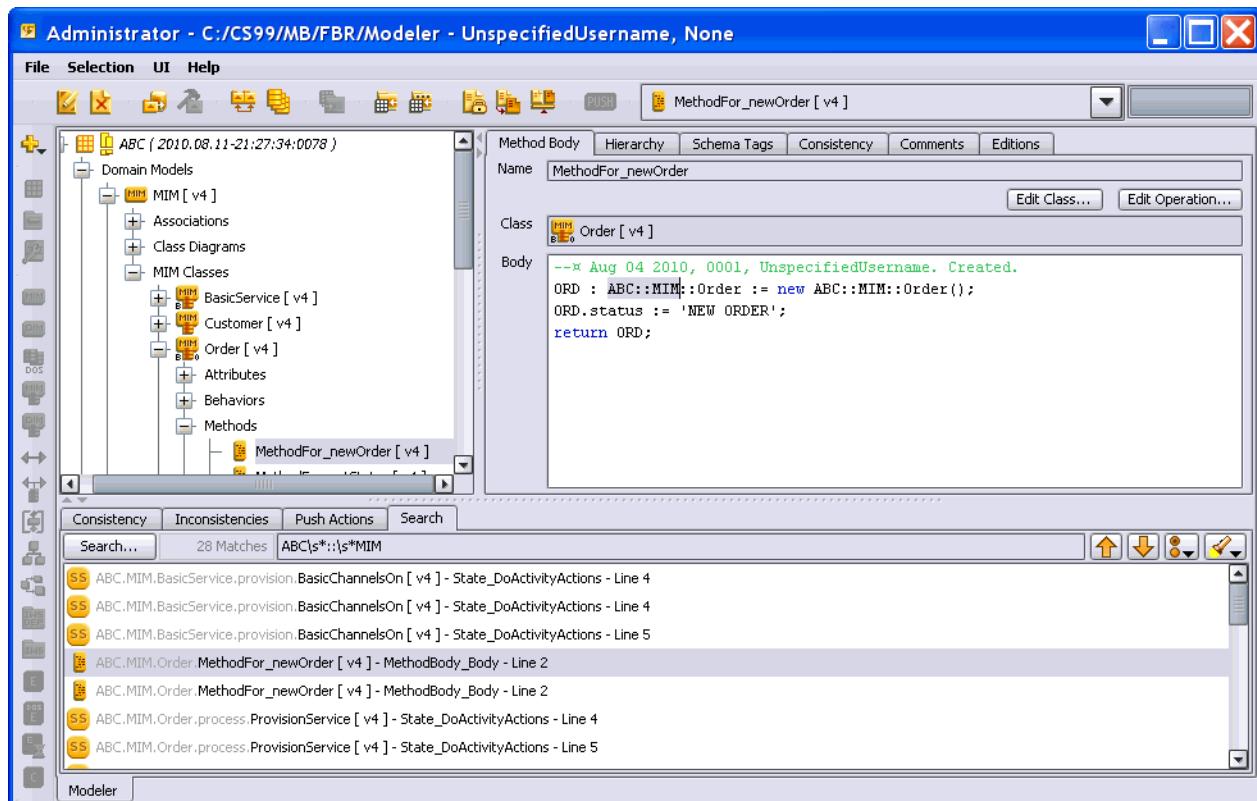


Figure A-9: Selected search result for MIM

Search dialog

You can specify search criteria in the Search dialog.

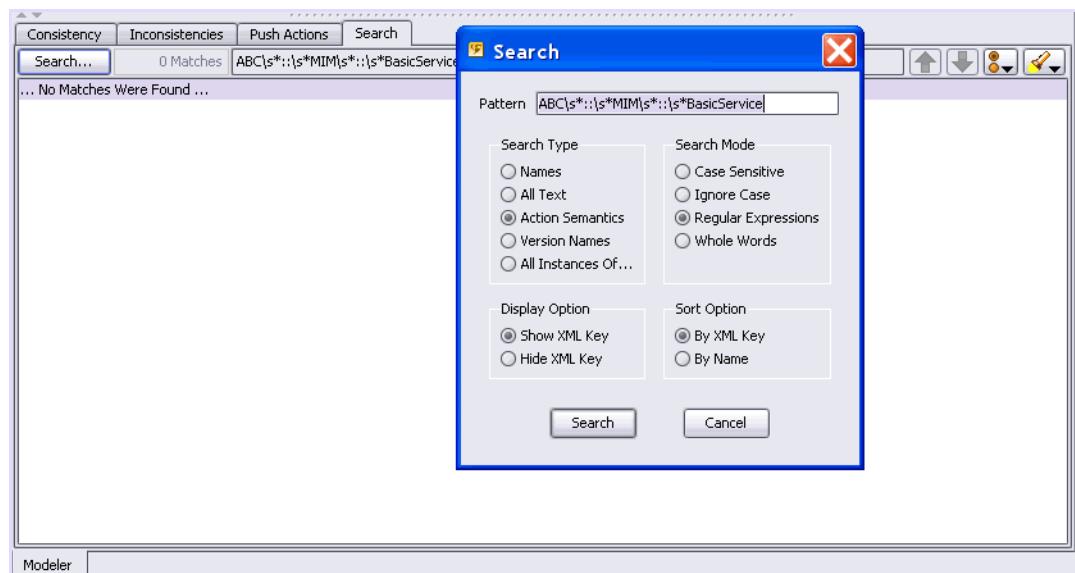


Figure A-10: Search tab

The search feature works at tree level, reporting the artifacts that include the searched-for string. To find and replace strings in text fields, use the Find/Replace feature. See “[Find / Replace](#)” on page 314.

To search for text strings in a model:

1. In the lowermost pane, click the **Search** tab.
2. Click **Search**.
3. Enter the pattern to search for in the **Pattern** field.
4. Select your search criteria options.
5. Click **Search**.

History

Cogility Modeler keeps track of the objects you have edited (that is, opened but not necessarily changed) and makes them available in a history list in the upper left corner of the window. The most recently edited model object appears at the top of the list, followed by the one before that and so on. To access the list, click the down arrow. You can return to a previously edited object by selecting it from the list.

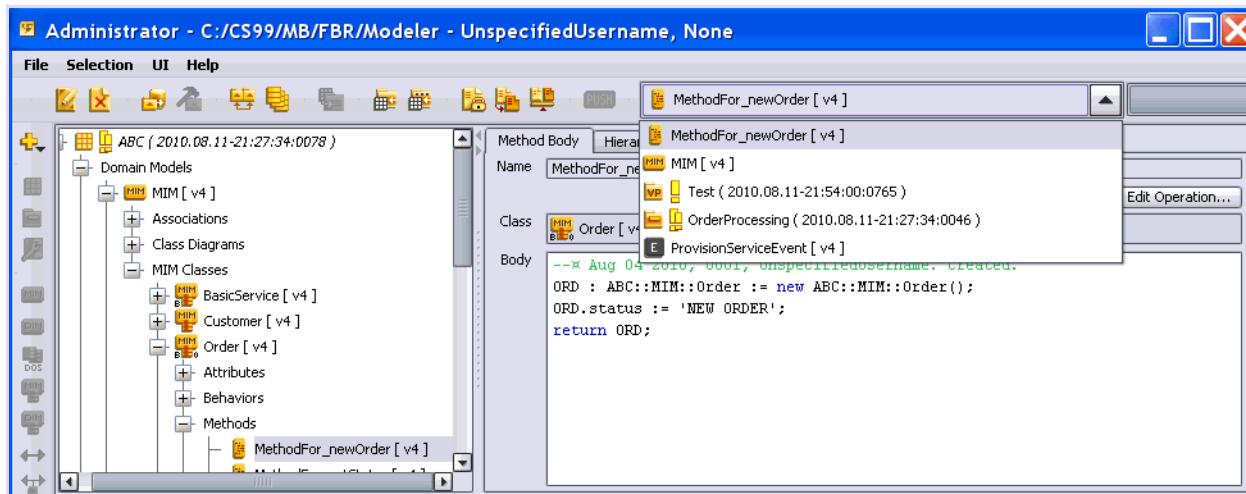


Figure A-11: History

By default, the History dropdown lists the last five objects selected in the Tree View. The following configuration parameter can be modified to change this number:

```
com.ohana.object.textsearch.historysize=
```

Note: An object appears only once in the list even if it has been accessed more than once in the last five selects.

Menus and tools

The full range of Cogility Modeler functionality is available through the menus at the top of the window. Some of this functionality is repeated in the tool bar icon buttons that appear below the menus and on the left edge of the window. All toolbar button will display a tooltip when the mouse cursor is moved over the button.

Also, right-click selections are available in the editor and tree views. In the editor view, the right-click selections are those of the buttons in the view. For example, if an Edit button appears in the tree view, a right-click selection for Edit will be available as well. In the tree view, the options of the Selection menu are repeated in the right-click selections. See “[Selection Menu](#)” on page 285.

Note: All menus whose label contains a “...” suffix, indicates that the user will be prompted for additional information.

File Menu

Menu Item	Icon	Description
Reload Configuration Parameters		This selection allows you to update the running Cogility Modeler with edited configuration parameters. See “ Reloading configuration parameters into Cogility Modeler ” on page 25.
Re-Parse WSDL / XSD Schema		This menu flushes the XSD Definition cache, re-parses the contents of each WSDL/XSD Schema object, and populates the aforementioned cache with the current definitions.
Exit		Exits Cogility Modeler. You can also click the close window icon  in the upper right corner. See “ Exit Cogility Modeler ” on page 18.

Selection Menu

To use the selection menu, you first select the model object for which you want complete a task from the menu. For example, if you are adding model objects to a model container, you must first select the model container. Then choose an option from the Selection menu. You can also bring up the Selection menu by placing the cursor in the tree view and right-clicking the mouse.

Menu Item	Icon	Description
Edit		Object editing functions.
Edit Selection...		Opens a new editor window for the selected object.
Delete Selection...		Deletes the selected object from the current context, but not from the authoring repository. You can replace deleted objects through the user modifications view. See “ User modifications ” on page 9 of the guide, <i>Change Management in Cogility Studio</i> .
Move Selection...		Opens the Select New Location dialog box and allows you to move the selected Modeler class to a different package. See “ Move selection ” on page 296.

Menu Item	Icon	Description
Copy Display Name		Places the name label of the selected object into your system's cut/paste buffer. You can then paste (with Ctrl v) the label into a new object that you want to have a similar name, then edit the new object for a new name based on the one you pasted in.
Copy Fully Qualified Name		Places the fully qualified name of the selected object into your system's cut/paste buffer. You can then paste (with Ctrl v) the label into a new object that you want to have a similar name, then edit the new object for a new name based on the one you pasted in.
CM Operations		Change management system operations.
Turn Over...		Turns over the model and creates a version of it in the authoring repository. See " "Turn-over version" on page 23 of the guide, <i>Change Management in Cogility Studio</i> .
Force a Turnover...		Allows you to create a new turn-over version of the model without modifying the version names of the contained model objects. Compare to " "Forcing versioning" on page 19 of the guide, <i>Change Management in Cogility Studio</i> . Whereas forcing versioning over a model applies the version name to all contained model objects, forcing a turn over applies the new version name to only the model container.
Place Model Under CM		Places a model under change management. See " "Placing a model under change management" on page 6 of the guide, <i>Change Management in Cogility Studio</i> .
Show User Modifications...		Opens the user modifications view for the selected object. See " "User modifications" on page 9 of the guide, <i>Change Management in Cogility Studio</i> .
Show Element Revisions...		Opens the element revision history window for the selected object. See " "Version history" on page 21 of the guide, <i>Change Management in Cogility Studio</i> .
CM Preferences...		Opens the change management preferences window.
Import/Export		Functions for importing and exporting model objects from and to files.
Export to Configured File Set		Exports the selected model objects. See " "Exporting to a configured file set" on page 27 of the guide, <i>Change Management in Cogility Studio</i> .
Export to File Set...		Performs a multi-file export of the selected Model into a specified directory. See " "Model export" on page 25 of the guide, <i>Change Management in Cogility Studio</i> .

Menu Item	Icon	Description
Import from Configured File Set		Imports the model objects from a selected configured file set. See “ Importing from a configured file set ” on page 30 of the guide, <i>Change Management in Cogility Studio</i> .
Import Selected File Set...		Imports a selected file set. See “ Importing from a selected file set ” on page 31 of the guide, <i>Change Management in Cogility Studio</i> .
Export to Single File...		Exports a model object to a COG file. See “ Exporting as a single file ” on page 26 of the guide, <i>Change Management in Cogility Studio</i> .
Import from Single File...		Imports a model object from a COG file. See “ Model artifacts ” on page 24.
Export Map of Table Names...		Produces a mapping of modeled objects to their corresponding table names in the existing runtime database. The exported file can be used to create tables in a different database with the same mapping to the model.
Import R2.x Formatted Model...		Imports a model object created under version 2.x for use in the current version.
Add Existing Model from Repository...		Adds to the current context a model object from the repository. See “ Adding a model from the repository ” on page 32 of the guide, <i>Change Management in Cogility Studio</i> .
Undo-Redo		<p>For the selected model object, the options in this menu let you revert the most recent change to the object’s editable fields. For more elaborate change management, see “User modifications” on page 9 of the guide, <i>Change Management in Cogility Studio</i>.</p> <p>These options work only for the object selected in the tree view. For example, if the TroubleTicket_Event object is selected, only changes to its editable fields (Name and abstract) are subject to undo-redo. The fields of its attributes, such as customerID, though they may be changed within the editor window, are not subject to undo-redo. If you changed the name of the customerID attribute, you could not undo this change because the customerID attribute is not selected in the tree view.</p> <p>Undo-redo is most useful for reverting changes to action semantics and other long listings in editable fields.</p>
Undo Last Attribute Change		Reverts the last change to the text in an editable field.
Preview Undo		Displays the last edition to the text field.
Redo Last Undo		Reverts the last undo to the original edit
Preview Redo		Displays the last revision to the text field.

Menu Item	Icon	Description
Versioning		Versioning options of the change management system.
Version...		Creates a version of the selected object. See “ Versioning specific artifacts ” on page 18 of the guide, <i>Change Management in Cogility Studio</i> .
Revise		Makes the selected object mutable and assigns it a version name with the current timestamp without actually changing its contents. You can select multiple unmutable objects (by holding down the Ctrl key as you select) and revise them to a mutable state with the current timestamp.
Duplicate		Makes a copy of the selected object in the current context.
Replace With...		Replaces the selected object with a version in the authoring repository. See “ Model element replacement ” on page 34 of the guide, <i>Change Management in Cogility Studio</i> .
Compare With...		Compares the selected object with a version in the authoring repository. See “ Version comparison ” on page 32 of the guide, <i>Change Management in Cogility Studio</i> .
Compare Non-Current Versions...		Compares to two versions of an object, both of which are not in the current context.
Actions		The Actions menu includes options for several aspects of modeling in Cogility Studio.
Push...		Pushes a model into execution. See “ Pushing the model into execution ” on page 273.
Push to AppServer...		Allows the entire model to be deployed to the application server without creating any database artifacts.
Push to Database...		Allows the entire model to be deployed to the database without creating any application server artifacts
Preview - Push to Database...		Allows you to see what will be pushed if a push is performed. No data is pushed during the preview. See “ Previewing a push ” on page 271
Push Single IWS...		Pushes into execution a specified inbound web service. See “ Pushing a single artifact ” on page 31.
Push Single Class Behaviors...		Pushes into execution specified class behaviors. See “ Pushing a single artifact ” on page 31.
Push Single Class Operations...		Pushes into execution specified class operations. See “ Pushing a single artifact ” on page 31.
Push all Custom Queries...		Pushes into execution all custom query objects regardless of a prior push.
Push BusinessObject Schema Only...		Pushes the business objects (MIM and DIM classes an such) into execution. See “ Pushing the model into execution ” on page 273.

Menu Item	Icon	Description
Create Behavior Invocation Artifacts...		Creates the "boilerplate" elements and associations needed to invoke/start a selected StateMachine.
Generate Application Clients...		OAS OEMS JMS Database Specific - Generates packaged J2EE application clients for Cogility Support Tools. This is done automatically during push.
Generate Deployed Model Comparison Report		Compare the current context model to what has been pushed. Results are written to the PEAR Action tab. See " "Deployed model comparison report" on page 39 in the <i>Model Deployment & Execution in Cogility Studio</i> .
Import WSDL from File...		Imports a web service definition language file from within the local file system. See " "Importing a WSDL from a file" on page 213.
Import WSDL from URL...		Imports a web service definition language file from a remote location. See " "Importing a WSDL from a URL" on page 213.
Try Outbound Web Service...		Tests an outbound web service. See " "Testing an outbound web service" on page 214.
Check Database Connection		<p>Checks the connection to the database for the selected object:</p> <ul style="list-style-type: none"> ■ If a LIM is selected, checks the connection using the LIM's User ID, Password and Connect String. ■ If a Model or Deployment Model is selected, prompts you to select an deployment (LIMs will NOT be included in this list) and then checks the connection using the User ID, Password, and Connect String specified for the deployment.

Menu Item	Icon	Description
Show Association Chains		<p>Shows the association chains when you select one or two DOMClasses (DIMClasses or MIMClasses). The selected classes are the "Source Class" and "Target Class" for the Association Chains. If only one DOMClass is selected, it is used as both the Source and Target Class.</p> <p>An "Association Chain" is a path from the Source Class to the Target Class via Roles, Associations and intermediate Classes. A chain could contain a single segment directly from the Source Class to the Target Class, or, it could contain numerous segments through several intermediate Classes. The chains are displayed in a tree with one branch per chain. Each chain starts with the Source Class and ends with the Target Class. Within a chain, each segment has five parts (a Class, the Role for that Class, the Association between the two Classes, the Role for the other Class, and the other Class).</p>
Output Message Editor Messages...		Creates a file with the JMS destinations defined for a model. See "Exporting message definitions" on page 108 .
Output Action Semantics to File...		Creates a text file with the model's action semantics scripts. You can choose which types of model objects to include.
Set History Comment Username...*		Allows you to enter a Username to be added to history comments. This Username will remain as the "active" Username until the end of the editing session or until it is changed to a different Username by this same menu pick.
Set Job Number...*		The "Set Job Number" menu item may have a different label. That label is set to whatever you call a Job Number in your Problem Tracking System (for example, if you call them Change Requests, the label might be "Set Change Request Number"). When "Set Job Number" is selected, a dialog will prompt you to enter the new Job Number.
* These menus can be hidden by disabling OCL History Comments		
Domain Modeling		Options for working with the domain model appear in this menu selection.
Add Model		Creates a model container in the current context. See "Model container" on page 21 .
Add Package		Creates a package branch in the model tree. You can use the package to hold other model objects such as outbound web services. See "Packages" on page 22 .
Add Factory		Creates a factory object in the selected model. See "Factories" on page 48 .

Menu Item	Icon	Description
Add Master Information Model (MIM)		Creates a master information model object in the selected model. See “ Master information model (MIM) ” on page 29.
Add Distinct Information Model (DIM)		Creates a distinct information model object in the selected model. See “ Distinct information model (DIM) ” on page 30.
Add Data Object Set (DOS)		Creates a data object set object for the selected DIM. See “ Data object set (DOS) ” on page 46.
Add MIM Class		Creates a MIM class object for the selected model. See “ Classes ” on page 33.
Add DIM Class		Creates a DIM class object for the selected model. See “ Classes ” on page 33.
Add Association		Creates an association object for the selected model. See “ Associations ” on page 51.
Add Association Class		Creates an association object for the selected model. See “ Association classes ” on page 54.
Add Behavior		Opens the state activity diagram editor to create a state machine for a selected class. See “ State machine ” on page 112.
Add Class Diagram		Opens the class diagram editor to create a class diagram for the selected model. See “ Class diagrams ” on page 49.
Add System View Diagram		Opens the system view diagram editor to create a system view diagram for the selected MIM or DIM. See “ System view diagrams ” on page 321.
Add Inbound Web Service Deployment		Creates an inbound web service deployment for the selected model. See “ IWS deployment ” on page 205.
Add Inbound Web Service		Creates an inbound web service for the selected model. See “ Inbound web services using SOAP over HTTP ” on page 228.
Add HTTP Deployment		Creates an HTTP Deployment for the selected model. See “ HTTP Deployment ” on page 234
Add HTTP Inbound Web Service		Creates an HTTP inbound web service for the selected model. See “ Modeling HTTP operations inbound web services ” on page 230
Add Web Service WSDL		Creates a WSDL for an outbound web service. See “ Automatic WSDL import ” on page 212.
Add Outbound Web Service		Creates an outbound web service for the selected model. See “ Outbound web services ” on page 212.
Add WSDL/XSD Schema		Creates a WSDL/XSD Schema object for the selected model. See Chapter 8, “XSD Artifacts” .
Add Java Package Alias		Creates a Java Package Alias object for the selected model. See “ Java package alias ” on page 320

Menu Item	Icon	Description
Add Event		Creates an event object for the selected model. See "Standard event" on page 84.
Add DOS Event		Creates a DOS event object for the selected model. See "DOS event" on page 85.
Add Time Event		Creates a time event object for the selected model. See "Time event" on page 86.
Add Transient Class		Creates a transient class for the selected model. See "Transient class" on page 89.
Add Custom Query		Creates a custom query object for the selected model. See Chapter 7, "Custom Queries" ..
Add Virtual Package		Creates a virtual package for the selected package. See "Virtual Packages" on page 23
Add Deployment Model		Creates a deployment model object in the current context. See "Configuration Parameters" on page 25.
Add Deployment		Creates a deployment object for a selected deployment container. See "Creating a deployment model" on page 257.
Conversions		Options for working with JMS message conversions are listed in this menu.
Add Message		Creates a message object for the selected model. See "Messages" on page 103.
Add M2E Conversion		Creates a message to event conversion object for the selected model. See "M2E conversion" on page 91.
Add E2M Conversion		Creates an event to message conversion object for the selected model. See "E2M conversion" on page 93.
Add M2D Conversion		Creates a message to DOS conversion object for the selected model. See "M2D conversion" on page 94.
Add D2M Conversion		Creates an DOS to message conversion object for the selected model. See "D2M conversion" on page 96.
Add Destination		Creates a destination object for the selected model. See "Destinations" on page 102.
Transformation Units		The options for working with data transformation objects are listed in this menu.
Add Opaque Transformation Map		Creates an opaque transformation map object for the selected model. See "Opaque transformation map" on page 146.
Add Transformation Map		Creates a transformation map object for the selected model. See "Transformation map" on page 149.
Add Classifier Map -		Creates a classifier map object for the selected model. See "Classifier map" on page 153.

Menu Item	Icon	Description
Add Feature Map		Creates a feature map object for the selected model. See “ Feature map ” on page 158.
Add Classifier Feature Map		Creates a classifier feature map object for the selected model. See “ Classifier feature map ” on page 160.
Add Serial Transformation Chain		Creates a serial transformation chain object for the selected model. See “ Serial transformation chains ” on page 168.
Add Parallel Transformation Chain		Creates a parallel transformation chain object for the selected model. See “ Parallel transformation chains ” on page 169.
Legacy Modeling		These options let you incorporate preexisting database information into your model.
Add Legacy Information Model (LIM)		Creates a legacy information model object for the selected model. See Chapter 3, “Legacy Information Model” .
Add LIMClass		Adds a legacy information class object for the selected model. See Chapter 3, “Legacy Information Model” .
Add Execute Statement		Creates an SQL execute statement object for the selected legacy information model. See “ Execute statements ” on page 68.
Add Select Statement		Creates an SQL select statement object for the selected legacy information model. See “ Select statements ” on page 68.
Add SQL Block		Creates an SQL block object for the selected legacy information model. See “ SQL block objects ” on page 73.
Discover Tables...		Discover and display for activation the tables for the selected legacy information model. See “ Discover Tables ” on page 62
Chart and HTML Views		These options for creating HTML Views and Chart Views.
Add Chart View		Creates a new Chart View object for the selected model. See “ Chart View ” on page 250
Add Chart Properties		Creates a new Chart Properties object for the selected model. See “ Chart View ” on page 250
Add HTML View		Creates a new HTML View object for the selected model. See “ HTML Views ” on page 237
Add HTML Table		Creates a new HTML Table object for the selected model. See “ HTMLTable ” on page 240
Add HTML Table Template		Creates a new HTML Table Template object for the selected model. See “ Creating an HTML table template ” on page 248
Add HTML Table Cell Template		Creates a new HTML Table Cell Template object for the selected model. See “ Creating an HTML table cell template ” on page 249

Menu Item	Icon	Description
Open Data Transformation Editor		Opens the data transformation editor. See “ Data Transformation Editor ” on page 173.
Search		
Search in this Hierarchy Only...		Searches in the current hierarchy.
Search for Name in Action Semantics		Searches for Full XML Key references in any action semantics. Full XML Key references are assumed to be double-colon separated and can have zero or more whitespace characters before and/or after the double-colons. The results appear on the “Search” tab at the bottom of the Modeler.

UI menu

The user interface menu lets you choose options for displaying objects and content.

Menu Item	Icon	Description
No Content		If this box is checked, no content is displayed in the editor view.
Tabbed Content		By default, content is displayed in the editor view separated by tabs at the top of the view.
Grouped Content		Check this box to display content all in the same pane of the editor view.
Show Mutable Objects in Italics		By default, when an object is edited or revised, its label appears in italics.
Show Object Icons		By default, the icons for model objects appear to the left of their labels. Uncheck this box to hide the icons.
Show Group Icons		Object groups also have icons, though usually the icon is just a file folder. By default these are hidden. Check the box to reveal the icons.
Show Version Names		By default each object label includes its version name in brackets. Uncheck this option to hide the version names.
Show Package Icons		By default, packages are shown with their icons in the tree view. Uncheck this box to hide the icons. See “ Packages ” on page 22.
Sort Packages to Top		By default, packages are sorted in the tree view by name (label). Uncheck this box to sort them by name starting at the top of the tree.

Menu Item	Icon	Description
Background Consistency Check		By default, each time you complete an edit (either by creating an object or clicking outside of an editable field) Cogility Modeler performs a consistency check. If you uncheck this box, consistency checking occurs only when you turn over the model. Action semantics compilation error checking, which is usually part of consistency checking, is unaffected by this option and still occurs when you click outside an action semantics field after completing an edit.

Help menu

Menu Item	Icon	Description
Open Documentation		Opens the documentation browser window.
About Cogility Modeler		Opens a window describing the release, version, licensing and copyright information for Cogility Modeler..

Additional menus

In addition to the functions that are available from the Main Menu, there are several options that are available on from the right-click menus. To access a right-click menu, place the cursor in the appropriate field, right-click the mouse, and select the menu option.

Menu Item	Location	Description
New...	List fields	Opens a dialog to either select the type of object to create or to name the object you wish to create.
Edit...	List fields	Opens an object editor for the selected object in the list field.
Delete...	List fields	Opens a confirmator to delete the selected object in the list field.
Find in Modeler Tree	Text/Action Semantics edit fields	Select the full XML key of the object, then select this menu option. Finds the selected item and navigates to it in the tree view pane.
Open Editor On	Text/Action Semantics edit fields	Opens a standalone Object Editor on the object.
Copy Display Name	Search results pane	Places the name label of the selected object into your system's cut/paste buffer. You can then paste (with Ctrl v) the label into a new object that you want to have a similar name, then edit the new object for a new name based on the one you pasted in.

Menu Item	Location	Description
Copy Fully Qualified Name	Search results pane	Places the fully qualified name of the selected object into your system's cut/paste buffer. You can then paste (with Ctrl v) the label into a new object that you want to have a similar name, then edit the new object for a new name based on the one you pasted in.
Find Syntax Error	Consistency pane	Locates a syntax error in the consistency pane. Use this option to find which Action Semantics contains the error. Once you have determined which Action Semantics has the error, you can open the editing pane for that Action Semantics and use the "Go to Syntax Error" menu option to find the erroneous syntax element.
Go to Syntax Error	Action Semantics edit fields	Selects the action semantics which has a syntax error (use the Find Syntax Error in the Consistency pane first to find which Action Semantics has the error.)

Move selection

The "Edit > Move Selection..." menu action allows many of the Modeler classes to be moved into and out of packages within the Modeler tree. The menu item is enabled if one or more Elements are selected. If the selected object cannot be moved, a dialog indicates that the object cannot be moved.

If the Element can be moved, a "Select New Location" dialog opens. This dialog lists the common locations to which all selected Elements can be moved.

To move a selected object:

1. Select the Element to be moved in the tree view pane.
2. Either right click on the Element and select Edit>Move selection, or select Edit>Move Selection from the Selection main menu list.
3. In the Select New Location dialog box, select the new location from the displayed list.
4. Click OK.

The element is moved to the next location.

All Elements are restricted as to where they can be moved. Following are the rules and the object types that are restricted by each rule:

- Rule 1 – Some objects must be located directly under the Model, or in a Package that is directly under the Model, or in a Package under a Package that is directly under the Model, and so on. These Elements can be moved from under the Model into a valid Package, or, from a valid Package back under the Model, or, from a valid Package into a different valid Package. This rule applies to the following Element types:
 - D2M E2M M2D M2E
 - Event DOSEvent TimeEvent
 - OpaqueTMap TMap CMap FMap CFMap
 - IWS OWS HTTP_IWS IWSDeployment HTTPDeployment
 - Handler HandlerChain
 - WSDL WSDL/XSDSchema
 - ChartView ChartProperties

- HTMLView HTMLTable HTMLTableTemplate HTMLTableCellTemplate
- CustomQuery Destination JavaPackageAlias
- Factory SerialTransformationChain ParallelTransformationChain (requires user approval)
- Rule 2 – Some Elements must be located directly under a PIM (MIM, DIM or LIM), or, in a Package that is directly linked to the PIM. These Elements can be moved from under the PIM into a valid Package, or, from a valid Package back under the PIM, or, from a valid Package into a different valid Package. This rule applies to the following Element types:
 - MIMClass DIMClass DOS (requires user approval)
 - Association AssociationClass (requires user approval)
 - ClassDiagram
 - ExecuteStmt SelectStmt SQLBlock
- Rule 3 – One object can be located under a Model or a PIM or in a Package under either. It can be moved any other location:
 - SystemViewDiagram
- Rule 4 Packages can be moved if all Elements in the Package can be moved, or, if you approve the move of the Package containing Elements that require approval. Valid locations are based on the Package's current location in the tree. If the Package is under a PIM, it can be moved elsewhere under that PIM (according to rule 2). Otherwise, it must be under the Model and can be moved elsewhere under the Model (according to rule 1). Additionally, a Package cannot be moved into itself or into any Package underneath itself. This would cause invalid circular references.
- Rule 5 – Messages can be moved, or, at least some can. If a Message is an Argument or Result of a WebService, it cannot be moved. If the Message is defined under the Model, it can be moved elsewhere under the Model (according to rule 1).
- Rule 6 – Comments cannot be moved because they are connected to their owning Element by an Aggregate association. An Element appears in the Modeler tree underneath the Element that is its NameSpace, or, underneath the Element it is connected to by an Aggregate association.
- Rule 7 – VirtualPackages can be moved. Valid locations are Packages that are linked to the Model (directly, or, through other Packages that are linked to the Model).

The following Element types cannot be moved:

- Model MIM DIM LIM
- ActivatedTable ActivatedColumn
- StateMachine State Transition Guard
- Operation Method
- HTMLTableCellArea HTMLTableColumn HTMLTableRow
- DOMAttribute CustomQueryAttribute CustomQueryArgument FeatureReference
- DeploymentModel Deployment DatabaseComponent AppServerType
- WSDeployment WSOperation
- Any other type that is not included in this list or any of the previous lists.

Select Object(s) dialog

Throughout Cogility Modeler, when you are creating model artifacts, you may be prompted by the Select Object(s) dialog to choose artifacts to associate with the artifact you are creating. For example,

when you set the return type of an operation, the available types in the model are listed in the Select Object(s) dialog, as shown in the figure below.

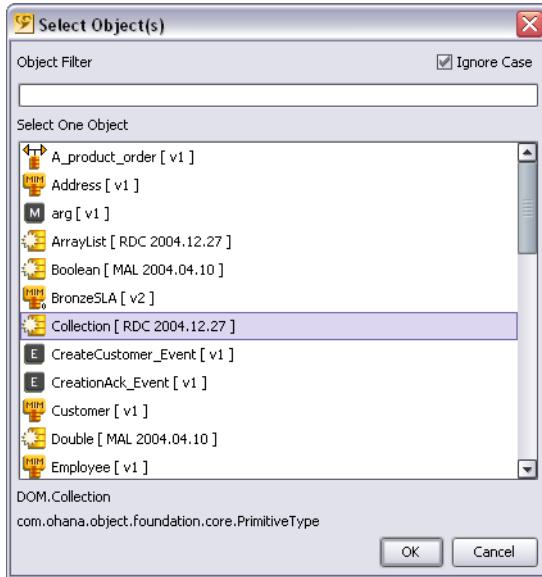


Figure A-12: Select Object(s) dialog

The artifacts and their icons appear in the diagram editor window. If the dialog displays a large list of elements, you can enter the name of class in the Object Filter field to search the list. By default, the filtered list will include artifacts with names that start with the characters in the field, regardless of case. Unchecking the Ignore Case check box forces the artifact names to conform to the case of the characters in the field. You can select multiple artifacts (where it is legal to do so) by holding down the Ctrl key as you select, or select a contiguous range of artifacts by holding down the shift key as you select.

Common editing functions

In any action semantics field in Cogility Modeler and in the edit view of Cogility Action Pad, you can right-click and select from the following editing functions.

Menu Item	Description
Cut, copy, paste	<Ctrl+X, Ctrl+C, Ctrl+V> These features are the standards found in most software.
Undo	<Ctrl+Z> Reverts the last keystroke performed in the action semantics field. This Undo operation uses a modified version of the Java UndoManager and doesn't work exactly the same as the Undo that you might find in Word or NotePad. The primary difference in the Java implementation is apparent when you type a sequence of characters. In NotePad, Undo reverts the entire sequence. In the Java implementation, Undo reverts one character at a time, or several characters at a time depending upon how fast you typed them.
Redo	<Ctrl+Y> Reverts the last Undo performed in the action semantics field.

Menu Item	Description
Save	<Ctrl+S> Performs the same "save/update" that a loss-of-focus performs, without losing focus.
Find/Replace...	<Ctrl+F> Brings up the Find/Replace dialog. See " Find / Replace " on page 314.
Go to Line Number...	<Ctrl+G> Opens the "Go To Line Number" dialog that reports the current line number and lets you enter a line number to move the cursor. If the line number is valid, the cursor moves to the beginning of that line (and the window scroll to make that line visible, if necessary).
Check Single Quotes...	<Ctrl+'> Performs a check of the action semantics field text for single-quote problems, including unmatched single quotes or too many consecutive single quotes. If problems are detected, a dialog appears that identifies the problems and the line numbers on which those problems occur.
Go To Syntax Error	<Ctrl+E>, in Action Semantic field) Scroll the OCL text to make the Syntax Error (found by "Find Syntax Error" visible).
Find Syntax Error	<Ctrl_E>, in Consistency pane) Locates an OCL Syntax Error, if any, and scroll the text to make it visible.
Discard Current Changes	<Ctrl+Backspace> Restores the field to the value it had the last time its changes were committed to the persistent repository (at the last FocusLost event). If there are changes, it will prompt you to make sure you want to discard all of your changes. If there are no changes, it will beep.
Search	<p>You have a choice of several search options. See "Search" on page 281.</p> <p>The search options are:</p> <ul style="list-style-type: none"> ■ Names ■ All Text ■ Action Semantics ■ Version Names
Model assist at Cursor	<Ctrl+Space> Brings up the Model Assist dialog. See " Model assist " on page 300.
Syntax assist at Cursor	<Ctrl+Shift+Space> Brings up the Syntax Assist dialog. See " Syntax assist " on page 313.
Print...	Allows you to print the entire contents of an action semantics field or Action Pad file; opens a standard print dialog where you can set the printer, number of copies, and other options.
Add History Comment...	Adds a OCL history comment is the Action Semantic field the cursor is in. See " History comments " on page 317.
Remove History Comment...	Removes a OCL history comment is the Action Semantic field the cursor is in. If there are multiple OCL history comments, the most current comment is removed. See " History comments " on page 317.

Tab and indent

Pressing the Tab key <Tab> inserts a tab at the location of the cursor. Selecting two or more lines and pressing the Tab key indents the lines, inserting a tab at the start of each line. A tab is equal to three spaces in Cogility Modeler and two spaces in Cogility Action Pad. With more than one line selected, holding the Shift key while pressing the Tab key <Shift+Tab> removes the indentation (unindenting).

These operations are affected by the following conditions:

- The selected code must contain at least one line terminator (a semicolon ;), and the selected code must contain whole lines. Un-indenting only removes leading tab characters -- it doesn't remove leading spaces.
- If you have several whole lines selected and press <Tab>, a tab character is inserted at the beginning of each line and the lines remain selected.
- If you have several whole lines selected and press <Shift+Tab>, a tab character is removed from the beginning of each line. If a line begins with any character other than a tab character, that line is left alone. At the end, the lines remain selected.
- If you select a few characters at the beginning of a line and press <Tab>, those characters are replaced by a tab character (the selection did not include a line terminator).
- If you select a few characters at the beginning of a line and press <Shift-Tab>, nothing happens (the selection did not include a line terminator, and <Shift-Tab> does not do anything, even if nothing is selected).
- If you have several lines selected, but the selection begins in the middle of the first line (or ends in the middle of the last line, or both), and press the <Tab> character (or <Shift-Tab>), nothing happens (indentation only occurs when whole lines are selected). The <Tab> is ignored and your selection remains unchanged.

Model assist

Model assist provides you with instant access to your model artifacts as well as bound variables, keywords, and Java actions. This meta-data driven command completion system makes the task of specifying action semantics much easier by intelligently displaying context-sensitive suggestions for what you might want to specify next in an action semantics listing or query expression field.

The model assist window displays a list of action semantics expressions that may be inserted in the line at the current cursor position. Based on the syntax of the current line, model assist attempts to complete the line with an appropriate function, Java action, variable, keyword, or qualified business object class. In the example below, using model assist following the line `custR := new` displays the model assist window with all of the available classes under the `SimpleModel` model. To complete

the line with the appropriate class, you select the class from the list and double-click to insert it in the line.

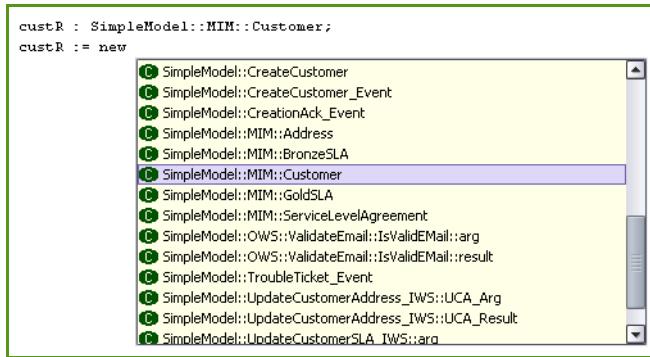


Figure A-13: The model assist window

For each of the recognized syntactical structures, model assist reads partial entries (the first few letters of an attribute name or method name, for example) and filters the list of possible targets accordingly. In the picture below the list has been filtered for those methods starting with the letter 's'.

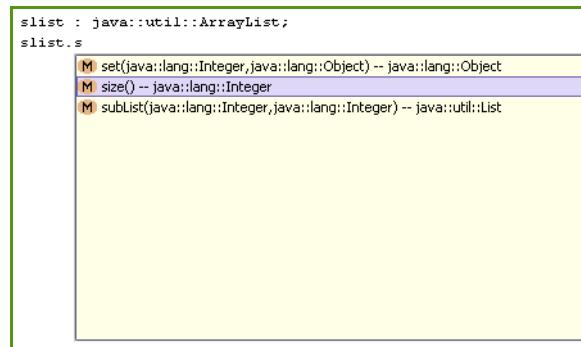


Figure A-14: Model assist list of filtered methods

For each recognized syntax, model assist displays a list of appropriate objects from which you may choose to insert in your listing. These objects may be model-defined such as classes, operations, parameters, attributes, associations, and so on. They may also be action semantics keywords, Java actions, methods, bound variables, and the like.

Model assist also works in custom query Expression fields. You can use it to find classes, class attributes, association role names, association classes, association class attributes, and query input arguments. See [Chapter 7, “Queries”](#). in the guide *Using OCLPlus with Cogility Studio*.

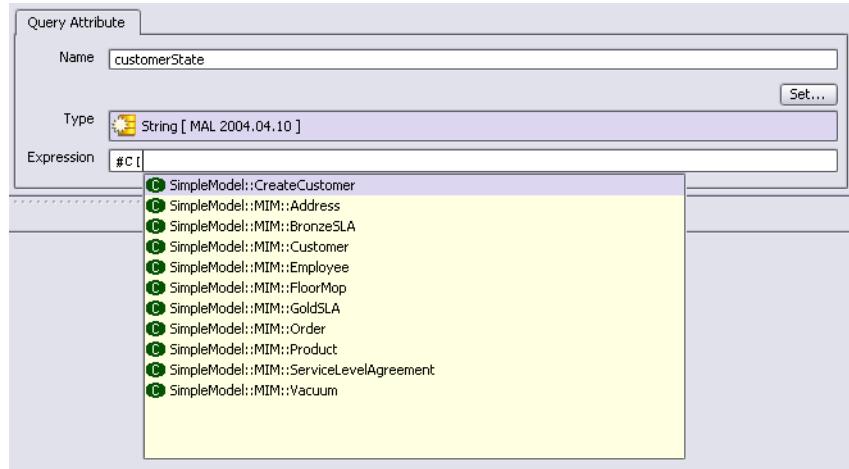


Figure A-15: Model assist with custom query expression syntax

Model assist recognizes the syntax of the current line of the action semantics, and displays a list of objects of the type appropriate to the syntax. The objects are identified by icons described as follows:

- | | | |
|--|------------------------|--|
| A | Association classes | See “ Association classes ” on page 303 |
| A | Class attributes | See “ Class attributes ” on page 304 |
| B | Bound variables | See “ Bound variables ” on page 305 |
| C | Model classes | See “ Model classes ” on page 304 |
| C | Java classes | See “ Java classes ” on page 305 |
| C | Transient classes | See “ Transient classes ” on page 305 |
| J | Java actions | See “ Java actions ” on page 306 |
| K | Keywords | See “ Keywords ” on page 306 |
| L | LIM | See “ LIM ” on page 306 |
| M | Methods | See “ Methods ” on page 307 |
| O | Operations | See “ Operations ” on page 307 |
| P | Parameters | See “ Parameters ” on page 308 |
| P | Publish message | See “ Publish ” on page 308 |
| P | Publish event | See “ Publish ” on page 308 |
| R | Roles | See “ Roles ” on page 309 |
| S | SQL statement | See “ LIM ” on page 306 |
| V | Variables | See “ Variables ” on page 310 |
| X | XSD element definition | See “ XSD element definition ” on page 311 |
| X | XSD type definition | See “ XSD type definition ” on page 311 |

Model assist does not provide assistance for a period after a method call. Nested method calls can be too complex for the parser to analyze successfully. Instead, define a variable, assign the results of the method call to that variable, then use Model assist after a period following that variable.

For example, the following use of model assist provides no items:

```
lastName : java::lang::String;
lastName := (locate Model::MIM::Customer( ssn: input.ssn )).<Ctrl-Space>
```

Rewriting the above as follows provides the desired assistance:

```
lastName : java::lang::String;
cust : Model::MIM::Customer;
cust := locate Model::MIM::Customer( ssn: input.ssn );
lastName := cust.<Ctrl-Space>
```

You can use model assist in either Cogility Modeler or Cogility Action Pad. See “[Using model assist](#)” on page 313.

Association classes

Within the context of an association class, model assist displays a list of possible targets from those defined in the model. They appear with an association class icon  in the list.

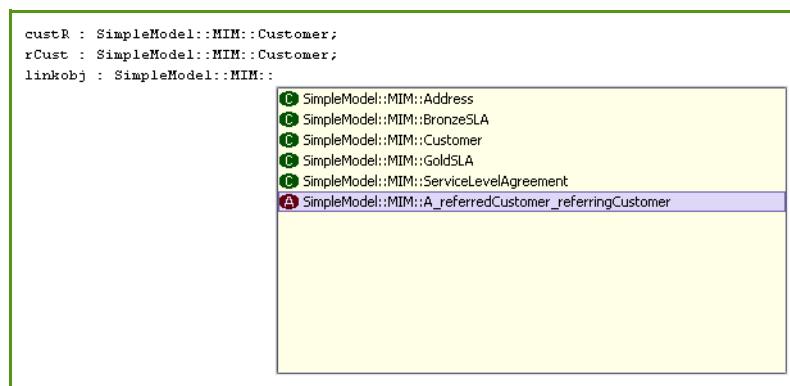
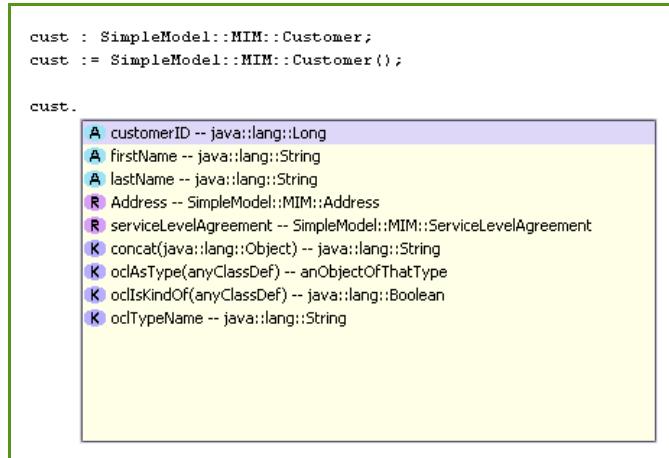


Figure A-16: Model assist list of association classes

Class attributes

Within the context of a class variable, model assist displays a list of class attributes. These appear with an attribute icon  in the list.



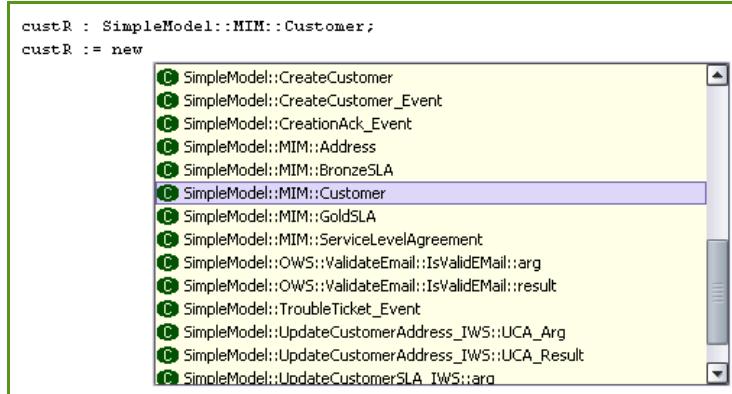
```
cust : SimpleModel::MIM::Customer;
cust := SimpleModel::MIM::Customer();

cust.
     customerID -- java::lang::Long
     firstName -- java::lang::String
     lastName -- java::lang::String
     Address -- SimpleModel::MIM::Address
     serviceLevelAgreement -- SimpleModel::MIM::ServiceLevelAgreement
     concat(java::lang::Object) -- java::lang::String
     oclAsType(anyClassDef) -- anObjectOfThatType
     oclIsKindOf(anyClassDef) -- java::lang::Boolean
     oclTypeName -- java::lang::String
```

Figure A-17: Model assist list of attributes

Model classes

Within the context of a class, model assist displays a list of classes. These appear with a class icon  in the list.



```
custR : SimpleModel::MIM::Customer;
custR := new
     SimpleModel::CreateCustomer
     SimpleModel::CreateCustomer_Event
     SimpleModel::CreationAck_Event
     SimpleModel::MIM::Address
     SimpleModel::MIM::BronzeSLA
     SimpleModel::MIM::Customer
     SimpleModel::MIM::GoldSLA
     SimpleModel::MIM::ServiceLevelAgreement
     SimpleModel::OWS::ValidateEmail::IsValidEMail::arg
     SimpleModel::OWS::ValidateEmail::IsValidEMail::result
     SimpleModel::TroubleTicket_Event
     SimpleModel::UpdateCustomerAddress_IWS::UCA_Arg
     SimpleModel::UpdateCustomerAddress_IWS::UCA_Result
     SimpleModel::UpdateCustomerSLA_IWS::arg
```

Figure A-18: Model assist list of classes

Java classes

Model assist provides a list of Java classes in the classpath. Each appears with the Java class icon  .

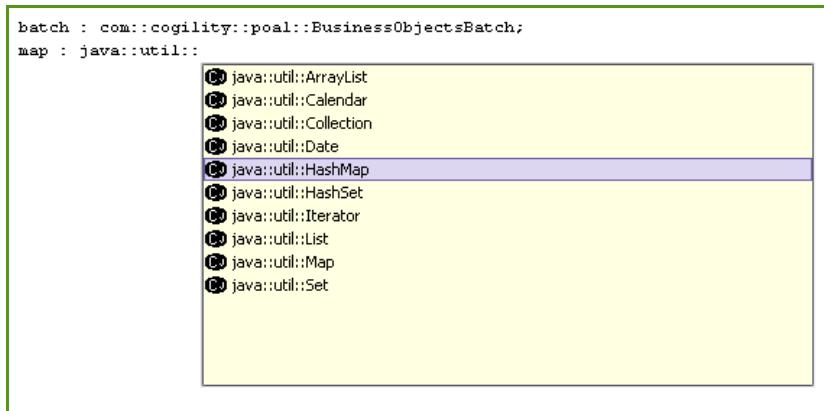
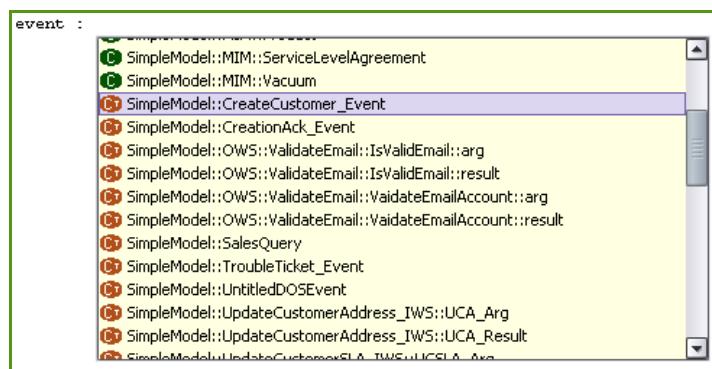


Figure A-19: Model assist list of Java classes

Transient classes

Transient classes are listed with the transient class icon  .



Bound variables

The bound variables associated with the artifact in focus appear in a model assist list. Each appears with a bound variable icon  .

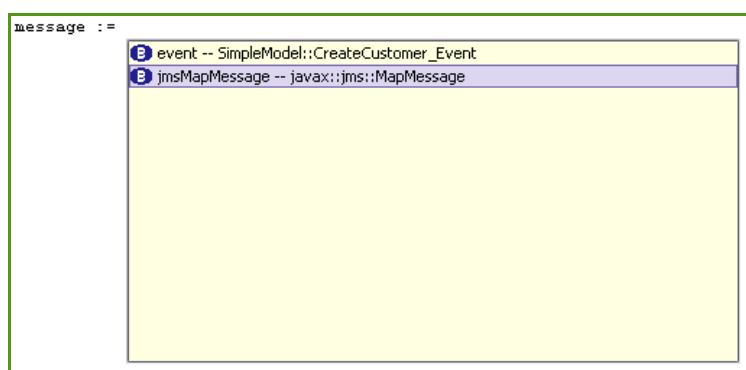


Figure A-20: Model assist list of bound variables

Java actions

When you type the `java` keyword then launch model assist, it displays a list of Java actions. Each appears next to the Java action icon  on the list.



Figure A-21: Model assist list of Java actions

Keywords

Several keywords are also available within the context of a class variable. When you type a defined variable and bring up the model assist window, the keywords available appear next to their keyword icons  in the list.

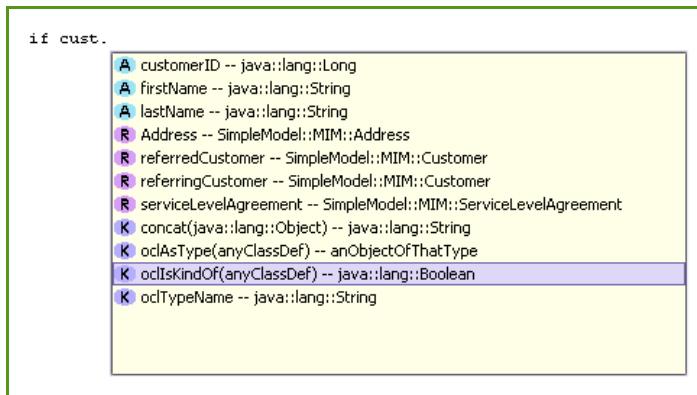


Figure A-22: Model assist list of keywords

LIM

When you enter the `executeSQL` or `selectSQL` keyword, and then launch model assist, it displays a list of LIM objects. This includes the LIM object itself, identified by the LIM icon  and SQL statements, identified by the SQL icon .

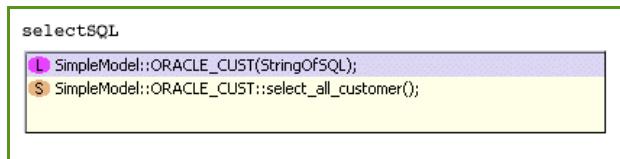


Figure A-23: Model assist list of LIM objects

Methods

Java methods are available through model assist within the context of a Java variable. The methods appear next to their icons  in the list. In the picture below the list has been filtered for those methods starting with the letter 's'.

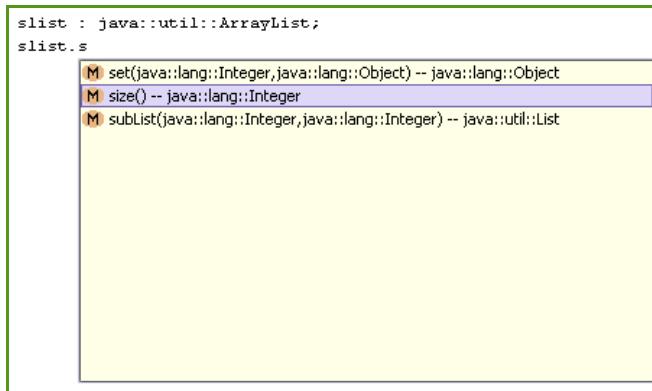


Figure A-24: Model assist list of methods

Operations

You can display a list of operations for a class when you bring up model assist within the context of a class variable that has operations. They appear in the list next to the operation icon .

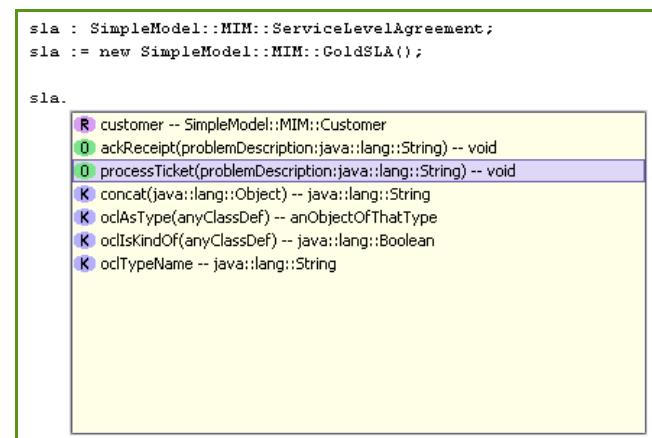


Figure A-25: Model assist list of operations

Parameters

Parameters for operations defined for model classes are available when you bring up model assist within the context of the operation. They are listed next to the parameter icons  .



A screenshot of a code editor showing a Java snippet. The code defines a variable 'sla' of type 'SimpleModel::MIM::ServiceLevelAgreement' and initializes it to a new 'GoldSLA()' object. It then calls the 'processTicket()' method on 'sla'. A callout box highlights the parameter 'problemDescription' with a red border and a small red icon next to it. The code is as follows:

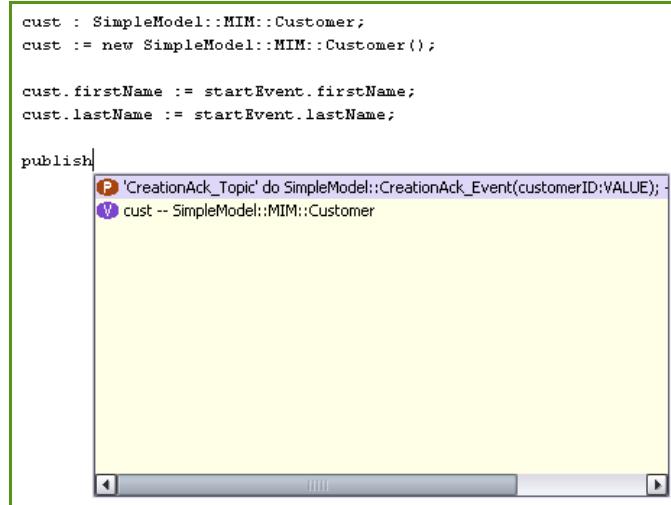
```
sla : SimpleModel::MIM::ServiceLevelAgreement;
sla := new SimpleModel::MIM::GoldSLA();

sla.processTicket()
    P problemDescription -- java::lang::String
```

Figure A-26: Model assist list of parameters

Publish

When you want to publish a message on a destination, model assist displays a list of possible target subjects alongside the publish icon  .



A screenshot of a code editor showing a Java snippet. The code creates a 'Customer' object named 'cust' from 'SimpleModel::MIM::Customer' and initializes its 'firstName' and 'lastName' fields with values from 'startEvent'. It then uses the 'publish' keyword to send a 'CreationAck_Event' message to a topic named 'CreationAck_Topic'. The message includes the 'customerID' field with a value of 'cust'. A callout box highlights the target subject 'cust' with a red border and a small red icon next to it. The code is as follows:

```
cust : SimpleModel::MIM::Customer;
cust := new SimpleModel::MIM::Customer();

cust.firstName := startEvent.firstName;
cust.lastName := startEvent.lastName;

publish
    P 'CreationAck_Topic' do SimpleModel::CreationAck_Event(customerID:VALUE);
    V cust -- SimpleModel::MIM::Customer
```

Figure A-27: Model assist list of message target subjects

When you want to publish an internal event to another Cogility object, model assist displays a list of possible target subjects alongside the publish icon .

```
self.setCardStatus(cardId: startEvent.cardId, statusVal: 'FOOTBALL CHANNELS ON');
CARD : ABC::MIM::SmartCard := locate ABC::MIM::SmartCard(id: startEvent.cardId);
ORD : ABC::MIM::Order := CARD.getOrder();
publish ORD do
    ABC::DataTransformation::HREvent(dateOfBirth: VALUE, dateOfBirthLong: VALUE)
    ABC::OrderProcessing::OrderBilledEvent();
    ABC::OrderProcessing::PartsShippedEvent();
    ABC::OrderProcessing::ServiceProvisionedEvent();
    ABC::OrderProcessing::StartOrderEvent(serviceType: VALUE); -- VALUE(s) to be
    CARD -- ABC::MIM::SmartCard
    ORD -- ABC::MIM::Order
    boPoid -- java::lang::String
    parent -- ABC::MIM::PremiumService::provision::_provision_Top
    self -- ABC::MIM::PremiumService
    smPoid -- java::lang::String
    startEvent -- ABC::OrderProcessing::ProvisionServiceEvent
    state -- ABC::MIM::PremiumService::provision::FootballChannelsOn
```

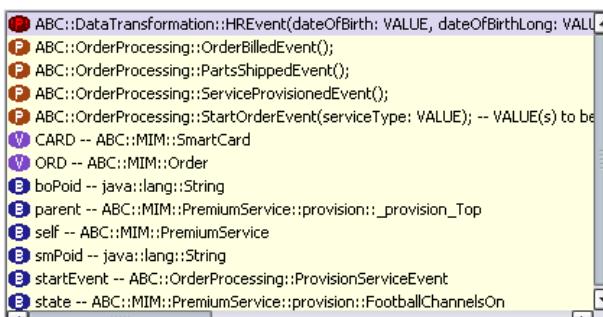


Figure A-28: Model assist list of internal event target subjects

Roles

When you want to create an association using the \$ operator or the new to keywords, model assist presents a list of possible roles that complete the association. These appear next to the role icon  in the list.

```
cust : SimpleModel::MIM::Customer;
cust := locate SimpleModel::MIM::Customer(lastName: 'Smelch');

ord : SimpleModel::MIM::Order;
ord := new SimpleModel::MIM::Order();

new cust to
    address -- SimpleModel::MIM::Address
    order -- SimpleModel::MIM::Order
    serviceLevelAgreement -- SimpleModel::MIM::ServiceLevelAgreement
```

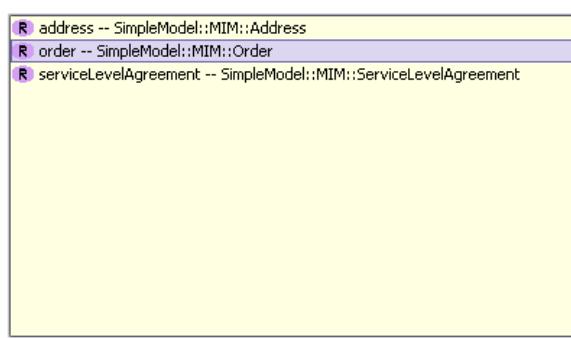


Figure A-29: Model assist list of roles

Variables

Model assist can display a list of variables defined for the workspace. These are listed with their variable icons  .

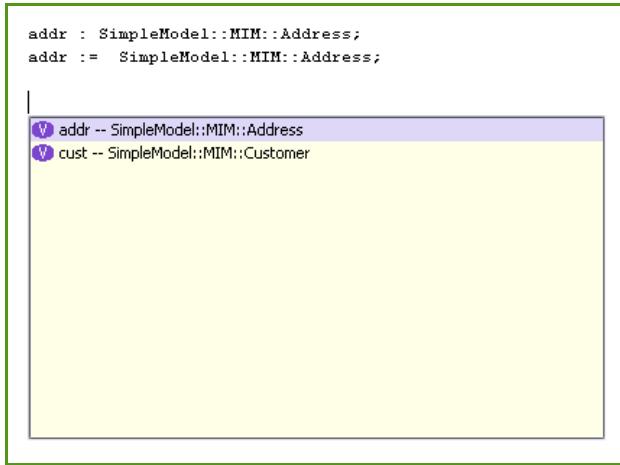
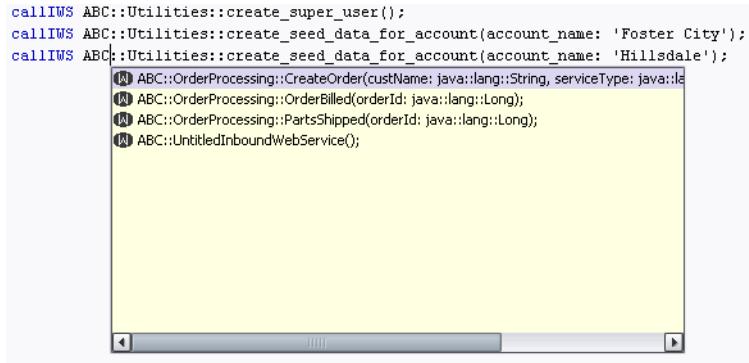


Figure A-30: Model assist list of variables

Web service

Model assist can display a list of webservices. These are listed with webservice icon  . Both inbound and outbound webservices use the same icon.



XSD element definition

Model assist can display a list of XSD element definitions. These are listed with the attribute icon  .

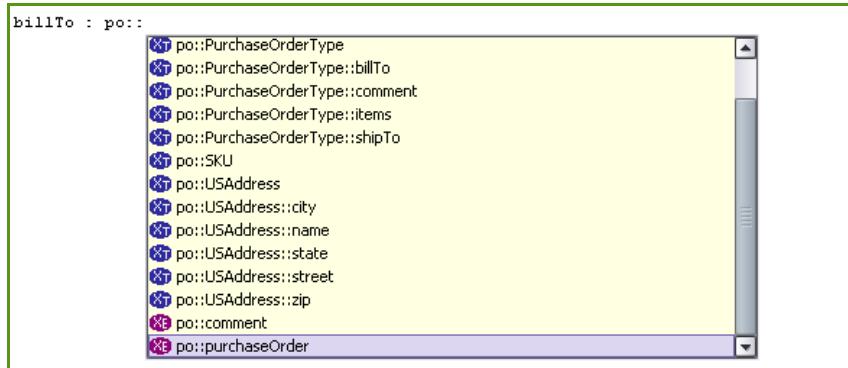


Figure A-31: Model assist list of XSD element definitions

XSD type definition

Model assist can display a list of XSD type definitions. These are listed with the XSD type icon  .

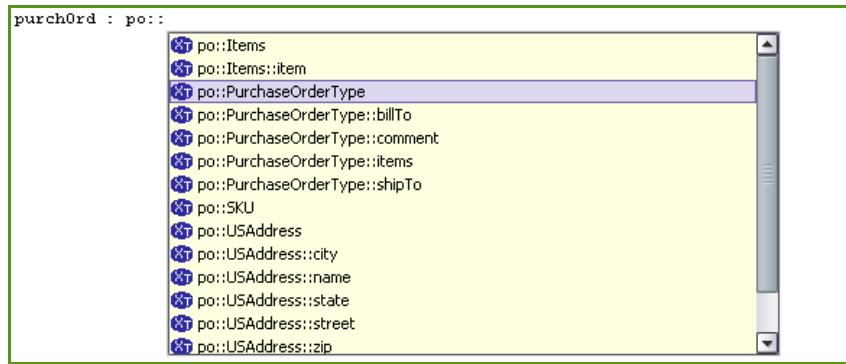


Figure A-32: Model assist list of XSD type definitions

Keystrokes and mouse click responses

When the model assist window opens, it can react to continued keystrokes and mouse clicks. Following is a list of the keystrokes and mouse click responses:

- Continue typing

If you continue typing, the list of options is reduced by filtering it with the additional characters that you've entered. If you enter a character that causes the filter to remove the last displayed item, the window closes. For example, if you look at [Figure A-32](#), if you enter the letter P, only those items following po:: that begin with a P will appear in the list.

- <Backspace>

Use of the backspace key may increase the list of options because more items may pass the filtering has been reduced by its right-most character. If you backspace beyond the point where code assist was going to insert the snippet, the window closes.

- <Up Arrow> and <Down Arrow>

The <Up Arrow> and <Down Arrow> keys move the current selection up and down the list, respectively.

- <Enter>
The <Enter> key inserts the current selection into your action semantics.
- <Double-Click>
Double-Clicking on an item inserts that selection into your action semantics.
- <Esc>
The <Esc> key closes the window.

Note that model assist does not have a smart scanner. You can easily fool it by bringing it up in the middle of a string constant or inside of a comment statement.

Model assist configuration parameters

Some configuration parameters exist for the code assist window. The defaults are defined in %DCHOME%\Config\Files\OCLCodeAssist.txt. You may override some of these in your \Config\customConfigurations.txt file. See “[Customizing installation configuration](#)” on page 13 of the guide, *Model Deployment & Execution in Cogility Studio*.

Window size

The size of the code assist window can be changed with the following configuration parameters:

```
com.cogility.OCLCodeAssist.windowWidth=400  
com.cogility.OCLCodeAssist.windowHeight=225
```

Custom Java actions

You can add custom Java actions to the list of predefined Java actions. A configuration parameter defines the number of custom Java actions, and other configuration parameters define the full class specification for each custom Java action. These configuration parameters are numbered and zero-based as follows:

```
com.cogility.OCLCodeAssist.JavaActions.custom.count=2  
com.cogility.OCLCodeAssist.JavaActions.custom.0=com.custom.oneJavaAction  
com.cogility.OCLCodeAssist.JavaActions.custom.1=com.custom.twoJavaAction
```

Also, in each Java action, you should define two methods that provide information that is combined with the full class specification to form an entry in the code assist window list. The ArgumentList is the string that appears between the parenthesis and the ReturnType is the string that appears after the double hyphen. The two methods that should be defined are:

```
public static String oclCodeAssist_ArgumentList()  
public static String oclCodeAssist_ReturnType()
```

Classes

You can add class specifications to the Class/AssociationClass list. There are too many Java classes to add them all, therefore, the list includes a subset of classes that are expected to be used frequently. If you have classes that you use frequently, you may add them to your list. A configuration parameter defines the number of additional classes, and, other configuration parameters define the full specification for each added class. These configuration parameters are numbered and zero-based as follows:

```

com.cogility.OCLCodeAssist.ClassDefs.count= 2
com.cogility.OCLCodeAssist.ClassDefs.0=java.lang.Short
com.cogility.OCLCodeAssist.ClassDefs.1=oracle.sql.SqlName

```

Using model assist

To use model assist:

1. In an action semantics listing, place the cursor where you need to complete a statement.
2. Right-click and select **Model Assist at Cursor**.
You can also hold down the Ctrl key and press the space bar to open the model assist window.
3. Select the appropriate element from the list and double-click to insert it into the listing.

Syntax assist

Syntax assist provides you with instant access to action semantics statement syntax. The syntax assist window displays a list of action semantics expressions that may be inserted into the action semantics listing. The syntax assist provides a template for the syntax for the chosen statement.

In the example below, opening syntax assist on a new line displays the syntax assist window with the available action semantics statements. To complete the line with the appropriate entries, you can place the cursor in the entry you wish to replace and use model assist (see “[Model assist](#)” on page 300) to select the appropriate item from the list and double-click to insert it in the line.

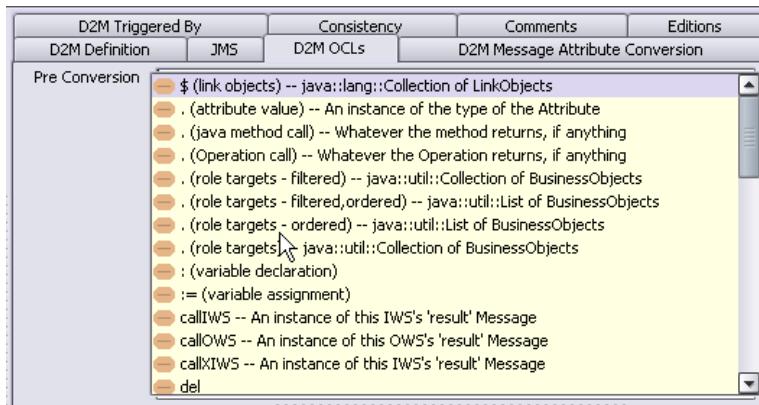


Figure A-33: Syntax assist

Note: Enter the keyword in the action semantics field and bring up syntax assist only the syntax for that keyword appears in the assist list. In the example below, the keyword **del** was selected. The keyword assist entered the syntax for that keyword. Model assist can be used to complete the statement.

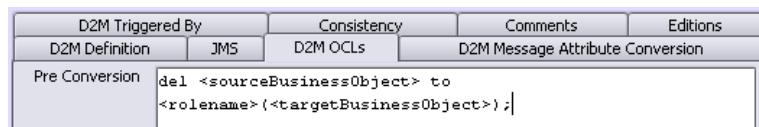
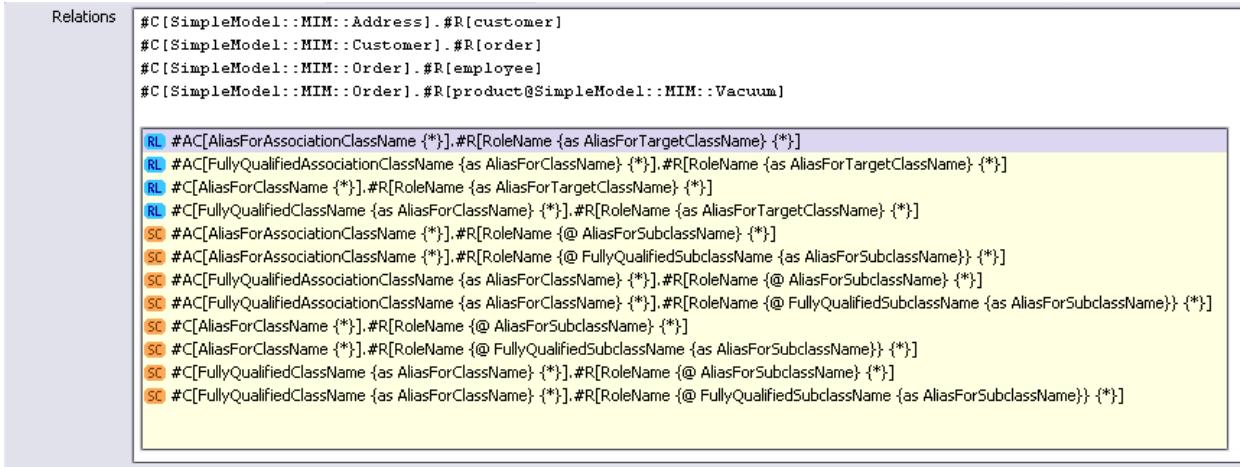


Figure A-34: Keyword assist inserts syntax

Note: When using syntax assist with custom queries, the curly brackets denote optional fields. These are shown in the figure below. See “[Custom queries](#)” on page 101 of the guide, *Using Actions in Cogility Studio*.



The screenshot shows the Cogility Studio interface with the 'Relations' section selected. The code editor displays several lines of UML-like syntax, including class names like 'SimpleModel::MIM::Address', 'Customer', 'Order', 'employee', and 'product'. A large number of syntax assist suggestions are overlaid on the code, appearing in a light yellow background. These suggestions include various role names and association class names, many of which are preceded by '#C[AliasForAssociationClassName {*}]' or '#AC[FullyQualifiedAssociationClassName {as AliasForClassName} {*}]' followed by curly braces. Some suggestions are enclosed in orange boxes.

```
#C[SimpleModel::MIM::Address].#R[customer]
#C[SimpleModel::MIM::Customer].#R[order]
#C[SimpleModel::MIM::Order].#R[employee]
#C[SimpleModel::MIM::Order].#R[product@SimpleModel::MIM::Vacuum]

RL #AC[AliasForAssociationClassName {*}].#R[RoleName {as AliasForTargetClassName} {*}]
RL #AC[FullyQualifiedAssociationClassName {as AliasForClassName} {*}].#R[RoleName {as AliasForTargetClassName} {*}]
RL #C[AliasForClassName {*}],#R[RoleName {as AliasForTargetClassName} {*}]
RL #C[FullyQualifiedClassName {as AliasForClassName} {*}],#R[RoleName {as AliasForTargetClassName} {*}]
SC #AC[AliasForAssociationClassName {*}].#R[RoleName {@ AliasForSubclassName} {*}]
SC #AC[FullyQualifiedAssociationClassName {as AliasForClassName} {*}].#R[RoleName {@ FullyQualifiedSubclassName {as AliasForSubclassName} {*}}]
SC #AC[FullyQualifiedAssociationClassName {as AliasForClassName} {*}].#R[RoleName {@ AliasForSubclassName} {*}]
SC #C[AliasForClassName {*}],#R[RoleName {@ AliasForSubclassName} {*}]
SC #C[FullyQualifiedClassName {as AliasForClassName} {*}].#R[RoleName {@ AliasForSubclassName} {*}]
SC #C[FullyQualifiedClassName {as AliasForClassName} {*}].#R[RoleName {@ FullyQualifiedSubclassName {as AliasForSubclassName} {*}}]
SC #C[FullyQualifiedClassName {as AliasForClassName} {*}].#R[RoleName {@ FullyQualifiedSubclassName {as AliasForSubclassName} {*}}]
```

Using syntax assist

To use syntax assist:

1. In an action semantics listing, place the cursor where you need to know the syntax.
2. Right-click and select **Keyword Assist at Cursor**.
You can also hold down the Shift and Ctrl keys and press the space bar to open the syntax assist window.
3. Select the appropriate syntax from the list and double-click to insert it into the listing.

Find / Replace

This section describes the functionality of the Find/Replace window that is opened from the right button menu or <Ctrl-F> in the action semantics field. This feature finds strings within text fields. To search the entire model for artifacts containing specific strings, use the Search feature. See “[Search](#)” on page 281.



Figure A-35: The Find/Replace dialog

Find String

The Find String text field allows you to enter a string that can be searched for in the action semantics field. You can also select the drop-down arrow to select a previously specified search string. The last ten search strings specified are shown in the drop-down list.

Replace With

The Replace With text field allows you to enter a string that can be substituted for other strings in the action semantics field. The other string is usually the text from the Find String field, but the Replace With value can also replace text that is simply highlighted in the action semantics field.

Find Next

The Find Next button searches the action semantics field for the next occurrence of the text in the Find String field. The search begins from the end of any highlighted text, or, from the cursor if no text is highlighted.

The Find Next button is the default button for this window. This means that pressing the <Enter> key is equivalent to clicking the Find Next button.

Find From Start

The Find From Start button searches the action semantics field, from the beginning, for the first occurrence of the text specified in the Find String field.

Replace

The Replace button replaces the highlighted text in the action semantics field with the text from the Replace With field. When completed, the replacement text is highlighted in the action semantics field.

Replace/Find

The Replace/Find button does a single Replace action followed by a single Find Next action.

Find/Replace All

The Find/Replace All button searches the entire action semantics field for all occurrences of the text specified in the Find String field and replaces each occurrence with the text from the Replace With field.

Find/Replace To End

The Find/Replace To End button searches the rest of the action semantics field (from the cursor to the end) for all occurrences of the text in the "Find String" field and replaces each occurrence with the text from the Replace With field.

Forward and Backward

The Forward and Backward radio buttons control the direction in which searches are performed. If you toggle the direction to Backward, three buttons are renamed to reflect their new functionality:

- Find Next becomes Find Prev
- Find From Start becomes From End
- Replace/Find To End becomes Replace/Find To Start.

This setting is not remembered for subsequent Find/Replace windows. Forward is selected each time a new Find/Replace window opens.

Ignore Case

The Ignore Case checkbox controls whether searches are case sensitive. This setting affects the text from the "Find String" field only. It does not affect the text from the "Replace With" field.

This setting is remembered for subsequent Find/Replace windows. A new Find/Replace window opens with the same setting as the previous Find/Replace window.

Process Backslashes As Escapes

The Process Backslashes As Escapes checkbox controls whether the backslash character is to be processed as an "escape" character. If enabled, the text from both the Find String and Replace With fields is processed with the appropriate characters being substituted for escaped characters. Then, Find and Replace operations are performed with the resulting text.

This setting is remembered for subsequent Find/Replace windows. A new Find/Replace window opens with the same setting as the previous Find/Replace window.

Escape Characters

A character is escaped if it's preceded by a backslash. It provides you with a way to enter certain characters that you could not normally enter via the keyboard. The most common escaped characters are: carriage return, linefeed and tab, though others do exist.

See the Java documentation for a complete list of the escaped characters that are supported in the Find String and Replace With fields. The following section contains a partial list.

Escape Sequences for Character and String Literals

The character and string escape sequences allow for the representation of some nongraphic characters as well as the single quote, double quote, and backslash characters in character literals and string literals.

\b	\u0008: backspace BS
\t	\u0009: horizontal tab HT
\n	\u000a: line feed LF
\f	\u000c: form feed FF
\r	\u000d: carriage return CR
\"	\u0022: double quote "
'	\u0027: single quote '
\\	\u005c: backslash \
OctalEscape	\u0000 to \u00ff: from octal value
OctalEscape:	\ OctalDigit \ OctalDigit OctalDigit \ ZeroToThree OctalDigit OctalDigit

Note: It is a compile-time error if the character following a backslash in an escape is not an ASCII b, t, n, f, r, ", ', \, 0, 1, 2, 3, 4, 5, 6, or 7.

Using Find/Replace

To use Find/Replace:

1. In any action semantics field, right-click and select **Find/Replace**.
You can also hold down the Ctrl key and press the F key.
2. In the **Find String** field, enter a string and choose from the options described in “[Find / Replace](#)” on page 314.

History comments

As several authors make changes to the model’s various action semantics listings, it is helpful, even essential, that these changes be traceable. Whenever an action semantics listing is modified, you need to know who made the change, when and why. Usually, the the comment also includes a tracking reference such as a change request number. A typical history comment might look like the following:

-- Dec 20 2005, CR10357, Joe Modeler. Added test for value less than zero.

The example above shows a history comment in the format defined for Cogility Studio. The first field is the date, taken from the system clock, the second field is a tracking reference, the third is the name of the user making the change and the fourth is a description.

Often, the greatest obstacle to tracking changes is getting authors to include comments like the above. Both Cogility Action Pad and Cogility Modeler have facilities that automate history comments and make them easier to include. These facilities can be set to automatically insert the comment date and username, and prompt the user to provide a tracking reference and a description. See “[History comment configuration parameters](#)” on page 317.

At any time during the session you can change the username and tracking reference. See “[Setting the history comment user name](#)” on page 319 and “[Setting the history comment tracking reference](#)” on page 319.

When automatic history comments are disabled, you can use the menu selections to insert history comments manually. See “[Adding history comments](#)” on page 319. You can also remove a comment defined for the tracking reference of the current session. See “[Removing history comments](#)” on page 320.

History comment configuration parameters

Four configuration parameters control the history comment settings; you include these with your custom configurations as described in “[Customizing installation configuration](#)” on page 13 of the guide, *Model Deployment & Execution in Cogility Studio*. The parameters (shown with their default values) are as follows:

```
com.cogility.OCL.historyComments.enabled=true
```

When history comments are enabled, the various menu selections for adding and removing comments, as well as setting the username and tracking reference will appear. Setting this parameter to `false` hides these menu selections and effectively disables the feature.

```
com.cogility.OCL.historyComments.automatic=true
```



When history comments are set to automatic, each time the user modifies existing action semantics, a comment line appears at the top of the listing and the user must change the default description, "Description of change goes here." If the user does not replace the default description, an inconsistency warning appears for the artifact.

When the user enters text into the field for the first time, either as a comment or executable action semantics, the history comment line contains the default description, "Created." This default description does not have to be changed, and no inconsistency will result if left as is.

Also, for each new Cogility Modeler session (each time the user logs in), the first time he makes a change to an action semantics field, the user is prompted to enter a tracking reference. That tracking reference will appear in the comment line immediately after the date for all subsequent editions during the session.

When this parameter is set to `false`, neither the tracking reference prompt nor the comment line appears automatically. However, the user may use the menu selections to insert a comment. The first time he inserts a comment, he will be prompted to provide a tracking reference; the comment line will contain a default description and if that description must be changed, the artifact will be inconsistent as when the comment line appears automatically.

`com.cogility.OCL.historyComments.username=UnspecifiedUsername`

When this parameter is set to `UnspecifiedUsername`, if automatic history comments are enabled (see above), the first time the user edits an action semantics field he will be prompted to enter a username. That username will be inserted in all subsequent comments during the session. For each new Cogility Modeler session, the user will be prompted as described here. If automatic history comments are disabled, the prompt for the username appears only when the user inserts a history comment using the menus, and that user name is also applied to all history comments for the duration of the session. If the username is specified, no prompts appear. The string for the username may not include commas. The user can always change the username for history comments with the menu selections. See "[Setting the history comment user name](#)" on page 319.

Once a history comment has been added to an action semantics listing, the user name associated with the tracking reference for that comment cannot be changed. For example, user A has made changes to an action semantics listing and identified those changes with a history comment that specifies tracking reference 001. If user B logs in, so that `username=B`, and then makes a change to the same listing using the same tracking reference 001, no new history comment will be added. The following warning will appear:

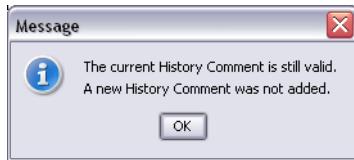


Figure A-36: History comment not added

If, however, user B makes a change to a field that does not have a history comment for tracking reference 001 defined, user B will be identified in the history comment for that field.

`com.cogility.OCL.historyComments.problemIdTitle=Change Request Number`

Different organizations may use different kinds of systems to track changes; consequently, the label that describes the tracking reference to that system may be different. One may use a "Change Request Number" while another uses a "Bug Report ID." You can change the label for the tracking reference according to your requirements. The string you enter for the value here will appear in the menu you select to set the tracking reference. By default, the string is Change Request Number and the menu item is Selection > Actions > Set Change Request Number in Cogility Modeler, and Actions

> Set Change Request Number in Cogility Action Pad. See “[Setting the history comment tracking reference](#)” on page 319.

Setting the history comment user name

To set the history comment user name:

1. Enable history comments.

See “[History comment configuration parameters](#)” on page 317.

2. Select from the following:

- a. In Cogility Action Pad, from the **Actions** menu, choose **Set History Comment User Name**.
- b. In Cogility Modeler, from the **Selection** menu, choose **Actions > Set History Comment User Name**.

3. In the dialog, enter a user name and click **OK**.

The string may include spaces, but not commas.

Setting the history comment tracking reference

To set the history comment tracking reference:

1. Enable history comments and specify the tracking reference label.

See “[History comment configuration parameters](#)” on page 317. The tracking reference label you set will appear in place of the <Tracking Reference> cited below.

2. Select from the following:

- a. In Cogility Action Pad, from the **Actions** menu, choose **Set <Tracking Reference>**.
- b. In Cogility Modeler, from the **Selection** menu, choose **Actions > Set <Tracking Reference>**.

3. In the dialog, enter tracking reference string and click **OK**.

The tracking reference may be any string not including commas.

Adding history comments

To add history comments manually:

1. In an action semantics field, right-click and select **Add History Comment**.

If the tracking reference has not been defined, a dialog will prompt you for it. You may want to set a new tracking reference. See “[Setting the history comment tracking reference](#)” on page 319.

2. In the dialog, enter the tracking reference string and click **OK**.

The tracking reference may be any string not including commas. Once you have submitted a tracking reference, a history comment is prepended to the top of the action semantics listing in the current field. If you have not yet made an edition to the field and the field has no text, the description of the comment reads “Created.” You may select and change this description, but it is not required. If your edition is a change to existing action semantics or comments, the description reads, “Description of change goes here.” and this text is selected. You must change this text in order to clear the inconsistency on the artifact.

3. Enter a description for the comment, replacing the selected text.

If you move the cursor out of the field before changing the text, then return to the field, the default description will not be selected and the inconsistency warning will appear in the Consistency view. You must reselect the default description and change it to clear the inconsistency.

Removing history comments

Only history comments defined for the current tracking reference may be removed in this fashion.

To remove a history comment:

1. In an action semantics field, right-click and select **Remove History Comment**.

If you want to remove history comments for a different tracking reference, you must first set that tracking reference. See “[Setting the history comment tracking reference](#)” on page 319.

Java package alias

In any action semantics field, you can use a Java package alias to abbreviate a fully-qualified Java classname. This is helpful when working with Java actions or any Java code in your action semantics. The Java package alias provides a shortcut to package containing the class, though not the class itself. You must know the classname of the class in the package and specify it after the package alias.

The Java package must be in the active project’s classpath. See “[Project Configuration Editor](#)” on page 51 of the guide, *Model Deployment & Execution in Cogility Studio* for information about adding classes to the classpath. Once established in the classpath and associated with a Java package alias in the model, the package may be referenced by its short name (alias) in the model’s action semantics. Also, if a variable is defined using a Java package alias, model assist will recognize that data type and provide appropriate responses, for example, after something like “`date`,” as illustrated in the diagram below. Notice that the class must be declared and a new object created before the variable for that class may be used with code assist to find the members of the class.

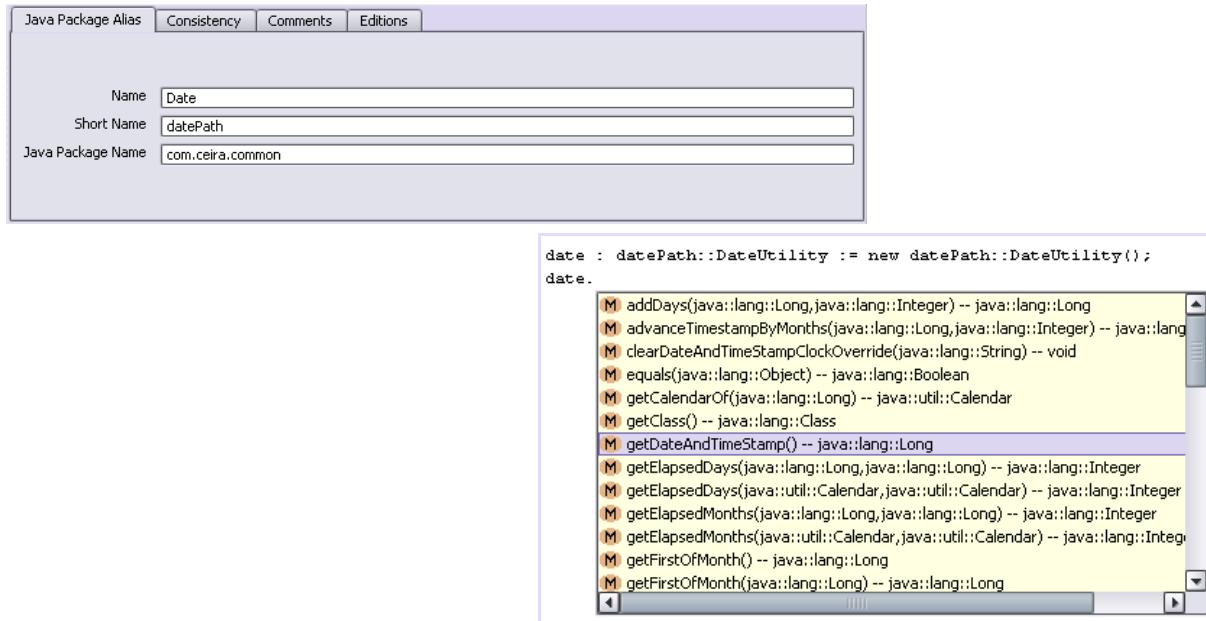


Figure A-37: Model assist for a Java package alias

Defining a Java package alias

To define a Java package alias:

1. In the tree view, select the model container or package.

2. Click the plus button and select **Add Java Package Alias**.
3. In the editor view, enter the following information.
 - a. In the **Name** field, enter a name for the artifact.
Note that this is not necessarily the same as the actual alias used in an action semantics field. It may be different, and is used only to identify the artifact.
 - b. In the **Short Name** field, enter the alias you will use in your action semantics.
This is the string that will serve as an abbreviation of the fully-qualified Java package.
 - c. In the **Long Name** field, enter the fully-qualified Java package.
Use dots (not double colons) for the namespace qualifiers.

System view diagrams

Like UML data flow diagrams, system view diagrams describe how data and control move through your system model and between the various model artifacts. System view diagrams describe the inputs to, outputs from and objects associated with model artifacts. They also show the sequencing of artifacts where control passes from one state to another or one artifact to another like an event to a state machine. System view diagrams are not executable. Below is illustrated the system view of the customer creation process from the SimpleModel. Control, inputs and outputs are illustrated with black arrows. Associated objects are described with purple lines. The dotted blue arrows illustrate derived outputs, those produced by the artifact's action semantics. These are the default display settings and may be changed. See “[System view diagram elements](#)” on page 323.

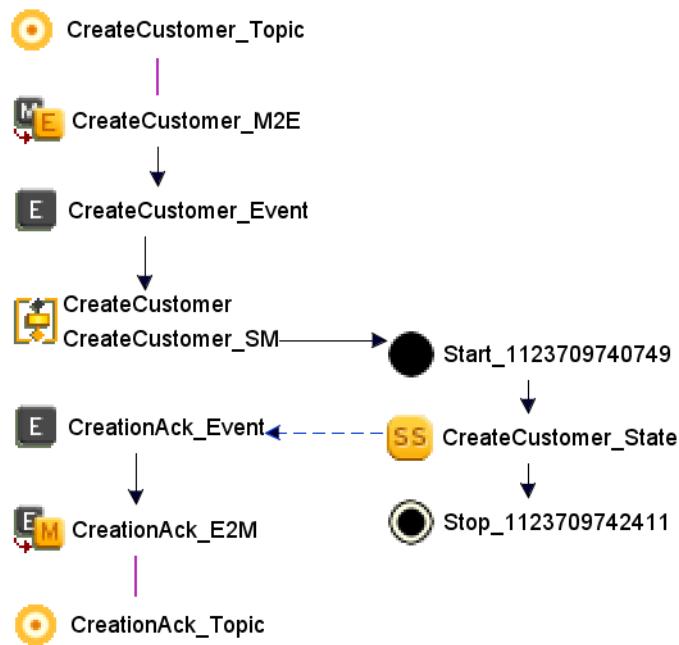


Figure A-38: System view diagram

Creating a system view diagram

To create a system view diagram:

1. In Cogility Modeler’s tree view, select the MIM or DIM within which you want to create the diagram.

If your model does not yet have a MIM or a DIM, see “Master information model (MIM)” on page 29 or “Distinct information model (DIM)” on page 30 to add these artifacts.

2. Click the **Add System View Diagram** button .

You can also choose, from the Selection menu, Domain Modeling > Add System View Diagram. The system view diagram editor then appears in the content view.

3. In the diagram editor, in the **Name** field, enter the name for the new system view diagram.

The name you enter must be unique within the MIM or DIM.

4. Add model inputs or outputs.

See “Model inputs and outputs” on page 322

5. Add existing elements to the diagram.

See “System view diagram elements” on page 323.

6. Display the connections to the elements.

See any of the following:

- “Displaying inputs to an element” on page 324.
- “Displaying outputs from an element” on page 324.
- “Displaying derived outputs from an element” on page 324.
- “Displaying elements associated to an element” on page 324.

7. Adjust the display settings, as needed.

See “Display settings” on page 327.

Displaying a system view diagram

To display a system view diagram:

1. In Cogility Modeler’s tree view, click the plus signs (+) to expand the **Domain Model**, the **MIM** or the **DIM**, and **System Diagrams**.

2. Select the class diagram you want to display.

Model inputs and outputs

Model inputs are those elements that can accept input from external systems such as IWS deployments and JMS topics. Once you have these elements in your diagram, you can connect the outputs to them, developing the diagram from start to finish. Model outputs allow you to start from outputs to external systems, such as topics and OWS definitions, and work backward.

Adding model inputs

To add model inputs to a system view diagram:

1. Create a system view diagram or open an existing one.

See “System view diagrams” on page 321.

2. From the **Edit** menu, select **Add model inputs**.

3. From the **Select Object(s)** dialog, select the artifact(s) and click **OK**.

See “Select Object(s) dialog” on page 297.

4. Display the outputs from the element.

See “Displaying outputs from an element” on page 324.

Adding model outputs

To add model outputs to a system view diagram:

1. Create a system view diagram or open an existing one.
See “[System view diagrams](#)” on page 321.
2. From the **Edit** menu, select **Add model outputs**.
3. From the **Select Object(s)** dialog, select the artifact(s) and click **OK**.
See “[Select Object\(s\) dialog](#)” on page 297.
4. Display the outputs from the element.
See “[Displaying outputs from an element](#)” on page 324.

System view diagram elements

As the elements of a system view diagram are related, they may be inputs to, outputs from, derived outputs from, or associated with other elements. Element inputs are those upon which the element depends for data or control; for example, an event depends upon an M2E. Likewise, element outputs are those that depend upon data or control from the element; the E2M is an output of the event. Both inputs and outputs are explicitly associated with the element in the model, and the modeling rules require the associated elements to avoid an inconsistency.

Derived outputs, however, are not explicitly associated with the element, they are specified in the element’s action semantics. For example, a `CreateCustomer` state’s do activity actions publish a `CreationAck_Event`, as illustrated in [Figure A-38 on page 321](#).

Associated elements are those that are related to the element but have no control flow implications (as in input and output). The JMS topics for which an M2E listens are associated elements, for example.

Adding elements to a system view diagram

To add elements to a system view diagram:

1. Create a system view diagram or open an existing one.
See “[System view diagrams](#)” on page 321.
2. In the tool bar, click any of the **Add Existing Element** buttons.
3. Move the cursor into the diagram where you want to locate the element and click again.
4. From the **Select Object(s)** dialog, select the artifact(s) and click **OK**.
See “[Select Object\(s\) dialog](#)” on page 297.
5. Edit the element, if necessary.
See “[Editing elements](#)” on page 325.
6. Display the inputs to and outputs from the element.
See any of the following:
 - “[Displaying inputs to an element](#)” on page 324.
 - “[Displaying outputs from an element](#)” on page 324.
 - “[Displaying derived outputs from an element](#)” on page 324.
 - “[Displaying elements associated to an element](#)” on page 324.

Displaying inputs to an element

To display the inputs to an element:

1. In the system view diagram, select the element.
2. Right-click and select **Inputs**.
3. From the **Select Object(s)** dialog, select the artifact(s) and click **OK**.
See “[Select Object\(s\) dialog](#)” on page 297.

Displaying outputs from an element

To display the outputs from an element:

1. In the system view diagram, select the element.
2. Right-click and select **Outputs**.
3. From the **Select Object(s)** dialog, select the artifact(s) and click **OK**.
See “[Select Object\(s\) dialog](#)” on page 297.

Displaying derived outputs from an element

To display the derived outputs from an element:

1. In the system view diagram, select the element.
2. Right-click and select **Derived Outputs**.
3. From the **Select Object(s)** dialog, select the artifact(s) and click **OK**.
See “[Select Object\(s\) dialog](#)” on page 297.

Displaying elements associated to an element

To display the elements associated to an element:

1. In the system view diagram, select the element.
2. Right-click and select **Associated Objects**.
3. From the **Select Object(s)** dialog, select the artifact(s) and click **OK**.
See “[Select Object\(s\) dialog](#)” on page 297.

Selecting elements

There are three ways to select elements in the system view diagram.

To select diagram elements:

1. In the system view diagram, click on an element, hold down the Ctrl key and click on other elements to select them as well.
2. Right-click over an empty part of the diagram, continue to hold down the mouse button while moving the mouse, and the mouse draws a shaded rectangle. All elements inside or crossing the rectangle will be selected when you release the mouse button.
3. From the **Edit** menu choose one of the following:
 - **Select All**
Selects all of the elements in a diagram.
 - **Select All Nodes**
Selects all of the element icons in the diagrams, but none of the connections.

- **Select All Connections**

Selects all of the connections between elements in a diagram, but none of the element icons.

Editing elements

The elements of a system view diagram represent model artifacts that may be edited directly from within the diagram. There are two ways to edit an artifact in a diagram.

To edit elements in a diagram:

1. In the system view diagram, double-click the element icon.

2. Right-click on the element and select **Edit**.

The editor window for the selected element opens.

Expanding nodes

When you add an element to a diagram, its or node appears fully expanded. Nodes may be collapsed, as described in “[Collapsing nodes](#)” on page 325.

To expand a node:

1. In the diagram, select a collapsed node.

You can select multiple nodes by holding down the Ctrl key as you click on the icons.

2. From the **Settings** menu, choose **Expand Selected Nodes**.

Collapsing nodes

When you add an element to a diagram, its or node appears fully expanded. You can collapse the node so that only the name appears.

To collapse a node:

1. In the diagram, select the expanded node.

You can select multiple nodes by holding down the Ctrl key as you click on the icons.

2. From the **Settings** menu, choose **Collapse Selected Nodes**.

Setting the node image

To set class node image:

1. In the diagram, select the class node(s).

See “[Selecting elements](#)” on page 324.

2. From the **Settings** menu, select **Node Image**.

3. Navigate to the location of the image, select the image, and click **OK**.

Only .jpg and .gif image files can be used as node images. The image you select is repeated over the entire icon space for a wall paper effect. To remove an image, set the node color (“[Setting the node color](#)” on page 325) instead.

Setting the node color

To set class node colors:

1. In the diagram, select the class node(s).

See “[Selecting elements](#)” on page 324.

2. From the **Settings** menu, select **Node Color**.
3. Select a color and click **OK**.

You can define a custom color using the settings in the dialog.

Setting the connection line color

To set the line color:

1. In the diagram, select the connection line(s).

See “[Selecting elements](#)” on page 324.

2. From the **Settings** menu, select **Line Color**.
3. Choose a color and click **OK**.

The selected line and its accompanying text will appear in the selected color.

Displaying element shadows

The elements in a system view diagram may be shadowed to give them a three-dimensional effect.

To turn on shadows:

1. In the diagram, select the elements.
See “[Selecting elements](#)” on page 324.
2. From the **Settings** menu, choose **Shadows**.

Reverting to default settings for elements

To revert element settings to default:

1. In the diagram, select the elements.
See “[Selecting elements](#)” on page 324.
2. From the **Settings** menu, choose **Apply Default Shape Style**.

Removing elements from the diagram

You can remove an element from the system view diagram only, as described here, or delete its corresponding artifact from the model entirely through the diagram editor. See “[Removing artifacts from the model](#)” on page 326.

To remove elements from the diagram only:

1. In the diagram, select the element(s).
See “[Selecting elements](#)” on page 324.
2. From the **Edit** menu select **Remove From Diagram**.
You can also right-click on the element and select Remove from Diagram.

Removing artifacts from the model

You can remove an element from the system view diagram only, see “[Removing elements from the diagram](#)” on page 326, or delete its artifact from the model entirely through the diagram editor, as explained here.

To delete elements from the model entirely:

1. In the diagram, select the element(s).
See “[Selecting elements](#)” on page 324.
2. From the **Edit** menu select **Delete From Model**.

You can also right-click on the element and select Delete from Model.

3. In the dialog, click **Yes** to continue with object deletion.

If you delete an object accidentally, you can recover it through the User Modifications window. See “[Reverting elements](#)” on page 12 of the guide, *Change Management in Cogility Studio*.

Display settings

There are several facilities for changing the display of your system view diagram and the elements within it.

Changing the display size

The View menu lets you reduce or enlarge the diagram as it displays in Cogility Modeler.

To change the display size of the diagram:

1. In the diagram, from the **View** menu, select **Zoom In** to make the diagram larger.
2. From the **View** menu, select **Zoom Out** to make the diagram smaller.

Setting the background color

The background of your diagram may be filled a color of your choice.

To set the background color:

1. In the diagram, from the **Settings** menu, choose **Background Color**.
2. Select a color and click **OK**.

You can define a custom color using the settings in the dialog. If you click cancel in the color dialog the background reverts to the default (white).

Setting the background image

The background of your diagram may be filled with an image that is repeated for a wall paper effect.

To set the background image:

1. In the diagram, from the **Settings** menu, choose **Background Image**.
2. Navigate to the location of the image, select the image, and click **OK**.

Only .jpg and .gif image files can be used as background images. The image you select is repeated over the entire diagram space for a wall paper effect.

Resetting the background color to the default

To reset the background color to the default (white):

1. In the diagram, from the **Settings** menu, choose **Background Color**.
2. Click **Cancel**.

Displaying the grid

Like any drawing editor, Cogility Modeler's system view diagram editor features a grid that you can display and use to align objects automatically.

To display the grid:

1. In the diagram, from the **Settings** menu, choose **Show Grid**.

Turning on snap to grid

To automatically align new elements to the grid:

1. In the diagram, from the **Settings** menu, choose **Snap to Grid**.

Displaying page breaks

Where your diagram is divided into pages is described by page breaks that appear as green lines in the diagram.

To display page breaks:

1. In the diagram, from the **Settings** menu, choose **Show Page Breaks**.

Print facilities

Standard printing facilities are available for printing system view diagrams.

Defining the page setup

To define the page setup:

1. In the diagram, from the **Diagram** menu, choose **Page Setup**.
2. Select the paper size and source, set page orientation, margins, and the printer as needed.

Defining the print scale

To define the print scale:

1. In the class diagram, from the **Diagram** menu, choose **Print Scale**.
2. Choose from three diagram size scales: 0.5 times original diagram size, original size (1 times), and 2 times original size.

You can also select Scale to fit page to fit a diagram onto one page. Note that the multiples refer to the diagram size relative to the page, not the page size relative to the diagram. Changing the print scale on a diagram revises the diagram version to a timestamp.

You can preview the printed output by displaying the page breaks. See “[Displaying page breaks](#)” on page 328.

Specifying printing options

To specify printing options:

1. In the class diagram, from the **Diagram** menu, choose **Print Scale**.
2. Choose the printer or file destination, page range and number of copies.

Creating a PDF document of the diagram

To create a PDF document of the diagram:

1. From the **Diagram** menu, select **Document Diagram Details**.
2. Navigate to the location to save the PDF file and click **Save PDF**.

Class Diagrams

Class diagrams allow you to create a visual display of the classes and relationships between classes in your model. This section describes the user interface features of the class diagram editor and explains how you can customize the display. For instructions on creating class diagrams, refer to “[Class diagrams](#)” on page 49.

To display a class diagram:

1. In Cogility Modeler’s tree view, click the plus signs (+) to expand the **Domain Model**, the **MIM** or the **DIM**, and **Class Diagrams**.
2. Select the class diagram you want to display.

Class diagram elements

A class diagram consists of class nodes and association or hierarchy connections. The class diagram editor lets you work with diagram elements directly. You can edit, delete from the diagram or delete from the model selected elements in a given diagram.

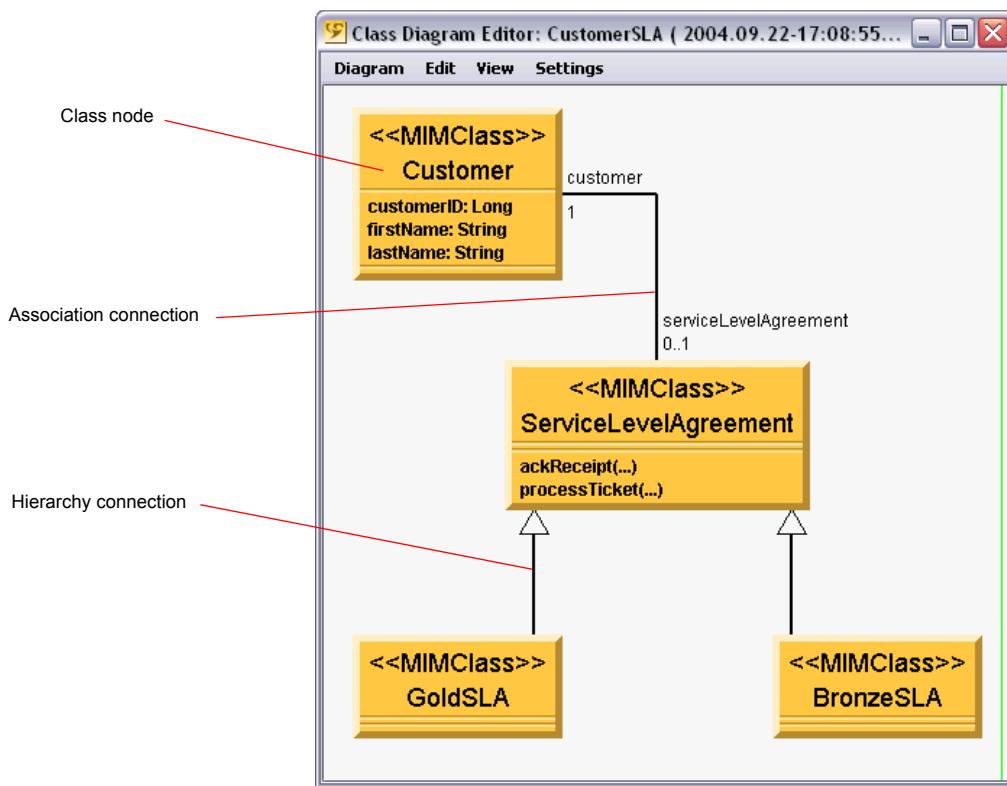


Figure A-39: Class diagram nodes and connections

Selecting elements

When you select elements, they appear darker than unselected elements. There are two ways to select elements in the class diagram.

To select diagram elements:

1. In the class diagram, click on an element, hold down the Ctrl key and click on other elements to select them as well.
2. From the **Edit** menu choose one of the following:
 - **Select All**
Selects all of the elements in a diagram.
 - **Select All Nodes**
Selects all of the class nodes in the diagrams, but none of the associations or inheritance arrows.
 - **Select All Connections**
Selects all of the associations and inheritance arrows in a diagram, but none of the class nodes.

Editing elements

The elements of a class diagram represent model artifacts that may be edited directly from within the diagram. There are two ways to edit an artifact in a diagram.

To edit elements in a diagram:

1. In the class diagram, double-click the element (association connection or class node).
2. Right-click on the element and select **Edit**.
The editor window for the selected element opens. See “[Classes](#)” on page 33, “[Superclass](#)” on page 33 or “[Associations](#)” on page 51.

Expanding nodes

When you add a class to a diagram, its or node appears fully expanded. Nodes may be collapsed, as described in “[Collapsing nodes](#)” on page 325.

To expand a node:

1. In the class diagram, select a collapsed node.
You can select multiple nodes by holding down the Ctrl key as you click on the icons.
2. From the **Settings** menu, choose **Expand Selected Nodes**.

Collapsing nodes

When you add a class to a diagram, its or node appears fully expanded. You can collapse the node so that only the class name appears, the class attributes and operations are hidden by default.

To collapse a node:

1. In the class diagram, select the expanded node.
You can select multiple nodes by holding down the Ctrl key as you click on the icons.
2. From the **Settings** menu, choose **Collapse Selected Nodes**.

Displaying connections

A class may be connected to another class in a class diagram with either a superclass definition or an association. If these connections are not displayed you can replace them with these steps. See “[Removing elements from the diagram](#)” on page 326 for information about hiding or removing elements.

To display connections in a class diagram:

1. In the class diagram, select the class icon whose connections you want to display.

2. Right-click and select one of the following from the menu:

- **Expand All Associations**

Displays associations between the selected class and all the classes in the model, whether displayed or not. If a class is connected to a non-displayed class, selecting this item adds the non-displayed class to the diagram and displays the connection.

- **Expand Associations To Others**

Displays associations between the selected class and displayed classes only. Classes not displayed in the diagram, though they may exist in the model, are not displayed, nor are the connections to those classes.

- **Expand To Superclass**

For a selected subclass, this selection displays the inheritance arrow to and the class icon for its superclass, if the superclass exists in the model. Note that this includes predefined superclasses such as the Element class from which all model objects are derived.

- **Expand To Subclass**

For a selected superclass, this selection displays the inheritance arrows from and the class icons for any subclasses to the superclass.

Setting the node image

To set class node image:

1. In the class diagram, select the class node(s).

See “[Selecting elements](#)” on page 330.

2. From the **Settings** menu, select **Node Image**.

3. Navigate to the location of the image, select the image, and click **OK**.

Only .jpg and .gif image files can be used as node images. The image you select is repeated over the entire icon space for a wall paper effect. To remove an image, set the node color (described above) instead.

Setting the node color

To set class node colors:

1. In the class diagram, select the class node(s).

See “[Selecting elements](#)” on page 330.

2. From the **Settings** menu, select **Node Color**.

3. Select a color and click **OK**.

You can define a custom color using the settings in the dialog.

Applying the shape style

To set the shape style for the node:

1. In the diagram, select the node(s).
See “Selecting elements” on page 330.
2. From the **Settings** menu, select **Apply Shape Style**.
3. In the **Selection** dialog, select the style type and click **OK**.

You can select either the Default style shown in [Figure A-40 on page 334](#), or the Plain style shown in [Figure A-41 on page 335](#).

Applying the default shape style

The default shape style for any node in a diagram is the same as the selected style shape for the entire diagram. For example, if the diagram is displayed in the Plain style, the default shape style will be the Plain style. You can select either the Default style shown in [Figure A-40 on page 334](#), or the Plain style shown in [Figure A-41 on page 335](#).

To apply the default shape style for the node:

1. In the diagram, select the node(s).
See “Selecting elements” on page 330.
2. From the **Settings** menu, select **Apply Default Shape Style**.

Setting the connection line color

To set the line color:

1. In the class diagram, select the connection line(s).
See “Selecting elements” on page 330.
2. From the **Settings** menu, select **Line Color**.
3. Choose a color and click **OK**.

The selected line and its accompanying text will appear in the selected color.

Displaying element shadows

The elements in a class diagram may be shadowed to give the diagram a three-dimensional effect.

To turn on shadows:

1. In the class diagram, select the elements.
See “Selecting elements” on page 330.
2. From the **Settings** menu, choose **Shadows**.

Reverting to default settings for elements

To revert element settings to default:

1. In the class diagram, select the elements.
See “Selecting elements” on page 330.
2. From the **Settings** menu, choose **Apply Default Shape Style**.

Removing elements from the diagram

You can remove an element from the class diagram only, as described here, or delete its corresponding artifact from the model entirely through the class diagram editor. See “[Removing artifacts from the model](#)” on page 326.

To remove elements from the diagram only:

1. In the class diagram, select the element(s).

See “[Selecting elements](#)” on page 330.

2. From the **Edit** menu select **Remove From Diagram**.

You can also right-click on the element and select Remove from Diagram.

Removing artifacts from the model

You can remove an element from the class diagram only, see “[Removing elements from the diagram](#)” on page 326, or delete its artifact from the model entirely through the class diagram editor, as explained here.

To delete elements from the model entirely:

1. In the class diagram, select the element(s).

See “[Selecting elements](#)” on page 330.

2. From the **Edit** menu select **Delete From Model**.

You can also right-click on the element and select Delete from Model.

3. In the dialog, click **Yes** to continue with object deletion.

If you delete an object accidentally, you can recover it through the User Modifications window. See “[Reverting elements](#)” on page 12 of the guide, *Change Management in Cogility Studio*.

Display settings

There are several facilities for changing the display of your class diagram and the elements within it.

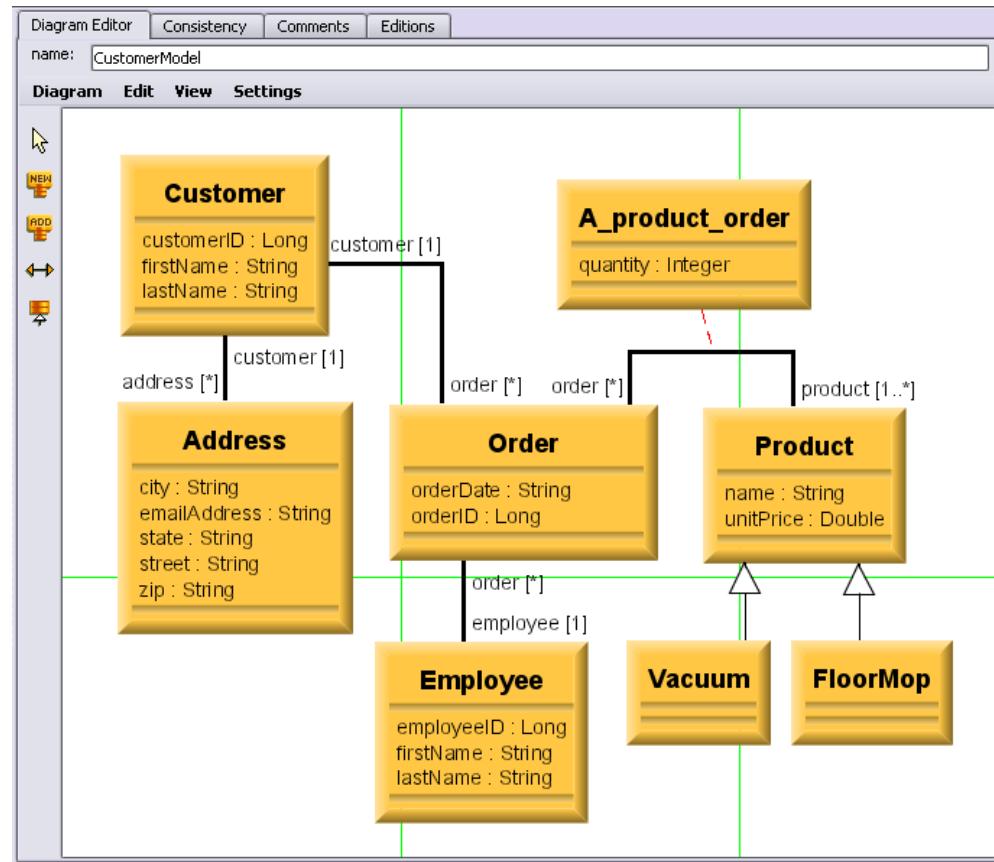


Figure A-40: Page breaks are displayed as green lines

The figure above shows a class diagram in its default style. You can also select a plain style, shown below that is more suited to black and white printing.

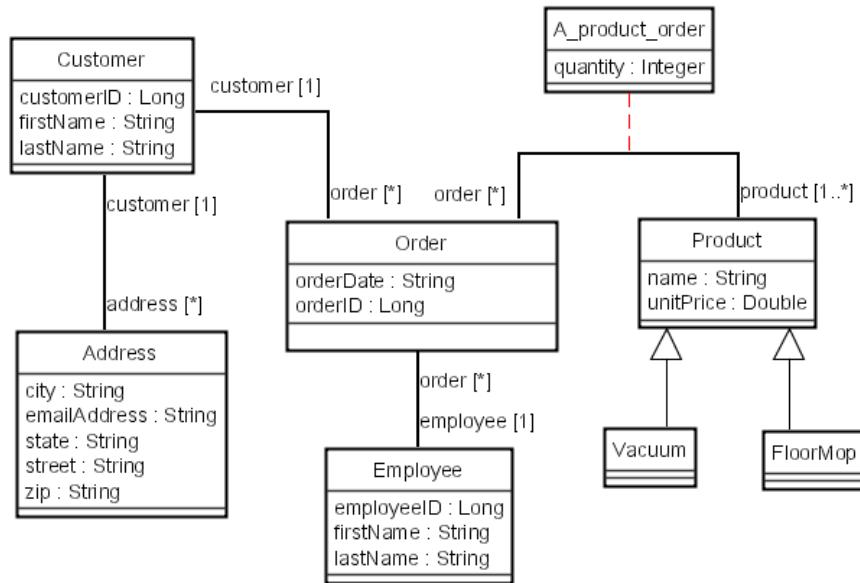


Figure A-41: Plain diagram style

Changing the diagram style

To change the diagram style:

1. In the diagram, from the **Settings** menu, select **Apply Diagram Style**.
2. In the **Selection** dialog, select the style type and click **OK**.

You can select either the Default style shown in [Figure A-40 on page 334](#), or the Plain style shown in [Figure A-41 on page 335](#).

Changing the display size

The View menu lets you reduce or enlarge the diagram as it displays in Cogility Modeler.

To change the display size of the diagram:

1. In the class diagram, from the **View** menu, select **Zoom In** to make the diagram larger.
2. From the **View** menu, select **Zoom Out** to make the diagram smaller.

Setting the background color

The background of your diagram may be filled a color of your choice.

To set the background color:

1. In the class diagram, from the **Settings** menu, choose **Background Color**.
2. Select a color and click **OK**.

You can define a custom color using the settings in the dialog. If you click cancel in the color dialog the background reverts to the default (white).

Setting the background image

The background of your diagram may be filled with an image that is repeated for a wall paper effect.

To set the background image:

1. In the class diagram, from the **Settings** menu, choose **Background Image**.
2. Navigate to the location of the image, select the image, and click **OK**.

Only .jpg and .gif image files can be used as background images. The image you select is repeated over the entire diagram space for a wall paper effect.

Resetting the background color to the default

To reset the background color to the default (white):

1. In the class diagram, from the **Settings** menu, choose **Background Color**.
2. Click **Cancel**.

Displaying the grid

Like any drawing editor, Cogility Modeler's class diagram editor features a grid that you can display and use to align objects automatically.

To display the grid:

1. In the class diagram, from the **Settings** menu, choose **Show Grid**.

Turning on snap to grid

To automatically align new elements to the grid:

1. In the class diagram, from the **Settings** menu, choose **Snap to Grid**.

Displaying page breaks

Where your diagram is divided into pages is described by page breaks that appear as green lines in the diagram. See [Figure A-40 on page 334](#).

To display page breaks:

1. In the class diagram, from the **Settings** menu, choose **Show Page Breaks**.

In the figure below, the diagram is set to a print scale of 2 times original diagram size (see ["In the class diagram, from the Diagram menu, choose Print Scale." on page 337](#)). The green lines show where the page breaks occur.

Print facilities

Standard printing facilities are available for printing class diagrams.

Defining the page setup

To define the page setup:

1. In the class diagram, from the **Diagram** menu, choose **Page Setup**.
2. Select the paper size and source, set page orientation, margins, and the printer as needed.

Defining the print scale

To define the print scale:

1. In the class diagram, from the **Diagram** menu, choose **Print Scale**.
2. Choose from three diagram size scales: 0.5 times original diagram size, original size (1 times), and 2 times original size.

You can also select Scale to fit page to fit a diagram onto one page. Note that the multiples refer to the diagram size relative to the page, not the page size relative to the diagram.

Changing the print scale on a diagram revises the diagram version to a timestamp.

You can preview the printed output by displaying the page breaks. See “[Displaying page breaks](#)” on page 328.

Specifying printing options

To specify printing options:

1. In the class diagram, from the **Diagram** menu, choose **Print Scale**.
2. Choose the printer or file destination, page range and number of copies.

Creating a PDF document of the diagram

To create a PDF document of the diagram:

1. From the **Diagram** menu, select **Document Diagram Details**.
2. Navigate to the location to save the PDF file and click **Save PDF**.



Modeling with Cogility Studio

Index

Symbols

%OEMHOME% 20

A

abstract class 34
abstract events 84
Actions after push 274
Actions After Startup 270
active mode 61
Add a Factory (button) 48
add a state to a state activity diagram 122
Add an Association button 53
Add an Association Class (button) 55
application server artifact 260
application server, select during login 18
association classes
 class diagram, adding with 55
 information, defining 55
 menus and buttons, adding with 55
Association Editor tab 53
associations
 class diagrams, adding with 53
 definition 51
 information, defining 53
 reflexive associations 57
 roles 51
attributes
 classes 35
 persistent object identifier (POID) 37
 sequenceable 36
 see also properties
auxiliary web applications 264

B

BarChart 250
batch processing
 state machines 132
 state variable 134
batch processing, example 135
behavior artifacts 115
Behavior modeling 15
business object locator, M2E conversion 99
business process 109
BusinessObjectsBatch 132

C

callOWS 219
ChartProperties 250
ChartView
 actions 252
 chart properties 250
 description 250
class diagrams
 association classes, adding 55
 associations, adding 53
 displaying 329
 displaying connections 331
classes
 abstract 34
 description 33
 superclass 33
classifier feature map
 specifying use in Data Transformation Editor 162
 usage 157
classifier feature map usage 157
classifier map
 description 153
 specifying use in Data Transformation Editor 152,
 154, 155
code assist 300
code completion 300
Cogility Modeler user interface 275
com.cogility.OCL.historyComments.automatic 317
com.cogility.OCL.historyComments.enabled 317
com.cogility.OCL.historyComments.problemIdTitle

318
com.cogility.OCL.historyComments.username 318
com.cogility.OCLCodeAssist.ClassDefs.0 313
com.cogility.OCLCodeAssist.ClassDefs.1 313
com.cogility.OCLCodeAssist.ClassDefs.count 313
com.cogility.OCLCodeAssist.JavaActions.custom.0 312
com.cogility.OCLCodeAssist.JavaActions.custom.1 312
com.cogility.OCLCodeAssist.JavaActions.cus-
tom.count 312
com.cogility.OCLCodeAssist.windowHeight 312
com.cogility.OCLCodeAssist.windowWidth 312
com.ohana.object.persistence.db.class.1 19
com.ohana.object.persistence.db.connection 19
com.ohana.object.persistence.db.count 19
com.ohana.object.persistence.db.databasename.1 19
com.ohana.object.persistence.db.interface.1 19
com.ohana.object.persistence.db.password.1 19
com.ohana.object.persistence.db.servername.1 19
com.ohana.object.persistence.db.username.1 19
complex type support
 inbound web services 205
 outbound web service 219
concurrent composite state 120
configuration map 16
configuration parameters
 com.cogility.OCL.historyComments.automatic 317
 com.cogility.OCL.historyComments.enabled 317
 com.cogility.OCL.historyComments.problemIdTitl
 e 318
 com.cogility.OCL.historyComments.username 318
 com.cogility.OCLCodeAssist.ClassDefs.0 313
 com.cogility.OCLCodeAssist.ClassDefs.1 313
 com.cogility.OCLCodeAssist.ClassDefs.count 313
 com.cogility.OCLCodeAssist.JavaActions.custom.
 0 312
 com.cogility.OCLCodeAssist.JavaActions.custom.
 1 312
 com.cogility.OCLCodeAssist.JavaActions.custom.c
 ount 312
 com.cogility.OCLCodeAssist.windowHeight 312
 com.cogility.OCLCodeAssist.windowWidth 312
 com.ohana.object.persistence.db.class.1 19
 com.ohana.object.persistence.db.connection 19
 com.ohana.object.persistence.db.count 19
 com.ohana.object.persistence.db.databasename.1

19
com.ohana.object.persistence.db.interface.1 19
com.ohana.object.persistence.db.password.1 19
com.ohana.object.persistence.db.servername.1 19
com.ohana.object.persistence.db.username.1 19
model assist 312
reloading (during development) 25
conversions, outbound (E2M) 93
CreateBatch 138
current context, defined 16
custom query
 arguments 183
 attributes 181
 conditions 182
 other clauses 184
 relations 183



D

D2M conversion 166
Data Transformation Editor
 creating a transformation map 152
 creating an opaque transformation map 148
 description 173
 editing existing transformations 178
 interface 173
 panel buttons 175
 panel menus 177
 specify feature map use 154, 155
 specifying a classifier map 152
 specifying classifier feature map use 162
 specifying classifier map use 154, 155
 specifying feature map use 159
 data transformations, in state machines 124
 data types, defining 36
 database artifact 257
 database connection 259
 database connection, LIM 61
 decision state 118
 default deployment model 256
 DeleteBatch 142
 deployment 255, 269
 deployment artifact 257
 deployment, outbound web service 215
 DIM
 see Distinct Information Model (DIM)
 Discover Tables 62
 discover tables
 selection menu 64
 view menu 64
 Discover Tables (Activated Tables)... 62
 Discover Tables (LIM Classes)... 62
 Distinct Information Model (DIM)
 classes 33
 creating 33
 description 30
 Document object 220
 DOM Document object 220
 DOS conversion 94
 DOS event, in state machines 128
 DOS populator 147
 dynamic schema 268

E

E2M conversions
 adding 93
 defined 93
 effect action, in state machines 129
 enterprise objects 110
 event conversion 91
 event trigger, in state machines 127
 events
 abstract events 84
 description 82
 start event 130
 start objects 90
 execute object 71
 execute statement 68

F

factory class 135
 factory classes 48
 feature map
 description 158
 specifying use in Data Transformation Editor 154, 155, 159
 feature map usage 156
 final state 122
 FindSubscriber 137
 FindSubscriberEvent 137
 FindSubscriberM2E 136
 FindSubscriberTopic 136
 fixed schema 268
 FROM SQL statement 181

G

guard condition, in state machines 128

H

HTML construct, HTML View 239
HTML Table Cell Template 249
HTML Table Column 247
HTML Table Row 247
HTML Table Template 248
HTML View
 actions 239
 HTML construct 239
 var construct 239
HTML view
 description 237
HTMLTable 240
HTMLTable object 240
HTMLTableCellArea 245
HTMLView
 defining 238

L

Latest Version Only checkbox 17
Legacy Information Model (LIM)
 activated table actions 77
 activating 67, 75, 76
 active mode 61
 Configuration Parameters 78
 container 61
 creating 61
 database connection 61
 deactivating 76
 description 61
 discover table actions 62
 discover tables dialog 63
 discover tables selection menu 64
 discover tables view menu 64
 In-line SQL Statements 67
 passive mode 61
 Statement objects 69
 test connection 62

LIM

 see Legacy information model (LIM)

LIMClass 66

limitations, batch processes 132

lock file 18

I

icons, model assist 302
immutable and mutable artifacts. 24
importing a WSDL 212
inbound web service
 complex type support 205
 SOAP over HTTP 228
inconsistencies
 red flag 269
 yellow 269
initial state 118
internal event 84
isSequenceable 39, 108
IWS endpoint 210
IWS Parameters 203

J

Java methods
 toXML 191, 199
 toXMLFragment 191, 199
JMS messages 103

M

M2D conversion 165
 map (configuration map) 16
 Master Information Model (MIM)
 classes 33
 creating 30
 description 29
 message definitions 108
 Message modeling 15, 81
 message modeling
 abstract events 84
 description 82
 DOS conversion 94
 event conversion 91
 events 82
 internal event 84
 JMS messages 103
 message definitions 108
 messages 103
 standard event 84
 transient class 89
 web service messages 104
 messages 103
 messages, defining for JMS 107
 MIM
 see Master Information Model (MIM)
 model assist
 configuration parameters 312
 icons 302
 model container 21
 multiplicity
 adding 52
 expression 53
 valid formats 52
 mutable and immutable artifacts 24

N

newBatchFor 132
 newRow bound variable 77

O

oldRow bound variable 77
 Omit Null in Result 204
 opaque transformation map
 creating 148
 creating in Data Transformation Editor 148
 description 146
 operations
 initialize 44
 outbound web service 217
 postAssociate 44
 postInitialize 44
 Oracle JMS Router 266
 outbound web service
 complex type support 219
 description 212
 importing a WSDL 212
 testing 214
 outbound web service artifacts 214
 OWS deployment artifact 215

P

parallel transformation chain
 creating 170
 description 169
 input and output DOS 169
 parameters
 inbound web service 203
 parent bound variable 139
 parent known variable 134
 passive mode 61
 PEAR
 see Persistent Element Attributes and Relationships (PEAR)
 Persistent Element Attributes and Relationships (PEAR)
 description 268
 fixed schema 268
 run-time repository 268
 Persistent Entities, Attributes, and Relationships (PEAR)
 deployment 268
 persistent object identifier (POID) 37
 ProcessBatch 139
 pushing a model 256

R

red flag inconsistencies 269
reflexive associations 57
reloading configurations 25
replicate state 121
repository.ser file 21
roles 51
row bound variable 77
run-time repository 257, 267

S

select object 70
SELECT SQL statement 181
sequential composite state 118
serial transformation chain 168
 creating 168
 description 168
session lock file 18
Session.slk 18
simple state 118
source class 53
source multiplicity 53
source role 51, 53
SQL block object 73
standard event 84
start event 130
start object 90
state activity diagram 110, 117, 122
state machine 112
 definition 109
 description 112
state machine editor 115
State transactions 111
state transition 125
state types 117
state variable 134
statement object 69
subscriber class 135
Subscriber MIM class 135
superclass 33

T

target class 53
target role 51
time event
 action 88
 configuration parameter 88
 in state machines 86
 time expression 87
time expression override 88
toXML 191, 199
toXMLFragment 191, 199
transacted JMS messages 112
transformation
 artifacts 146
 description 145
transformation chain
 adding transformation units 172
 D2M conversion 166
 description 163
 M2D conversion 165
transformation map
 creating in Data Transformation Editor 152
 defining 153
 description 149
 hierarchy 149
Transformation modeling 15
transformations, in state machines 124
transient class 89
transition
 description 125
 effect action 129
 guard condition 128
Transition transactions 111
trigger event, in state machines 127

U

UML (Unified Modeling Language) state diagram 109

V

var construct, HTML View 239
version (model version) 17
virtual package 23



W

web service action 203
web service definition language (WSDL) 215
web service messages 104
Web service modeling 15
web service transaction 112
web service, inbound
 parameters 203
 SOAP over HTTP 228
web service, outbound
 deployment 215
 description 212
 importing a WSDL 212
 operations 217
 testing 214
 WSDL 215
WSDL
 importing 212
 outbound web service 215
WSDL file import 212

X

XML encoding style 218
XmlGetString() 220
XmlLoad() 220
XSD artifacts 188
XSD fragment 204
XSD types 190

Y

yellow flag inconsistencies 269

