



Cogility Studio Tutorial

Copyright © 2001-2010 Cogility Software Corporation.

All Rights Reserved.

Cogility Studio Tutorial 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 North Market St. Suite #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.



Preface

Cogility Studio documentation	7
-------------------------------------	---

Chapter 1 Information modeling

Building the model	9
Repositories	11
Authoring repository	11
Run-time repository	11
Starting Cogility Modeler	11
Loading the model repository	11
Logging in	12
Defining the model container	13
Creating the information model	14
Master Information Model	14
Classes	15
Class diagram	16
Customer class	16
Order class	17
Address class	18
Product class	19
Class inheritance	20
Subclasses	20
Hierarchy	20
Class relationships	21
Associations	21
Association classes	23
Turn over and push	25

Chapter 2 Using action semantics

Importing the existing model	27
Hello world	28
Cogility Action Pad	28
Working with action semantics	28
Compiling and executing	30
Working with action pads	31
Object instance	33
Locating objects	35
Flow control	37
Creating associations	38
Working with collections	40

Chapter 3 Inbound JMS

Importing the existing model	43
Message model	44
Creating the event artifact	44
Creating the conversion artifact	46
Object creation	48

Creating an instance with a factory	48
Behavior model	50
Turn over and push	56
Executing the model	57
Exporting message definitions	58
Running Cogility Message Traffic Viewer	59
Running Cogility Insight	61

Chapter 4 Outbound JMS

Importing the existing model	65
Adding a new attribute	65
Extending the message model	67
Creating the event artifact	67
Creating the conversion artifact	68
Updating factory behavior	69
Turn over and push	70
Executing the model	71
Running Cogility Message Traffic Viewer	71
Running Cogility Insight	73

Chapter 5 Inbound web services

Importing the existing model	76
Creating an inbound web service	76
Defining the argument message	76
Defining the result message	77
Defining the web service logic	78
Creating a deployment	80
Turn over and push	81
Executing the model	82
Running Cogility Message Traffic Viewer	82
Running Cogility Insight	84
Viewing the web service deployment	85
Executing the web service	87

Chapter 6 Outbound web services

Importing the existing model	92
Updating the information model	92
Creating the outbound web service	93
Importing a WSDL from a file	93
Testing the web service	94
Updating the inbound web service	95
Adding a parameter to the inbound web service	95
Updating the inbound web service logic	95
Turn over and push	97
Executing the model	97
Running Cogility Message Traffic Viewer	97
Running Cogility Insight	99

Chapter 7 Class operations

Importing the existing model	103
Creating a new class diagram	104
Define the class hierarchy	105
Define the association	105

Defining class behavior	107
Defining operations	108
Defining the method signatures	109
Defining the method bodies	110
Invoking operations	112
Event and message conversion	113
Creating an inbound web service	116
Defining the web service	116
Defining the deployment	117
Defining the web service logic	118
Turn over and push	118
Writing the message file	119
Executing the model	119
Running Cogility Message Traffic Viewer	119
Running Cogility Insight	122
Sending a trouble ticket message	124

Chapter 8 Custom queries

Importing the existing model	127
Extending the information model	127
Adding a class	128
Creating the association	128
Creating the query	129
Attributes	130
Arguments	131
Conditions	132
Relations	132
Other clauses	133
Turn over and push	134
Verifying SQL	134
Testing the query	135
Running Cogility OCL Development Tool	135
Creating the sample data	136
Calling the query	136



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>.



Information modeling

This tutorial consists of several self-contained modules that lead you step by step through the procedures for building an application integration model with Cogility Modeler and testing it as an enterprise application on a J2EE application server with other Cogility Studio utilities. The procedures review the concepts described in the training course slides, though this book and the training course may be pursued independently. With each chapter of procedures in this book, you work from basic to more advanced modeling functionality. The chapters are described as follows:

- **Information modeling** (this chapter) describes the tutorial model, presents the basic steps for getting Cogility Studio up and running, and starts you off with building the initial model artifacts. You learn about loading repositories and the Cogility Modeler user interface.
- **“Using action semantics” on page 27** introduces you to the *action semantics* that define model logic. Action semantics are based on the Object Constraint Language of UML and include several enhancements for working with model objects at run time.
- **“Inbound JMS” on page 43** guides you through building the model artifacts that represent the message data sent from the applications (inbound) to the model and the events that spawn business processes in the model.
- **“Outbound JMS” on page 65** continues the data path that started in the previous model, this time showing you how to build artifacts for communicating data and events from the model (outbound) to the integrated applications.
- **“Inbound web services” on page 75** considers communication from the integrated applications to the model using web services as an alternative to JMS messages.
- **“Outbound web services” on page 91** describes how your model can communicate with third-party web services.
- **“Class operations” on page 103** shows you how to create a super class for model artifacts and subclasses that inherit operations from the super class.
- **“Custom queries” on page 127** shows you how to develop complex SQL queries to work with run-time data.

Cogility Modeler is the design and development environment, akin to an integrated development environment (IDE), for building an application integration model. Once you have a model created, you then push it into execution as an enterprise application on a J2EE application server. The application represents a composite of all the applications integrated in the model and the model functionality to communicate from one external application to another.

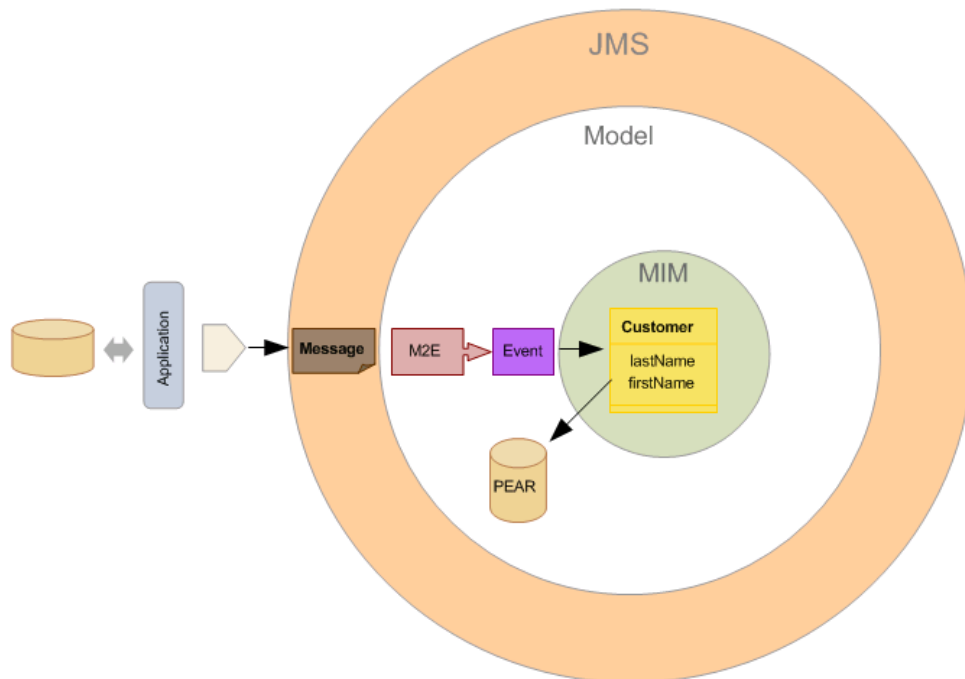
Building the model

For the first part of this tutorial, you have only one application to integrate with the model. The enterprise application you push into execution consists of the application classes and the model objects that facilitate communication between the model and the external application. The idea is to

teach you how to integrate one application, and once you understand the basic concepts involved, you can move on to integrating two or more applications in a model and creating a true composite application in later chapters.

The model you build in this tutorial starts with a simple customer creation scenario wherein a customer is created in a hypothetical external application. During execution that application sends a JMS (Java Messaging Service) message to the model's enterprise application with the new customer information. A message conversion takes the new customer information and spawns a customer creation event. The model then creates a new Customer object with the information passed from the message to the event, placing it in the new Customer object. For each step in the process, you create model artifacts to handle the customer creation scenario: a message conversion artifact, an event artifact, a Customer artifact and so on. The enterprise application you push on to the J2EE application server calls these objects when it executes.

Throughout this tutorial, illustrations showing the model artifacts and how they fit together help explain the building process. In the figure below, the first artifacts you create for the Customer creation scenario are shown. Starting from the left, the external application sends a JMS message to the model. The JMS messaging layer separates the model from the external application and is shown as a tan circle surrounding the model artifacts. Within the circle (that is, within the model), are the conversion artifact that takes the message and converts it to a customer creation event and the event artifact itself. Finally, a Master Information Model (MIM) that holds all the class information for the Customer artifact is shown. In this scenario, during execution, the model's enterprise application creates a new Customer object each time it receives a new Customer message, and writes the new Customer to the run-time repository (PEAR in the figure below).



By the way, the external application is purely hypothetical. You do not have to create it. You only have to create a message artifact to handle messages from the application. You simulate broadcasting the messages with Cogility Message Traffic Viewer to test your enterprise application.

Also in this chapter, you add an Order class to the model and associate it with the Customer. The Order class also is associated with a Product class that you create as well. As you work through the

tutorial, you add other model artifacts as the model increases in complexity. Each chapter of this book introduces new concepts and functionality.

Repositories

Cogility Studio uses two types of repositories to hold your models, an authoring repository and a run-time repository.

Authoring repository

As you build and work with a model in Cogility Modeler, the changes you make are automatically saved to a file on your system. You designate this file as the authoring repository for the model. Each time you create a model artifact or make a change, the edition is recorded in the authoring repository. In the event of a system failure, when you restart Cogility Modeler and open the repository, your model opens up to date with the latest edition.

In your configuration file, you can set up several authoring repositories to hold several models, as explained in the guide, *Model Deployment & Execution in Cogility Studio*. Each time you run Cogility Modeler, you are prompted to choose a repository to open from a list of those you have set up.

Once you have set up a repository, you must load it with several Cogility files that keep track of your model artifacts. You see how to do this in [“Loading the model repository” on page 11](#).

Run-time repository

When you are ready to test a model, you push it as an enterprise application onto a J2EE application server. During the push process, all of the model objects and data are stored in an SQL database that becomes the run-time or execution repository, also called the PEAR (Persistent Element Attributes and Relationships) repository.

The guide, *Installing and Configuring Cogility Studio* describe how to set up the run-time repository. The instructions assume you use only one run-time repository, which means that each time you push a new model into execution, you must remove the old model from the repository. For this tutorial, you do not have to remove the existing model each time you push the model at the end of each chapter because with each push you are adding to the database schema, not changing it. If you open other Cogility example models, however, you must remove the existing model before pushing the new one.

Starting Cogility Modeler

If Cogility Studio is not already installed on your system, follow the instructions in the guide, *Getting Started with Cogility Studio*. This tutorial assumes you are running a default installation of Cogility Studio as prescribed in the installation instructions.

Loading the model repository

After you have installed and configured Cogility Studio, you need to create an authoring repository for use with this book’s tutorial model. The authoring repository holds the model while you work with it in Cogility Modeler. See [“Authoring repository” on page 11](#) for more information.

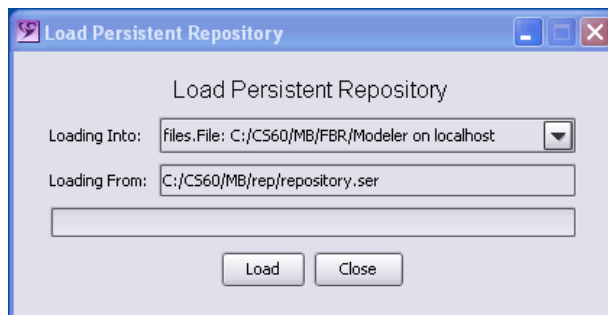
To load the repository:

1. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Modeler > Load Persistent Repository**.

The Load Persistent Repository dialog displays.

2. For the **Loading Into** field, leave the default selection unchanged.

This assumes that you have configured only one authoring repository, according to the installation instructions in *Getting Started with Cogility Studio*. If you have configured Cogility Studio to work with more than one authoring repository, select the repository to hold your tutorial model.



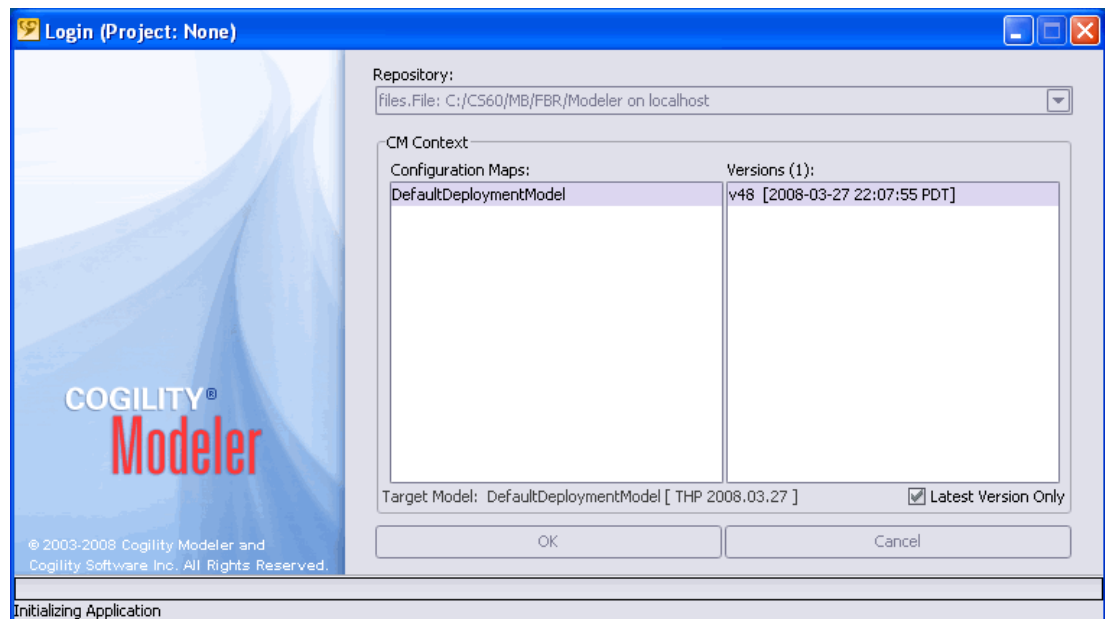
3. Click **Load** to proceed with repository creation.

Logging in

To log in to Cogility Modeler:

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

Because you are logging in with a new, empty repository, only the default deployment model is shown in the list of configuration maps. The empty repository loads automatically.




This deployment model is selected by default and Cogility Modeler opens on your desktop with the default deployment model loaded. The deployment model defines several deployment configurations for your model. Each of these configurations specifies an application server and a database. When you deploy your model, you will select the appropriate configuration for your installation. See the guide, *Modeling with Cogility Studio* for more information about deployment models.

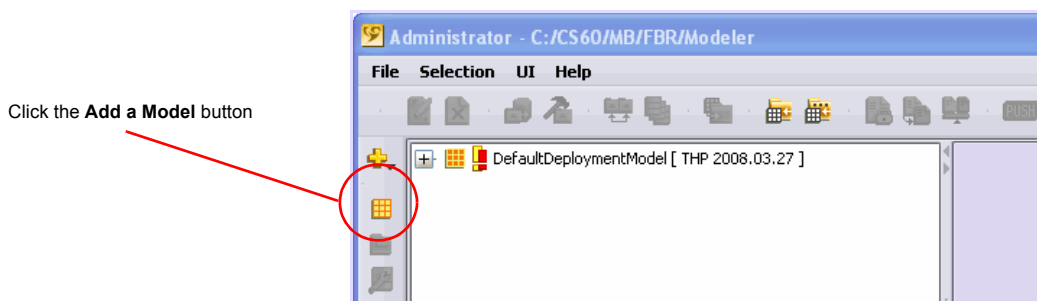
Defining the model container

You are now ready to begin building an integration model. The first task is to create the model container under which you define all other model artifacts. When you refer to fully-qualified model objects with action semantics, you specify an object in relation to the model container with the following format:

```
modelContainerName::containerName::objectName
```

To add a model container:

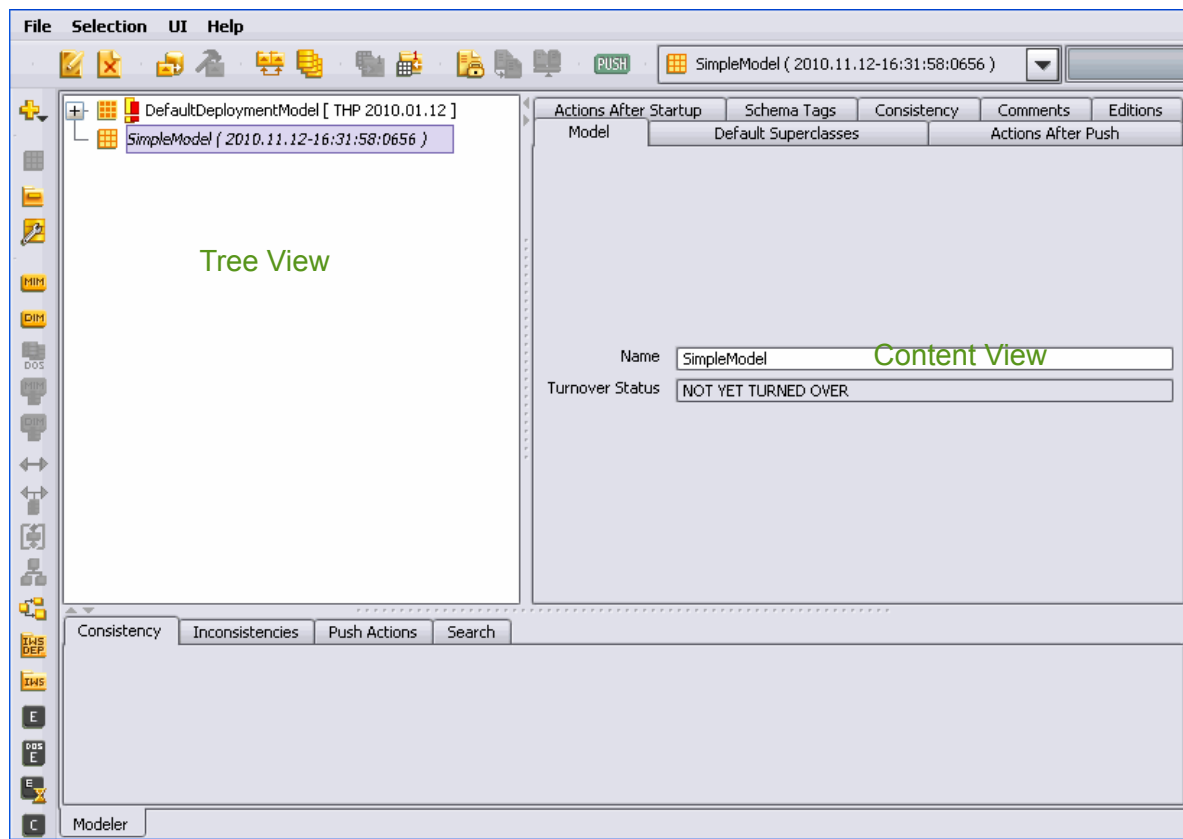
1. In Cogility Modeler, click the **Add a Model** button .



An unnamed model container appears as the top level artifact in Cogility Modeler's tree view, the left pane.

2. In the content view, in the **Name** field, enter **SimpleModel**.
Leave the other fields unchanged. The content view shows the fields for the model's deployment settings. These are the default settings included with the model for

convenience. However, you won't use this deployment definition, rather one from the deployment model.



You have now created the model container and set the properties that allow you to push your model as an enterprise application and run it with your application server.

Creating the information model

The information model portion of your integration model describes the classes for the data objects that are created and updated in the enterprise application at run time. Entities such as customers are described in these classes. When the enterprise application executes, it creates Customer objects according to the class definition in the information model.

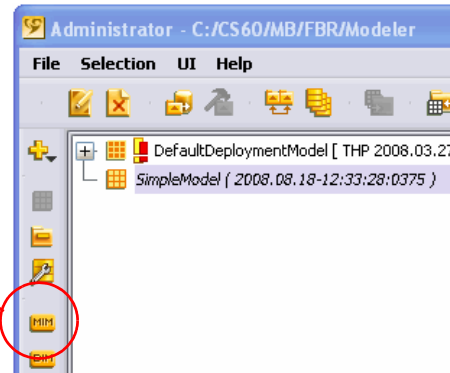
Master Information Model

Before it may be pushed into execution, a model must have a Master Information Model (MIM) with at least one class. The MIM is the artifact that includes class definitions for all data objects. At run time, it is the container for all data objects. A model has only one MIM.

To add a MIM:

1. In the tree view, select the **SimpleModel** model container and click the **Add a Master Information Model (MIM)** button .

Click the **Add a Master Information Model (MIM)** button




An icon representing the MIM artifact appears in the tree view.

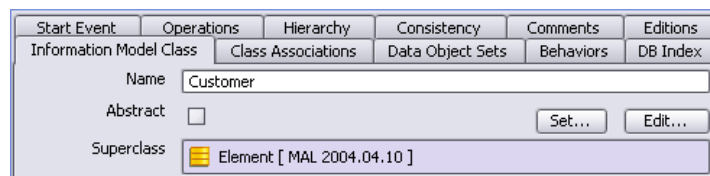
2. In the content view, under the **Information Model** tab, leave the default name, **MIM**, as is.

Classes

Each data object in your enterprise application is defined by a class in the MIM. There are two ways to create a class in your model. The first, described here, uses the tool bar to access the class editor, the second uses a class diagram (as described in [“Class diagram” on page 16](#)).

To add the Customer class:

1. In Cogility Modeler's tree view, click the plus sign (+) to expand the **Domain Model**.
2. Select the **MIM** and click the **Add a MIM Class** button .
3. In the content view, under the **Information Model Class** tab, in the **Name** field, enter **Customer**.



4. Above the **Attributes** field, click **New**.
 - a. In the dialog, for **Name**, enter **firstName**.
 - b. For **Type**, select **String** from the drop-down menu, and click **OK**.
The **firstName** attribute appears in the Feature list.
5. Above the **Attributes** field, click **New** again.
 - a. In the dialog, for **Name**, enter **lastName**.
 - b. For **Type**, select **String** from the drop-down menu, and click **OK**.

The lastName attribute appears in the Feature list.

The screenshot shows the 'Information Model Class' editor for the 'Customer' class. The 'Name' field is 'Customer'. The 'Abstract' checkbox is unchecked. The 'Superclass' is 'Element [MAL 2004.04.10]'. The 'Attributes' list contains two attributes: 'firstName [Chapter1]' and 'lastName [Chapter1]'. Below the list, the 'Attribute' editor shows the details for 'firstName': Name is 'firstName', Size is '0', and Type is 'String [MAL 2004.04.10]'.

Class diagram

A class diagram lets you represent your information model visually according to standard UML notation. It shows the classes of your model and their relationships. So far you have defined a Customer class. You also define Order, Address and Product classes, this time using the class diagram editor.

To create a class diagram:

1. In Cogility Modeler's tree view, under **Domain Models**, select the **MIM**, and click the **Add Class Diagram** button .


The class diagram editor appears in the content view.

2. In the **Name** field, enter **CustomerModel**.

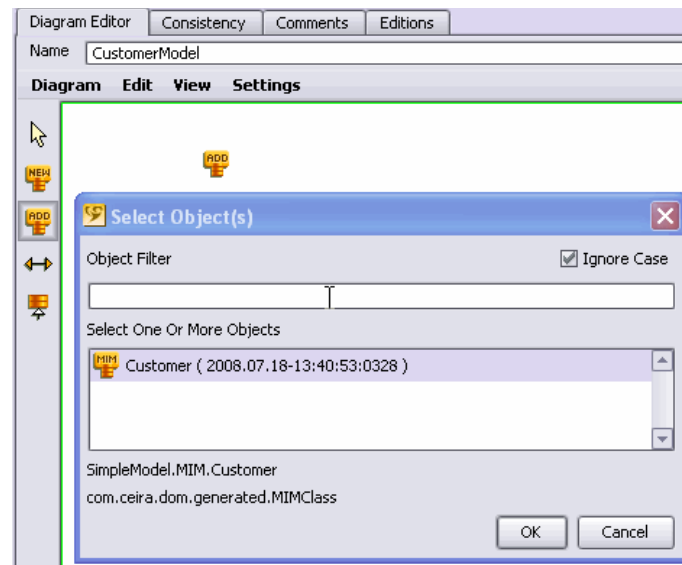
Customer class

You have a Customer class in your model already; all you have to do is add it to the diagram.

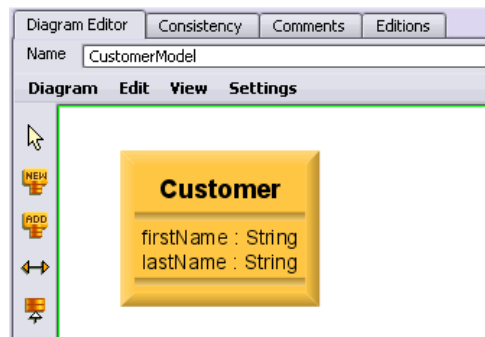
To add an existing class to the diagram:

1. In the class diagram editor tool bar, click the **Add Existing Class**  button.
2. Place the cursor in the class diagram editor window and click again.

This places the icon for the class in the diagram and opens a class chooser dialog.




3. In the selection dialog, select **Customer** and click **OK**.
The diagram editor displays the Customer class icon.



Order class

You can also create new classes through the class diagram editor. Here, you create an Order class with orderDate and orderID attributes.

To create the Order class:

1. In the class diagram editor tool bar, click the **Create New Class** button .
2. Place the cursor below the Customer class icon in the diagram and click again.
This locates a new class icon in the diagram.
3. Double-click the **UntitledMIMClass** icon to open its editor window.
4. In the class editor window, under the **Information Model Class** tab, in the **Name** field, enter **Order**.
5. Above the **Attributes** field, click **New**.
 - a. In the dialog window, for **Name**, enter **orderDate**.
 - b. For **Type**, select **String** from the drop-down menu and click **OK**.

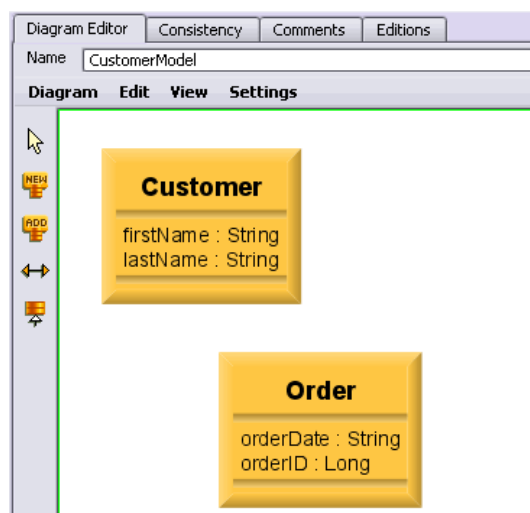
6. Above the **Attributes** field, click **New** again.
 - a. In the dialog window, for **Name**, enter **orderID**.
 - b. For **Type**, select **Long** from the drop-down menu and click **OK**.
 - c. In the **Attribute** tab, click the **Is Sequenceable** checkbox.

By designating this attribute as sequenceable, you are telling the database to automatically create a unique orderID value for each instance of the class.

The screenshot shows the 'Order' class editor. The 'Attributes' tab is active, displaying a list of attributes. The 'orderID' attribute is selected, and its details are shown in the 'Attribute' tab below. The 'Is Sequenceable' checkbox is checked.

7. Close the class editor window.


Your class diagram now resembles the one shown below.



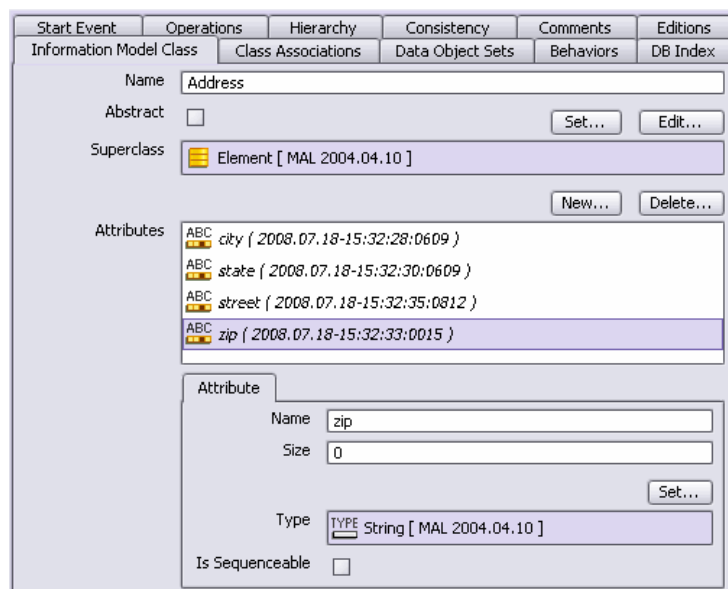
Address class

An Address class has street, city, state, and zip information.

To create the Address class:

1. In the class diagram editor tool bar, click the **Create New Class** button .
2. Place the cursor below the Customer class icon in the diagram and click again.
This locates a new class icon in the diagram.
3. Double-click the **UntitledMIMClass** icon to open its editor window.
4. In the class editor window, under the **Information Model Class** tab, in the **Name** field, enter **Address**.
5. Above the **Attributes** field, click **New**.
 - a. In the dialog window, for **Name**, enter **street**.
 - b. For **Type**, select **String** from the drop-down menu and click **OK**.
6. Repeat step 5 to add the remaining attribute names and types:
 - ❑ **city** (type: **String**)
 - ❑ **state** (type: **String**)
 - ❑ **zip** (type: **String**)

The class editor window should appear as in the figure below.




7. Close the class editor.

Product class

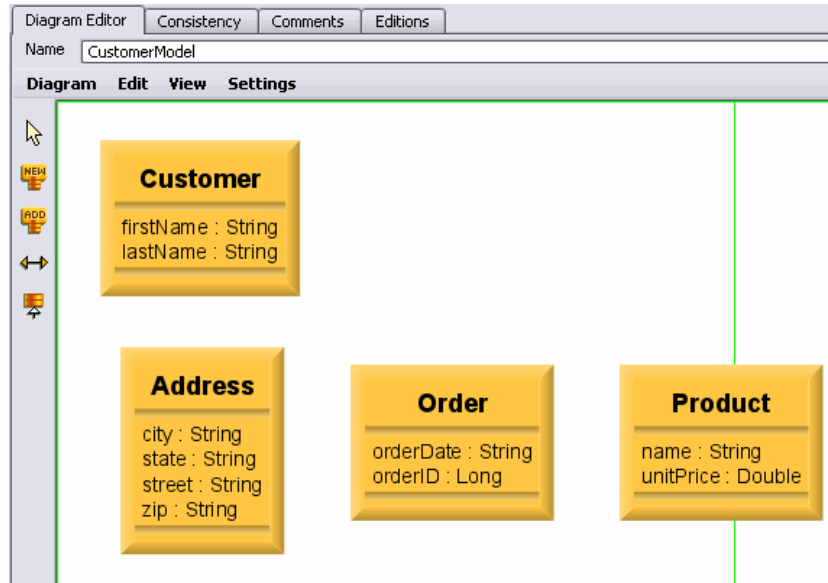
You create the Product class just as you did the Address class.

To create the Product class:

1. In the class diagram editor tool bar, click the **Create New Class** button .
2. Place the cursor next to the Order class icon in the diagram and click again.
This locates a new class icon in the diagram.
3. Double-click the **UntitledMIMClass** icon to open its editor window.
4. In the class editor window, under the **Information Model Class** tab, in the **Name** field, enter **Product**.
5. Above the **Attributes** field, click **New**.

- a. In the dialog window, for **Name**, enter **name**.
 - b. For **Type**, select **String** from the drop-down menu and click **OK**.
6. Above the **Attributes** field, click **New** again.
 - a. In the dialog window, for **Name**, enter **unitPrice**.
 - b. For **Type**, select **Double** from the drop-down menu and click **OK**.
7. Close the class editor window.

Your class diagram should look something like the one in the picture below.




Class inheritance

The class diagram editor also lets you describe a class hierarchy. Subclasses inherit members from superclasses. In this example, the **Product** superclass has two subclasses that inherit the **Product** class attributes.

Subclasses

The **Product** class has two subclasses, a **Vacuum** class and **FloorMop** class.


To create the subclasses:

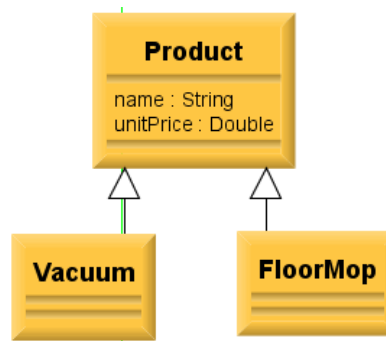
1. In the class diagram editor tool bar, click the **Create New Class** button .
2. Place the cursor below the **Product** class icon in the diagram and click again.
This locates a new class icon in the diagram.
3. Double-click the **UntitledMIMClass** icon name.
You can edit class icon labels directly just by double-clicking them.
4. Enter the name, **Vacuum**.
5. Repeat the previous steps to create a new class named **FloorMop**.

Hierarchy

A class inheritance arrow from the subclass to the superclass describes the class hierarchy.

To define a class hierarchy:

1. In the class diagram editor tool bar, click the **Define Superclass** button  .
2. Click the **Vacuum** icon, then move the cursor to the **Product** icon and click again.
A class inheritance arrow is drawn from the Vacuum class to the Product class.
3. Repeat the previous step for the **FloorMop** class, specifying the **Product** class as its superclass.



Class relationships

The classes in the diagram are related to each other. The Customer has Orders, and the Order has Products. These entity relationships are described in UML with associations or association classes. In this section you use the class diagram editor to create these relationships.


Associations

An association describes the relationship between an owning class and a target class. The owning class has the source role; the target class has the target role.

Customer to Order

The Customer class has the source role, owning one or more Orders.

To create an association between the Customer and Order classes:

1. In the class diagram tool bar, click the **Create New Association** button  .
2. Place the cursor over the Customer class and click.
3. Move the cursor over the Order class and click again.
When you release, a selection window appears.
4. In the selection window, select **Association** and click **OK**.
An undefined association appears as a line between the Customer and Order classes.
5. Double-click the association line.
An association editor appears.
 - a. For **Source Role**, leave the default, **customer**.
The role describes the relationship that a class has to another. The source role owns the relationship.
 - b. For **Source Multiplicity**, enter 1.

The Order may be owned by only one instance of a Customer. The source multiplicity refers to how many instances of the source role may pertain to the target role.

- c. For **Target Role**, leave the default, **order**.

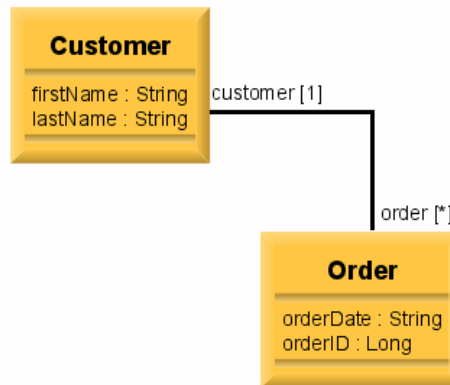
The target role describes the target class' relationship to the source class.

- d. For **Target Multiplicity**, leave the default, *****.

The Customer instance may have zero to many Orders. This is connoted with an asterisk * (or the notation, 0..*).

6. Close the Association editor.


The relationship should appear in the diagram as in the picture below.



Customer to Address

The Customer class may be associated with one or more Addresses.

To create an association between the classes in the diagram.

1. In the class diagram tool bar, click the **Create New Association** button .

2. Place the cursor over the **Customer** class and click.

3. Move the cursor over the **Address** class and click again.

When you release, a selection window appears.

4. In the selection window, select **Association** and click **OK**.

An undefined association appears as a line between the Customer and Address classes.

5. Double-click the association line.

An association editor appears.

- a. For **Source Role**, leave the default, **customer**.

The role describes the relationship that a class has to another. The source role owns the relationship.

- b. For **Source Multiplicity**, enter **1**.

The Address may have one and only one Customer. The source multiplicity refers to how many instances of the source role may pertain to the target role.

- c. For **Target Role**, leave the default, **address**.

The target role describes the target class' relationship to the source class.

- d. For **Target Multiplicity**, leave the default, *****.

The Customer may have zero to one Address. This is connoted with the notation, 0..1. Your association editor should appear as in the figure below.

Association editor dialog box showing the configuration for an association between Customer and Address classes.

Field	Value
Name	A_address_customer
Source Class	Customer (2008.07.18-13:40:53:0328)
Source Role	customer
Source Multiplicity	1
Target Class	Address (2008.07.18-15:44:42:0468)
Target Role	address
Target Multiplicity	0..1
Delegates Events	<input type="checkbox"/>


6. Close the Association editor.

The model now includes an Address class that is associated with a Customer.

Association classes

Like an association, an association class describes the relationship between two classes with source and target roles. The association class goes further, however, by letting you define attributes that describe the relationship. The association class is best used when the attributes that describe the relationship do not properly belong to either class, but are meaningful only within the context of the relationship. In this example an association class attribute describes the quantity of a specific Product on an Order. Notice that the quantity attribute could not properly belong to either class, but is meaningful when describing the relationship between an Order and a Product.

To create an association class:

1. In the class diagram tool bar, click the **Create New Association** button .
2. Place the cursor over the **Order** class and click.
3. Move the cursor over the **Product** class and click again.
When you release, a selection window appears.
4. In the selection window, select **AssociationClass** and click **OK**.
An undefined association appears as a line between the Order and Product classes.
5. Double-click the association line.
An association editor appears.
 - a. For **Source Role**, leave the default, **order**.
 - b. For **Source Multiplicity**, enter *****.
The same Product may be referenced by several Orders. Note that this is not an inventory Product (or the instance of a Product in inventory) which could pertain to only one order. Setting up the relationship this way allows the system to discover how many Orders refer to the Product.
 - c. For **Target Role**, leave the default, **product**.

- d. For **Target Multiplicity**, enter 1..*.

An Order must consist of one or more Products.

6. Click on the **AC Attribute Definition** tab.

7. For **Attributes**, click **New**.

A dialog appears.

- a. In the **Name** field, enter **quantity**.

- b. For **Type**, select **Integer** from the drop-down list and click **OK**.

- c. Close the **A_product_order** association class editor.

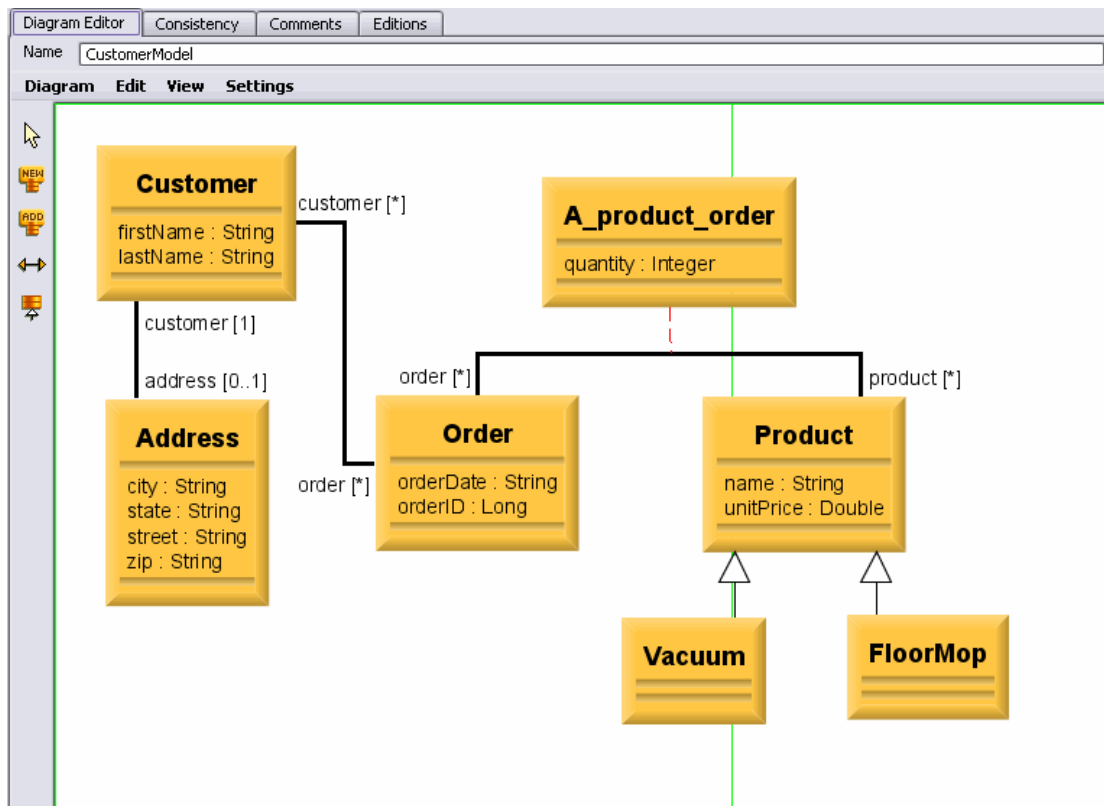
8. In the diagram editor tool bar, click the **Add Existing Class**  button.

9. Place the cursor in the class diagram near **Product** or **Order** class and click again.

10. In the dialog, select the **A_product_order** association class and click **OK**.

The association class appears in the diagram connected with a red dashed line to the association between the Order and Product classes. Note that you have to add the association class to the diagram to display it, just like any other class.

Your class diagram should now look something like the following:




You have now created the basic model artifacts required for an enterprise application. At this point the model still does not function, but you can test the model to see if it deploys successfully as a J2EE application.

For more information about MIMs, classes and associations, see the guide, *Modeling with Cogility Studio*.

Turn over and push

If your model meets the minimum requirements for deployment, you can push it on to your application server as a enterprise application.

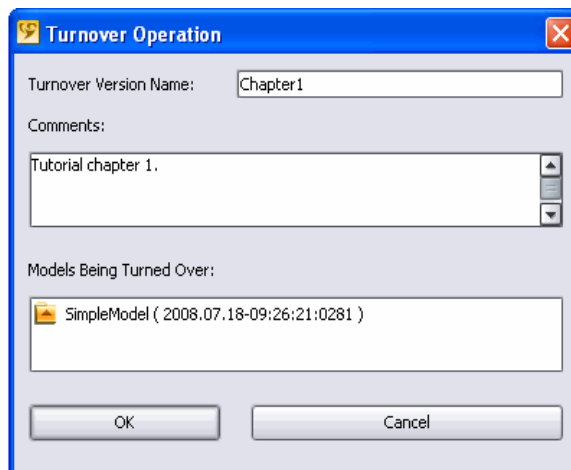
To see if the model pushes successfully:

1. In the tree view, select the **SimpleModel** model container.
2. In the Cogility Modeler tool bar, click the **Turn Over** button .

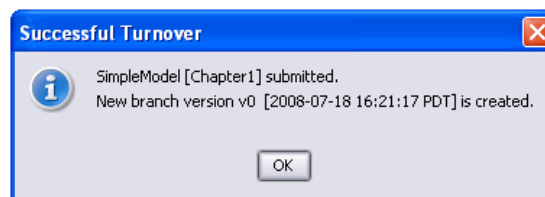
Before you can push the model, you must turn it over. Turning over the model updates the change management system with the latest changes to your model. Each time you turn over the model you have the opportunity to create a new version. For this turnover, you establish the original version of your model.

For more information about the change management system, see the guide, *Change Management in Cogility Studio*.

3. In the Turnover Operation dialog, for **Turnover Version Name**, enter **Chapter1**.
4. In the Comments field, enter some comments like those in the figure below and click **OK**.




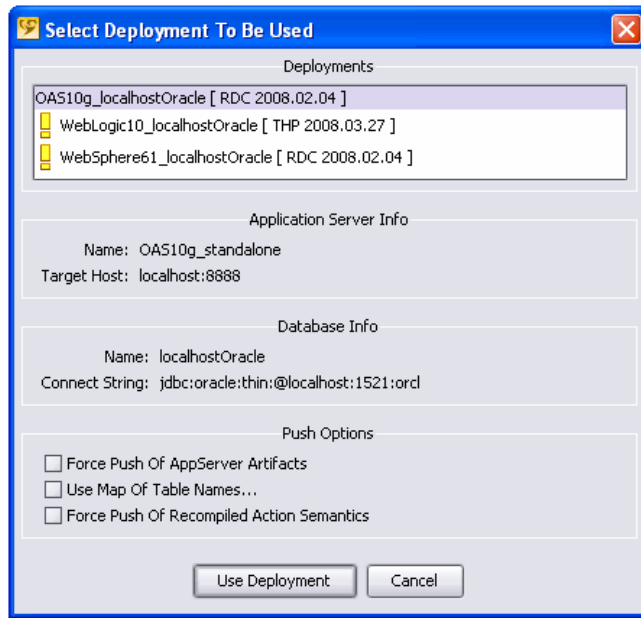
If the turnover operation is successful, a notification appears as in the figure below.



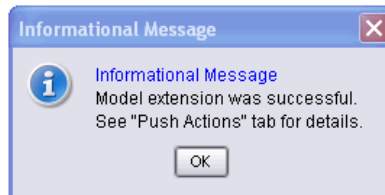
5. Click **OK**.
6. If you have already pushed a model into your run-time repository, remove the current model from the database. (Skip this step if you have not yet pushed a model.)
 - ❑ Navigate to the %DCHOME%\MB\Scripts\Dbtools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.

A command window appears and shows the Drop Tables utility executing. Close the window when it finishes.
7. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.

8. With the **SimpleModel** model container selected, click the Push button .
This pushes the enterprise application onto the application server, and the model objects and schema into the run-time repository.
9. In the Select Source of Deployment Parameters, select the object that matches your configuration and click **Use Deployment**.



When the push operation completes, a dialog appears notifying you of successful deployment.



10. Click **OK**.

The model you created now has a representation as an enterprise application, though it does not have any functionality. The data objects you created have tables in the run-time repository. In later chapters, you add functionality to create Customer objects and populate the tables. You also add business logic to the model and test it as a functional enterprise application on the J2EE application server.



Using action semantics

In the previous chapter, “[Information modeling](#)” on page 9, you started creating a basic model with a several classes. In this chapter, you work with the model and its schema using *action semantics*.

You use action semantics in Cogility Studio to describe the logic or actions of model objects. Modeling according to the UML standard requires a language that understands UML concepts. Typical programming languages fall short in this area, and come with more functional overhead than is practical. Action semantics are designed to be consistent with UML rules. They are easier to work with than most programming languages and include built-in database and web services connectivity, type checking, error and consistency checking, and code assistance (or code completion). Furthermore, action semantics include facilities for using Java code within the action semantics statements, allowing the use of existing Java libraries and types.

Action semantics are an extension of the Object Management Group’s standard Object Constraint Language (OCL) that expresses constraints on relationships between objects of a system. While OCL specifies conditions on changes to the instance model, these specifications are not executable. OCL is not an action language so action semantics are required for executable specifications. Action semantics combine OCL with a specification for the execution syntax developed by Cogility. For more information about UML and OCL, visit the following site:

http://www.omg.org/gettingstarted/what_is_uml.htm

This chapter’s exercises get you familiar with action semantics, starting with the basic “hello world” program and proceeding to working with keywords and Java actions.


Importing the existing model

In the last chapter, you developed a model called SimpleModel. The exercises in this chapter work with that model. If you do not have that version of the model loaded in Cogility Modeler, you can import it with the following procedure. If you are continuing from “[Information modeling](#)” on page 9, skip to “[Hello world](#)” on page 28.

To import the existing model:

1. If you have loaded in Cogility Modeler a model other than the SimpleModel created in “[Information modeling](#)” on page 9, close Cogility Modeler and follow the steps under “[Loading the model repository](#)” on page 11.
2. If Cogility Modeler is not running, follow the steps under “[Logging in](#)” on page 12.
3. In Cogility Modeler, from the **Selection** menu, select **Import/Export > Import from Single File**.
4. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial directory, select **SimpleModel_Chapter1.cog** and click **Open**.

%DCHOME% is the directory where you installed Cogility Studio.

5. In the dialog that prompts you, click **Yes** to make the model visible in Cogility Modeler.
6. In the tree view, select the **SimpleModel** model container and click the **Place Model Under CM** button .

You are now ready to follow the procedures in this chapter.

Hello world

Bowing to tradition, this introduction to action semantics starts with a discussion of the “hello world” program. This simple program illustrates variable declaration and definition, and the standard output utility.

```
str : java::lang::String;  
str := 'Hello Wayne's World';  
oclprintln(str);
```

In the first statement the `str` variable is declared as a String. Notice that action semantics make use of Java primitives in addition to their own standard primitives. Action semantics use double colons `::` to qualify the String type, and declarations proceed from right to left with the declared variable to the left of a single colon `:`. The second statement, a variable assignment, uses a single colon in combination with an equal sign `:=`. Variable assignment also proceeds right to left and String values are held in single quotes. All statements end with single semicolon `;`. The last line utilizes the standard output keyword `oclprintln()`. Pass it the previously declared `str` variable.

Cogility Action Pad

To see if the “hello world” program above actually works, you need a utility that lets you write and test action semantics. Cogility Action Pad provides an editor, compiler, and execution environment for action semantics. For more information about Cogility Action Pad, see the guide, *Model Deployment & Execution in Cogility Studio*.

Working with action semantics

Cogility Action Pad lets you compile and execute action semantics. Its editor comes with code assist (or code completion) which provides a shortcut to completing lines of code with known variables and types.

To run “hello world” with Cogility Action Pad:

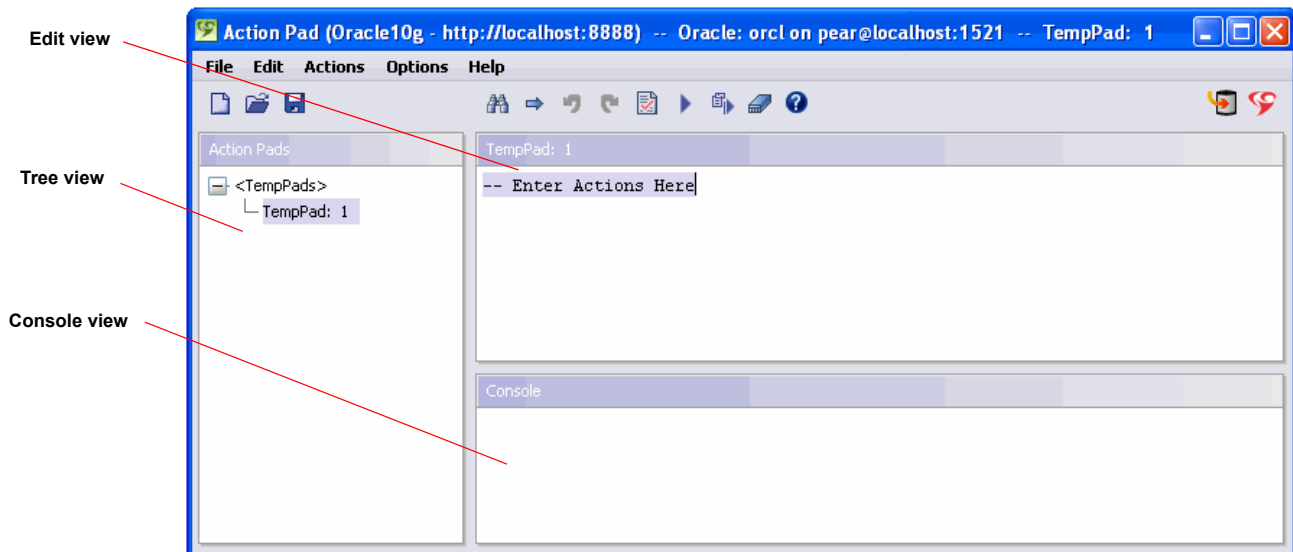
1. If you have not already done so, turn over and push the model you created in the previous chapter. See “[Turn over and push](#)” on page 25.

The exercises in this chapter utilize the SimpleModel schema created in “[Information modeling](#)” on page 9. The run-time repository must be loaded with this model, though the application server need not be running for you to work with it.

2. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Manager > Action Pad**.

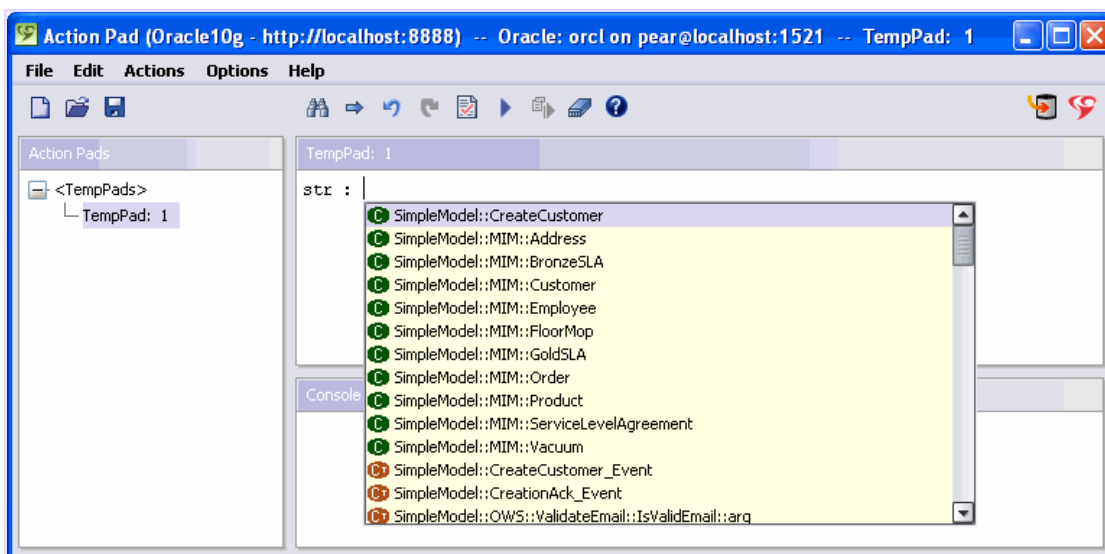
Cogility Action Pad opens with an empty action pad in the edit view. The pad is named **TempPad: 0** by default. A TempPad is an unsaved action pad. The comment line `-- Enter`

`Actions Here` appears in the edit view. Recall that comment lines are preceded with two dashes `--`.



3. In the edit view, select the line, `-- Enter Actions Here` and press the **Backspace** key.
4. Declare a String variable as follows:
 - a. Enter the following statement with a space on the end:


```
str :
```
 - b. Hold down the **Ctrl** key and press the space bar.
The code assist window displays.
 - c. In the code assist window, select `java::lang::String` and press the **Enter** key.

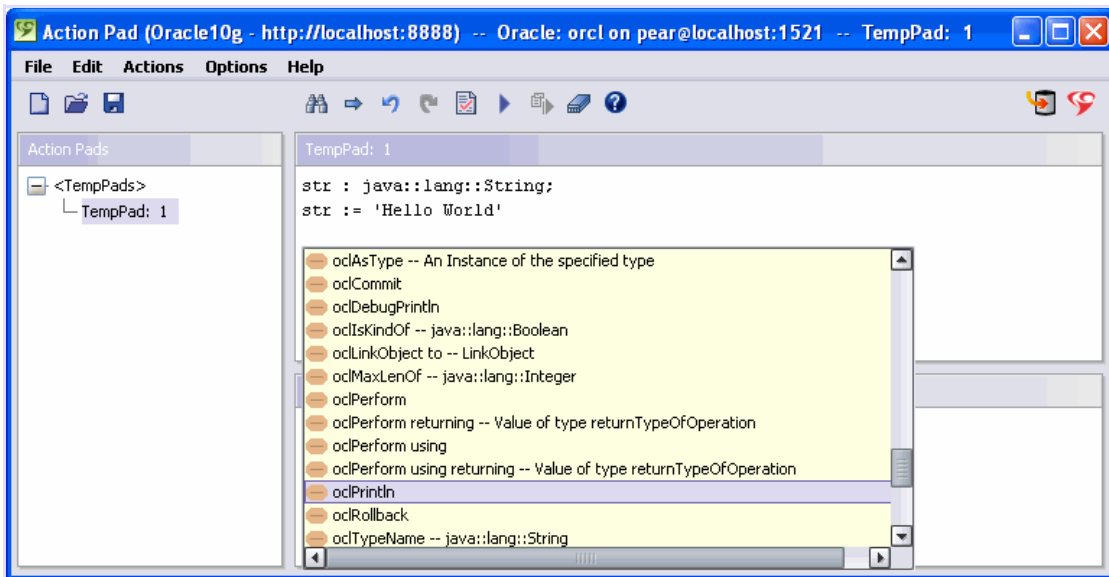


- d. Complete the line by entering a semicolon `;`.

This completes the first line of the “hello world” program described above. This first statement declares the `str` variable as a String.

5. Press **Enter** key to advance the cursor to the next line.

6. In the next line, assign the `str` variable to the string, `Hello World`.
Recall that variable assignment proceeds from right to left, that Strings are enclosed in single quotes, and that a colon with an equal sign `:=` designates the assignment.
For now, don't end the line with a semicolon `;` so you can see how Cogility Action Pad deals with compiler errors in the next section.
7. Press the **Enter** key.
8. Use the OCL keyword, `oclprintln()` and pass to it the `str` variable as follows:
 - e. Hold down the **Shift Ctrl** key and press the space bar.
 - f. Select the **oclPrintln** keyword, as shown below, and press the **Enter** key.




- g. Within the parentheses, select `expression` and replace it with `str`.
This completes the “hello world” program.

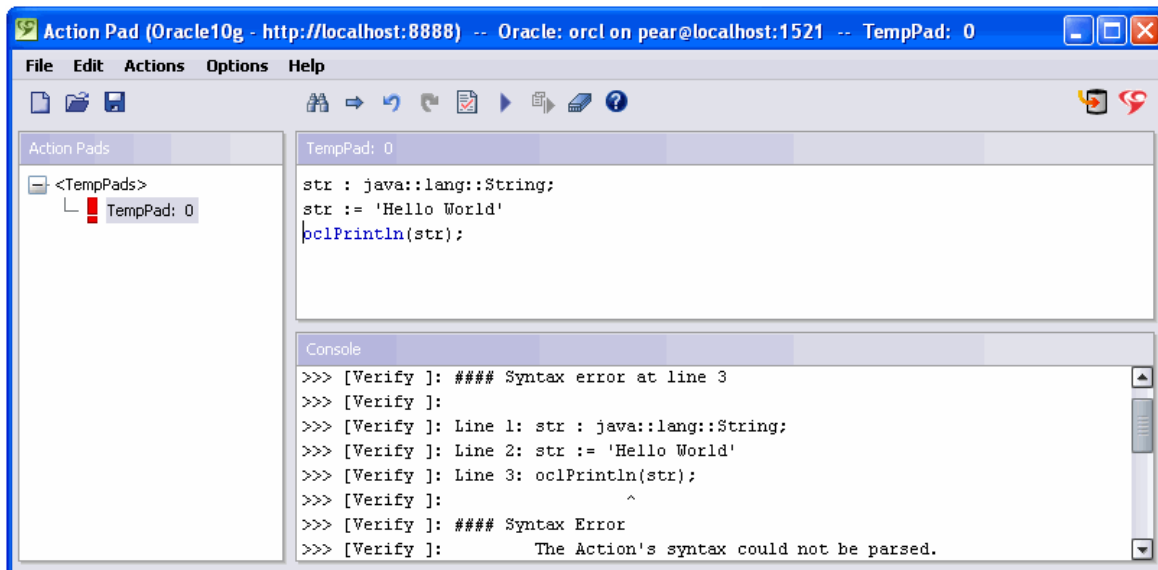
Compiling and executing

Cogility Action Pad includes both a compiler and run-time environment for action semantics scripts. Messages about compilation and execution appear in the console view.

To compile and execute action semantics:

1. In the tool bar, click the Run button  .
The compiler detects the syntax error and generates a failure message. Recall that you omitted the semicolon from the second line. Because of that omission, the compiler cannot

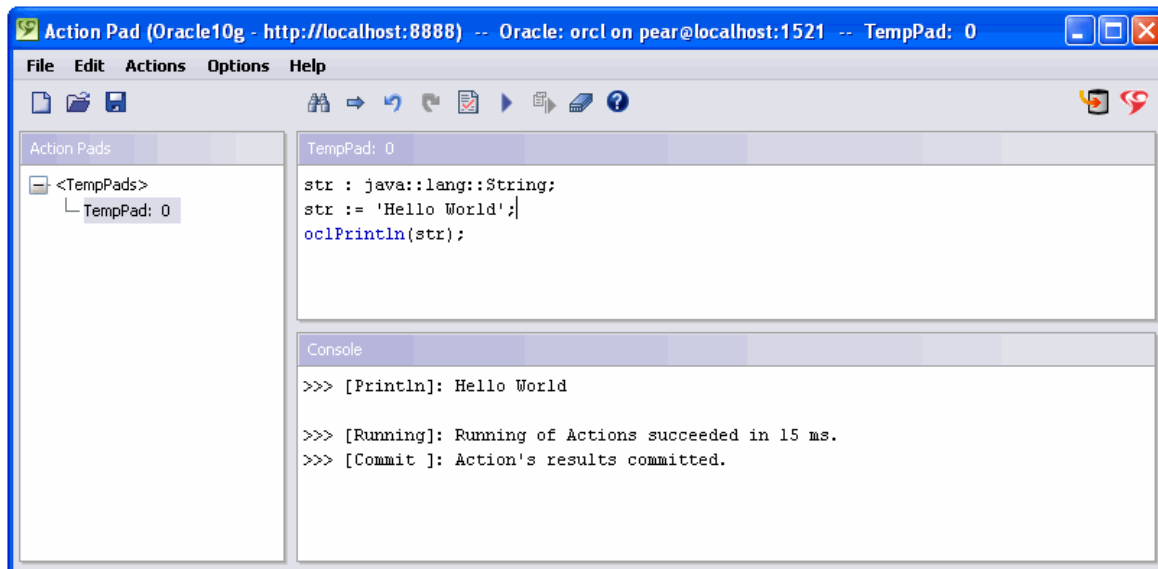
process the third line and generates a syntax error at the location marked by a caret (^) as shown below.



2. In the editor view, append a semicolon ; to the second line.

3. In the tool bar, click the Run button .

The script's output and a successful commit message appear in the console view.



Working with action pads

As mentioned earlier, a TempPad is an unsaved action pad. You can rename a TempPad and save it as a file which can then be reloaded. An action pad is a saved action semantics script file.

Renaming a TempPad

You can rename a TempPad. Each new pad that you create appears with its default name in the tree view. Renaming only changes the default name to something more useful, and the file is still a TempPad that will persist only as long as the current Cogility Action Pad session. You must save the TempPad as an action pad (see below) in order to work with it again in a new session.

To rename a TempPad:

1. In the tree view, select the **TempPad: 0**.
2. Right-click and select **Rename TempPad**.
3. In the dialog, enter **HelloWorld** and click **OK**.

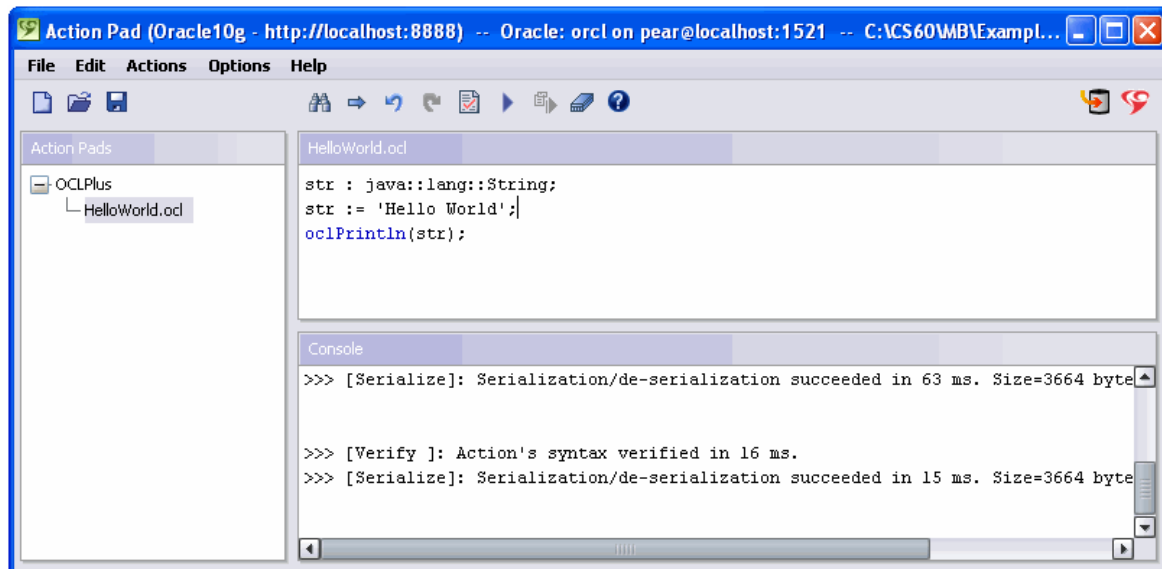
Saving files

You can save your work in a text file to be reloaded later.

To save an action pad:

1. In the tree view, select the **HelloWorld** TempPad.
2. Right-click and select **Save As**.
3. Navigate to the location where you want to save the file.
4. In the **File Name** field, enter a name and click **Save**.

The location of the file appears at the top of the tree, and the saved action pad is appended with a .ocl extension. In the following figure, the HelloWorld.ocl action pad is saved to the desktop.



Opening files

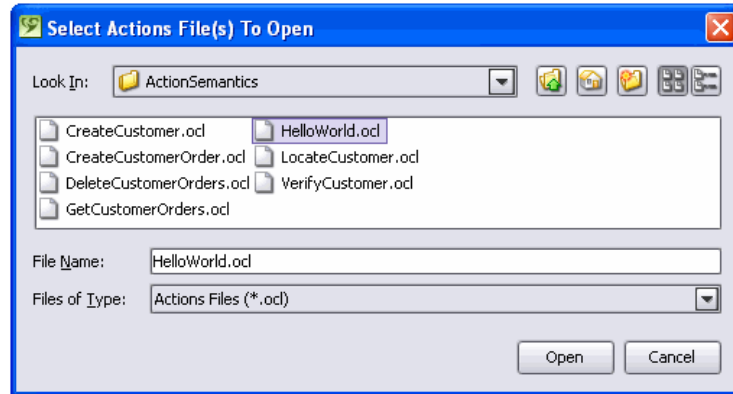
The preceding example, and all of the examples described in this chapter are available in text files located in the %DCHOME%\Examples\SimpleModel\Tutorial\ActionSemantics folder. You can open these files in Cogility Action Pad to work with the examples.

To load an example file:

1. From the **File** menu, select **Open**.

2. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial\ActionSemantics folder, select the file and click **Open**.

The files are appended with .ocl suffix, but they are readable text files.



Object instance

In the previous chapter, “[Information modeling](#)” on page 9 you created a Customer class in the SimpleModel schema. In this exercise you create an instance of that class with action semantics.

Action semantics uses the keyword `new` to create an object of a class. The syntax reads as follows:

```
new <qualifier>::<qualifier>::<class>(<attr_name>:<value>,
<attr_name>:<value>, ...);
```

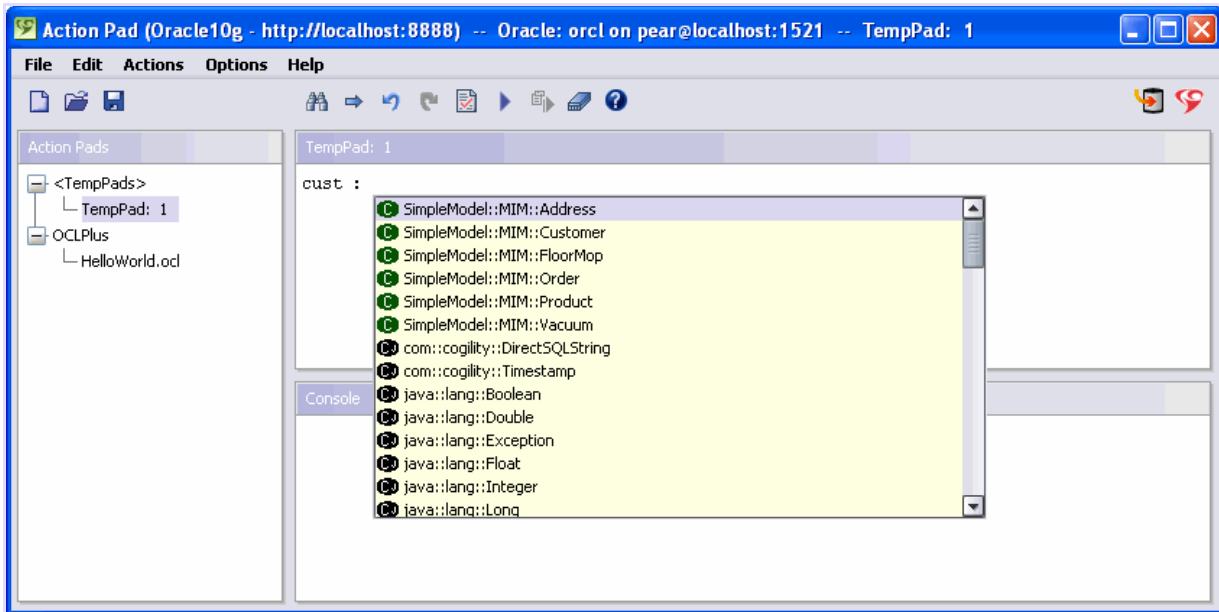
Usually you create the new object and assign it to a variable in the same line. The variable is first declared as a fully-qualified type of the object’s class from the model you have pushed into execution.

To create a Customer object:

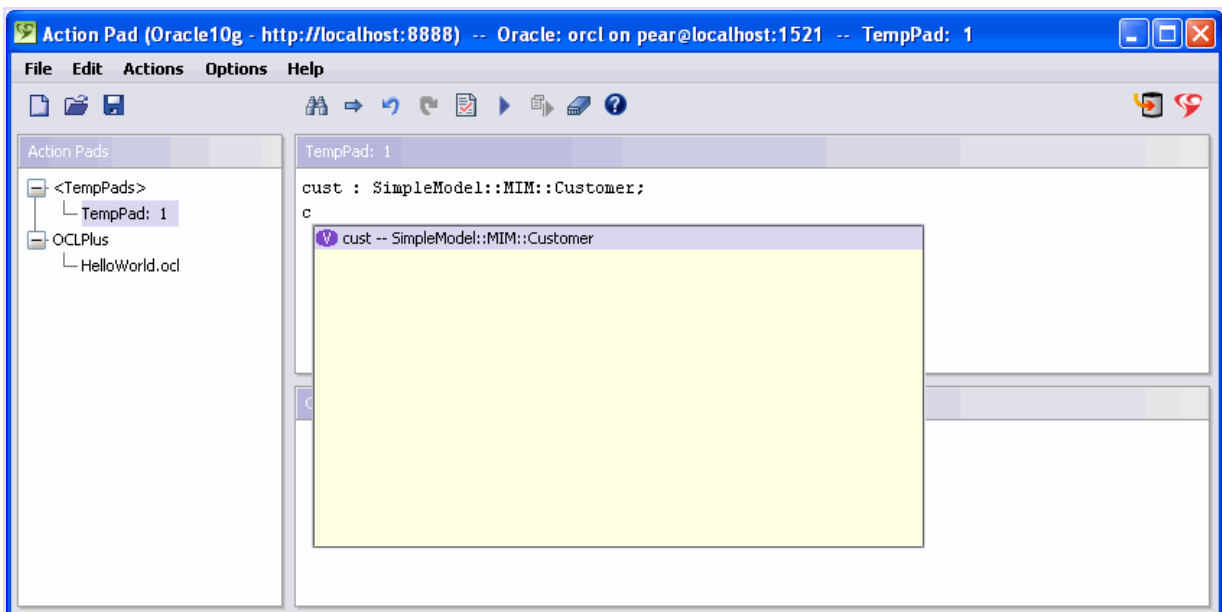
1. In Cogility Action Pad’s tool bar, click the **New TempPad** button .
2. In the editor view, select the line, `-- Enter Actions Here` and press the **Backspace** key.
3. Declare a Customer variable as follows:
 - a. Enter the following statement with a space on the end:


```
cust :
```
 - b. Hold down the **Ctrl** key and press the space bar.
The code assist window displays.

- c. In the code assist window, select **SimpleModel::MIM::Customer** and press the **Enter** key.



- d. Finish the line with a semicolon ; and advance the cursor to the next line with the **Enter** key.
4. Create a new Customer object and assign it to the variable as follows:
- Enter a **c**, hold down the **Ctrl** key and press the space bar.
The code assist window appears with a listing of all known variables that begin with the letter **c**.
 - With **cust -- SimpleModel::MIM::Customer** selected, press **Enter**.



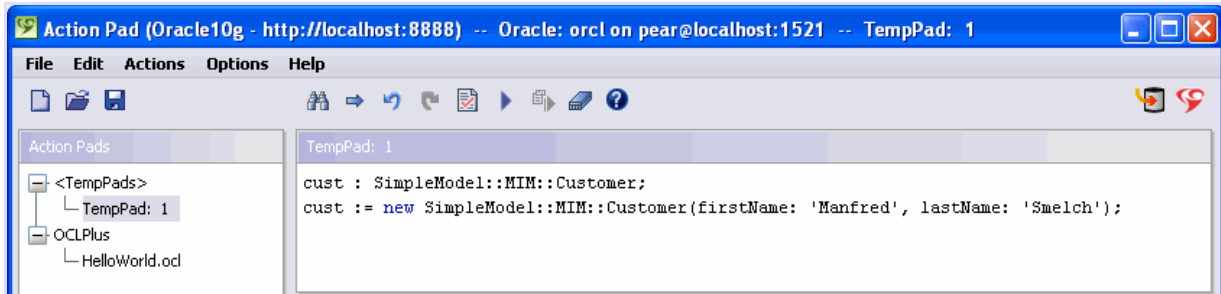
- c. Following **cust** and a space, enter the assignment operator (**:=**), a space, the **new** keyword and a space.
- d. Hold down the **Ctrl** key and press the space bar to bring up code assist again.

- e. Select **SimpleModel::MIM::Customer** and press **Enter**.

From the SimpleModel, you know that the Customer class has two attributes, a `firstName` and a `lastName`, both Strings. These are passed as parameters to the creation method named for the class.

- f. Within parentheses, assign the `firstName` and `lastName` parameters to some String value and finish the line with a semicolon `;`.

You can use code assist after the semicolon `(` and then again after the comma `,` to lookup the variables for the class. Your code should look like the following:



5. In the tool bar, click the Run button .

A message indicating successful execution should appear.

The run-time repository now includes a Customer named Manfred Smelch (or whatever you named your Customer). In the next exercise you learn how to retrieve that Customer.

The script for the example above is located as

`%DCHOME%\Examples\SimpleModel\Tutorial\ActionSemantics\CreateCustomer.ocf`.


Locating objects

You can now write action semantics to retrieve the Customer you created in the last exercise. The keyword `locate` accomplishes this; here is its syntax:

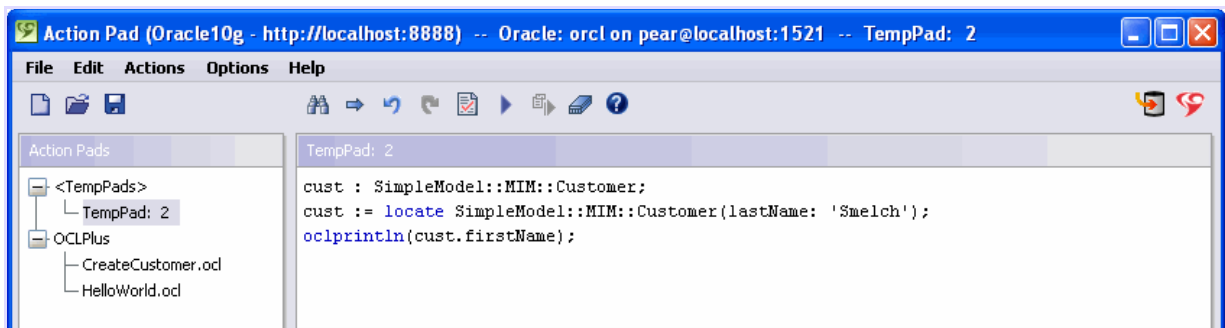
```
locate <classname>(<attr_name>:<value>,<attr_name>:<value>, ...);
```


Usually you locate the object and assign it to a variable in the same line. The variable is first declared as a fully-qualified type of the object's class from the model you have pushed into execution.

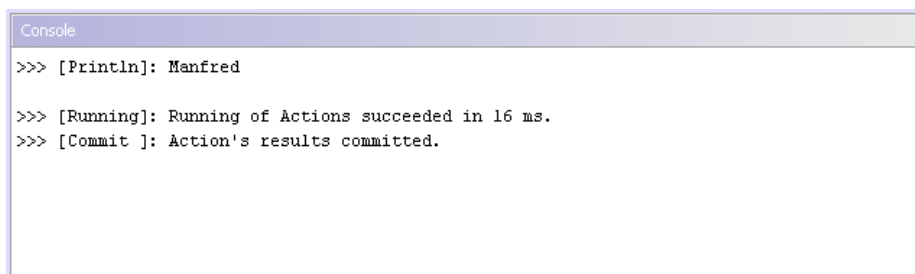
To locate a Customer object:

1. In Cogility Action Pad's tool bar, click the **New TempPad** button .
2. In the edit view, select the line, `-- Enter Actions Here` and press the **Backspace** key.
3. Declare a Customer variable as follows:
 - a. Enter the following statement with a space on the end:
`cust :`
 - b. Hold down the **Ctrl** key and press the space bar.
The code assist window displays.
 - c. In the code assist window, select **SimpleModel::MIM::Customer** and press the **Enter** key.
 - d. Finish the line with a semicolon `;` and advance the cursor to the next line with the **Enter** key.
4. Locate an existing Customer object and assign it to the variable as follows:

- a. Enter a `c`, hold down the **Ctrl** key and press the space bar.
The code assist window appears with a listing of all known variables that begin with the letter `c`.
 - b. With `cust -- SimpleModel::MIM::Customer` selected, press **Enter**.
 - c. Following `cust` and a space, enter the assignment operator (`:=`), the `locate` keyword and a space.
 - d. Hold down the **Ctrl** key and press the space bar to bring up code assist again.
 - e. Select `SimpleModel::MIM::Customer` and press **Enter**.
 - f. Enter an open parentheses.
 - g. Hold down the **Ctrl** key and press the space bar to bring up code assist again.
 - h. Select `lastName` and press **Enter**.
 - i. Following the named attribute `lastName`, enter a colon `:`, define the search string `'Smelch'`, and finally enter a closing parentheses.
5. Enter the keyword, `oclPrintln()` and pass to it the `cust` variable as follows:
- a. Enter the keyword, `oclPrintln`.
 - b. From within the parentheses, select `expression` and replace it with `cust`. (the `cust` variable and the dot operator).
Object attributes are qualified by the dot operator. You can use code assist to retrieve the attributes.
 - c. Hold down the **Ctrl** key and press the space bar.
The code assist window displays.
 - d. In the code assist window, select `firstName -- java::lang::String` and press the **Enter** key.
- Your code should look like the following:



6. In the tool bar, click the Run button  .
The output appears as follows:



The script for the example above is located as
%DCHOME%\Examples\SimpleModel\Tutorial\ActionSemantics\LocateCustomer.ocl.

Flow control

Often in your scripts you need to control the program flow with an if/then/else statement. In this exercise you use such a statement to verify the existence of a Customer object. The syntax for an if/then/else clause is as follows:

```
if <conditional_expression> then
    <statements>
else
    <statements>
endif;
```

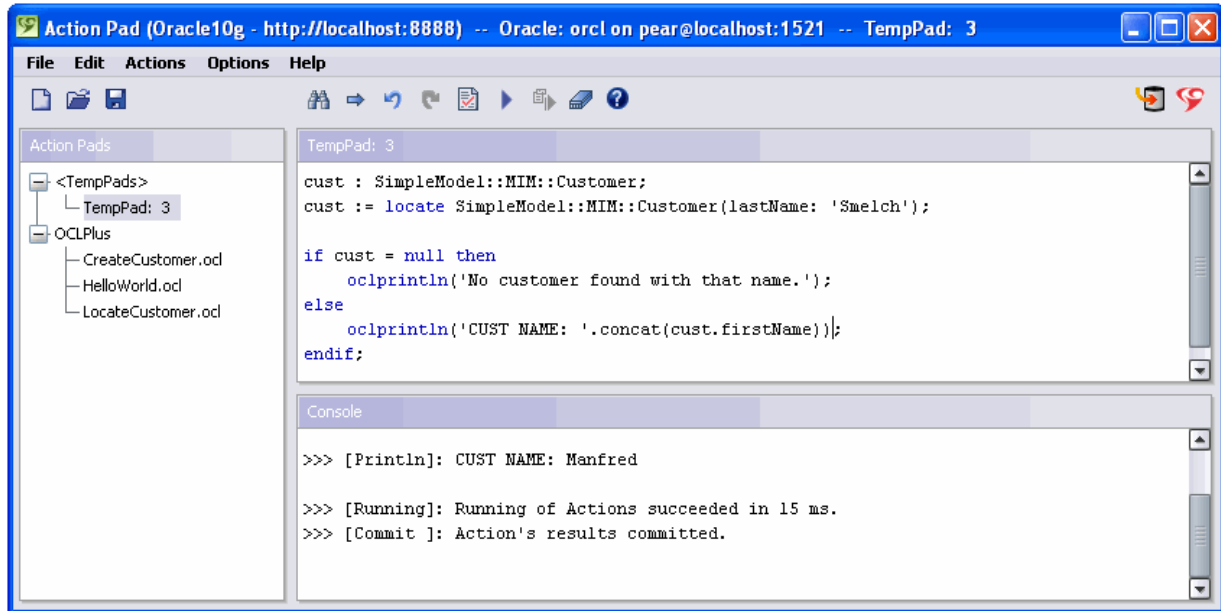
A conditional expression uses a relational operator such as the equal sign `=`. For a complete description of relational operators, see the guide, *Using Actions in Cogility Studio*.

For this exercise, you reuse the code you created in “[Locating objects](#)” on page 35.

To impose a flow control statement on output:

1. Open Cogility Action Pad with the script completed in the previous exercise, “[Locating objects](#)” on page 35.
If you haven’t changed Cogility Action Pad since the last exercise, that code is already loaded.
2. Place two empty lines between the second and last lines of code.
3. Place the cursor above the last line and enter a conditional expression to test if the `cust` variable value is equal to `null`.
4. Copy the last line to a new line below it as follows:
 - a. Select the last line, hold down the **Ctrl** key and press **c** to copy the line.
 - b. Place the cursor at the end of the line and advance the cursor to a new line with the **Enter** key.
 - c. Hold down the **Ctrl** key and press **v** to paste the line.
5. Indent the next to last line three spaces; in the parentheses of the `oclPrintln` expression, select `cust.firstName` and replace it with the following:
`'No customer found with that name.'`
6. Place the cursor at the end of the line, advance the cursor to a new line with the **Enter** key and type `else`.
7. Indent the last line three spaces; in the parentheses of the `Println` expression, before `cust.firstName`, insert the following:
`'CUST NAME: '.concat(`
8. Place a right parenthesis `)` just after `cust.firstName`.
9. Place the cursor at the end of the line, advance the cursor to a new line with the **Enter** key and type `endif;`.

10. In the tool bar, click the Run button  .



The script for the example above is located as
%DCHOME%\Examples\SimpleModel\Tutorial\ActionSemantics\VerifyCustomer.ocl.

Creating associations

In the SimpleModel class diagram, CustomerModel that you created in [“Information modeling” on page 9](#) there is an association between the Customer and Order classes. An association defines the relationship between two classes; in this case, one Customer may have many Orders. In this exercise, you create an Order object and associate it with an existing Customer object. The syntax for creating an association is as follows:

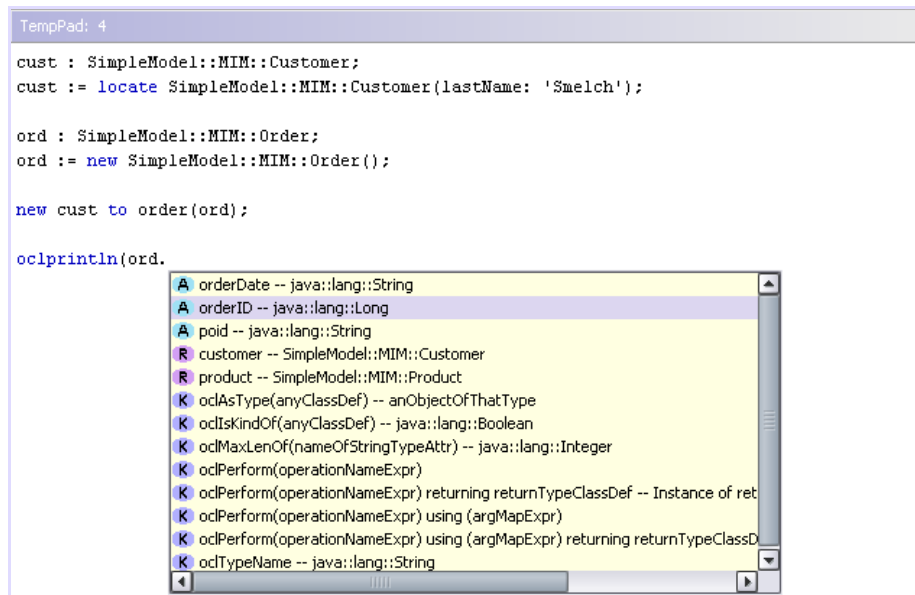
```
new <owningObject> to <targetRole>(<ownedObject>);
```

The association owning object is specified first, then the target role of the owned object and the owned object itself in parentheses.

To create an association between a Customer and an Order:

1. Create a new TempPad, declare a Customer variable and locate an existing Customer as you did in steps 1 through 4 of [“Locating objects” on page 35](#).
2. Declare a variable of type Order and create a new Order object using the same process as in steps 3 through 4 of [“Object instance” on page 33](#).
Use `ord` for the name of the Order variable. Do not specify any parameters for the creation method.
3. In a new line, create an association from the Customer (`cust`) to the target role (`order`) of the Order object (`ord`).
4. In a new line, use `oclPrintln()` to print out the Order's `orderID` value using the same process as in step 5 of [“Locating objects” on page 35](#).

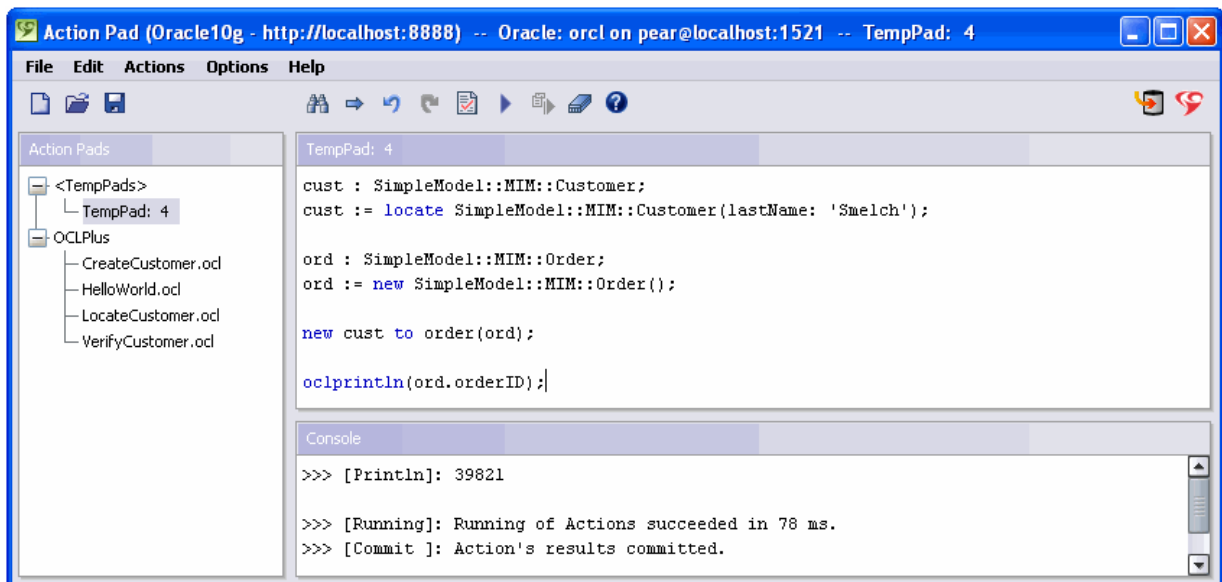
You can use code assist to retrieve the `orderId` attribute of the `Order` object variable as shown below.



Recall that the `orderId` attribute is sequenceable, meaning that its value is automatically generated by the database.

5. In the tool bar, click the Run button .

In the figure below, the `orderId` 1041 is printed to standard output. The value may be different on your system.



The script for the example above is located as
`%DCHOME%\Examples\SimpleModel\Tutorial\ActionSemantics\CreateCustomerOrder.ocl`.

Working with collections

Many objects may be held in a collection. For example, a Customer may have many Orders, and those Orders associated with the Customer comprise a Collection. A Collection is one of many Java classes that you can use in action semantics.

The arrow operator (\rightarrow) combines an iteration over a collection (`java.lang.Collection` or any subclass) with an operation on the members of the collection. To iterate over a collection of associated objects, you specify the owning object with its target role for the owned object(s) in the collection then iterate over each owned object held in a variable. The syntax is as follows:

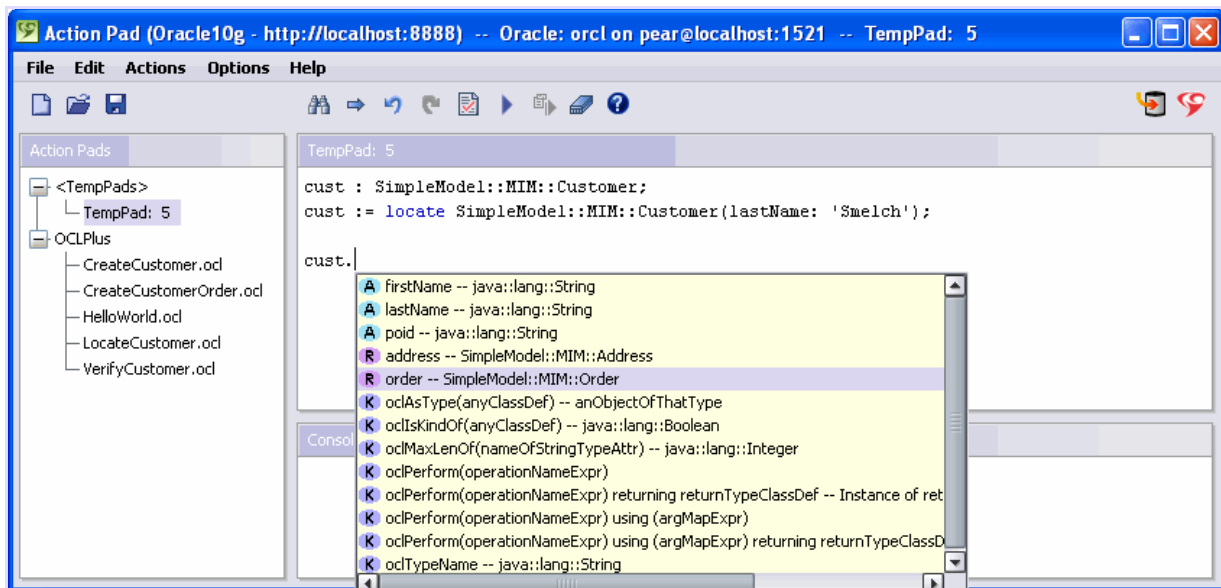
```
aColl -> iterate (varName : classSpec | <statements>);
```

In the line above, `aColl` is the Collection object over which you want to iterate, `iterate` is the keyword that performs an iteration operation, `varName` holds a the current member of the Collection, the `<statements>` evaluate the members for the operations, and the `classSpec` describes the fully-qualified classname of the members of the Collection. For more information about working with collections, see the guide, *Using Actions in Cogility Studio*.

To create a collection of Customer Orders:

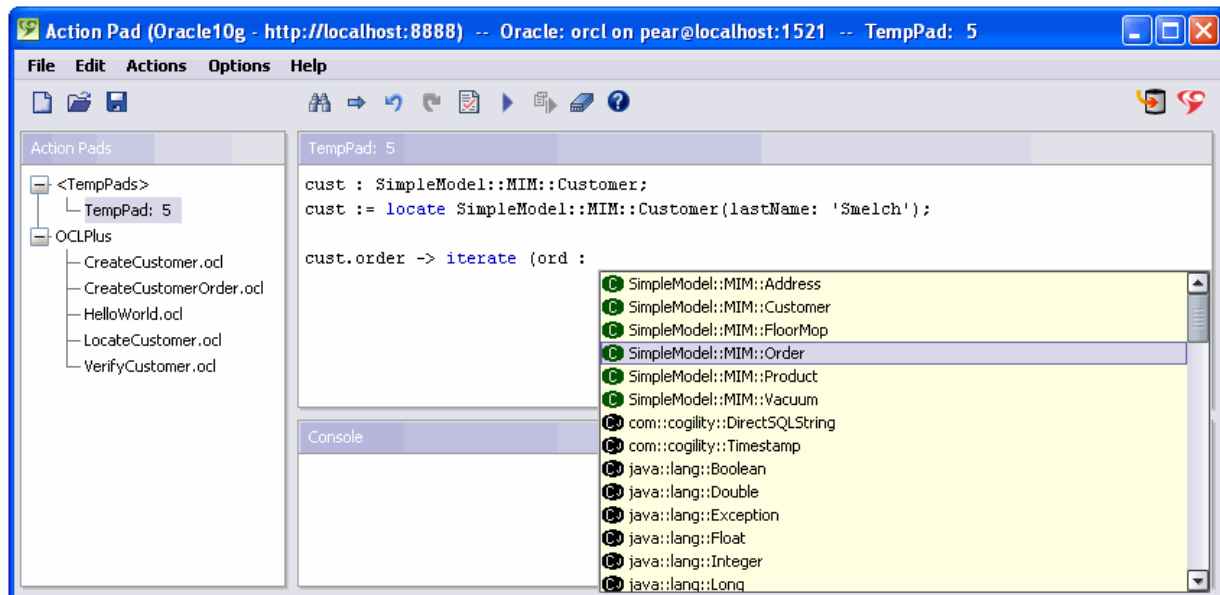
1. Create a new TempPad, declare a Customer variable and locate an existing Customer as you did in steps 1 through 4 of “Locating objects” on page 35.
2. In a new line, iterate over a Collection of Customer Orders and print the `orderId` to standard output.
 - a. Specify a Collection of Customer Orders using the existing association.

You can use code assist to find the target role, as shown below. Note that the expression `cust.order` evaluates to a Collection type so you don’t have to declare a variable of that type beforehand.



- b. Iterate over the Collection, specifying the Order class.

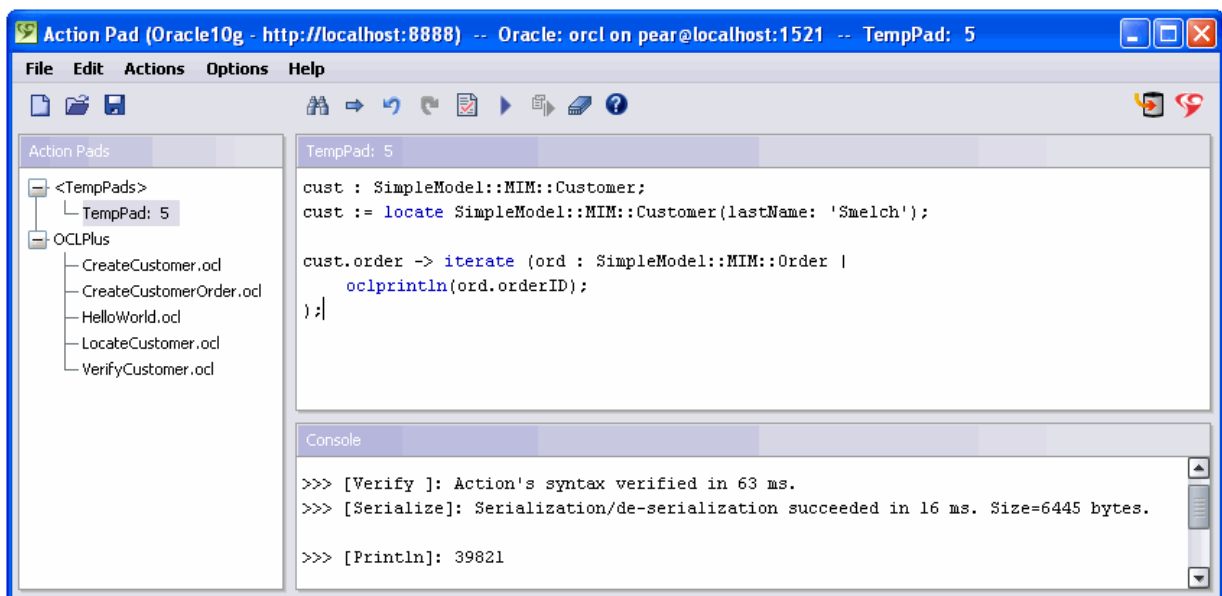
Use the variable, `ord` for the orders. Use code assist to find its class specification, as shown below. This is effectively defining the `ord` variable.



- c. Use `oclPrintln()` and pass to it the `orderId`.

Use code assist to retrieve the `orderId` as you did in step 4 of "Creating associations" on page 38.

3. In the tool bar, click the Run button .



The script for the example above is located as
`%DCHOME%\Examples\SimpleModel\Tutorial\ActionSemantics\GetCustomerOrders.ocl`.



In the chapter, “[Information modeling](#)” on page 9, you started creating a basic model with a Customer class. In this chapter, you add model artifacts that use that class information to create a new Customer object in the run-time repository.


The customer creation scenario starts with a hypothetical external application. It is hypothetical because the application itself isn’t required, only the messages it sends to your model’s enterprise application are required, and you simulate those with Cogility Message Traffic Viewer. In this scenario, the external application sends a message to the enterprise application notifying it of a new customer. The enterprise application then converts that new customer message data into an event that starts a business process. The business process creates the new Customer object in the run-time repository.

Following a new Customer object creation, the new customer data can then be communicated to all of the applications integrated with the enterprise application. For the first part of this tutorial you are integrating only one application, so your enterprise application communicates with only that one external application. This chapter shows how an external application using Java Messaging Service (JMS) communicates with the model’s enterprise application *inbound* to effect data change in the run-time repository.

Importing the existing model

In a previous chapter, you created a model called SimpleModel. The steps in this chapter start with that model and add more features. If you do not have that version of the model loaded in Cogility Modeler, you can import it with the following procedure. If you are continuing from “[Information modeling](#)” on page 9, skip to “[Creating the event artifact](#)” on page 44.

To import the existing model:

1. If you have loaded in Cogility Modeler a model other than the SimpleModel created in “[Information modeling](#)” on page 9, close Cogility Modeler and follow the steps under “[Loading the model repository](#)” on page 11.
2. If Cogility Modeler is not running, follow the steps under “[Logging in](#)” on page 12.
3. In Cogility Modeler, from the **Selection** menu, select **Import/Export > Import from Single File**.
4. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial directory, select **SimpleModel_Chapter1.cog** and click **Open**.
%DCHOME% is the directory where you installed Cogility Studio.
5. In the dialog that prompts you, click **Yes** to make the model visible in Cogility Modeler.
6. In the tree view, select the **SimpleModel** model container and click the **Place Model Under CM** button .

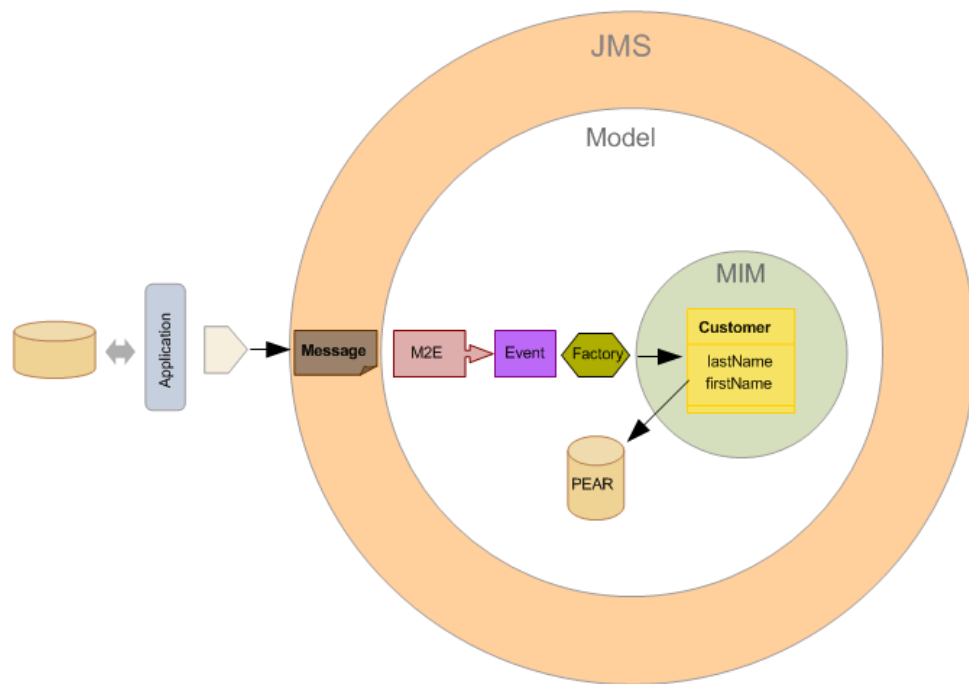
You are now ready to follow the procedures of this chapter.

Message model

This inbound communication starts with a JMS message from the external application. The message includes new customer data. That message and its data must be converted into a UML event that models the “event” of a new customer being created. The event contains the data in the format needed to work with it in the model.

Messages, events and conversions make up the message model portion of your integration model. See the guide, *Modeling with Cogility Studio* for more information about message conversion and events.

The figure below illustrates inbound communication, the conversion process and the new Customer object creation. A message to event (M2E) conversion takes the message with the new customer data and converts it into an event. A factory model artifact takes the event data and creates a new Customer.



You need to develop the event and M2E conversion artifacts to complete this process. You start with the event.

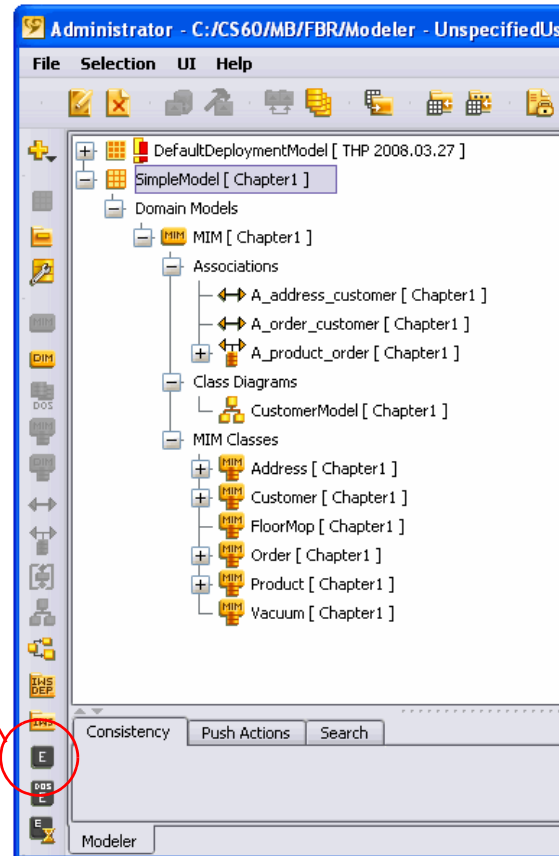
Creating the event artifact

The event artifact contains the new customer data and makes it available to the business process that creates the new Customer. Just as the Customer class defines two attributes, the new customer event artifact should define these same attributes so that it may pass them on to the business process that creates the new Customer.

To create the event artifact:

1. In Cogility Modeler's tree view, select the **SimpleModel** model container and click the **Add an Event** button .


Click the **Add an Event** button



2. In the content view, under the **Event** tab, in the **Name** field, enter **CreateCustomer_Event**.
3. For **Attributes**, click **New**.
 - a. In the dialog window that appears, for **Name**, enter **firstName**.
 - b. For **Type**, from the drop-down menu select **String** and click **OK**.
In the content view, the **firstName** is selected and the **Attribute Definition** tab displays.
4. For **Attributes**, click **New**.
 - a. In the dialog window that appears, for **Name**, enter **lastName**.
 - b. For **Type**, from the drop-down menu select **String** and click **OK**.
In the content view, the **lastName** is selected and the **Attribute Definition** tab displays.

Your CreateCustomer_Event should look like the following:


The screenshot shows the 'Event' configuration window in Cogility Modeler. The 'Name' field is 'CreateCustomer_Event'. The 'Is Internal Event' checkbox is unchecked. The 'Abstract' checkbox is unchecked. The 'Superclass' is 'Element [MAL 2004.04.10]'. The 'Attributes' section shows two attributes: 'firstName' and 'lastName', both with type 'String' and size '0'. The 'Comments' tab is selected, showing the 'Name' field with 'firstName' and the 'Type' field with 'String [MAL 2004.04.10]'. The 'Size' field is '0' and the 'Is Required' checkbox is unchecked.

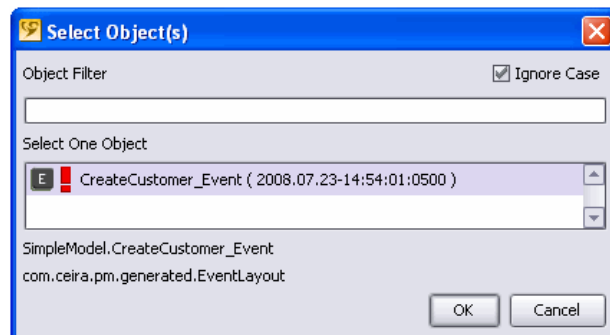
Notice that your new CreateCustomer_Event artifact is shown in the tree view with a red exclamation mark  indicating an inconsistency. The consistency view (in the bottom pane) of Cogility Modeler explains that the event artifact must be associated with a message conversion artifact. You create that artifact next.

Creating the conversion artifact

In order to get the new customer message data into the event, it must be converted with a message to event (M2E) conversion.

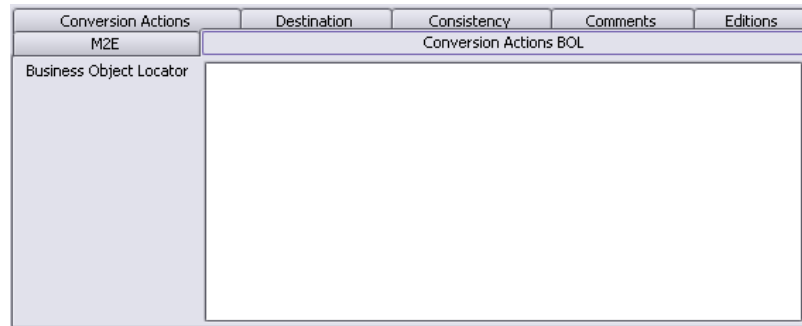
To create the conversion artifact:

1. In tree view, select the **SimpleModel** model container and click the **Add an M2E Conversion** button .
2. In the content view, under the **M2E** tab, in the **Name** field, enter **CreateCustomer_M2E**.
3. Above the **Event** field, click the **Set** button, select **CreateCustomer_Event** and click **OK**.



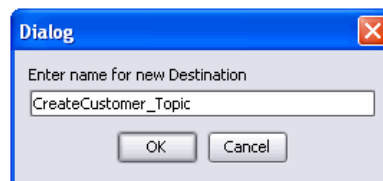
This associates the conversion artifact with the event artifact, thus removing the inconsistency on the event artifact described earlier.

4. Click the **Conversion Actions** tab and leave checked the **Default Conversion** checkbox.



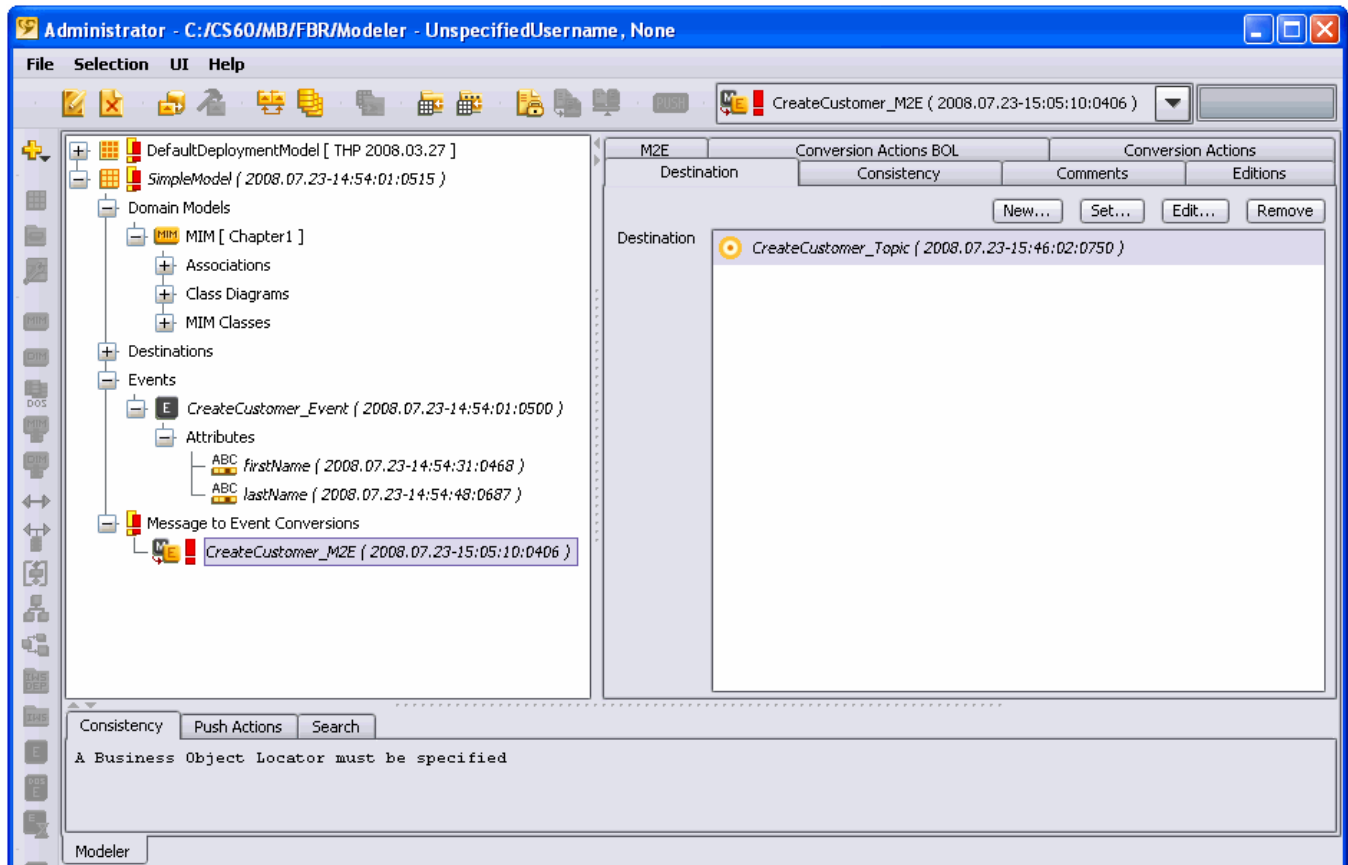
By leaving Default Conversion checked, you are setting the conversion artifact to look in the incoming JMS message for attribute values as defined by the event attribute names. Otherwise you would define the conversion in either the Pre Conversion or Post Conversion fields.

5. Click the **Destination** tab, click **New**, enter **CreateCustomer_Topic** and click **OK**.



Your model must know for which message to listen. Messages are published on a topic (or for a destination) that describes the message. The hypothetical external application

publishes a message on the topic (destination), CreateCustomer_Topic. Your model should now look like the following:



The CreateCustomer_M2E has an inconsistency, a business object locator is not specified. To eliminate this inconsistency, you have to specify where to send the event in the M2E conversion's business object locator. You do that in the next set of steps.

Object creation

To complete the inbound communication with the model's enterprise application at run time, you need to create a new Customer object. In the first chapter, ["Information modeling" on page 9](#), you created a class for the new Customer object. In this section you use the Customer class information to create a new Customer object and you further describe its behavior.

Creating an instance with a factory

At run time, the model's enterprise application receives new customer information, and the conversion object that communicates the new customer information requires an object that can create a Customer object. According to the modeling rules, to create a Customer named John Shiu, the CreateCustomer_M2E must locate an object that can create a Customer named John Shiu.


Usually, creating a Customer would be a behavior of the Customer object itself. But that only presents a chicken-and-egg problem because the Customer object does not yet exist. To solve the problem, Cogility Modeler employs a factory object. A factory object is like a business object (such as

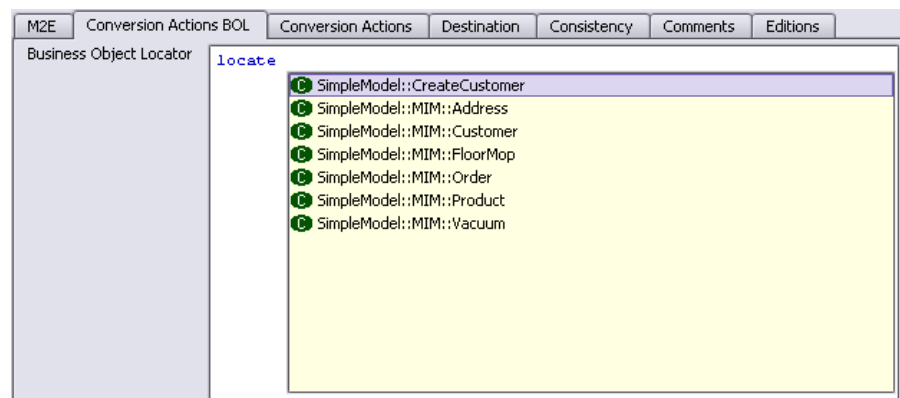
a Customer), but it cannot have attributes or associations, nor may it have a superclass; it may have only behavior, usually that behavior is to create an instance of a class.

When you push your model, the factory object is created, so you automatically have an object that can create the Customer. Specific conversions designate specific factories which initialize specific objects. At run time, when the model's enterprise application receives a message, it converts the message to an event that is sent to the factory object to create the Customer.

For more information about factories, see the guide, *Modeling with Cogility Studio*.

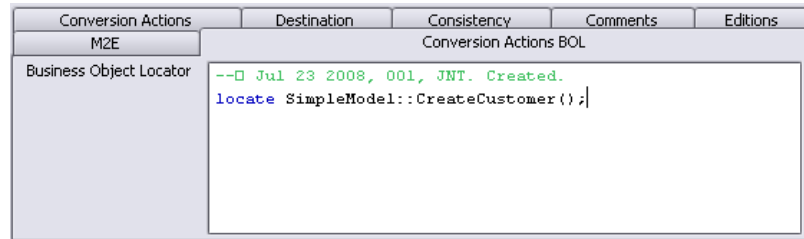
To create a factory artifact:

1. Select the **SimpleModel** model container and click the **Add a Factory** button .
2. In the content view, under the **Factory** tab, in the **Name** field, enter **CreateCustomer**.
Now you can designate this factory in the business object locator for the CreateCustomer_M2E and eliminate the inconsistency.
3. In the tree view, under **Message to Event Conversions**, select the **CreateCustomer_M2E**.
4. In the **Conversion Actions BOL** tab, in the **Business Object Locator** field, enter **locate** and a space.
5. Hold down the **Ctrl** key and press the space bar to open the code assist window.
6. Select the **SimpleModel::CreateCustomer** factory artifact and press **Enter**.



7. Specify the object's creation method by appending parentheses **()** to the line.
8. Finish the line with a semicolon **;** and click in the Pre Conversion field below.
This will bring up a warning about defining a user name.
9. Click **OK** to close the warning.
10. In the dialog, enter a username and click **OK**.
Whenever you edit an action semantics field in Cogility Modeler, comments for that edition are added automatically. In your CustomConfigurations.txt file, you can set the information that will appear in these comments. See the guide, *Modeling with Cogility Studio* for more information. Since you haven't configured the comments information, Cogility Modeler prompts you for a user name and change request number in the comments it enters.
11. In the dialog, enter a change request number and click **OK**.

Your Business Object Locator field should look like the following. The user comments include the date, the change request number, the user name and the default comment, "Created." for a new edition.



Also, the user name and change request number appear on the title bar of the Cogility Modeler's Window frame, following the login name (Administrator) and the path to the authoring repository for the model in the current context.

When you click out of the Business Object Locator field, the action semantics you entered compile and the inconsistency is eliminated.

Behavior model

The previous steps created the first part of the *data path* between the hypothetical external application and the run-time repository. In this case, the data path follows the means by which the new Customer object is created on the run-time repository. The data path starts with message conversion and ends with object creation.

In this section you complete the behavior model portion of your integration model. The behavior model describes the business process that completes the transfer of data across the data path. See the guide, *Modeling with Cogility Studio* for more information.

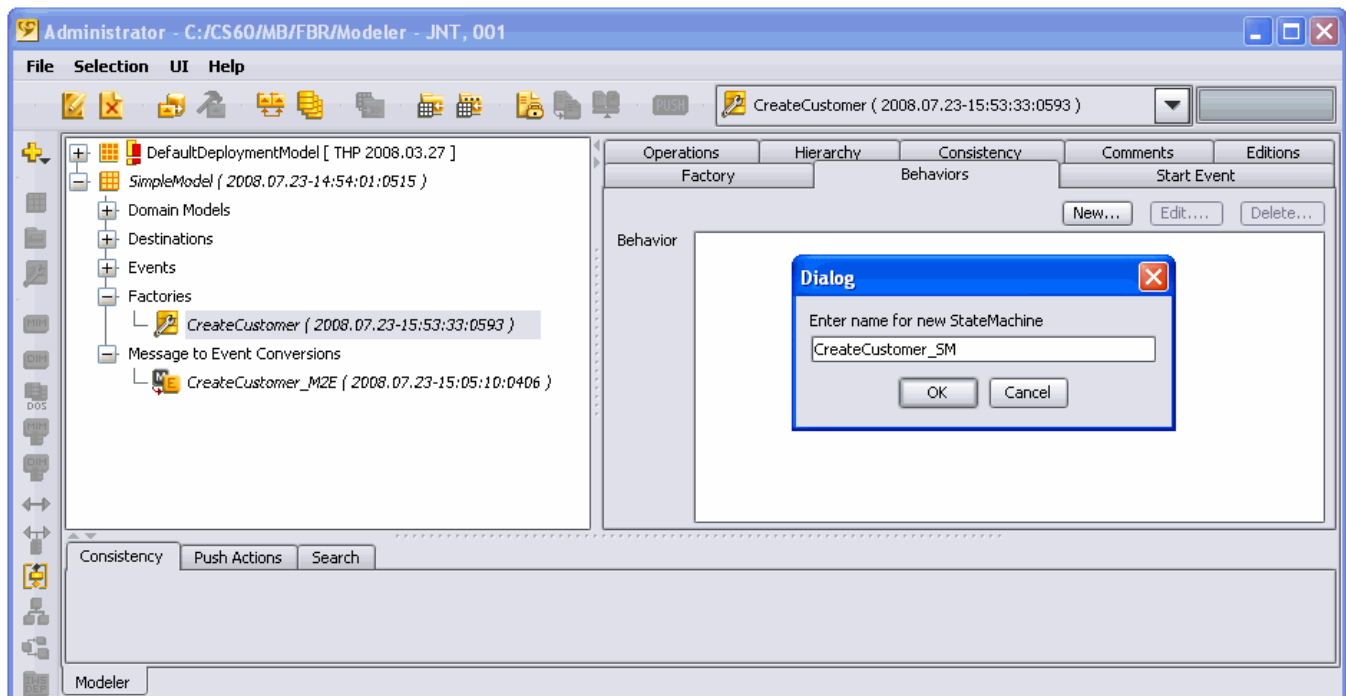
State machine

As mentioned earlier, the factory artifact may have only behavior, and that behavior, in the case of our CreateCustomer factory artifact, is to create a Customer object. In Cogility Modeler, object behavior is defined with a state machine. Based on a UML state diagram, a state machine describes object behavior as a series of states in a business process.

To create a state machine:

1. In the tree view, select the **CreateCustomer** factory artifact and, in the content view, click the **Behaviors** tab.

2. Click the **New** button, and in the dialog that appears, enter **CreateCustomer_SM** and click **OK**.

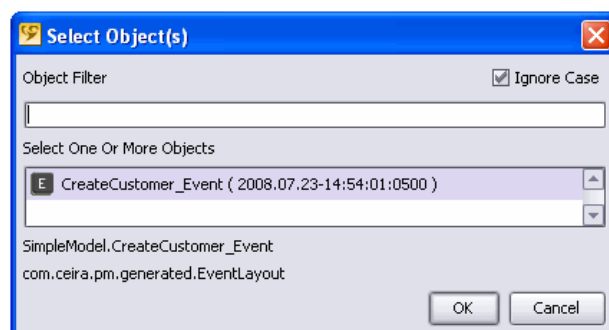


Start event

Every behavior, and the state machine that defines it, must be instantiated by an event. In this case, the **CreateCustomer_Event** defined earlier is used to trigger the **CreateCustomer** factory's **CreateCustomer_SM** behavior.

To associate the start event with the factory's behavior:

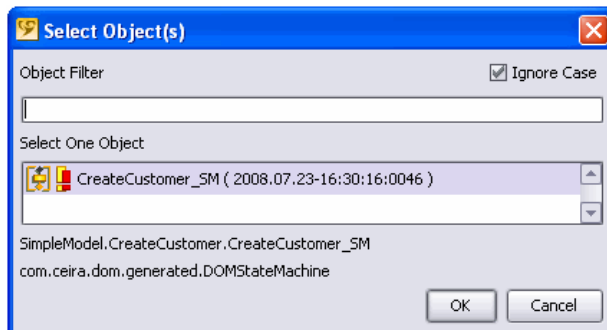
1. In the content view, click the **Start Event** tab and click the **Add** button.
2. Select the **CreateCustomer_Event** and click **OK**.



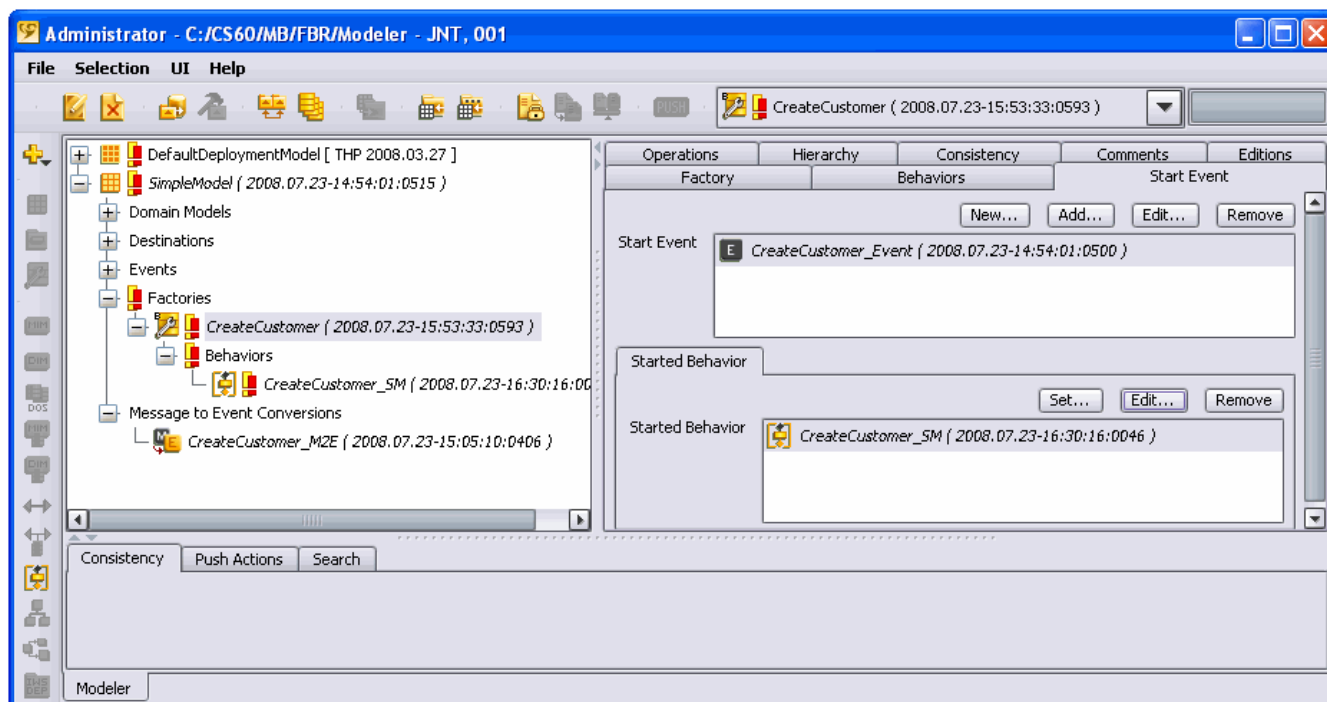
This adds the start event to the artifact, but it must still be associated with a particular behavior.

3. In the content view, select the **CreateCustomer_Event** artifact, and under the **Started Behavior** tab that appears, click **Set**.

4. In the dialog that appears, select **CreateCustomer_SM** and click **OK**.



Your model should now look like the following:



States and transitions

The next task in defining the CreateCustomer factory artifact's behavior is drawing the UML state diagram that describes the state machine. A state machine consists of states that describe the object at specific points in its lifecycle and transitions between states. Either of these components may contain business logic that performs some task in the business process.

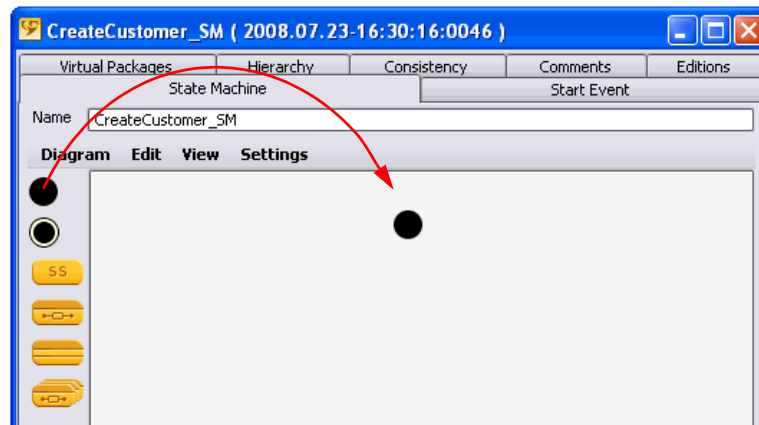
The CreateCustomer_SM state machine has only one state with business logic plus the two transitions into and out of the state.



To create the state diagram:

1. In the content view, under the **Behaviors** tab, select the **CreateCustomer_SM** state machine and click the **Edit** button.

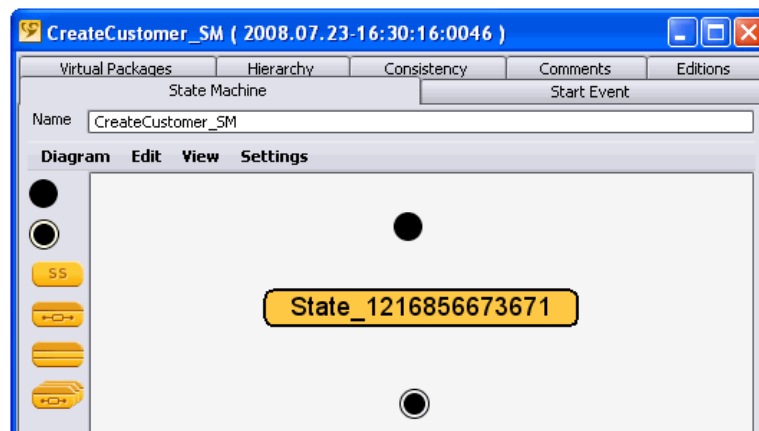
The state machine editor opens in a new window. On the left side, in the icon bar are the state icons representing the types of states. You place these in your diagram by dragging them into the editor window.

2. From the icon bar, select the **Initial State** icon , drag it into the editor window and drop it.



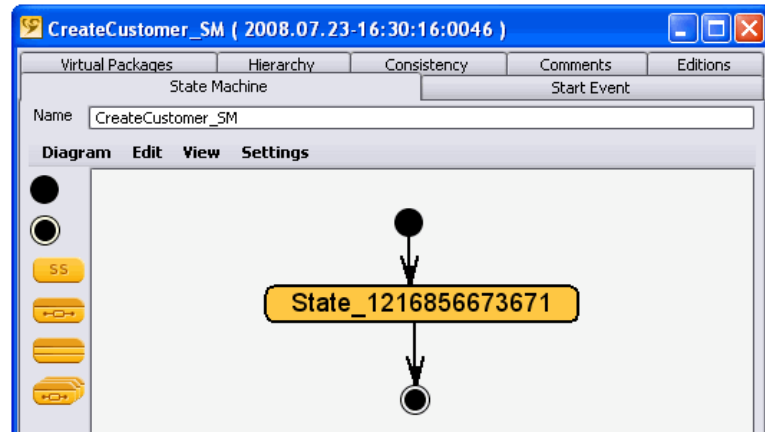
3. From the icon bar, select the **Simple State** icon , drag it into the editor window and drop it.
4. From the icon bar, select the **Final State** icon , drag it into the editor window and drop it.

You now have a state diagram with three states; it should resemble the one in the figure below.



5. Select the **Initial State** icon, hold down the **Ctrl** key and drag the cursor to the **Simple State** icon to create a transition.
6. Select the **Simple State** icon, hold down the **Ctrl** key and drag the cursor to the **Final State** icon to create another transition.

You now have a basic state diagram like the one in the figure below.



State logic

The states of a state diagram may perform actions that accomplish specific tasks in a business process. For the CreateCustomer object behavior, you define actions in the simple state that create the new Customer object.

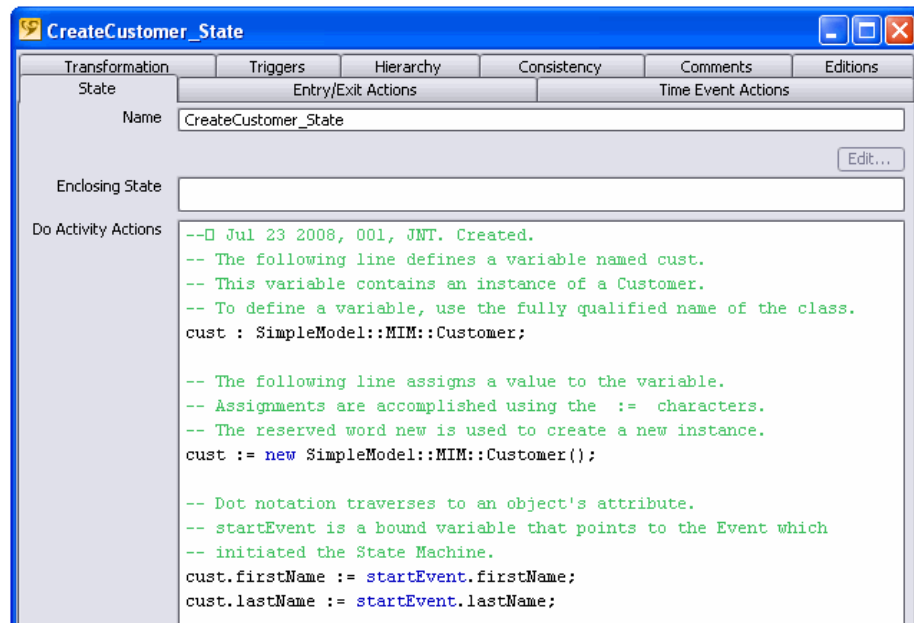
To define state actions:

1. In the state machine editor window, double-click the simple state of your diagram.
The editor window for the state appears with the Simple State Name tab foremost.
2. In the state's editor window, in the **Name** field, enter **CreateCustomer_State**.
3. Click the **State** tab, and in the **Do Activity Actions** field, enter the following action semantics:

```
-- The following line defines a variable named cust.  
-- This variable contains an instance of a Customer.  
-- To define a variable, use the fully qualified name of the class.  
cust : SimpleModel::MIM::Customer;  
  
-- The following line assigns a value to the variable.  
-- Assignments are accomplished using the := characters.  
-- The reserved word new is used to create a new instance.  
cust := new SimpleModel::MIM::Customer();  
  
-- Dot notation traverses to an object's attribute.  
-- startEvent is a bound variable that points to the Event which  
-- initiated the State Machine.  
cust.firstName := startEvent.firstName;  
cust.lastName := startEvent.lastName;
```

In the listing above, comments denoted by a double dash -- describe each line. The first two lines establish the variable for the object. The last two lines assign the values from the start

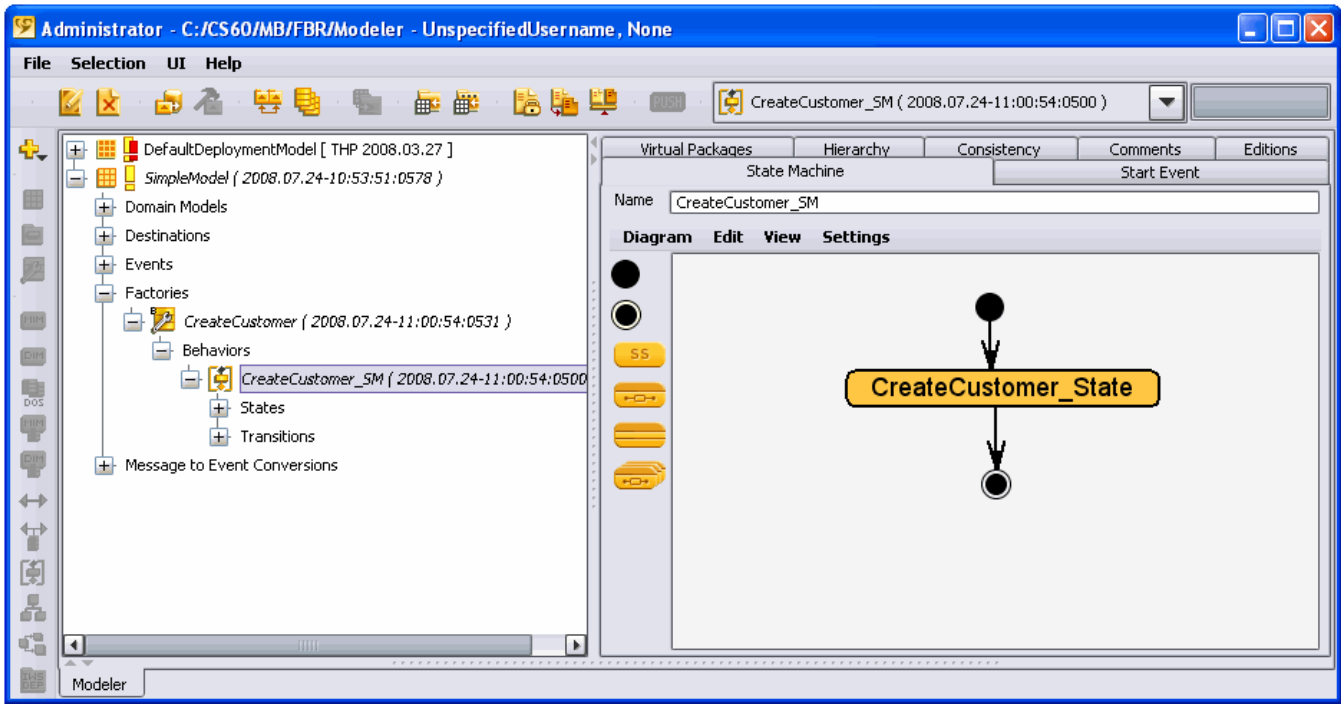
event attributes to the corresponding Customer class attributes for the object. The editor window for the CreateCustomer_State should look like the following.



For more information about action semantics, see “Using action semantics” on page 27 and the guide, *Using Actions in Cogility Studio*.

4. Close the CreateCustomer_State window.
5. Close the CreateCustomer_SM state machine editor window.


When you click on the CreateCustomer_SM state diagram artifact in Cogility Modeler's tree view, your model should look like the one in the figure below, with no inconsistencies. If you have inconsistencies, retrace the steps above.



Turn over and push

Now you have a model that actually does something: it creates a Customer object. Before you run the model, turn it over and push it into execution.


To turn over and push the model:

1. In the tree view, select the **SimpleModel** model container.
2. In the Cogility Modeler tool bar, click the **Turn Over** button .
3. In the Turnover Operation dialog, for **Turnover Version Name**, enter **Chapter3**.
4. In the Comments field, enter some comments and click **OK**.

A successful notification appears.

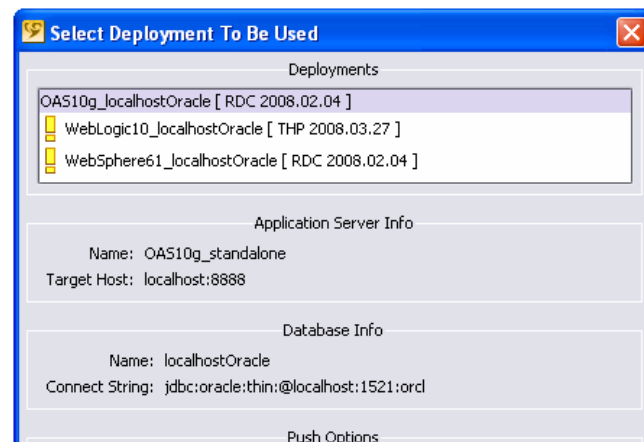
5. Click **OK**.
6. If you have already pushed a model into your run-time repository, remove the current model from the database. (Skip this step if you have not yet pushed a model.)
 - Navigate to the %DCHOME%\MB\Scripts\Dbtools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.

A command window appears and shows the Drop Tables utility executing. Close the window when it finishes.

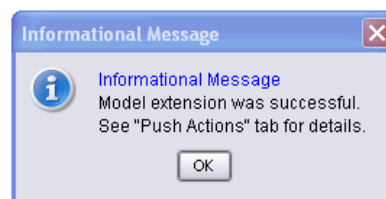
7. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
8. With the **SimpleModel** model container selected, click the Push button .

This pushes the enterprise application onto the application server, and the model objects and schema into the run-time repository.

9. In the Select Source of Deployment Parameters, select the object that matches your configuration and click **Use Deployment**.



When the push operation completes, a dialog appears notifying you of successful deployment.



10. Click **OK**.

Pushing the model accomplishes the following:

- Verifies compilation of all action semantics
- Writes tables to the execution repository
- Saves the connection parameters of the execution repository to the parameter facility
- Populates tables with factory instances and object definitions
- Extracts model information for deployment
- Creates a new enterprise application and deploys it on the application server

For more information about turn over and push see the *Cogility Studio Deployment & Execution Guide*.

Executing the model

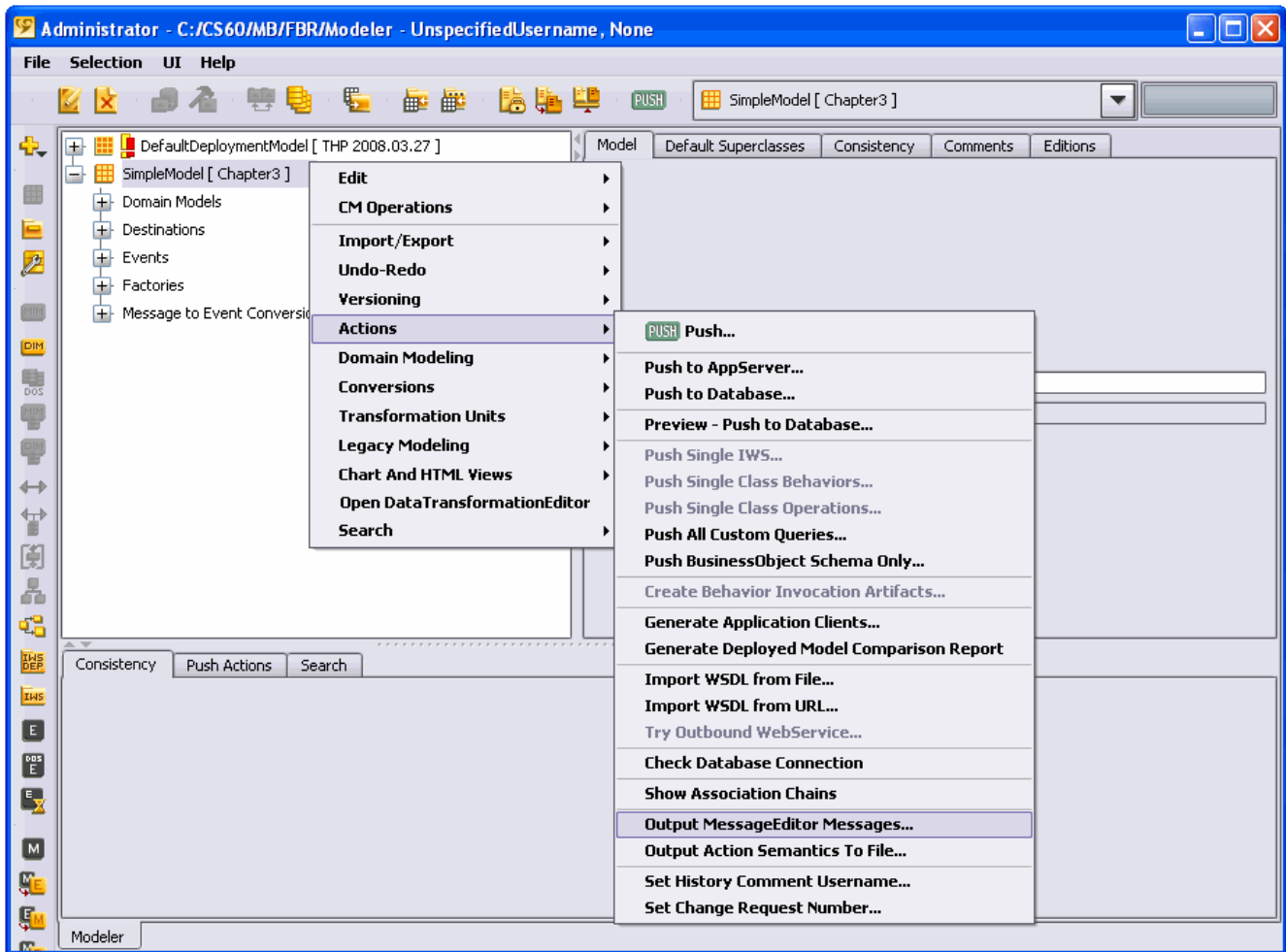
Your enterprise application now can create a new Customer with a factory behavior initiated by the arrival of a JMS message. In this section you test the application using Cogility Message Traffic Viewer and Cogility Insight. The first provides a means to send a simulated JMS message to the enterprise application, the second provides a view of the data created on the run-time repository. For more information about these and other Cogility utilities, see *Cogility Studio Deployment & Execution Guide*.

Exporting message definitions

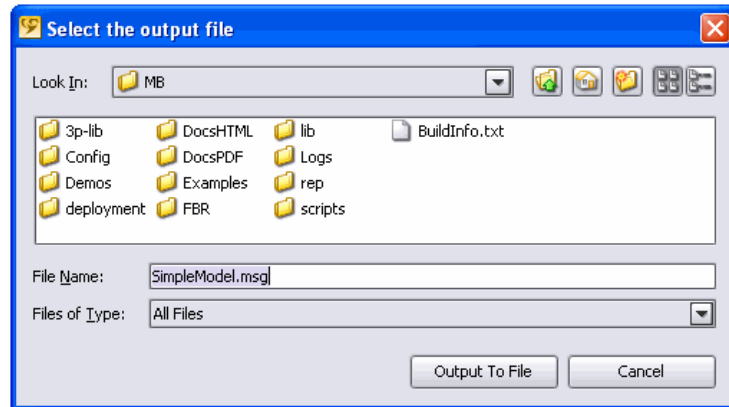
With Cogility Message Traffic Viewer you publish messages on the topics (or destinations) for which your enterprise application is listening. These are listed in Cogility Modeler's tree view under Destinations. So far, you have only created one message destination, CreateCustomer_Topic. To make message simulation easier, Cogility Modeler exports a set of JMS message definitions which can be published on these topics.

To export message definitions:

1. In Cogility Modeler's tree view, select the **SimpleModel** model container, right-click and select **Actions > Output MessageEditor Messages**.



2. Navigate to a location for the output file and click **Output to File**.



3. Note the location where the **SimpleModel.msg** file is generated and click **OK**.

When you run Cogility Message Traffic Viewer, you load this file and publish messages according to the definitions listed in it.

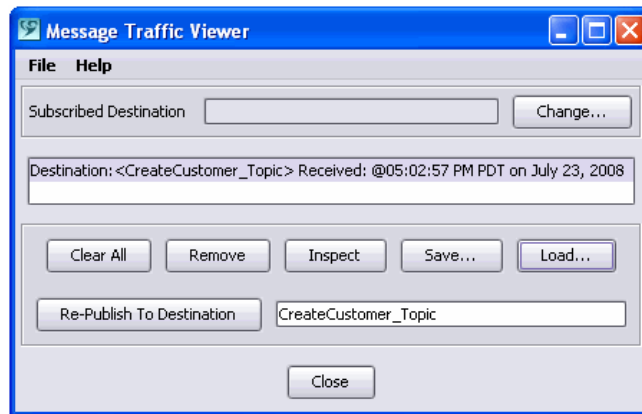
Running Cogility Message Traffic Viewer

As mentioned earlier, you don't have an actual external application to send a JMS message with new customer information. Instead, you use Cogility Message Traffic Viewer to simulate the message to your model's enterprise application.

To run Cogility Message Traffic Viewer:

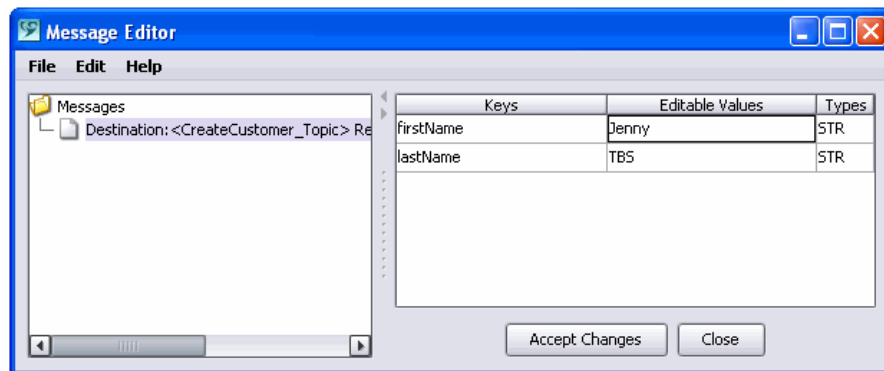
1. If your application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
A console window displays. When it shows the server is running, you may proceed with the following steps.
2. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Manager > JMS Message Viewer**.
3. In the Cogility Message Traffic Viewer window, click **Load**.
4. Navigate to where the **SimpleModel.msg** file was created in ["Exporting message definitions" on page 58](#) and click **Open**.

Cogility Message Traffic Viewer displays the message destinations from the .msg file. The only destination defined in your model, CreateCustomer_Topic, is selected in the view.



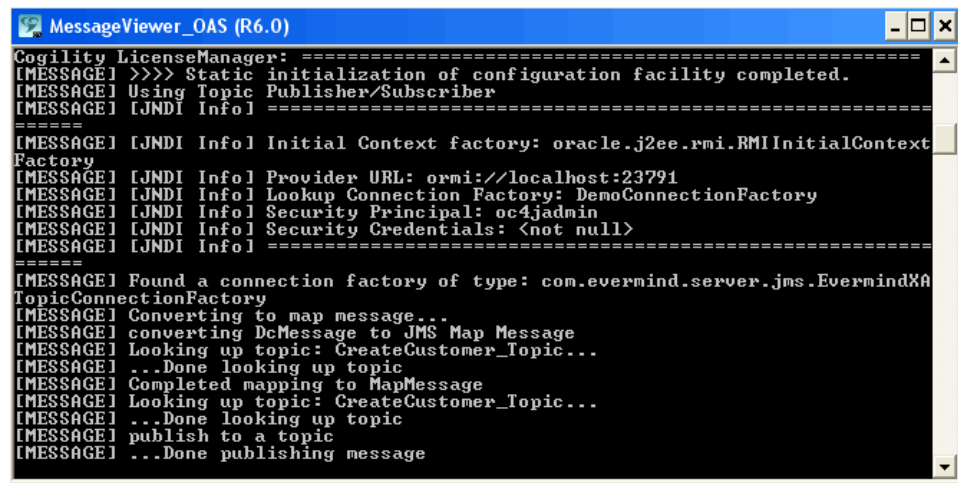
You send a message with new customer information to this destination. The next step is to define the new customer attributes with the Cogility Message Editor.

5. Click **Inspect** to open Cogility Message Editor and set the values for the message attributes.
 - a. In the tree view, under **Messages**, select **Subject: <CreateCustomer_Topic>**.
Cogility Message Editor displays the attributes that the CreateCustomer_Topic destination expects.
 - b. Under **Key**, select **firstName**, and in the bottom pane, replace **TBS** with **Jenny** (or some other first name).



- c. Under **Key**, select **lastName**, and in the bottom pane, replace **TBS** with **Watts** (or some other last name).
 - d. Click on **Accept Changes**.
 - e. Close the Cogility Message Editor window.
6. In Cogility Message Traffic Viewer, click **Save**, navigate to the location where you created the SimpleModel.msg message file in *"Exporting message definitions"* on page 58, select the file, click **Save** and click **Yes** to overwrite the original file.
7. In Cogility Message Traffic Viewer, click **Re-Publish To Destination**.

The console window for Cogility Message Traffic Viewer describes the publication process. When complete, it looks like the following.



```

MessageViewer_OAS (R6.0)
Cogility LicenseManager: =====
[MESSAGE] >>> Static initialization of configuration facility completed.
[MESSAGE] Using Topic Publisher/Subscriber
[MESSAGE] [JNDI Info] =====
[MESSAGE] [JNDI Info] Initial Context factory: oracle.j2ee.rmi.RMIInitialContext
Factory
[MESSAGE] [JNDI Info] Provider URL: ormi://localhost:23791
[MESSAGE] [JNDI Info] Lookup Connection Factory: DemoConnectionFactory
[MESSAGE] [JNDI Info] Security Principal: oc4jadmin
[MESSAGE] [JNDI Info] Security Credentials: <not null>
[MESSAGE] [JNDI Info] =====
[MESSAGE] Found a connection factory of type: com.evermind.server.jms.EvermindXA
TopicConnectionFactory
[MESSAGE] Converting to map message...
[MESSAGE] converting DcMessage to JMS Map Message
[MESSAGE] Looking up topic: CreateCustomer_Topic...
[MESSAGE] ...Done looking up topic
[MESSAGE] Completed mapping to MapMessage
[MESSAGE] Looking up topic: CreateCustomer_Topic...
[MESSAGE] ...Done looking up topic
[MESSAGE] publish to a topic
[MESSAGE] ...Done publishing message
  
```

You have published a message for the CreateCustomer_Topic destination of your model's enterprise application. Now you want to inspect the run-time repository to see if the new Customer object was created. You do that with Cogility Insight.

Running Cogility Insight

Cogility Insight is a web-based application that lets you monitor your enterprise application as it executes.

To run Cogility Insight:

1. From the **Start** menu, select **All Programs > Cogility Studio > Insight Web Access**.
2. For **UserName**, enter **admin**; for **Password**, enter **password** and click **Submit**.
3. From the **View** menu, select **Business Objects**.

- From the **Business Objects** drop-down list, select **SimpleModel.MIM.Customer** and click **Search**.

Business Object Definitions (7)

Concrete

- SimpleModel.CreateCustomer
- SimpleModel.MIM.Address
- SimpleModel.MIM.Customer**
- SimpleModel.MIM.FloorMap
- SimpleModel.MIM.Order
- SimpleModel.MIM.Product
- SimpleModel.MIM.Vacuum

☐ Only those with behaviors

Search

Show	Property Name	Property Value	Type
<input checked="" type="checkbox"/>	firstName		string
<input checked="" type="checkbox"/>	lastName		string
<input checked="" type="checkbox"/>	poid		string

Reset Uncheck All Check All

This page allows you to specify by attribute values which objects you want to see. Since there are only two objects created, you don't need to specify any values. The Search Results page then displays. It should look like the following. The run-time repository contains two Customer objects, the one you created in the last lesson (Manfred Smelch, in the figure below) and the one you created with the inbound message.

Business Object Definitions (7)

Concrete

- SimpleModel.CreateCustomer
- SimpleModel.MIM.Address
- SimpleModel.MIM.Customer**
- SimpleModel.MIM.FloorMap
- SimpleModel.MIM.Order
- SimpleModel.MIM.Product
- SimpleModel.MIM.Vacuum

☐ Only those with behaviors

Search

Show	Property Name	Property Value	Type
<input checked="" type="checkbox"/>	firstName		string
<input checked="" type="checkbox"/>	lastName		string
<input checked="" type="checkbox"/>	poid		string

Reset Uncheck All Check All

Found 2 business object(s). Retrieved 2. Displaying 2 object(s), from 1 to 2. Page 1 / 1.

Last DB access: 07/24/08 09:30:09 AM PDT

Business Object	firstName	lastName	poid
SimpleModel.MIM.Customer	Manfred	Smelch	BO^SimpleModel.MIM.Customer^355d997b:11b55e62549:7fff
SimpleModel.MIM.Customer	Jenny	Watts	BO^SimpleModel.MIM.Customer^d0c7d0a:11b55d931cb:7ffa

Found 2 business object(s). Retrieved 2. Displaying 2 object(s), from 1 to 2. Page 1 / 1.

The inbound message with new customer information that you sent with Cogility Message Traffic Viewer to your model's enterprise application has been converted to an event that launched a business process creating a new Customer object.




In the previous chapter, “[Inbound JMS](#)” on page 43, you developed the inbound data path between a hypothetical external application and your model. New customer information was sent to the model in a JMS message that the model converted to an event and used to create a Customer object in the run-time repository.

In this chapter you enhance the model to send an acknowledgement back to the external application upon receipt of new customer information and creation of the Customer object. This scenario illustrates *outbound* communication from the model to the external application.

Importing the existing model

In the last chapter, you developed a model called SimpleModel. The steps in this chapter start with that model and add more features. If you do not have that version of the model loaded in Cogility Modeler, you can import it with the following procedure. If you are continuing from “[Inbound JMS](#)” on page 43, skip to “[Adding a new attribute](#)” on page 65.

To import the existing model:

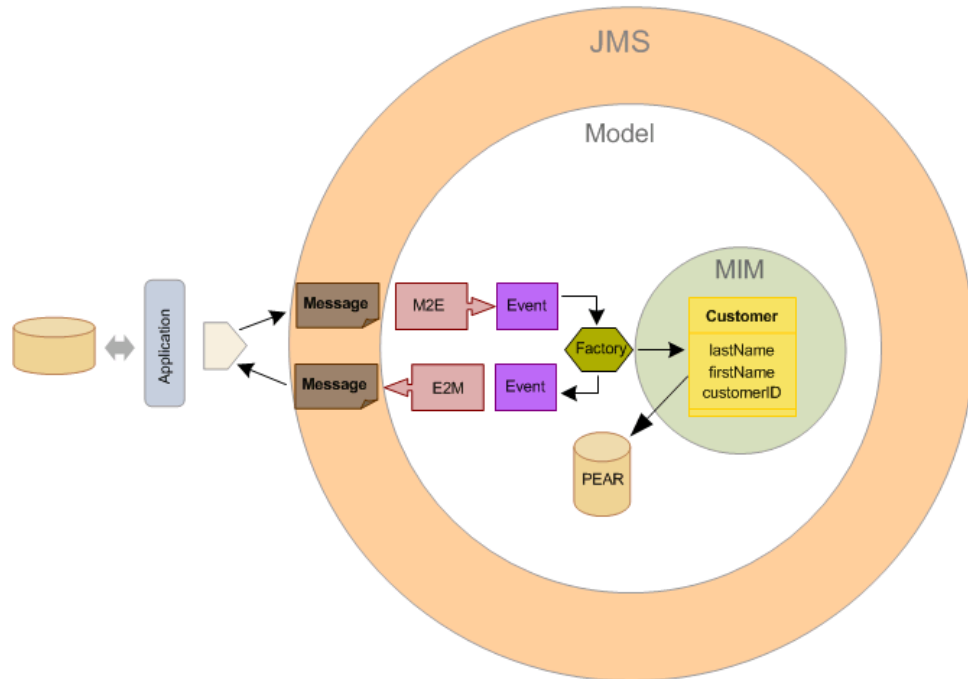
1. If you have loaded in Cogility Modeler a model other than the SimpleModel created in “[Inbound JMS](#)” on page 43, close Cogility Modeler and follow the steps under “[Loading the model repository](#)” on page 11.
2. If Cogility Modeler is not running, follow the steps under “[Logging in](#)” on page 12.
3. In Cogility Modeler, from the **Selection** menu, select **Import/Export > Import from Single File**.
4. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial directory, select **SimpleModel_Chapter3.cog** and click **Open**.
%DCHOME% is the directory where you installed Cogility Studio.
5. In the dialog that prompts you, click **Yes** to make the model visible in Cogility Modeler.
6. In the tree view, select the **SimpleModel** model container and click the **Place Model Under CM** button .

You are now ready to follow the procedures of this chapter.

Adding a new attribute

You extend the Customer class to include a new attribute, customerID. When the factory creates the new Customer object, it also creates an event that gets converted to a JMS message. The message is

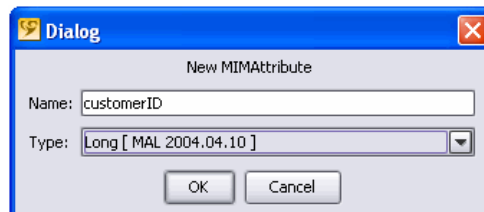
published on a topic (or destination) for the external application; it acknowledges the new Customer creation and reports the customerID for the new Customer object.



For your model to assign a customerID to each Customer object it creates, the Customer class must be extended to include the new attribute.

To add the new attribute to the class:

1. In Cogility Modeler's tree view, expand (by clicking the plus sign +) the **SimpleModel**, **Domain Models**, **MIM**, **MIM Classes** and select the **Customer** class.
The Customer class properties display in the content view.
2. In the content view, in the **Information Model Class** tab, above the **Attributes** field, click **New**.
 - a. In the dialog that appears, for **Name**, enter **customerID**.
 - b. For **Type**, from the drop-down menu, select **Long** and click **OK**.



The Attribute Editor tab appears in the content view.

3. Under the **Attribute** tab, check the **Is Sequenceable** checkbox.
Identifying an attribute as sequenceable tells the run-time repository to automatically generate a unique ID for each instance of the attribute.

Your Customer class should now look like the following.

The screenshot shows the 'Customer' class configuration in Cogility Modeler. The 'Name' field is 'Customer'. The 'Abstract' checkbox is unchecked. The 'Superclass' is 'Element [MAL 2004.04.10]'. The 'Attributes' list contains three items: 'customerID (2008.07.24-10:04:32:0875)', 'firstName [Chapter1]', and 'lastName [Chapter1]'. The 'Attribute' tab is selected, showing details for 'customerID': Name 'customerID', Size '0', Type 'Long [MAL 2004.04.10]', and 'Is Sequenceable' checked.

Extending the message model


Your model's outbound communication follows a data path from the run-time repository to the factory object and proceeds with an event, an event to message conversion, and finally, a message to the external application, as illustrated in the figure under ["Adding a new attribute" on page 65](#).

The factory object publishes a message on an event with an acknowledgement and a customerID to the external application. In the following sections, you create the new event artifact and the event to message conversion artifact.

Creating the event artifact

The acknowledgement event includes the attribute values from the Customer object.

To create the event artifact:


1. In Cogility Modeler's tree view, select the **SimpleModel** model container and click the **Add an Event** button .
2. In the content view, for **Name**, enter **CreationAck_Event**.
3. Above the **Attributes** field, click **New**.
 - a. In the dialog that appears, for **Name**, enter **customerID**.

- b. For **Type**, from the drop-down menu, select **Long** and click **OK**.

Creating the conversion artifact

Just as in the previous chapter you converted a message to an event (M2E) for inbound communication, you must have an event to message (E2M) conversion for outbound communication.

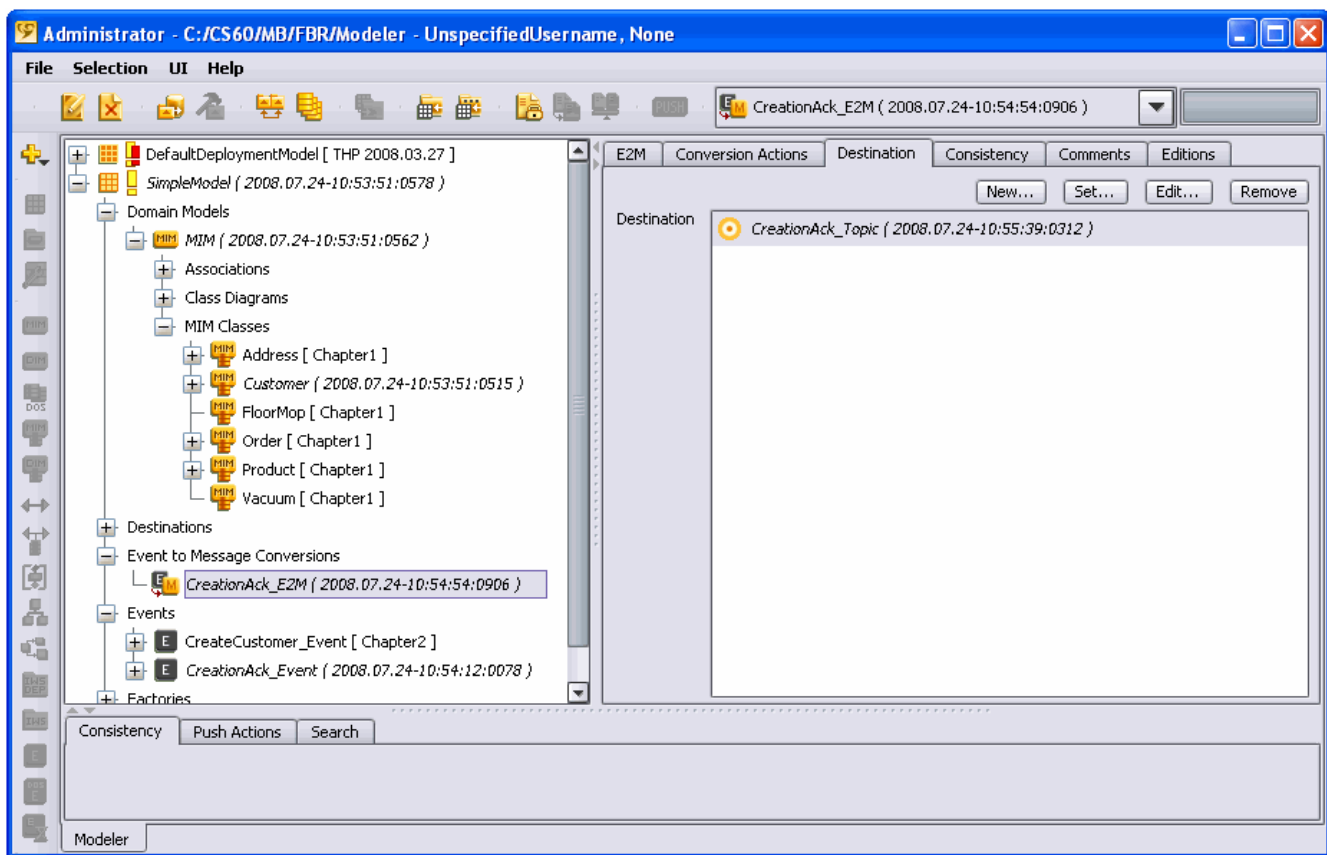
To create the conversion artifact:

1. In Cogility Modeler's tree view, select the **SimpleModel** model container and click the **Add an E2M Conversion** button .
2. In the content view, for **Name**, enter **CreationAck_E2M**.
3. For **Event**, click **Set**, select **CreationAck_Event** and click **OK**.
4. Click the **Conversion Actions** tab and leave checked the **Default Conversion** checkbox.

By leaving Default Conversion checked, you are setting the conversion artifact to look in the outbound event for attribute values and place them in the JMS message.

5. In the **Destination** tab, click **New**.
6. In the dialog that appears, enter **CreationAck_Topic** and click **OK**.

Your model should now look like the one in the following figure.



Updating factory behavior

With the outbound data path now in place, you can add new logic to the CreateCustomer factory behavior to populate the new customerID attribute, generate the CreationAck_Event, populate its attribute value, and publish a JMS message to the external application.

To update the factory behavior:

1. In the tree view, expand by clicking the plus signs (+) for the **Factories** package, **CreateCustomer**, **Behaviors**, **CreateCustomer_SM**, **States** and select the **CreateCustomer_State**.

The CreateCustomer_State editor window appears in the content view.

2. Click the **State** tab, and under **Do Activity Actions**, append the following action semantics:
 - a. Begin a new line with **publish** ' '.
 - b. Hold down the Ctrl key and press the space bar.
 - c. Select '**CreationAck_Topic**' **do...** and press **Enter**.

The code completion utility automatically supplies the rest of the code to publish a message on the CreationAck_Topic. You just need to specify a value for the **customerID** parameter.

- d. On the second line, between the parentheses, select **VALUE** and replace it with `cust.customerID`.

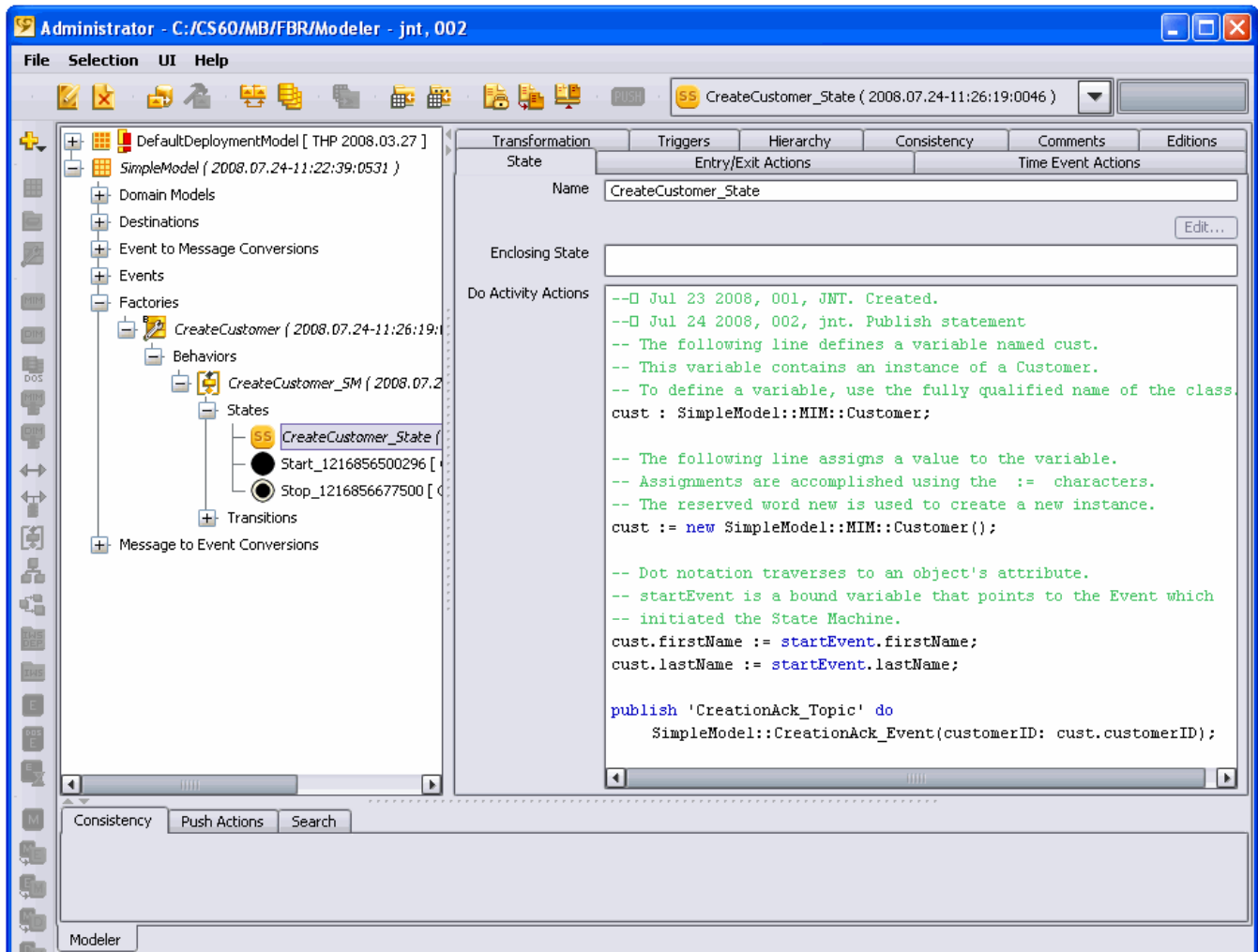
The code you add should read as follows:

```
publish 'CreationAck_Topic' do
    SimpleModel::CreationAck_Event(customerID: cust.customerID);
```

The `publish` reserved word tells the model's enterprise application to publish a message with the destination `CreationAck_Topic` using the data from the `CreationAck_Event`. The `customerID` gets its value from the `Customer` object held in the `cust` variable.

For more information about action semantics, see the guide, *Using Actions in Cogility Studio*.



Your model should now resemble the one shown in the following figure.



Turn over and push

Before you run the model, turn it over with the latest changes and push it into execution.

To turn over and push the model:

1. In the tree view, select the **SimpleModel** model container.
2. In the Cogility Modeler tool bar, click the **Turn Over** button .
3. In the Turnover Operation dialog, for **Turnover Version Name**, enter **Chapter4**.
4. In the Comments field, enter some comments and click **OK**.
A successful notification appears.
5. Click **OK**.
6. If you have already pushed a model into your run-time repository, remove the current model from the database. (Skip this step if you have not yet pushed a model.)
 - Navigate to the %DCHOME%\MB\Scripts\Dbtools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.
 A command window appears and shows the Drop Tables utility executing. Close the window when it finishes.
7. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
8. With the **SimpleModel** model container selected, click the Push button .
This pushes the enterprise application onto the application server, and the model objects and schema into the run-time repository.
9. In the Select Source of Deployment Parameters, select the object that matches your configuration and click **Use Deployment**.
When the push operation completes, a dialog appears notifying you of successful deployment.
10. Click **OK**.

Executing the model

Once again you test the enterprise application using Cogility Message Traffic Viewer and Cogility Insight.

Running Cogility Message Traffic Viewer

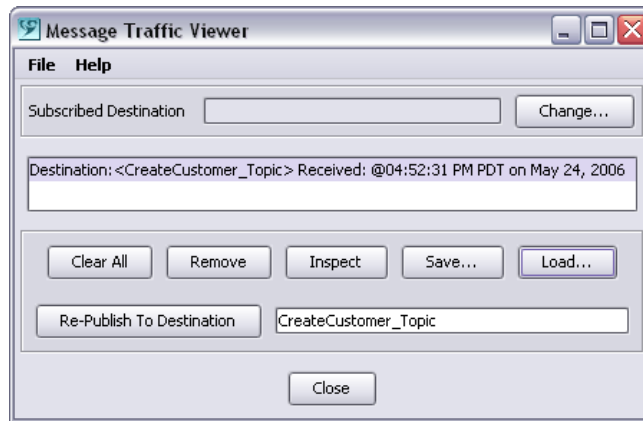
As mentioned earlier, you don't have an actual external application that sends a JMS message with new customer information. Instead, you use Cogility Message Traffic Viewer to simulate the message to your model's enterprise application. Likewise, you use Cogility Message Traffic Viewer to receive the acknowledgement from your enterprise application.

To run Cogility Message Traffic Viewer:

1. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
A console window displays. When it shows the server is running, you may proceed with the following steps.
2. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Manager > JMS Message Viewer**.
You must select the Cogility Message Traffic Viewer appropriate for your application server.
3. In the Cogility Message Traffic Viewer window, click **Load**.

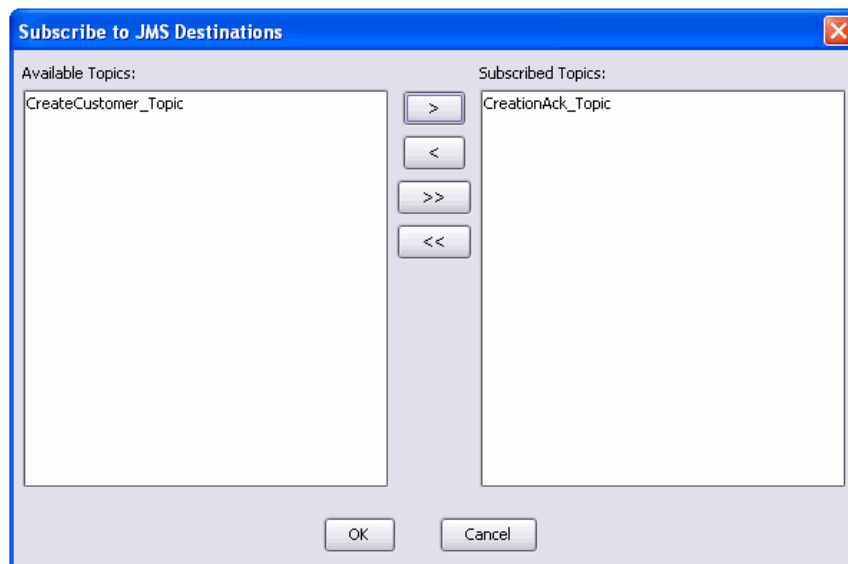
4. Navigate to where you saved the **SimpleModel.msg** file in step 6 under “**Running Cogility Message Traffic Viewer**” on page 59 and click **Open**.

Cogility Message Traffic Viewer displays the message destinations from the .msg file. The only destination defined in your model, `CreateCustomer_Topic`, is selected in view.



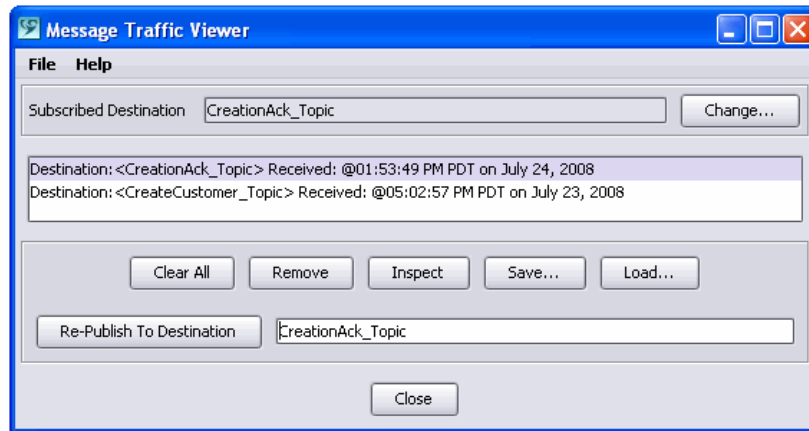
You resend the message with new customer information (you saved that message in step 6 under “**Running Cogility Message Traffic Viewer**” on page 59). At the same time Cogility Message Traffic Viewer listens for the acknowledgment message from the model’s enterprise application.

5. Next to the **Subscribed Destination** field, click **Change**.
6. In the dialog that appears, select **CreationAck_Topic** in the left pane, move it to the right pan, and click **OK**.



7. In Cogility Message Traffic Viewer, click **Re-Publish To Destination**.

The acknowledgement message from the enterprise application, `CreationAck_Topic` appears in Cogility Message Traffic Viewer at the top of the list as show in the following figure.

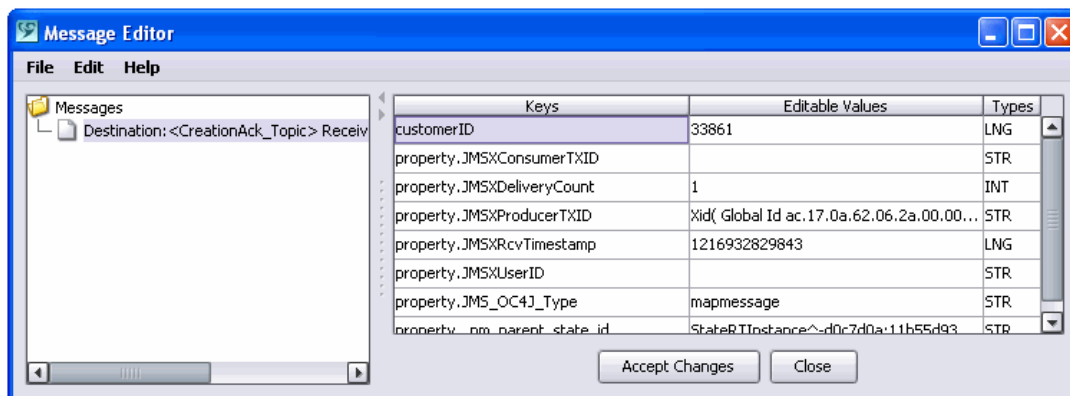


8. With `<CreationAck_Topic> Received` selected, click **Inspect**.

Cogility Message Editor appears.

- a. In the tree view, under **Messages**, select **Subject <CreationAck_Topic> Received**.

As shown in the figure below, the acknowledgement message includes a `customerID`.



- b. Close Cogility Message Editor.

Now you want to see if the `customerID` value was assigned to the new customer.

Running Cogility Insight

Cogility Insight is a web-based application that lets you monitor your enterprise application as it executes, and it affords a look at the data residing in the run-time repository.

To run Cogility Insight:

1. From the **Start** menu, select **All Programs > Cogility Studio > Insight Web Access**.
2. For **UserName**, enter **admin**; for **Password**, enter **password** and click **Submit**.
3. In the **View** menu, select **Business Objects**.
4. From the **Business Objects** drop-down list, select **SimpleModel.MIM.Customer** and click **Search**.

If you haven't cleared the database from the previous chapter's exercise, you see three Customer objects, two with the same name values. The most recent Customer has a customerID.

Search Business Objects

Business Object Definitions (7)

Concrete
SimpleModel.CreateCustomer
SimpleModel.MIM.Address
SimpleModel.MIM.Customer
SimpleModel.MIM.FloorMop
SimpleModel.MIM.Order
SimpleModel.MIM.Product
SimpleModel.MIM.Vacuum

☐ Only those with behaviors

Search

Show	Property Name	Property Value	Type
<input checked="" type="checkbox"/>	customerID		long
<input checked="" type="checkbox"/>	firstName		string
<input checked="" type="checkbox"/>	lastName		string
<input checked="" type="checkbox"/>	poid		string

Reset Uncheck All Check All

Found 3 business object(s). Retrieved 3. Displaying 3 object(s), from 1 to 3. Page 1 / 1.

Last DB access: 07/24/08 02:09:45 PM PDT

Business Object	customerID	firstName	lastName	poid
SimpleModel.MIM.Customer		Manfred	Smelch	BO^SimpleModel.MIM.Customer^355d897b:11b55e62549:7fff
SimpleModel.MIM.Customer	33861	Jenny	Watts	BO^SimpleModel.MIM.Customer^d0c7d0a:11b55d931cb:7ff4
SimpleModel.MIM.Customer		Jenny	Watts	BO^SimpleModel.MIM.Customer^d0c7d0a:11b55d931cb:7ffa

Found 3 business object(s). Retrieved 3. Displaying 3 object(s), from 1 to 3. Page 1 / 1.

The inbound message with new customer information that you sent with Cogility Message Traffic Viewer to your model's enterprise application has been converted to an event that launched a business process creating a new Customer object. That same business process published an acknowledgement message with a new attribute and value for the Customer object.

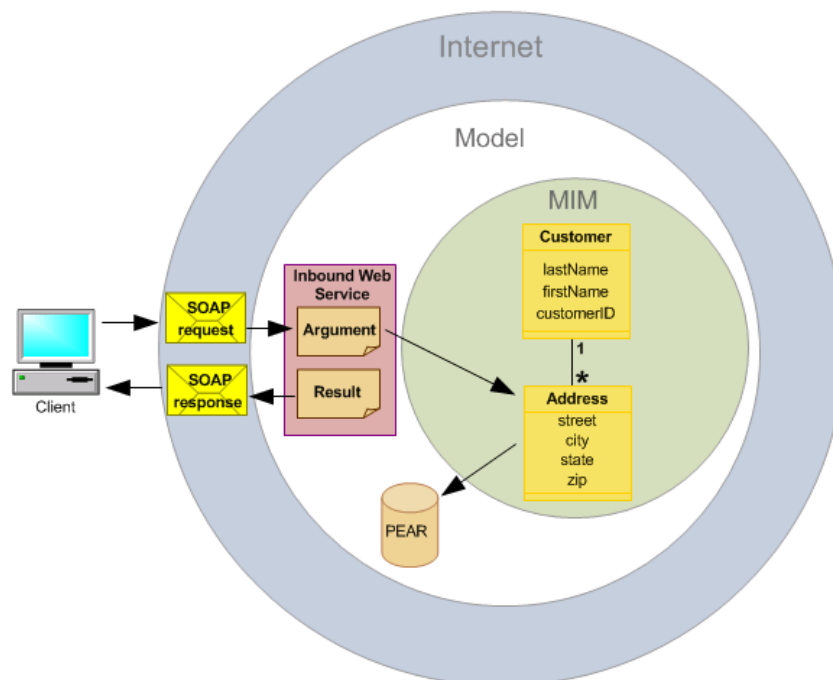


Inbound web services

In the chapter, “[Inbound JMS](#)” on page 43 you created a message model that handled inbound communication from an external application to your integration model. That message model is predicated entirely upon JMS. The inbound JMS message is converted to an event, and so on. But what if the external application needs to communicate with your model from a non-J2EE platform?

Inbound web services provide external applications with a non-J2EE communication capability using industry standard web services. In this chapter you create an inbound web service to update a customer address. This is illustrated in the figure below.

The client sends a SOAP request to the inbound web service, which decodes the request into an argument message containing the data attributes to be updated. The inbound web service’s action semantics take the attributes and assign them to an Address object. The inbound web service also encodes a SOAP response with the values from its result message, and sends the response back to the client.




For more information about inbound web services, see the guide, *Modeling with Cogility Studio*.

Importing the existing model

In the last chapter, you developed a model called SimpleModel. The steps in this chapter start with that model and add more features. If you do not have that version of the model loaded in Cogility Modeler, you can import it with the following procedure. If you are continuing from “Outbound JMS” on page 65, skip to “Creating an inbound web service” on page 76.

To import the existing model:


1. If you have loaded in Cogility Modeler a model other than the SimpleModel created in “Outbound JMS” on page 65, close Cogility Modeler and follow the steps under “Loading the model repository” on page 11.
2. If Cogility Modeler is not running, follow the steps under “Logging in” on page 12.
3. In Cogility Modeler, from the **Selection** menu, select **Import/Export > Import from Single File**.
4. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial directory, select **SimpleModel_Chapter4.cog** and click **Open**.
%DCHOME% is the directory where you installed Cogility Studio.
5. In the dialog that prompts you, click **Yes** to make the model visible in Cogility Modeler.
6. In the tree view, select the **SimpleModel** model container and click the **Place Model Under CM** button .

You are now ready to follow the procedures of this chapter.

Creating an inbound web service

In the chapter, “Inbound JMS” on page 43, a hypothetical external application sends a JMS message to the model’s enterprise application to effect a data change, creating a new Customer object, in the run-time repository. In this section you use an inbound web service to effect a similar data change, this time creating and updating an Address object.

To create an inbound web service:

1. In Cogility Modeler’s tree view, select the **SimpleModel** model container, and click the **Add Inbound Web Service** button .
The new inbound web service appears in the tree view. Its editor window appears in the content view.
2. In the content view, in the **Name** field, enter **UpdateCustomerAddress_IWS**.

Defining the argument message

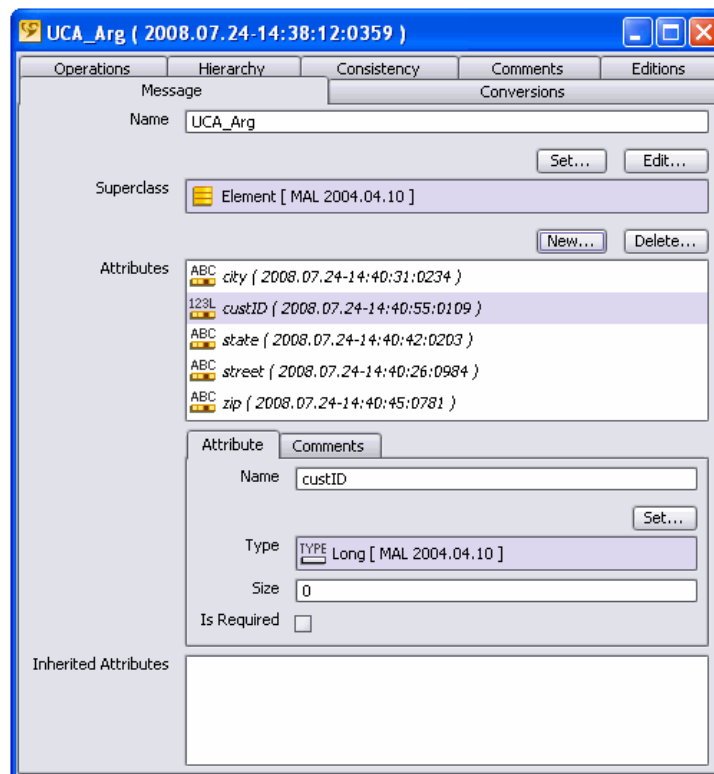
The inbound web service allows an application on any platform to communicate with the model’s enterprise application. Instead of using JMS messages, the inbound web service decodes a SOAP request and creates an input argument message. The web service’s logic (action semantics) uses the attribute values of the argument message to effect the data change.

To create the argument message:

1. Click the **IWS Arguments/Results** tab; in the **Argument Message** field, select **arg** and click **Edit**.
The argument message editor appears.

2. In the editor window, under the **Message** tab, in the **Name** field, change the name to **UCA_Arg**.
3. Above the **Attributes** field, click **New**.
The message attributes describe the data that is updated in the run-time repository.
 - a. In the dialog window, for **Name**, enter **street**.
 - b. For **Type**, enter **String** and click **OK**.
4. Repeat step 3 for the following attributes:
 - **city** (type: **String**)
 - **state** (type: **String**)
 - **zip** (type: **String**)
 - **custID** (type: **Long**)

The argument editor should appear as in the figure below.



5. Close the argument message editor.

Defining the result message

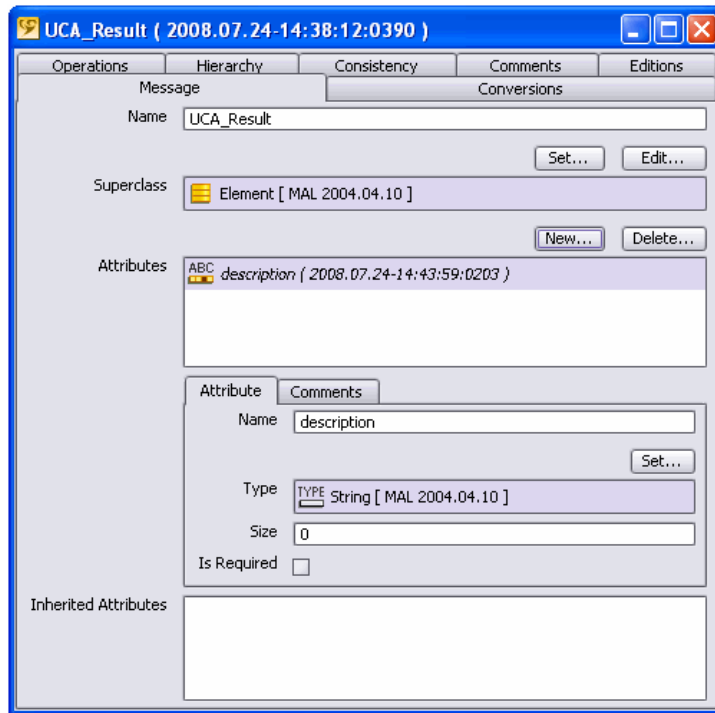
The inbound web service also encodes a SOAP response for the client calling the web service. It uses the attribute values of the result message that you create in this section.

1. In the content view, under the **Web Service** tab, in the **Result Message** field, select **result** and click **Edit**.
The result message editor appears.
2. Under the **Message** tab, in the **Name** field, change the name to **UCA_Result**.
3. Above the **Attributes** field, click **New**.
 - a. In the dialog, for **Name**, enter **description**.

b. For **Type** field, select **String** and click **OK**.

The response the web service sends back to the client is held in this description attribute.

The result message editor should appear as in the figure below.



4. Close the result message editor.

Defining the web service logic

With the argument and result messages defined, you can now describe the logic that uses the attribute values of those messages. The web service logic is defined with action semantics. These actions use the **input** and **output** bound variables to refer to the argument and result messages, respectively.

To define the web service logic:

1. In the content view, under the **IWS** tab, in the **Action** field, enter the following action semantics:

```
-- 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
addr := java com::cogility::WalkAssoc(cust, 'address');

-- Test whether an Address was found.
-- If so, continue, if not create a new Address object,
-- associate it to the Customer, and then continue
if addr = null then
    addr := new SimpleModel::MIM::Address();
    new cust to address(addr);
endif;

-- Update the Address variable with the values supplied by the argument
message
addr.street := input.street;
addr.city := input.city;
addr.state := input.state;
addr.zip := input.zip;

-- Update the Result message appropriately
output.description := 'Success: The customer address has been
updated.';
```

The Action field should appear as in the figure below.


Deployments for IWS		HTTP Deployments for IWS		Consistency	Comments	Editions
IWS	IWS Arguments/Results	IWS Parameters	WS Handler Chain		Chart	
Name	UpdateCustomerAddress_IWS					
Action	<pre>--□ Jul 24 2008, 003, jnt. Created. -- 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 addr := java com::cogility::WalkAssoc(cust, 'address'); -- Test whether an Address was found. -- If so, continue, if not create a new Address object, -- associate it to the Customer, and then continue if addr = null then addr := new SimpleModel::MIM::Address(); new cust to address(addr); endif; -- Update the Address variable with the values supplied by the argument message addr.street := input.street; addr.city := input.city; addr.state := input.state; addr.zip := input.zip; -- Update the Result message appropriately output.description := 'Success: The customer address has been updated.';</pre>					

2. To compile the action semantics, click anywhere outside the **Actions** field.

Creating a deployment

An inbound web service must specify a deployment, the location (URL) where clients can find it. Since the deployment is on the J2EE application server where the enterprise application runs, you need only specify the relative path from the application server's main URL.

To create a deployment for the inbound web service:

1. In the tree view, select the **SimpleModel** model container, and click the **Add Inbound Web Service Deployment** button .

An untitled deployment appears in the tree view and the deployment editor appears in the content view.

2. In the content view, under the **IWS Deployment** tab, enter the following values:

- In the **Name** field, enter **CogilityIWS**
- In the **Context Root** field, enter **services**
- In the **Namespace Host** field, enter **CogilityServer**.

- In the **Namespace Path** field, enter **servicePath**

IWS Deployment Handler Chain		Consistency	Comments	Editions
IWS Deployment		IWS Deployment Operations		
Name	CogilityIWS			
Context Root	services			
Namespace Host	CogilityServer			
Namespace Path	servicePath			
Service Name				
Port Name				
Port Type Name				
Binding Name				

3. In the content view, select the **IWS Deployment Operations** tab.
You can now add the web service to this deployment.
4. Above the **IWS For Deployment** field, click the **Add** button.
5. In the selection window, select **UpdateCustomerAddress_IWS** and click **OK**.


IWS Deployment Handler Chain		Consistency	Comments	Editions
IWS Deployment		IWS Deployment Operations		
		Add...	Edit...	Remove
IWS For Deployment		UpdateCustomerAddress_IWS (2008.07.24-14:38:12:0281)		
IWS Operation Deployment Details				
Operation Name	UpdateCustomerAddress_IWS			
Input Message Name	UCA_Arg			
Input Message Type Name	UCA_Arg			
Output Message Name	UCA_Result			
Output Message Type Name	UCA_Result			
Input Parameter Order	city;custID;state;street;zip;			
Output Parameter Order	description;			


The last inconsistency disappears. The model is now ready to push.

Turn over and push

Before you run the model, turn it over and push it into execution.

To turn over and push the model:

1. In the tree view, select the **SimpleModel** model container.
2. In the Cogility Modeler tool bar, click the **Turn Over** button .
3. In the Turnover Operation dialog, for **Turnover Version Name**, enter **Chapter5**.
4. In the Comments field, enter some comments and click **OK**.
A successful notification appears.
5. Click **OK**.

6. If you have already pushed a model into your run-time repository, remove the current model from the database. (Skip this step if you have not yet pushed a model.)
 - Navigate to the %DCHOME%\MB\Scripts\Dbtools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.A command window appears and shows the Drop Tables utility executing. Close the window when it finishes.
7. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
8. With the **SimpleModel** model container selected, click the Push button . This pushes the enterprise application onto the application server, and the model objects and schema into the run-time repository.
9. In the Select Source of Deployment Parameters, select the object that matches your configuration and click **Use Deployment**.
When the push operation completes, a dialog appears notifying you of successful deployment.
10. Click **OK**.

Executing the model

After you create another Customer object with Cogility Message Traffic Viewer, you run Cogility Insight to test the inbound web service.

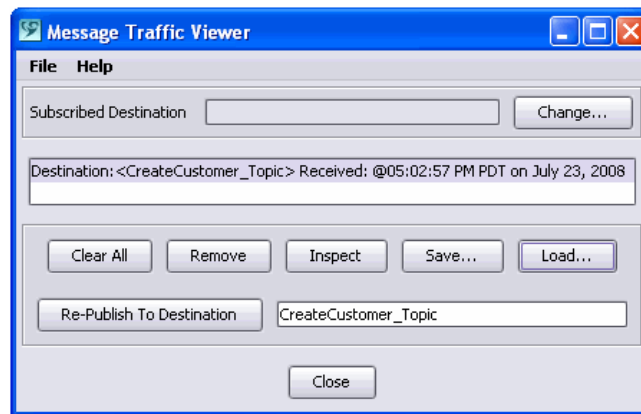
Running Cogility Message Traffic Viewer

As you did in “[Outbound JMS](#)” on page 65, you create a Customer object with its own unique customerID by publishing the same JMS message from the previous chapters.

To run Cogility Message Traffic Viewer:

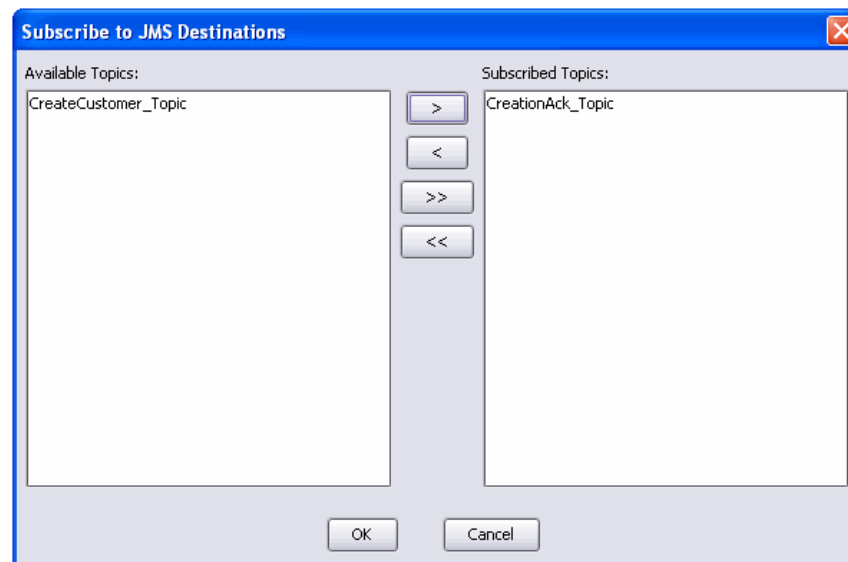
1. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
A console window displays. When it shows the server is running, you may proceed with the following steps.
2. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Manager > JMS Message Viewer**.
3. In the Cogility Message Traffic Viewer window, click **Load**.
4. Navigate to the file you saved in step 6 under “[Running Cogility Message Traffic Viewer](#)” on page 59 and click **Open**.

Cogility Message Traffic Viewer displays the message destinations from the .msg file. The only destination defined in your model, CreateCustomer_Topic, is selected in view.



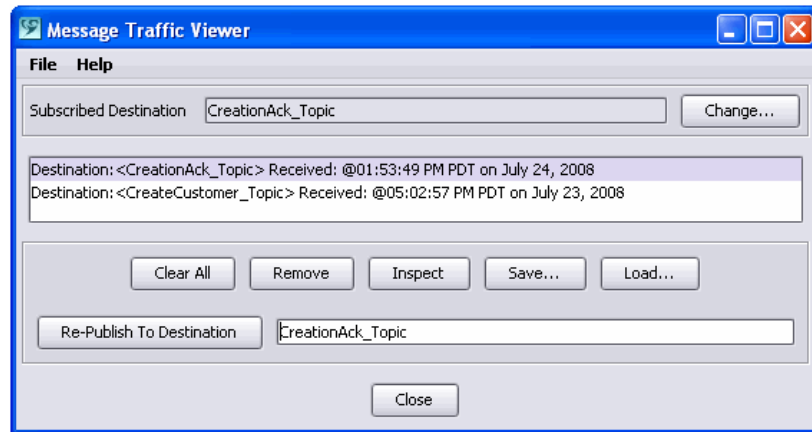
You resend the message with new customer information (you saved that message in step 6 under “[Running Cogility Message Traffic Viewer](#)” on page 59). At the same time Cogility Message Traffic Viewer listens for the acknowledgment message from your model's enterprise application.

5. Next to the **Subscribed Destination** field, click **Change**.
6. In the dialog that appears, specify **CreationAck_Topic** as a Subscribed Topic and press **OK**.



7. In Cogility Message Traffic Viewer, click **Re-Publish To Destination**.

The acknowledgement message from the enterprise application, `CreationAck_Topic` appears in Cogility Message Traffic Viewer at the top of the list as show in the following figure.

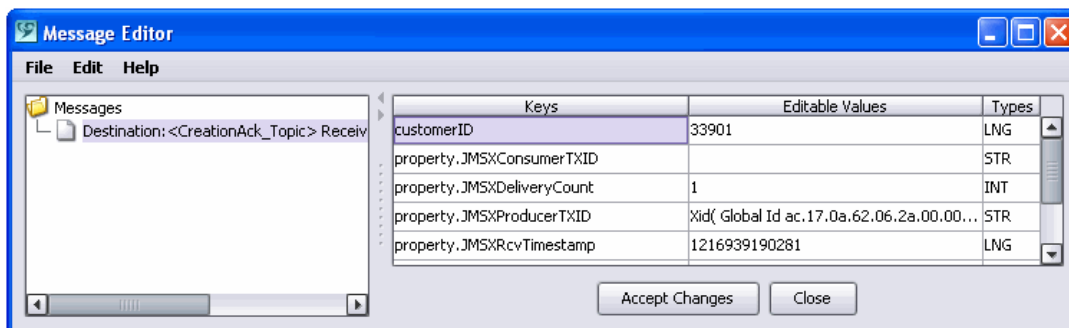


8. With `<CreationAck_Topic> Received` selected, click **Inspect**.

Cogility Message Editor appears.

- a. In the tree view, under **Messages**, select **Subject <CreationAck_Topic> Received**.

As shown in the figure below, the acknowledgement message includes a `customerID`. This is a new Customer object, distinct from the one you created in the previous chapter, and it should have a new `customerID`.



- b. Make a note of the `customerID` value generated. You will refer to it in a later exercise.
c. Close Cogility Message Editor.

Now you want to see if the `customerID` value was assigned to the new customer.

Running Cogility Insight

Cogility Insight is a web-based application that lets you monitor your enterprise application as it executes.

To run Cogility Insight:

1. From the **Start** menu, select **All Programs > Cogility Studio > Insight Web Access**.
You must run the Cogility Insight version specific to your application server. The application opens in the default web browser for your system.
2. For **UserName**, enter **admin**; for **Password**, enter **password** and click **Submit**.
3. From the **View** menu, select **Business Objects**.

- From the **Business Objects** drop-down menu, select **SimpleModel.MIM.Customer** and click **Search**.

If you haven't cleared the database from the previous chapters' exercises, there are now four records.

Last DB access: 07/24/08 03:43:31 PM PDT

Business Object	customerID	firstName	lastName	poId
SimpleModel.MIM.Customer		Manfred	Smelch	BO^SimpleModel.MIM.Customer^355d897b:11b55e62549:-7fff
SimpleModel.MIM.Customer	33861	Jenny	Watts	BO^SimpleModel.MIM.Customer^d0c7d0a:11b55d931cb:-7ff4
SimpleModel.MIM.Customer	33901	Jenny	Watts	BO^SimpleModel.MIM.Customer^d0c7d0a:11b55d931cb:-7fee
SimpleModel.MIM.Customer		Jenny	Watts	BO^SimpleModel.MIM.Customer^d0c7d0a:11b55d931cb:-7ffa

Found 4 business object(s). Retrieved 4. Displaying 4 object(s), from 1 to 4. Page 1 / 1.

- Select the record for the Customer whose customerID value you noted in step 8b on page 84 (in this example, customerID 1601) by clicking on the **SimpleModel.MIM.Customer** link in the **Business Object** column.

The Business Object Details window appears.

- From the **Associations** drop-down list, select **address** and press the **Search** button.

Source: SimpleModel.MIM.Customer

Property Name	Property Value
customerID	33901
firstName	Jenny
lastName	Watts
poId	BO^SimpleModel.MIM.Customer^d0c7d0a:11b55d931cb:-7fee

Associations (2):

Found 0 business object(s). Retrieved 0. Displaying 0 object(s), from 0 to 0. Page 0 / 0.

Although the Customer has a valid association with a role called address, (which you assigned to the Address object) currently this association does not have a target. That is, there is no Address object with an address role for the Customer object selected.

Viewing the web service deployment

Now that you have verified that the record was created, you can use Cogility Insight to verify that the web service was deployed. First, use Cogility Insight to view the web services deployed on the application server.

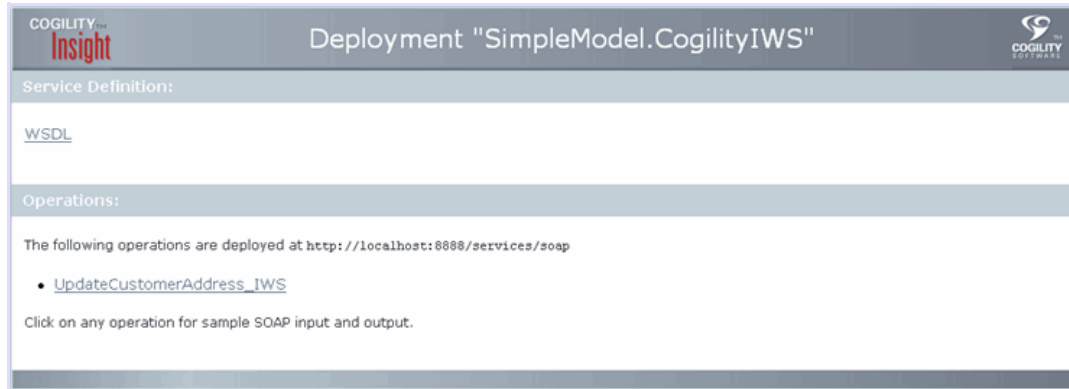
To view the web service deployment:

- In Cogility Insight, from the **View** menu, select **IWS Deployments**.

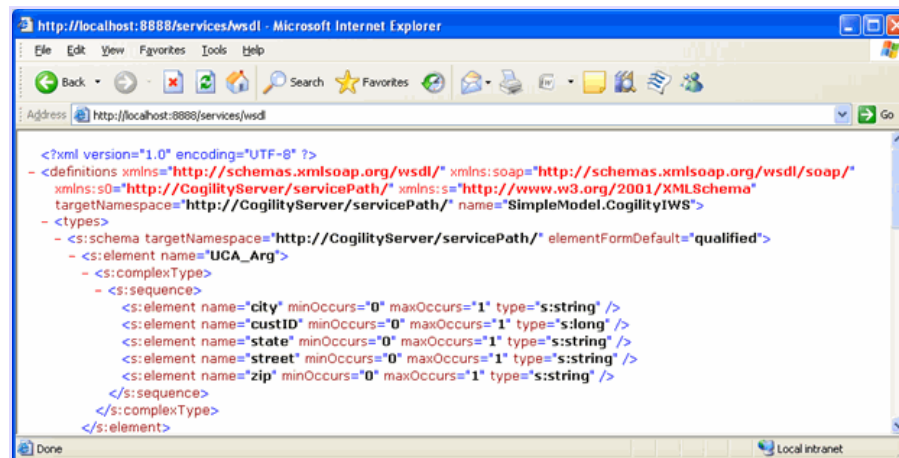
A list of inbound web service deployments appears. There is only the one you created, CogilityIWS. For each deployment in the list, in the Deployed Web Services column, there is a drop-down list of all the inbound web services associated with that deployment. Again, there is only one inbound web service, UpdateCustomerAddress_IWS and it is selected.

- Under the **SOAP Link** column heading, click on **/services/soap**.

As you recall, the context root of the web service deployment was specified as **services** in the model. This means that the web services are located at this path relative to the URL of the application server. The next path, **soap** locates the web services definition language (WSDL) file that describes the web service. The deployment, CogilityIWS appears in a new window, as in the figure below. This deployment contains the UpdateCustomerAddress_IWS inbound web service.




3. Under **Service Definition**, click the **WSDL** link.
The WSDL file for the web service displays, as shown below.




4. Click on the browser's **Back** button to go back to the previous page.
5. Under **Operations**, click the **UpdateCustomerAddress_IWS** link.
The web service operation describes the UpdateCustomerAddress_IWS inbound web service. You recognize the UCA_Arg and its attributes in the Sample SOAP request field

and the UCA_Result message in the Sample SOAP response field as they are described in XML.



SimpleModel.CogilityIWS : UpdateCustomerAddress_IWS



Links:

[WSDL \(service definition\)](#)
[Back to Operations List](#)

Sample SOAP request:

```

POST /services/soap HTTP/1.1
Host: localhost:8888
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://CogilityServer/servicePath/UpdateCustomerAddress_IWS"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
  <UCA_Arg xmlns="http://CogilityServer/servicePath/">
    <city>string</city>
    <custID>integer</custID>
    <state>string</state>
    <street>string</street>
    <zip>string</zip>
  </UCA_Arg>
</soap:Body>
</soap:Envelope>

```

Sample SOAP response:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
  <UCA_Result xmlns="http://CogilityServer/servicePath/">
    <description>string</description>
  </UCA_Result>
</soap:Body>
</soap:Envelope>

```

6. Close this second browser window. Leave the first one opened.

Executing the web service

In this section you use the inbound web service to create an Address and associate it with a Customer.

To execute a web service:

1. In Cogility Insight, in the **Execute** menu, select **Cogility WebServices**.
2. From the **Deployments** drop-down list, select **SimpleModel.CogilityIWS** (already selected).
3. From the **Web Services** drop-down list, select **SimpleModel.UpdateCustomerAddress_IWS** (already selected).

4. Under **Input parameters**, enter the following values:
 - a. In the **city** field, enter **Sunnyvale**.
 - b. In the **custID** field, enter the **customerID** value you noted in step 8b on page 84 (in this example, customerID 1601).
 - c. In the **state** field, enter **CA**.
 - d. In the **street** field, enter **530 Lakeside Dr.**
 - e. In the **zip** field, enter **94085**.
5. Click the **Execute** button.

This sends the SOAP request encoded with the input values you entered and executes the web service. The enterprise application looks up the Customer object by customerID and associates the Address created in the previous step. The web service returns a SOAP response to Cogility Insight with the description attribute value populated as shown in the last line below.

Select an IWS Deployment (1):
SimpleModel.CogilityIWS

Select an IWS (1):
SimpleModel.UpdateCustomerAddress_IWS

Input Parameters for SimpleModel.UpdateCustomerAddress_IWS

Argument Name	Argument Value	Type
city	Sunnyvale	string
custID	33901	long
state	CA	string
street	530 Lakeside Drive	string
zip	94085	string

ExecuteReset

6. From the **View** menu, select **Business Objects**.
7. From the **Business Objects** drop-down list, select **SimpleModel.MIM.Customer** and click **Search**.
8. Select the record for the Customer you just created (in this example, customerID 33901) by clicking on the link in the **Business Object** column.
The Business Object Details window appears.
9. From the **Associations** drop-down list, select **address**.

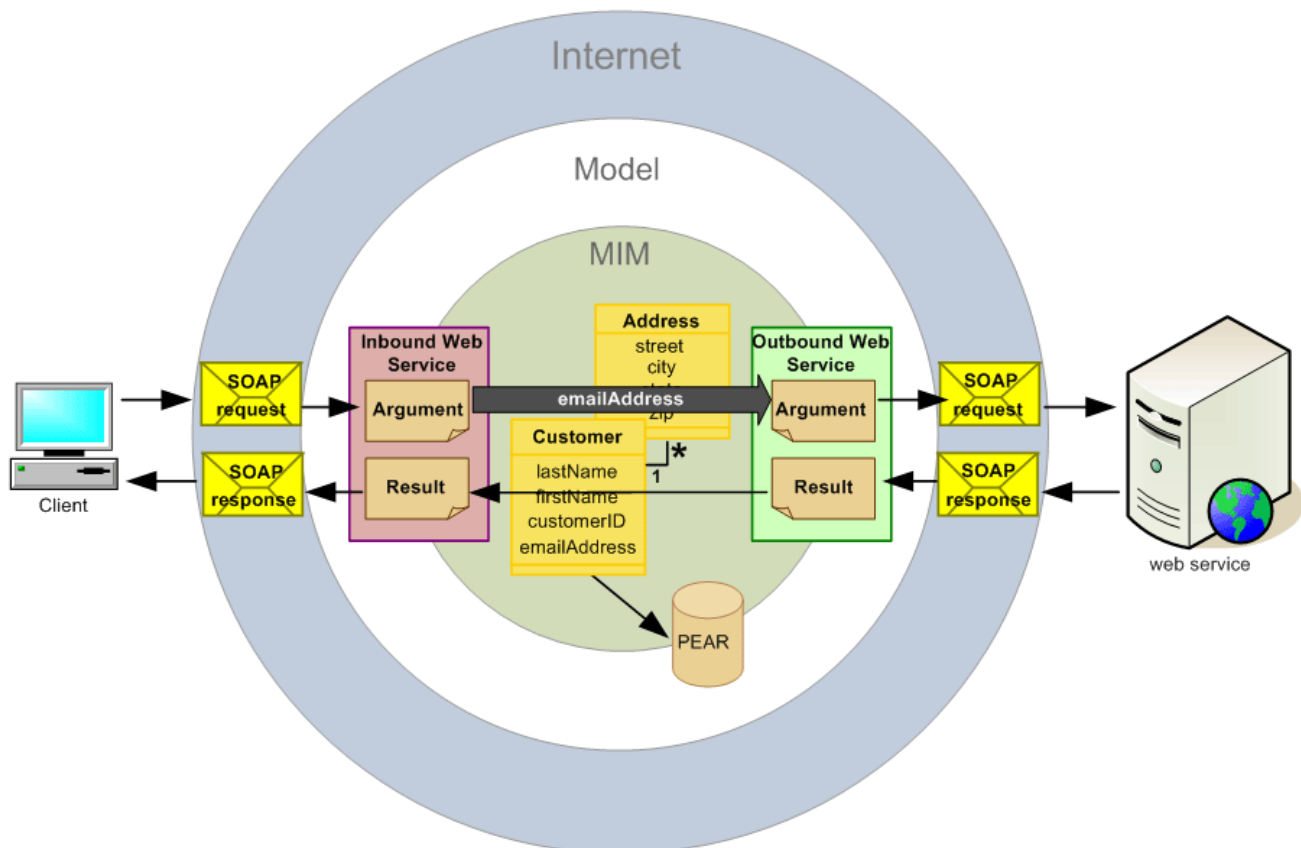


Outbound web services

Outbound web services in Cogility Modeler invoke third party web services running on any platform. In this chapter, you create an outbound web service artifact that invokes a web service to validate an e-mail address.

You may call your outbound web service with a Java action in the action semantics of a state machine or an inbound web service. You call the e-mail validation outbound web service from within the inbound web service you created in [“Inbound web services” on page 75](#).

The outbound web service tries to validate the e-mail address attribute of a Customer; if the e-mail address is valid, the Customer object gets updated.




To support this effort, you need to extend the information model again, adding an attribute for the e-mail address to the Customer class. For more information about outbound web services, see the guide, *Modeling with Cogility Studio*.

Importing the existing model

In the last chapter, you developed a model called SimpleModel. The steps in this chapter start with that model and add more features. If you do not have that version of the model loaded in Cogility Modeler, you can import it with the following procedure. If you are continuing from “Inbound web services” on page 75, skip to “Updating the information model” on page 92.

To import the existing model:

1. If you have loaded in Cogility Modeler a model other than the SimpleModel created in “Inbound web services” on page 75, close Cogility Modeler and follow the steps under “Loading the model repository” on page 11.
2. If Cogility Modeler is not running, follow the steps under “Logging in” on page 12.
3. In Cogility Modeler, from the **Selection** menu, select **Import/Export > Import from Single File**.
4. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial directory, select **SimpleModel_Chapter5.cog** and click **Open**.
%DCHOME% is the directory where you installed Cogility Studio.
5. In the dialog that prompts you, click **Yes** to make the model visible in Cogility Modeler.
6. In the tree view, select the **SimpleModel** model container and click the **Place Model Under CM** button .

You are now ready to follow the procedures of this chapter.

Updating the information model

The outbound web service you create in this chapter attempts to validate an e-mail address. You need to add the e-mail address attribute to your model.

The Customer class currently has attributes for firstName, lastName and customerID. In this section, you add the emailAddress attribute to the Customer class.

To add the new e-mail attribute to the Address class:

1. In Cogility Modeler’s tree view, click the plus signs (+) to open the **Domain Models**, **MIM**, **MIM Classes** and select the **Customer** class.
The content view displays the Customer class editor.
2. In the content view, under the **Information Model Class** tab, above the **Attributes** field, click **New**.
 - a. In the dialog window, for **Name**, enter **emailAddress**.
 - b. For **Type**, select **String** and click **OK**.

The attribute appears in the Attributes field, as shown in the figure below.

The screenshot shows the 'Customer' class in the Cogility Modeler. The 'Attributes' field lists four attributes: 'customerID [Chapter4]', 'emailAddress (2008.07.28-13:08:18:0390)', 'firstName [Chapter1]', and 'lastName [Chapter1]'. Below this, the 'Attribute' details for 'emailAddress' are shown, including its name, size (0), type (String [MAL 2004.04.10]), and a checkbox for 'Is Sequenceable'.

Your model is now ready to include the new outbound web service.


Creating the outbound web service

A web service definition language (WSDL) document defines a web service. It is an XML description of all the information needed to access a web service. To create your outbound web service, you import the web service's WSDL file into your model.

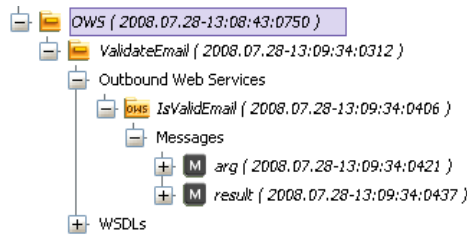
Importing a WSDL from a file

When you import the WSDL from a URL or file, the outbound web service model artifacts are created automatically.

To create the outbound web service:

1. In the Cogility Modeler window, select the **SimpleModel** model container and click the **Add a Package**  button.
Typically, you might have several outbound web services, and you want to group them in a package or container. When you add a package, an **UntitledPackage** is created within the model container.
2. In the content view, under the **Package** tab, in the **Name** field, enter **OWS**.
3. In the tree view, select the **OWS** package, right-click and select **Actions > Import WSDL from File**.
4. Navigate to the **%DCHOME%\Examples\SimpleModel\Tutorial\OWS** folder, select **ValidateEmail.wsdl** and click **Open**.
5. Click **OK** to close the confirmation message window.

The imported outbound web service appears in the tree view in a package named for its WSDL file. Click the plus signs (+) to expand its artifacts.



6. In the tree view under the **OWS** package and **ValidateEmail** package, click the plus signs (+) to expand **WSDLs**, **ValidateEmail**, **WebService Deployments** and select the **ValidateEmailSoap** web service deployment.

The content view displays the deployment information for the outbound web service.

WS Deployment	OWS Deployment Handler Chain	Consistency	Comments	Editions
Name: ValidateEmailSoap				
Endpoint: http://www.websvcx.net/ValidateEmail.asmx				
WSDL: ValidateEmail { 2008.07.28-13:09:34:0312 }				
WSDL Operation				
Name: ValidateEmail				
WS Service: ValidateEmail				
WS Port: ValidateEmailSoap				
WS Namespace: http://www.websvcx.net				

The ValidateEmail web service takes an argument with a single string, the e-mail address. It returns a Boolean. As shown in the figure above, it is hosted at the following endpoint: <http://www.websvcx.net/ValidateEmail.asmx>. You could have also imported the web service from this URL.

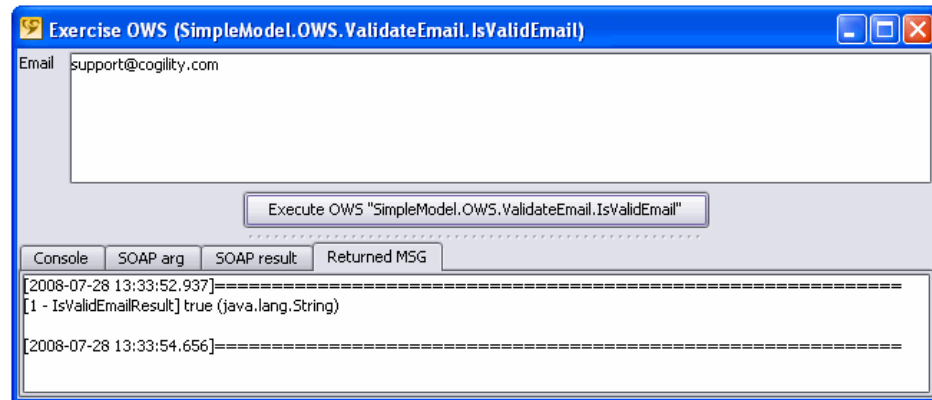
Testing the web service

Cogility Modeler includes a facility to test the web service you imported.

To test the web service:

1. In the tree view, in the **ValidateEmail** package, above **WSDLs**, click the plus sign (+) to expand **Outbound Web Services** and select the **IsValidEmail** web service.
2. Right-click and select **Actions > Try Outbound WebService**.
The Exercise OWS window appears.
3. In the **Email** field, enter a valid e-mail address (for example, support@cogility.com) and click the button labeled **Execute OWS "SimpleModel.OWS.ValidateEmail.IsValidEmail"**.

4. Click the **Returned MSG** tab.



This verifies both that the web service is live and working, and that the e-mail address you submitted is valid. Now you can use the outbound web service in your model.

5. Close the Exercise OWS window.

Updating the inbound web service

Your SimpleModel includes an inbound web service that updates a customer address. So far you have added the emailAddress attribute to the Customer class and you have an outbound web service ready to validate the value of the emailAddress attribute at run time. You need to update the inbound web service with a new parameter to accommodate the emailAddress attribute and new logic to validate it with the outbound web service.

Adding a parameter to the inbound web service

The inbound web service requires a new parameter to hold the emailAddress attribute value.

To add the parameter to the inbound web service:

1. In the Cogility Modeler's tree view, click the plus sign (+) to expand **Inbound Web Services** and select the **UpdateCustomerAddress_IWS** web service.
2. In the content view, under the **IWS Arguments/Results** tab, above the **Argument Message** field, click **Edit**.
The argument message editor window appears.
3. Above the **Attributes** field, click **New**.
 - a. In the dialog window, for **Name**, enter **emailAddress**.
 - b. For **Type**, select **String** and click **OK**.
 - c. Close the argument message editor window.

The new attribute is added to the Attributes field.

Updating the inbound web service logic

To call the outbound web service that validates the e-mail address, you use an action semantics keyword in the logic of your inbound web service. In this case, you use the `callOWS` keyword. The signature is as follows:

```
callOWS <web service name> (<argument1>, <argument2>);
```

This keyword is followed by the fully-qualified outbound web service name `<web service name>` and a list of named arguments `<argument*>` in parentheses. More than one arguments are separated by commas.

To update the IWS logic:

1. In the Cogility Modeler's tree view, under **Inbound Web Services**, select **UpdateCustomerAddress_IWS**.
2. In the content view, under the **WS** tab, in the **Action** field, add the following action semantics at the end of the existing code:

```
-- additional email validation
res : SimpleModel::OWS::ValidateEmail::IsValidEmail::result;
res := callOWS SimpleModel::OWS::ValidateEmail::IsValidEmail(
    Email: input.emailAddress);

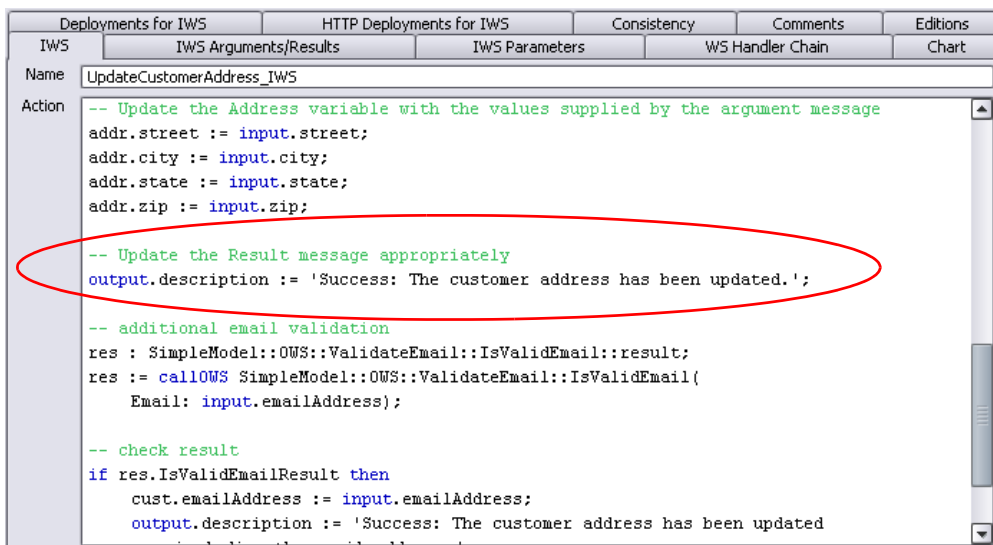
-- check result
if res.IsValidEmailResult then
    cust.emailAddress := input.emailAddress;
    output.description := 'Success: The customer address has been
updated
including the email address.';
else
    output.description := 'Success: The customer address has been
updated,
but the email address could not be validated.';
endif;
```

Note the use of `callOWS` to call to the web service. The argument, `IsValidEmail`, includes the fully-qualified path to the outbound web service.

3. Remove the following lines (circled in the picture below):

```
--Update the Result message appropriately

output.description := 'Success: The Customer address has been
updated.';
```



The condition block that you added makes these lines unnecessary.



4. Click the **Consistency** tab to verify the action semantics.

There should be no inconsistencies. If there are, try recopying the action semantics into the Action field. Also, remove any trailing new line characters after the last line.

Turn over and push

Before you run the model, turn it over and push it into execution.

To turn over and push the model:

1. In the tree view, select the **SimpleModel** model container.
2. In the Cogility Modeler tool bar, click the **Turn Over** button .
3. In the Turnover Operation dialog, for **Turnover Version Name**, enter **Chapter6**.
4. In the Comments field, enter some comments and click **OK**.
A successful notification appears.
5. Click **OK**.
6. If you have already pushed a model into your run-time repository, remove the current model from the database. (Skip this step if you have not yet pushed a model.)
 - Navigate to the %DCHOME%\MB\Scripts\Dbtools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.
A command window appears and shows the Drop Tables utility executing. Close the window when it finishes.
7. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
8. With the **SimpleModel** model container selected, click the Push button .
This pushes the enterprise application onto the application server, and the model objects and schema into the run-time repository.
9. In the Select Source of Deployment Parameters, select the object that matches your configuration and click **Use Deployment**.
When the push operation completes, a dialog appears notifying you of successful deployment.
10. Click **OK**.

Executing the model

After you create another Customer object with Cogility Message Traffic Viewer, you run Cogility Insight to test the inbound web service.

Running Cogility Message Traffic Viewer

As in [“Inbound web services” on page 75](#), you create another Customer object with its own unique customerID by publishing the same JMS message from the previous chapters.

To run Cogility Message Traffic Viewer:

1. If your application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.

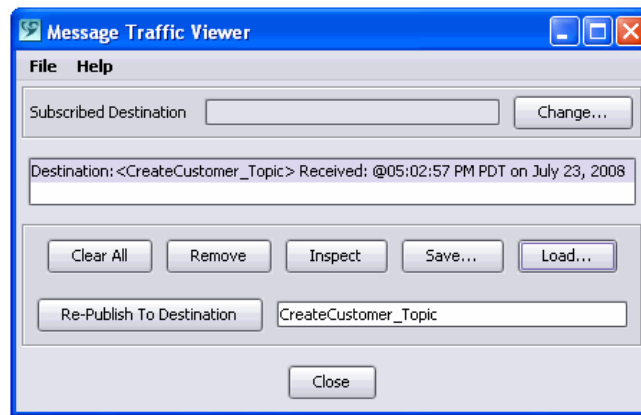
A console window displays. When it shows the server is running, you may proceed with the following steps.

2. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Manager> JMS Message Viewer**.

You must select the Cogility Message Traffic Viewer appropriate for your application server.

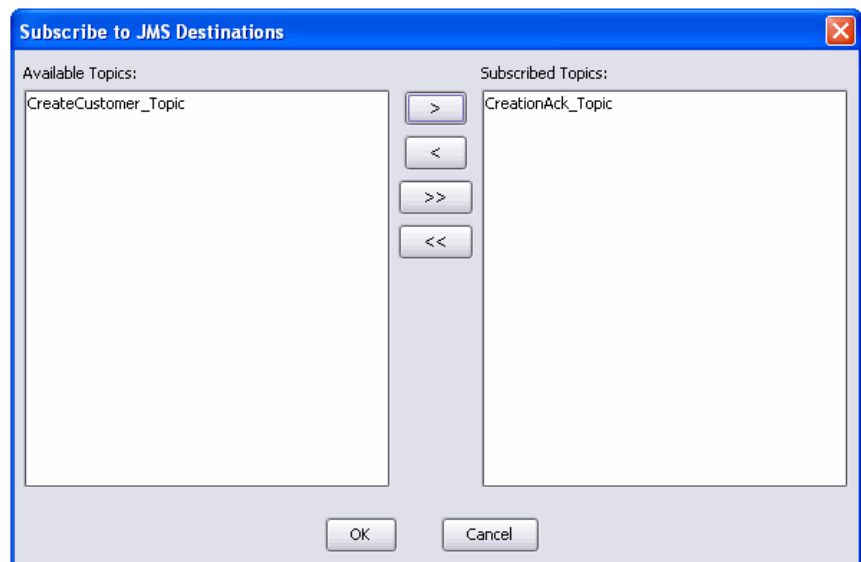
3. In the Cogility Message Traffic Viewer window, click **Load**.
4. Navigate to the file you saved in step 6 under **“Running Cogility Message Traffic Viewer” on page 59** and click **Open**.

Cogility Message Traffic Viewer displays the message destinations from the .msg file. The only destination defined in your model, `CreateCustomer_Topic`, is selected in view.



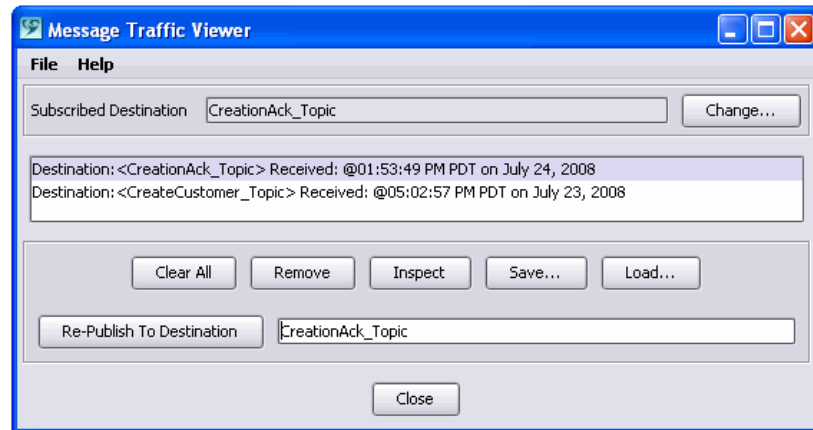
You resend the message with new customer information (you saved that message in step 6 under **“Running Cogility Message Traffic Viewer” on page 59**). At the same time Cogility Message Traffic Viewer listens for the acknowledgment message from your enterprise application.

5. Next to the **Subscribed Destination** field, click **Change**.
6. In the dialog that appears, specify **CreationAck_Topic** as the Subscribed Topic, and click **OK**.



7. In Cogility Message Traffic Viewer, click **Re-Publish To Destination**.

The acknowledgement message from the enterprise application, `CreationAck_Topic` appears in Cogility Message Traffic Viewer at the top of the list as show in the following figure.



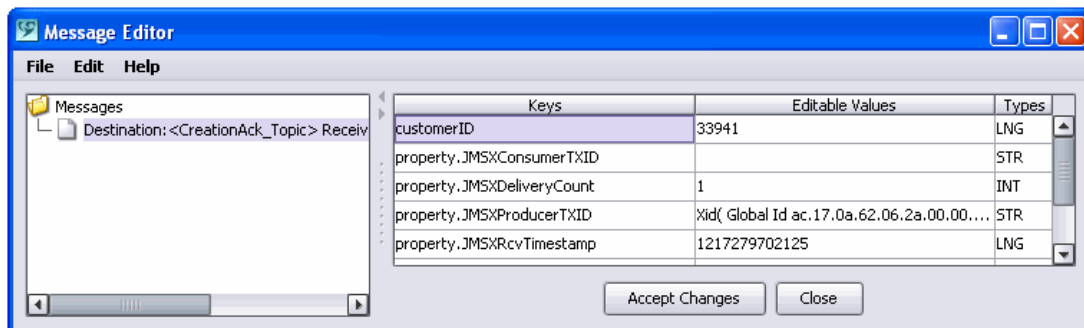
8. With `<CreationAck_Topic> Received` selected, click **Inspect**.

Cogility Message Editor appears.

a. In the tree view, under **Messages**, select **Subject <CreationAck_Topic> Received**.

As shown in the figure below, the acknowledgement message includes a `customerID`. This is a new Customer object, distinct from the one you created in the previous chapter, and it should have a new `customerID`.

b. Make a note of the `customerID` as you will refer it later in these exercises.



Next, you use the `customerID` generated for the new Customer to associate the Customer with an Address and validate an e-mail address.

Running Cogility Insight

Cogility Insight is a web-based application that lets you monitor your enterprise application as it executes.

To run Cogility Insight:

1. From the **Start** menu, select **All Programs > Cogility Studio > Insight Web Access**.

a. For **UserName**, enter **admin**.

b. For **Password**, enter **password** and click **Submit**.

2. In the **Execute** menu, select **Cogility WebServices**.
3. From the **Deployments** drop-down list, select **SimpleModel.CogilityIWS**.
4. From the **Web Services** drop-down list, select **SimpleModel.UpdateCustomerAddress_IWS**.
5. Under **Input parameters**, enter the following values:
 - a. In the **city** field, enter **Sunnyvale**.
 - b. In the **custID** field, enter the **customerID** value you noted in step 8b on page 99 (in this example, customerID 1641).
 - c. In the **emailAddress** field, enter **nomail@cogility.goof**.
You first try an e-mail address that doesn't work to see if the behavior is correct.
 - d. In the **state** field, enter **CA**.
 - e. In the **street** field, enter **530 Lakeside Dr., Suite 260**.
 - f. In the **zip** field, enter **94085**.
6. Click the **Execute** button.

This sends the SOAP request encoded with the input values you entered and executes the web service. The enterprise application looks up the Customer object by customerID and associates the Address created in the previous step. The web service returns a SOAP response to Cogility Insight with the description attribute value populated as shown in the last line below. The invalid e-mail address produced the proper behavior.

Select an IWS Deployment (1):
SimpleModel.CogilityIWS

Select an IWS (1):
SimpleModel.UpdateCustomerAddress_IWS

Input Parameters for SimpleModel.UpdateCustomerAddress_IWS

Argument Name	Argument Value	Type
city	Sunnyvale	string
custID	33941	long
emailAddress	nomail@cogility.goof	string
state	CA	string
street	530 Lakeside Drive	string
zip	94085	string

ExecuteReset

IWS Execution Results

description: "Success: The customer address has been updated, but the email address could not be validated."

7. From the **View** menu, select **Business Objects**.
8. From the **Business Objects** drop-down menu, select **SimpleModel.MIM.Customer** and click **Search**.

If you have followed the exercises in the previous chapters, this is the fifth time you have created a Customer object, reusing the same JMS message. If you have not cleared your database, all five Customer objects appear, and three have a customerID.

9. Select the record for the Customer whose value you noted in step 8b on page 99 (in this example, customerID 33941) by clicking on the **SimpleModel.MIM.Customer** link in the **Business Object** column.

The Business Object Details window appears.

Although the new Customer has been created, the emailAddress attribute value is not populated, as expected.

Details for Business Object: **BO^SimpleModel.MIM.Customer^-42f11b9e:11b6b759636:-7ffa**

Source: SimpleModel.MIM.Customer

Property Name	Property Value
customerID	33941
emailAddress	
firstName	Jenny
lastName	Watts
poId	BO^SimpleModel.MIM.Customer^-42f11b9e:11b6b759636:-7ffa

Associations (2):
-- Select an association --

Process Status for Business Object: **BO^SimpleModel.MIM.Customer^-42f11b9e:11b6b759636:-7ffa**
This business object has no behavior instances.

10. Repeat the previous steps under “Running Cogility Insight” on page 99, this time enter **support@cogility.com** (a valid e-mail address) in step 5c.

This time the Customer emailAddress attribute is populated with a valid e-mail address.

Details for Business Object: **BO^SimpleModel.MIM.Customer^-42f11b9e:11b6b759636:-7ffa**

Source: SimpleModel.MIM.Customer

Property Name	Property Value
customerID	33941
emailAddress	support@cogility.com
firstName	Jenny
lastName	Watts
poId	BO^SimpleModel.MIM.Customer^-42f11b9e:11b6b759636:-7ffa

Associations (2):
-- Select an association --

Process Status for Business Object: **BO^SimpleModel.MIM.Customer^-42f11b9e:11b6b759636:-7ffa**
This business object has no behavior instances.



Class operations

An operation defines an interface to reusable behavior of a class. An operation has a signature that consists of the following:

- An operation name
- Zero or more operation parameters (name and type)
- Zero or one operation return type (if anything is returned at all)

An operation contains a method body that defines the operation's implementation.

In this chapter, your model acknowledges receipt of and processes a customer service request. The service request is processed according to a service level agreement (either gold or bronze). You define a class hierarchy with a service level agreement as the parent class, and gold and bronze agreements as the child classes.

The lower levels in a class hierarchy inherit information from the levels above. For example, attributes defined for the parent are inherited by the child classes. You define an operation for the parent with a default method body. Because of inheritance, the children inherit the operation's signature but not the method body implementation. Each child has its own method body that extends or overrides the parent's method body.

In this chapter, you create an operation for the service level agreement parent class, then override its method body in the bronze and gold service level child classes. To do that you need to define the new classes and relate them in a class hierarchy.


For more information about operations and about extending or overriding method bodies, see *Using Actions in Cogility Studio*.

Importing the existing model

In the last chapter, you developed a model called SimpleModel. The steps in this chapter start with that model and add more features. If you do not have that version of the model loaded in Cogility Modeler, you can import it with the following procedure. If you are continuing from [“Outbound web services” on page 91](#), skip to [“Creating a new class diagram” on page 104](#).

To import the existing model:

1. If you have loaded in Cogility Modeler a model other than the SimpleModel created in [“Outbound web services” on page 91](#), close Cogility Modeler and follow the steps under [“Loading the model repository” on page 11](#).
2. If Cogility Modeler is not running, follow the steps under [“Logging in” on page 12](#).
3. In Cogility Modeler, from the **Selection** menu, select **Import/Export > Import from Single File**.




4. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial directory, select **SimpleModel_Chapter6.cog** and click **Open**.
%DCHOME% is the directory where you installed Cogility Studio.
5. In the dialog that prompts you, click **Yes** to make the model visible in Cogility Modeler.
6. In the tree view, select the **SimpleModel** model container and click the **Place Model Under CM** button .

You are now ready to follow the procedures of this chapter.

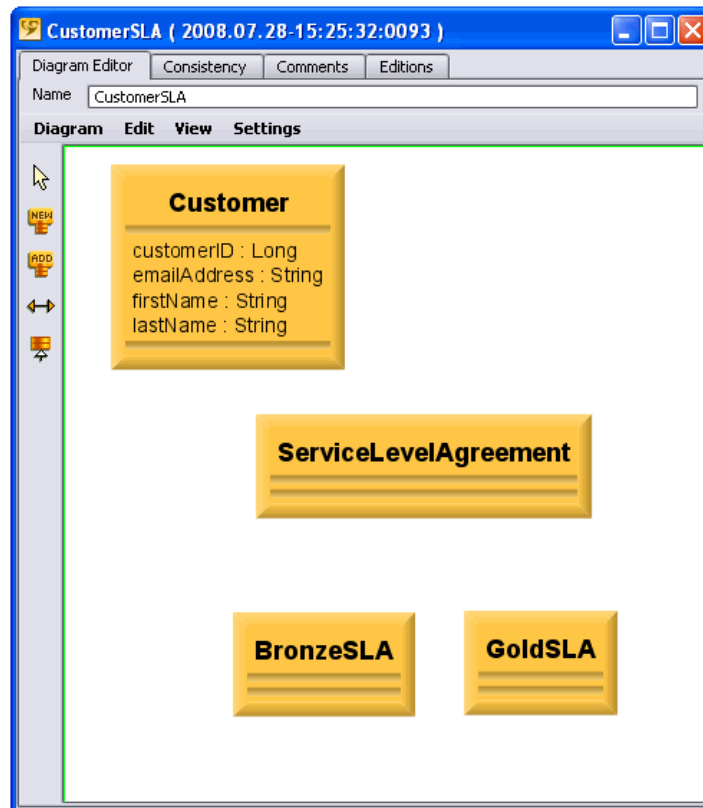
Creating a new class diagram

Set up a class diagram that associates a Customer with a ServiceLevelAgreement, and two subclasses, GoldSLA and BronzeSLA that inherit from the ServiceLevelAgreement.

To define the class diagram:

1. Select the **MIM**, and click the **Add Class Diagram** button .
The class diagram editor appears in the content view.
2. In class diagram editor, in the **Name** field, enter **CustomerSLA**.
3. In the diagram editor tool bar, click the **Create New Class** button .
4. Place the cursor in the diagram editor window and click again.
An untitled class icon appears in the diagram editor.
5. Double-click the class label, **UntitledMIMClass**.
6. Enter **ServiceLevelAgreement** for the new label.
7. Repeat steps 3 through 6 to create two more classes, named **GoldSLA** and **BronzeSLA**.
8. In the diagram editor tool bar, click the **Add Existing Class** button .
9. Place the cursor in the diagram editor window and click again.
A selection dialog appears.
10. Select the **Customer** class and click **OK**.


11. Arrange the classes in the class diagram as in the picture below.



Define the class hierarchy

Define the class hierarchy with ServiceLevelAgreement as the parent class, GoldSLA and BronzeSLA as the two subclasses.


To define the class hierarchy:

1. In the class diagram editor tool bar, click the **Define Superclass** button .
2. Click the **GoldSLA** icon, then move the cursor to the **ServiceLevelAgreement** icon and click again.
A class inheritance arrow is drawn from the GoldSLA class to the ServiceLevelAgreement class.
3. Repeat the previous step for the **BronzeSLA** class, setting **ServiceLevelAgreement** as its superclass.

Define the association

The Customer class must be associated with the ServiceLevelAgreement class.

To create the association:

1. In the class diagram tool bar, click the **Create New Association** button .
2. Place the cursor over the **Customer** class and click.
3. Move the cursor over the **ServiceLevelAgreement** class and click again.

When you release, a selection window appears.

4. In the selection window, select **Association** and click **OK**.

An undefined association appears as a line between the Customer and ServiceLevelAgreement classes.

5. Double-click the association line.

An association editor appears.

- a. For **Source Role**, leave the default, **customer**.

The role describes the relationship that a class has to another. The source role owns the relationship.

- b. For **Source Multiplicity**, enter *****.

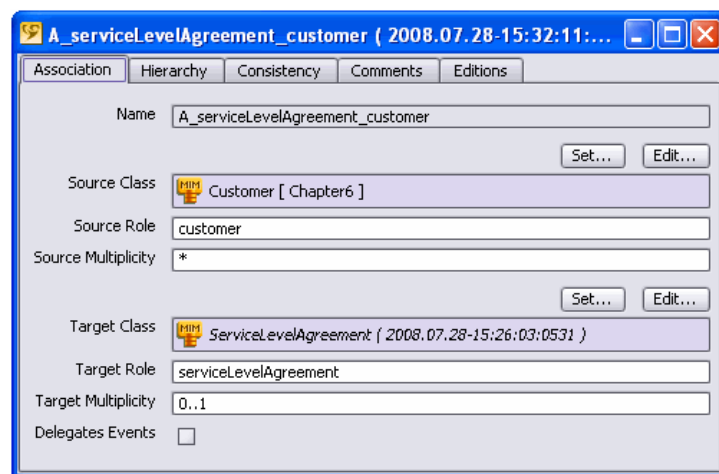
The ServiceLevelAgreement may have many Customers. The source multiplicity refers to how many instances of the source role may pertain to the target role.

- c. For **Target Role**, leave the default, **serviceLevelAgreement**.

The target role describes the target class' relationship to the source class.

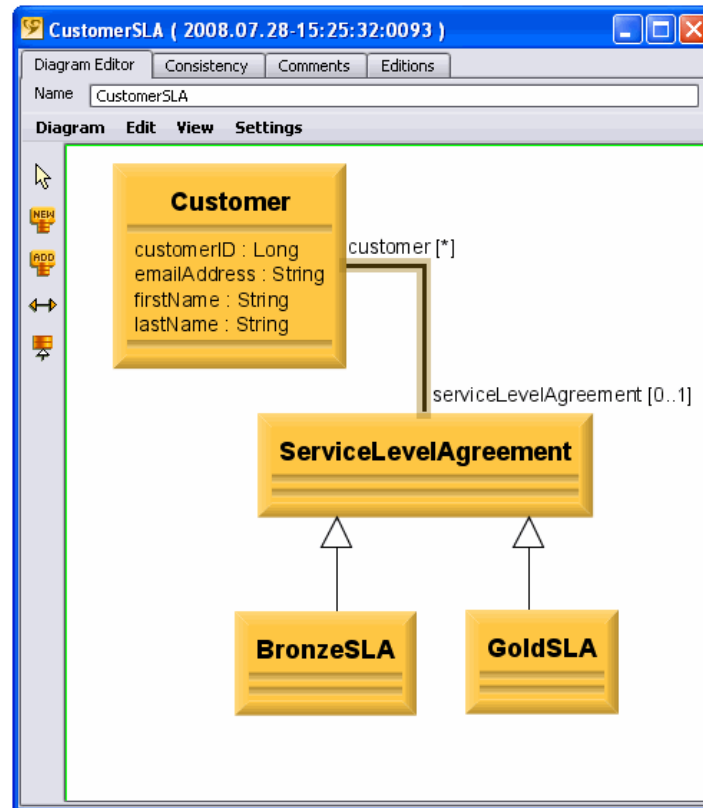
- d. For **Target Multiplicity**, enter **0..1**.

A Customer may have zero or one ServiceLevelAgreement objects. The association editor should look like the picture below.



6. Close the association editor window.




Your new class diagram should appear as follows.



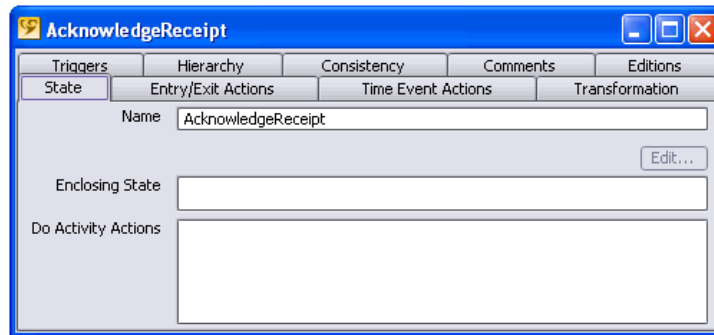
Defining class behavior

In this section you define the `ServiceLevelAgreement` class behavior to acknowledge receipt of a customer service request and process the request based on the customer's service level agreement.

To define class behavior:

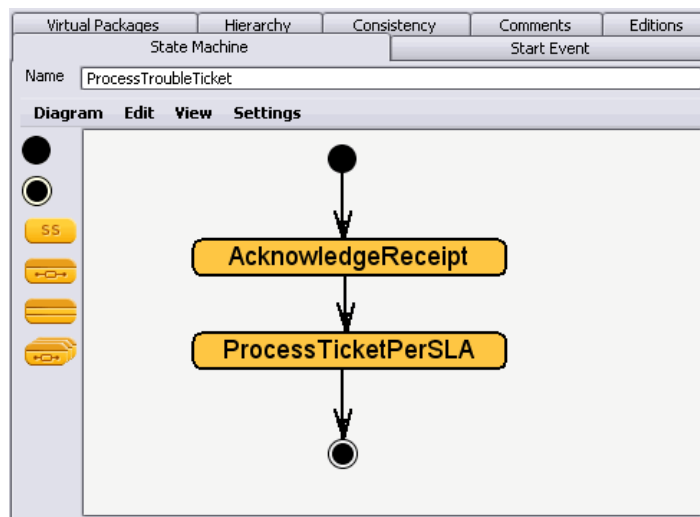
1. In Cogility Modeler's tree view, under the **Domain Model**, under the **MIM Classes**, select the **ServiceLevelAgreement** class and click the **Add a Behavior** button .
- A new behavior displays in the tree view, and a new state machine definition appears in the content view.
2. In the content view, for **Name**, enter **ProcessTroubleTicket**.
3. Drag an **Initial State** icon  from the tool bar into the diagram editor window.
4. Drag two **Simple State** icons  into the diagram.
5. Double-click on the first simple state.

- a. In the editor that displays, in the **Name** field, enter **AcknowledgeReceipt**.



- b. Close the editor window.
6. Repeat step 5 for the second simple state, name the state **ProcessTicketPerSLA**.
 7. Drag a **Final State** icon into the diagram.
 8. Select the **Initial State**, hold down the **Ctrl** key, drag the cursor to the **AcknowledgeReceipt** state and release to create a transition.
 9. Select the **AcknowledgeReceipt** state, hold down the **Ctrl** key, drag the cursor to the **ProcessTicketPerSLA** and release to create a transition.
 10. Select the **ProcessTicketPerSLA** state, hold down the **Ctrl** key, drag the cursor to the **Final State** and release to create a transition.

Your state diagram should like the following.



Defining operations

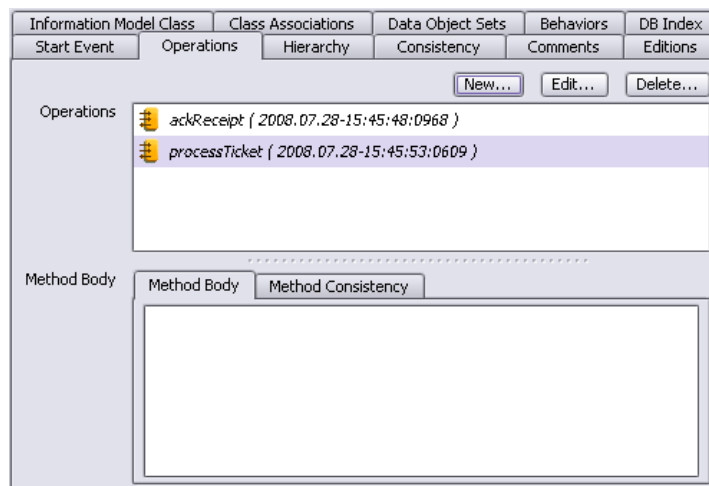
As described at the beginning of this chapter, an operation defines an interface to reusable behavior of a class. Now that you have created your class hierarchy for the service level agreement and the state machine that processes the agreements, you define an operation for the `ServiceLevelAgreement` class. For this simple example, the operation will print statements to the console acknowledging receipt and processing.

Defining the method signatures

Each operation has a method signature that describes the operation, its parameters and its return type.

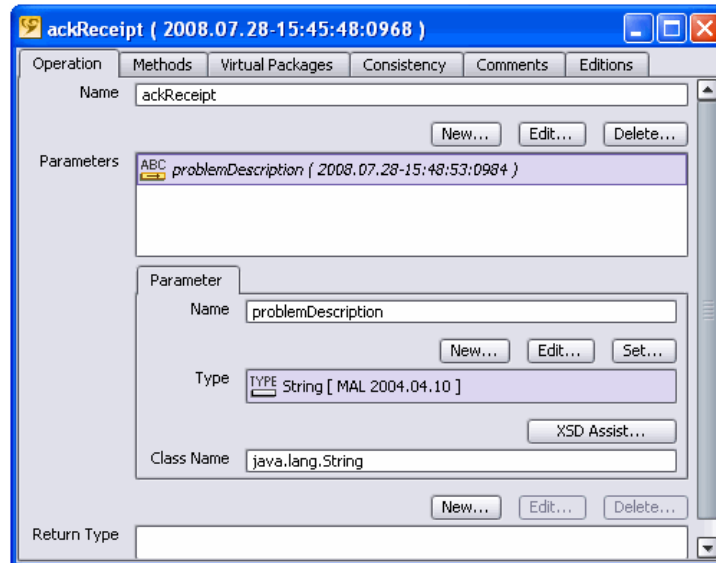
To define an operation:

1. In the Cogility Modeler window, in the tree view, under the **MIM**, select the **ServiceLevelAgreement** class.
 - a. In the content view, in the **operations** tab, click **New**.
 - b. In the dialog window, enter **ackReceipt** and click **OK**.
The name displays in the operations field. The method body field appears.
 - c. Above the **Operations** field, click the **New** button again.
 - d. In the dialog window, enter **processTicket** and click **OK**.



2. In the **operations** field, select the **ackReceipt** operation and click the **Edit** button.
The **ackReceipt** operation editor displays.
 - a. Above the **parameters** field, click **New**.
 - b. In the dialog window, enter **problemDescription** and click **OK**.
The name displays in the parameters field and a **Parameter** tab displays below the field.
 - c. Under the **Parameter** tab, above the **type** field, click the **Set** button.
An artifact selection window displays.
 - d. In the selection window, select **String** and click **OK**.
 - e. Leave the **returnType** field empty.

The operation returns a null return type.



- f. Close the **ackReceipt** operation editor.
3. In the **operations** field, select the **processTicket** operation and click the **Edit** button.
The processTicket operations editor displays.
 - a. Above the **parameters** field, click **New**.
 - b. In the dialog window, enter **problemDescription** and click **OK**.
The name displays in the parameters field and a **Parameter** tab displays below the field.
 - c. Under the **Parameter** tab, above the **type** field, click the **Set** button.
An artifact selection window displays.
 - d. In the selection window, select **String** and click **OK**.
 - e. Leave the **Return Type** field empty.
The operation returns a null return type.
 - f. Close the operation editor.

This completes the method signatures for the operations. Now you need to define the method bodies.

Defining the method bodies

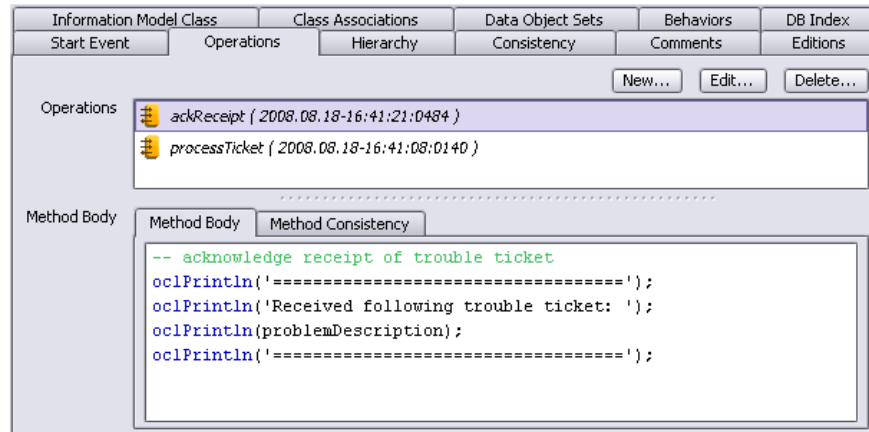
The method body is the implementation of the operation, described with action semantics. A subclass may extend or override the superclass method body. In this example, you define a method body for the parent and overriding method bodies for the child classes.

To define the method body:

1. In the content view, under the **Operations** tab, select the **ackReceipt** operation.
2. In the **Method Body** field, enter the following action semantics:

```
-- acknowledge receipt of trouble ticket
oclPrintln('=====');
oclPrintln('Received following trouble ticket: ');
oclPrintln(problemDescription);
oclPrintln('=====');
```

The Method Body tab should display as in the figure below.



3. In Cogility Modeler's content view, under the **Operations** tab, select the **processTicket** operation.

4. In the **Method Body** field, enter the following action semantics:

```
-- process trouble ticket
oclPrintln('=====');
oclPrintln('Default processing for trouble ticket: ');
oclPrintln(problemDescription);
oclPrintln('=====');
```

5. In the tree view, select the **GoldSLA** class.

6. In the content view, under the **Operations** tab, select the **processTicket** operation.

The subclass inherits the operation's signature. You can extend or override the implementation. In this case, you override the implementation.

7. Under the **Method Body** tab, enter the following action semantics:

```
-- process trouble ticket
oclPrintln('=====');
oclPrintln('GoldSLA processing for trouble ticket: ');
oclPrintln(problemDescription);
oclPrintln('=====');
```

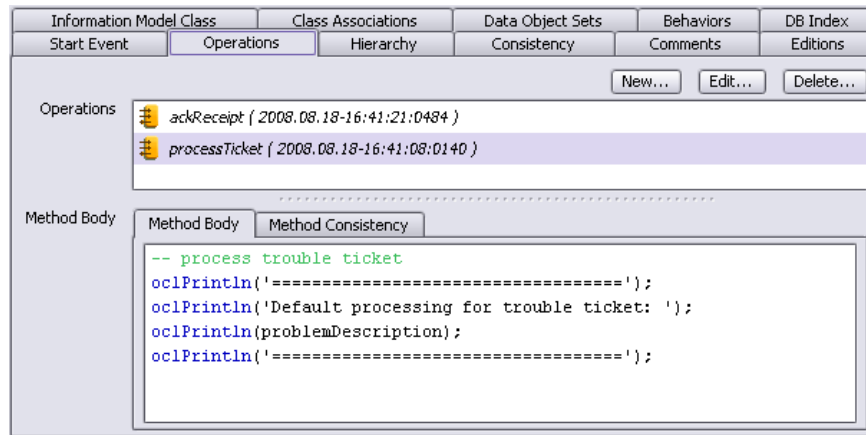
8. In the tree view, select the **BronzeSLA** class.

9. In the content view, under the **Operations** tab, select the **processTicket** operation.

10. Under the **Method Body** tab, enter the following action semantics:

```
-- process trouble ticket
oclPrintln('=====');
oclPrintln('BronzeSLA processing for trouble ticket: ');
oclPrintln(problemDescription);
oclPrintln('=====');
```

The BronzeSLA processTicket operation now has the following signature and method body.



Invoking operations

Now that you have defined the operation and the method bodies that implement it, you must define the business process logic that invokes the operations. Operations can be invoked from within any Cogility Modeler artifact's action semantics. In this case, the operation you have defined is invoked from within the state machine you created in [“Defining class behavior” on page 107](#).

To specify the business process logic:

1. In Cogility Modeler's tree view, under the **ServiceLevelAgreement** class, select the **ProcessTroubleTicket** state machine.

The state machine diagram displays in the content view.

2. In the content view, double-click on the **AcknowledgeReceipt** state.

The AcknowledgeReceipt state editor window displays.

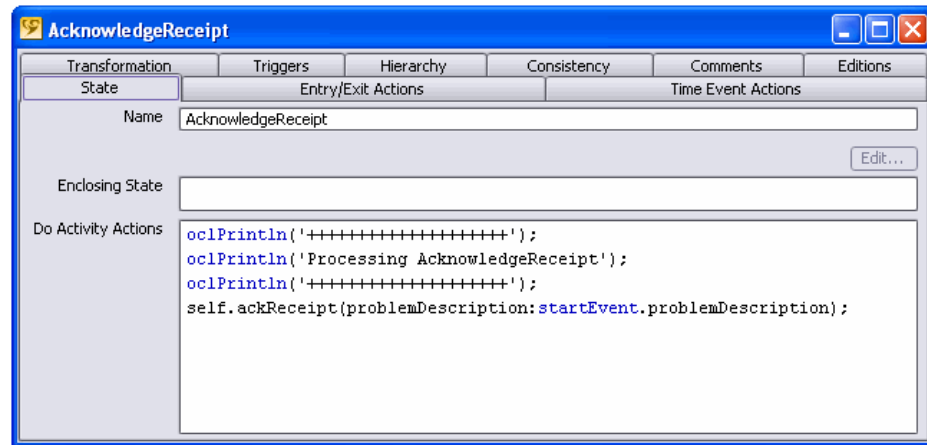
3. Click the **State** tab, and in the **Do Activity Actions** field, enter the following action semantics:

```
oclPrintln('+++++++');
oclPrintln('Processing AcknowledgeReceipt');
oclPrintln('+++++++');
self.ackReceipt(problemDescription:startEvent.problemDescription);
```

The **self** variable is an action semantics keyword that references the instance with which the operation is associated, in this case, the instance of ServiceLevelAgreement. The system looks for an operation called **ackReceipt** associated with that class.

startEvent is an action semantics keyword that references the start event associated with the behavior, in this case TroubleTicket_Event (which you have yet to create). The system looks to that event to find the attribute called **problemDescription**. As described in the syntax above, this provides the value for the parameter (**problemDescription**) being passed to the **ackReceipt** operation.

There is an inconsistency for this state until you define the TroubleTicket_Event.



4. Close the AcknowledgeReceipt state editor window.
5. In the content view, double-click on the **ProcessTicketPerSLA** state.
The state editor window displays.
6. Click the **State** tab, and the **Do Activity Actions** field, enter the following action semantics:

```
oclPrintln('+++++');
oclPrintln('Processing ProcessTicketPerSLA');
oclPrintln('+++++');
self.processTicket(problemDescription: startEvent.problemDescription);
```

This calls the processTicket operation. This state also has an inconsistency until you define the start event for the state machine.

7. Close the state editor window.

Event and message conversion

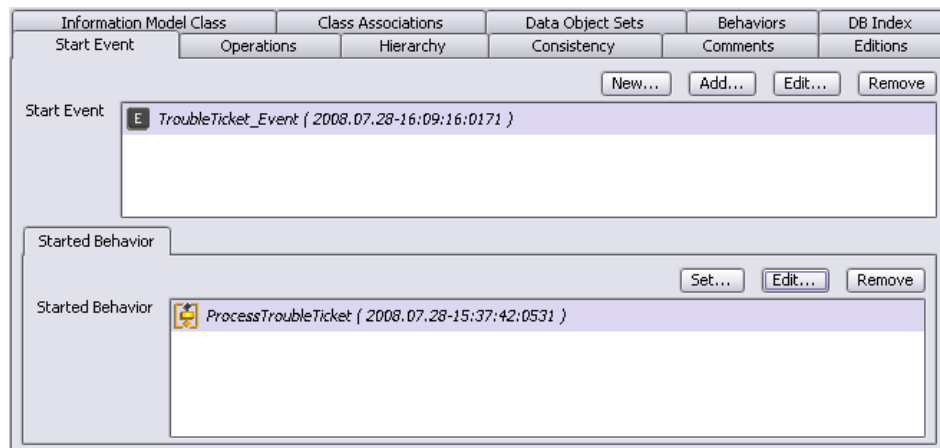
Now that you have defined a behavior, you need a start event to initiate that behavior. In addition, you need a conversion artifact to transmit the information from the external application to the event.

Connecting the behavior to a start event

To connect the behavior to a start event:

1. In Cogility Modeler's tree view, select the **ServiceLevelAgreement** class.
2. In the content view, click the **Start Event** tab, and above the **Start Event** field, click **New**.
The event type selection window displays.
3. Select **Event** and click **OK**.
4. In the dialog window, enter **TroubleTicket_Event** and click **OK**.
The event displays in the Start Event field and the Started Behavior tab displays in the lower part of the view.
5. Above the **Started Behavior** field, click **Set**, select **ProcessTroubleTicket** and click **OK**.

The behavior displays in the Started Behavior field.

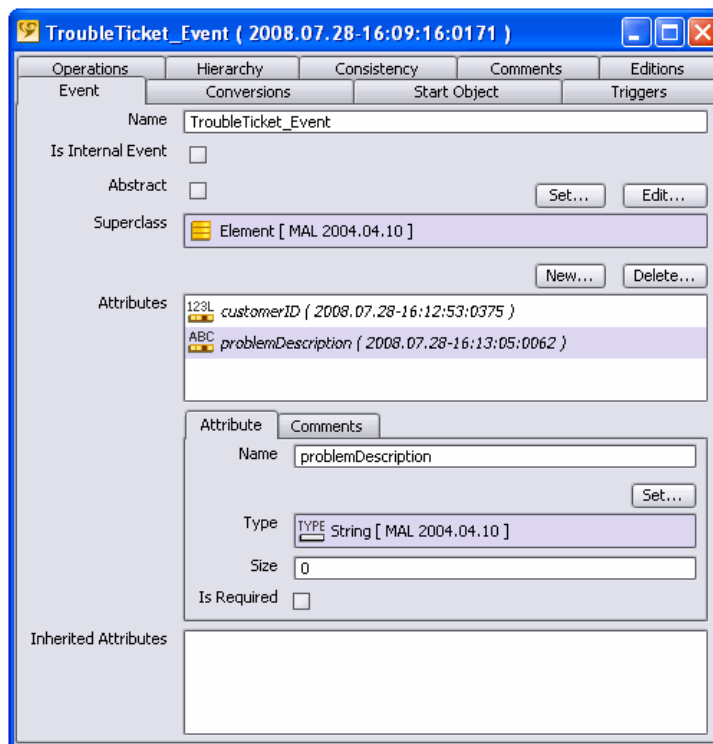


6. Above the **Start Event** field, for **TroubleTicket_Event**, click **Edit**.

The event editor displays.

7. Above the **Attributes** field, click the **New** button.
 - a. In the dialog window, for **Name**, enter **customerID**.
 - b. For **Type**, select **Long** and click **OK**.
8. Above the **Attributes** field, click the **New** button.
 - a. In the dialog window, for **Name**, enter **problemDescription**.
 - b. For **Type**, select **String** and click **OK**.

The event editor appears as in the following figure.



Defining an M2E conversion for the event

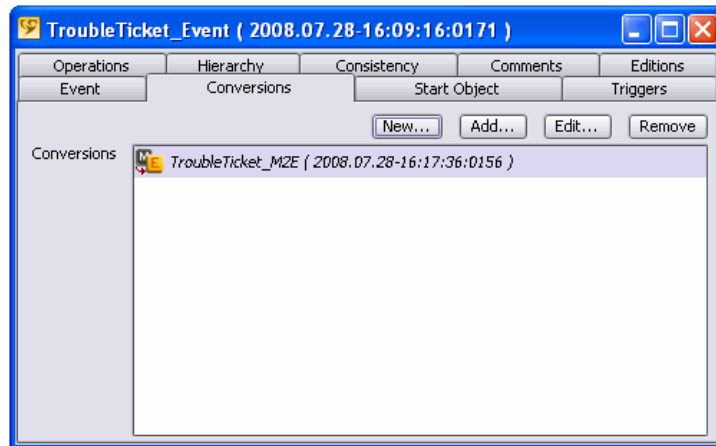
To define an M2E conversion:

1. In the **TroubleTicket_Event** editor, click the **Conversions** tab and click **New**.

A selection window displays.

- a. Select **MessageToEventConversion** and click **OK**.
- b. In the dialog window, enter **TroubleTicket_M2E** and click **OK**.

The conversion displays in the Conversions field.



2. In the event editor, above the **Conversions** field, click the **Edit** button.

The conversion editor window appears.

- a. Click the **Destination** tab, and above the **Destination** field, click **New**.
- b. In the dialog window, enter **TroubleTicket_Topic** and click **OK**.
- c. Click the **Conversion Actions BOL** tab and in the **Business Object Locator** field, enter the following action semantics:

```

cust : SimpleModel::MIM::Customer;
cust := locate SimpleModel::MIM::Customer(customerID:jmsMapMessage
    .getLong('customerID'));

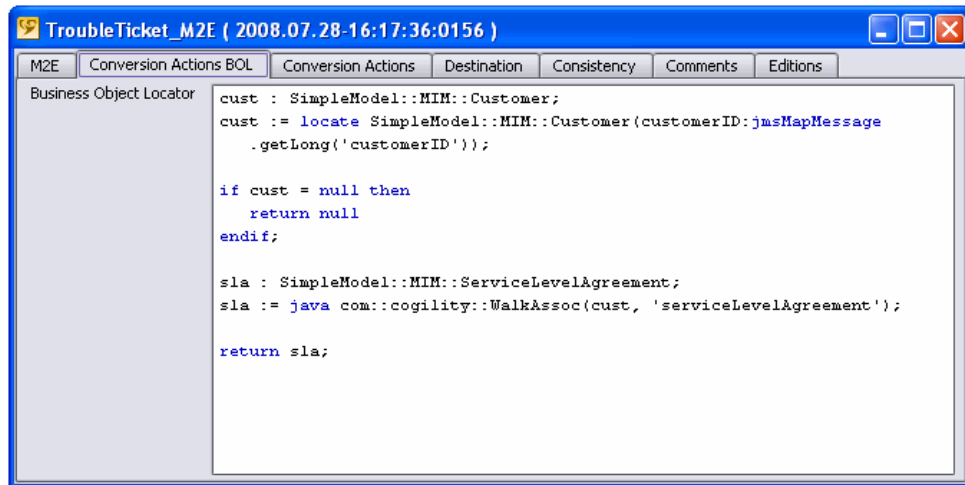
if cust = null then
    return null
endif;

sla : SimpleModel::MIM::ServiceLevelAgreement;
sla := java com::cogility::WalkAssoc(cust,
'serviceLevelAgreement');

return sla;

```

A business object locator is an action semantics expression that uses message data to identify the business object instance in your model that is to receive the event.



3. Close the **Troubleshoot_M2E** conversion editor window.
4. Close the **Troubleshoot_Event** editor window.

The inconsistencies in the ProcessTroubleshoot state machine are resolved.


Creating an inbound web service

Inbound web services serve as input points for various applications or front ends to communicate with your model's enterprise application using a standard interface. In ["Inbound web services" on page 75](#), you created an inbound web service that updated the customer's address. Next, you create an inbound web service that associates a customer with a service level agreement.

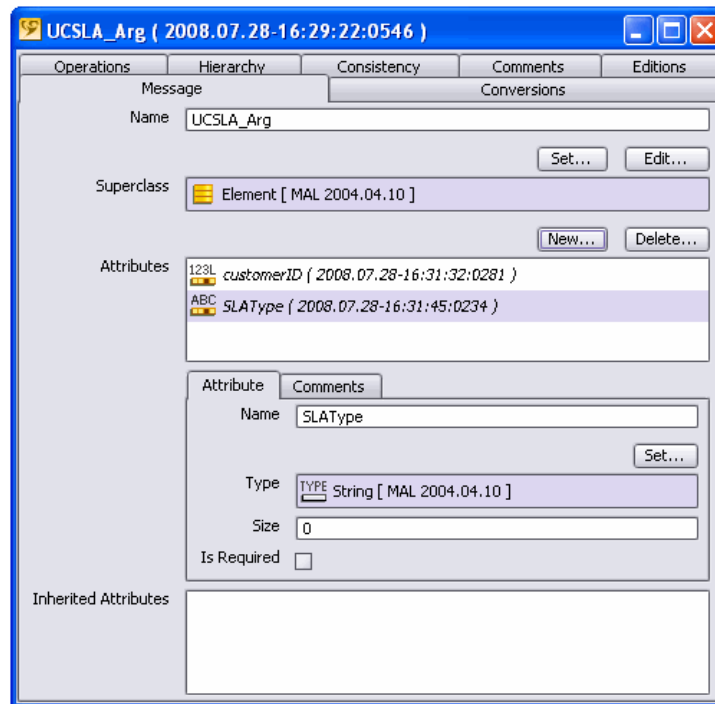
Defining the web service

The web service definition includes the argument and result messages and their attributes.

To define the web service:

1. In Cogility Modeler's tree view, select the **SimpleModel** model container, and click the **Add Inbound Web Service** button .
An untitled web service appears in the tree view and the web service editor appears in the content view.
2. In the content view, under the **IWS** tab, in the **Name** field, enter **UpdateCustomerSLA_IWS**.
3. In the content view, under the **IWS Arguments/Results** tab, in the **Argument Message** field, select the **arg** message and click **Edit**.
An editor window displays for the argument message.
4. In the **Name** field, change the name to **UCSLA_Arg**.
5. Above the **Attributes** field, click the **New** button.
 - a. In the dialog window, for **Name**, enter **customerID**.
 - b. For **Type**, select **Long** and click **OK**.
The attribute appears in the **Attributes** field.
6. Above the **Attributes** field, click **New**.

- a. In the dialog window, for **Name**, enter **SLAType**.
- b. For **Type**, select **String** and click **OK**.



7. Close the **UCSLA_Arg** message editor window.
8. In the content view, in the **Result Message** field, select the **result** message and click **Edit**.
An editor window displays for the result message.
9. In the **Name** field, change the name to **UCSLA_Result**.
10. Above the **Attributes** field, click **New**.
 - a. In the dialog window, for **Name**, enter **description**.
 - b. For **Type**, select **String** and click **OK**.
The attribute appears in the Attributes field.
11. Close the **UCSLA_Result** editor window.

Defining the deployment

An inbound web service must specify a deployment, the endpoint where external clients can access the web service.

To define the deployment:

1. In the content view, click the **Deployments for IWS** tab, and above the **Deployment For IWS** field, click the **Add** button.
2. Select **CogilityIWS** and click **OK**.

Notice that you are reusing the deployment you created in [“Inbound web services” on page 75](#).

Defining the web service logic

Finally, define the logic that associates a customer with a service level agreement.

To define the web service logic:

1. In the content view, click the **IWS** tab.
2. In the **Action** field, enter the following action semantics:

```
-- create SLA if is of valid type
cust : SimpleModel::MIM::Customer;

cust := locate
SimpleModel::MIM::Customer(customerID:input.customerID);

if cust = null then
    output.description := 'Error: Could not locate customer';
    return 0;
endif;

sla : SimpleModel::MIM::ServiceLevelAgreement;
sla := java com::cogility::WalkAssoc(cust, 'serviceLevelAgreement');

if input.SLAType = 'Gold' then
    if not sla = null then
        del sla;
    endif;
    sla := new SimpleModel::MIM::GoldSLA();
    new cust to serviceLevelAgreement(sla);
    output.description := 'Created Gold SLA for cust
'.concat(input.customerID);
    return 0;
endif;

if input.SLAType = 'Bronze' then
    if not sla = null then
        del sla;
    endif;
    sla := new SimpleModel::MIM::BronzeSLA();
    new cust to serviceLevelAgreement(sla);
    output.description := 'Created Bronze SLA for cust
'.concat(input.customerID);
    return 0;
endif;



output.description := 'Could not create SLA for cust
'.concat(input.customerID);
```

Turn over and push

Before you run the model, turn it over and push it into execution.

To turn over and push the model:

1. In the tree view, select the **SimpleModel** model container.

2. In the Cogility Modeler tool bar, click the **Turn Over** button .
3. In the Turnover Operation dialog, for **Turnover Version Name**, enter **Chapter7**.
4. In the Comments field, enter some comments and click **OK**.
A successful notification appears.
5. Click **OK**.
6. If you have already pushed a model into your run-time repository, remove the current model from the database. (Skip this step if you have not yet pushed a model.)
 - Navigate to the %DCHOME%\MB\Scripts\Dbtools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.
 A command window appears and shows the Drop Tables utility executing. Close the window when it finishes.
7. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
8. With the **SimpleModel** model container selected, click the Push button .
This pushes the enterprise application onto the application server, and the model objects and schema into the run-time repository.
9. In the Select Source of Deployment Parameters, select the object that matches your configuration and click **Use Deployment**.
When the push operation completes, a dialog appears notifying you of successful deployment.
10. Click **OK**.

Writing the message file

Because you have added a new topic since the last time you exported these message definitions, you need to export the message definitions as you did previously in [“Exporting message definitions” on page 58](#).

To recreate the message file:

1. In the Cogility Modeler’s tree view, select the **SimpleModel** model container.
2. Right-click and select **Actions > Output MessageEditor Messages**.
3. Navigate to a location for the output file and click **Output to File**.
Cogility Modeler creates the file, SimpleModel.msg in the directory you specify.

Executing the model

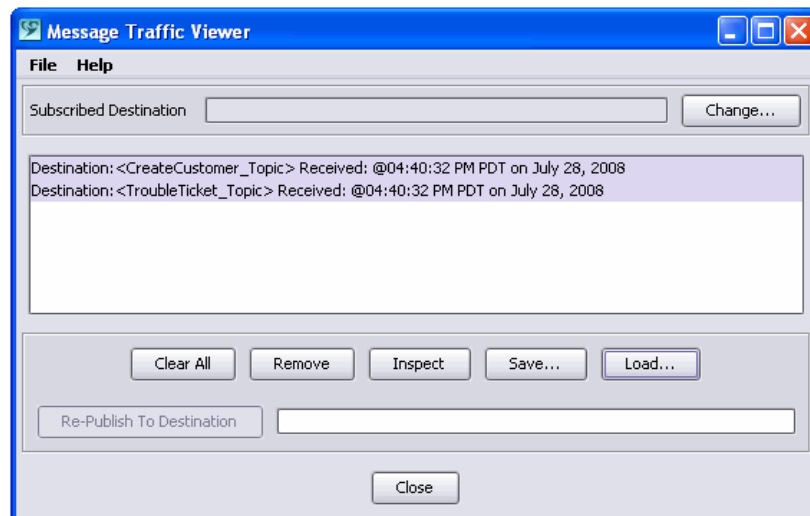
You first use Cogility Message Traffic Viewer to send a message to the model’s enterprise application to create a new customer. Then you use Cogility Insight to associate the customer with a trouble ticket.

Running Cogility Message Traffic Viewer

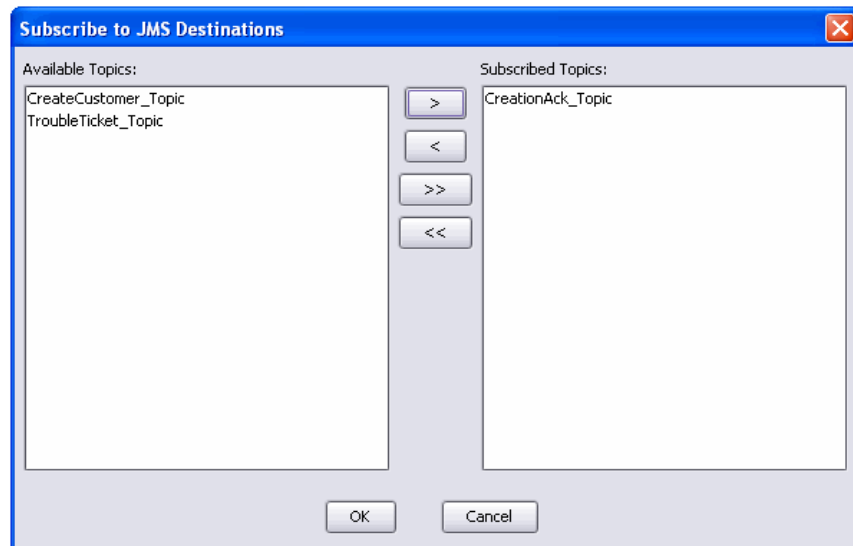
As mentioned earlier, you don’t have an actual external application to send a JMS message with new customer information. Instead, you use Cogility Message Traffic Viewer to simulate the message to your model’s enterprise application.

To run Cogility Message Traffic Viewer:

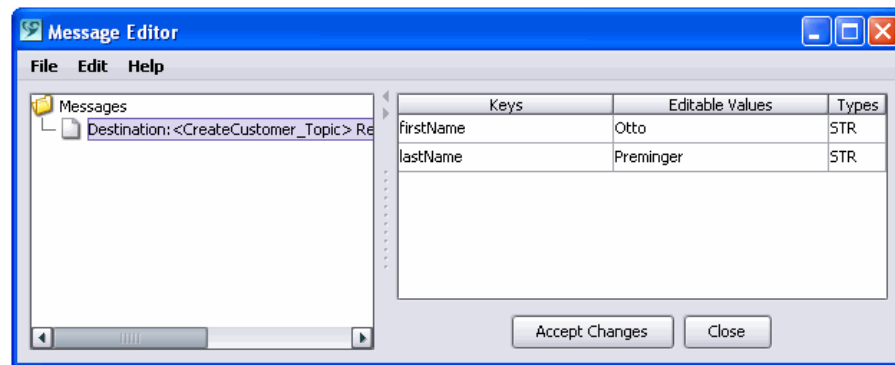
1. If your application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
A console window displays. When it shows the server is running, you may proceed with the following steps.
2. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Manager > JMS Message Viewer**.
3. In the Cogility Message Traffic Viewer window, click **Load**.
4. Navigate to the file you saved in *“Writing the message file” on page 119* and click **Open**.
Cogility Message Traffic Viewer displays the message destinations from the .msg file. These now include both the CreateCustomer_Topic destination from the previous chapters and the TroubleTicket_Topic destination from this chapter.



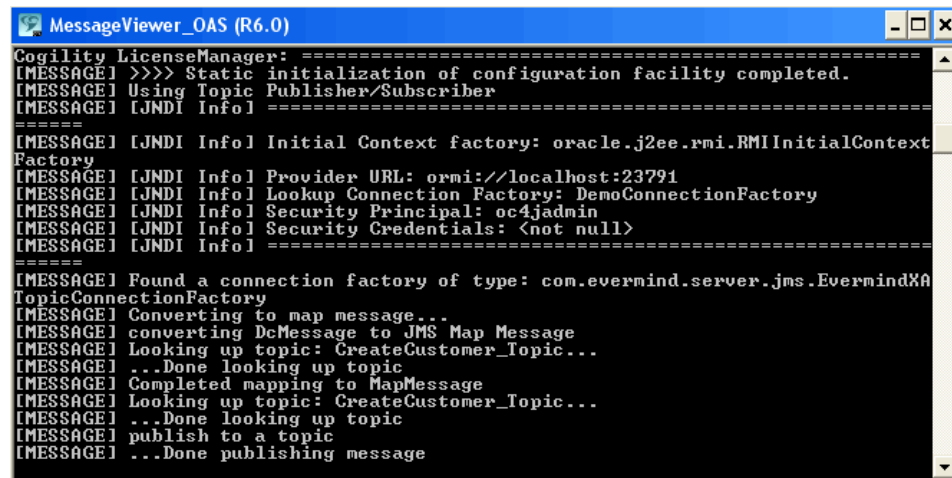
5. Next to the **Subscribed Destination** field, click **Change**.
6. In the dialog that appears, specify **CreationAck_Topic** as the **Subscribed Topic** and click **OK**.



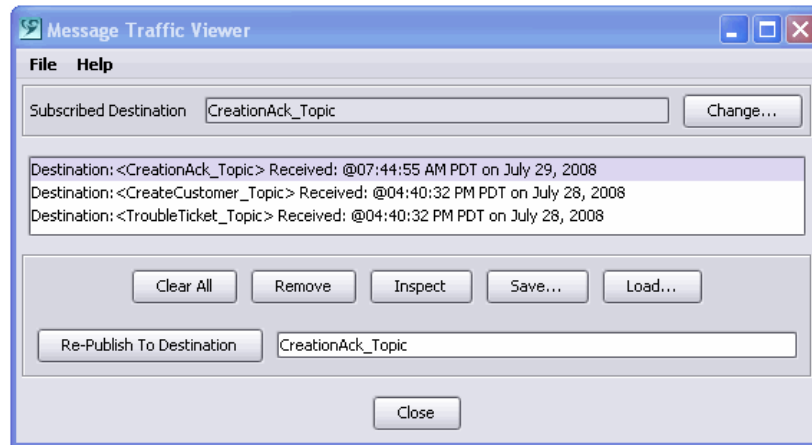
7. Select the **CreateCustomer_Topic** and click **Inspect** to open **Cogility Message Editor** and set the values for the message.
 - a. In the tree view, under **Messages**, select **Subject: <CreateCustomer_Topic>**.
Cogility Message Editor displays the attributes that the CreateCustomer_Topic destination expects.
 - b. Under **Key**, select **firstName**, and in the bottom pane, replace **TBS** with **Otto** (or some other first name)..
 - c. Under **Key**, select **lastName**, and in the bottom pane, replace **TBS** with **Preminger** (or some other last name).
 - d. Click **Accept Single Changed Value**.



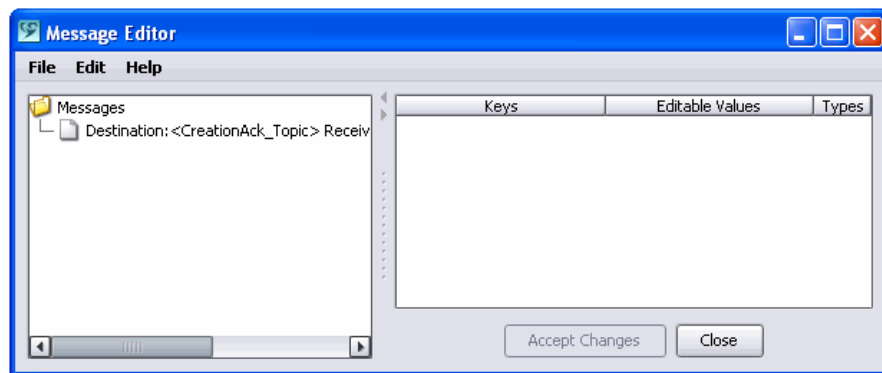
- e. Close the Cogility Message Editor window.
8. Click **Re-Publish To Destination**.
The console window for the Cogility Message Traffic Viewer describes the publication process. When complete, it looks like the following.



The **CreationAck_Topic** message is received, as shown in Cogility Message Traffic Viewer.



9. Select the **CreationAck_Topic** message, click **Inspect** and make a note of the customerID created (1681 in the example below).



10. Close the Cogility Message Editor window.

Running Cogility Insight

Cogility Insight is a web-based application that lets you monitor your enterprise application as it executes. First, you verify that the customer you created exists, then you use the inbound web service to associate the customer with a service level agreement.

Verifying the customer

To run Cogility Insight:

1. From the **Start** menu, select **All Programs > Cogility Studio > InsightWeb Access**.
2. For **UserName**, enter **admin**; for **Password**, enter **password** and click **Log In**.
3. In the **View** menu click **Business Objects**.

- From the **Business Objects** drop-down menu, select **SimpleModel.MIM.Customer** and click **Search**.

Business Object Definitions (10)

Business Object Filter
SimpleModel.

Concrete

- SimpleModel.CreateCustomer
- SimpleModel.MIM.Address
- SimpleModel.MIM.BronzeSLA
- SimpleModel.MIM.Customer**
- SimpleModel.MIM.FloorMop
- SimpleModel.MIM.GoldSLA
- SimpleModel.MIM.Order
- SimpleModel.MIM.Product
- SimpleModel.MIM.ServiceLevelAgreement
- SimpleModel.MIM.Vacuum

☐ Only those with behaviors

Search

Show	Property Name	Property Value	Type
<input checked="" type="checkbox"/>	customerID		long
<input checked="" type="checkbox"/>	emailAddress		string
<input checked="" type="checkbox"/>	firstName		string
<input checked="" type="checkbox"/>	lastName		string
<input checked="" type="checkbox"/>	poid		string

Reset **Uncheck All** **Check All**

- Select the record for the Customer you created in step 9 on page 122 (in this example, customerID 33981) by clicking on the **SimpleModel.MIM.Customer** link in the **Business Object** column.

The Business Object Details window appears.

- From the associations drop-down list, select **serviceLevelAgreement**.

Although the Customer has a valid association with a role called serviceLevelAgreement, (which you assigned to the ServiceLevelAgreement object) currently this association does not have a target. That is, there is no ServiceLevelAgreement object with a serviceLevelAgreement role for the Customer object selected.

Source: SimpleModel.MIM.Customer

Property Name	Property Value
customerID	33981
emailAddress	
firstName	Otto
lastName	Preminger
poid	BO^SimpleModel.MIM.Customer^-42f11b9e:11b6b759636:-7ff3

Associations (3):
serviceLevelAgreement

Search

Found 0 business object(s). Retrieved 0. Displaying 0 object(s), from 0 to 0. Page 0 / 0.

Making the association

In this section you run the inbound web service that associates a customer with a service level agreement.

To continue with Cogility Insight:

- In Cogility Insight, in the **Execute** menu, select **Cogility WebServices**.
- From the **Deployments** drop-down list, select **SimpleModel.CogilityIWS**.
- From the **Web Services** drop-down list, select **SimpleModel.UpdateCustomerAddress_IWS**.
- Under **Input parameters**, enter the following values:

- a. For **SLAType**, enter **Gold**.
- b. For **customerID**, enter the customerID assigned in step 9 of “Running Cogility Message Traffic Viewer” on page 119. In this example, that customerID is 1681.
- c. Click **Execute**.

The inbound web service executes and returns the result message as shown in the figure below.

Select an IWS Deployment (1):

SimpleModel.CogilityIWS

Select an IWS (2):

SimpleModel.UpdateCustomerAddress_IWS
SimpleModel.UpdateCustomerSLA_IWS

Input Parameters for SimpleModel.UpdateCustomerSLA_IWS

Argument Name	Argument Value	Type
SLAType	Gold	string
customerID	33981	long

Execute
Reset

IWS Execution Results

description: "Created Gold SLA for cust 33981"

5. Repeat the instructions under “Verifying the customer” on page 122, this time you should see that the customer is associated with the service level agreement, as in the following figure.

Source: SimpleModel.MIM.Customer

Property Name	Property Value
customerID	33981
emailAddress	
firstName	Otto
lastName	Preminger
poid	BO^SimpleModel.MIM.Customer^42f11b9e:11b6b759636:-7ff3

Associations (3):

serviceLevelAgreement

Search

Found 1 business object(s). Retrieved 1. Displaying 1 object(s), from 1 to 1. Page 1 / 1.

Last DB access: 07/29/08 08:08:25 AM PDT

Business Object	poid
SimpleModel.MIM.GoldSLA	BO^SimpleModel.MIM.GoldSLA^42f11b9e:11b6b759636:-7ff1

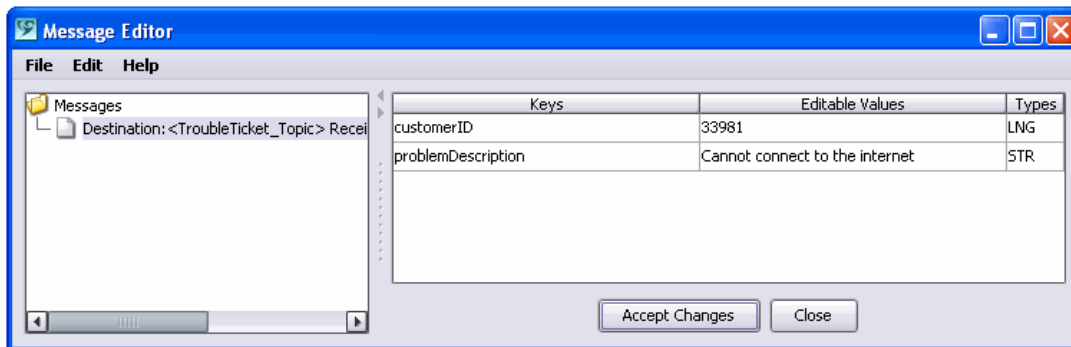
Found 1 business object(s). Retrieved 1. Displaying 1 object(s), from 1 to 1. Page 1 / 1.

Sending a trouble ticket message

Finally, you use Cogility Message Traffic Viewer to send a message with trouble ticket information to your model’s enterprise application. The message is converted to a start event for the ProcessTroubleTicket state machine, which executes and sends a message to standard output.

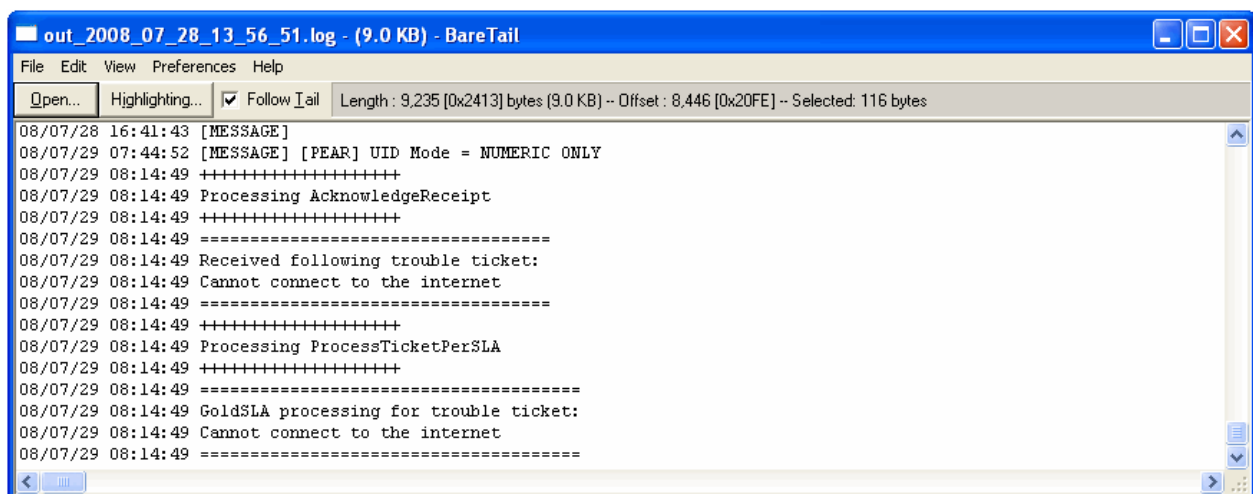
To send a trouble ticket message:

1. Return to Cogility Message Traffic Viewer.
2. Select **TroubleTicket_Topic** and click **Inspect**.
 - a. In the tree view, under **Messages**, select **Subject:<TroubleTicket_Topic>**.
 - b. In the content view, under **Keys**, select **customerID**.
 - c. In the field below, enter the customerID assigned in step 9 of “Running Cogility Message Traffic Viewer” on page 119 (1681 in this example).
 - d. In the content view, under **Keys**, select **problemDescription**.
 - e. In the field below, enter a problem description
 - f. Click **Accept Changes**.



- g. Close Cogility Message Editor.
3. In Cogility Message Traffic Viewer, with the **TroubleTicket_Topic** message selected, click **Re-publish On Subject**.

The output messages you coded into the logic for the ProcessTroubleTicket state machine are transmitted to standard output, as shown in the figure below.



These two sets of statements come from the two different states on the ProcessTroubleTicket state machine. The first state uses logic defined in ServiceLevelAgreement for processing, but the second state uses logic that over-rides the default logic and is provided by the GoldSLA class.




Within your model's logic actions, you can define custom queries to work with run-time data. Custom queries use model artifacts to represent the query and its attributes, conditions, arguments, and relationships. To use the query you specify the SQL statements within your action semantics. For more information about queries, see the guide, *Modeling with Cogility Studio*.

The exercises in this chapter further develop the SimpleModel schema created in “[Information modeling](#)” on page 9. First, you need another class for an Employee. Then you develop a query to describe product sales per Employee.

Importing the existing model

In the chapter, “[Class operations](#)” on page 103, you developed a model called SimpleModel. The exercises in this chapter work with that model. If you do not have that version of the model loaded in Cogility Modeler, you can import it with the following procedure. If you are continuing from “[Class operations](#)” on page 103, skip to “[Extending the information model](#)” on page 127.

To import the existing model:

1. If you have loaded in Cogility Modeler a model other than the SimpleModel created in “[Class operations](#)” on page 103, close Cogility Modeler and follow the steps under “[Loading the model repository](#)” on page 11.
2. If Cogility Modeler is not running, follow the steps under “[Logging in](#)” on page 12.
3. In Cogility Modeler, from the **Selection** menu, select **Import/Export > Import from Single File**.
4. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial directory, select **SimpleModel_Chapter7.cog** and click **Open**.
%DCHOME% is the directory where you installed Cogility Studio.
5. In the dialog that prompts you, click **Yes** to make the model visible in Cogility Modeler.
6. In the tree view, select the **SimpleModel** model container and click the **Place Model Under CM** button .

You are now ready to follow the procedures of this chapter.


Extending the information model

You use the class diagram editor to create the new Employee class and associate it with the existing Order class.

Adding a class

Open the existing class diagram and add the new class.


To add a new class to the class diagram:

1. In Cogility Modeler's tree view, click the plus signs (+) to expand the **SimpleModel**, **Domain Models**, **MIM**, and **Class Diagrams**.
2. Select the **CustomerModel** class diagram.
3. In the class diagram editor tool bar, click the **Create New Class** button .
4. Place the cursor below the **Order** class icon in the diagram and click again.
This locates a new class icon in the diagram.
5. Double-click the **UntitledMIMClass** icon to open its editor window.
6. In the class editor window, under the **Information Model Class** tab, in the **Name** field, enter **Employee**.
7. Above the **Attributes** field, and click **New**.
 - a. In the dialog window, for **Name**, enter **firstName**.
 - b. For **Type**, select **String** from the drop-down menu and click **OK**.
8. Click **New** again.
 - a. In the dialog window, for **Name**, enter **lastName**.
 - b. For **Type**, select **String** from the drop-down menu and click **OK**.
9. Click **New** again.
 - a. In the dialog window, for **Name**, enter **employeeID**.
 - b. For **Type**, select **Long** from the drop-down menu and click **OK**.
 - c. In the Attribute Editor tab, click the **Is Sequenceable** checkbox.
By designating this attribute as sequenceable, you are telling the database to automatically create a unique employeeID value for each instance of the class.
10. Close the class editor window.

Creating the association

The Employee class may be associated with one or more Orders. The class diagram editor allows you to describe this relationship visually.

To create an association between the classes in the diagram.

1. In the class diagram tool bar, click the **Create New Association** button .
2. Place the cursor over the **Employee** class and click.
3. Move the cursor over the **Order** class and click again.
When you release, a selection window appears.
4. In the selection window, select **Association** and click **OK**.
An association line appears between the Employee and Order classes.
5. Double-click the association line.
An association editor appears.
 - a. For **Source Role**, leave the default, **employee**.
The role describes the relationship that a class has to another. The source role owns the relationship.
 - b. For **Source Multiplicity**, enter 1.

The Order may have one and only one Employee. The source multiplicity refers to how many instances of the source role may pertain to the target role.

- c. For **Target Role**, leave the default, **order**.

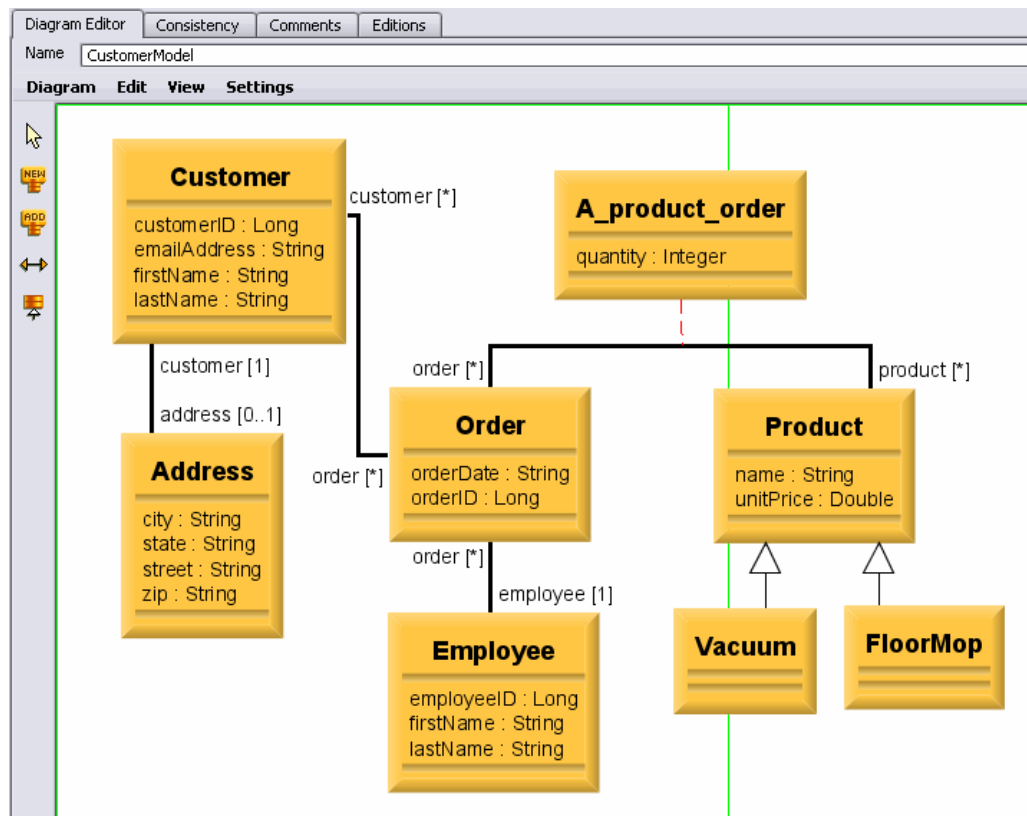
The target role describes the target class' relationship to the source class.

- d. For **Target Multiplicity**, leave the default, *****.

The Employee may have zero to many Orders. This is connoted with an asterisk * (or the notation, 0..*).

6. Close the Association editor.

The association from the Employee to the Order should appear as in the figure below.




Creating the query

In this exercise you create a custom query that reports the sales of Vacuum products for each Employee. The query reports information by state, by product name, how many items were ordered and the total amount ordered for each product name. It includes sales from the beginning of last year and excludes the sale of products whose total sales did not exceed \$100.

A query artifact in Cogility Modeler may have attributes, conditions, arguments, and relationships. You use these to define the parameters for the query.

To create a custom query:

1. In Cogility Modeler's tree view, select the **SimpleModel** container, click the **Plus** button  and select **Add Custom Query**.
2. Under the **Custom Query** tab, for **Name**, enter **SalesQuery**.

Create the query attributes, condition, arguments, relations and other clauses as described in the following sections.

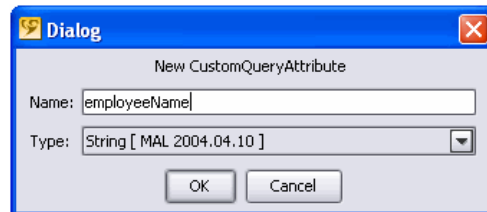
Attributes

A query's attributes describe the information it returns. 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 this query you need to establish the employee name, customer's state, product name, sales amount and number sold.

To add the query attributes:

1. In the tree view with the custom query selected, in the content view under the **Custom Query** tab, for **Attributes**, click **New**.
2. In the dialog, in the **Name** field enter a **employeeName**.
3. For **Type**, select **String** from the drop down list and click **OK**.



4. In the **Query Attribute** tab, in the **Expression** field, do the following:
 - a. Enter **#C[**, hold down the **Ctrl** key and press the spacebar.
The code assist window appears.
 - b. Select the **SimpleModel::MIM::Employee** class and press **Enter**.
 - c. Enter **].#A[**, hold down the **Ctrl** key and press the spacebar.
 - d. Select the **lastName -- java::lang::String** attribute and press **Enter**.
 - e. Finish the expression with a close bracket **]**.

You can use code assist in any query expression field. The complete expression is entered as follows:

```
#C[SimpleModel::MIM::Employee].#A[lastName]
```

The query attribute expression uses some shorthand for describing the column names in combination with ordinary SQL. The **#C** is shorthand for class, meaning the class table, and

the #A is shorthand for attribute, meaning the attribute of the table. The expression specifies that the employeeName is the lastName attribute (#A) of the Employee class (#C).

The screenshot shows the 'Custom Query' tab in the Cogility Studio interface. The 'Name' field is set to 'SalesQuery'. The 'Superclass' is 'Element [MAL 2004.04.10]'. The 'Abstract' and 'Is Distinct' checkboxes are unchecked. The 'Attributes' section shows a single attribute named 'employeeName' with a timestamp '(2008.08.01-12:40:06:0656)'. Below this, the 'Query Attribute' section is expanded, showing the 'Name' as 'employeeName', the 'Type' as 'String [MAL 2004.04.10]', and the 'Expression' as '#C[SimpleModel::MIM::Employee].#A[employeeID]'. The 'Inherited Attributes' section is empty.

For a complete description of the Cogility custom query shorthand, see the guide, *Using Actions in Cogility Studio*.

5. Repeat the previous steps to create the following attributes:

Attribute name	Type	Expression
customerState	String	#C[SimpleModel::MIM::Address].#A[state]
productName	String	#C[SimpleModel::MIM::Vacuum].#A[name]
numberSold	Integer	SUM(#AC[SimpleModel::MIM::A_product_order].#A[quantity])
saleAmount	Double	#C[SimpleModel::MIM::Vacuum].#A[unitPrice] *SUM(#AC[SimpleModel::MIM::A_product_order].#A[quantity])

Note that in the expression for the numberSold and the saleAmount, the #AC refers to the association class that describes the relationship between a Product and an Order.

Arguments

When you call the query in action semantics, you pass arguments that specify the minimum sales amount for those product sales you wish to report and the start date for the reporting period.

To create the arguments:

1. In the tree view with the **SalesQuery** selected, in the content view click the **Conditions and Arguments** tab.
2. For **Arguments**, click **New**.
 - a. In the dialog, in the **Name** field enter **minSalesAmount**.
 - b. For **Type**, from the drop down list select **Double** and click **OK**.
3. For **Arguments**, click **New** again.
 - a. In the dialog, in the **Name** field enter a **dateSince**.
 - b. For **Type**, from the drop down list select **String** and click **OK**.

The arguments appear in the content view.

Conditions

At this point your model shows an inconsistency on the SalesQuery artifact: a query must have a condition. In this query, the condition is that the sales data must be reported for a period starting on the dateSince argument specified in the action semantics that call the query.

To describe the condition:

1. In the **Conditions** field, enter the following statement:

```
#C[SimpleModel::MIM::Order].#A[orderDate] > #input[dateSince]
```

The SalesQuery editor's Conditions and Arguments tab should appear as follows:

The screenshot shows the 'SalesQuery' editor interface. At the top, there are tabs: 'Generated SQL', 'Hierarchy', 'Consistency', 'Comments', and 'Editions'. Below these, there are three sub-tabs: 'Custom Query', 'Condition & Arguments', and 'Relations & Other Clauses'. The 'Condition & Arguments' tab is active. It is divided into two main sections: 'Condition' and 'Arguments'. The 'Condition' section contains a text area with the expression: `#C[SimpleModel::MIM::Order].#A[orderDate] > #input[dateSince]`. Below this text area are three buttons: 'New...', 'Edit...', and 'Delete...'. The 'Arguments' section contains a list of arguments. The first argument is 'dateSince' with a value of '2008.08.01-12:59:38:0828'. The second argument is 'minSalesAmount' with a value of '2008.08.01-12:59:52:0359'. Below the list of arguments is a 'Query Argument' section. It has a 'Name' field containing 'dateSince' and a 'Type' dropdown menu set to 'String [MAL 2004.04.10]'. There is a 'Set...' button next to the 'Type' dropdown.

Relations

The associations between your model's classes are referenced as relations in the SQL join statement. You represent these associations with query expressions.

To represent the relations used in the query:

1. In Cogility Modeler's content view, click the **Relations & Other Clauses** tab.
2. In the **Relations** field, do the following:
 - a. Enter `#C[`, hold down the **Ctrl** key and press the spacebar.
The code assist window appears.
 - b. Select the **SimpleModel::MIM::Address** class and press **Enter**.
 - c. Enter `].#R[`, hold down the **Ctrl** key and press the spacebar.
 - d. Select the **customer -- SimpleModel::MIM::Customer** role and press **Enter**.
 - e. Finish the expression with a close bracket `]`.
3. Enter the remaining relations, each on a separate line as follows:

```
#C[SimpleModel::MIM::Customer].#R[order]
#C[SimpleModel::MIM::Order].#R[employee]
#C[SimpleModel::MIM::Order].#R[product@SimpleModel::MIM::Vacuum]
```

To refer to the Vacuum class for the product role, use code assist after the @ character.

Other clauses

The SQL join clause includes the GROUP BY, HAVING and ORDER BY statements. These further qualify the information returned in the query.

GROUP BY

The results of the query shall be grouped by the Employee's last name, the Customer's state and the name of the Product sold.

To specify the GROUP BY statement:

1. In the **Other Clauses** field, enter the following:

```
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

From the results, show only those sales having a total amount over \$100.

To specify the HAVING statement:

1. In the **Other Clauses** field, append the following in a new line:

```
HAVING max(#C[SimpleModel::MIM::Vacuum].#A[unitPrice]) *
SUM(#AC[SimpleModel::MIM::A_product_order].#A[quantity]) >
#input[minSalesAmount]
```

ORDER BY

Present the results in alphabetical order by Employee's last name.

To specify the ORDER BY statement:

1. In the **Other Clauses** field, append the following in a new line:

```
ORDER BY #C[SimpleModel::MIM::Employee].#A[lastName]
```



Your Relations & Other Clauses tab should look like the following:

Generated SQL	Hierarchy	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[lastName] 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>			

Turn over and push

With a new Employee class in the information model, you must push the model before the query can work. Also you need to push the query objects into execution.

To turn over and push the model:

1. In the tree view, select the **SimpleModel** model container.
2. In the Cogility Modeler tool bar, click the **Turn Over** button .
3. In the Turnover Operation dialog, for **Turnover Version Name**, enter **Chapter8**.
4. In the Comments field, enter some comments and click **OK**.
A successful notification appears.
5. Click **OK**.
6. If you have already pushed a model into your run-time repository, remove the current model from the database. (Skip this step if you have not yet pushed a model.)
 - Navigate to the %DCHOME%\MB\Scripts\Dbtools directory and double-click the **Run_Universal_PEAR_Table_Dropper.bat** file to drop your database tables.
A command window appears and shows the Drop Tables utility executing. Close the window when it finishes.
7. If the application server is not running, from the **Start** menu, select **All Programs > Cogility Studio > Application Server > (application server) > Start (application server)**.
8. With the **SimpleModel** model container selected, click the Push button .
This pushes the enterprise application onto the application server, and the model objects and schema into the run-time repository.
9. In the Select Source of Deployment Parameters, select the object that matches your configuration and click **Use Deployment**.
When the push operation completes, a dialog appears notifying you of successful deployment.
10. Click **OK**.

Verifying SQL

Now the run-time repository has the latest model and the query can refer to the objects pushed into execution.

To verify the generated SQL:

1. In the tree view, select the **SalesQuery**.
2. In the content view, click the **Generated SQL** tab.

Cogility Modeler generates a final SQL query from custom query objects you created.

Custom Query	Condition & Arguments	Relations & Other Clauses	Generated SQL	Hierarchy	Consistency	Comments	Editions
--------------	-----------------------	---------------------------	---------------	-----------	-------------	----------	----------

```

SELECT
    SUM(PEAR_A_product_order.A_quantity_),
    PEAR_SimpleModel_MIM_Vac_1.A_unitPrice_ * SUM(PEAR_A_product_order.A_quantity_),
    PEAR_SimpleModel_MIM_Vac_1.A_name_,
    PEAR_SimpleModel_MIM_Emp_1.A_employeeID_,
    PEAR_SimpleModel_MIM_Add_1.A_state_
FROM
    PEAR_SimpleModel_MIM_Add_1
    INNER JOIN PEAR_A_address_customer ON PEAR_SimpleModel_MIM_Add_1.PK=PEAR_A_address_customer.N_addressID_
    INNER JOIN PEAR_SimpleModel_MIM_Cus_1 ON PEAR_A_address_customer.N_customerID_=PEAR_SimpleModel_MIM_Cus_1.PK
    INNER JOIN PEAR_A_order_customer ON PEAR_SimpleModel_MIM_Cus_1.PK=PEAR_A_order_customer.N_customerID_
    INNER JOIN PEAR_SimpleModel_MIM_Order ON PEAR_A_order_customer.N_orderID_=PEAR_SimpleModel_MIM_Order.PK
    INNER JOIN PEAR_A_order_employee ON PEAR_SimpleModel_MIM_Order.PK=PEAR_A_order_employee.N_orderID_
    INNER JOIN PEAR_SimpleModel_MIM_Emp_1 ON PEAR_A_order_employee.N_employeeID_=PEAR_SimpleModel_MIM_Emp_1.PK
    INNER JOIN PEAR_A_product_order ON PEAR_SimpleModel_MIM_Order.PK=PEAR_A_product_order.N_orderID_
    INNER JOIN PEAR_SimpleModel_MIM_Vac_1 ON PEAR_A_product_order.N_productID_=PEAR_SimpleModel_MIM_Vac_1.PK
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_
  
```

Testing the query

You use Cogility Action Pad to load some sample data and test the query.

Running Cogility OCL Development Tool

You can call a query from any action semantics script whether the action semantics are part of a model or in a stand-alone text file that you load in Cogility Action Pad.

To run Cogility Action Pad and load the action semantics scripts:


1. From the **Start** menu, select **All Programs > Cogility Studio > Cogility Manager > Action Pad**.
2. From the **File** menu, select **Open**.
3. Navigate to the %DCHOME%\Examples\SimpleModel\Tutorial\Query folder.
4. Hold down the **Ctrl** key, select **DataSetUp.ocl** and **RunSalesQuery.ocl** and click **Open**.

The DataSetUp.ocl file creates the sample data on the run-time repository. The RunSalesQuery.ocl includes the action semantics that run the query.

Creating the sample data

The run-time repository needs to have some sample data against which you can run the query you just developed.

To create the sample data:

1. In Cogility Action Pad's tree view, select the **DataSetUp.ocl** script.
2. Click the Run button .

In the console view, a message appears confirming the actions were committed. The run-time repository is now populated.

Calling the query

Calling the query returns a collection of query result objects. Each has the attributes of the query populated with values based on the condition and other clauses. You use action semantics to work with this information. For more about action semantics see ["Using action semantics" on page 27](#).

To call the query:

1. In Cogility Action Pad's tree view, select the **RunSalesQuery.ocl** script.

```
RunSalesQuery.ocl
oclPrintln('Employee\t\t\t\t\tState\t\t\t\t\tProduct Name\t\t\t\t\tItems Sold\t\t\t\t\tSale Amount');
oclPrintln('=====');
query SimpleModel::SalesQuery(dateSince: '20040101', minSalesAmount: 10.00) -> iterate {
    qry : SimpleModel::SalesQuery |
        oclPrintln(qry.employeeName.
            concat('\t\t\t\t\t').concat(qry.customerState).
            concat('\t\t\t\t\t').concat(qry.productName).
            concat('\t\t\t\t\t').concat(qry.numberSold).
            concat('\t\t\t\t\t').concat(qry.saleAmount));
};
```

To call the query you need a collection to hold the query result objects; you populate the collection by calling the query, **SalesQuery** and passing the query's arguments, **dateSince** and **minSalesAmount**, as shown in line 2 above.

Line 3 reports the number of query result objects in the collection, and the following two lines print the column names.

Then you need a variable to hold the query result object, **qry** in line 6 above. Each result object is of the query's type, in this case **SalesQuery**, and has the attributes defined for the query.

Finally, you iterate through the collection of **SalesQuery** result objects and print to the console the values of the query attributes. The iteration procedure is described in ["Working with collections" on page 40](#).

2. Click the Run button .

The results are printed to the console view, as shown below.

```
Console
>>> [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 297 ms.
>>> [Commit ]: Action's results committed.
```




A

argument message (inbound web service) 76
authoring repository 11

B

behavior model 50

C

callOWS() java action 95
class hierarchy 105
class operation 103
content view 13
conversion
 E2M 68
 M2E 46
custom queries
 attributes 130

D

data path 50
deployment (inbound web service) 80

E

event 44
execution repository 11

F

factory 48
FROM SQL statement 130

I

importing a WSDL 93
inbound web service
 deployment, viewing 85
 executing 87
inbound web service deployment 80
inbound web services 116
information model 14
inheritance 103
Is Sequenceable 66

J

Java action 91

M

master information model (MIM) 14
message definitions 58
message model 44
method body 103, 110
model container 13

O

operation
 class operation 103
operations, invoking 112
outbound web service
 creating 93
outbound web services 91

P

PEAR repository 11

R

- repository
 - authoring 11
 - authoring, loading 11
 - execution 11
 - PEAR 11
 - run-time 11
- result message (inbound web service) 77
- role 21, 22, 106, 128
- run -ime repository 11

S

- SELECT SQL statement 130
- sequenceable 66
- signature 103
- start event 51, 113
- state machine 50
- states 52

T

- transitions 52
- tree view 13

V

- views 13

W

- WSDL
 - importing 93