Using model-driven development and pattern-based engineering to design SOA: Part 4. Model-to-model transformations and connecting models to EMFT JET transformations

Skill Level: Intermediate

Lee Ackerman (ackerman@ca.ibm.com)
Sr. Product Manager
IBM

Bertrand Portier (bportier@ca.ibm.com)
IT Architect
IBM

Chris Gerken (cgerken@us.ibm.com)
MDD and Patterns Mentor
IBM

17 Jun 2008

This tutorial explains how you can extend IBM® Rational® Software Architect to create model-to-model transformations. You will create your own plug-in with transformation mapping, define the mappings, build and then test the resulting plug-in. You can use this type of transformation both to move between levels of abstraction represented through UML models and, by using EMFT JET transformations, to move from a UML model into text-based artifacts.

Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this series

To reap the benefits of model-driven development (MDD), your design and development environment needs to have the following characteristics:
• **Best practices for reuse**: People can reuse proven solutions to recurring problems, as well as provide solutions for others to reuse.

• **Role-based tools**: Tools are targeted to the task at hand and to the role of the person performing that task (for instance, Business Analyst or IT Architect).

• **Process support and guidance**: There is always method or process guidance in context.

• **An extensible platform**: Teams can extend or customize the environment to fit their needs.

• **Automation**: The framework’s underlying meta-model and mappings allow for the semi-automatic transformation of models, from higher to lower levels of abstractions, and eventually to executable code. It is also possible to trace back from lower to higher levels of abstractions.

These are all characteristics of the IBM® Rational® Software Delivery Platform and, more specifically, of IBM® Rational® Software Architect. In this series of four tutorials, you learn how to extend the platform and its capabilities to help you as you create SOA-based solutions. We also explain what modeling is and how to leverage the extensible features of Rational Software Architect.

• Part 1 relates service-oriented architecture (SOA) to model-driven development.

• Part 2 walks you through creating your own Unified Modeling Language (UML) profiles and model templates.

• Part 3 builds on Part 2 by diving into pattern-based engineering (PBE) and the creation of patterns and transformations.

• Part 4 (this tutorial) describes packaging your software assets for reuse.

After working through this series, you should be able to describe the features at your disposal to extend Rational Software Architect in the design of SOA. You will know what modeling is and how to create UML profiles, model templates, UML patterns, transformations, and reusable assets.

**About this tutorial**

In this tutorial, Part 4 of the series, we discuss how you can use your own custom transformations in Rational Software Architect to automate the design of an SOA solution. A key aspect of transformations is that they enable you to cross levels of abstraction. In addition, you can use these automations to improve the quality of the solution and to support the overall governance process.

In this final part of the series, you will create your own model-to-model
transformation. This transformation takes a Unified Modeling Language (UML) model that uses the UML profile for Software Services (a service model) and transforms it into an EMF (Eclipse Modeling Framework) model that represents the service model report.

In Part 3, we talked about model-to-text transformations and Eclipse Modeling Framework Technology Java Emitter Template (EMFT JET). In this part, we will connect the output of the model-to-model transformation to the EMFT JET project from Part 3. In fact, the EMF model that is the output of the model-model transformation is same as the EMF model that serves as the input to the EMFT JET transformation from Part 3.

Objectives

After completing this tutorial, you will have a better understanding of how you can take advantage of tools and features within Rational Software Architect to build your own transformations. These automations encode your best practices, which are often specific to your organization and are part of your organization’s competitive advantage.

More specifically, you will have learned how to create, define, build, and test your own model-to-model transformation by using IBM Rational Software Modeler or Rational Software Architect Version 7. You will see these tools and artifacts in use in this tutorial:

- EMF projects
- Rational Software Architect or Rational Software Modeler plug-in projects with transformation mappings
- UML models and profiles
- Java code snippets that use the EMF, UML, and extensibility Application Programming Interfaces (APIs)

Prerequisites

You will get the most benefit from this tutorial if you are familiar with these tools:

- Eclipse, the open source integrated development environment (IDE) or
- Rational Software Architect, Rational Systems Developer, or Rational Software Modeler or
- Java™ Standard Edition

Familiarity with the following is also helpful, but not necessary:
• UML, the Unified Modeling Language
• SOA, Service-Oriented Architecture
• EMF, the Eclipse Modeling Framework
• Eclipse plug-in development
• The EMF, UML, and extensibility APIs

See Resources for useful links to more information on these topics.

System requirements

To complete this tutorial, you need to have the following installed (see Resources for links to trial versions):

• Rational Software Architect Version 7.0
  or
• Rational Software Modeler Version 7.0
  or
• Rational Systems Developer Version 7.0

To view the demos included in this tutorial, JavaScript must be enabled in your browser and Macromedia Flash Player 6 or higher must be installed. You can download the latest Flash player at http://www.macromedia.com/go/getflashplayer/.

Section 2. Context of this tutorial

In the "Design SOA Services with Rational Software Architect" series (listed under Resources), we have described how you can use models to capture the design of your SOA-based solution. We’ve been able to work at a number of different levels of abstraction and use a number of model elements along the way. These model elements have helped to guide us in our design and enabled us to generate detailed artifacts as needed. In this series, we are looking at the underlying support that is available for the design of SOA (and other) solutions.

• In Part 1, we looked at two types of artifacts supported within Rational Software Architect, namely UML profiles and model templates.

• In Part 2, we looked at pattern-based engineering and how to create pattern implementations, specifically UML patterns, in Rational Software
In Part 3, we introduced transformations, their benefits, and the transformations provided by Rational Software Architect. We explained why you would create your own transformations and what types of transformations you could create. We then described in detail how to create your own model-to-text transformation by using the EMFT JET framework in Rational Software Architect. As you may recall, this tooling is especially suited to generating text-based artifacts. It is quick and easy to learn and use. However, we ended that tutorial where we were using XML-based (EMF) documents to represent the input model. In some situations, using this simple XML-based input will suffice. In many others, though, you will need to create a more user-friendly experience as a frontend to the transformation.

In this tutorial we are going to look at another kind of transformation in detail: a model-to-model transformation. You are going to create your own plug-in with transformation mapping, define the mappings, build and then test the resulting plug-in. You can use this type of transformation both to move between levels of abstraction represented through UML models and, by using EMFT JET transformations, to move from a UML model into text-based artifacts.

Section 3. Scenario description

When using UML models, we often expect to be able to leverage the model to achieve four goals:

- To communicate the design or solution to others by using a standard notation
- To use the tooling and mechanisms to help in thinking though the design before actual implementation
- To use the model to generate other necessary artifacts (for example, detailed models, code, test scripts, deployment scripts, and documentation)
- To provide a user-friendly frontend that is used in conjunction with transformations built through EMFT JET

For this tutorial, assume that you are already using models for those first two items. As such, we will focus here on the third and fourth aspects. You want to use the investment that has been made in designing a solution and get a further return on that investment. If you look at this from a top-down perspective, you want to use any UML models that you create to generate supporting artifacts. If you look at this from
a bottom-up perspective, you want to make any EMFT JET transformations that you build as easy to use as possible.

When creating SOA-based solutions, developers often intend to create a model that represents the services that will either be part of or be used by the solution. The implementation of the services may or may not be performed by the same group of people. In addition, after the services have been implemented, you need to document the interfaces that are provided by the services. This documentation may be used by others within your organization or by employees of partner organizations.

To build an automated solution for generating this documentation, you are going to look at building a two-part transformation. The part covered here will be used to gather information from a UML model and generate a simplified input model that will then be used as input for a model-to-text transformation (covered in Part 3) that will then generate a set of HTML pages. Although this is a simple and somewhat contrived example, this scenario enables you to explore both model-to-model and model-to-text (Part 3) transformations.

Tip:
In general, to generate reports from your Rational Software Architect models, we recommend that you use the included functionality. We use a model reporting example here merely as an educational model-to-text transformation case study, not as the recommended way to generate HTML reports from Rational Software Architect UML models.

In Part 3, you specified an Eclipse Modeling Framework Technology Java Emitter Templates (EMFT JET) transformation. By using the exemplar authoring technique, you started from a typical HTML report. Based on that, you specified the schema and JET templates for generating similar reports from your (EMF) service models.

The EMFT JET transformation from Part 3 takes an EMF model as input (as EMFT JET transformations do). This is a good start, but now, let's assume that you want to use a UML model to serve as a more user-friendly frontend for report generation. To do this, in this part of the tutorial, you are going to create a model-to-model transformation that generates an EMF model based on a UML (service) model. The output of this transformation is the input of the EMFT JET transformation from Part 3. You are then going to connect the two transformations so that you can generate text (HTML) from the model (UML). This process is illustrated in Figure 1.

**Figure 1. Diagram of how a model-to-model transformation generates an EMF model based on a UML (service) model**
Section 4. Summary of steps

In this tutorial, you will create the artifacts for the model-to-model transformation, connect that transformation to the EMFT JET transformation from Part 3, and test the overall transformation. You will perform these tasks:

1. Prepare the workspace by importing the required input model project from Part 3 and configuring the workbench capabilities.
2. Create a new EMF project and generate the model code from the schema (input.ecore) from Part 3.
3. Create a new plug-in project with transformation mappings.
4. Specify the five mappings required to transform the input UML model into the output EMF Ecore model.
5. Generate the transformation code.
6. Connect the new transformation to the EMFT JET project.
7. Test the overall transformation.

To perform these tasks, you will use the following three projects:

- **com.ibm.serviceSpec.report.xform**: The EMFT JET project with exemplar authoring (from Part 3)

- **com.ibm.serviceSpec.report.xform.model**: The EMF project that holds the Ecore model and the model code, which you create in this tutorial
(Part 4). We use the term *model* here in its more generic sense. As such, it is represented in EMF rather than in UML.

- **com.ibm.serviceSpec.report.xform.frontend**: The plug-in project with transformation mappings, which you will create now, in Part 4.

This is illustrated in Figure 2.

**Figure 2. Diagram of the process with the 3 projects**

The EMF project in this case provides a way to represent and manipulate the information that needs to be passed from the UML representation into the EMFT JET-based transformation. Recall that when you built the EMFT JET project, you defined the information that the user of the transformation was required to input. This input model then serves as the target output model that you use for the UML-based transformation.

---

**Section 5. Create the initial set of projects**

In this section, you will walk through the steps necessary to set up the transformation mapping, EMF, and EMFT JET projects needed for the solution. To get started, you'll need to launch Rational Software Architect, create a new workspace, and import the project from Part 3:

1. Launch Rational Software Architect.
2. In the Workspace Launcher dialog, pick a location for the workspace, specify a name of MDD-PBE-Part4-workspace, and click OK.
3. Close the Welcome screen.

You will now import a project that includes the EMFT JET solution (including the associated EMF model) from Part 3 of the series. You'll use this as the starting point
for the work that you do in this part of the series:

4. Select **File > Import**.

5. In the Import wizard (Figure 3), type project in the **Select an import source filter** field, and then select **Project Interchange** and click **Next**.

![Figure 3. Import wizard](image)

6. Beside the "From zip file" field, click the **Browse** button and then select this file: `com.ibm.serviceSpec.report.xform.pi.zip`.

7. Select the `com.ibm.serviceSpec.report.xform` project and click **Finish** (see Figure 4).

![Figure 4. Import Projects view](image)
You have imported the EMFT JET project from Part 3 into a new workspace. You will now set the Rational Software Architect capabilities so that you can perform the tasks in this part of the tutorial:

8. Select **Window > Open Perspective > Other** and then **Plug-in Development** to switch to the Plug-in Development perspective.

9. To make sure that the **XML Developer and EMF Developer** capabilities are fully enabled, select **Window > Preferences** (Figure 5).

10. Under **General > Capabilities**, check **XML Developer** and **Eclipse Developer**.

11. Click **Apply** and then **OK**.

**Figure 5. Preferences view**
Important

If your Eclipse Developer capabilities are partially enabled (green square and not green checkmark), then you need to click the square next to Eclipse Developer twice until it shows a checkmark. This is very important for the instructions in this part to work.

Create a new EMF project
In this task, you will create a new EMF project to hold the EMF representation of the input to the JET transform and its associated code (which you will generate). The Ecore model that you will use as input is the one that was generated in the EMFT JET project in Part 3.

1. On the File menu, select New > Project.

2. Replace the Wizards type filter text with emf (Figure 6), select EMF Project, and then click Next.

Figure 6. New Project view
3. Enter the project name, `com.ibm.serviceSpec.report.xform.model`, and click **Next**.

4. Select **Ecore model** for Model Importers, and click **Next**.

5. In the `com.ibm.serviceSpec.report.xform` project, select **Browse Workspace** to find the file named `input.ecore` and select it.

6. Click **OK** and then click **Next**.

The `input.ecore` file was created as part of the EMFT JET project in Part 3. More specifically, it was created and specified as you defined the input schema as part of exemplar authoring in the EMFT JET project. It is the input for the EMFT JET transformation. In this tutorial, you will use the schema defined by the EMF model as the output of the transformation. (See Figure 7.)

**Figure 7. New EMF Project view**

7. Leave the defaults for the Package Selection and click **Finish**.

8. The file `input.genmodel` will display in the editor (Figure 8). Right-click the **Input** node and select **Generate Model Code**.

**Figure 8. Editor view**
The EMF genmodel is the file associated with the Ecore file, and that allows you to configure the EMF code generation.

9. Close the input.genmodel.

From the `com.ibm.serviceSpec.report.xform.model` project, you can open `input.ecore` (in the EMF ecore editor, shown in Figure 9) from the model folder. Then, you should see the definition of the same schema as the one that you specified in Part 3 for the EMFT JET project (ServiceGroup, Operation, Argument, and so on). Remember, this Ecore was created as part of the EMFT JET project.

Figure 9. Ecore editor

11. From the **Package Explorer** view, observe the generated EMF Model code under the **src** directory (Figure 10) of the com.ibm.serviceSpec.report.xform.model project.

**Figure 10. Package Explorer view**
The `inputSchema` package contains the Java interfaces for the types defined in the schema (ServiceGroup, Service, and so forth). All of these interfaces extend the `EObject` EMF interface.

The `inputSchema.impl` package contains the Java classes that implement the interfaces in the `inputSchema` package. These classes extend the `EObjectImpl` EMF class.

The `inputSchema.util` package contains the EMF utility classes for this model.

For more information about EMF, see the Eclipse site (listed under Resources).

Create a new plug-in project with transformation mapping

In this task, you will create a new Plug-in Transformation project named `com.ibm.serviceSpec.report.xform.frontend` to define the mapping from the UML service model to the EMF Ecore model that is used by the JET
transformation (Part 3). As the name implies, this will be the frontend that the end users of the transformations will use. In this way, they will be able to model in UML and then still use the EMFT JET transformation to generate the service specification report.

1. From the File menu, click **New > Project**.
2. Replace the type filter text with **Plug**
3. Select **Plug-in Project** and click **Next** (Figure 11).

**Figure 11. New Project view**

![New Project view](image)

4. Name the project `com.ibm.serviceSpec.report.xform.frontend`, and then click **Next**.
5. Review the Plug-in Content screen, leave all of the defaults, and click
6. On the Templates screen (Figure 12), make sure that "Create a plug-in using one of the templates" is selected, and select **Plug-in with Transformation Mapping**.

7. Click **Next**.

**Figure 12. Templates view in New Plug-in Project**

At this point, you may be wondering why it is called a Transformation Mapping project. The reason behind this naming is that you are creating a mapping between a set of elements captured in one model and detailing how they convert to elements specified by another model. The **source** model is known as the **input model**, and the **target** model is known as the **output model**. You can use UML profiles and EMF.
models as the basis for the models that work with these mapping projects.

8. On the New Transformation Mapping screen (Figure 13), specify a name of ServiceSpecificationFrontend.

9. Next to Input models, click the Add Model button.

10. In the Load Resources dialog, click Browse Registered Packages.

11. Replace the * (asterisk) with *UML.

12. Select the package http://www.eclipse.org/uml2/2.0.0/UML, and then click OK twice.

This selects the UML Ecore model for the Input model. This Ecore model defines the elements that are available to users as they create and work with UML models.

**Figure 13. New Transformation Mapping view**

![New Transformation Mapping view](image)

**Tip:**
The mapping model uses Ecore models as the common model format for mapping.

13. Next to Output models, select Add Model.
14. Click **Browse Workspace** and then select the `input.ecore` file from the `com.ibm.serviceSpec.report.xform.model` project from within the `model` folder (Figure 14).

15. Click **OK** twice.

**Figure 14. File Selection view**

![File Selection view](image)

The New Transformation Mapping wizard page should now look like Figure 15.

**Figure 15. Updated wizard parameters**
Note: You can have multiple models specified, both within the input and output models field. In this case, for instance, you need to recognize elements from both UML2 as well as from the Software Services profile (which extends the definition of UML to be specific to SOA).

Next, you need to add the Software Services profile that will be used in the input models.

16. Click **Add Model** beside the **Input models** field.

17. Click **Browse Registered Profiles** and select **Software Services**.

18. Click **OK** twice.

The New Transformation Mapping wizard page should now look like Figure 16.

**Figure 16. Updated parameters**
19. Click Finish. If you are asked to switch to the Plug-in Development perspective, select Yes.

---

Section 6. Define transformation mappings

In the following sections, you will create the five mappings that will comprise the model-to-model transformation. These mappings allow for the transformation of the UML Service model input to an EMF Ecore model that is needed to generate the service report output:

1. **ModelToRoot** is the mapping between the UML service model and the EMF Ecore service report model.

2. **ServiceGroupPackageToServiceGroup** is the mapping between package elements in the UML service model and elements of the EMF Ecore service report model.

3. **ServiceSpecificationToService** is the mapping between the service
specification in the UML service model and the service element in the
EMF Ecore model.

4. **OwnedOperationToOperation** is the mapping between an operation in
the source interface (serviceSpecification) and the operation
element in the EMF Ecore model.

5. **OwnedParameterToArgument** is the mapping between the operation
parameter (arguments) in the input UML service model and the output
EMF Ecore service report model.

Notice that this is a hierarchical structure, where mapping 1 uses mapping 2,
mapping 2 uses mapping 3, mapping 3 uses mapping 4; and mapping 4 uses
mapping 5. We have established a set of rules to navigate the model, with
instructions on what elements participate in the transformation, filtering out elements
that are not needed, and where additional details and information needs to be
calculated or added to the output model. These rules start at the most general, outer
elements in the model, then work inward, getting more specific. For example, we
consider the entire model to be the most general, thus the outermost element in the
model. From there, we navigate into the model, working through packages to
contained elements, such as classes, to the elements within classes. Along the way,
we either perform calculations, delegate to a more specific level of mapping, or both.

Specify the model-to-root mapping

In this task, you will create the first mapping to use in the transformation.

1. Select the **ServiceSpecificationFrontend.mapping** that is created and
   opened in the mapping editor. The project manifest for the project might
currently be the file that is selected in the editor. If it is, click the tab
   labeled **ServiceSpecificationFrontend.mapping**.

Figure 17. Mapping editor

2. In the editor, right-click the **ServiceSpecificationFrontend** button and
   select **Create Map**. Name the map **ModelToRoot**.
The mapping editor toolbar shows your new map (Figure 18).

**Figure 18. Mapping editor toolbar**

3. Click the left-most icon in the toolbar (Figure 19) to add an input object. The source of this input object is the list of elements within the input models that you specified earlier. Therefore, the elements you have available are from the UML2 metamodel, as well as from the Software Services profile.

**Figure 19. Icon for adding an input object**

4. When the Add Input screen appears, simply start typing the letters `mod`, and the UML model will be highlighted.

5. In this case, working with UML models is too generic, so you need to be more specific and work with service models. Select the ServiceModel element from the Stereotype pane (Figure 20) and then click OK.

**Figure 20. Add Input view**
6. Click the second icon from the left in the toolbar (Figure 21) to add an output object.

Much like when you worked with the input object, the source of the output object is the output model that you specified earlier. In this case, you need to select an element that was defined within the input.ecore model (originally generated in the EMFT JET project).

**Figure 21. Icon to add an output object**
Note:
If you forgot to add a model when you created a project, notice that there is an option to add a model in both the Add Input and Add Output screens while you are specifying the mappings.

7. Select Root and click OK (see Figure 22).

Figure 22. Add Output view

Now you are ready to define the transformation between input (the UML model that uses the Software Services profile) and output (the Ecore model that defines the service report). You need to map the `packagedElement` from the UML model to the `serviceGroup` element in the Ecore model.

Note:
If you want to know more about what each element in the UML metamodel is, what is contained within the elements, and so forth, refer to the UML2 project from Eclipse (listed in Resources).
8. Hover the cursor over the `packagedElement` property of the input model until a handle appears. Select this handle and drag it onto the `serviceGroup` element of the target, `Root`.

*Figure 23. serviceGroup element under Root target*

**Tip:**
In mapping terms, submap means that a map is using another map as part of its definition. In this case, `ModelToRoot` will use this new submap. *Remember:* Map 1 uses Map 2 in the map hierarchy. Here, Rational Software Architect created a submap by default, because you are mapping two different, complex types.

9. Select the `Submap` in the mapping editor. Right-click and then select *Show in Properties*.

10. In the Properties view, click the *Details* tab.

11. Click the *New* button beside the Map field.

12. Name the new map `ServiceGroupPackageToServiceGroup` and then click *OK*.

*Note:*
You will work on this new mapping in a later step.

Next, you need to do further work on the `ModelToRoot` mapping to make it more
specific in its processing. Your Outline view should now look like Figure 24.

**Figure 24. Outline view**

13. Select **File > Save All**.

At this point, the mapping will take all of the packages and create a `serviceGroup` for each one. However, that is not the behavior that you are seeking. Instead, you need only packages with a keyword of `<serviceGroup>` to be used to create the `serviceGroup`. Therefore, you need to create a custom extractor to filter out the packages that are not of interest.

**Tip:**
A keyword is like an annotation on a UML model element, which specifies that the element is more than just its base UML type. For example, here, we wanted to label certain packages as packages that contain services, and we used the `<<serviceGroup>>` keyword for that. Unlike stereotypes, keywords are not defined in profiles.

**Specify the ServiceGroup custom extractor**

In this task, you will enhance the mapping with a custom extractor to constrain the elements that are transformed. A custom extractor allows you to use code to selectively choose which of the matching elements from the input model will be operated on as you move to the output model. For instance, as you see in our example, we don't want all packages, just those with the appropriate keyword.

For this tutorial, we wrote the custom extractor Java source code for you.

1. In the Package Explorer, under the `com.ibm.serviceSpec.report.xform.frontend` project, select
2. From the file system, import the ServiceGroupExtractor.java file (needed now) and ServiceSpecificationExtractor.java file (needed later).

Listing 1 shows the source Java code for the ServiceGroupExtractor class. The code in the addContainedSpecPckgs method recursively goes through UML packages and adds them to the collection of specification packages, but only if they have the serviceGroup keyword.

You need this custom extractor because only the serviceGroup packages are relevant here, not all packages.

Listing 1. Java code for the ServiceGroupExtractor class

```java
public class ServiceGroupExtractor implements ExtractorExtension {

    public Collection execute(EObject umlPkg) {
        List containedSpecPkgs = new ArrayList();
        addContainedSpecPkgs((Package) umlPkg, containedSpecPkgs);
        return containedSpecPkgs;
    }

    private void addContainedSpecPkgs(Package umlPkg, List containedSpecPkgs) {
        if (umlPkg.hasKeyword("serviceGroup")) {
            containedSpecPkgs.add(umlPkg);
            // add in everything that this package contains
            containedSpecPkgs.addAll(umlPkg.getOwnedTypes());
        }

        List containedPkgs = umlPkg.getNestedPackages();
        for (Iterator i = containedPkgs.iterator(); i.hasNext();)
        {
            Package nestedPkg = (Package) i.next();
            addContainedSpecPkgs(nestedPkg, containedSpecPkgs);
        }
    }
}
```

3. Open ModelToRoot mapping and select the Submap from packagedElement to serviceGroup:

   A. In the Properties view, on the Custom Extractor tab, select the checkbox for Custom Extractor.

   B. Select External for the Code option.

   C. Select Browse, and start entering the text for the ServiceGroupExtractor until you can select the class that you just imported.

   D. Click OK.

Figure 25. Custom Extractor tab in the Properties view
4. Select **File > Save All** to save your work.

Now, when the transformation runs, it will examine the packages and select only those that match the rules that are specified in the custom extractor.

### Specify the ServiceGroupPackage to ServiceGroup mapping

In the previous task, you created a Submap mapping between `packagedElement` and `serviceGroup`. We created a new mapping for this Submap named `ServiceGroupPackageToServiceGroup`. In this task, you will edit this mapping.

1. In the Outline view, double-click the `ServiceGroupPackageToServiceGroup` map to open it in the mapping editor.

2. Create a mapping between the `name` attribute in `PackageableElement` to the `name` attribute in `ServiceGroup`.

**Tip:**
In mapping terms, a *Move* is the simplest transformation, where the value of the output attribute is set to the value of the input attribute. Rational Software Architect created a Move because you are mapping two elements of the same type (string) and with the same cardinality.

By default (Basic Feature Filter), not all features of an input or output element appear in its table. The features are filtered to help simplify the interface and navigation through the fields in the element. If you cannot find a feature, simply change the filtering so that more features are shown. Here, you need to map from an intermediate attribute on the input element; therefore, you need to change the feature filter settings.

3. Right-click `PackageableElement` and select **Feature Filters > Intermediate** (Figure 26).
4. Create a mapping between the `ownedElement` attribute of `PackageableElement` to the `service` attribute of `ServiceGroup`. This will be a Submap, because there is not a 1-to-1 mapping between the attributes.

5. Select the **Submap** and then, in the Properties view, click the **Details** tab.

6. Click **New** beside the Map field.

7. Specify `ServiceSpecificationToService` as the name and then click **OK**.

8. Select **File > Save All**.

**Specify the ServiceSpecification custom extractor**

In this task, you will enhance the mapping with a custom extractor to constrain the elements that are transformed.

1. Open the `ServiceGroupPackageToServiceGroup` mapping and select the **Submap** from `ownedElement` to `service`.

   A. In the Properties view, under the **Custom Extractor** tab, select the checkbox for **Custom Extractor**.
B. Select **External** for the Code option, because you are going to get the extractor from a class rather than define it inline.

C. Select **Browse**, and start entering the text for `ServiceSpecificationExtractor` until you can select the class that you just imported.

D. Click **OK**.

2. Select **File > Save All** to save your work.

**Specify the ServiceSpecificationClass to Service mapping**

In the `ServiceGroupPackageToServiceGroup` mapping, you created a submap between `ownedElement` and `service`. In this task, you will specify the mapping for this submap -- a mapping that associates the `ServiceSpecification` with the `Service` element in the output model. This will enable you to pass the name of the service to the output model and documentation about the service and then to prepare for another, more detailed pass where you start to look at the operations associated with each service.

1. In the Outline view, double-click the **ServiceSpecificationToService** mapping to open it in the mapping editor (Figure 27).

**Figure 27. Mapping editor**

![Mapping editor](image)

**Tip:**
Notice that the input and output elements were selected for you when the mapping was created.

2. For this mapping, you need the input model element to be more specific than `Element`, which was automatically selected when the mapping was created. To remove this input object, select the input element, `Element`, right-click and select **Delete**.
3. Now you need to add a new, and more specific input object:
   A. Click the **Add an Input Object** icon.
   B. Select **Interface** from the Element pane and then **ServiceSpecification** from the Stereotype pane.
   C. Click **OK**.

4. Set the Basic Feature Filter setting for this input Interface.

5. Create a transformation between the name of the input called **Interface** and the name of the output called **Service**. This will result in a mapping of type Move being created. In this case, you are just copying the name value from the service specification to the service in your output model.

6. Create a transformation between the **ownedComment** of the input **Interface** and the documentation of the output **Service**. The result will be a new mapping of type Custom being created.

   **Tip:**
   In mapping terms, Custom means that the value of the output attribute is calculated using the custom code that you provide. Software architect created a Custom mapping because you are mapping a complex type (array of UML comments) to a simple type (EString).

7. Create a transformation between the **ownedOperation** attribute of the input **Interface** and the operation of the output **Service**. The result will be a transformation of type submap (Figure 28.)

**Figure 28. Submap transformation**
8. Select **File > Save All.**

At this point, you have Custom and Submap mappings that you need to further refine.

9. Select the **Custom mapping** from `ownedComment` to `documentation`.

10. In the Properties view (Figure 29), select the **Detail** tab.

11. Ensure that Code is set to **In-line**, add the code shown in Listing 2, and click **Apply**.

**Listing 2. Code to add**

```java
if (Interface_src.getOwnedComments().size() > 0) {
```
Service_tgt.setDocumentation(((Comment) Interface_src.getOwnedComments().get(0)).getBody());
} else {
    Service_tgt.setDocumentation("");
}

Figure 29. Properties view of the custom transformation

The code for this extractor simply retrieves the values of the Comments for the source Interface, and sets the value of the target service element to that value.

12. Select File > Save All.
13. Select the Submap from the ownedOperation to operation.
14. In the Properties view, select the Details tab.
15. Click the New button beside the Map field.
16. Specify OwnedOperationToOperation as the Map name.
17. Click OK, and select File > Save All.

Specify the OwnedOperation to Operation mapping

In the previous task, you created a mapping between interfaces, specifically serviceSpecifications, in the input and output models. That mapping included a submap between their operations. Now you need to pass along information related to the operations provided on the service and ensure that information gets passed along to the output model. As such, you will now specify that OwnedOperationToOperation mapping.

1. In the Outline view, double-click the OwnedOperationToOperation mapping (Figure 30) to open it in the mapping editor.
2. Right-click the input element named Operation and select Feature Filters > Intermediate.
3. Create these four mappings between input and output attributes:
   • From **name** to **name**, which provides the name of the operation to the output model
   • From **ownedComment** to **documentation**, which provides any documentation associated with the operation
   • From **type** to **returns**, which indicates what the return type is for the operation
   • From **ownedParameter** to **argument**, which provides the parameters that serve as arguments for the operation

**Figure 30. Mapping editor**

The mapping from **Operation:name** to **Operation:name** is a **Move**, because they match in type and multiplicity. There is nothing more that you need to do for this mapping.

The mapping from **Operation:ownedComment** to **Operation:documentation** is a Custom mapping, and a mapping that you have worked through previously.

4. Select this mapping and then, in the Properties view, select the **Details** tab.

5. Ensure that Code is set to **In-line**, add the code in Listing 3, and click
Apply.

Listing 3. Code to add

```java
if (Operation_src.getOwnedComments().size() > 0) {
    Operation_tgt.setDocumentation(((Comment) Operation_src
        .getOwnedComments().get(0)).getBody());
} else {
    Operation_tgt.setDocumentation("");
}
```

6. **Select File > Save All.**

The mapping from type to returns is a Custom mapping. As such, you need to add code to the mapping to transfer information from the input element to the output element.

7. **Select this mapping.**

8. **In the Properties view, select the Details tab.**

9. **Ensure that Code is set to In-line, add the code in Listing 4, and click Apply.**

Listing 4. Code to add

```java
InputSchemaFactory factory = InputSchemaFactoryImpl.init();
Returns aReturns = factory.createReturns();
EList returnsList = Operation_tgt.getReturns();
aReturns.setScalar("" + Operation_src.getUpper());
Type aType = Operation_src.getType();
if (aType != null) {
    aReturns.setType(Operation_src.getType().getName());
} else {
    aReturns.setType("No Return Type specified");
} returnsList.add(aReturns);
```

9. **Click Apply.**

10. **The mapping from Operation: ownedParameter to Operation: argument is a submap. Select this mapping and then, in the Properties view, click the Details tab.**

11. **Create a new map called OwnedParameterToArgument.**

12. **Click the Input Filter tab (Figure 31), and select Filter Input Elements.**

13. **Enter the code shown in Listing 5.**
This filter limits the set of elements that go into this transformation. When working with an operation, a parameter can either be passed into the operation or out from the operation. In this case, you need to be concerned only about those being passed in.

14. Click **Apply**.

**Figure 31. Properties tab for Transformation Submap**

15. Select **File > Save All**.

Specify the OwnedParameter to Argument mapping

In this task, you specify the last mapping: OwnedParameter to Argument.

1. In the Outline view, double-click the **OwnedParameterToArgument** mapping to open it in the mapping editor.

2. Right-click the input element named **Parameter**, and select **Feature Filters > Advanced**.

3. Create these transformations (see Figure 32):
   - From **Parameter**: name to **Argument**: name to provide the name of the parameter
   - From **Parameter**: upper to **Argument**: scalar to provide the multiplicity for the parameter
• From Parameter: type: name to Argument: type to provide the type for the parameter

**Tip:**
Notice that, in Advanced view, there is a plus sign next to the type element. Click on that plus sign and then use the name attribute under that.

**Figure 32. Creating transformations to Argument**

4. Select **File > Save All.**

You have now created the five required maps and their element-to-element and attribute-to-attribute transformations between the input model and the output model. Your outline view should now look something Figure 33.

**Figure 33. Outline view**
Part 4. Model-to-model transformations and connecting models to EMFT JET transformations
Page 40 of 55
Section 7. Finalize and test the transformation

In this task, you will generate the transformation code from the transformation mapping.

Generate the transformation code

1. Before you generate code, review the Java source files that are in the project so far by opening the packages under the src folder (Figure 34) of the com.ibm.serviceSpec.report.xform.frontend project in the Package Explorer view. All of these were created when the project was created and as you have been editing the .mapping file.

![Figure 34. Java source files created in the Src folder](image)

2. In the mapping editor, right-click on the area to the right of the ServiceSpecificationFrontend button and select **Generate transformation source code** from the drop-down menu.

3. After the code generates, notice that you have two classes that have compilation errors. For both classes it is merely a matter of adding the proper imports.

4. To resolve the error in the file named OwnedOperationToOperationTransform.java:
   
   A. Double-click that file and, in the editor, enter `ctrl-shift-o`
organize imports. Select:

1. org.eclipse.uml2.uml.Comment. Then click Next.
2. org.eclipse.uml2.uml.Type. Then click Finish.

B. Use Ctrl-S to save this file.

5. To resolve the error in the ServiceSpecificationToServiceTransform.java file:

   A. Double-click the file name and, in the editor, enter ctrl-shift-o to organize the imports.

   B. Select org.eclipse.uml2.uml.Comment and then click Finish.

   C. Use Ctrl-S to save this file. You should not have any errors now.

6. Review the transformation files that have been generated under the `com.ibm.servicespec.report.xform,frontend.transforms` package (Figure 35).

   **Figure 35. Transformation files**

   ![Transformation files]

   Notice that, for each mapping that you see in the Outline view for the .mapping file (Figure 36), a .java file is generated that contains the source code for the transformation.

   **Figure 36. Outline view for the .mapping file**

   ![Outline view for the .mapping file]

   In addition, there is a .java file for each external custom extractor (the ones we provided for this tutorial), and a .java file for the overall transformation.
Connect the transformation to JET

In this task, you will add the code that calls the JET transformation from the mapping transformation. This is the final step that connects everything together (UML, EMF and EMFT JET)

1. In the Package Explorer view, in the
   com.ibm.serviceSpec.report.xform.frontend project under the
   src\com.ibm.servicespec.report.xform.frontend package,
   find and open this file:

2. In the createRootTransformation method, replace the body of the
   method with the code in Listing 6.

Listing 6. Code to replace in the createRootTransformation method

```java
return new RootTransformation(descriptor, new MainTransform()) {
    protected void addPostProcessingRules() {
        add(new JETRule("com.ibm.serviceSpec.report.xform"));
    }
};
```

3. Enter ctrl-shift-o to organize imports and resolve JETRule.

4. Change the @generated tag in the method to @!generated (Figure 37).

Figure 37. Changed code

```java
/**
 * Creates a root transformation. You may add more rules to the transformation here
 * @param transform The root transformation
 */
protected RootTransformation createRootTransformation(ITransformationDescriptor descriptor) {
    return new RootTransformation(descriptor, new MainTransform()) {
        protected void addPostProcessingRules() {
            add(new JETRule("com.ibm.serviceSpec.report.xform"));
        }
    };
}
```

Tip:
The @generated tag marks code that the code generator may overwrite on subsequent code generation. By negating this tag, you protect the code you added from being overwritten.

5. Use Ctrl-S to save your work.
Launch a runtime workbench

In this task, you will launch a runtime instance of the workbench to use in testing the newly created overall transformation. As you may recall from the introduction, this transformation takes a UML service model as input and generates the EMF Ecore for it (the result of Part 4, this tutorial). It then uses this Ecore as input to the EMFT JET transformation to generate an HTML service model report out of it (the result of Part 3). The diagram in Figure 38 illustrates this.

Figure 38. Comparison of Part 3 and Part 4 tutorial series results

![Diagram showing the process flow from UML service model to EMFT JET-generated HTML report.]

1. Within the Package Explorer view (Figure 39), double-click the `plugin.xml` file for the `com.ibm.serviceSpec.report.xform.frontend` project to open it in the Plug-in Manifest editor.

Figure 39. Package Explorer view

![Image of the Package Explorer view with the `plugin.xml` file highlighted.]

2. Switch to the Overview tab (Figure 40).

3. Under Testing, click on this text: **Launch an Eclipse application.**

Test the transformation

In this task, you will test the newly created transformation in the runtime workbench.

1. In the runtime workbench, close the Welcome screen and switch to the **Modeling** perspective.

2. Import the project interchange file **TestModel.zip** and select both the **EMFTJET Test** and **Test** projects.

3. Within the project, open and review the Service Model. For example, locate the Pre-Order Service Spec diagram (Figure 41).

**Figure 41. Modeling view**
Notice that the Pre-Order service specification (interface) has two methods (Figure 42):

- `getPreOrder`
- `updatePreOrder`

Later, you will verify that these are part of the report.

**Figure 42. Pre-Order service specification**

You will now create a new transformation configuration for the `ServiceSpecificationFrontEnd` transformation that you created in this tutorial.

4. Under the Modeling menu, select **Transform > New Configuration**.

5. Name the configuration `MyConfig` and make these selections (Figure 43):

   - **Forward transformation**: `ServiceSpecificationFrontEnd Transform`
6. Click **Next**.

**Figure 43. New Transformation Configuration view**
7. Select the **ServiceModel** (under EMFT.JET Test, then Models) as the input model and **Blank Model.emx** as the output model (Figure 44).

8. Click **Finish**.

**Tip:**
In this case, you need to care only about the input model. As such, you ensure that you pick the logical model as the Selected source file. For the target, you won't really end up using this file. If you were going to share this transformation with others, you would customize the configuration screen so that it did not ask for a target.

**Figure 44. Source and Target selection view**

Now you're ready to run this transformation:

9. Locate the **MyConfig.tc** file in the Project Explorer (Figure 45). Right-click it, and select **Transform > ServiceSpecificationFrontend Transform**.

10. As a result of the transformation execution, two new projects are created in the workspace. Examine the contents of the projects and validate that the elements of the input UML model have been mapped to the transformed text elements.
11. In the Atomic Service Specification Report folder, select the main.html file (Figure 46), right-click, and then select **Open with > Web Browser**.

12. When main.html file is open in your Web browser, click **Pre-Order Service** (Figure 47) to see the report. You should see that the report has these two operations defined in the UML model:

   - `getPreOrder`
   - `updatePreOrder`
Section 8. Wrap-up

You have completed this tutorial.

You have taken a look at several different ways in which Rational Software Architect enables you to capture and automate best practices. In particular, you looked at model-to-model transformations and how to connect them to EMFT JET transformations. You also saw how the capabilities of Rational Software Architect support a model-driven, asset-based, and service-oriented approach to software development, including plug-ins, Java Emitter Templates (JET), Eclipse Modeling Framework (EMF), and Unified Modeling Language (UML).

If it turns out that sometime later your organization requires a different format for the reports, you can change the underlying templates quite easily. In cases where the input model stays the same, you will need to change only the EMFT JET templates. In cases where more information is required, you will probably need to also update
the model mapping work that was completed. In most situations, the users of the services profile will not be aware of these changes and will continue to focus on the important aspects of their work: modeling their SOA solutions.

Although, in this case, you used the model mapping capabilities to migrate from UML, you can use these skills when you need to move from a UML model to another UML model. The only difference is that the output model that you select when setting up the project would be UML2, rather than a customer EMF model.

The reality of the technology industry is that we all continue to face pressure to get more done sooner and meet more stringent quality standards, all while often having to work in situations where our team spans not only across the campus but across the world. These automations help you improve productivity, enforce architectural integrity and improve the quality of your solutions.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project interchange containing the JET project</td>
<td>com.ibm.serviceSpec.report.xform.pi.zip</td>
<td>17KB</td>
<td>HTTP</td>
</tr>
<tr>
<td>UML Model to test your transformation</td>
<td>TestModel.zip</td>
<td>43KB</td>
<td>HTTP</td>
</tr>
<tr>
<td>Source code for the custom extractors</td>
<td>ServiceGroupExtractor.java</td>
<td>2KB</td>
<td>HTTP</td>
</tr>
<tr>
<td>Source code for the custom extractors</td>
<td>ServiceSpecificationExtractor.java</td>
<td>2KB</td>
<td>HTTP</td>
</tr>
<tr>
<td>Part 4 solutions file</td>
<td>part-4-solution.zip</td>
<td>64KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

*Information about download methods*
Resources

Learn

• Part 1 of this series, Using model-driven development and pattern-based engineering to design SOA: Part 1. Creating UML profiles and model templates.

• Part 2 of this series, Using model-driven development and pattern-based engineering to design SOA: Part 2. Patterns-based engineering

• Part 3 of this series, Using model-driven development and pattern-based engineering to design SOA: Part 3. Eclipse Modeling Framework Technology Java Emitter Template transformations

• Visit the The Eclipse Modeling Framework (EMF) site.

• Consult EMFT JET: Building code generators to support Model Driven Development.

• Review Rational Software Architect Help, specifically the Authoring JET transformations topic.

• See IBM® Redbooks™ for Building SOA Solutions with the Rational Software Delivery Platform.

• See IBM Redbooks for Patterns: Model-Driven Development Using IBM Rational Software Architect.

• Visit the IBM developerWorks Pattern Solutions site to learn how to use patterns to improve productivity in software design and development.

• Read the IBM Systems Journal article on Model-driven Software Development, Volume 45, Number 3 (2006).

• Learn about reusable assets, recipes, and patterns in this developerWorks article by Grant Larsen and Eoin Lane: Building SOA applications with reusable assets: Part 1. Reusable assets, recipes, and patterns (March 2006).

• Complete these other tutorial series:
  • Model service-oriented architecture with Rational Software Architect (IBM developerWorks, 2007)
  
  • Design SOA services with Rational Software Architect, a 4-part developerWorks tutorial series (2006-2007)

• Explore Rational computer-based, Web-based, and instructor-led online courses. Hone your skills and learn more about Rational tools with these courses, which range from introductory to advanced. The courses on this catalog are available for purchase through computer-based training or Web-based training. Additionally, some "Getting Started" courses are available free of charge. In particular, you'll find these courses helpful:
  • DEV325: Essentials of Model-driven Architecture (MDA), an introductory-level Web-based training course
• DEV510: Applying the Enterprise Patterns in IBM Rational Software Architect V6, an introductory-level Web-based training course
• DEV498: Pattern Implementation Workshop with IBM Rational Software Architect V7.0 (RD801), a 4-day, instructor-led classroom course with hands-on labs

• Stay current with developerWorks technical events and Webcasts.
• Browse the technology bookstore for books on these and other technical topics.

Get products and technologies
• Download a trial version of Rational Software Architect v7
• Download trial versions of IBM Rational software.
• Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss
• Check out developerWorks blogs and get involved in the IBM developerWorks community.

About the authors
Lee Ackerman
Lee Ackerman is a Sr. Product Manager with the Rational Expertise Development & Innovation team. He focuses on creating intellectual capital assets that enable users of the Rational model driven development tooling to succeed in creating J2EE and SOA solutions.

Bertrand Portier
Bertrand Portier is an IT Architect with IBM’s worldwide SOA Technical Sales organization. He used to work for IBM SWG SOA Advanced Technologies to help customers with strategic SOA transformations and create SOA assets for IBM. He is heavily involved in model-driven and asset-based development, and has extensive experience with Web services. A regular speaker at conferences and the author of several technical articles, he also co-authored a redbook on SOA solutions.
Chris Gerken

**Chris Gerken** is a member of the Asset Reuse Enablement Team within the IBM Software Services for WebSphere group. He created and wrote the Design Pattern Toolkit.

**Trademarks**

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.