IBM Rational University

Rational. software

Pattern Implementation Workshop with IBM Rational Software Architect

RD801/DEV498 April 2007
Student Workbook
Part No. 800-027313-000

IBM Corporation
Rational University
Pattern Implementation Workshop with IBM Rational Software Architect
Student Workbook

April 2007

Printed in the United States of America.

This manual prepared by:
IBM Rational Software
555 Bailey Ave.
Santa Teresa Lab
San Jose CA 95141-1003
USA

Lab 1: Introducing JET

Objectives

After completing this lab, you will be able to:

- Create a new EMFT JET project
- Configure the plug-in
- Run a JET Transformation

Scenario

In this lab exercise you will create a new JET Transform and learn about the transformation's component parts.

You will need Rational Software Architect V7 or later. These instructions are targeted to Rational Software Architect V7.

Task 1: Create a New EMFT JET Transformation Project

1. On the **File** menu, click **New Project > EMFT JET Transformation Project**. Click **Next**.
2. Enter `my.first.transform` as the Project name and click **Finish**.
3. In the Package Explorer view in the Java perspective, expand the newly created project named `my.first.transform`.

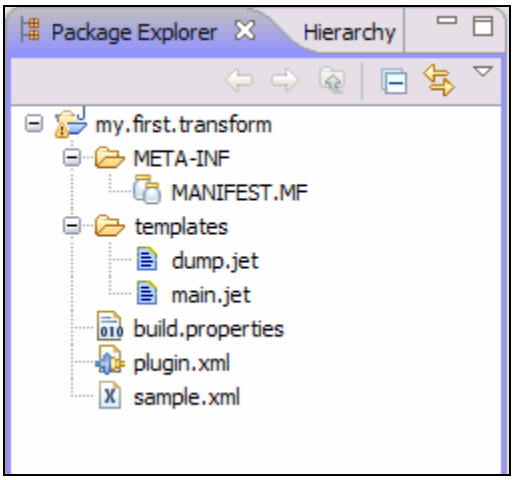


Figure 1 - 1: EMFT JET Transformation Project, my.first.transform

Task 2: Configure the Plug-in

1. Open the editor for the `plugin.xml` file and click the editor's **plugin.xml** tab.

Figure 1 - 2: EMFT JET Transformation Project, my.first.transform

Although each JET transform is implemented as an Eclipse plug-in, you really don't have to know about Eclipse plug-ins in order to build a JET transform.

2. Click the plugin.xml editor's **MANIFEST.MF** tab to view the transform's metadata.

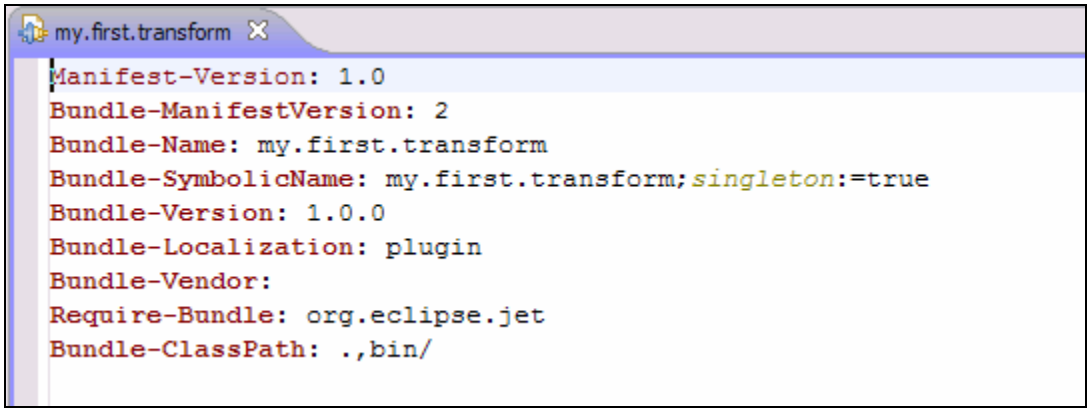


Figure 1 - 3: Meta-information for my.first.transform

The only piece of metadata that you may care about is the symbolic name which is the transform's id. This string value is used in several advanced transform functions.

3. Open the editor for `main.jet` to see the high-level template (named in the `plugin.xml` file in the `startTemplate` attribute).

Most of the template test is static, but there are several interesting features:

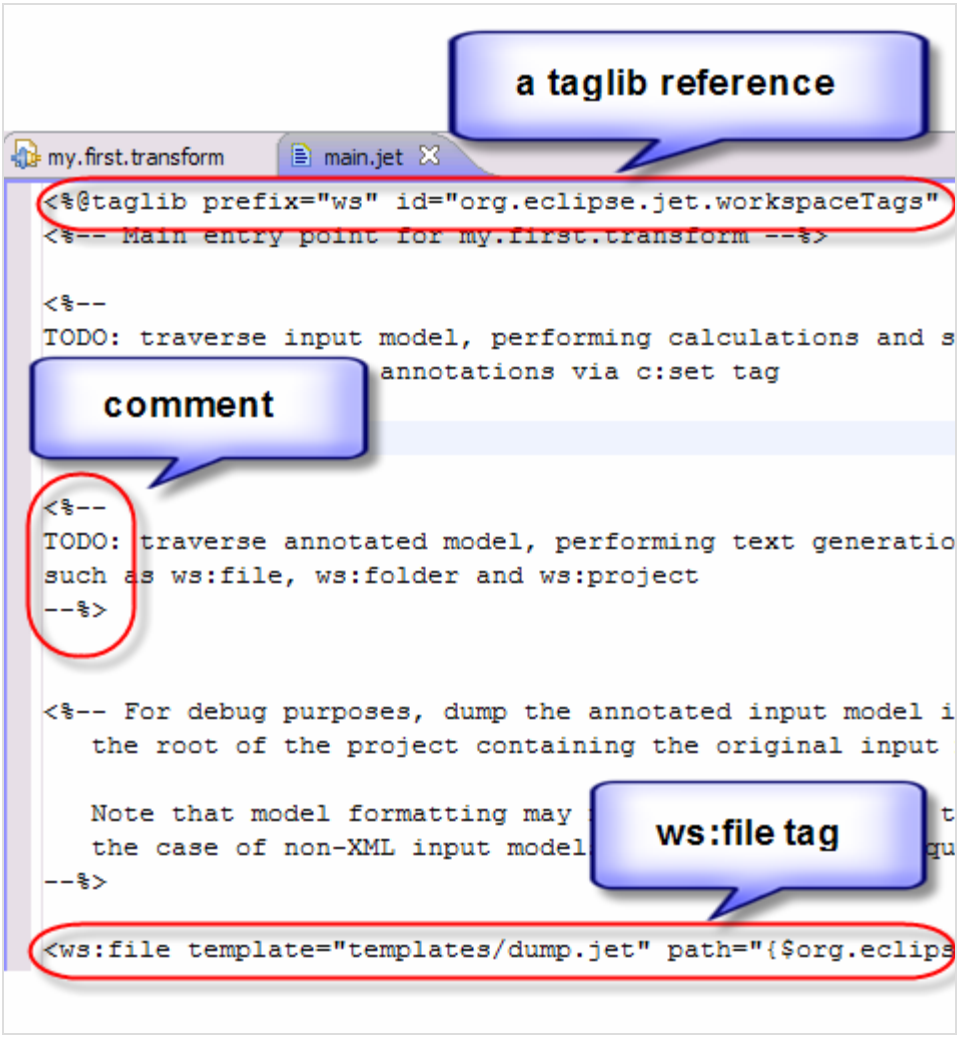


Figure 1 - 4: Key Information in the Plug-in Manifest

The **taglib** reference defines a tag library containing tags that may be used in this template.

The **ws:file** tag is one such tag.

The rest of the template is static text, mostly containing **JET template comments** that do not appear in the generated text.

The dump.jet template also contains a mix of tags and static text.

Task 3: Filter the Project Explorer View

1. Click the **Filters** icon (circled in the image below) and the **Filters** menu item to change the Package Explorer filters.

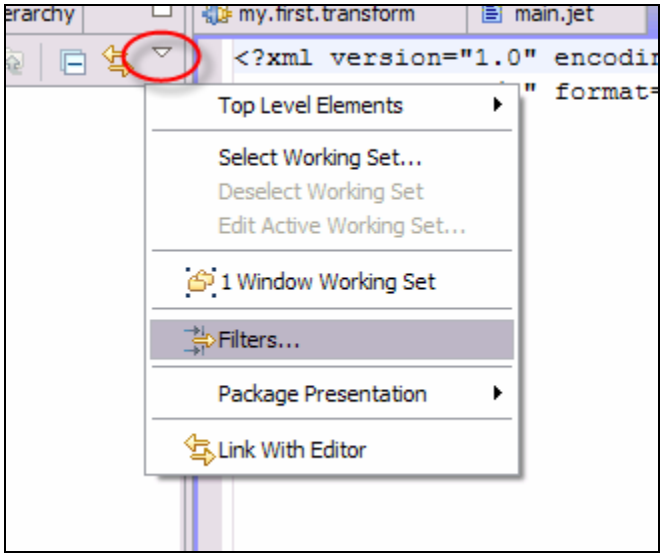


Figure 1 - 5: Opening the Filters Dialog for the Project Explorer

2. Clear the box next to **Java elements from JET Transformation projects** and click **OK**.

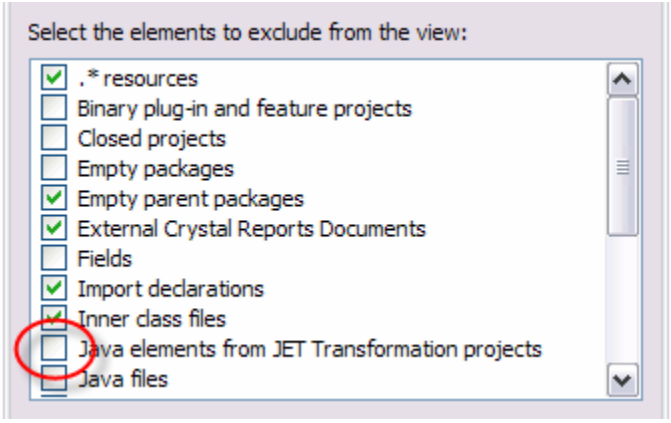


Figure 1 - 6: Turning off Java Elements filtering

3. The Package Explorer will now display several additional project items. Fully expand the `jet2java` package.

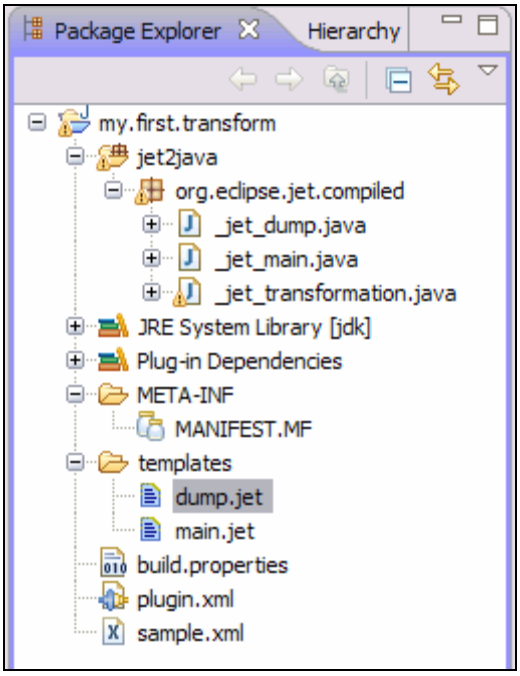


Figure 1 - 7: The Package Explorer View

The generated Java classes (created when the transform's templates are edited) are normally hidden (because you don't need to interact with them). Note that there is a Java class for each of the two templates, as well as a _jet_transformation.java class that acts as an index into the other classes.

4. Use the **Filters** menu item to hide these Java elements again.

Task 4: Run the Transformation

1. Open the `sample.xml` file and, using the editor's source tab, add some arbitrary, but valid, XML.

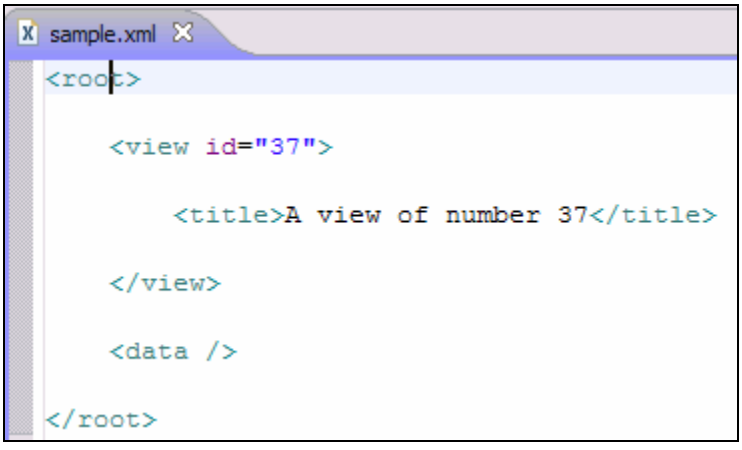


Figure 1 - 8: The contents of the sample.xml file

2. Transform the model in the XML file (the XML content) with the transformation. Click the **Run** icon and then click the **Run** menu item.

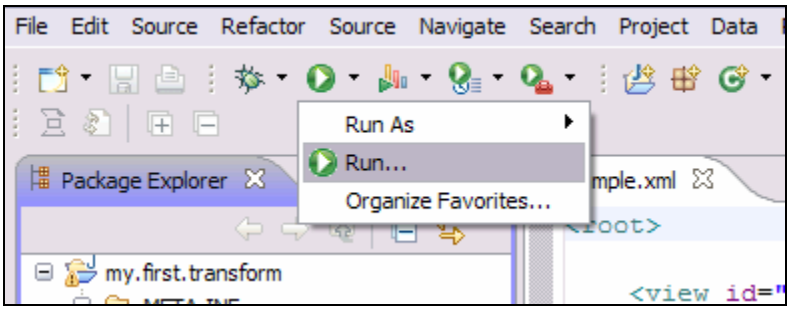


Figure 1 - 9: Running the Transformation

TIP: You can also right-click the `sample.xml` file and select **Run As > Input for JET Transformation**.

3. The list of available configurations will vary based on the specific IDE that you're running. Select the Jet Transformation configuration and click the **New** button (circled in the image below).

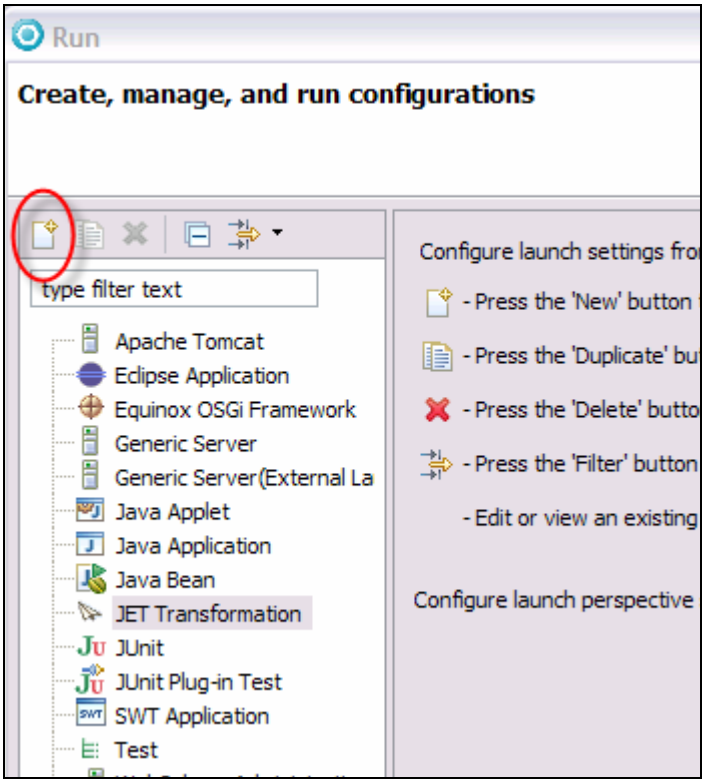


Figure 1 - 10: Running the Transformation

4. If the file to be transformed is in the project containing the transform, then the new transformation instance should be initialized correctly. Otherwise you would have to set the various properties manually.

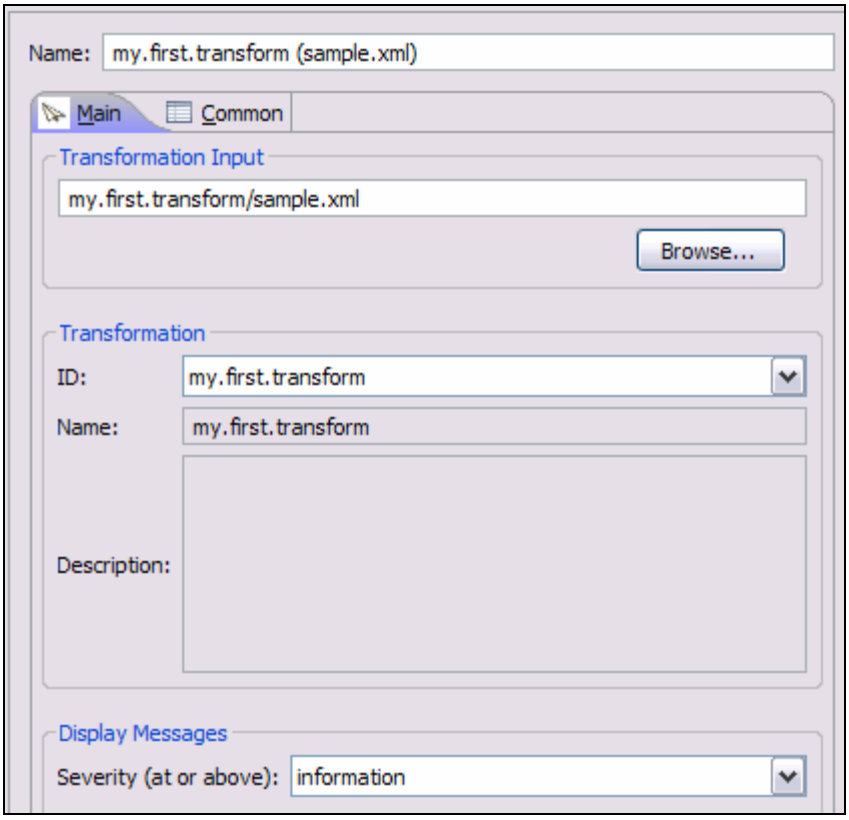


Figure 1 - 11: Setting Transformation Properties

5. Click the **Run** button and you should see a new file, `dump.xml`, created by the transformation. Open that file and you should see the original XML. By default, new transforms simply write out the input model in a file called `dump.xml`.

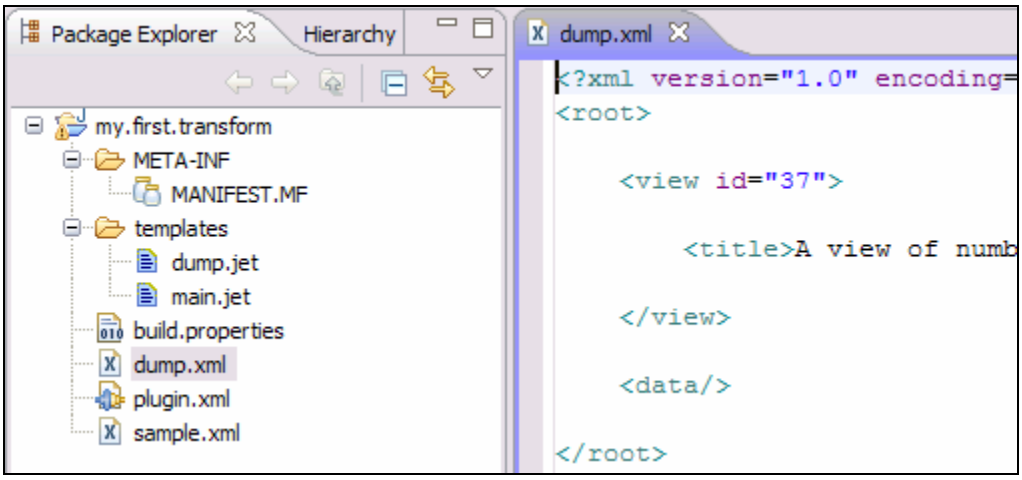


Figure 1 - 12: Setting Transformation Properties

6. Delete the `dump.xml` file and run the transform again by simply clicking the **Run** icon.

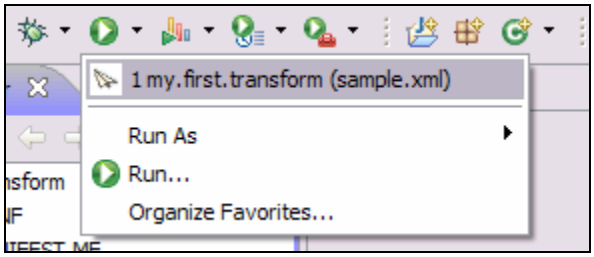


Figure 1 - 13: Running the Transformation

Lab 2: Using XPath

Objectives

After completing this lab, you will be able to:

- Use basic JET tags and XPath to access a model

Given

- The project interchange file `UsingXPath.zip`

Scenario

In this lab exercise, you will use basic JET tags and XPath to access a sample model in a number of common ways.

Task 1: Set up the Lab

1. Begin by using the Import from Project Interchange wizard to import the `XPath Exerciser` project in file `UsingXPath.zip`.

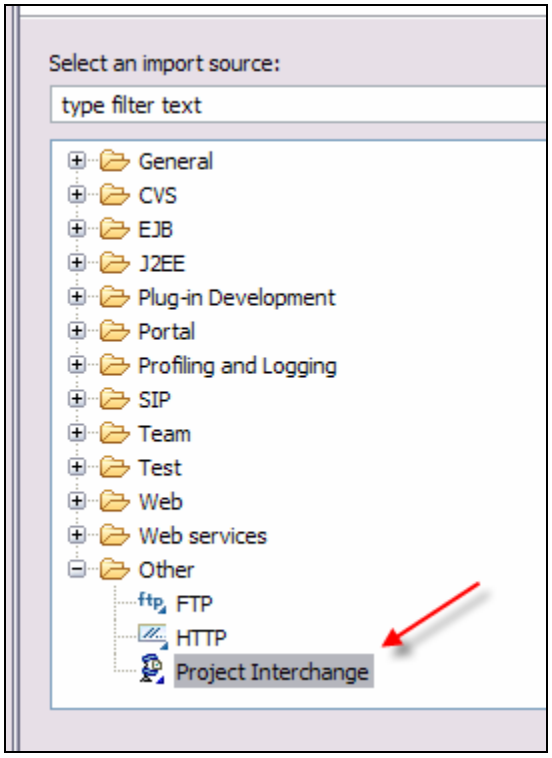


Figure 2 - 1: EMFT JET Transformation Project, my.first.transform

The transformation project contains several files. You will need to modify files `xpath001.jet` – `xpath015.jet`. Each .jet template has instructions describing a small number of JET tags to be written. Each template also shows the expected output from those tags.

2. The file `sample.xml` contains the input model to be used for this exercise. To test your work, apply the XPath Exerciser transformation to the `sample.xml` file.

Lab 3: Authoring a JET Transform Manually

Objectives

After completing this lab, you will be able to:

- Revise an existing JET Transformation

Given

- The project interchange file `AuthoringTransformsManually.zip`

Scenario

In this lab you will author a transform that uses the basic JET tags, and which generates both Java and non-Java artifacts.

Task 1: Revise the Transformation lab.ibean.transform

In this task, you will work through a transform that has been partially completed.

1. Begin by using the Import from Project Interchange wizard to import both projects in file `AuthoringTransformsManually.zip`.
2. Look at the projects that were imported. There is a Java project named `IBean Java Project` that contains some Java classes. The `lab.ibean.transform` project is a transform that can generate Java projects.

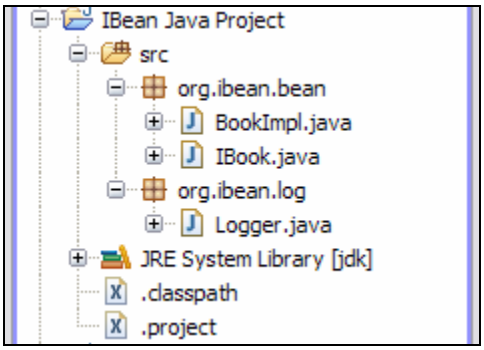


Figure 3-1: The contents of IBean Java Project

The `IBean Java Project` project represents the kind of project that `lab.ibean.transform` will generate. In addition to the project and its required meta-data files, there is always a `Logger` class and pairs of business classes. Each pair of classes contains a specialized Java bean and an interface for that bean.

The bean is specialized in that every `setter` method invokes the logger to log an "object modified" message. The interface names each `getter` and `setter` method.

Note that `getter` methods for boolean properties begin with `is` and variable names begin with `field` to avoid accidental use of reserved words such as *package* or *class*.

The transform has in it several sample XML files that illustrate the variability in the pattern. The transform as originally loaded in your workspace, however, only generates the Java project, meta-data files, and the logger class. It does not generate either the interfaces or the bean implementations.

3. Your task in this lab is to add the necessary tags and templates to the `lab.ibean.transform` transform so that it also generates those interfaces and bean implementations correctly.

Lab 4.1: Authorization Bean Exemplar Authoring

Objectives

After completing this lab, you will be able to:

- Create an EMFT JET based transform using Exemplar Authoring

Given

- The project interchange file `AuthorizationBean-ExemplarAnalysis.zip`

Scenario

In this lab you will perform Exemplar Authoring on a working bean. As a result, the transform will be able to take information about a set of beans as input, and then generate the Java code necessary for the set of beans.

Task 1: Set up the Lab

1. Use the Import from Project Interchange wizard to import all of the projects in file `AuthorizationBean-ExemplarAnalysis.zip`.
2. Look at the project that was imported, this project makes up the exemplar:
   - The exemplar is in a single project: **Authorization Beans**. The transform you must build from this exemplar is the same transform you built by hand in Lab 3. The difference here is that you will be using the Exemplar Authoring tools in Rational Software Architect to build the transformation.
3. Complete the lab using the Exemplar Authoring tool. If you need assistance, there is a step-by-step guide to completing the task located in the `solution` folder for this lab on the Student CD.

Lab 4.1 Solution: Authoring the AuthorizationBean Exemplar

Objectives

After completing this lab, you will be able to:

- Create an EMFT JET based transform using Exemplar Authoring

Given

- The project interchange file `AuthorizationBean-ExemplarAnalysis.zip`

Scenario

In this lab, you will perform Exemplar Authoring on a working Java bean. As a result, the transform will be able to take information about a set of beans as input, and then generate the Java code necessary for the set of beans.

Task 1: Set up the Lab

In this task you will set up your environment for this lab.

1. Begin by using the Import from Project Interchange wizard to import all of the projects in file `AuthorizationBean-ExemplarAnalysis.zip`
2. Look at the project that was imported.

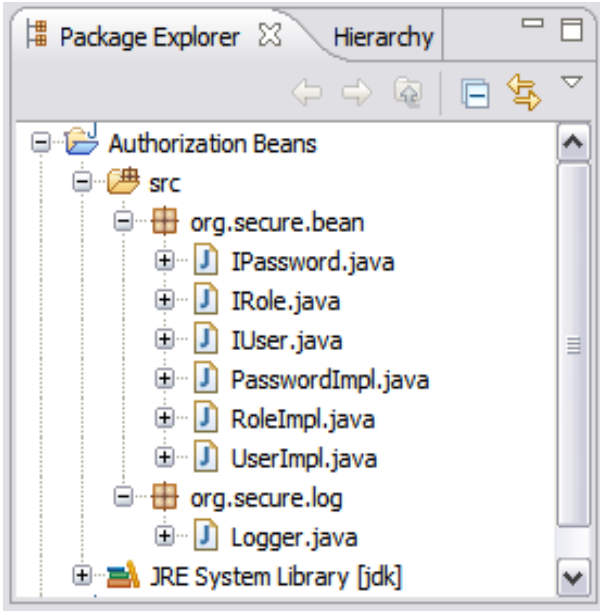


Figure 4.1 - 1: Authorization Beans Exemplar Project

The project `Authorization Beans` Java Project is your exemplar. As such, it represents the kind of project that your transform will generate. In addition to the project and required meta-data files, there is always a `Logger` class and pairs of business classes. Each pair of classes contains a specialized bean and an interface for that bean.

The bean is specialized in that every setter method invokes the logger to log an “object modified” message. The interface names each `getter` and `setter` method.

Note that `getter` methods for Boolean properties begin with `is` and variable names begin with `field` to avoid accidental use of reserved words such as “package” or “class”.

As with any Exemplar Authoring exercise, be sure to ask the SME (the instructor in this case) if you have any questions about the implementation of the exemplar application or about the points of variability to be supported by the transform.

Task 2: Create Exemplar Authoring Project

In this task, you will create an Exemplar Authoring project.

1. Create a new JET transformation project called `authorization.bean.transform`. Use the EMFT JET Project with Exemplar Authoring project wizard.

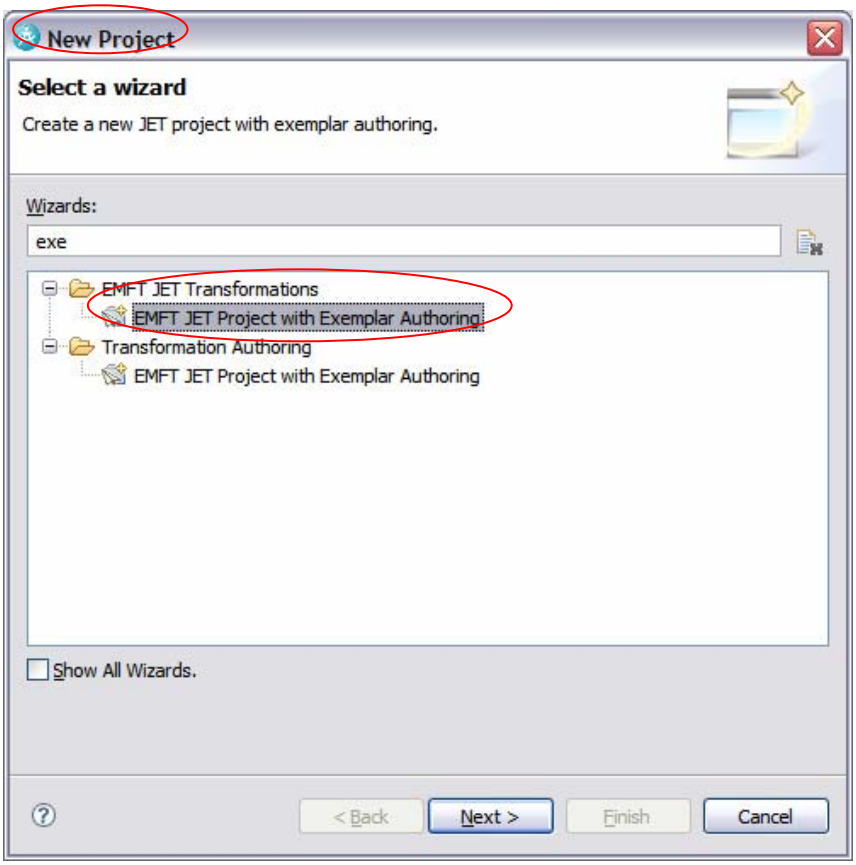


Figure 4.1 - 2: Creating an EMFT JET Project with Authoring Exemplar

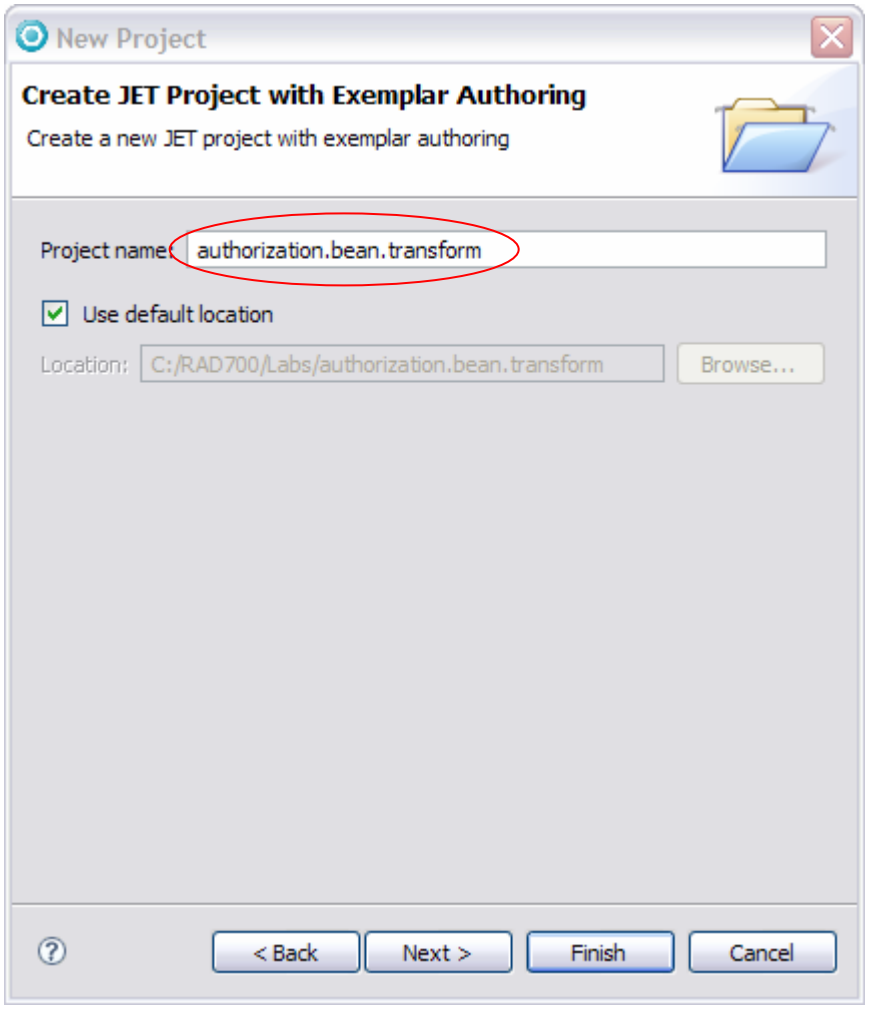


Figure 4.1 - 3: Specifying the project name

2. Be sure to specify that the Authorization Beans project is selected as the **Exemplar scope**.

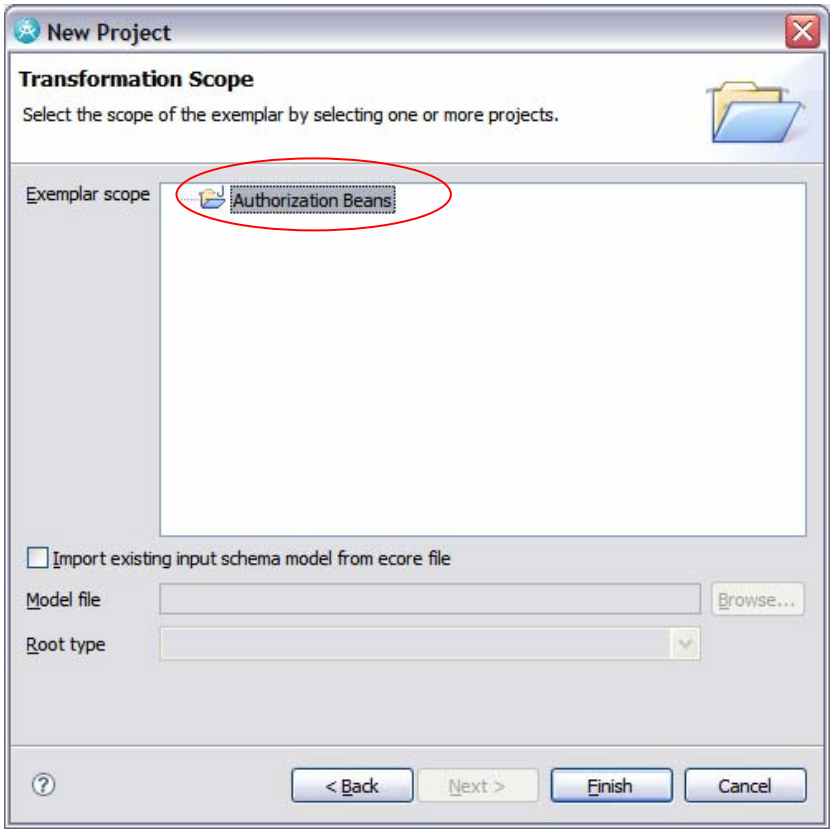


Figure 4.1 - 4: Specifying the Exemplar Scope

3. The Exemplar Authoring tool should now display the Authorization Beans exemplar and an empty model

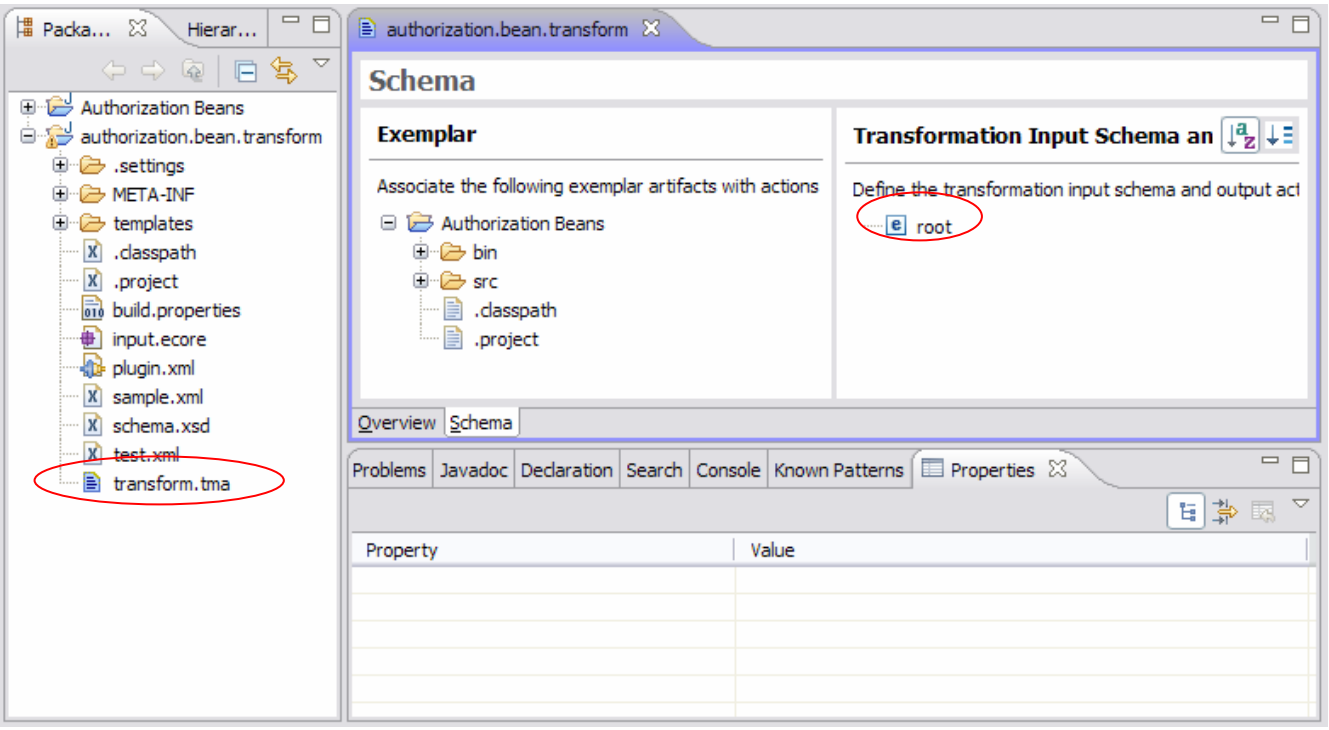


Figure 4.1 - 5: The authorization beans exemplar and the empty model

When the wizard completes, you'll see a new plug-in project with the name you entered into the wizard. This plug-in project contains the same files and folders that the New EMFT JET wizard creates, but it also contains a file named transform.tma, this file will contain the model you build by performing Exemplar Analysis on your exemplar.

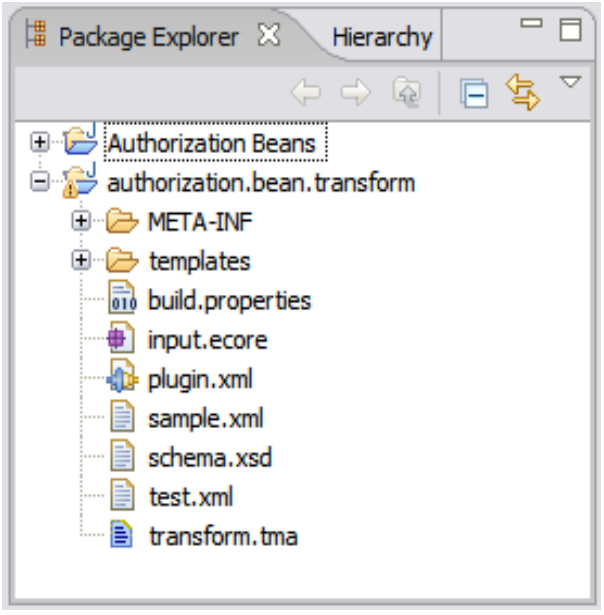


Figure 4.1 - 6: The files within the transform project

The editor for transform.tma is shown above. It has two side-by-side panes. The left pane contains a file system view of the project(s) you said contains your exemplar. The right pane contains several kinds of information. The right pane contains the transformation input model schema (for now there's only a single element type called "root"). For each element type the right pane will show the actions to be taken by the transform whenever it

encounters an element of that type.

Task 3: Populate the Model: Items Created Once

1. You will use a one-word name, `beanSet`, to describe the entire set of files in the exemplar. Create a second-level model type by that name.

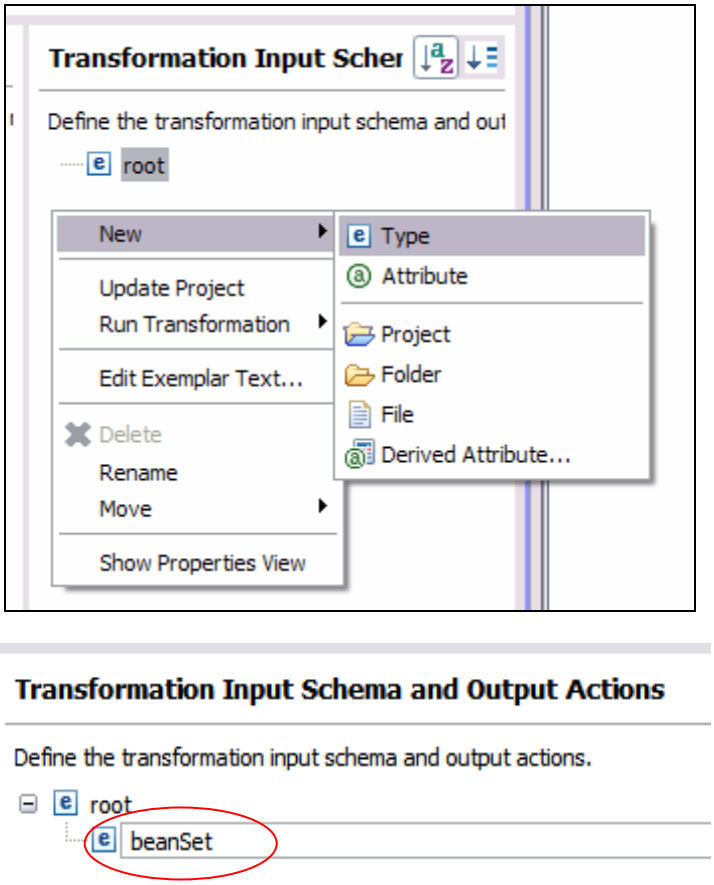


Figure 4.1 - 7: Creating the beanSet type

2. Add an attribute called `name` to `beanSet` . This attribute will be used to capture the name of the project.

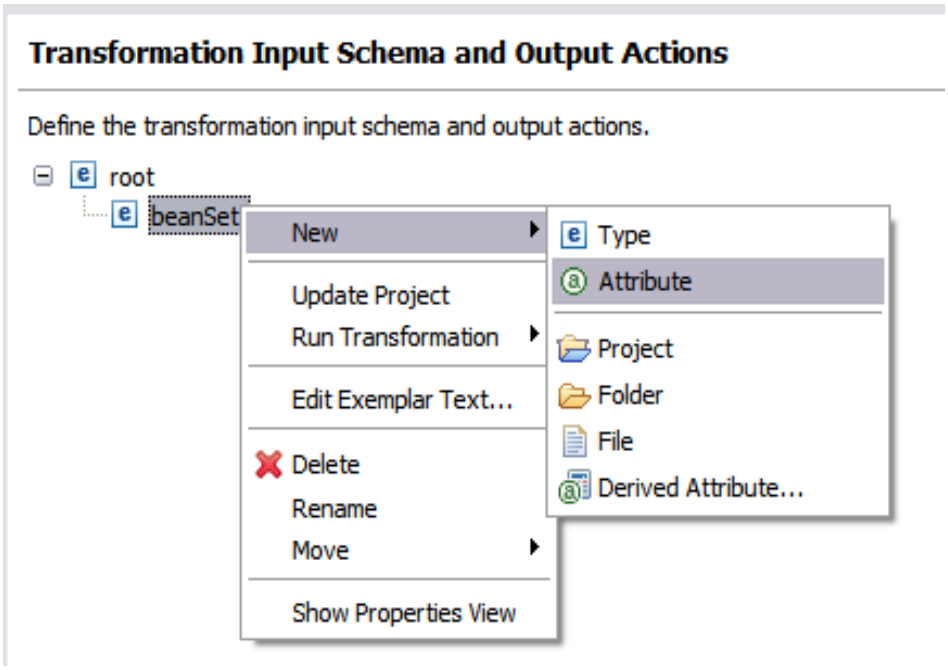


Figure 4.1 - 8: Creating the console type

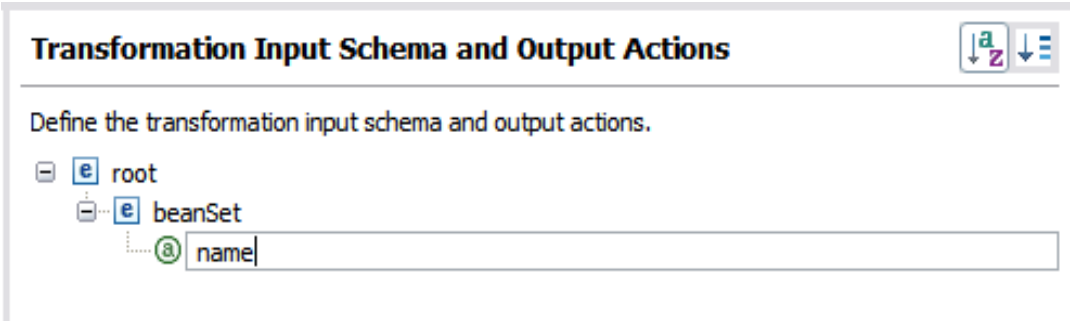


Figure 4.1 - 9: Adding a name attribute to the beanSet type

3. Identify the artifacts that will be created only once for each application of the transform. They include:
   - The Java project, **Authorization Beans**
   - The project meta-data files **.classpath** and **.project**
   - The Logger class **org.secure.bean.log.Logger.java**
4. Drag each of these artifacts from the left pane onto the **beanSet** type icon in the right pane. Be careful not to drop any of the artifacts onto the `Create Project` action.

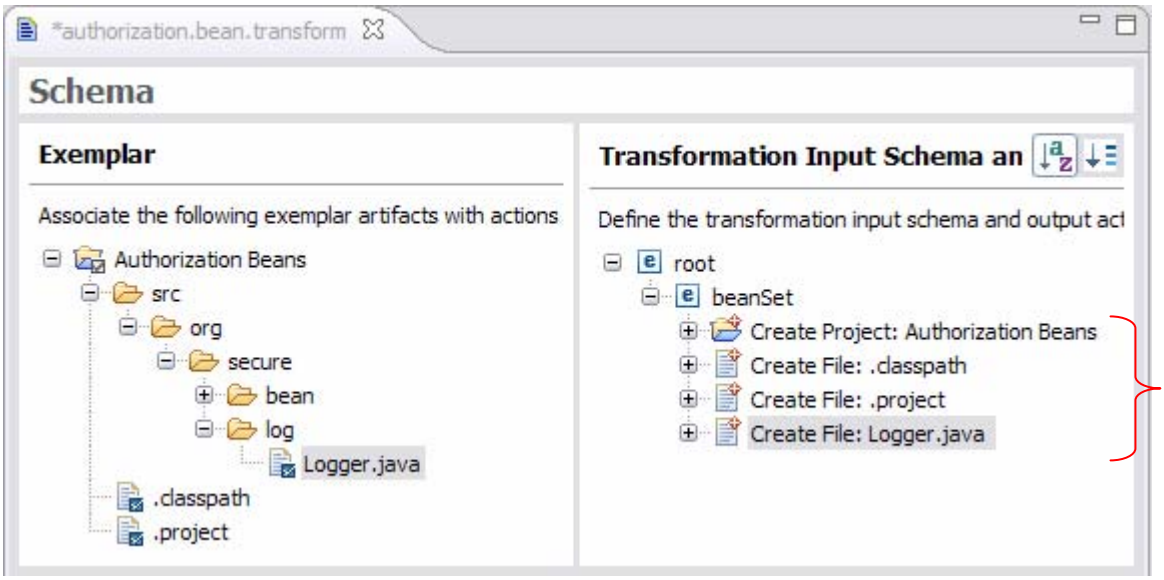


Figure 4.1 - 10: Artifacts added under beanSet

TIP: Notice that in the left-hand pane the view is updated so that a checkmark is added once an artifact is associated with an action.

TIP: The name parameter shows that this project will always be created with the name “Authorization Beans”. You want this name to be variable, and best practices call for using a derived attribute to specify and hold that variable project name. The derived attribute, in turn, will be based on an attribute that’s included as part of the input model.

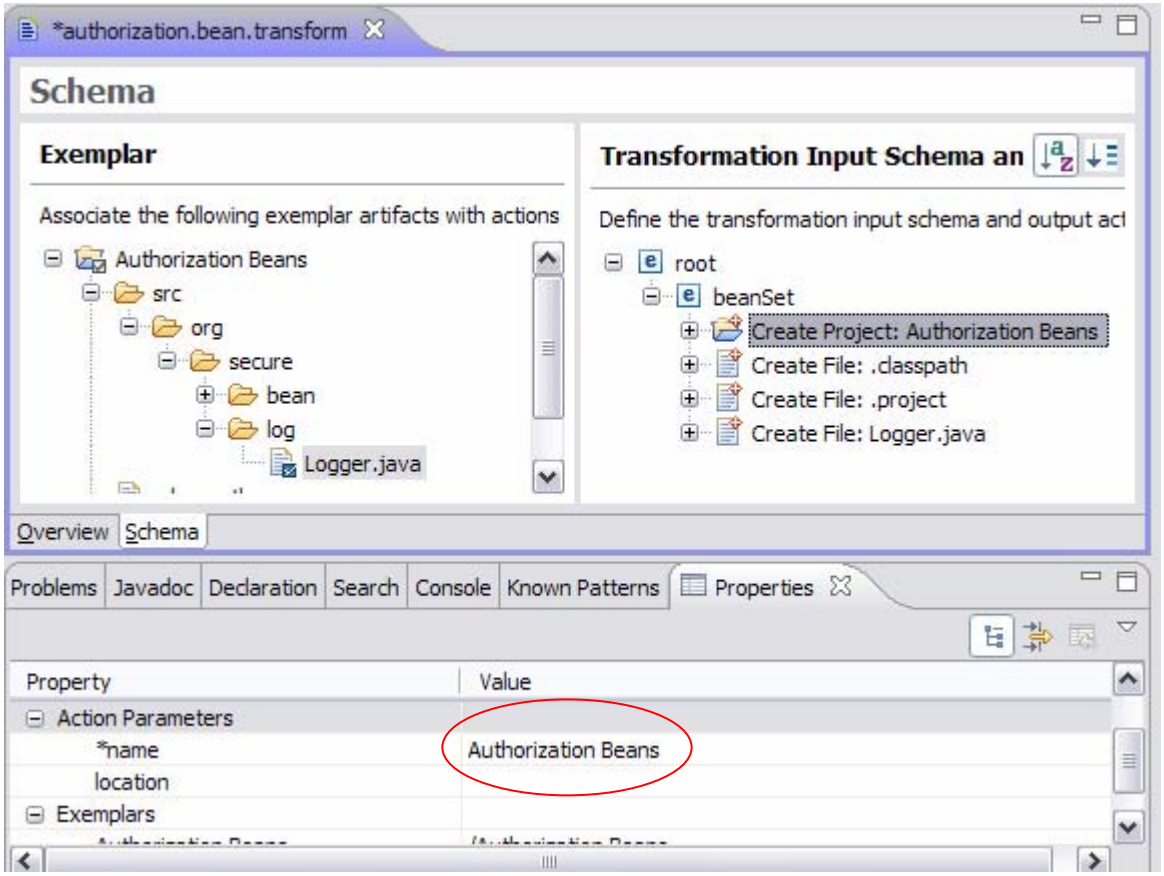


Figure 4.1 - 11: The name property for the Create Project: Authorization Beans action

TIP: You will need to use the Properties view in your perspective to accomplish this task (and many others). To add the view, select **Window > Show View > Properties**.

5. The project name will be taken completely from the value of a derived attribute that you are about to define. Select the **name** parameter value ("`Authorization Beans`") and click on the **Replace with Model Reference** menu action.

TIP: Note that you've already added a new attribute called **name** to **beanSet**. The value passed into the transform in this attribute will be used to build the derived attribute.

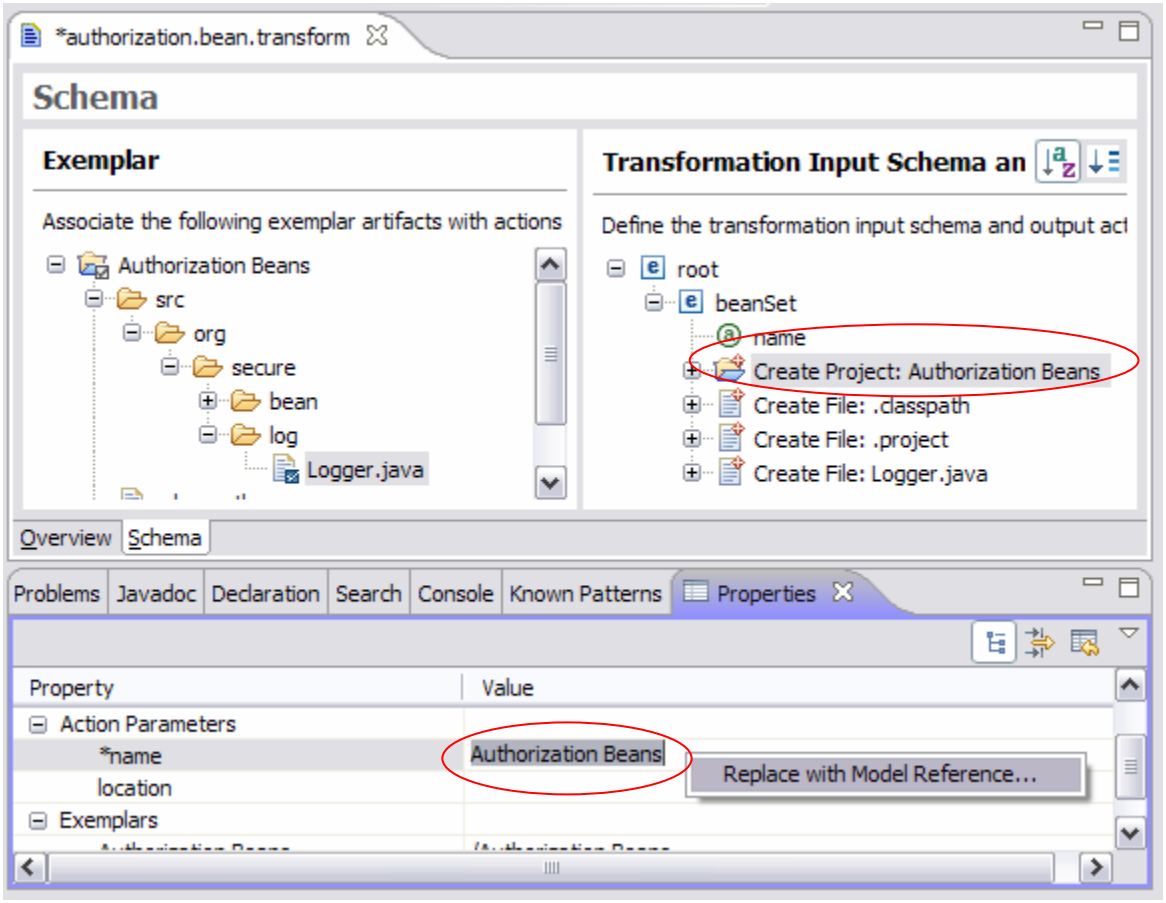


Figure 4.1 - 12: Replacing the default text with a model reference

TIP: The **Replace with Model References** dialog lets you select the model attribute whose value will replace the selected text.

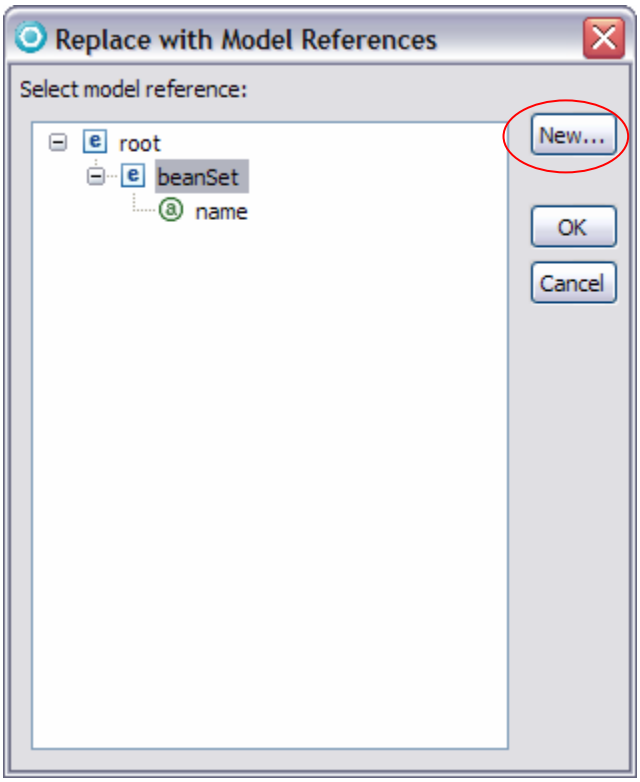


Figure 4.1 - 13: Creating a new model reference

6. You need a derived attribute that's not been defined yet, so you select **beanSet** and then click **New** to define that derived attribute.

TIP: Note that since this derived attribute will contain the name of the project to be created, and since the **beanSet** type (and its subtree) contains all of the information needed to apply the transform once, you need to select the **beanSet** type before clicking **New** so that the derived attribute is defined on the **beanSet** type.

TIP: The **Create New Derived Attribute** dialog lets you define the new derived attribute. The calculation field lets you insert model references to define the formula used to build the derived attributes value.

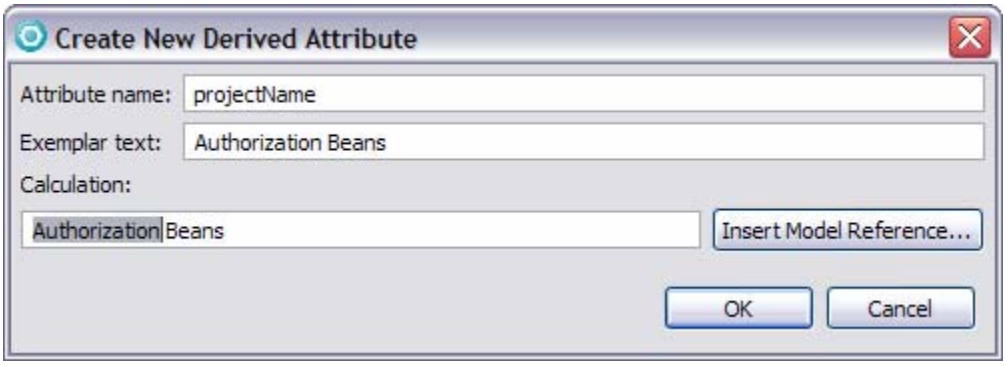


Figure 4.1 - 14: Preparing to insert a model reference

7. Specify `projectName` as the value for the **Attribute name**.

8. You want the value of the derived attribute to be the value of the **name** attribute from the beanSet (with first character uppercased) folloyoud by the string " `beans`". Within the **Calculation** field, select `Authorization` and then click on **Insert Model Reference**.
9. In the **Select Model Reference** dialog, select the **name** attribute, and then click **OK**.

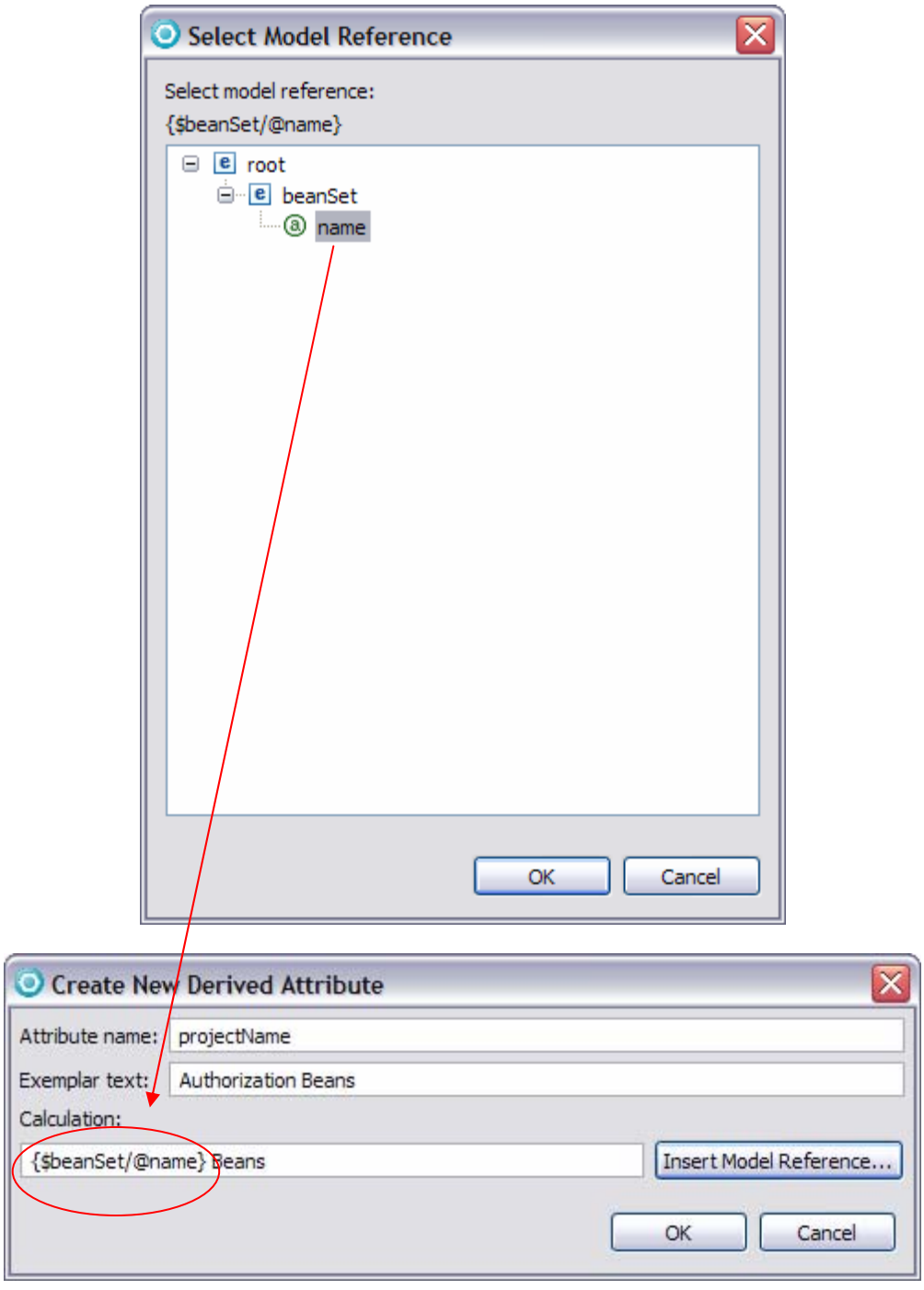


Figure 4.1 - 15: The updated calculation field that uses the name attribute from the beanSet type

TIP: Note that the XPath query expression $beanSet/@name assumes that the variable $beanSet is associated with a model node of type <beanSet>

10. Add the `uppercaseFirst` function to uppercase the first character in the value of $beanSet/@name

Figure 4.1 - 16: Updated calculation with the uppercaseFirst function applied

11. Click **OK** in the **Create New Derived Attribute** dialog.

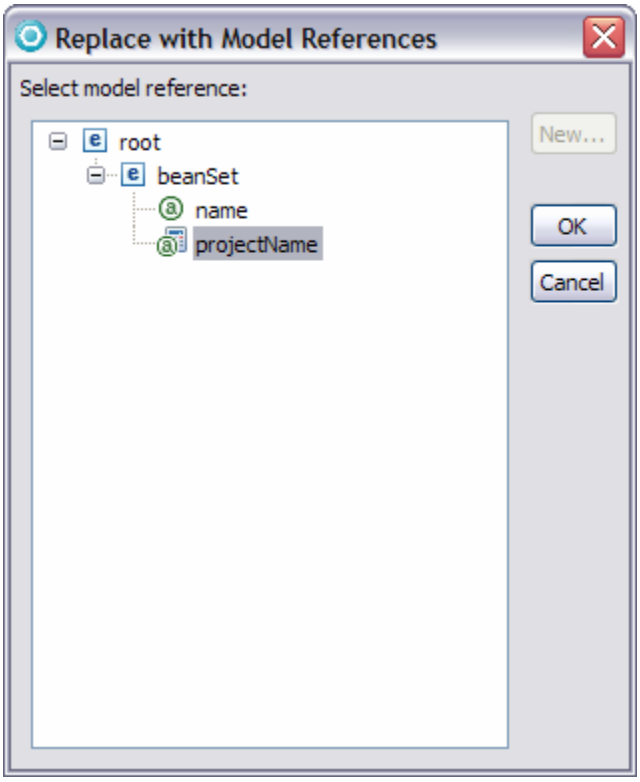


*Figure 4.1 - 17: The newly created derived attribute now appears in the **Replace with Model References** dialog.*

12. Select **projectName** and click **OK** to return to the **name** parameter property.

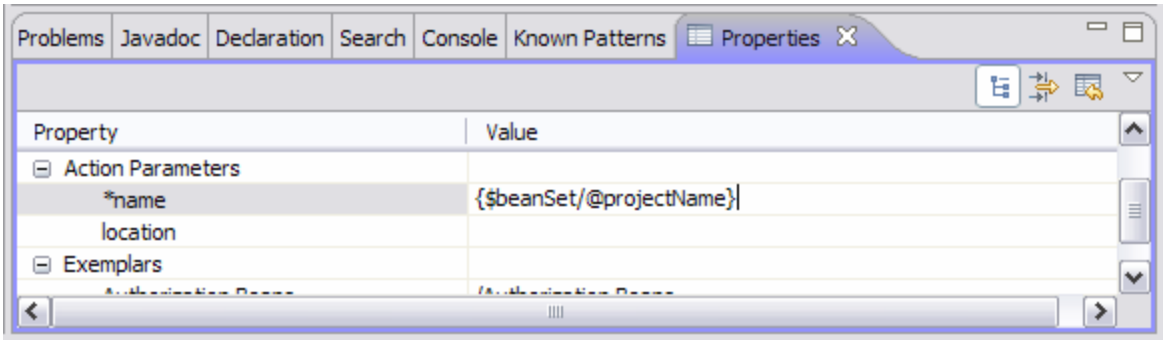


Figure 4.1 - 18: Artifacts added under console

13. When you run the transform and the project is created, the project name will now be taken from derived attribute **projectName**.

Next, let's start to define naming attributes for use with Logger.java.

14. Add an attribute called `basePackage` to `beanSet`, from which you'll derive package names for Java classes and the corresponding directory names.

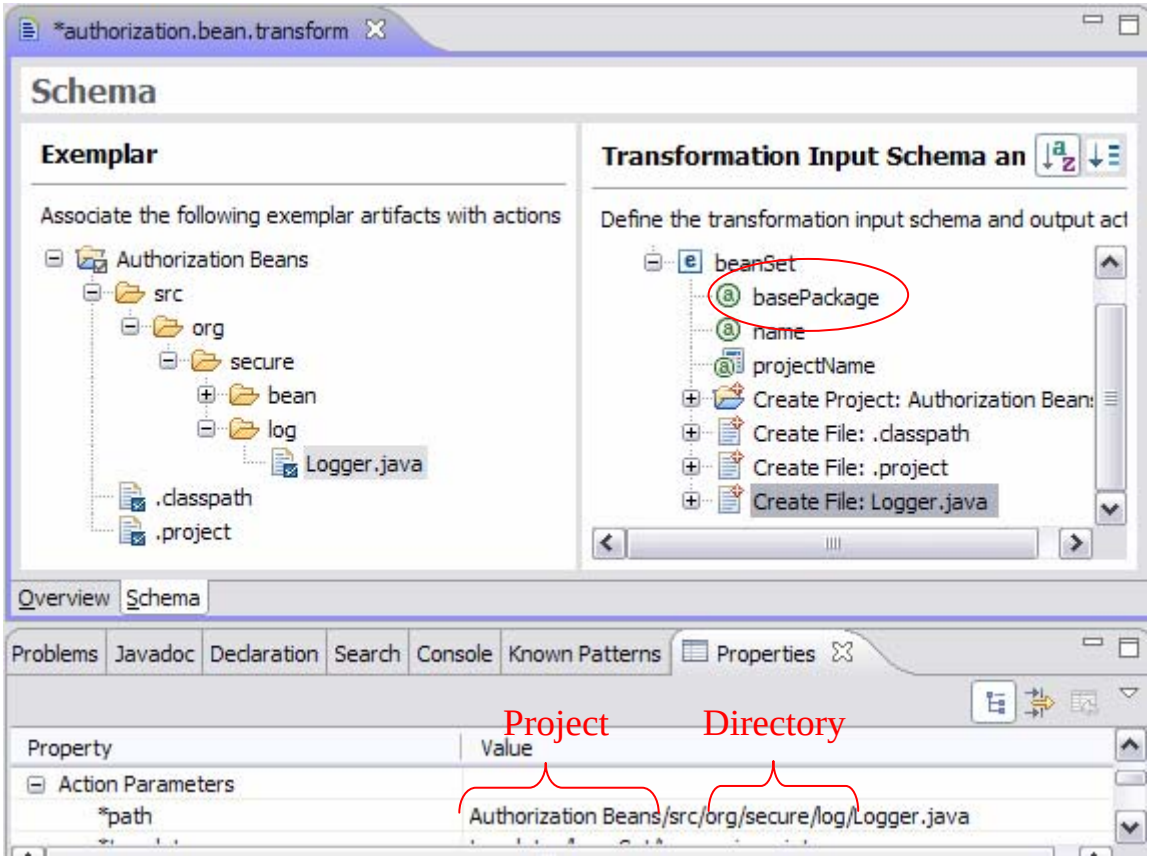


Figure 4.1 - 19: Note the newly created attribute. Also, when looking at the properties for the path of Logger.java there are aspects that will vary.

15. Select the **Create File: Logger.java** action.

16. In the **path** field within the **Properties** view, select `Authorization Beans`, right-click and click **Replace with Model Reference**.

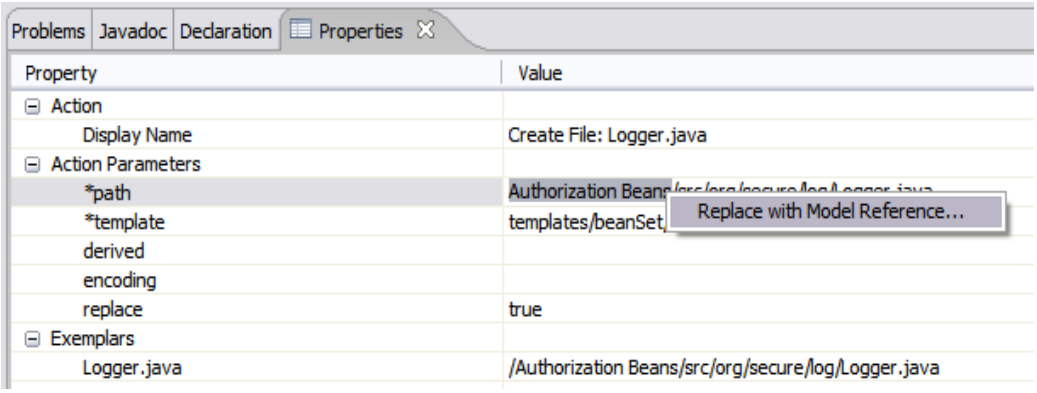


Figure 4.1 - 20: Selecting the text from the path property that needs to be replaced with a model reference

17. Select **projectName** and then click **OK**.

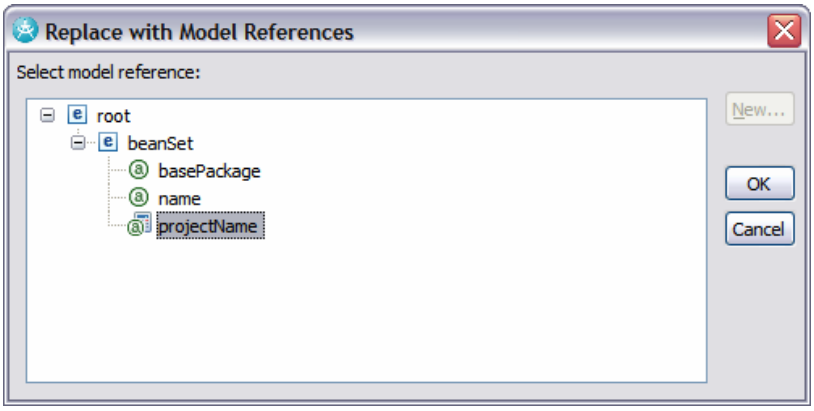


Figure 4.1 - 21: Selecting the projectName derived attribute

TIP: The remaining part of the path property for the "**Create File: Logger.java**" action needs to be marked up with references to two new derived attributes:

- logPackage will reference the package that the Logger.java file belongs two.
- logDirectory: will reference the directory that the Logger.java file should be written to.

These derived attributes are related to each other as well as to the basePackage attribute. The logPackage includes the basePackage, but as seen in our exemplar, you need to append another package to the end for the Logger class. This additional package is called `log`. Once you have the logPackage attribute, all you need to do for calculating the logDirectory attribute is to convert the "." characters into "/" characters.

18. In the **path** field within the **Properties** view, select `org/secure/log`, right-click and click **Replace with Model Reference**.
19. Select **beanSet** and click **New**.
20. Specify `logPackage` as the **Attribute name**.
21. Update the text found in the Exemplar text field, replacing the "/" character with "." character.

TIP: Later when editing the templates associated with this transform, the Exmplar text will be used to help guide you in replacing static text with references to the attributes you've created.

22. Select the `org/secure/log` text in the **Calculation** field.

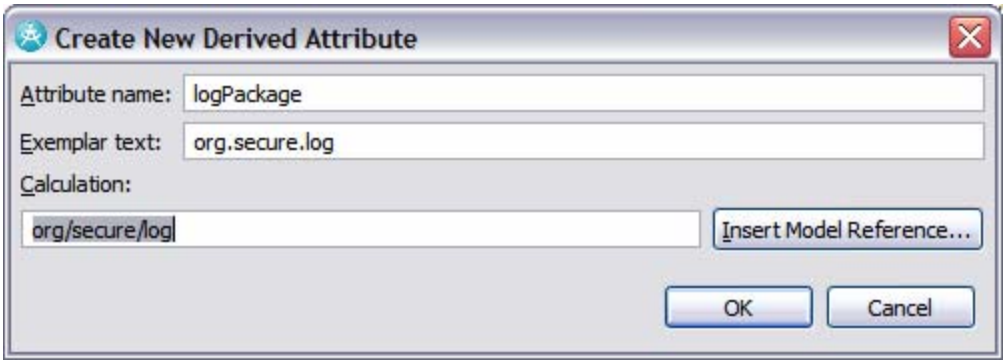


Figure 4.1 - 22: Selecting the text to be replaced in the Calculation field

23. Click **Insert Model Reference**.

24. Select **basePackage** and then click **OK**.

25. Add .log at the end of the Calculation field.

26. Click **OK**.

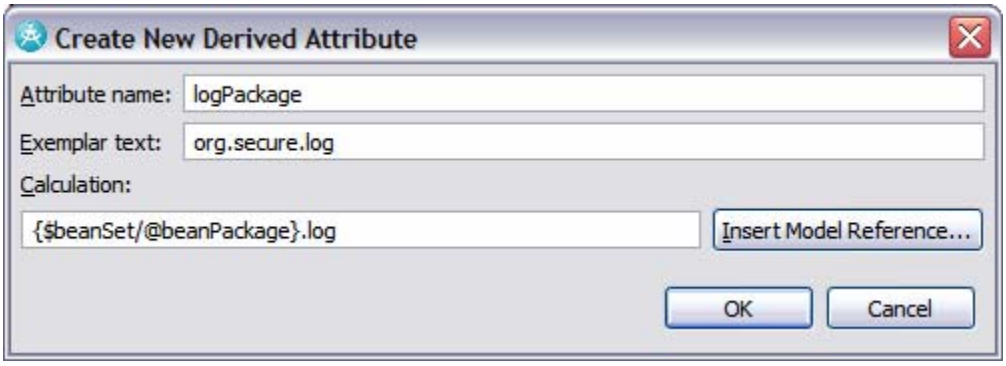


Figure 4.1 - 23: The details for the logPackage derived attribute

TIP: At this point you have the path value that is needed for the Logger class. However, you need to format the string so that it is an acceptable directory path. To do so, you replace the '.' character with a '/' character. As such, you'll continue to work in the Replace with Model References dialog and add another new derived attribute.

27. In the **Replace with Model References** dialog, select **beanSet** and then click **New**.

28. Specify `logDirectory` as the **Attribute name**.

29. Select the `org/secure/log` text in the **Calculation** field.

30. Click **Insert Model Reference**.

31. Select **logPackage** and then click **OK**.

32. Update the text in the Calculation field so that it appears as follows:

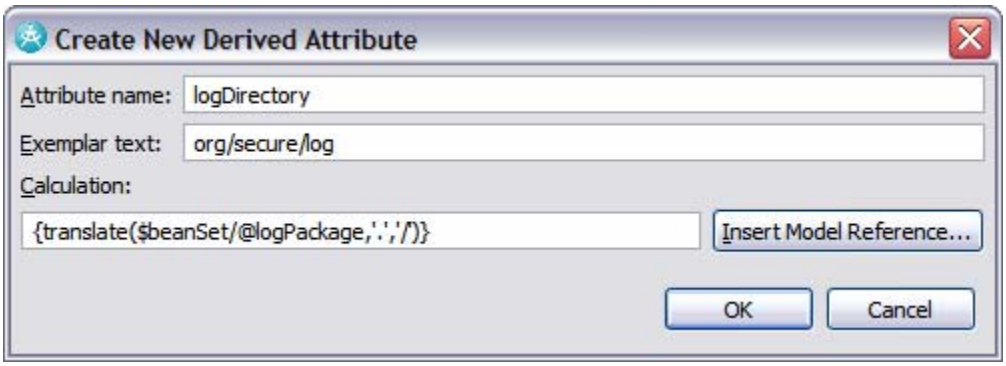


Figure 4.1 - 24: The details for the logDirectory derived attribute

33. Click **OK**.

34. Select **logDirectory** and then click **OK**.

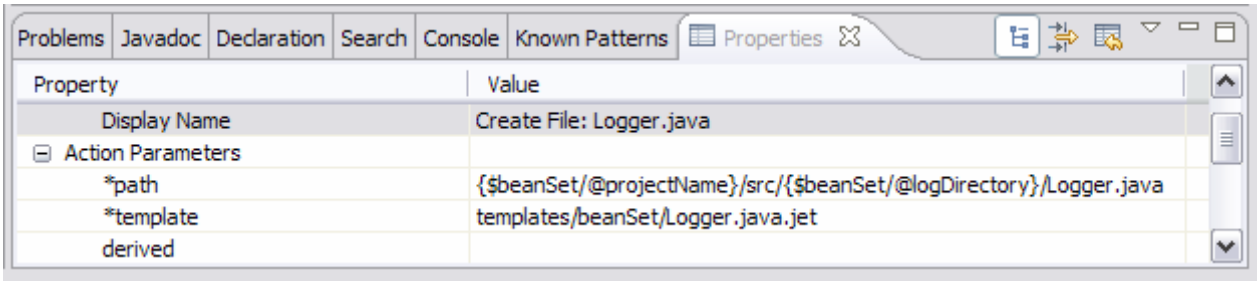


Figure 4.1 - 25: The updated path property for the Logger.java artifact

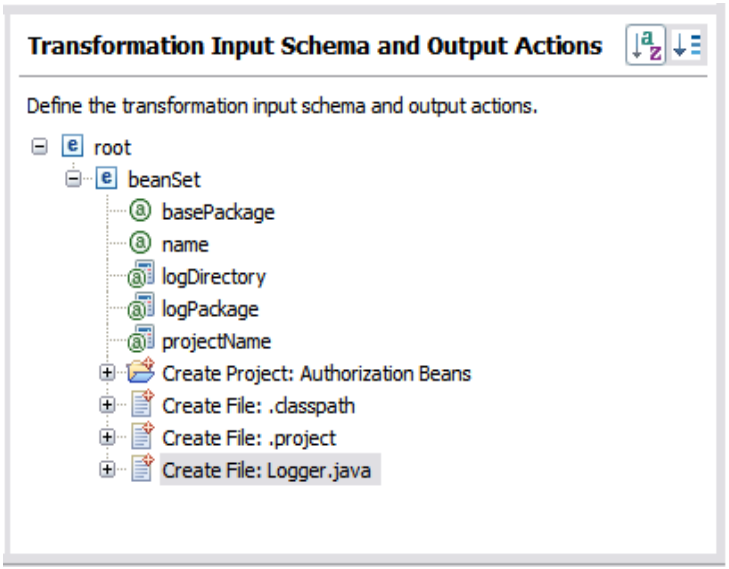


Figure 4.1 - 26: Updated input schema and output actions

35. Mark up the path parameter for .classpath and .project, too.

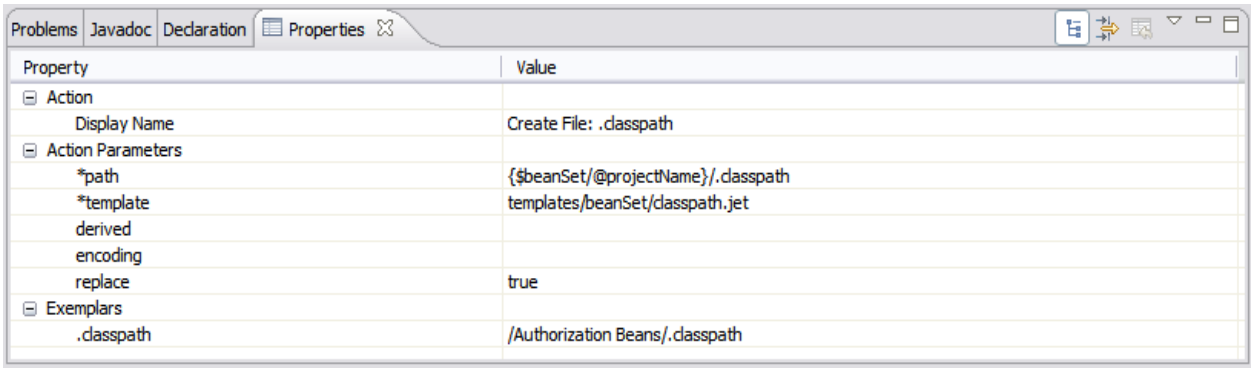


Figure 4.1 - 27: The updated path attribute for .classpath

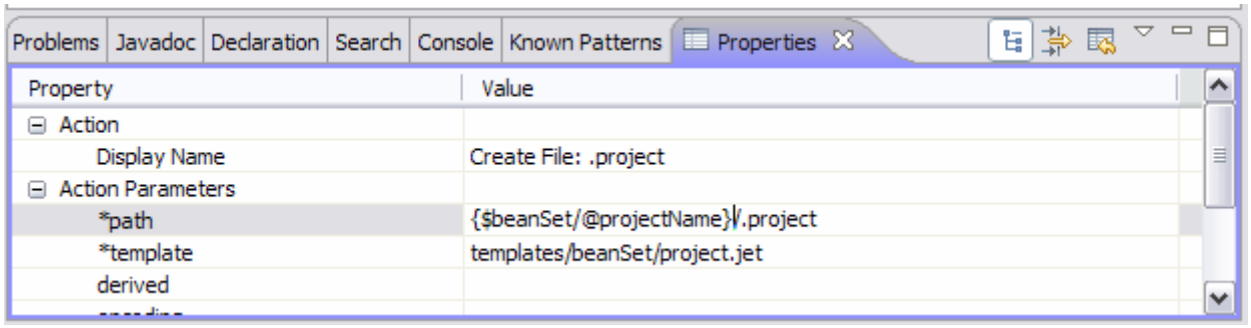


Figure 4.1 - 28: The updated path attribute for .project

TIP: Even though the .classpath and .project files have constant names, the name of the project containing them will change.

36. The singly-occurring artifacts have been modeled. Select **File > Save All**.

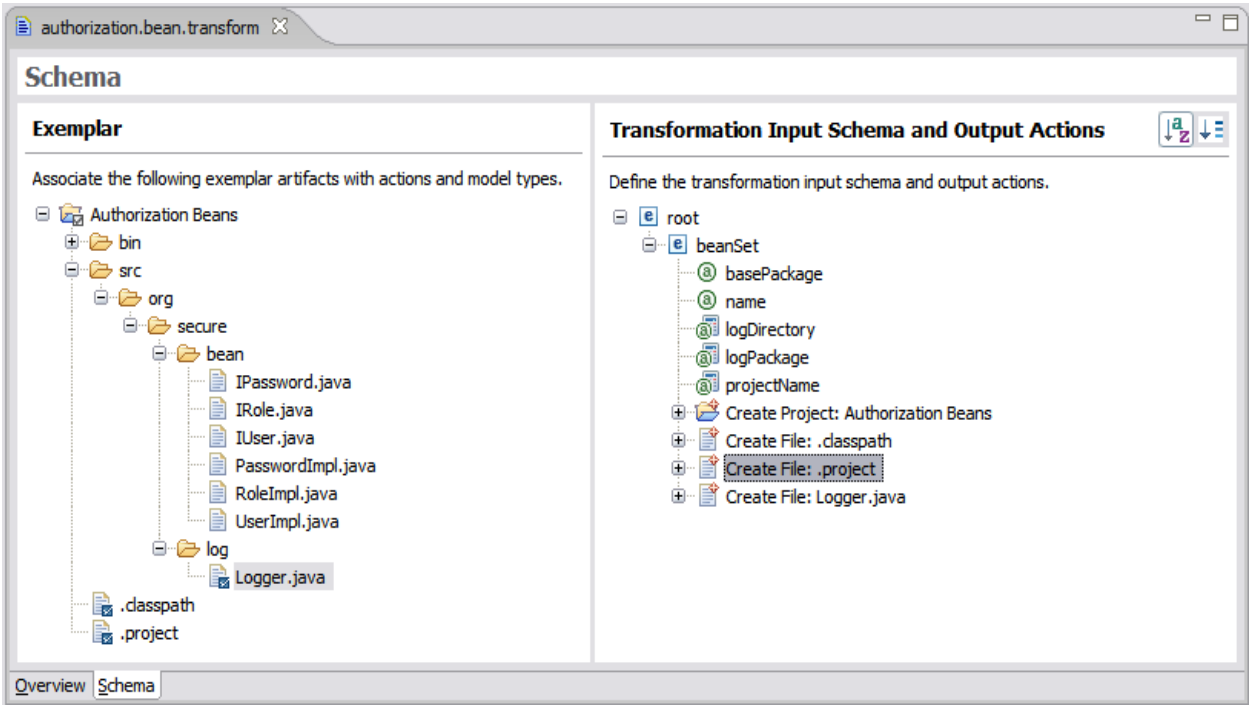


Figure 4.1 - 29: After modeling the schema and actions for the singly occurring artifacts

Task 4: Add Supporting Derived Attributes to beanSet

You know that all of the beans generated will be placed into one directory (and package). Earlier, when you created the directory and package attributes for the log class, you used the basePackage attribute as a starting point. You'll follow a similar approach here as you create derived attributes that support the beans. As such, you will add two new derived attributes called beanPackage and beanDirectory.

1. Right-click `beanSet` and select **New > Derived Attribute**.
2. Specify `beanPackage` as the **Attribute name**.
3. Specify `org.secure.bean` as the **Exemplar text**.
4. Click **Insert Model Reference**.
5. Select `basePackage` and then click **OK**.
6. Append `.bean` to the end of the text in the **Calculation** field.

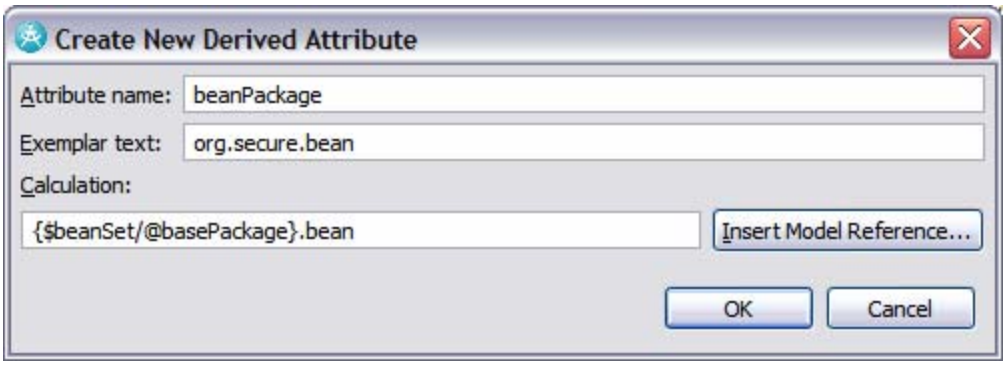


Figure 4.1 - 30: Creating the beanPackage derived attribute

7. Click **OK**.
8. Right-click `beanset` and click **New > Derived Attribute**.
9. Specify `beanDirectory` as the **Attribute name**.
10. Specify `org/secure/bean` as the **Exemplar text**.
11. Click **Insert Model Reference**.
12. Select BeanPackage and then click **OK**.
13. Update the text in the Calculation field so that it matches the following screen capture.

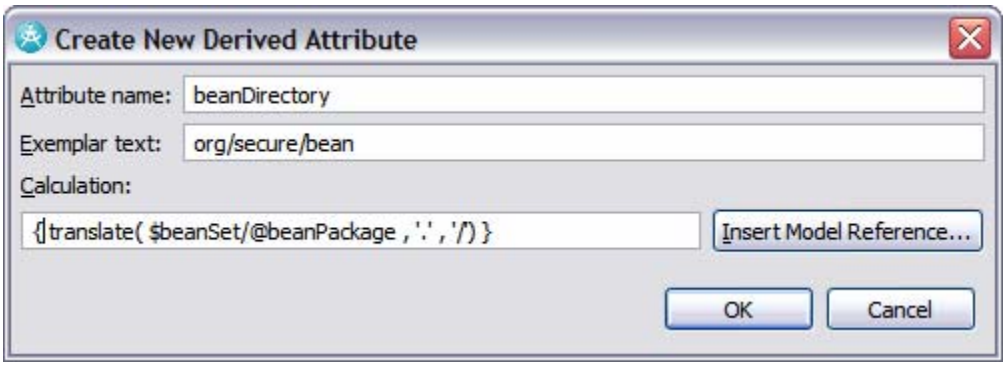


Figure 4.1 - 31: Creating the beanDirectory derived attribute

14. Click **OK**.

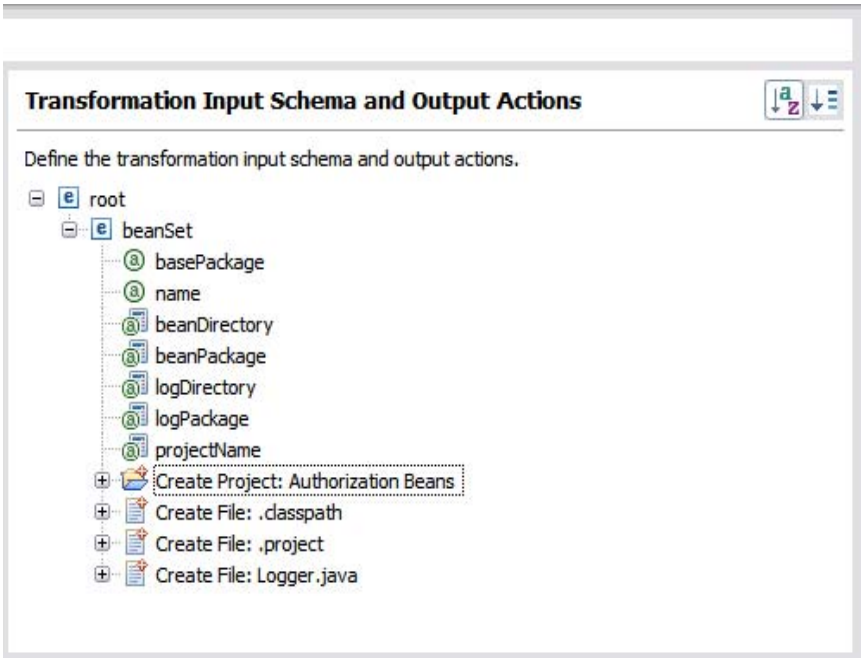


Figure 4.1 - 32: Updated view with the newly created derived attributes

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

Task 5: Create a New Type: bean

In this task, you will update the Schema with a new type called bean.

1. Create a new type (`bean`) under `beanSet`. Select `beanSet`, right-click and click **New > Type**. Enter `bean` as the name.

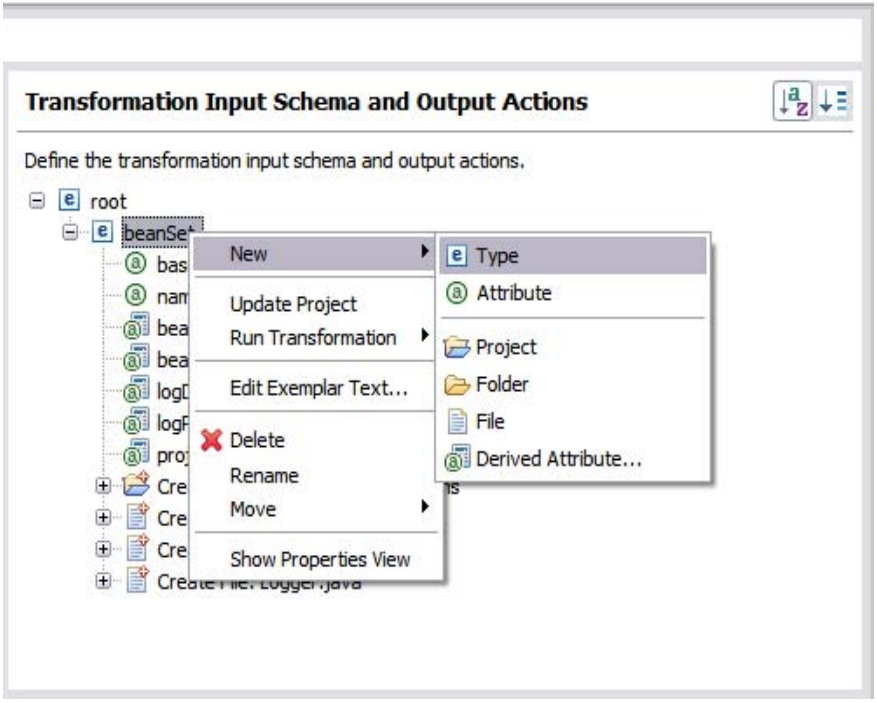


Figure 4.1 - 33: Artifacts added under console

2. Right-click `bean` and select **New > Attribute**. Enter `name` as the value for the attribute's Name.

Task 6: Add Supporting Derived Attributes to bean

In this task, you will create a set of derived attributes within the bean type to support the names associated with the implementation and interface classes. You add these to the bean type rather than the beanSet. This is because this attribute will need to be available for each bean created, whereas the earlier attributes are based on the beanSet.

1. Right-click on `bean` and select **New > Derived Attribute**.
2. Specify `interfaceName` as the **Attribute Name**.
3. Specify `IPassword` as the **Exemplar text**.
4. Click on **Insert Model Reference**.
5. Select `bean\name`. Click **OK**.
6. Update the **Calculation** field as shown in Figure 4.1 - 34.

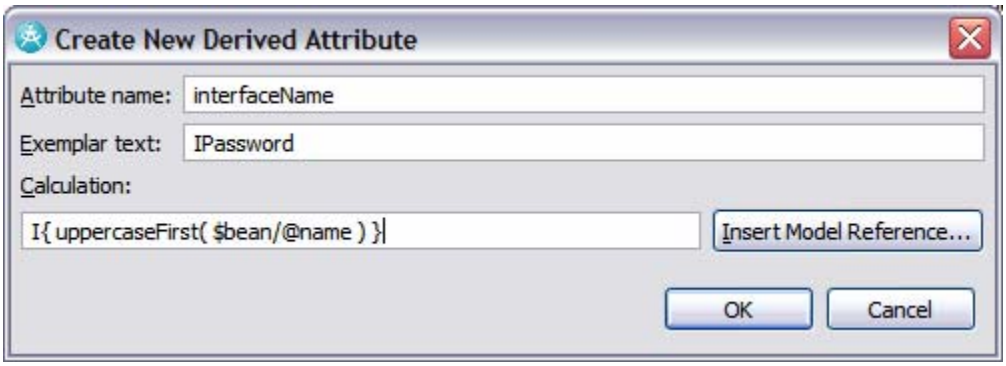


Figure 4.1 - 34: Creating the interfaceName derived attribute

7. Click **OK**.
8. Right-click `bean` and select **New > Derived Attribute**.
9. Specify implementationName as the **Attribute Name**.
10. Specify `PasswordImpl` as the **Exemplar text**.
11. Click on **Insert Model Reference**.
12. Select `bean\name`. Click **OK**.
13. Update the Calculation field as shown in Figure 4.1 - 35.

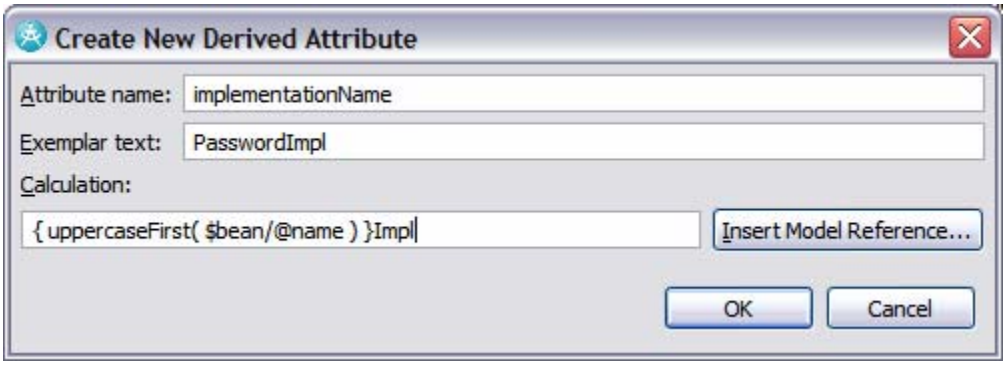


Figure 4.1 - 35: Creating the implementationName derived attribute

14. Click **OK**.

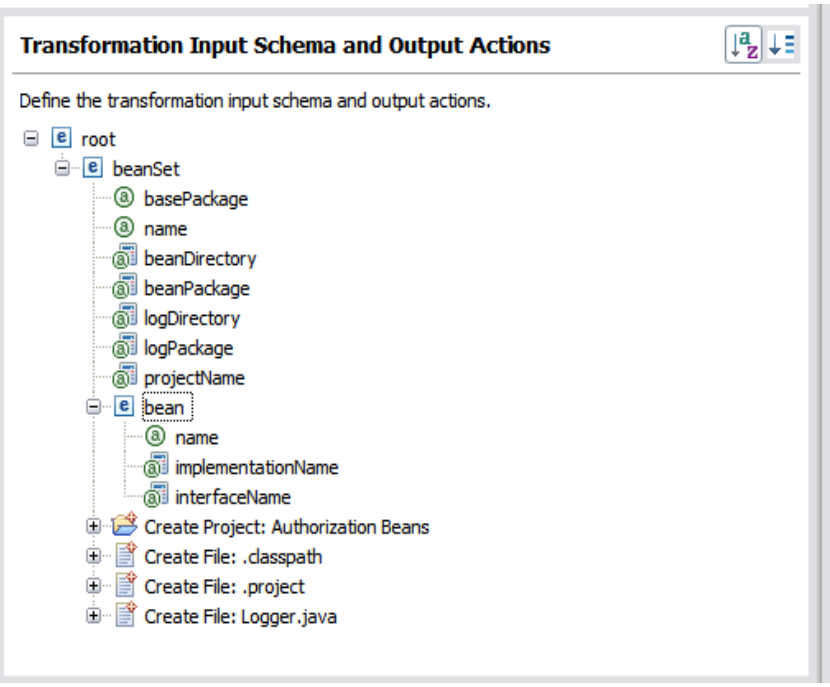


Figure 4.1 - 36: Updated view with the new derived attributes

Task 7: Populate the Model: Items Created Multiple Times

You still need to model the repeating sets of artifacts. Each repeating set of artifacts has a Java interface and a Java bean implementation (for example, IPassword.java and PasswordImpl.java)

1. Drag an example of each artifact in the repeating set, `IPassword.java` and `PasswordImpl.java`, onto the `bean` type to create two new actions.

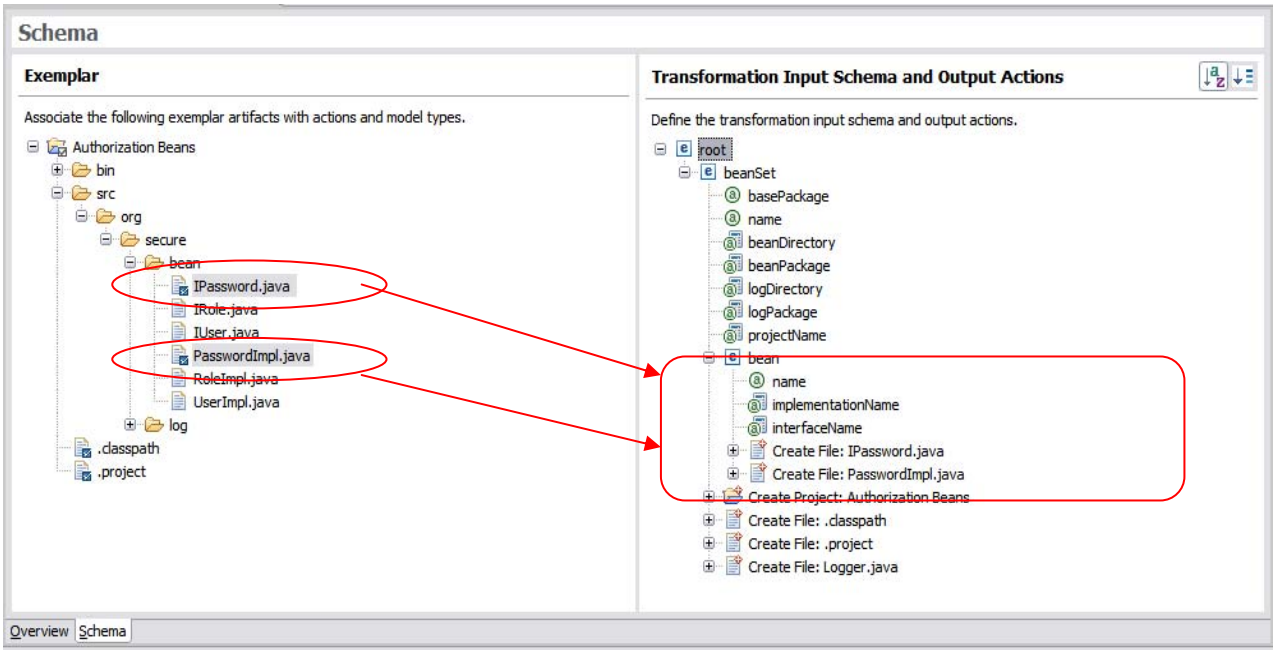


Figure 4.1 - 37: Artifacts added under console

TIP: You'll want to allow the user to provide a name for the set of beans. With the exemplar, you can see that the bean was called "Password", and then customized based on whether it was the interface or the implementation.

TIP: As you did earlier with the Logger.java artifact, you need to update the path value for the implementation and interface artifacts.

2. Select the **Create File: IPassword.java** action.
3. In the **path** field within the **Properties** view, select `Authorization Beans`, right-click and click **Replace with Model Reference**.

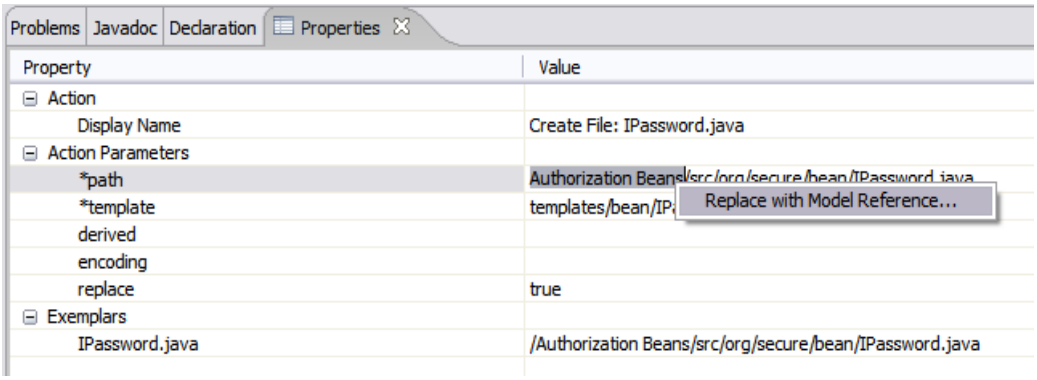


Figure 4.1 - 38: Selecting the text from the path property that needs to be replaced with a model reference

4. Select **projectName** and then click **OK**.

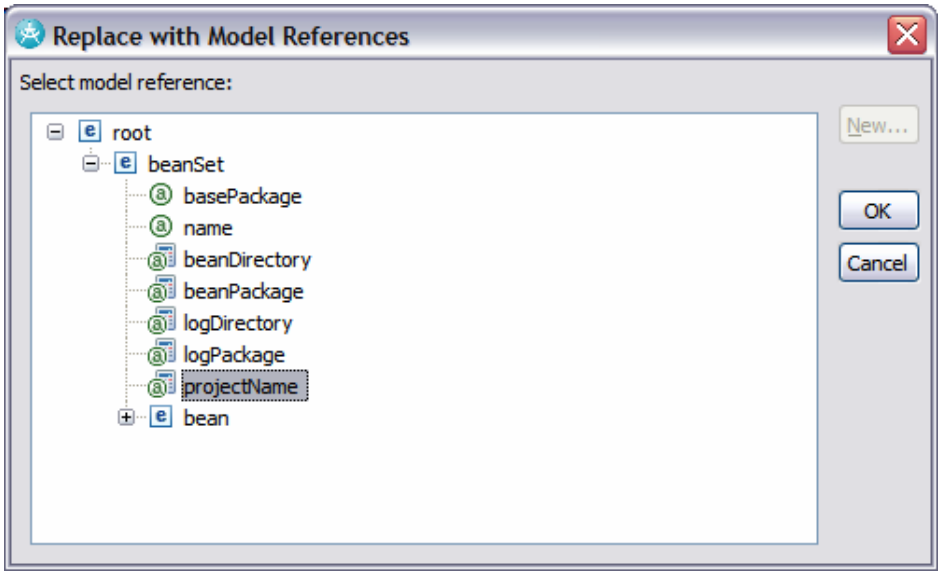


Figure 4.1 - 39: Selecting the projectName derived attribute

5. In the **path** field within the **Properties** view, select `org/secure/bean`, right-click and click **Replace with Model Reference**.
6. Select `beanDirectory` and then click **OK**.

7. In the **path** field within the **Properties** view, select `IPassword`, right-click and click **Replace with Model Reference**.
8. Select bean\interfaceName and then click **OK**.

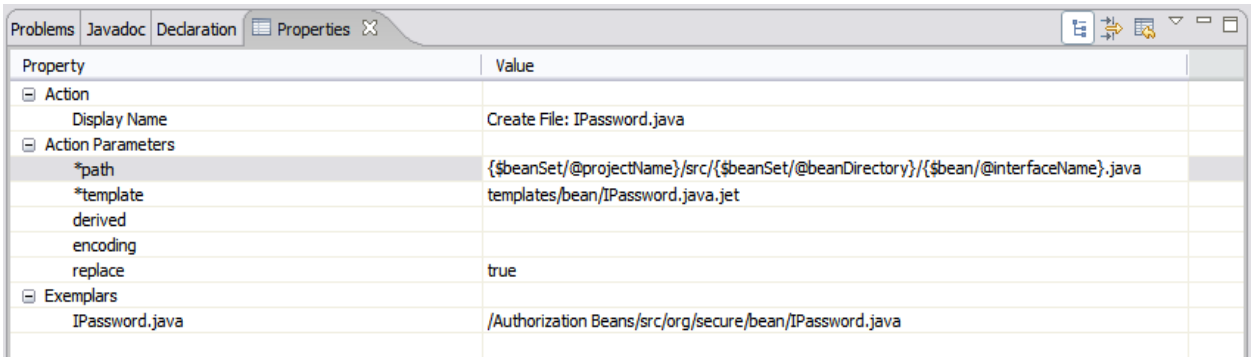


Figure 4.1 - 40: The completed path entry with references to the appropriate attributes

9. Select the **Create File: PasswordImpl.java** action.
10. In the **path** field within the **Properties** view, select `Authorization Beans`, right-click and click **Replace with Model Reference**.

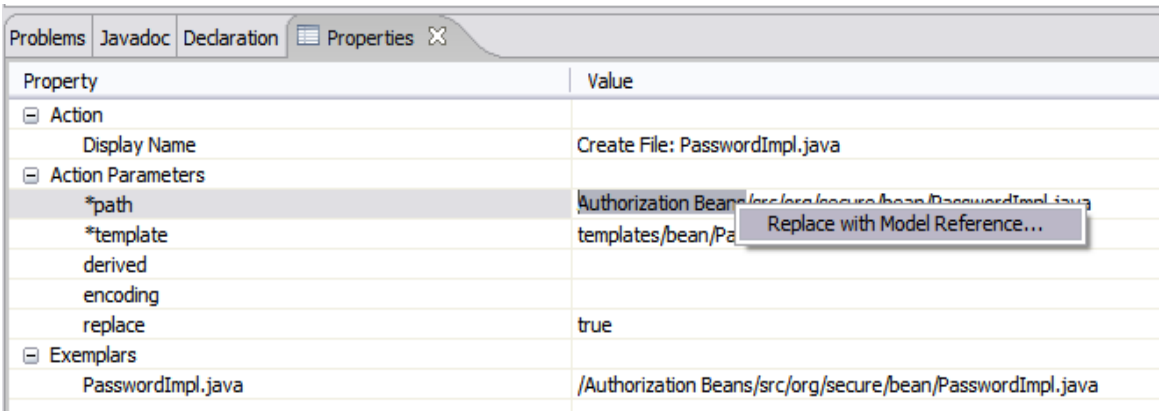


Figure 4.1 - 41: Selecting the text from the path property that needs to be replaced with a model reference

11. Select **projectName** and then click **OK**.
12. In the **path** field within the **Properties** view, select `org/secure/bean`, right-click and click **Replace with Model Reference**.
13. Select `beanDirectory` and then click **OK**.
14. In the **path** field within the **Properties** view, select `PasswordImpl`, right-click and click **Replace with Model Reference**.
15. Select `bean\implementationName` and then click **OK**.

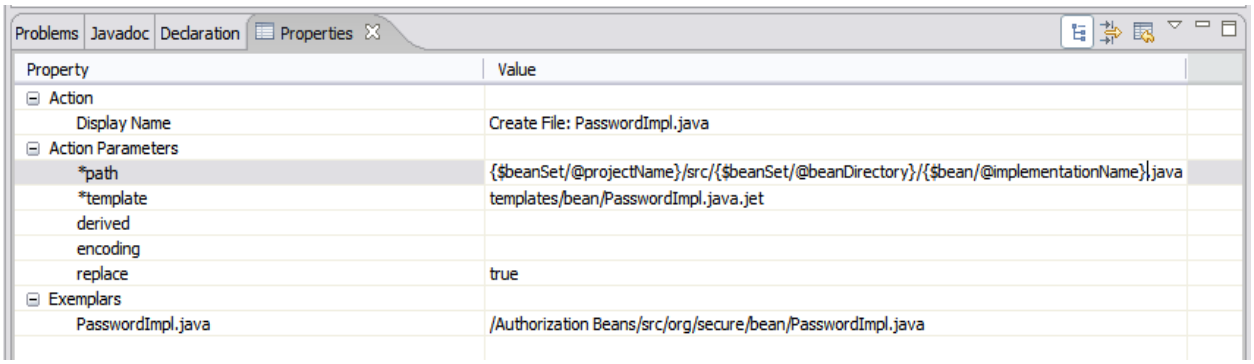


Figure 4.1 - 42: The completed path entry with references to the appropriate attributes

You've modeled the repeating set of artifacts and have defined: one attribute, two derived attributes, and two transform actions.

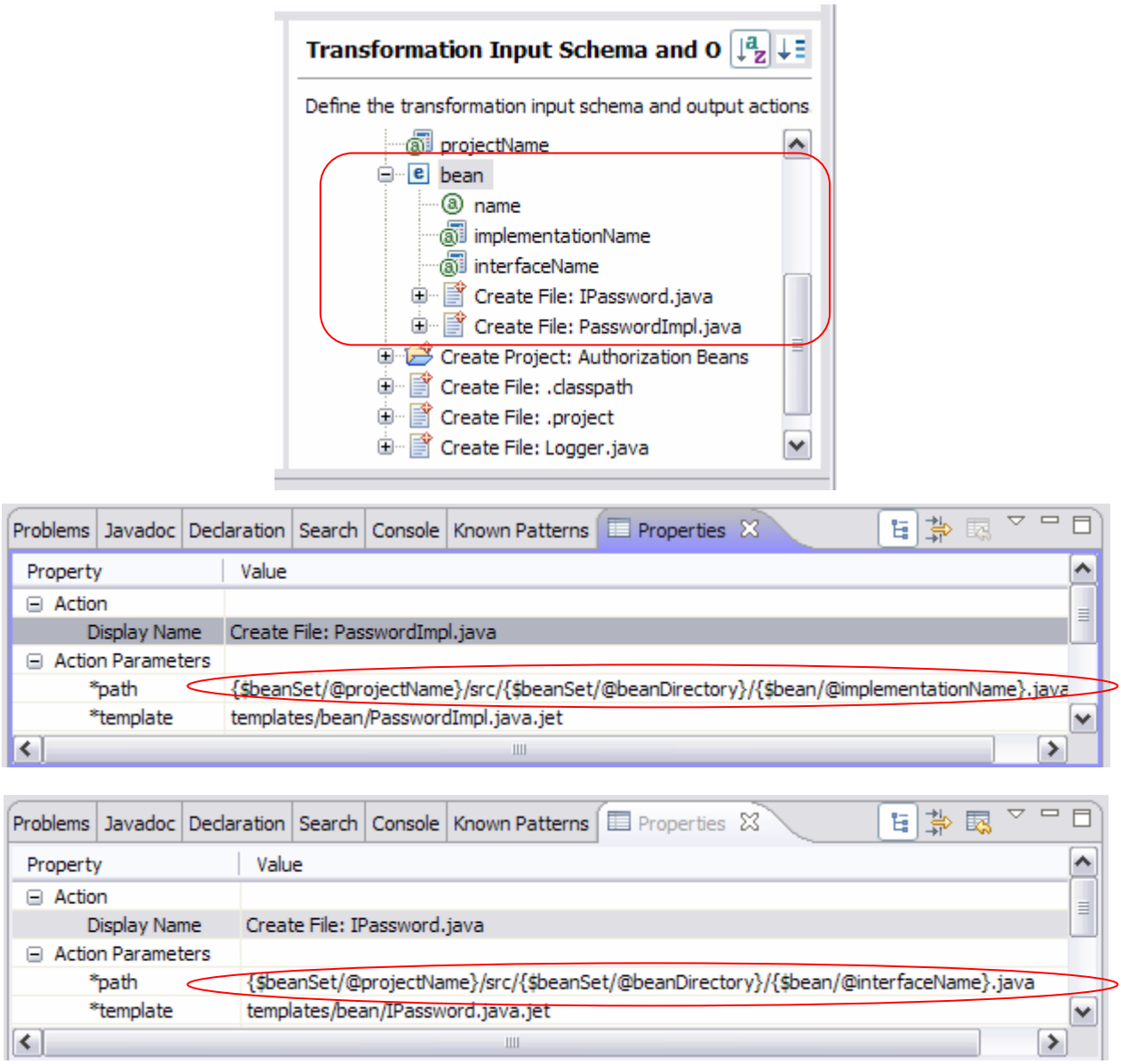


Figure 4.1 - 43: Artifacts added under console

The bean model object contains all the information required to generate the interface and implementation

Task 8: Modeling Additional Information Needed

In this task, you'll model additional information that is needed to address the points of variability within the code files.

TIP: But there's more information needed than what you have already modeled: Property name, Property type, Getter name, Setter name, Variable name from within implementation file. Also, there is one set of these names for each property, and there are multiple properties for each bean.

*authorization.bean.transform *PasswordImpl.java

```
package org.secure.bean;

import org.secure.log.Logger;

public class PasswordImpl implements IPassword {

    private String      field_value;
    private boolean     field_expired;
    private int         field_length;


    public String getValue() {
        return field_value;
    }

    public void setValue(String value) {
        Logger.log("Property value changed",Logger.SEVERITY_INFO);
        this.field_value = value;
    }

    public boolean isExpired() {
        return field_expired;
    }
```

Figure 4.1 - 44: Additional points of variability within the implementation class

2. Add a new type within `bean` called `property`, to represent a set of repeating property information. Add attributes to capture the `name` and `type` for the `property`.

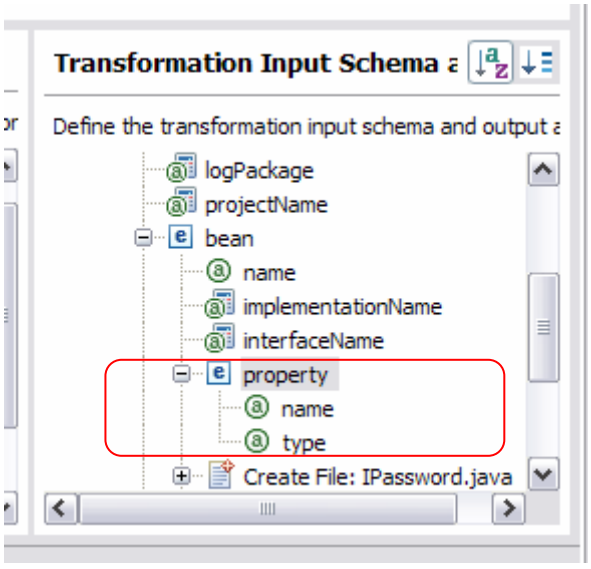


Figure 4.1 - 45: A new type of property with attributes of name and type.

3. Add a new derived attribute for the getterName. Right-click `property` and click **New > Derived Attribute**.

4. Specify a name of `getterName`.
5. Specify exemplar text of `getValue`.
6. Click on **Insert Model Reference**.
7. Select beanset > bean > property > name and then click **OK**.
8. Update the **Calculation** field so that it matches the screen capture below, and then click **OK**.

Figure 4.1 - 46: A new derived attribute for the getter name.

9. Add a new derived attribute for the Boolean getterName. Right-click `property` and select **New > Derived Attribute**.
10. Specify a name of booleanGetterName.
11. Specify exemplar text of `isExpired`.
12. Click **Insert Model Reference**.
13. Select **beanset > bean > property > name** and then click **OK**.
14. Update the **Calculation** field so that it matches the screen capture below, and then click **OK**.

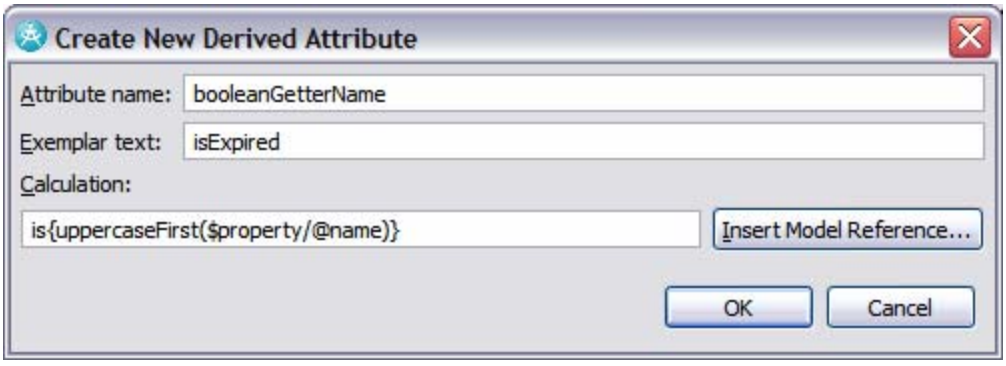


Figure 4.1 - 47: A derived attribute for the Boolean getter name.

15. Add a new derived attribute for the setterName. Right-click `property` and click **New > Derived Attribute**.
16. Specify a name of `setterName`.
17. Specify exemplar text of `setValue`.
18. Click on **Insert Model Reference**.
19. Select **beanset > bean > property > name** and then click **OK**.
20. Update the **Calculation** field so that it matches the screen capture below and then click **OK**.

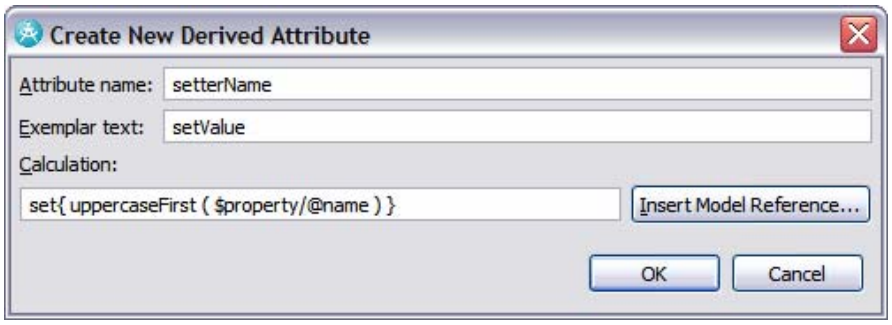


Figure 4.1 - 48: A new derived attribute for the setter name.

21. Add a new derived attribute for the varName. Right-click `property` and click **New > Derived Attribute**.

22. Specify a name of `varName`.

23. Specify exemplar text of `field_value`.

24. Click **Insert Model Reference**.

25. Select **beanset > bean > property > name** and then click **OK**.

26. Update the **Calculation** field so that it matches the screen capture below and then click **OK**.

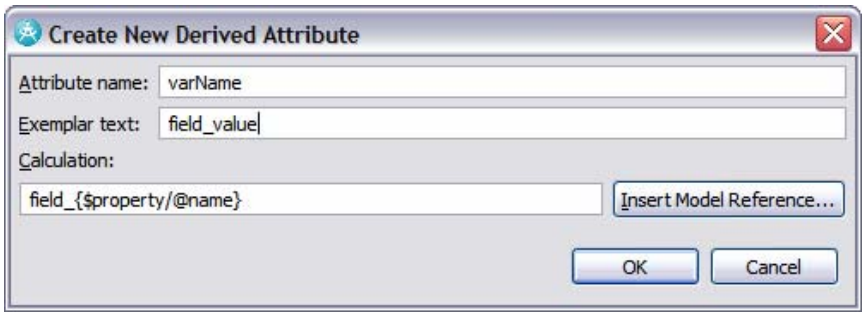


Figure 4.1 - 49: A new derived attribute for the variable name.

27. Select **File > Save All**.

Task 9: Create the Transform's Templates

In this task, you will generate the templates for the transform.

1. To create the transform's templates, right-click in the right-hand side of the Schema editor and click **Update Project**.

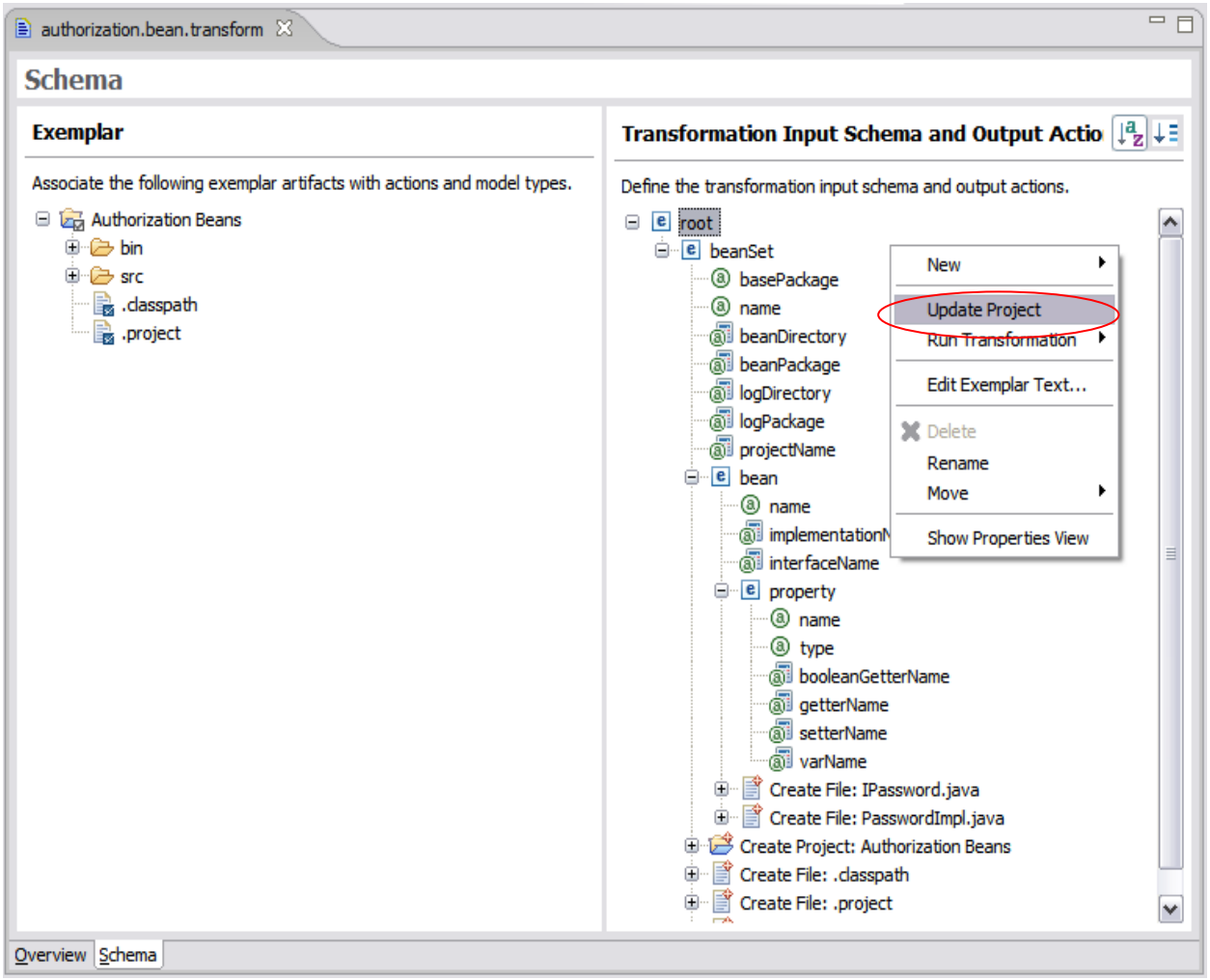


Figure 4.1 - 50: Artifacts added under console

2. **Update Project** will create a template folder for each type with create file actions, and will create a template in that folder for each of those actions.

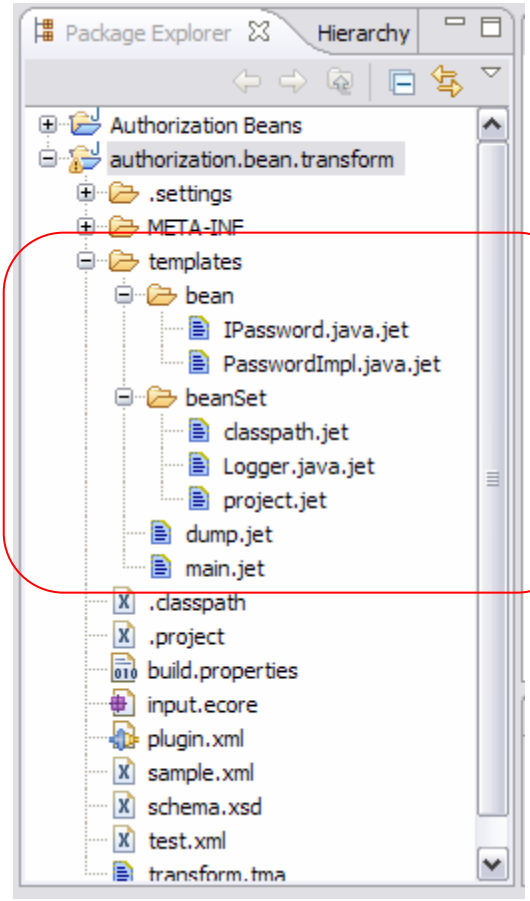


Figure 4.1 - 51: Artifacts added under console

Task 10: Edit the Transform's Templates: project.jet

In this task, you will set up the environment to allow you to test the transform.

1. Open the sample.xml file for editing.
2. Replace the contents of the file with the following:

```
<root>
    <beanSet basePackage="com.dev498.test" name="TestBeans">
          <bean name="Curly">
                <property name="age" type="String" />
                <property name="funny" type="Boolean" />
          </bean>
    </beanSet>
</root>
```

3. Select **File > Save All**.
4. In the Package Explorer, right-click the `sample.xml` file and click **Run As > Input for JET Transformation**.
5. Open the `dump.xml` file.
6. Review the contents of the `dump.xml` file.

TIP: As you work through the following tasks and complete the updates to the generated templates, remember that you can test them quickly and easily as you proceed. In addition, when combined with the output from the dump.xml file, you can get an understanding of the way that the input data is being interpreted.

Task 11: Edit the Transform's Templates: project.jet

In this task, you will edit the templates associated with the transform.

1. Open the `project.jet` template.

TIP: The string "Authorization Beans" is known to be associated with a model attribute, so the editor highlights the string.

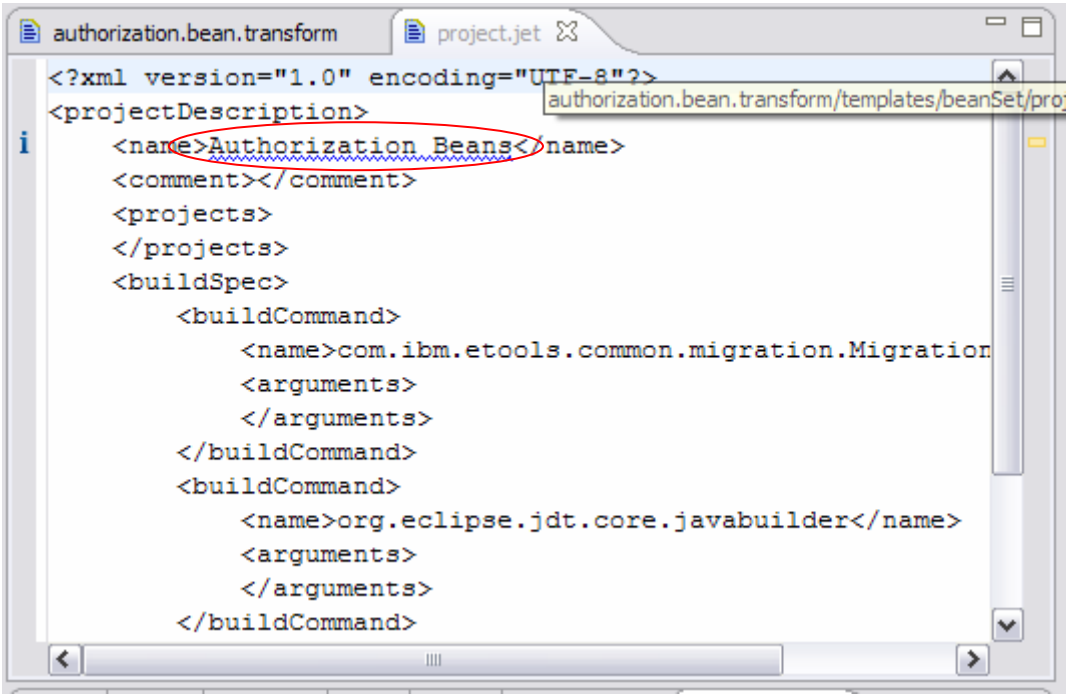


Figure 4.1 - 52: Artifacts added under console

2. Select the Authorization Beans string, right-click and click **Find/Replace with JET Model Reference.**

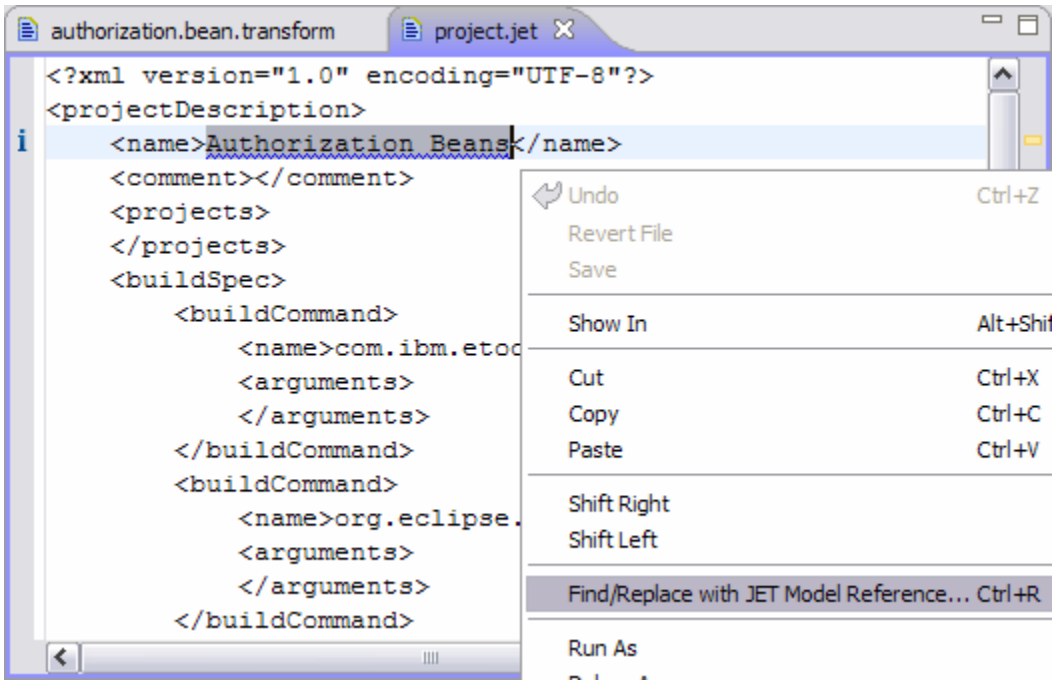


Figure 4.1 - 53: Artifacts added under console

3. Select the `projectName` attribute and click **Replace**, then click **Close**.

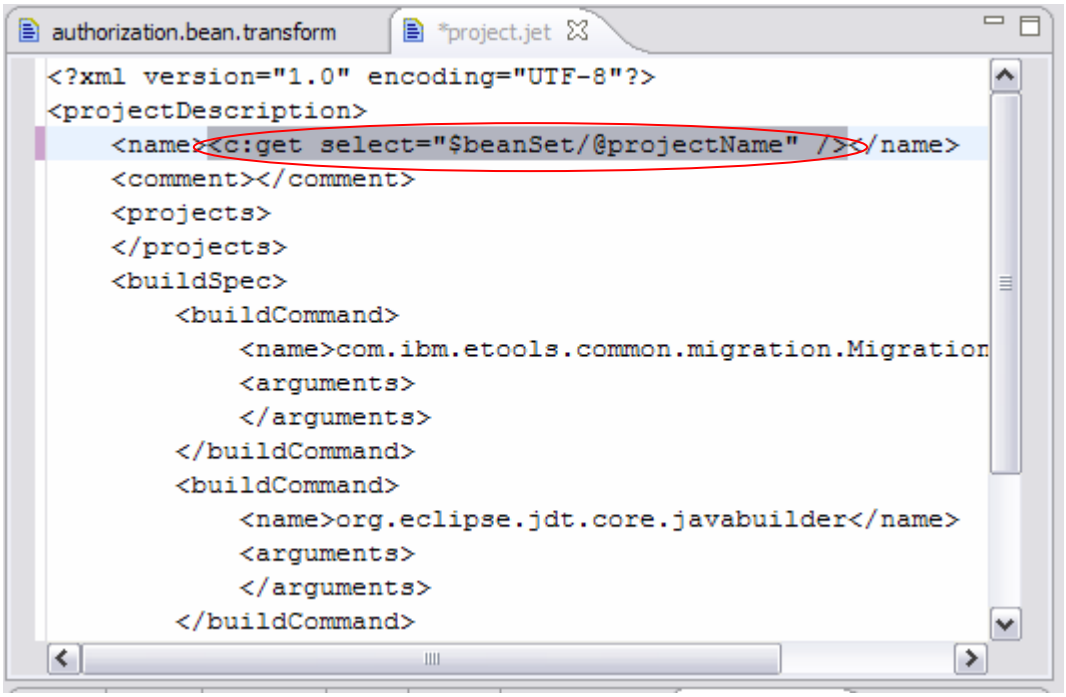


Figure 4.1 - 54: And the correct JET tag replaces the string.

4. Save and then close the `project.jet` template.

Task 12: Edit the Transform's Templates: Logger.java.jet

In this task you'll update the template to generate the `Logger.java` file.

1. Open the Logger.java.jet template for editing.
2. Select the text org.secure.log and then right-click and click **Find/Replace with JET Model Reference**.
3. Select `logPackage`, click **Replace** and then click **Close**.

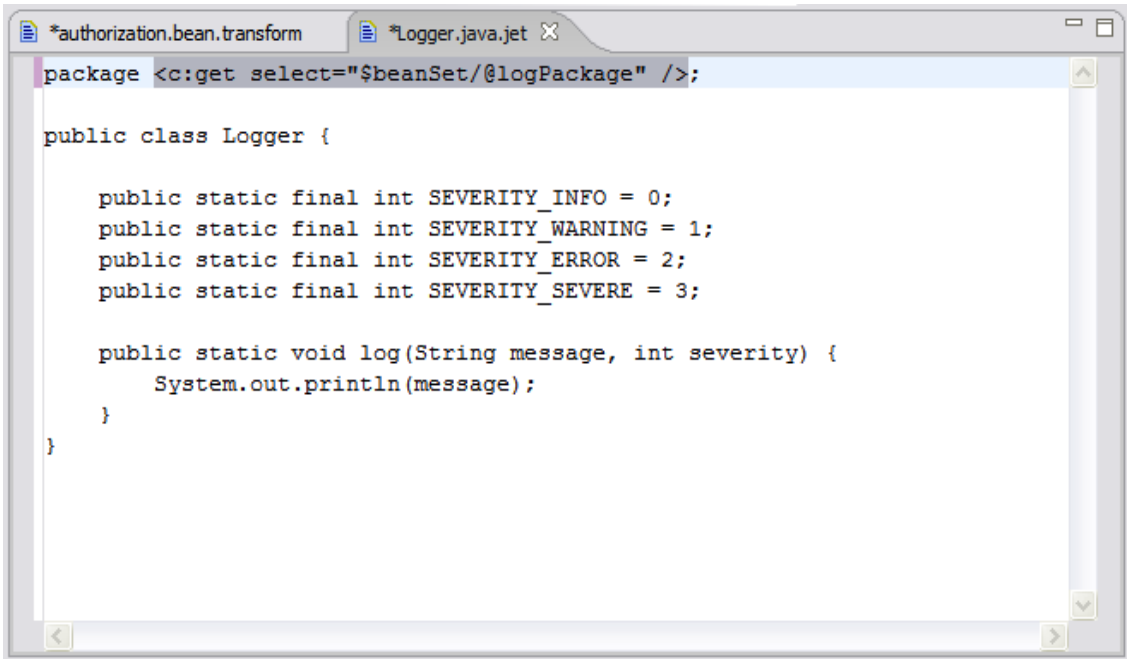


Figure 4.1 - 55: The updated Logger.java.jet template.

4. Save and close the `Logger.java.jet` template.

Task 13: Edit the Transform's Templates: IPassword.java.jet

In this task you'll update the IPassword.java.jet template that is used to generate the I<beanName>.java file.

1. Open the `IPassword.java.jet` template.
2. Select the text `org.secure.bean` and then right-click and click **Find/Replace with JET Model Reference**.
3. Select `beanPackage`, click **Replace**, and then click **Close**.
4. Select the text `IPassword` and then right-click and click **Find/Replace with JET Model Reference**.
5. Select `interfaceName`, click **Replace** and then click **Close**.

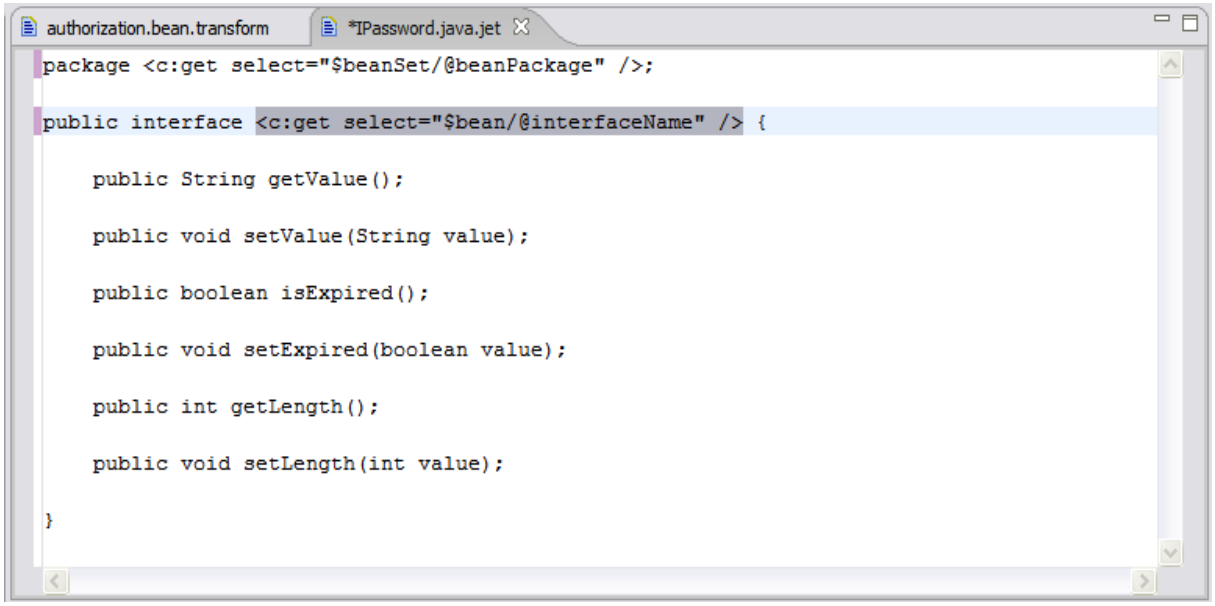


Figure 4.1 - 56: The updated Logger.java.jet template.

TIP: At this point you now have to parameterize the template to handle the set of properties that are associated with the bean. For each property you need to create a getter and setter, with the methods using the appropriate types. In addition, in the case of Boolean parameters, you need to change the name of the getter method to "is".

6. First, add in some code for the setter and getter methods. You need to iterate through the set of properties. Remove the current method declarations and add the following text to the file:

```
<c:iterate select="$bean/property" var="property">


      public <c:get select="$property/@type" /> <c:get
select="$property/@getterName" />();


      public void <c:get select="$property/@setterName" />(<c:get
select="$property/@type" /> value);
</c:iterate>
```

TIP: If you copy and paste the code, note that the editor will not like the " (curly quotation marks) character as supplied by Microsoft Word. If you get an error on the line, replace the " character with one typed in place within the editor.

7. At this point, however, the code does not account for the case where the type is Boolean. You need to add in some additional code to determine if the type is Boolean, and if so, to use the booleanGetterName in place of the getterName. Replace the following code:

```
public <c:get select="$property/@type" /> <c:get select="$property/@getterName"
/>();
```

with:

```
<c:choose select=" $property/@type" >
```

```
        <c:when  test="  'Boolean' " >
   public <c:get select="$property/@type" /> <c:get
select="$property/@booleanGetterName"/>();
        </c:when>
        <c:otherwise>
   public <c:get select="$property/@type" /> <c:get select="$property/@getterName"
/>();
        </c:otherwise>
    </c:choose>
```

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

Task 14: Edit the Transform's Templates: PasswordImpl.java.jet

In this task you'll update the code in the PasswordImpl.java.jet file that is used to generate the `<beanName>Impl.java` file.

1. Open the `PasswordImpl.java.jet` template.
2. Select the text `org.secure.bean` and then right-click and click **Find/Replace with JET Model Reference**.
3. Select `beanPackage`, click **Replace** and then click **Close**.
4. Select the text `org.secure.log` and then right-click and select **Find/Replace with JET Model Reference**.
5. Select `logPackage`, click **Replace** and then click **Close**.
6. Select the text `PasswordImpl` and then right-click and select **Find/Replace with JET Model Reference**.
7. Select `implementationName`, click **Replace** and then click **Close**.
8. Select the text `IPassword` and then right-click and select **Find/Replace with JET Model Reference**.
9. Select `interfaceName`, click **Replace** and then click **Close**.

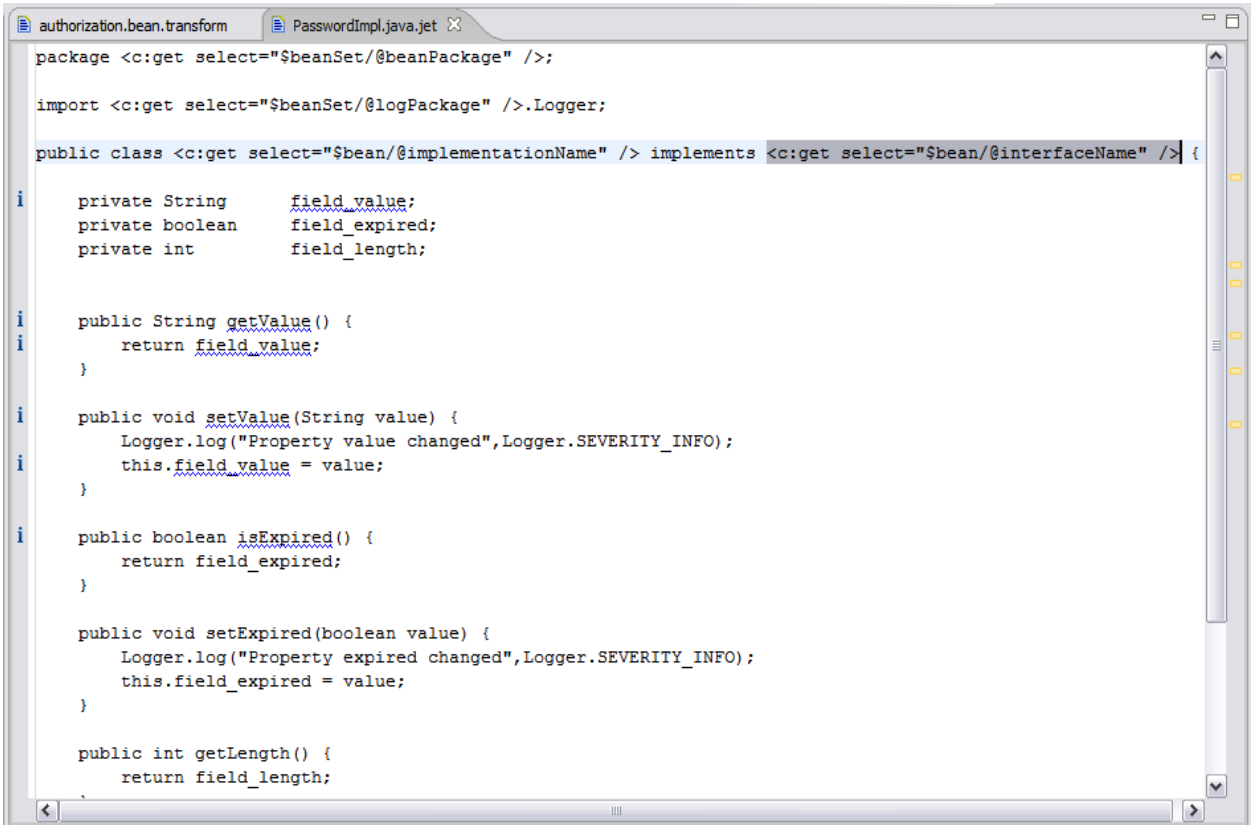


Figure 4.1 - 57: Template updated with package, import, class name and implements reference.

10. Add an iterate statement for the creation of the variable declarations. Replace the current variable declarations with the following text:

```
<c:iterate select="$bean/property" var="property">
    private <c:get select="$property/@type" />         <c:get
select="$property/@varName" />;
</c:iterate>
```

11. Now, you just need to add the code for creating the methods. Replace the current method bodies, with the following text:

```
 <c:iterate select="$bean/property" var="property">
    <c:choose select=" $property/@type" >
         <c:when  test="'Boolean'" >
    public <c:get select="$property/@type" /> <c:get
select="$property/@booleanGetterName"/>(){
         </c:when>
         <c:otherwise>
    public <c:get select="$property/@type" /> <c:get select="$property/@getterName"
/>(){
         </c:otherwise>
     </c:choose>
           return <c:get select="$property/@varName" />;
    }
```

```
    public void <c:get select="$property/@setterName" />(<c:get
select="$property/@type" /> value) {
            Logger.log("Property <c:get select="$property/@name" />
changed",Logger.SEVERITY_INFO);
            this.<c:get select="$property/@varName" /> = value;
    }
</c:iterate>
```

12. Select **File > Save All**.

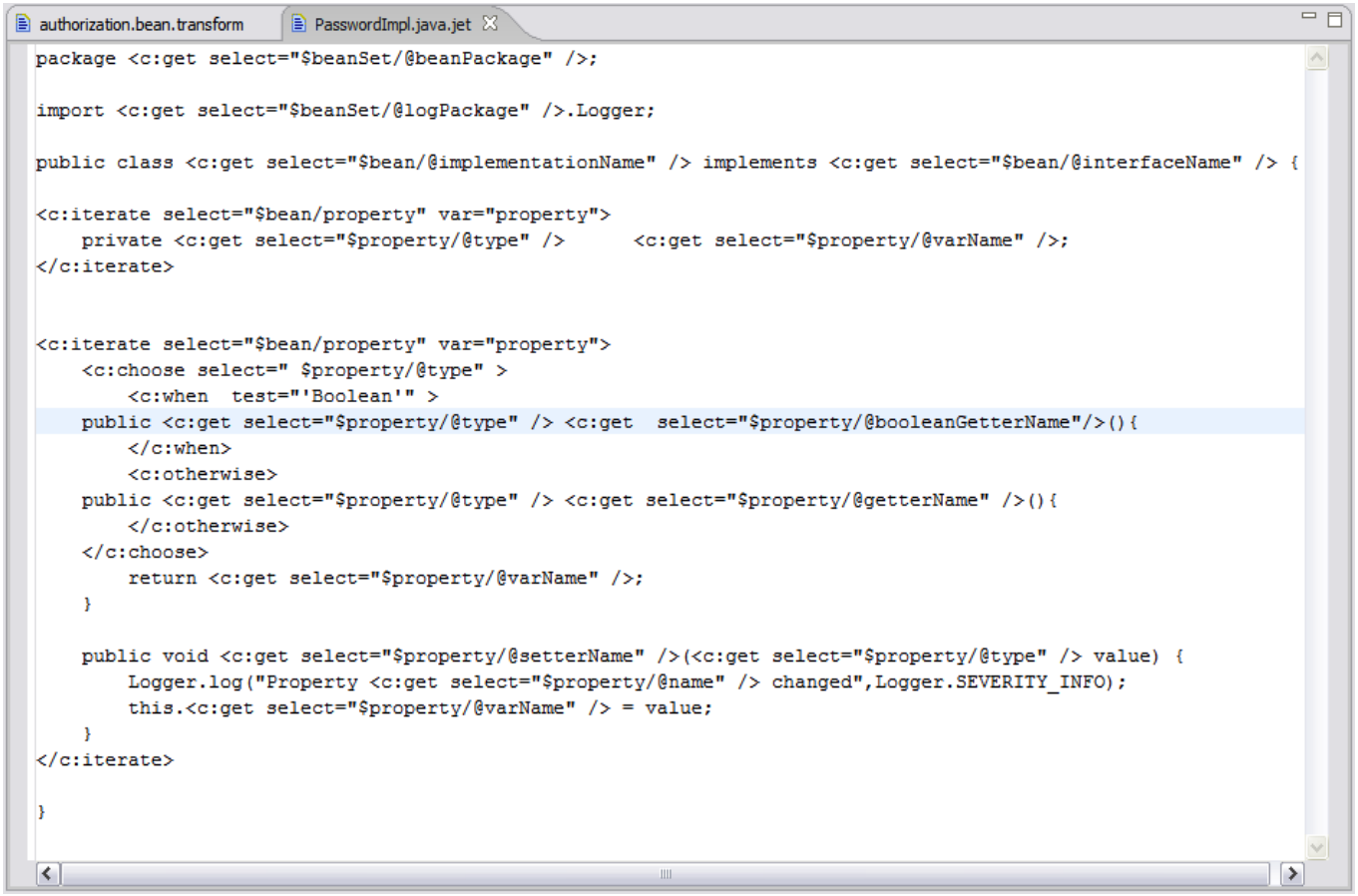


Figure 4.1 - 58: The completed PasswordImpl.java.jet template

13. You have now completed all of the customization needed for the transform. Test and review.

Lab 4.2: Exemplar Authoring

Objectives

After completing this lab, you will be able to:

- Perform exemplar analysis

Given

- The project interchange file, `ExemplarAnalysis.zip`

Scenario

In this lab you will perform Exemplar Analysis on an exemplar based on a set of feature projects and an Eclipse update site.

Task 1: Set up the Lab

1. Use the Import from Project Interchange wizard to import all of the projects in the `ExemplarAnalysis.zip` file.
2. Look at the project that was imported. This project contains the exemplars.
   - The exemplar stretches across 12 projects. The transform to be authored from these projects will generate a number of feature projects, and a single update site project.
3. Update sites are the usual way that Eclipse tools are distributed. The tools exist in one or more plug-in projects. The tool builder has to create a number of Eclipse feature projects for the plug-in projects, and must also create an update site for the feature projects.

Each feature project has three files:

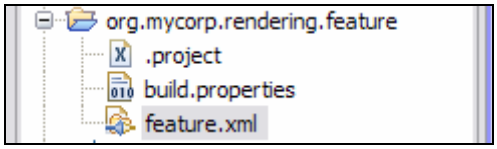


Figure 4.2 - 1: Feature Project org.mycorp.rendering.feature

The only file you may not have seen yet is the `feature.xml` file (which always has that same name). The file is a simple XML file that, among other things, lists the plug-ins to be contained in this feature:

```
<plugin
      id="org.mycorp.rendering"
      download-size="0"
      install-size="0"
      version="1.1.7"
      unpack="false"/>

<plugin
      id="org.mycorp.extra"
      download-size="0"
      install-size="0"
      version="1.0.9"
      unpack="false"/>
```

Figure 4.2 - 2: Plug-ins in the Feature org.mycorp.rendering.feature

4. The update site is also a simple project:

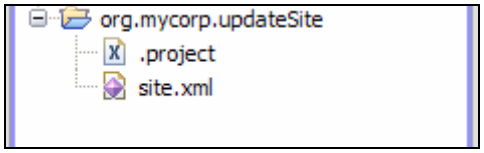


Figure 4.2 - 3: Update Site project org.mycorp.updateSite

The only two files you need to generate are the .project file and the `site.xml` file. The `site.xml` file lists a number of categories and the features that go into those categories.

A number of dummy plug-in projects are also included. Your transform will not generate those. It will assume they already exist. They are included here to avoid validation errors on the feature plug-ins. Your exemplar consists of the projects whose names end in `.feature` or `.updateSite`.

Lab 5: The Console Transform

Objectives

After completing this lab, you will be able to:

- Perform exemplar analysis on a Java application

Given

- The project interchange file `TheConsoleTransform.zip`

Scenario

In this lab, you will perform Exemplar Analysis on a working Java application.

Task 1: Set up the Lab

1. Begin by using the Import from Project Interchange wizard to import all of the projects in the TheConsoleTransform.zip file.
2. Look at the project that was imported.

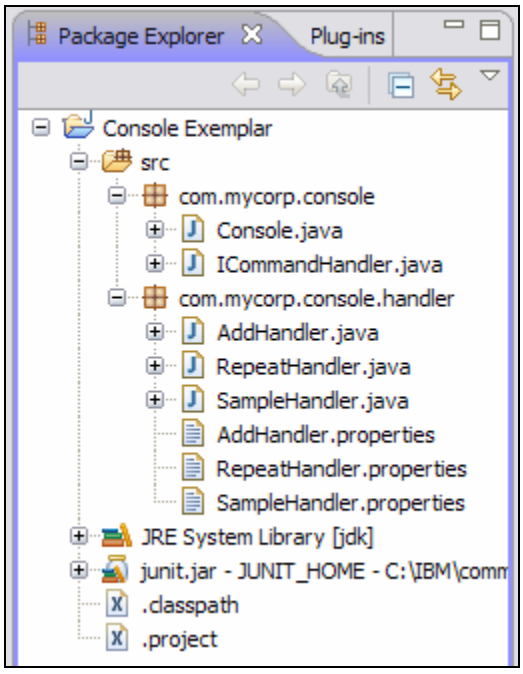


Figure 5 - 1: Console Exemplar Project

This Java application is a working command line console. It supports three commands: Add, Sample, and Repeat. Each of those commands is implemented by a Handler class, which in turn implements the `ICommandHandler` interface. Each handler also has its own properties file to hold translatable strings.

The `Console` class is the main class in the application. It listens to input on its `System.in` stream. For each entered command, `Console` will try to match the command (the first token of the input string) to the command handled by each of the handlers. If a handler matching the command is found, then that handler is passed the full command and

is expected to process that command.

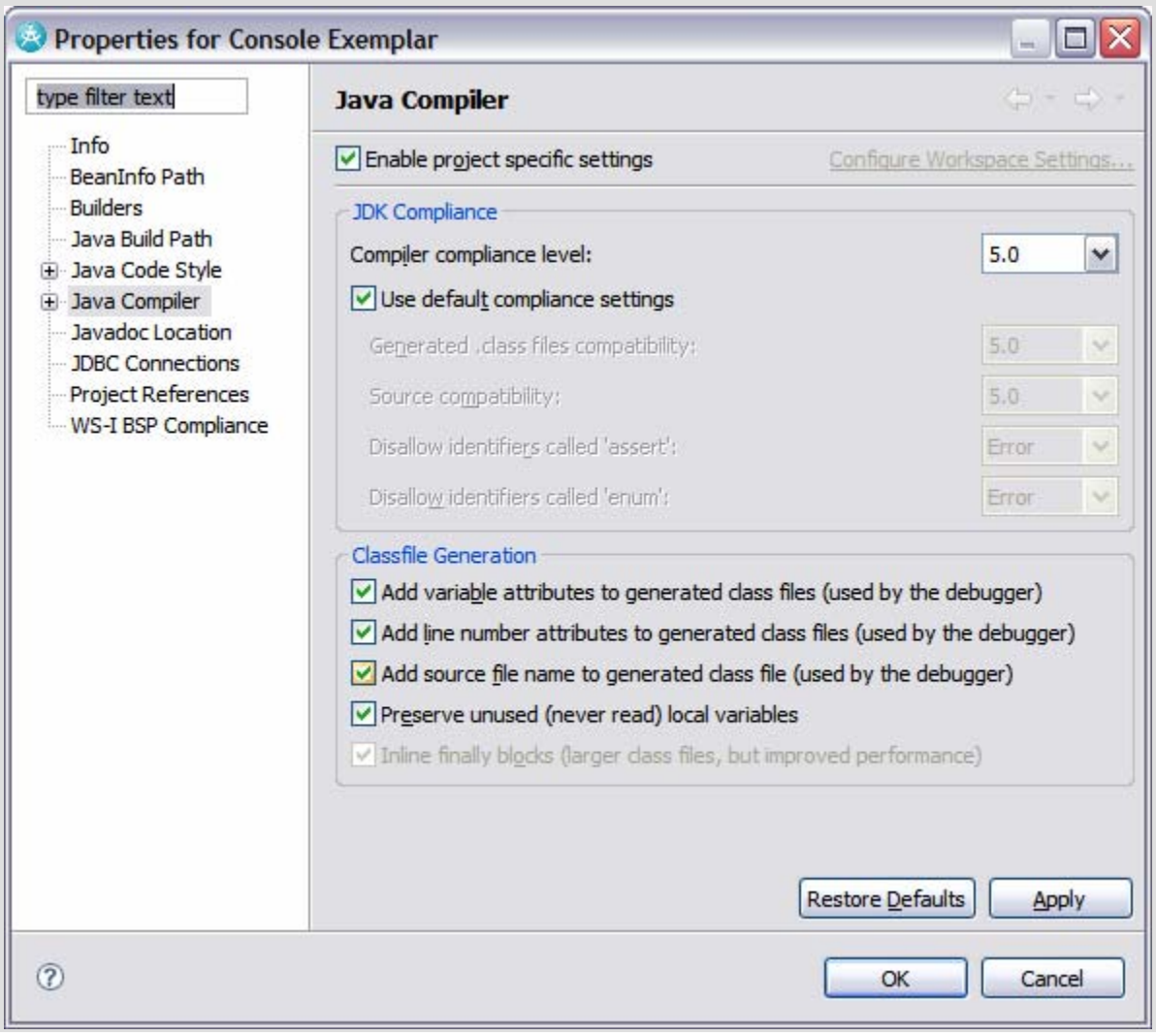
Each command accepts a specific set of typed arguments. There is code in the handler to convert each string token to an appropriately typed local variable.

This exemplar is representative of a class of applications that accept command line input, and then invoke the appropriate command. The transform you will author will generate instances of these command line applications.

As with any Exemplar Analysis exercise, be sure to ask the SME (the instructor in this case) if you have any questions about the implementation of the exemplar application or about the points of variability to be supported by the transform.

You should now have the Java project containing the console exemplar in your workspace.

TIP: The project was written using features of Java 5. To get the code to compile you must be using a JRE that supports that version of Java. If the code does not compile for you, right-click the project and select **Properties**. With the Properties window, select **Java Compiler**, click **Enable project specific settings** and then set the **Compiler compliance level** to **5.0**.



3. Create a new JET transformation project called **console.transform**. Use the EMFT JET Project with Exemplar Authoring wizard.

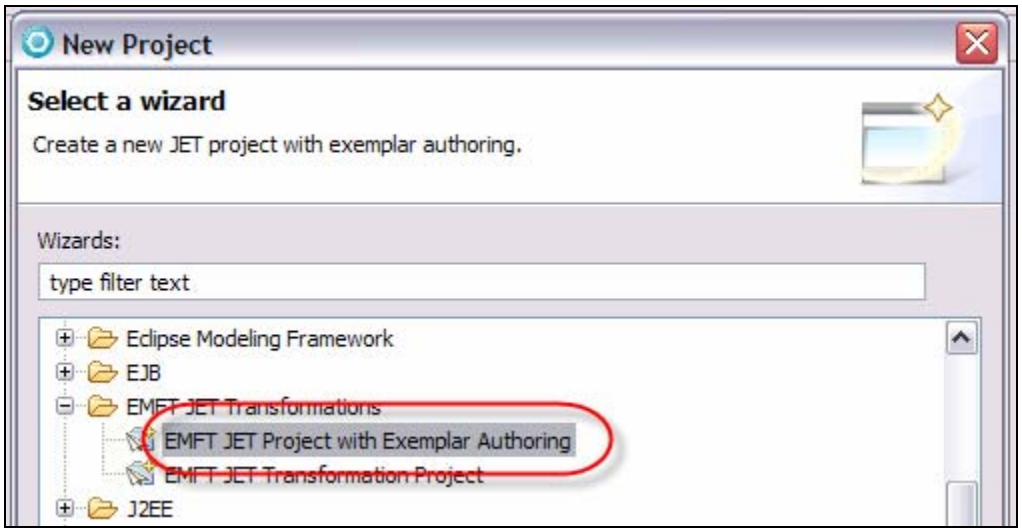


Figure 5 - 2: Creating an EMFT JET Project with Authoring Exemplar

4. Be sure to specify that the Console Exemplar project is selected as the **Exemplar scope**.

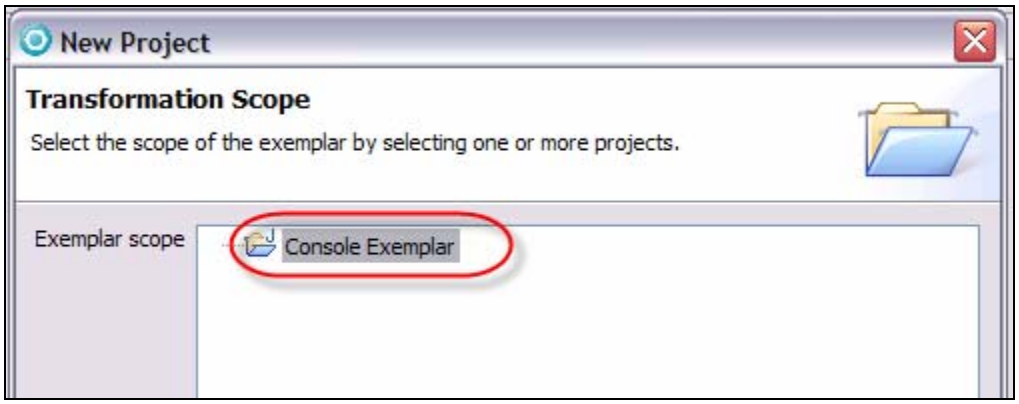


Figure 5 - 3: Specifying the Exemplar Scope

5. The Exemplar Authoring tool should now display the console exemplar and an empty model

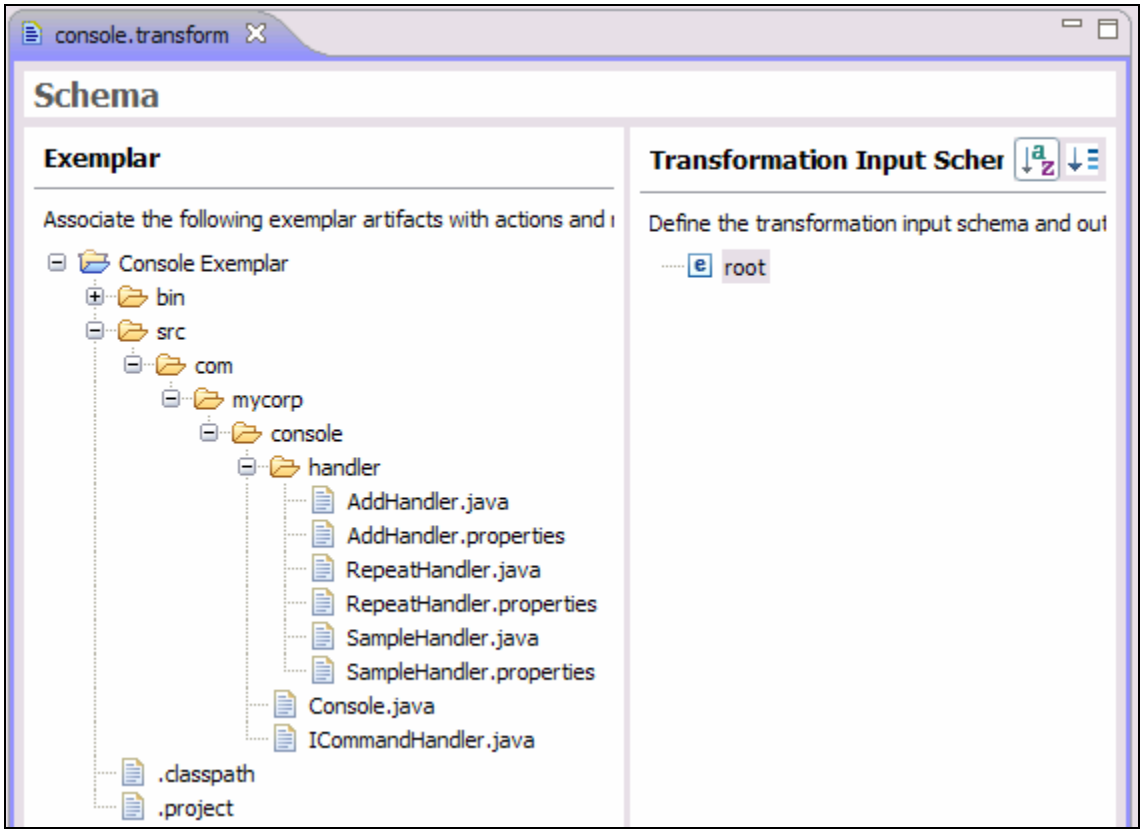


Figure 5 - 4: The console exemplar

Task 2: Populate the Model

1. You propose a one-word name, `console`, to describe the entire set of files in the exemplar and create a second-level model type by that name.

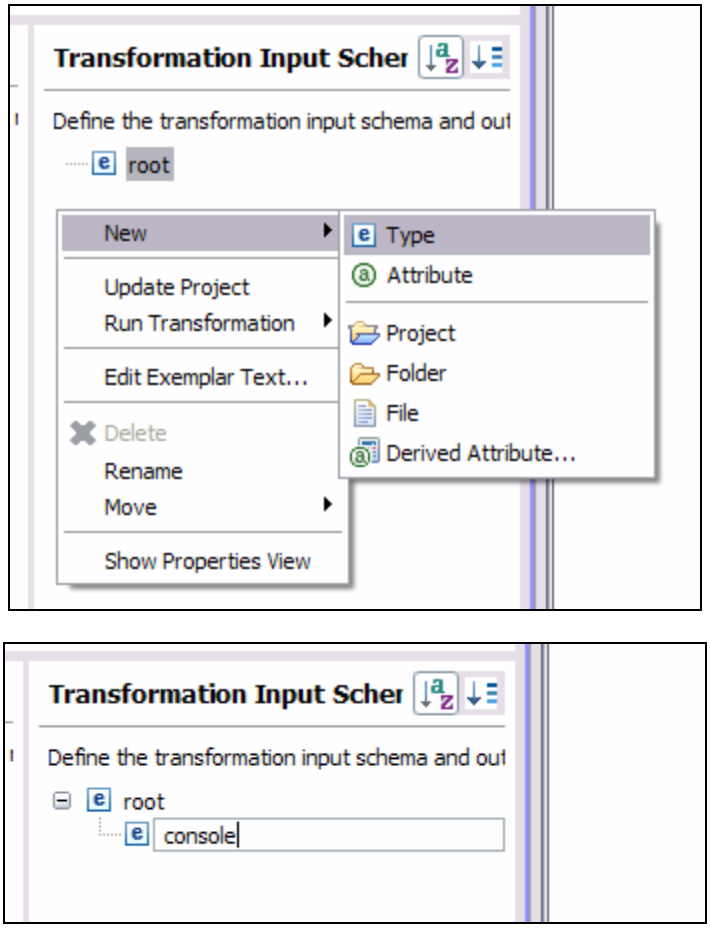


Figure 5 - 5: Creating the console type

2. Identify the artifacts that will be created only once for each application of the transform. They include:
   - The Java project **Console Exemplar**
   - The project meta-data files **.classpath** and **.project**
   - The main class **com.mycorp.console.Console.java**
   - The handler interface **com.mycorp.console.ICommandHandler**
3. Drag each of these artifacts from the left pane onto the console type icon in the right pane. Be careful not to drop any of the artifacts onto the `Create Project` action.

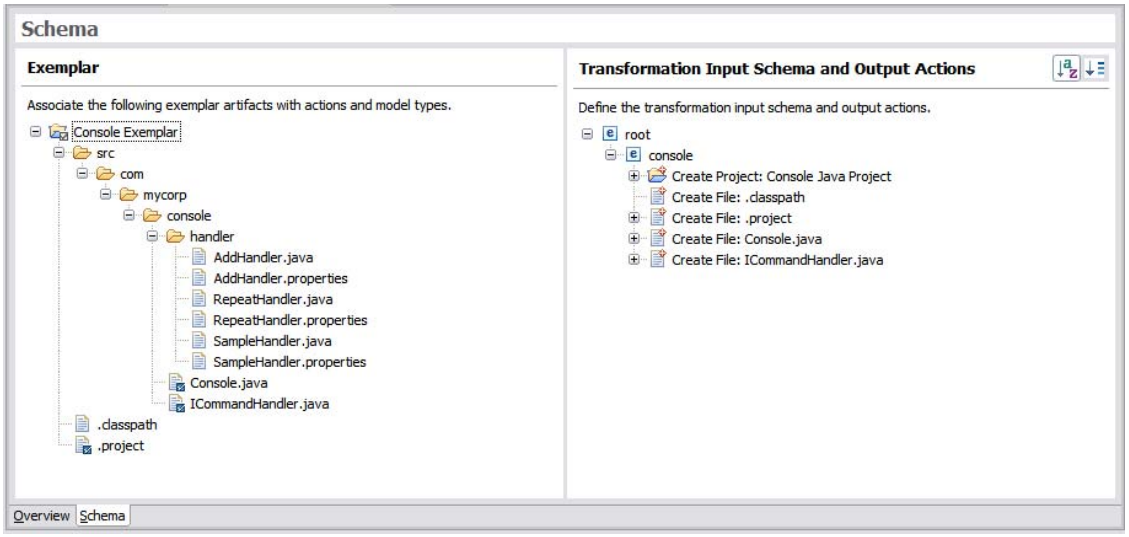


Figure 5 - 6: Artifacts added under console

4. Note that the remaining files (`xxxHandler.java` and `xxxHandler.properties`) seem to be repeated in pairs, with each pair having a Java source file and a properties file with a common root name. Because of the one-to-many relationship between the Java project and these pairs of files, you will create a new nested type under the console type.

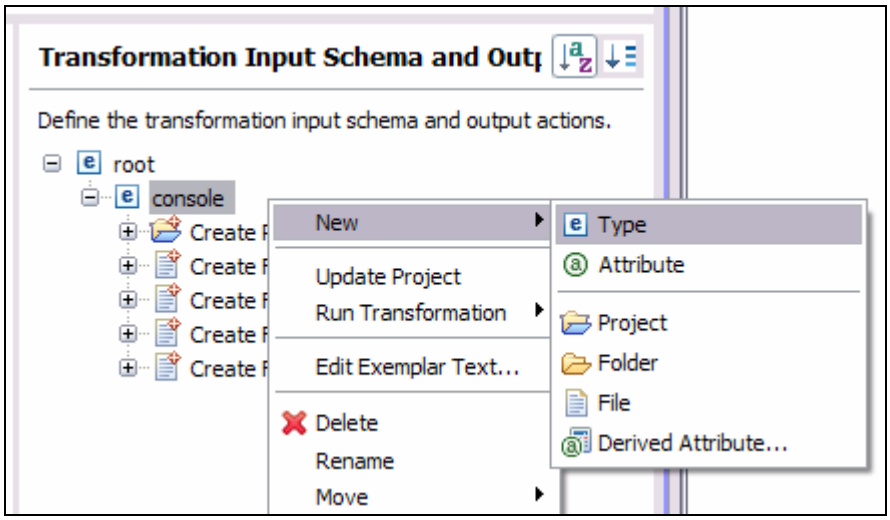


Figure 5 - 7: Creating a new type under console

5. The type represents a pair of files, a Java class and a properties file, in support of one of the commands implemented by the console. The name you choose for this new type, “command”, describes this pair of files.

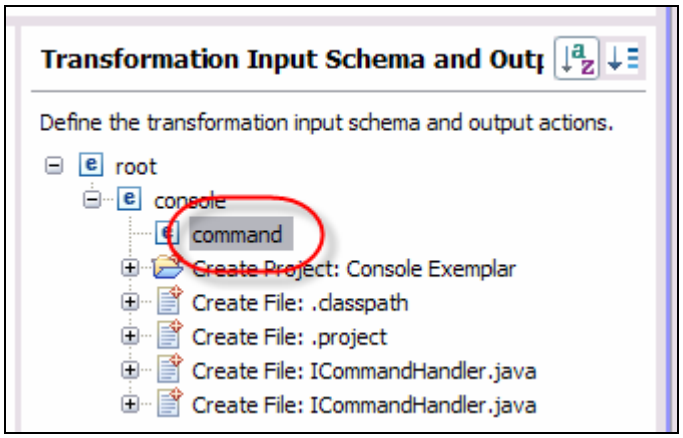


Figure 5 - 8: The command type

6. You need to drag representative samples of each of the files to be generated for this command type. The question is, which files should you use?

The choice is important, because the content of the files will be used as the initial template for each resulting action. You want to choose the exemplar files that are most representative of the points of variability in the pattern. In this example, the files for the Sample command demonstrate the most variety of parameter types.

7. Drag the two files, `SampleHandler.java` and `SampleHandler.properties` on top of the command type in the right pane.

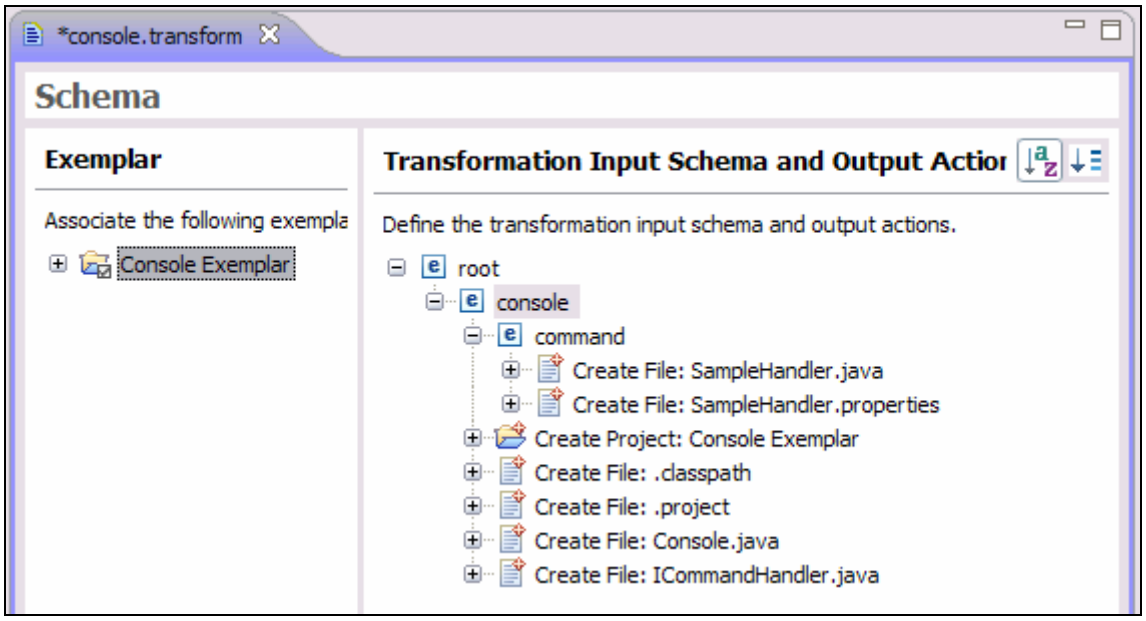


Figure 5 - 9: Sample files under command

Each of the create file actions, as well as the create project action, will create an Eclipse resource with a variable name. The list below shows the names of those associated exemplar artifacts.

- `ConsoleExemplar`
- `ConsoleExemplar/.project`
- `ConsoleExemplar/.classpath`
- `ConsoleExemplar/src/com/mycorp/console/Console.java`
- `ConsoleExemplar/src/com/mycorp/console/ICommandHandler.java`
- `ConsoleExemplar/src/com/mycorp/console/handler/SampleHandler.java`
- `ConsoleExemplar/src/com/mycorp/console/handler/SampleHandler.properties`

Within each of the above names you can identify a number of substrings that are likely to vary from application to application of the transform:

- `ConsoleExemplar` (name of the project)
- `com/mycorp/console` (name of the console directory under the source folder)
- `com/mycorp/console/handler` (name of the handler directory)
- `SampleHandler` (name of a command handler)

These names, according to best practices, are to be stored in derived attributes in the model. These names are derived from a number of other attributes:

- The name of the console being generated
- The console package
- The handler package (this turns out to be a derived attribute, too)
- The command name

Task 3: Add and Derive Attributes

1. Add the three attributes above into the model.

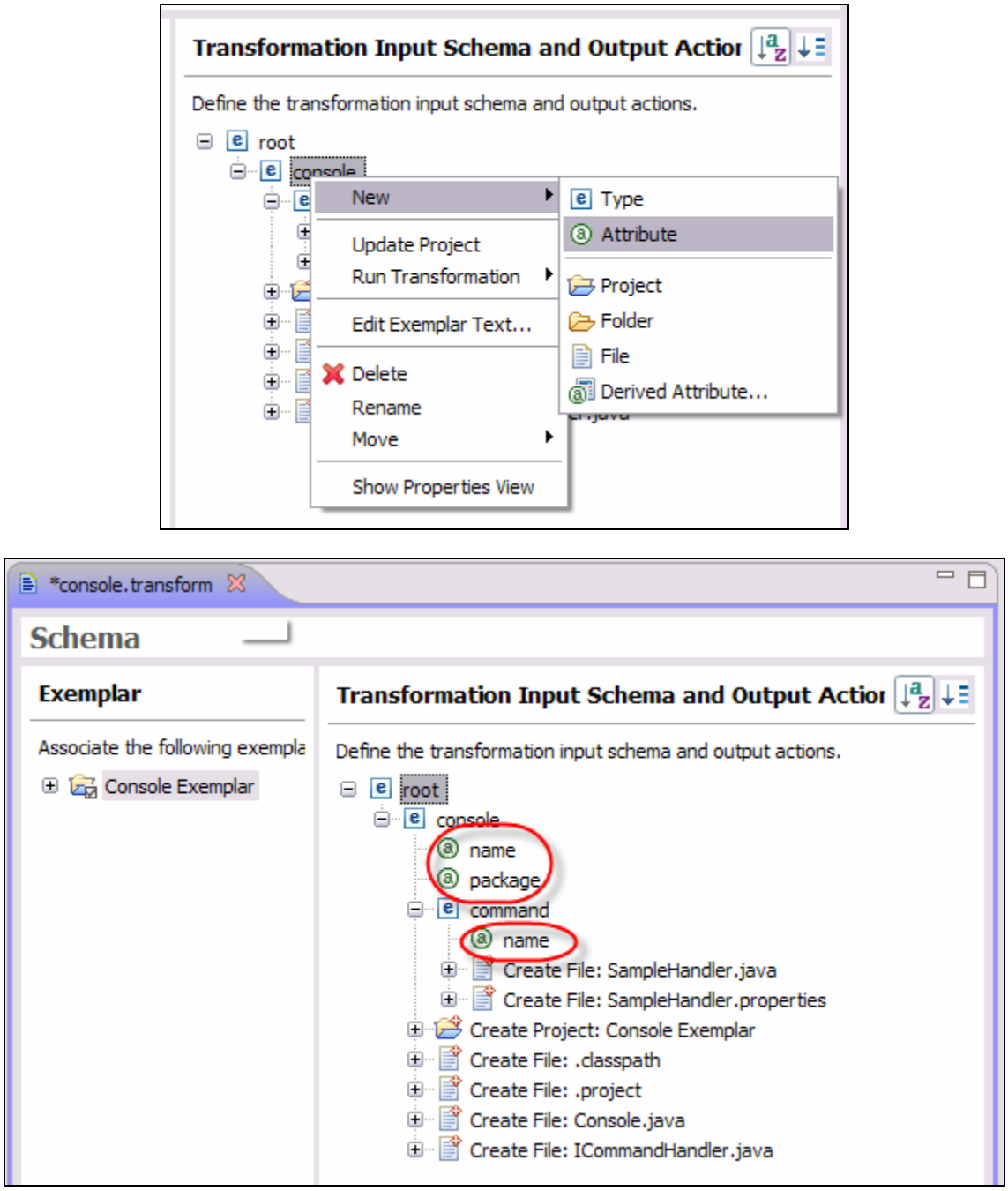


Figure 5 - 10: Adding attributes to console and command

2. Select the `Create Project: Console Exemplar` action and view the properties for that action in the Properties view.

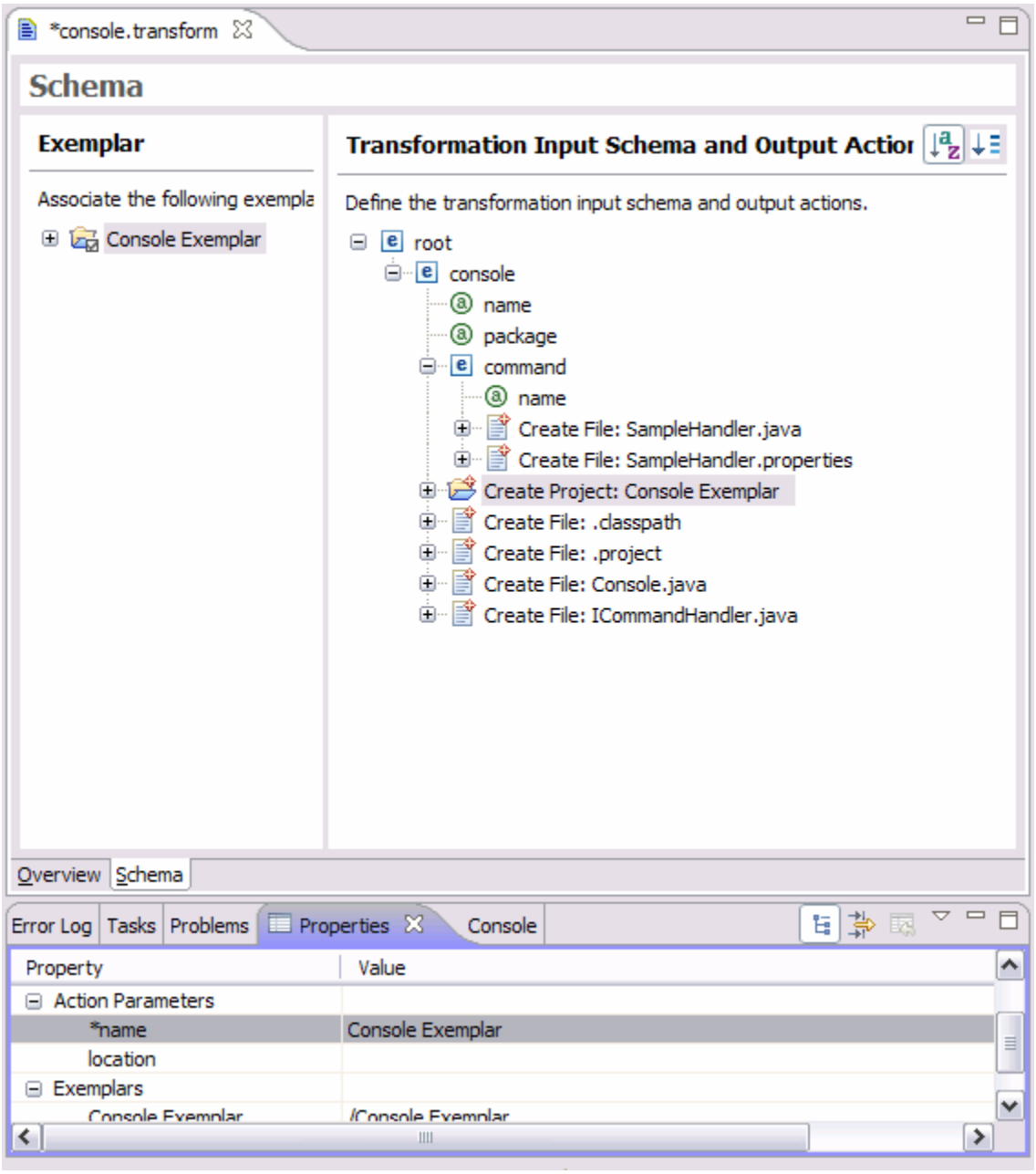


Figure 5 - 11: Properties of Create Project: Console Exemplar

In particular, note the value of the name action parameter. The value of that parameter will be used by the transformation to name the console project when it is first created. Since that project name needs to be variable, you need to define the calculation to be used to determine the project's name. Since the name of the project, according to best practices, needs to be kept in a derived attribute, you need to define such an attribute and indicate that that attribute's value is to be used as the project's name.

3. Begin by selecting the entire text of the exemplar name and clicking the **Replace with Model Reference** button.

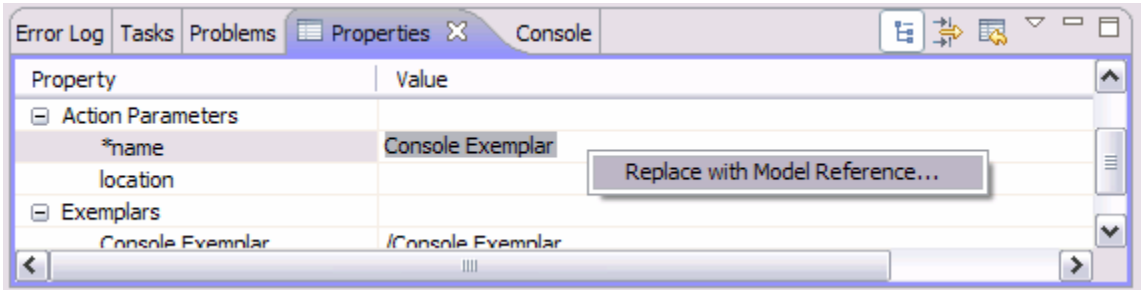


Figure 5 - 12: Replace with Model Reference

4. A dialog box will display the known model types and attributes.

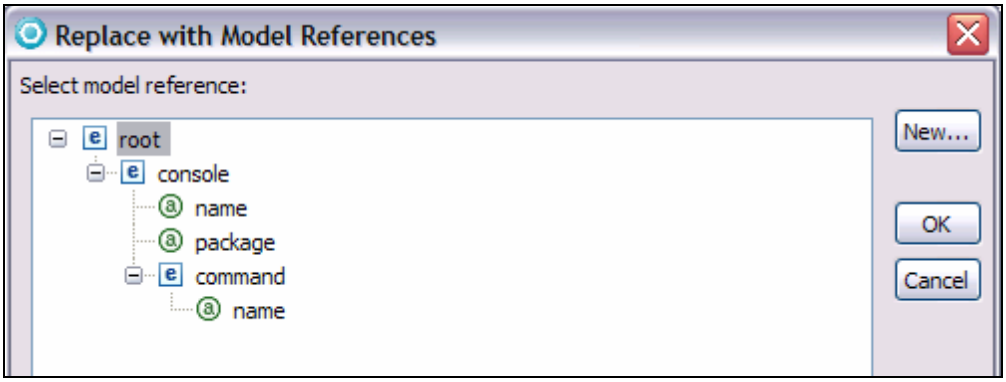


Figure 5 - 13: Replace with Model References dialog box

5. Since the derived variable that you want to use to hold the project name isn't defined yet, click the **New** button to create that derived attribute definition.

Note: Be sure to select the console type before clicking the **New** button, since the console type is the type that has to contain this new derived attribute.

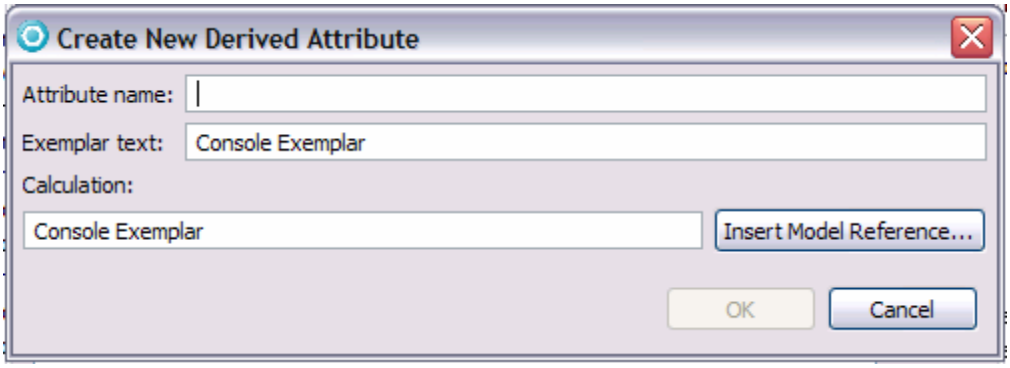


Figure 5 - 14: Create New Derived Attribute dialog box

6. The name of the new derived attribute will be `projectName` and the value of the attribute will be calculated by concatenating the console name with the constant string `Console`.
7. Point the cursor to the start of the Calculation field and click **Insert Model Reference**.

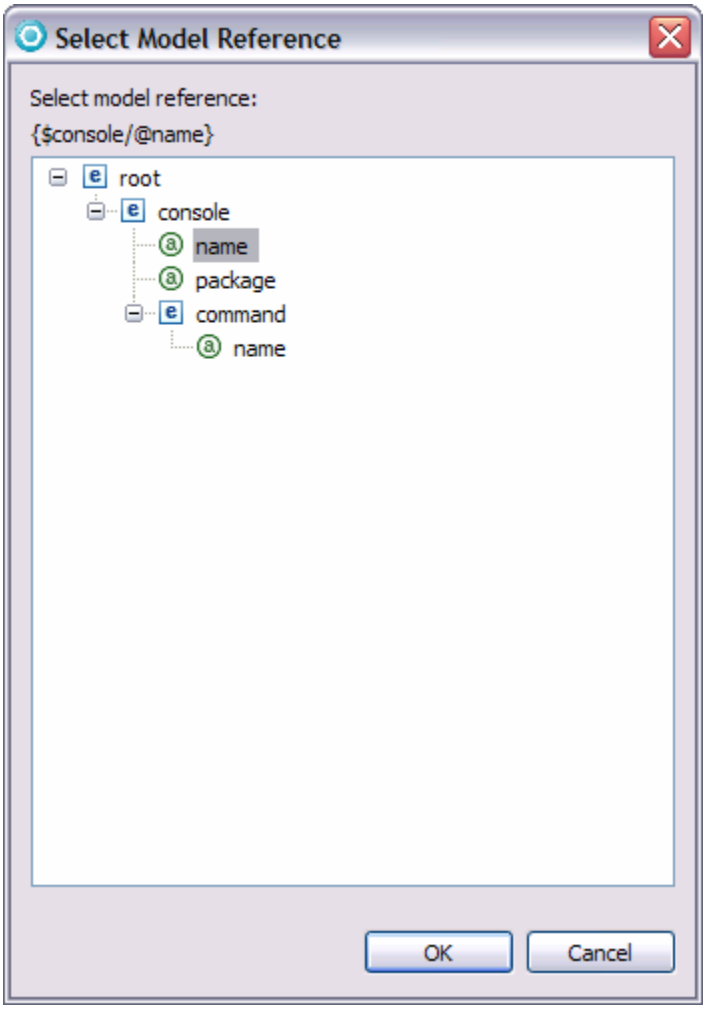


Figure 5 - 15: Select Model References dialog

8. Select the `name` attribute for model type console and click **OK**.

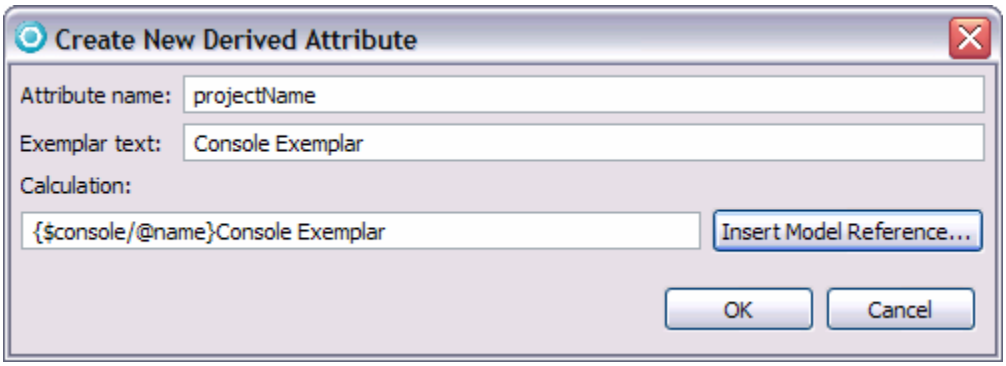


Figure 5 - 16: Modifying the name attribute

9. Note that the query expression for the `name` attribute has been inserted into the **Calculation** field. Edit the rest of the field to define the calculation correctly.

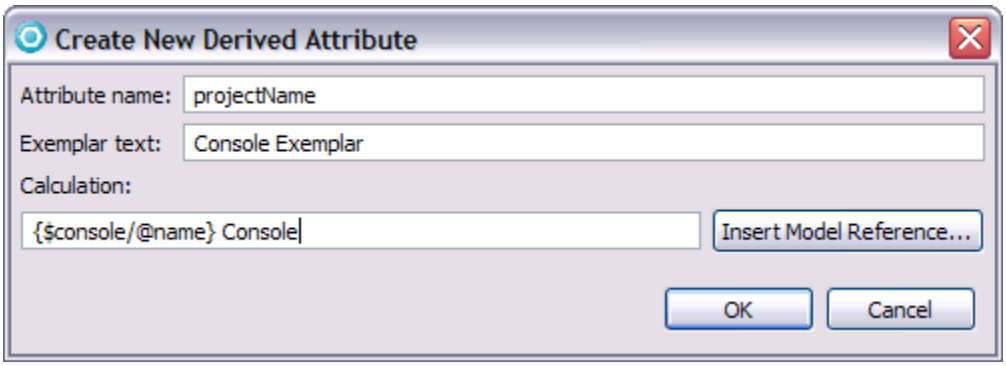


Figure 5 - 17: Adding a calculation to a new attribute

10. Click **OK** to return to the Replace with Model References dialog. Note that a new derived attribute named `projectName` has been added to the model.

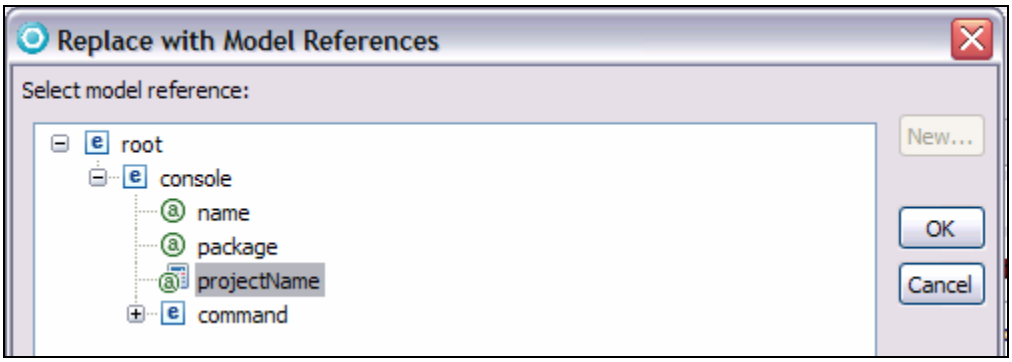


Figure 5 - 18: A new derived attribute

11. Select the `projectName` attribute and click **OK**. Note that the action parameter name is now set to a query expression referring to `projectName`.

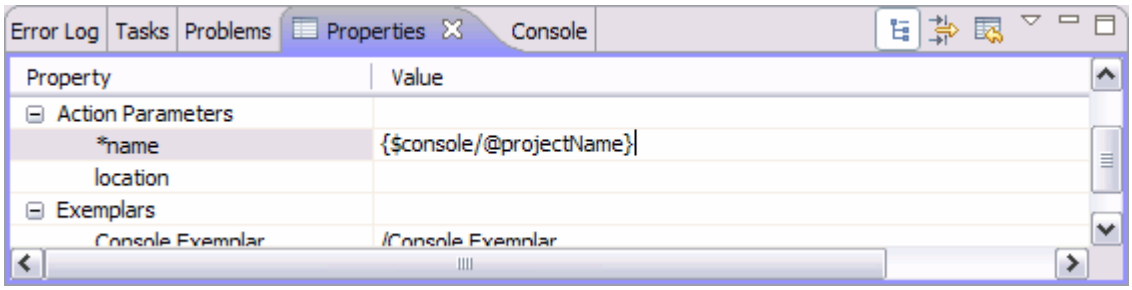


Figure 5 - 19: Action Parameters name property

12. Select the `.classpath` file action and edit the path action parameter.

13. Select the string “Console Exemplar” and use **Replace with Model Reference** to replace the string with a reference to the `projectName` derived attribute.

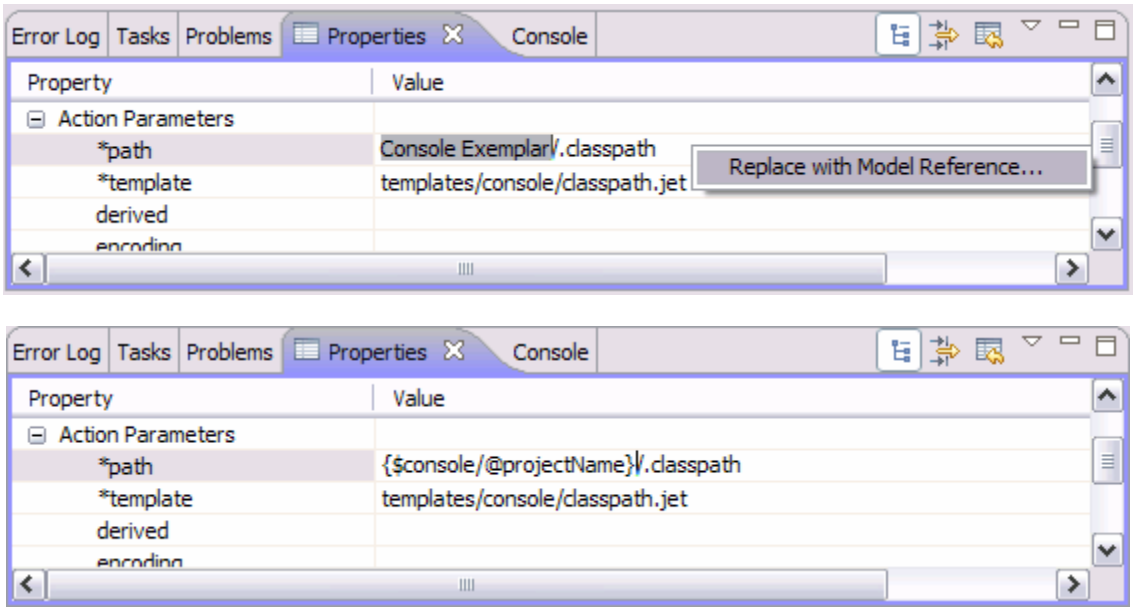


Figure 5 - 20: Changing the path Action Parameters property

14. Do the same for the .project action:

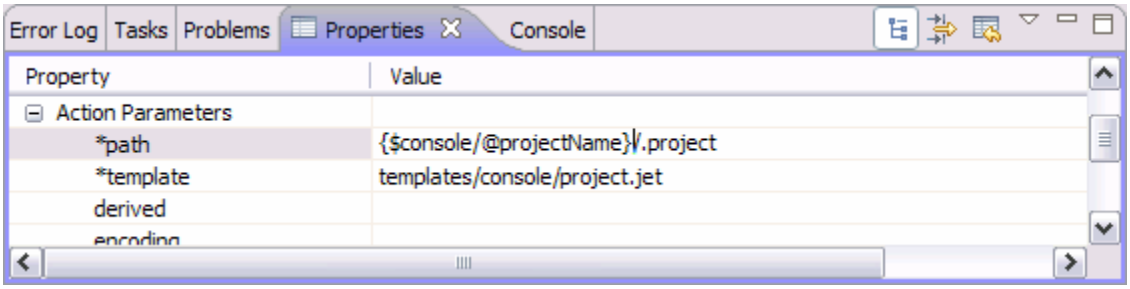


Figure 5 - 21: Changing the path Action Parameters property

15. Select the Console.java action and add a reference to the project name

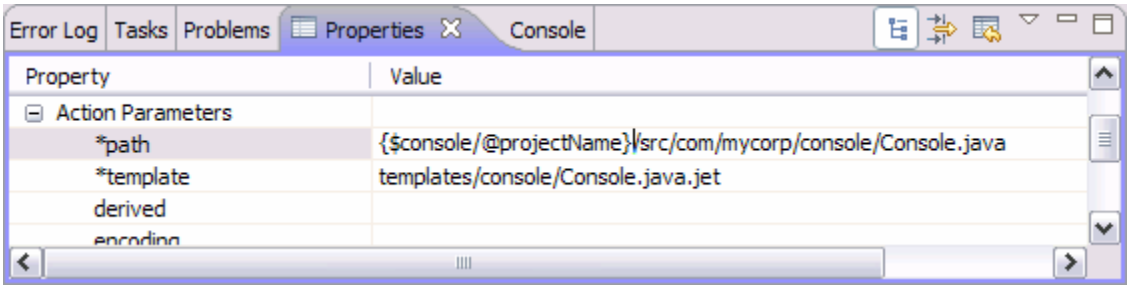


Figure 5 - 22: Changing the path Action Parameters property

16. Of the remaining path value, only the substring `com/mycorp/console` has been identified as possibly changing from transform application to application. You need to replace that substring with a reference to a new derived attribute.

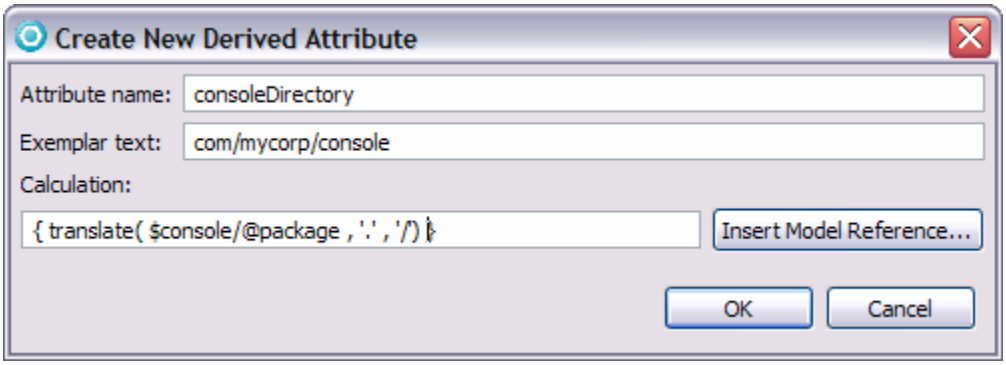


Figure 5 - 23: Creating a new derived attribute

17. The attribute is derived by replacing all periods in the package value with forward slashes.

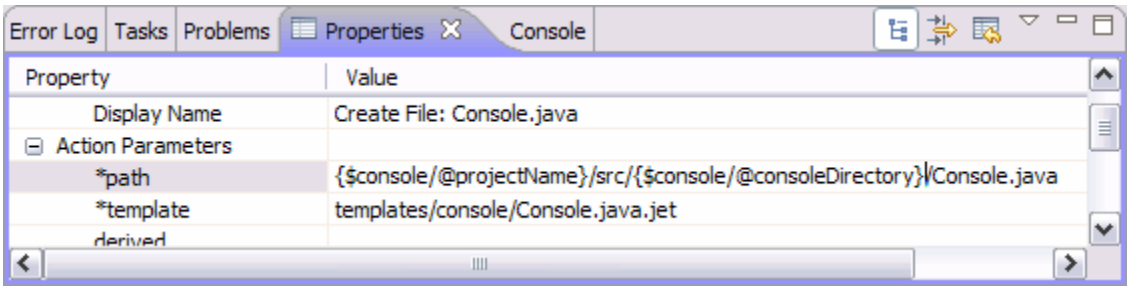


Figure 5 - 24: Creating a new derived attribute

18. The path parameter is similarly modified for the `ICommandHandler` action:

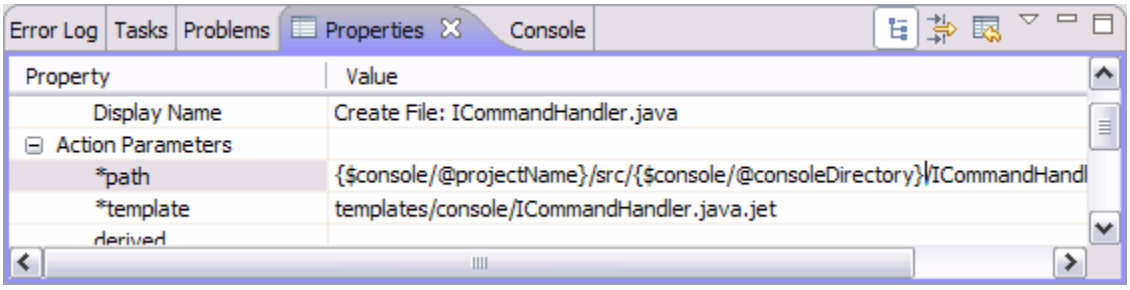


Figure 5 - 25: Changing the path parameter for ICommandHandler

19. Select the `SampleHandler.java` action and review its path parameter:

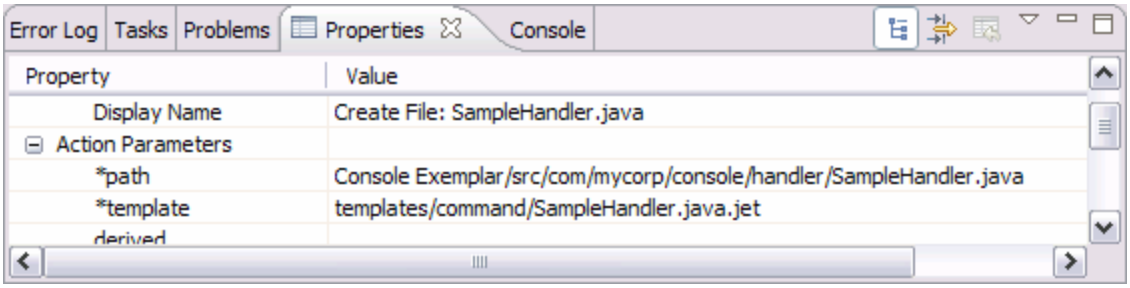


Figure 5 - 26: Reviewing the path property

20. There are two substrings which need to be replaced by derived attributes. The substring `com/mycorp/console/handler` needs to be replaced by a reference to derived attribute `handlerPackage`, which in turn is derived from attribute package:

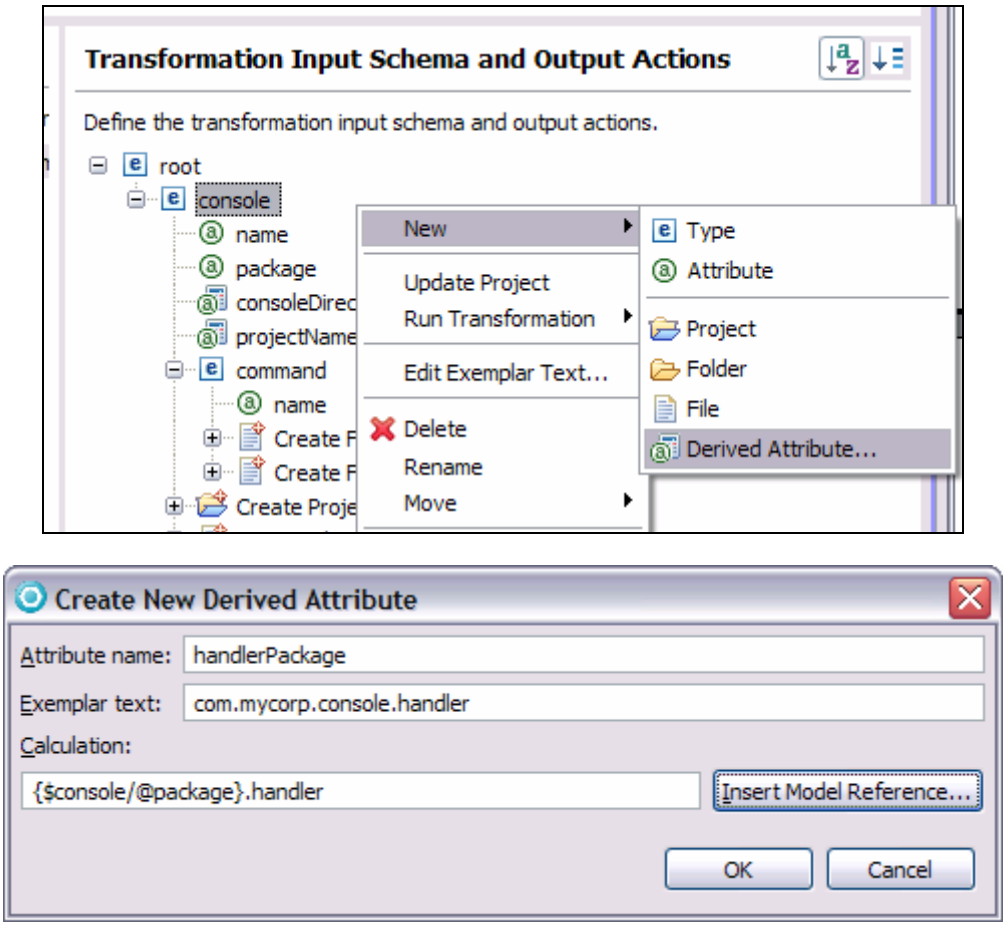


Figure 5 - 27: Creating a new derived attribute

21. The `handlerDirectory` attribute is derived from attribute `handlerPackage`.

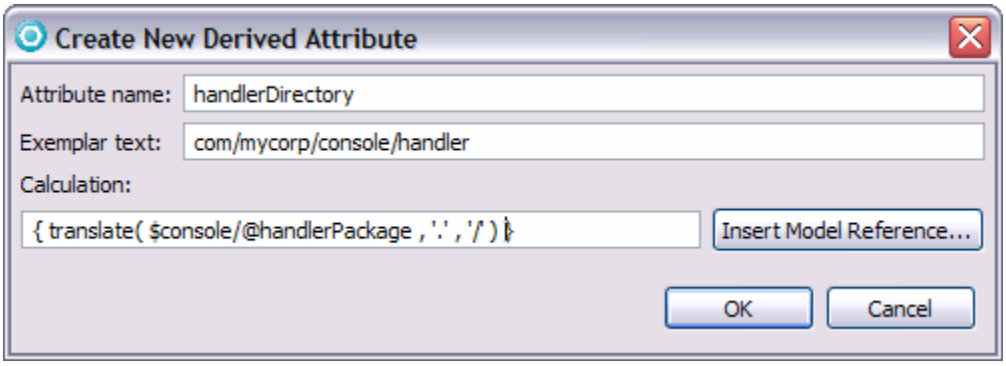


Figure 5 - 28: Creating the handlerDirectory derived attribute

22. The substring `SampleHandler` needs to be replaced by the new derived attribute `handlerName` on model type command.

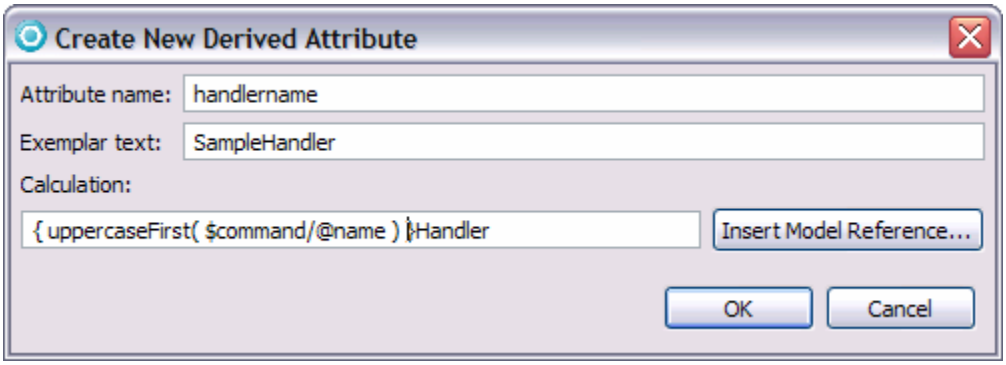


Figure 5 - 29: Creating the handlername derived attribute

23. And the path parameter for `SampleHandler.java` should be finished.

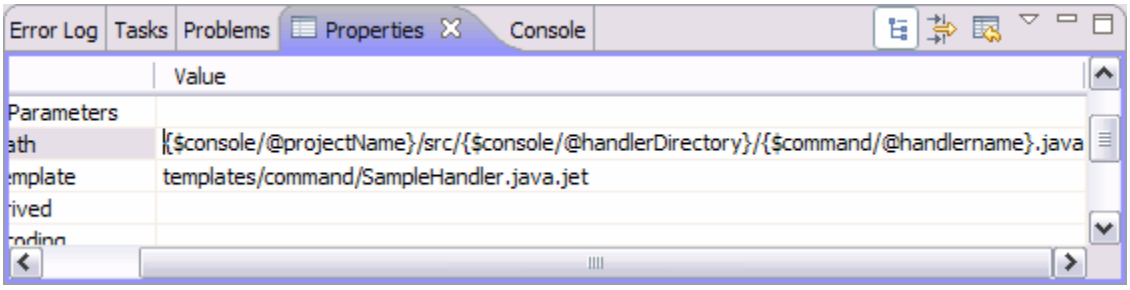


Figure 5 - 30: Editing the Path property for SampleHandler.java

24. Edit the path parameter for the `SampleHandler.properties` action in the same way:

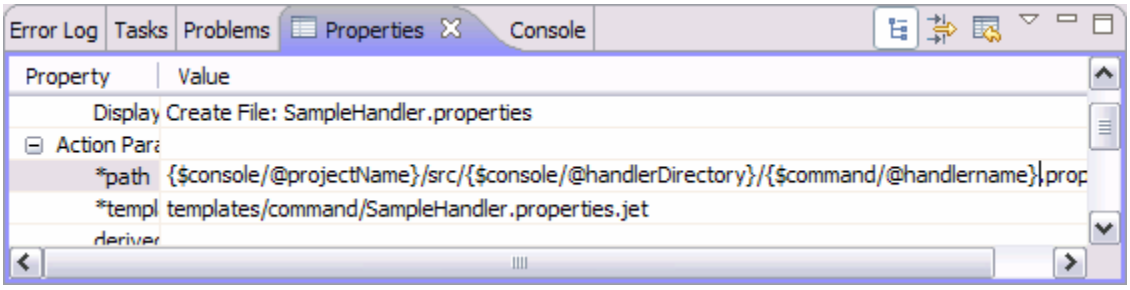


Figure 5 - 31: Editing the Path property for SampleHandler.properties

The completed model looks like this.

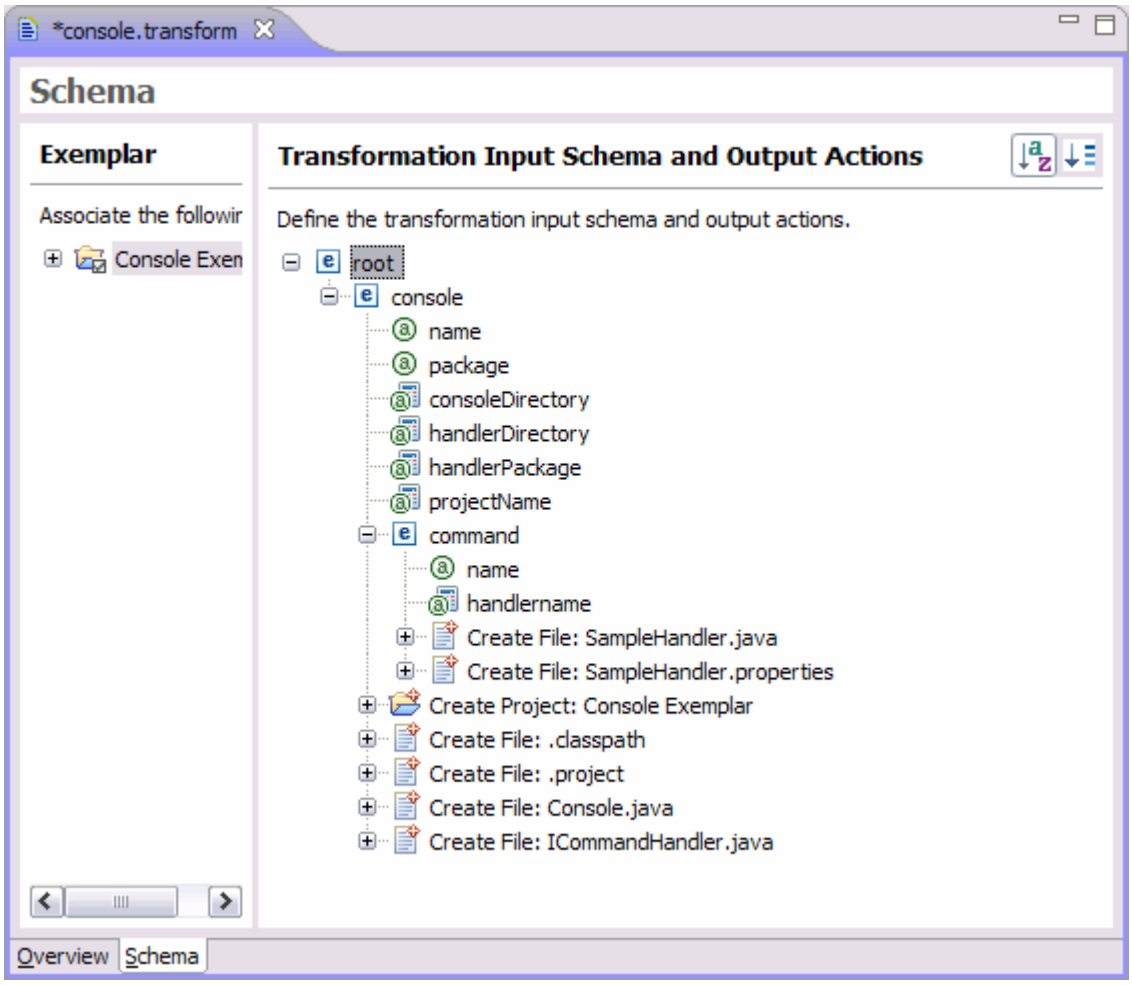


Figure 5 - 32: The completed model

25. Select **File > Save All**.

Task 4: Generate and Edit Templates

It's now time to generate the templates for the JET transform.

1. Use the **Update Project** action.

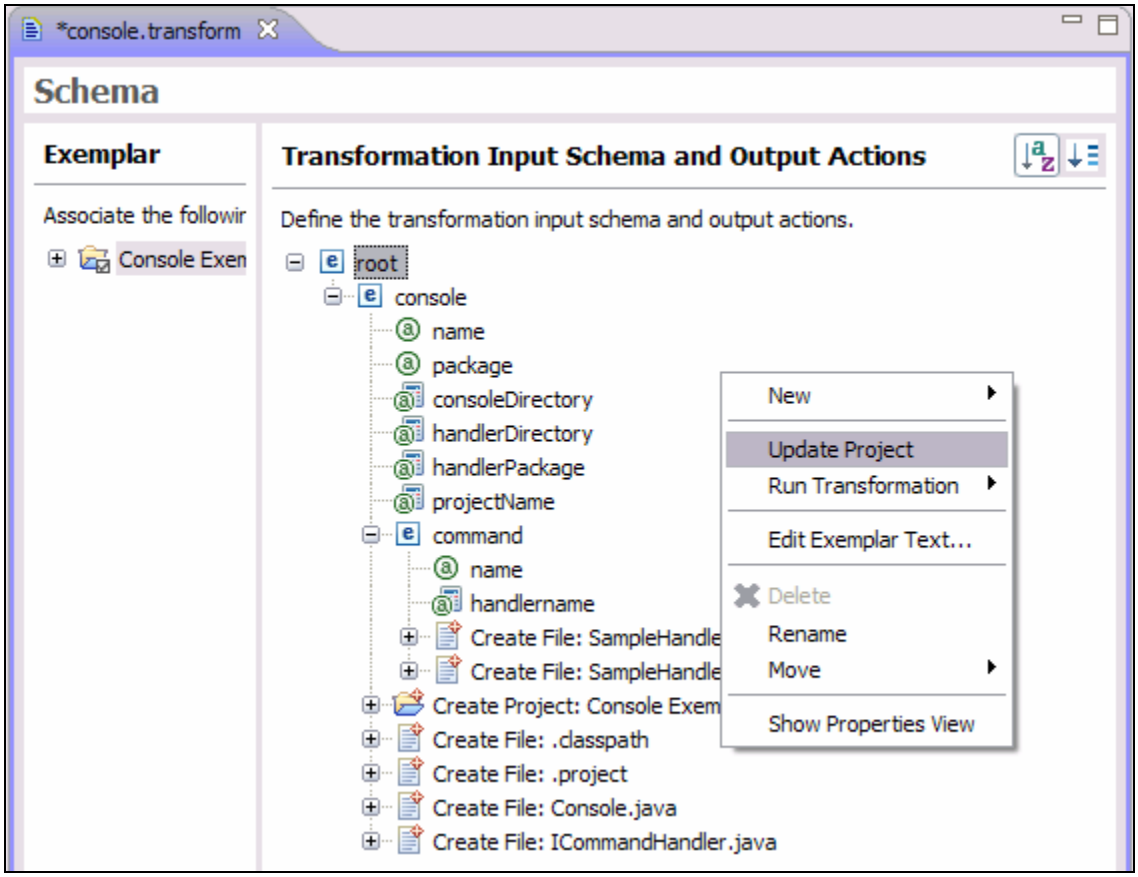


Figure 5 - 33: Updating the Project

Note the new templates that have been generated.

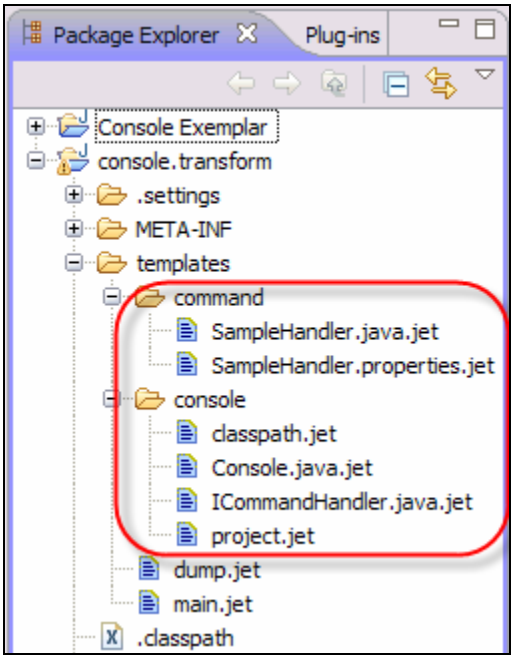


Figure 5 - 34: New generated templates

2. Edit the templates one at a time, starting with `project.jet`.

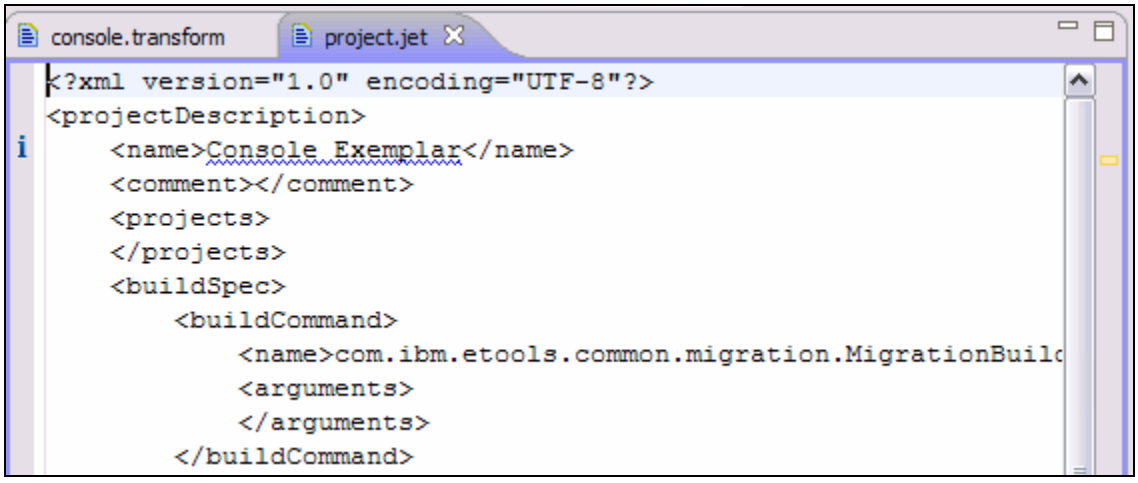


Figure 5 - 35: Editing project.jet

Note the blue underscore under the name element, `Console Exemplar` that indicates that that string matches one of the exemplar strings for one of the attributes. It's likely that the string should be replaced by a query expression referencing that attribute.

3. Select the underlined string and click **Find/Replace with JET Model Reference**.

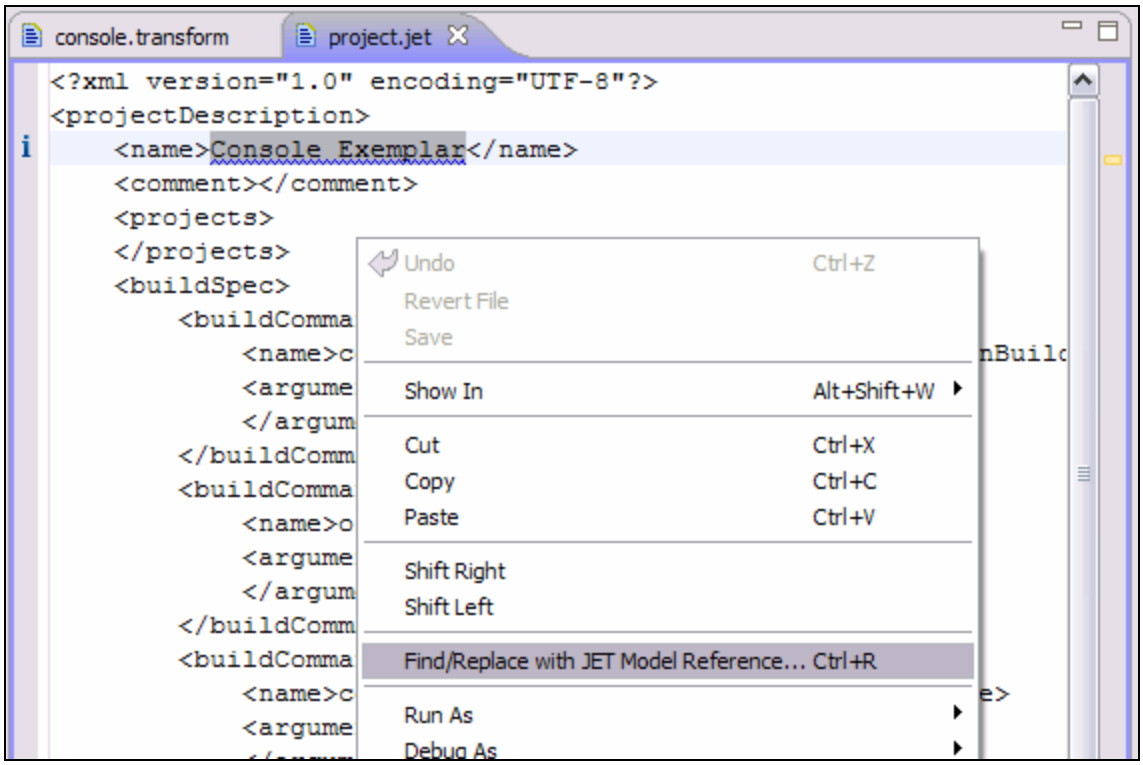


Figure 5 - 36: Clicking Find/Replace with JET Model Reference

4. Select the `projectName` attribute and click **Replace**, and then click **Close**.

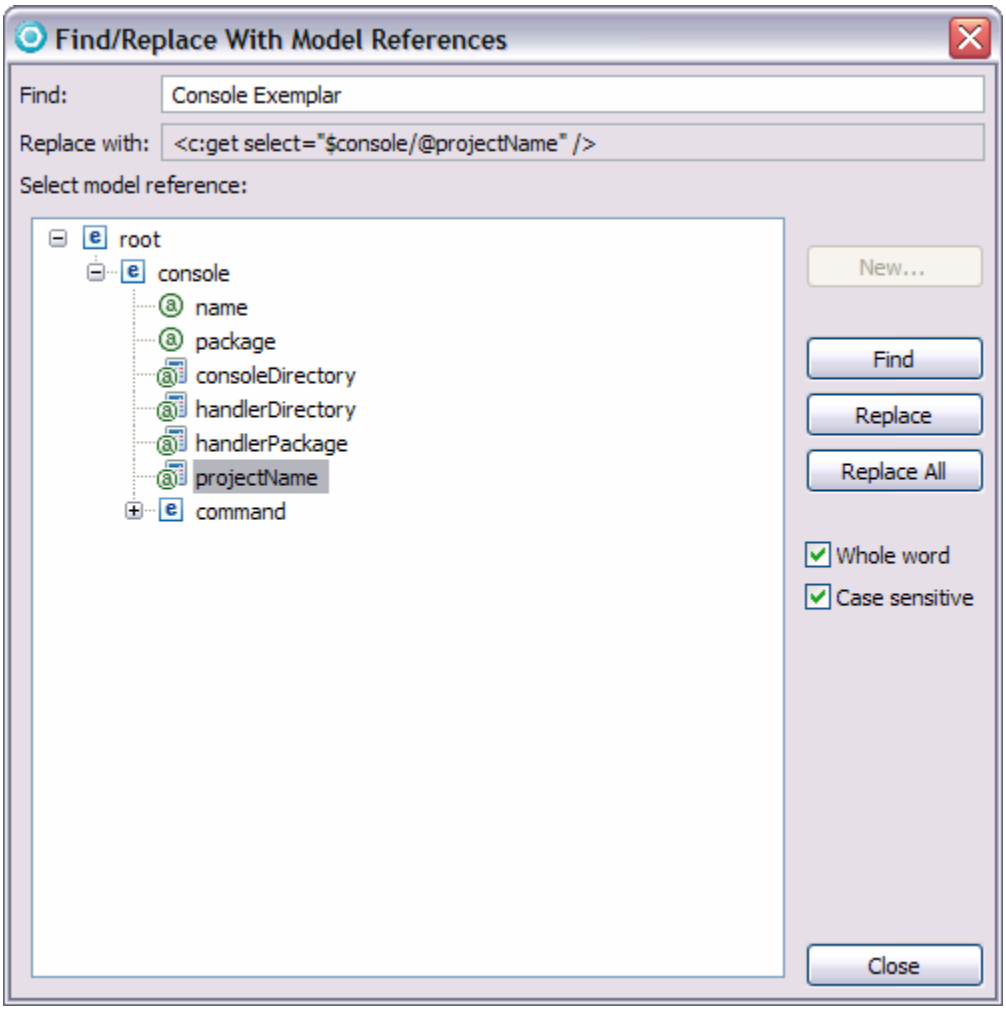


Figure 5 - 37: Selecting projectName

5. The string in the template will be replaced by the correct `<c:get>` tag.

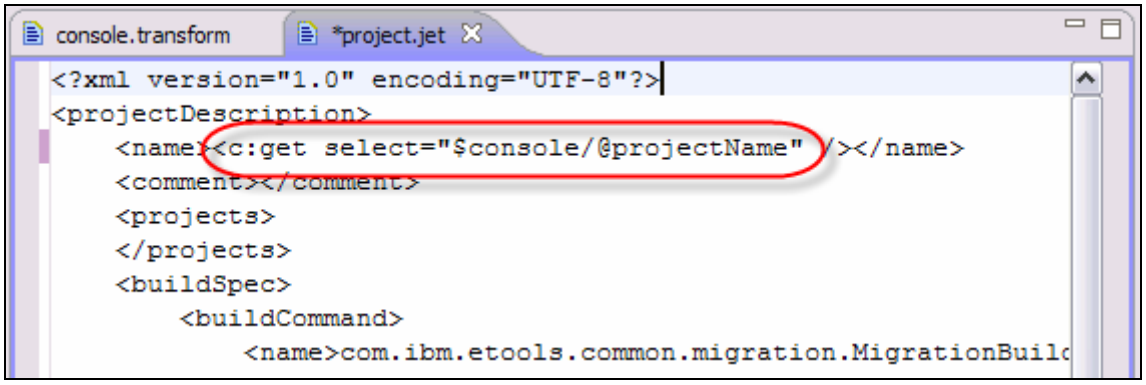


Figure 5 - 38: The string replaced by <c:get>

6. Close the `project.jet` template and open the `ICommandHandler.java.jet` template. Use the **Find/Replace with JET Model References** dialog to replace the package name with the correct `<c:get>` tag.

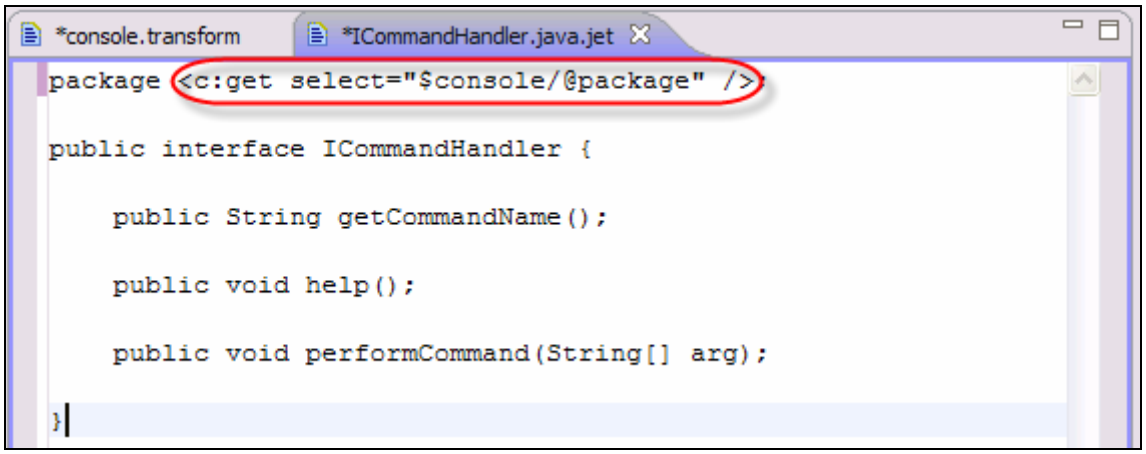


Figure 5 - 39: The string replaced by <c:get>

7. Open template `Console.java.jet` and replace the package name with a reference to the package attribute.

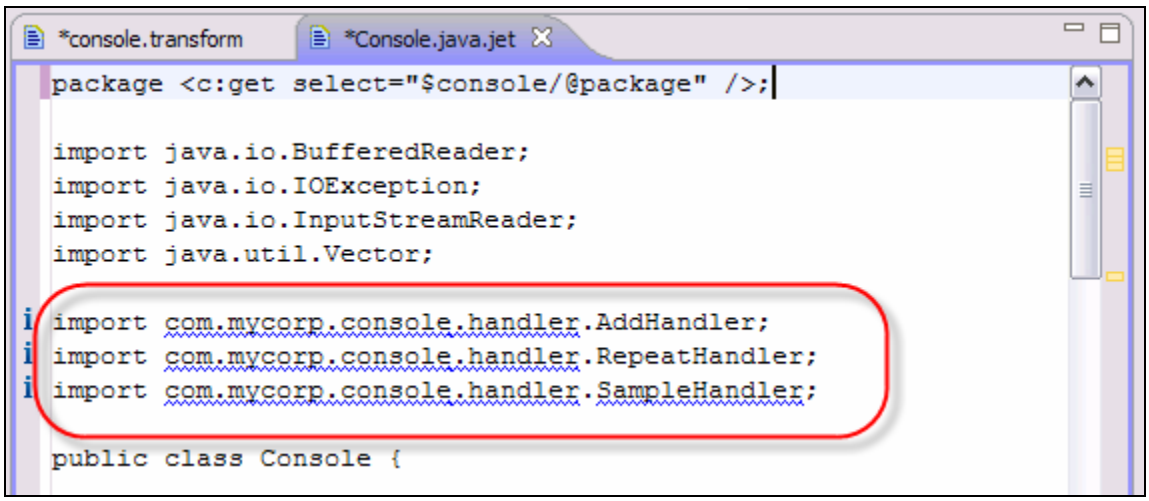


Figure 5 - 40: The import statements that need to be updated

8. Note that there is a list of three import statements which will vary from application to application of the transform. You need to generate one import line for each command object defined for the console. You first add the <c:iterate> tag:

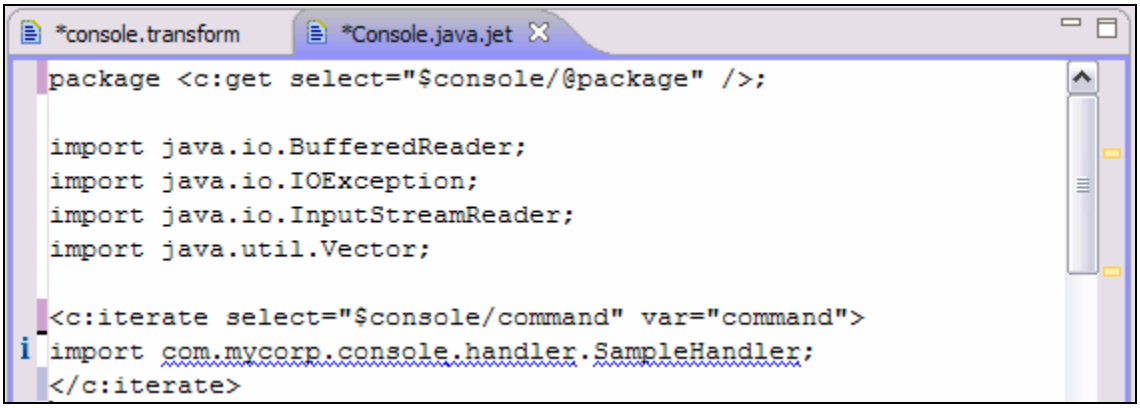


Figure 5 - 41: Replacing the package name with a reference to the package attribute

9. Now use the Find/Replace with JET Model Reference dialog to replace the strings `com.mycorp.console.handler` and `SampleHandler` with the appropriate tags.

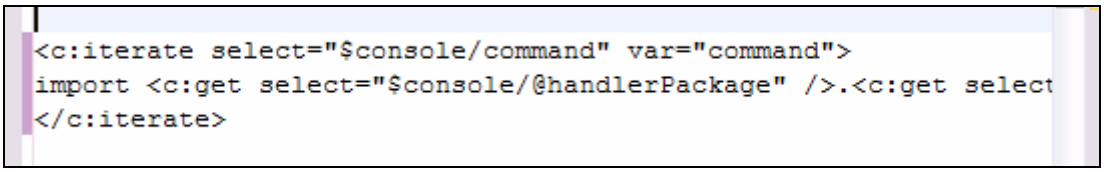


Figure 5 - 42: Replacing strings with tags

10. Mark up a similar list further down in the template:

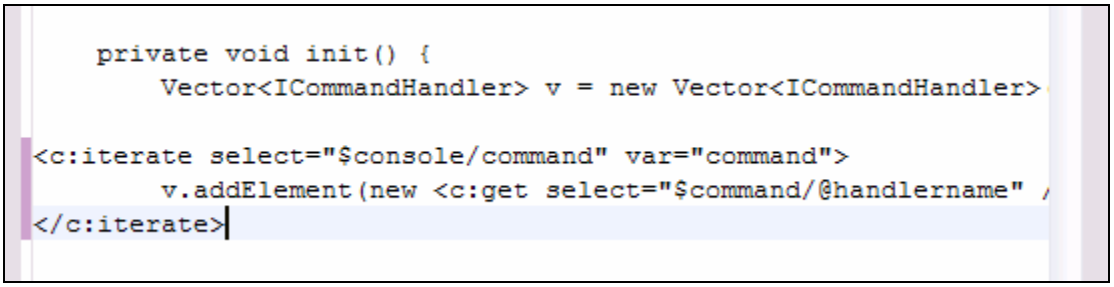


Figure 5 - 43: Replacing strings with tags

11. Edit the template `SampleHandler.java.jet`. Replace the following strings with references to the appropriate attributes in the following order:

- `com.mycorp.console.handler`
- `com.mycorp.console`
- `SampleHandler`
- `sample` (with command name)

12. Note the implementation of the `performCommand` method. There are what turn out to be a list of local variables that correspond to the types command arguments. Each argument seems to have a name, a local variable name, and a type.

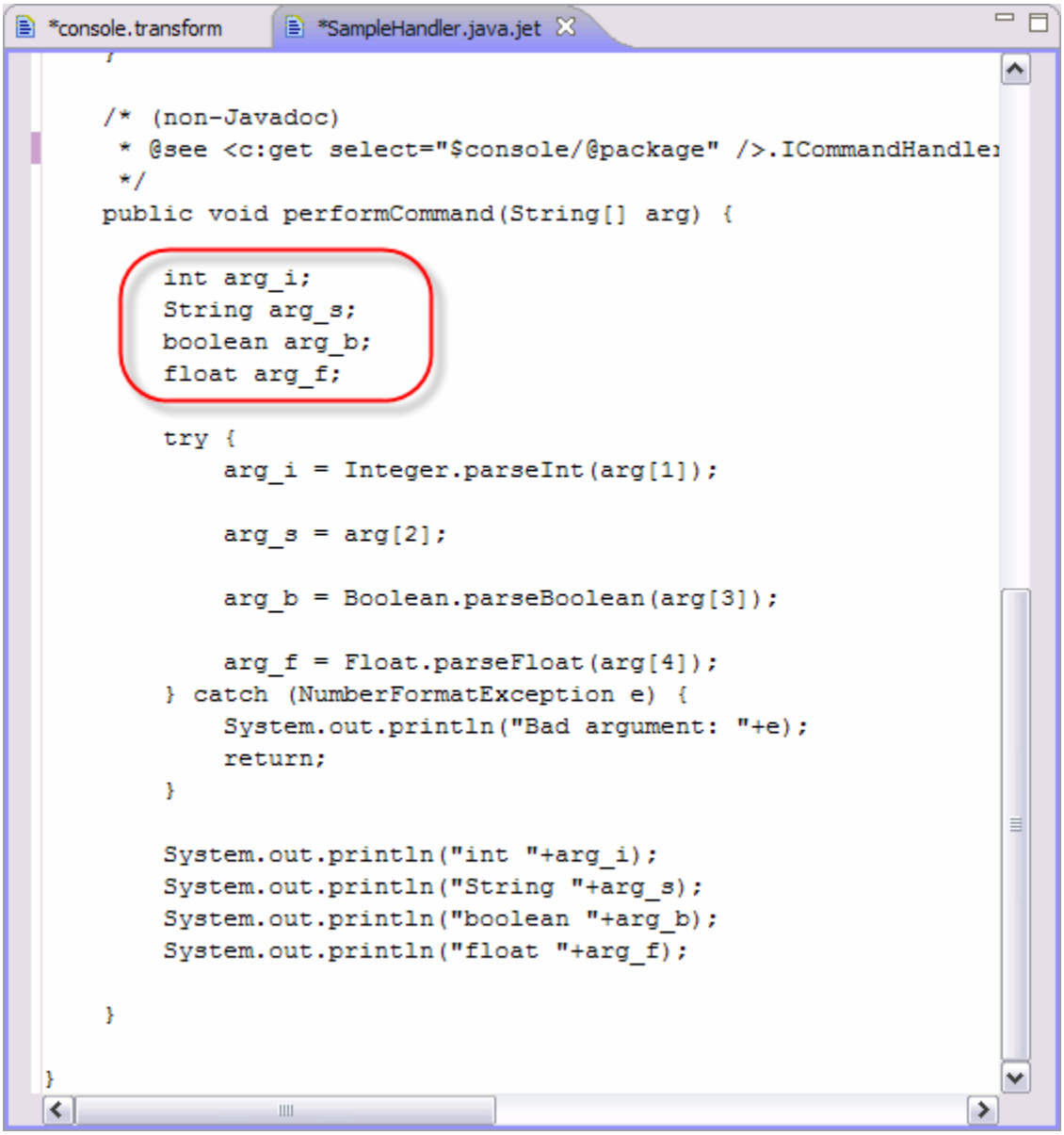


Figure 5 - 44: Local variables of performCommand

13. You need to go back to the Exemplar Authoring tool and add a new model type (`argument`) and two attributes (`name` and `type`) to the model.

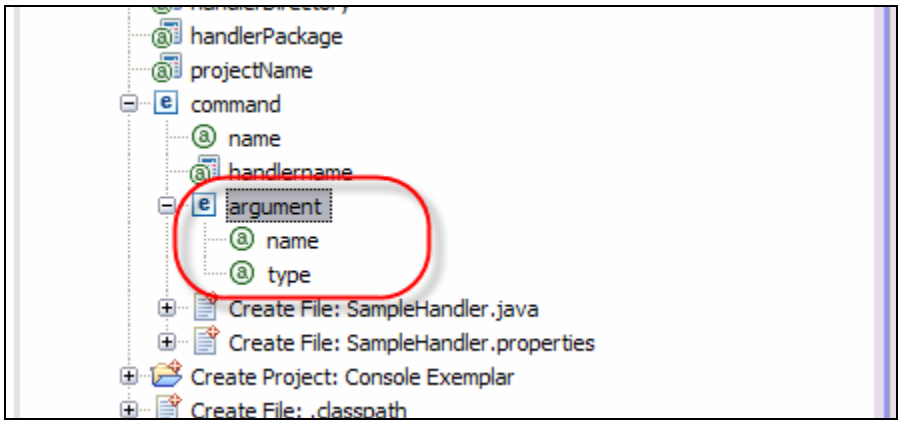


Figure 5 - 45: Adding argument model type

14. After running the Update Project action again, you can continue to mark up the `SampleHandler.java.jet` template. In the process you determine that a new derived attribute needs to be created. Return to the Exemplar Authoring tool and add a new derived attribute named `varName` under the argument element:

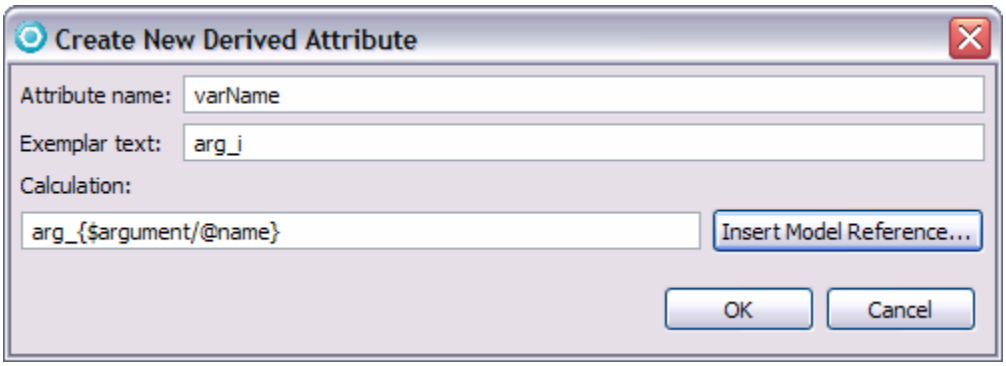


Figure 5 - 46: Defining the varName derived attribute

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

16. Run the Update Project action again.

17. Add in a variable that will be used for accessing the array of elements passed into the handler. Then add in a declaration for the local variables:

```
     */
    public void performCommand(String[] arg) {

<c:iterate select="$command/arg" var="arg">
        <c:get select="$arg/@type" /> arg_<c:get select="$arg/@name" />;
</c:iterate>
<c:setVariable select=" 1 " var="index" />
        try {
<c:userRegion>
    //
```

Figure 5 - 47: Defining a counter and the local variable

18. Mark up the section of code that converts the string arguments into the correct types:

```
     */
    public void performCommand(String[] arg) {

<c:iterate select="$command/arg" var="arg">
        <c:get select="$arg/@type" /> arg_<c:get select="$arg/@name" />;
</c:iterate>
<c:setVariable select=" 1 " var="index" />
        try {
<c:userRegion>
    //
    // Start of business logic
<c:initialCode>
    //
<c:iterate select="$command/arg" var="arg">
<c:choose select="$arg/@type">
<c:when test="'String'">
            <c:get select="$arg/@varName" /> = arg[<c:get select="$index" />];<c:setVariable select=" $index + 1 " var="index" />
</c:when>
<c:when test="'int'">
            <c:get select="$arg/@varName" /> = Integer.parseInt(arg[<c:get select="$index" />]);<c:setVariable select=" $index + 1 " var="index" />
</c:when>
<c:when test="'float'">
            <c:get select="$arg/@varName" /> = Float.parseFloat(arg[<c:get select="$index" />]);<c:setVariable select=" $index + 1 " var="index" />
</c:when>
<c:when test="'boolean'">
            <c:get select="$arg/@varName" /> = Boolean.parseBoolean(arg[<c:get select="$index" />]);<c:setVariable select=" $index + 1 " var="index" />
</c:when>
<c:otherwise>
    <c:log severity="error"><c:get select="$arg/@type"/> not known</c:log>
</c:otherwise>
</c:choose>
</c:iterate>
        } catch (NumberFormatException e) {
            System.out.println("Bad argument: "+e);
            return;
        }

<c:iterate select="$command/arg" var="arg">
        System.out.println("<c:get select="$arg/@type" /> <c:get select="$arg/@name" /> "+arg_<c:get select="$arg/@name" />);
</c:iterate>
    //
</c:initialCode>
    // End of business logic
    //
</c:userRegion>
    }

}
```

Figure 5 - 48: Final Markup

19. Select **File** > **Save All**.

20. Update the sample.xml file to include the following:

```
<root>
  <console name="Fred"  package="org.fred.test">
        <command name="multiply" help="multiplies two numbers">
            <arg name="op1" type="int" />
              <arg name="op2" type="int"  />
        </command>
        <command name="log" help="logs a message">
            <arg name="severe" type="boolean" />
              <arg name="message" type="String"  />
        </command>
        <command name="paint" help="paints a portion of the screen">
            <arg name="length" type="float" />
              <arg name="width" type="float"  />
              <arg name="color" type="String"  />
        </command>
  </console>
</root>
```

21. Select **File** > **Save All**.

22. Review and Test.

Lab 6.1: Introduction to EMF

Create EMF Model and Editor for Console Transformation Input File

Objectives

After completing this lab, you will be able to:

- Import an XML Schema file into EMF.
- Generate EMF Framework based code.
- Create an EMF based Editor which acts as a front-end to a JET transformation

Given

This lab has no inputs.

Scenario

In this lab, you will create an EMF based API for the input for the Console Transformation example. You will also create an automatically generated non-graphical editor for Console Transformation input files.

Task 1: Create and Prepare the Workspace

In this task you make sure switch to and prepare a new Workspace.

1. Open Rational Software Architect with a new workspace for this lab, such as `C:\EMF Lab Workspace`.
2. Open the Preferences Window, select menu **Window > Preferences**. Expand the **General** option and select **Capabilities**. Find **Eclipse Developer**, **Developer**, or **Development** in the **Capabilities** list and make sure that the checkbox is selected. If the checkbox is empty or is filled in with a square, click it until you see a check mark. This enables all of the Eclipse Developer capabilities, which includes EMF.

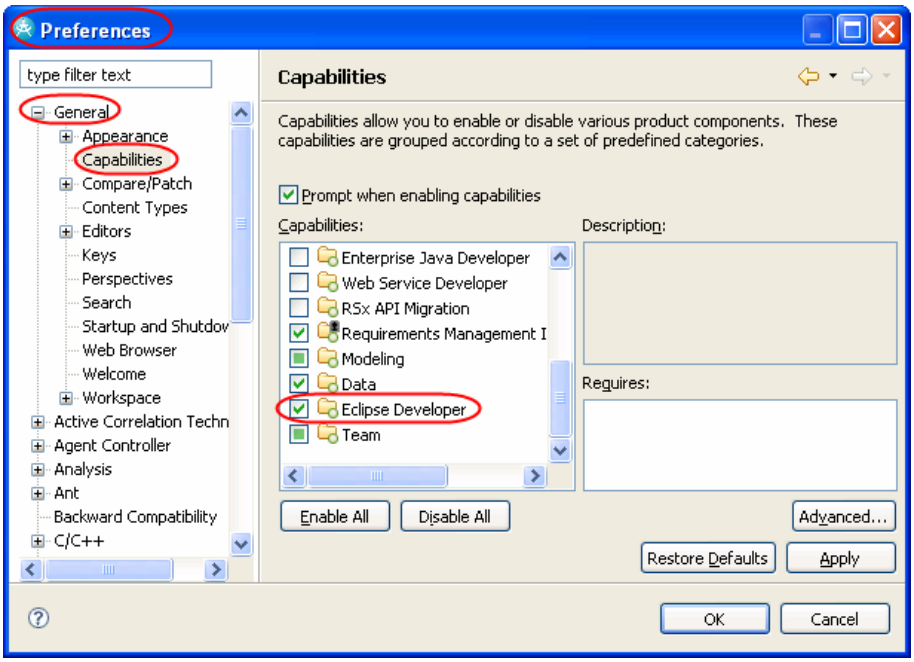


Figure 6.1-1: *Enabling the Eclipse Developer capabilities*

3. Click **OK** when you are done.
4. Import the project called `lab.console.transform` from the Project Interchange file `LabConsoleTransformPI.zip`.

Task 2: Create an EMF Project

In this task you use the generated input file format from the Console Transformation to create an EMF model of the input file. Specifically, an ECore file named `input.ecore` and an XML Schema Definition file named `schema.xsd` both describe the input file format. For this lab, you will actually use `schema.xsd`.

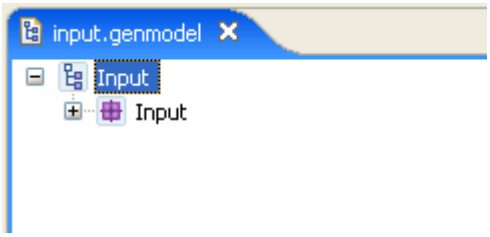
1. Open the project `lab.console.transform`. Make a copy of `schema.xsd` named **input.xsd**. The name of the EMF project files are based on the name of the schema file.
2. Right-click `input.xsd` and select **New > Project**. Select the project type of **EMF Project** and click **Next**.
3. Name the project **lab.console.input** and click **Next**.
4. Select **XML Schema** for the Importer and click **Next**.
5. The `input.xsd` file should already be entered into the **Model URI** text field. Click the **Load** button next to it and then **Next**.
6. Click **Finish** on the final page of the project wizard.

A new project named `lab.console.input` is created. The file `model/Input.ecore` contains the EMF Data Schema and the file `model/input.genmodel` contains the code generation settings. Review both files.

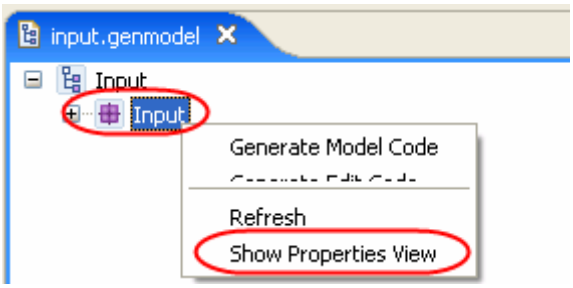
Task 3: Modify Code Generation Settings and Generate Code

In this task, you will fine-tune the code generation settings and generate the code.

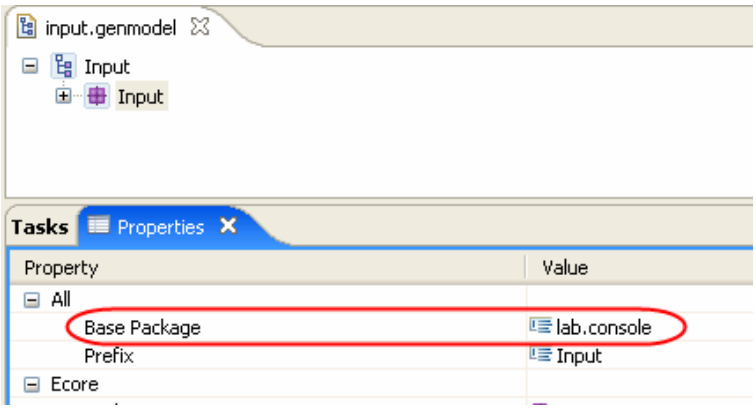
1. Make sure that the file `input.genmodel` is open. You should see an editor like the one pictured below. If you just see a text file, go back to Task 1 and make sure that your workspace has **Eclipse Development** (or just **Development**) capabilities turned on.



2. Right-click the nested **Input** node and select **Show Properties View**.



3. In the Properties view, go to the top of the list of properties, find the property named **Base Package** in the **All** section, and change it to `lab.console`. For the code that is generated, that is the prefix that will be used for all new projects and packages.

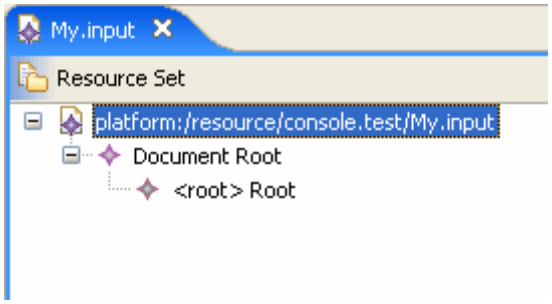


4. Right-click anywhere in the input.genmodel editor and click **Generate All**. That adds the input model API code to the current project (`lab.console.input`). It creates the following new projects: `lab.console.input.edit`, `lab.console.input.editor` and `lab.console.input.test`. `lab.console.input.editor` is a fully functional non-graphical editor.

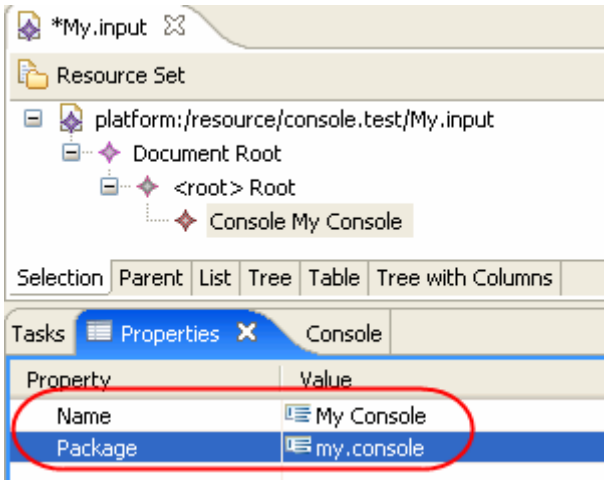
Task 4: Test the Generated Editor

In this task, you will test the generated editor.

1. In Navigator or Package Explorer, right-click the project named `lab.console.input.editor` and select **Run As > Eclipse Application**. Then wait for the run-time instance of the workbench to launch.
2. In the run-time workbench, create a simple project named `console.test`.
3. Right-click the new project name and select **New > Other**. Select the Input Model wizard and click **Next**.
4. Accept the default of `My.input` and click **Finish**.
5. `My.input` should be opened in an editor that looks like the following.



6. Expand the nodes as shown, right-click **<root> Root** and select **New Child > Console**. That creates a new **Console** entry in the XML file.
7. Right-click the new **Console** node and select **Show Properties View**. In the Properties view, enter `My Console` as the **Name** and `my.console` as the **Package**.



8. Right-click **Console** and click **New Child > Command**. Name the new Command `echo`.
9. Right-click the new Command `echo` and click **New Child > Arg**. Enter the **Name** of `arg0` and **Type** of `String`.
10. Enter any additional Commands and Args that you want. You can even enter multiple consoles.
11. Save and close **My.input**.
12. It's easier to test the existing transformation if the file has an XML extension, so rename **My.input** to **My.input.xml**.

TIP: Right-click `My.input` and click **Refactor > Rename**. Also note that after the file is renamed the generated editor is no longer applicable.

13. Right-click **My.input.xml** and click **Run As > Input for JET Transformation**. In the Properties page that appears, select `lab.console.transform` as the **ID**. Then click **OK** to run the transformation.
14. The project `My Console Console` (and any other consoles in your file) are generated.
15. Review the generated code.
16. Close the run-time workbench by selecting **File > Exit**.

Lab 6.2: EMF Optional Lab

Create Organization Chart Model

Objectives

After completing this lab, you will be able to:

- Define an object model using the EMF framework.
- Generate EMF Framework based code.
- Use an outline-based text editor to enter and manage data based on your object model saving the results to an XML file.

Given

This lab has no inputs.

Scenario

This lab creates a simple model of an Organizational Chart from scratch. This lab also creates a simple non-graphical editor.

Task 1: Make sure that EMF Capabilities are turned on

In this task, you make sure that EMF Capabilities are turned on in the Rational Software Architect Workspace that you are using for this lab.

1. Open Rational Software Architect with a new workspace for this lab, such as `c:\EMF Lab Workspace`.
2. Open the Preferences window, select menu **Window > Preferences**. Expand the **General** option and select **Capabilities**. Find **Eclipse Developer**, **Developer** or **Development** in the Capabilities list and make sure that the checkbox is selected. If the checkbox is empty or is filled in with a square, click it until you see a check mark. This enables all of the Eclipse Developer capabilities, which includes EMF.

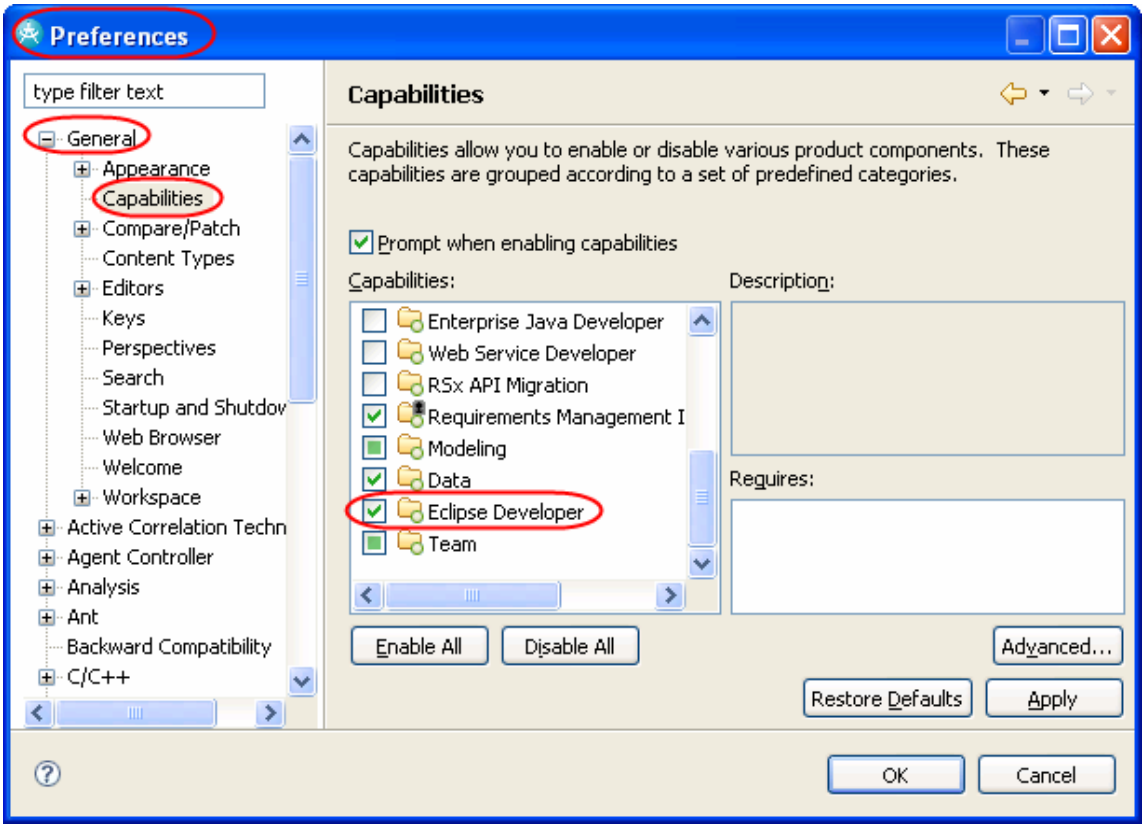


Figure 6.2-1: *Enabling the Eclipse Developer capabilities*

3. Click **OK** when you are done.

Task 2: Create an empty EMF Project

In this task you create an empty EMF project.

1. Select **File > New > Project**.
2. In the **New Project** wizard, type `emf` in the entry field on the top of the window. That will show all of the project types that have EMF in their name. Then select **Empty EMF Project** and click **Next**.

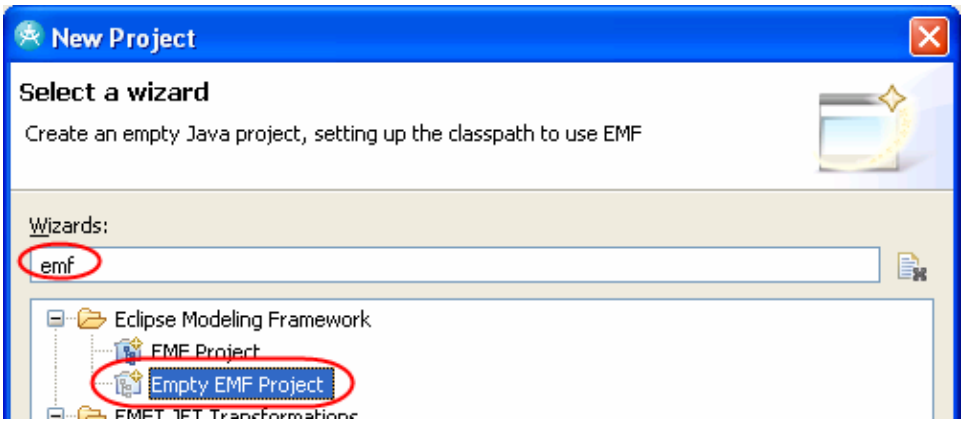


Figure 6.2-2: *Creating an empty EMF Project*

3. For the name of the project enter `com.tutorial.orgchart`. Then click **Finish**.

Task 3: Create and initialize orgchart.ecore

Ecore is the file format and extension for defining EMF-based data structures. In this task, you create an ecore file for the Orgchart definition.

1. Expand the project (in Navigator or Project Explorer), right-click the model directory and click **New > Other**.
2. In the wizard dialog, enter `ecore` in the topmost edit field. Then double-click **Ecore Model** from the list.

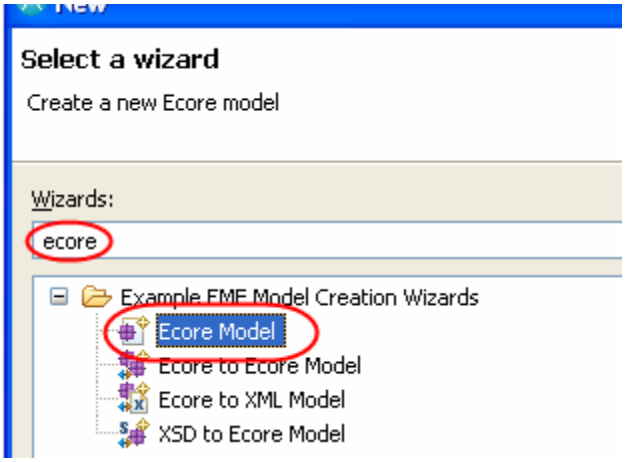


Figure 6.2-3: *Creating an Ecore Model*

3. Name the file `orgchart.ecore` and click **Finish**.
4. The new ecore file is automatically opened with an Ecore Model Editor, which displays the contents of the file in a tree structure. Expand the root level node. Under that you will find a node labeled **null**. Right-click that **null** node and select **Show Properties View**.

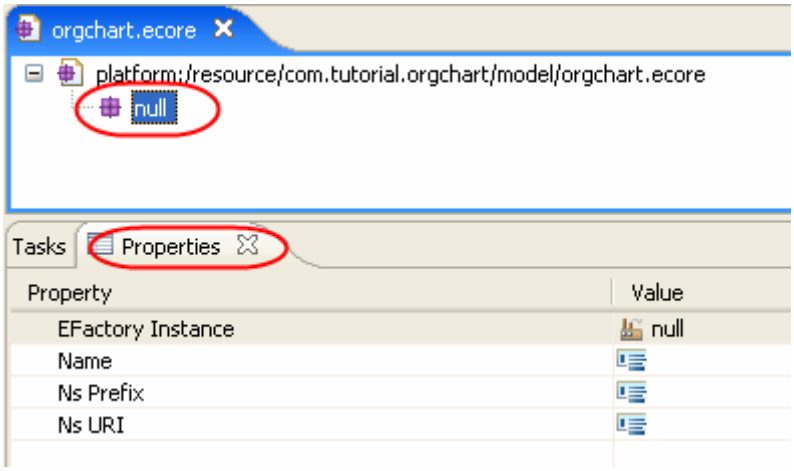


Figure 6.2-4: *Viewing the properties for the null node*

5. In the Properties view, set **Name** to `orgchart`, **Ns Prefix** to `oc`, and **Ns URI** to `com.tutorial.orgchart`.

TIP: Note that **Ns Prefix** is the namespace prefix used in XML files used to store orgchart data, and **Ns URI** is the unique namespace URI for the orgchart data. In this example, you are simply using the project name as the URI, but it does not have to be the same.

Task 4: Define the data structures

Now it is time to define the structure of the orgchart data.

1. When you work with the resulting Org Chart data, you want to be able to store an Org Chart in a single XML file. The simplest way to do that is to define a class in the ecore file that corresponds to the contents of the XML file. In the Ecore Editor for `orgchart.ecore`, right-click the `orgchart` package and click **New Child** > **EClass**.

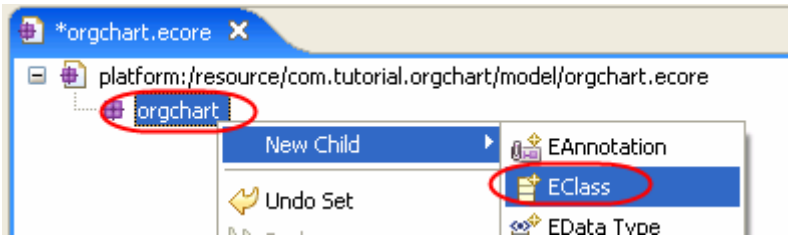


Figure 6.2-5: *Adding a child*

2. In the Properties view for the new class, enter `OrgChart` as the **Name** of the class. This is the class that corresponds to one OrgChart (and its corresponding data file).

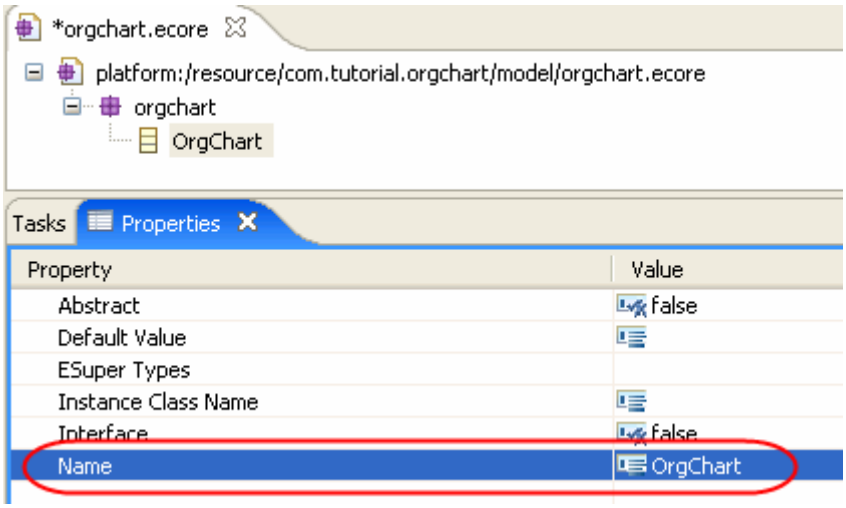


Figure 6.2-6: *Specifying a name*

3. Add two more classes to the orgchart package the same way: `Employee` and `Department`. You will keep track of employee and department information in the org charts.

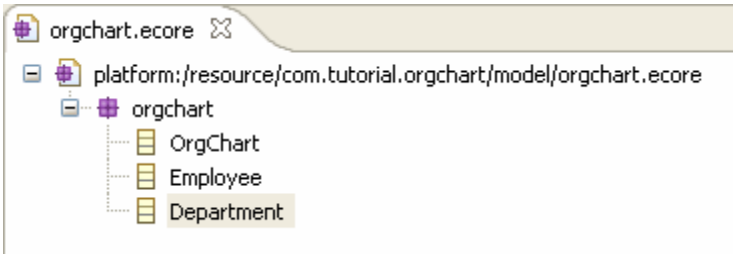


Figure 6.2-7: *View after adding two additional classes*

4. You need to specify that Employee classes will be stored in an OrgChart (in the same file). To do that, create a containment relationship from OrgChart to Employee. Right-click **OrgChart** in the tree and select **New Child > EReference**. In the Properties, set **Containment** to `true`, **EType** to `Employee`, **Name** to `employees`, and **Upper Bound** to `-1`. The other default values should be OK. **Containment** of `true` indicates that this is a containment relationship. An **Upper Bound** of `-1` indicates that their can be any number of employees in an OrgChart.

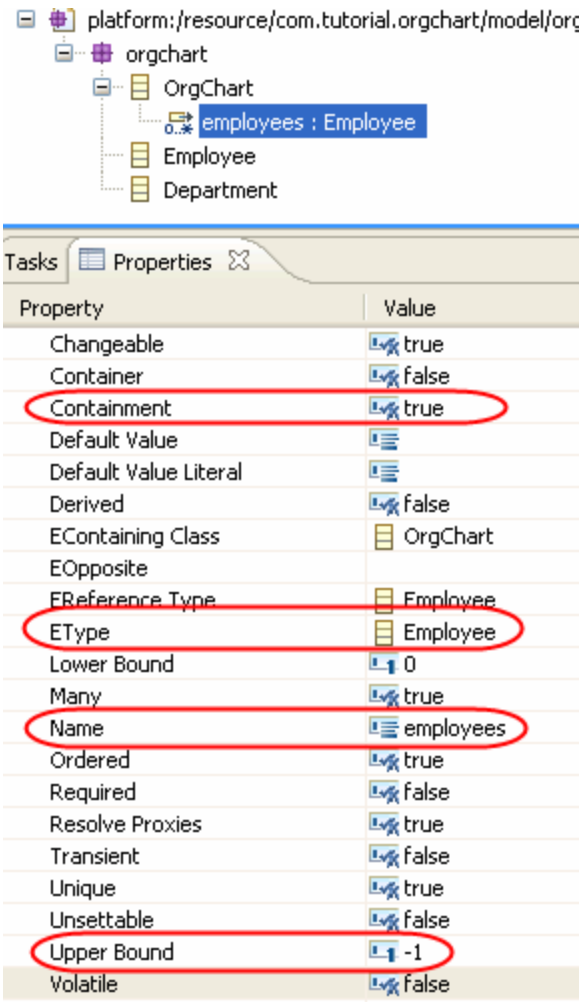


Figure 6.2-8: *Specifying properties for the containment relationship*

5. Likewise, add another containment relationship for **Departments**. Repeat the last steps, but this time set **EType** to `Department` and **Name** to `departments`.

6. Next, you will define the name field for Departments. In the tree, right-click the **Department** class and select **New Child > EAttribute**. In the Properties of the new Attribute set the **EType** to `EString <java.lang.String>` and the **Name** to `name`. Note that the type of the attribute is the EMF type EString. The additional text of <java.lang.String> is a reminder that the EString EMF data type corresponds to the java.lang.String Java type.

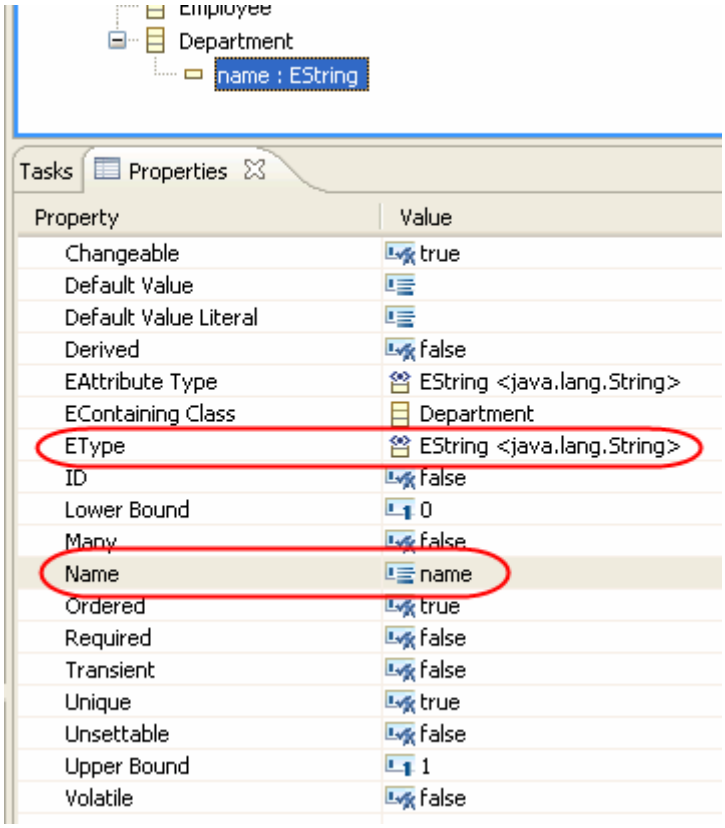


Figure 6.2-9: *Creating a name field for Departments*

7. Likewise, add the following attributes to Employee: **Name** of type **EString** and **jobTitle** of type **EString**.

8. Next, add a relationship from Department to Employee so that Departments can reference the multiple Employees that are in them. Note that this will NOT be a containment relationship. In EMF, containment relationships correspond to physical storage of related classes. Both Employees and Departments are already stored in the same Org Chart. Right-click the **Department** class and select **New Child > EReference**. In the **Properties**, set **EType** to `Employee`, **Name** to `members` and **Upper Bound** to `-1`, since a Department can have any number of employees.

9. Next, add a relationship from Employee to Employee to indicate which other employees are being managed. Right-click the **Employee** class and select **New Child > EReference**. In the **Properties**, set **EType** to `Employee`, **Name** to `manages` and **Upper Bound** to `-1`.

10. Your model is defined. Save the results by selecting **File > Save All**.

Task 5: Create the 'EMF Model' (orgchart.genmodel)

Orgchart.ecore now contains the definition of your Org Chart data model. Next, you need to create another file with an extension of genmodel. EMF refers to this file as the 'EMF Model'. The genmodel (or 'EMF Model') file contains all of the additional information and settings needed to generate Java source files that correspond to the model. Genmodel files maintain a link to their corresponding ecore file.

1. Make sure that the current contents of the ecore file are saved.

2. In the **Navigator** or **Package Explorer** view, right-click the file orgchart.ecore, which is located in the model directory of the `com.tutorial.orgchart` project. From the pop-up menu select **New > Other**.

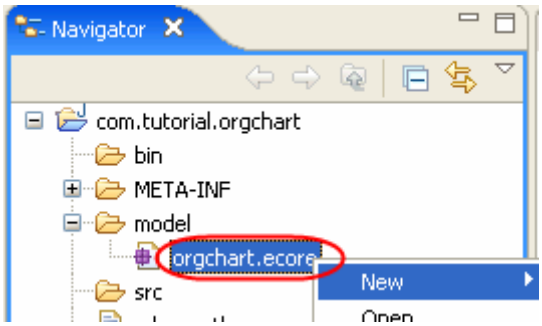


Figure 6.2-10: *Launching the New wizard*

3. Type `EMF` in the new wizard's text field, select **EMF** Model, and click **Next**.

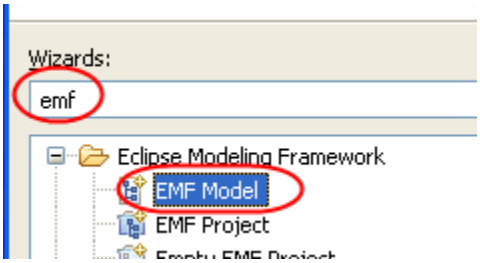


Figure 6.2-11: *Selecting the EMF Model*

4. The name of the file should already be set to `orchart.genmodel` in the model directory of the `com.tutorial.orgchart` project. Correct it if it isn't. Click **Next**.
5. Select **Ecore model** as the **Model Importer** and click **Next**.
6. Make sure that the `orgchart.ecore` file is selected as the Model URI (as shown below). Then click the **Load** button next to the text box. That actually loads the definition from the ecore file. Then click **Next**.

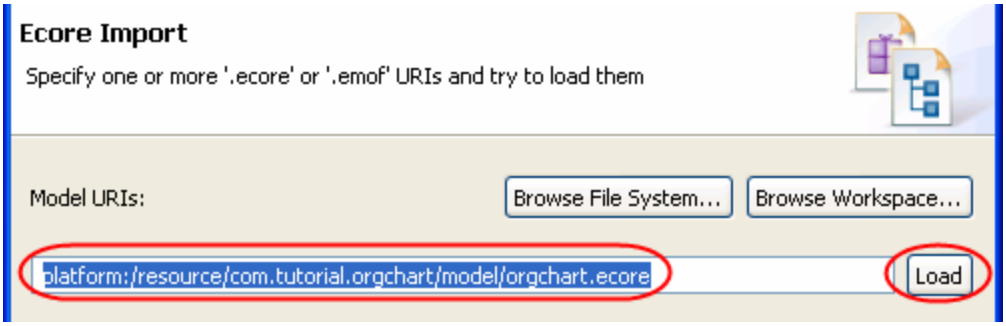


Figure 6.2-12: *Load the definition from the ecore file*

7. The checkbox next to **orgchart** should be selected. Click **Finish**.
8. The new file `orgchart.genmodel` is created and automatically opened. It contains numerous options for controlling how Java code is created that corresponds to the ecore definition.
9. Right now, you will only make one change. You want to generate source code for the orgchart in the package `com.tutorial.orgchart`. Because you defined a package called `orgchart` in ecore, right now, the default output java package is just `orgchart`. You need to define a package prefix which is called the **base package**. In the genmodel editor, expand the root node and select the nested orgchart package node. In the properties change the property **Base Package** to `com.tutorial`. That prefixes com.tutorial in front of orgchart in the generated Java files.

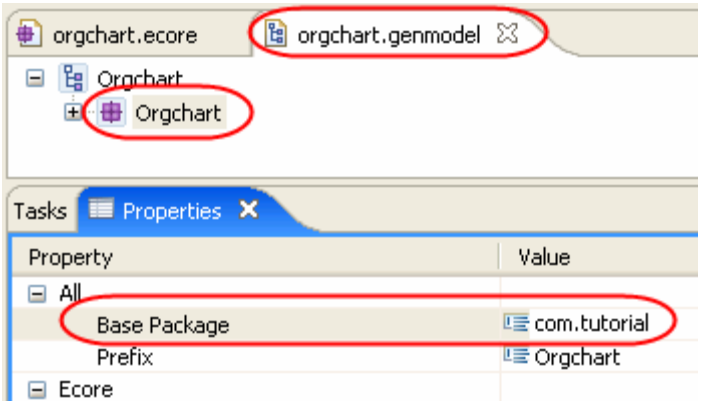


Figure 6.2-13: *Specifying the Base Package*

10. Save the genmodel file.

Task 6: Generate the runtime Java code

Next, you need to generate the custom Java code that implements your model.

1. In the genmodel editor, right-click anywhere in the editor and select **Generate Model Code**. That adds the Java code and plug-in definition information to the current project.

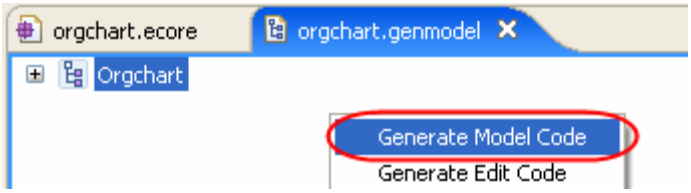


Figure 6.2-14: *Generating the Model Code*

2. Review the files in the **Package Explorer**. The circled files and packages were added as a result of generating the Model Code.

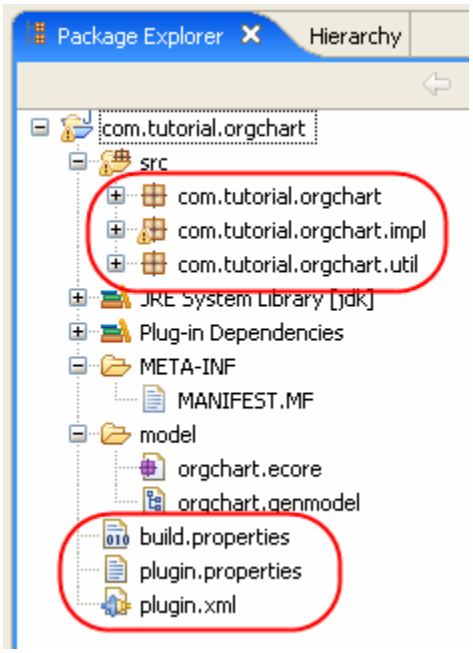


Figure 6.2-15: *The files that were generated*

3. In the genmodel editor, right-click anywhere and select **Generate Edit Code**. This creates a brand new plug-in project called `com.tutorial.orgchart.edit`. The edit project contains model specific utility classes. In particular, it is used by the editor code (see the next step).

4. Likewise, **Generate Editor Code** which creates a new plug-in project called `com.tutorial.orgchart.editor`. The editor is a non-graphical editor for working with orgchart data files (which are currently defined as XML).

Task 7: Generate the runtime Java code

You now have the source code for a fully functional non-graphical OrgChart Eclipse Editor.

1. In the **Package Explorer** view in the **Java** (or **Plug-in Development**) Perspectives (**Window > Open Perspective >** , right-click `com.tutorial.orgchart.editor` and click **Run As > Eclipse Application**. That will launch a run-time instance of the workbench. with an active OrgChart editor plug-in.

2. In the run-time workbench, close the Welcome screen (if it is open). Select **File > New > Project**. In the New Project wizard, select **General > Project** (which is a simple, general purpose project).. Click **Next**, name the project `Test OrgChart`, and click **Finish**.

3. In **Navigator** (or **Package Explorer**), right-click the **Test OrgChart** project and click **New > Other** . Then type in `Org` in the **Filter** text box, select **Orgchart Model** and click **Next**.

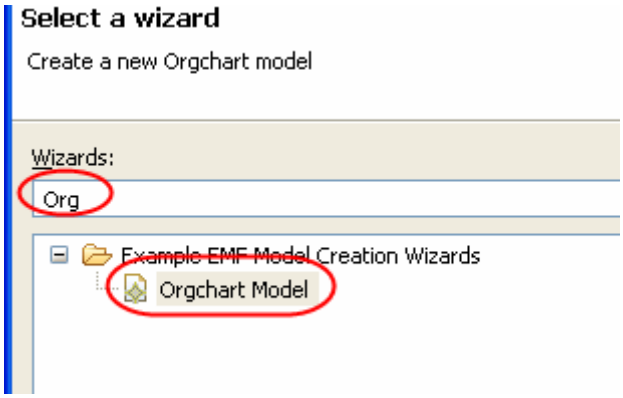


Figure 6.2-16: *Selecting the Orgchart Model*

4. Name it `test.orgchart` and click **Next**.
5. Select **Org Chart** as the model object and click **Finish**.
6. The new Org Chart is automatically opened up in your (non-graphical) custom editor. Expand the root node so that you can see the nested Org Chart object. You can right-click it and add Employees and Departments.
7. This is one sample test scenario:
   a. Add the following employees: Pat S, John D, Susan R, Bill C, Fred M, and Betty A. Set their job titles to anything that you want. Remember that you need to go to the **Properties** view to edit names and Job Titles. One way is to right-click a class object and select **Show Properties View**.
   b. Specify that Pat S manages John D and Susan R. To do that, go to the properties for Pat S. Click the ‘’ button next to **Manages**. Select John D and click **Add**, and then select Susan R and click **Add**.
   c. Likewise, John D manages Bill C, Fred M, and Betty A.
   d. Add a Department called ‘Information Services’ and add John D, Bill C, Fred M, and Betty A to it.
   e. You should see something like the following screen. Note that you see the properties for John.

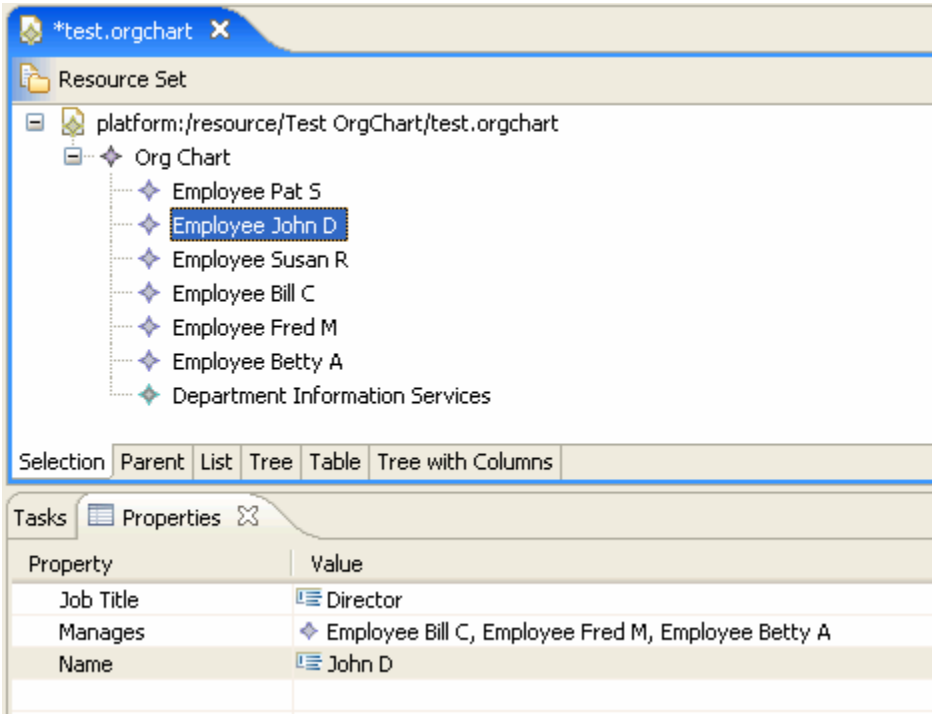


Figure 6.2-17: *The resulting org chart*

8. Save the current orgchart, select **File > Save**.

9. The file **test.orgchart** is an XML file. Let's take a quick look at the contents. Right-click the file **test.orgchart** and select **Open With > Text Editor**. You should see the contents of the XML file.

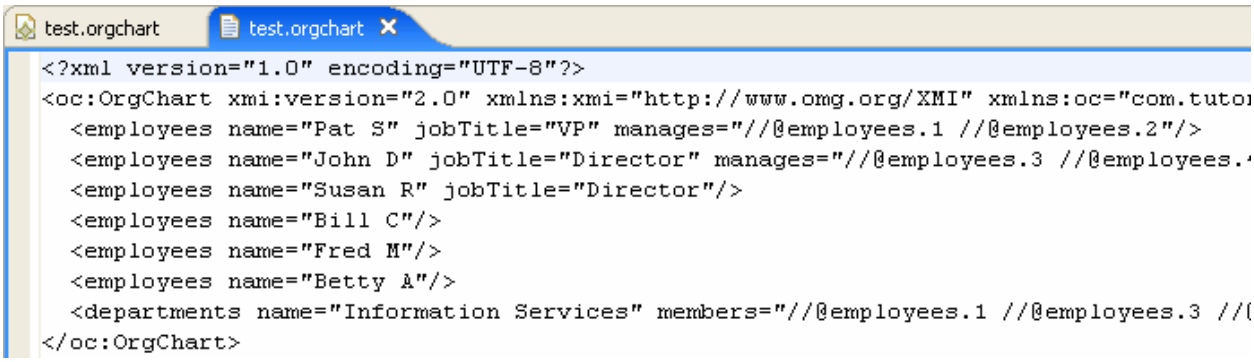


Figure 6.2-18: *Viewing the xml source for the org chart*

10. Close the run-time instance of the workbench when you are done.

Task 8: Generate the runtime Java code

The default display label for employees is "Employee," followed by their name. In this optional task, you will change it to their job title followed by their name.

To do this, you will modify some of the generated code, but also indicate that you want to save the custom changes so that it is preserved the next time(s) that code is generated.

1. Open either the **Java** or **Plug-in Development** perspective.

2. Open the file `com.tutorial.orgchart.edit/src/com.tutorial.orgchart.provider/EmployeeItemProvider(.java)'.`

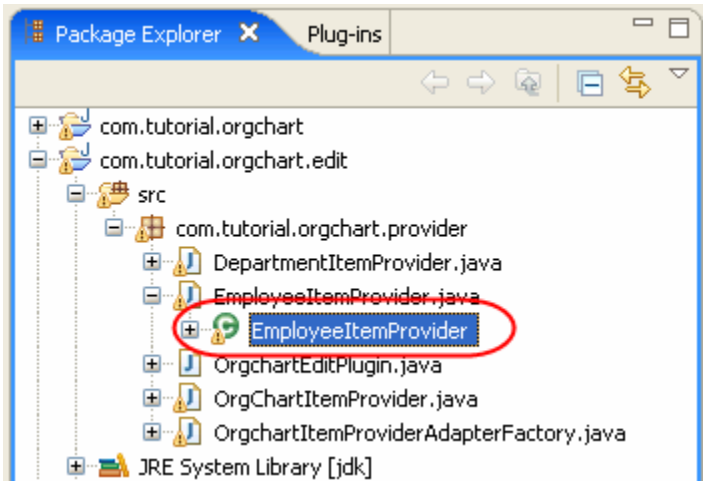


Figure 6.2-19: *The EmployeeItemProvider class in the Package Explorer*

3. Go to the function `getText(Object object)`. There is a javadoc tag `@generated` in the comments for `getText()`. That is a flag that this function was automatically generated, and will be overwritten if you generate code again. Change it to anything else or delete it to take manual ownership of the function. In this example, change it to `@not-generated`. If you use a consistent naming guideline, then you can quickly find all of the functions that you are manually maintaining.

4. Change the body of the function to the following:

```
public String getText(Object object) {
          String jobTitle = ((Employee)object).getJobTitle();
          String name = ((Employee)object).getName();
          String retval = getString("_UI_Employee_type"); // generic label
          if (jobTitle != null) {
                retval = jobTitle;
          }
          if (name != null) {
                retval = retval + " " + name;
          }
          return retval;
    }
```

This `getText` function returns the display label for any employee. This new version uses their job title, if it is available.

5. Run and test the results again as described above. Note that there is no need to regenerate the code. You should now see something like the following.

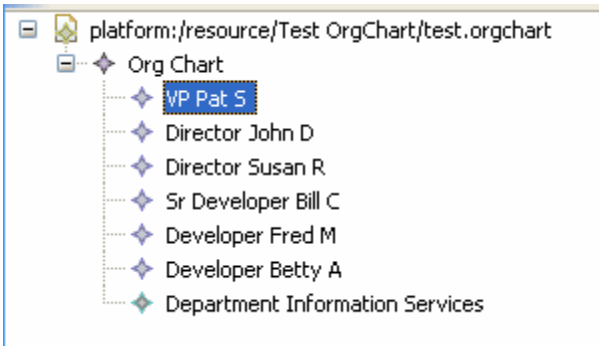


Figure 6.2-20: *The updated view of your Org Chart*

Lab 7 – Customize a Transformation

Objectives

After completing this lab, you will be able to:

- Apply a transformation.
- Customize the transformation to configure the location of the generated code.

Given

No lab artifacts are provided for this lab.

Scenario

In this lab, you will create a new workspace so that you will have a clean area in which to perform your development. Next you will create projects that will be used by the UML-to-Java transformation to generate Java™ classes from UML model elements.

- The first project will be the source project that will be populated with the UML modeling elements.
- The second project will be the target project that will contain the Java classes that are a result of applying the standard IBM Rational Software Architect UML-to-Java transformation.

When the transformation is run, default names will be assigned to the files and folders it generates. Your team uses a naming convention so you will need to customize the transformation to comply with the naming convention. A mapping model will be used to implement your naming convention by specifying alternate names for the generated files and folders.

Task 1: Create the Workspace

In this task, you will switch to a new workspace named `CustomizeTransformationWorkspace` that you will create.

1. From the **File** menu, select **Switch Workspace**.
2. In the **Workspace Launcher** dialog, replace the displayed text with `C:\Workshop\StudentWork\CustomizeTransformationWorkspace` and click the **OK** button.
3. Close the **Welcome** screen.

Task 2: Create the Source and Target Projects

In this task, you will set up two new projects.

1. Create a new UML project named `TransformationModels` with a model named `Source Model`.
   a) On the File menu, select New > Project.
   b) Replace `type filter text` with `UML`

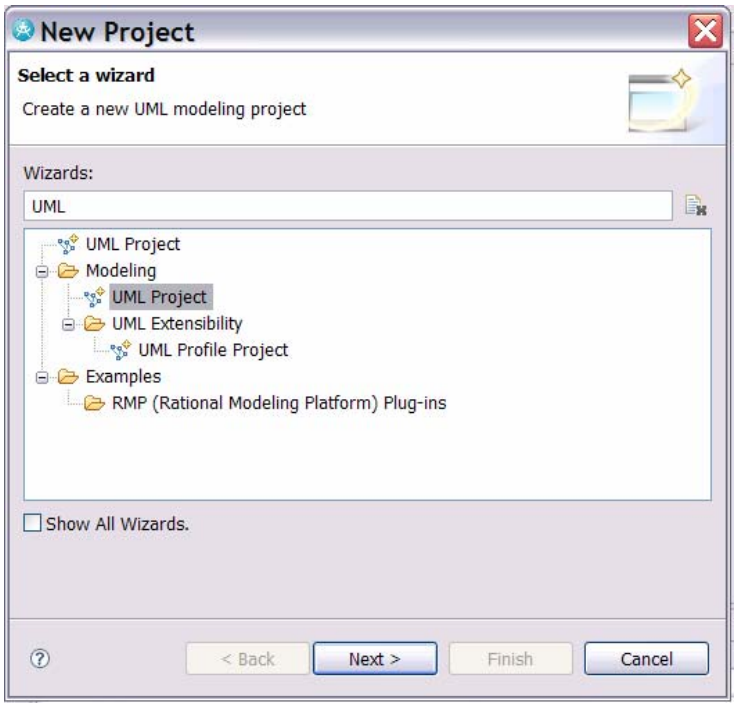


Figure 7-1: *Creating a New UML Project*

   c) Select **UML Project** and click **Next**.
   d) Name the project `TransformationModels` and click **Next**.
   e) Change the file name to `Source Model,` select the default diagram type as **Class Diagram** and click **Finish**.
   f) If asked to switch modeling perspectives, click **Yes**.

2. Create a new Java project named TransformationTarget.

   a) On the **File** menu, select **New > Project.**

   b) In the New Project wizard, filter for and select **Java Project**. Click **Next**.

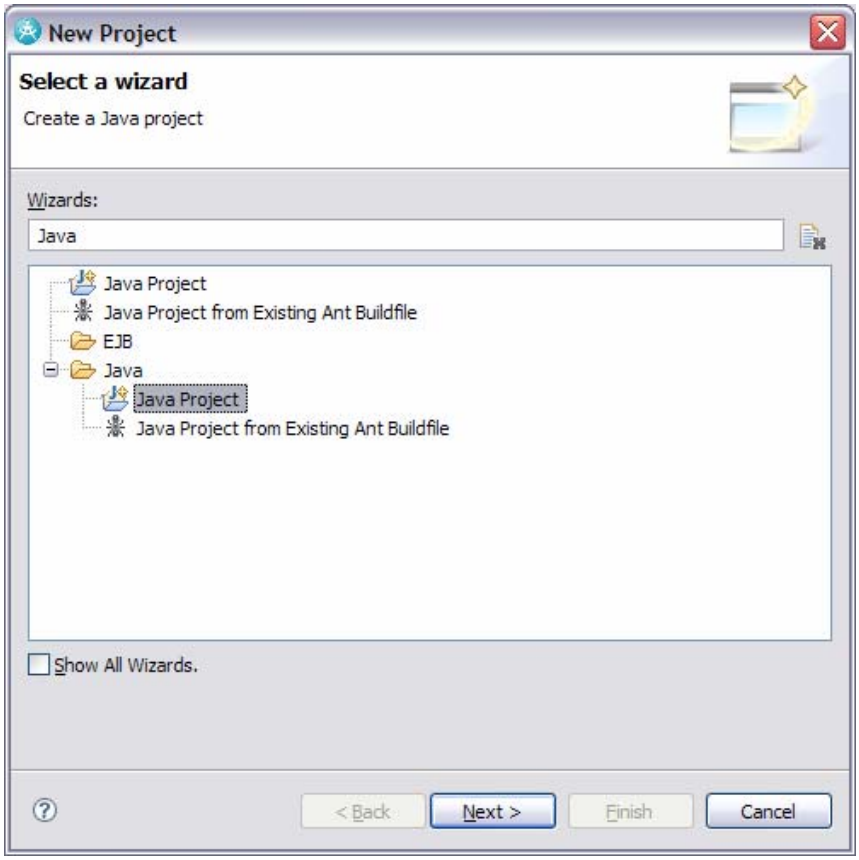


***Figure 7-2:** Creating a New Java Project*

   c) Name the project `TransformationTarget` and click **Finish**.

   d) If asked to enable the Java Development capability, click **Yes.**

   e) If asked to switch to the Java perspective, click **No.**

Task 3: Populate the Source Project

In this task, you will create UML modeling elements in the Source Model.

1. Open the `Main` diagram within `Source Model` and add two new classes named `Employee` and `Department` using the action bar on the diagram editor.

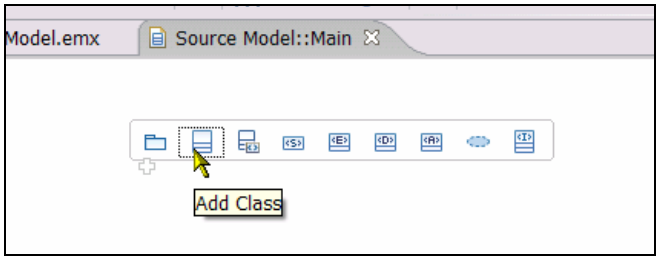


Figure 7-3: *Adding classes on a diagram using the action bar*

2. On the main diagram, use the action bar to add the following attributes and operations to the **Employee** class.

Attributes	Operations
• salary: float • id: String • name: String	• fire() • giveRaise(amount : float)

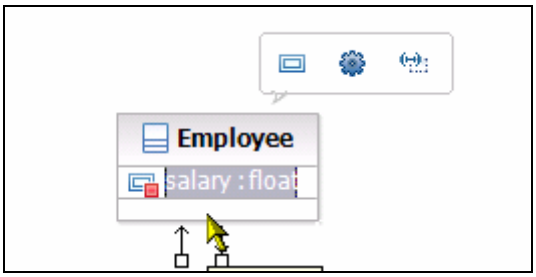


Figure 7-4: *Adding an attribute to a class using the action bar*

TIP: When you add the attribute to the class, you can immediately name it using the syntax `name: type`. A similar process can be followed for operations.

3. Right-click on the class and select **Filter-> Show Signature** to see operation parameters on the diagram.
4. On the main diagram, use the action bar to add the following attributes and operations to the `Department` class.

Attributes	Operations
• `id: String` • `budget: float` • `maxEmployees: int`	• `calculatePayRaises()`

5. On the Main Diagram, draw a directed association from `Department` to `Employee`.

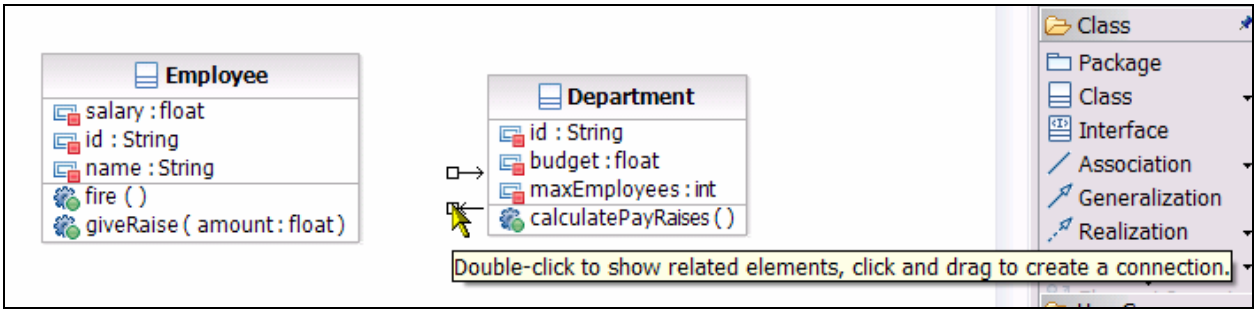


Figure 7-5: *Drawing an association using the diagram*

Task 4: Apply a UML-to-Java Transformation

You will now apply the standard out of the box transformation to generate some code.

1. Configure the transformation:
2. On the **Modeling** menu, click **Transform** > **New Configuration**.
3. Name the configuration `My UML to Java`.
   a) Select the UML to Java V 1.4 found within the IBM Rational Transformations folder.
   b) Set the configuration file destination to /TransformationModels
   c) Select Next.

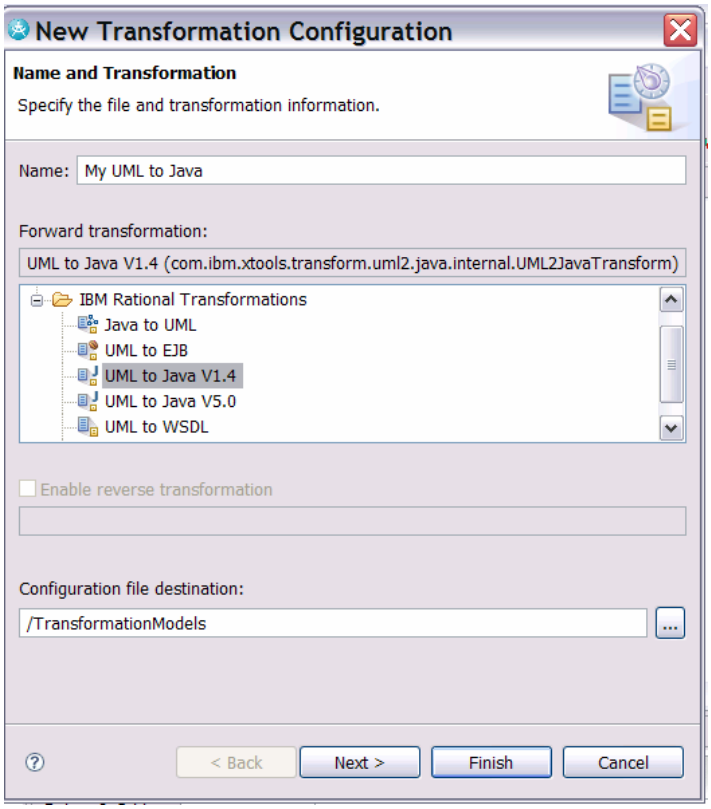


Figure 7-6: *Creating the Transformation Configuration*

   d) Open the `Models` folder and select the `Source Model` model as the **Select source**.
   e) Select `TransformationTarget` project as the **Selected target**.
   f) Click **Next.**

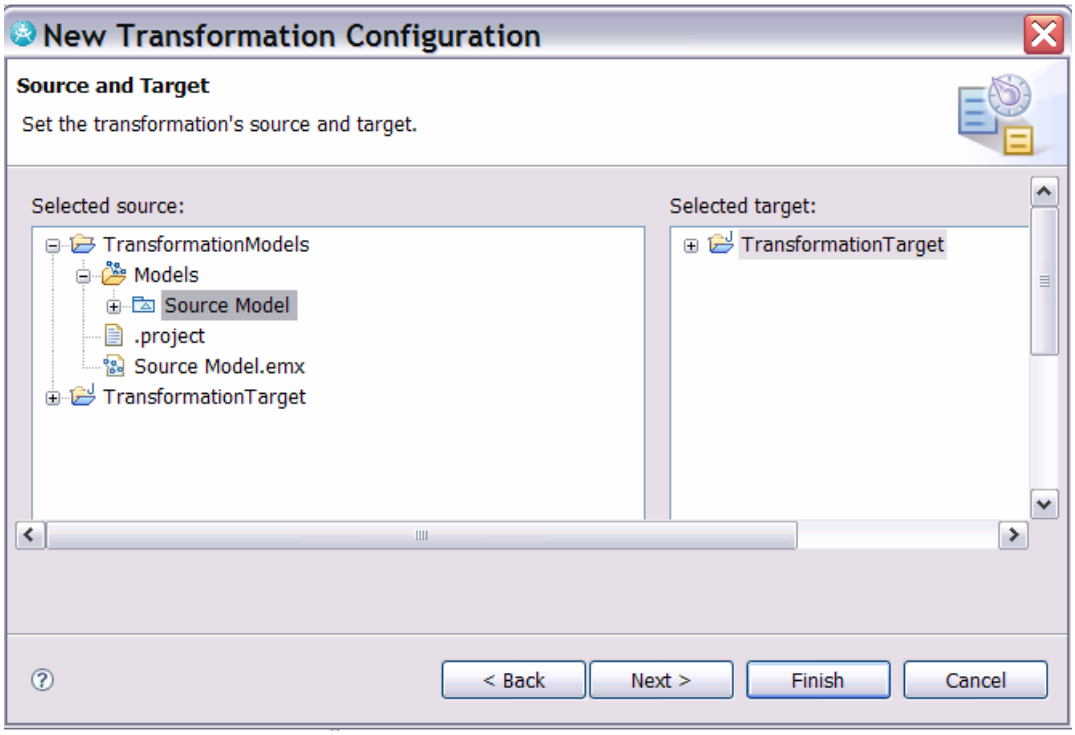


Figure 7-7: *Setting the source and target for a transformation*

g) Click **Next** through the next three screens, reviewing the available transformation options.

h) On the **Common** screen, and enable **Create source to target relationships** as the Transformation options.

i) Click **Finish.**

j) Locate the configuration file in the **Project Explorer.**

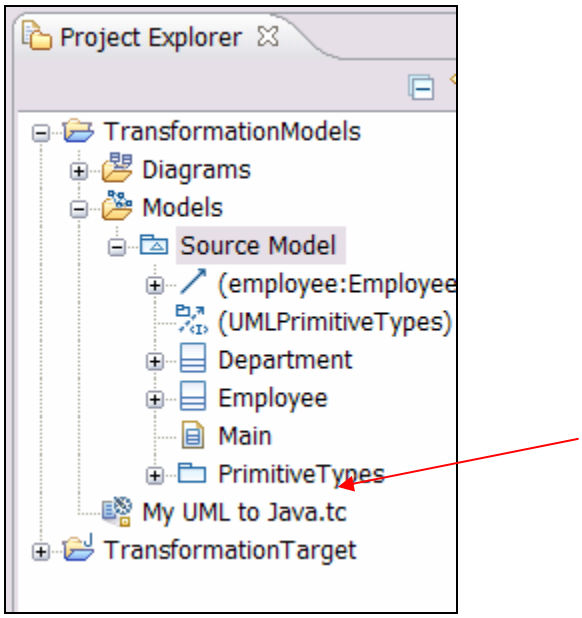


Figure 7-8: *The Configuration File in Project Explorer*

4. Run the transform.

a) In the Project Explorer, select `My UML to Java.tc`, right-click and choose **Transform > UML to Java V1.4**.

b) Drag the newly generated classes found in the `TransformationTarget` onto the `Main` diagram in the `Source Model`. If asked to enable java Modeling capability, click **OK.**

c) Select the «Java Class» Employee class, right-click and select **Filters >Show Type as Association** to show the employee attribute as an association relationship, not as an attribute.

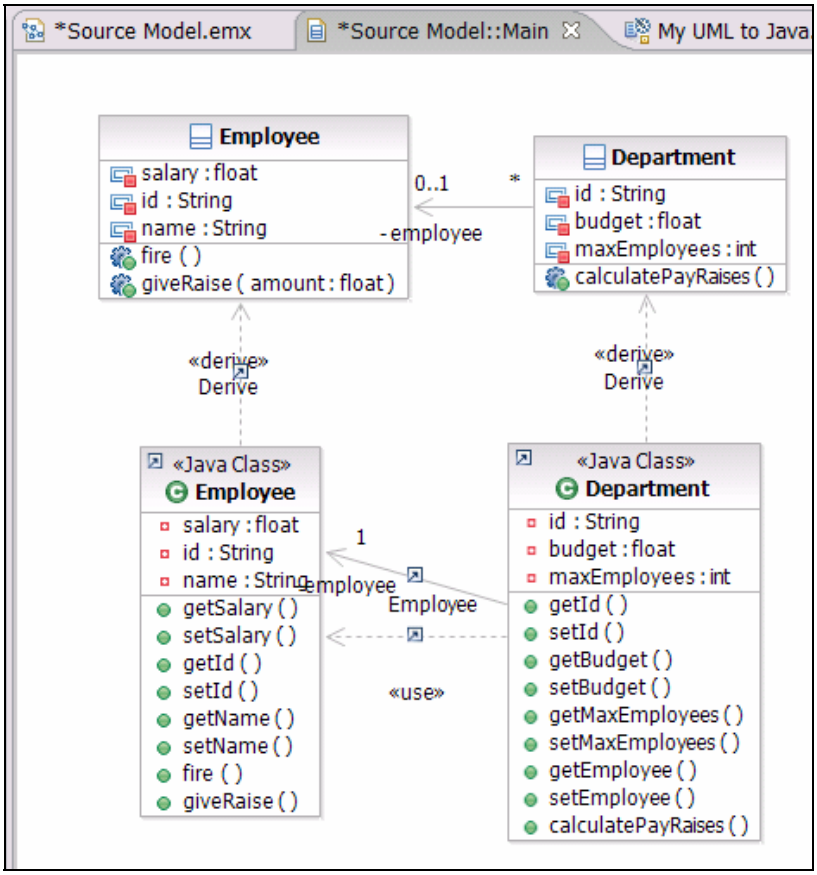


Figure 7-9: *Diagram of UML elements and generated Java classes*

d) Double-click on the «**Java Class**» **Employee** to view the generated code.

Task 5: Use a Mapping Model

Use a mapping model to change the names of the classes, and have them generate into specific locations inside the target model.

1. Setup the mapping model.

a) Double-click on the file `My UML to Java.tc` in the **Project Explorer** to open it in the editor view.

b) Select the **Mapping** tab.

c) On the **Mapping** tab, choose **Enable Mapping**, and click **New…**. Enter a filename of `JavaMappingModel.emx` and click **Save**.

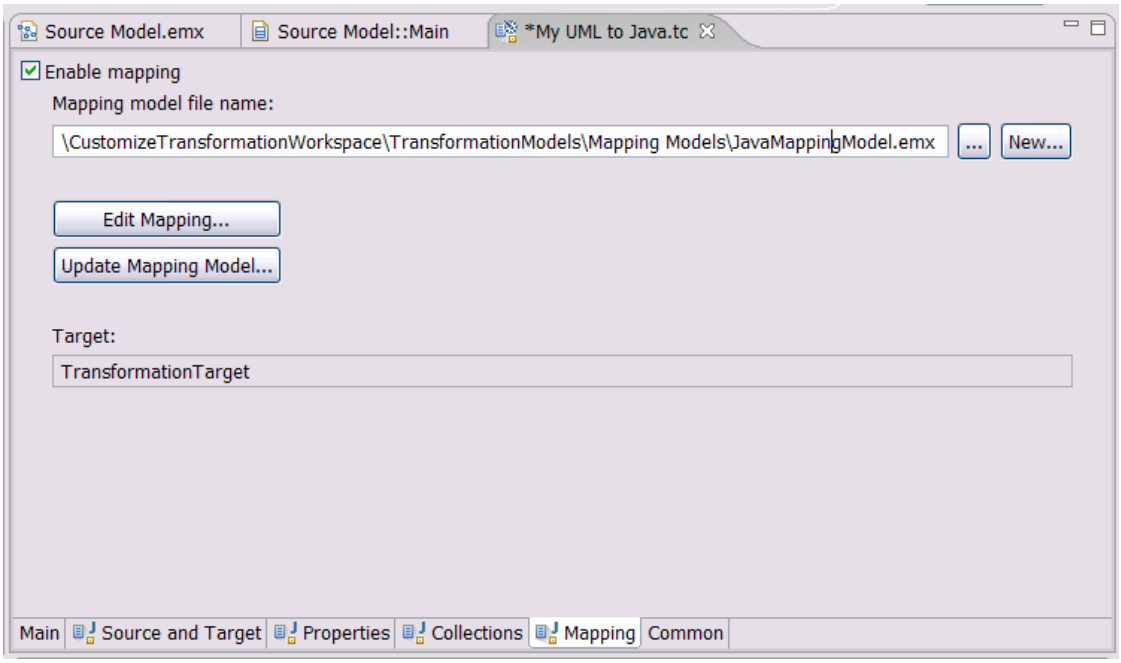


Figure 7-10: *Setting the mapping model to be created*

d) Click Edit Mapping….

e) Select the Department class, change its Mapped Name to be `com.ibm.rational.MyDepartment` and click Apply.

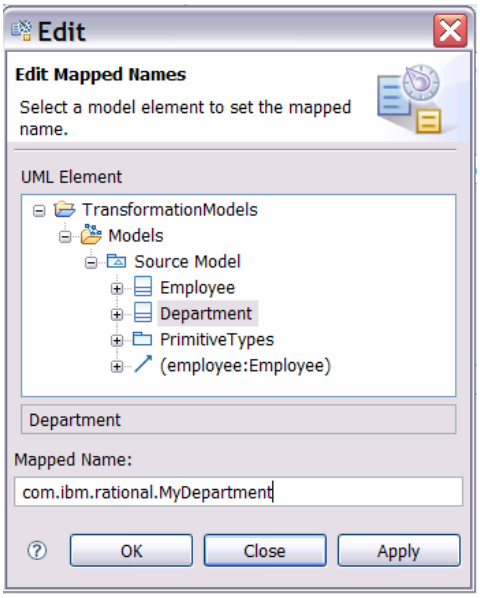


Figure 7-11: *Specifying an alternate name for the generated file*

f) Select the **Employee** class, change its Mapped Name to be `com.ibm.rational.employee.MyEmployee` and click **Apply**.

g) Click **OK**.

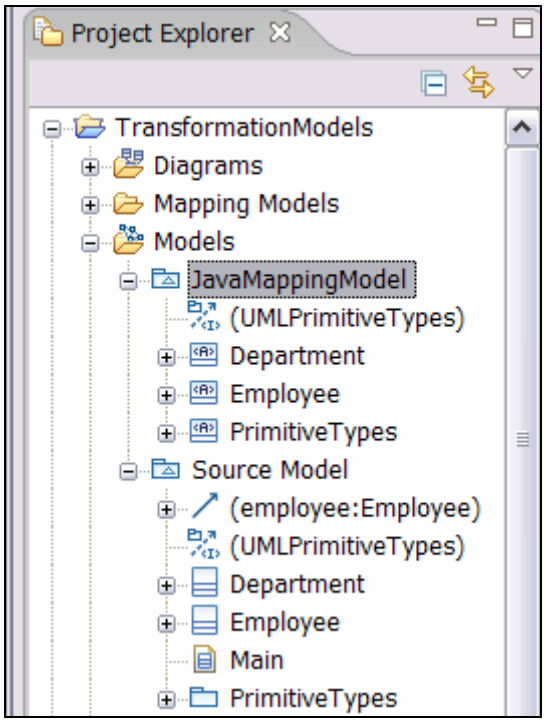


Figure 7-12: *Mapping model in the Model Explorer*

2. Delete the classes from the TransformationTarget project.
3. Rerun the My UML to Java configuration.
4. Observe where the classes get created in the target project.

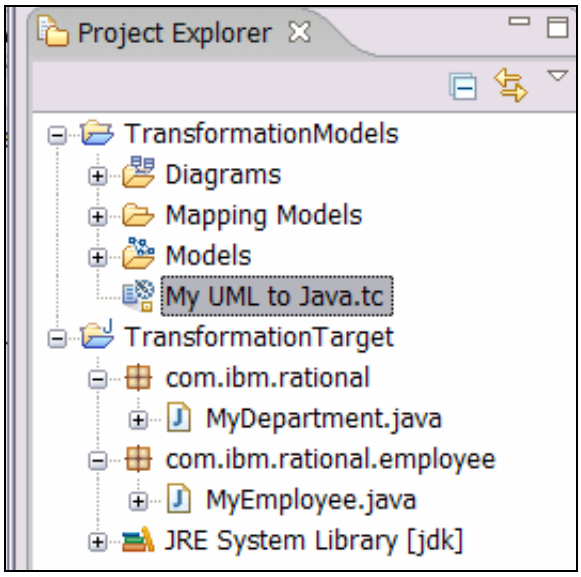


Figure 7-13: *Generated classes in the Model Explorer*

5. From the File menu, select Save All to save all the projects.

Lab 8 – Create a Model to JET2 Transformation

Objectives

After completing this lab, you will be able to:

- Author, run, and test a custom model-to-text transformation using a JET transformation previously created

Given

The following lab artifacts can be found in the `Inputs` folder for this lab:

- `NestedPackageContentsExtractor.java`
- `OwnedCommentToHelp.txt`
- `LabConsoleTransformPI.zip`
- `TestConsoleModel.zip`

Scenario

In this lab, you want to provide a graphical front end for defining the classes and operations that need to become console objects in your JET-implemented console generation transformation. This solution will allow business analysts to identify which functionality of a system needs to be supported with console operations and the resulting transformation will create the Java project with the solution. The business analyst will only have to make simple markups in the UML model of the system and will therefore not see the details of the XML syntax of the input to the JET transformation. Likewise, this saves a designer or developer from the tedious task of writing the same kind of console application over and over.

You will use the Transformation with Model mapping capabilities of Rational Software Architect to define how the source model elements will be mapped to the model that is used as input to the JET transformation. Then you will generate and run the transformation from this model mapping.

Task 1: Create and Prepare the Workspace

In this task, you will switch to a new workspace named `M2JET_TransformationWorkspace` that you will create.

1. From the **File** menu, select **Switch Workspace**.
2. You may use the workspace in which you previously created the `lab.console.transform` project by switching to that workspace and then skipping ahead to step 6.
3. In the Workspace Launcher dialog, replace the displayed text with `C:\Workshop\StudentWork\`

`M2JET_TransformationWorkspace` and click **OK**.

4. Close the Welcome screen.
5. From the `C:\Workshop\Labs\Inputs` folder in the project interchange file `LabConsoleTransformPI.zip`, import the project called `lab.console.transform`.
6. Switch to the Modeling perspective.
7. Make sure the **XML Developer** and **EMF Developer** capabilities are enabled. Go to **Window > Preferences** and under **General > Capabilities**, check **XML Developer**. Click the **Advanced** button and, under the **Eclipse Development** branch, select **Eclipse Modeling Framework**. Select **OK** twice to return to the workbench.

Task 2: 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.

1. On the **File** menu, click **New > Project**
2. Replace `type filter text` with `EMF` and select **EMF Project**, then click **Next**.

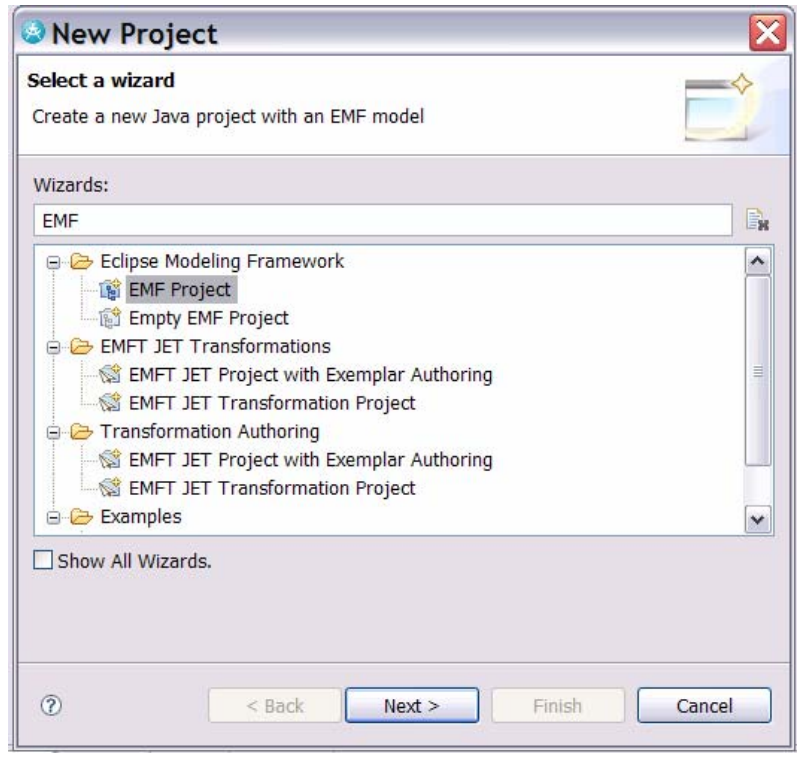


Figure 8-1: *Creating the EMF project*

3. Enter the project name `lab.console.transform.model`, then click **Next**.
4. Select Ecore model as the Model Importers, then click **Next**.

5. Click **Browse Workspace** to find the file `input.ecore` in the `lab.console.transform` project and select it. Click **OK** then Click **Next**.

TIP: The input.ecore file was created as part of the JET project creation.

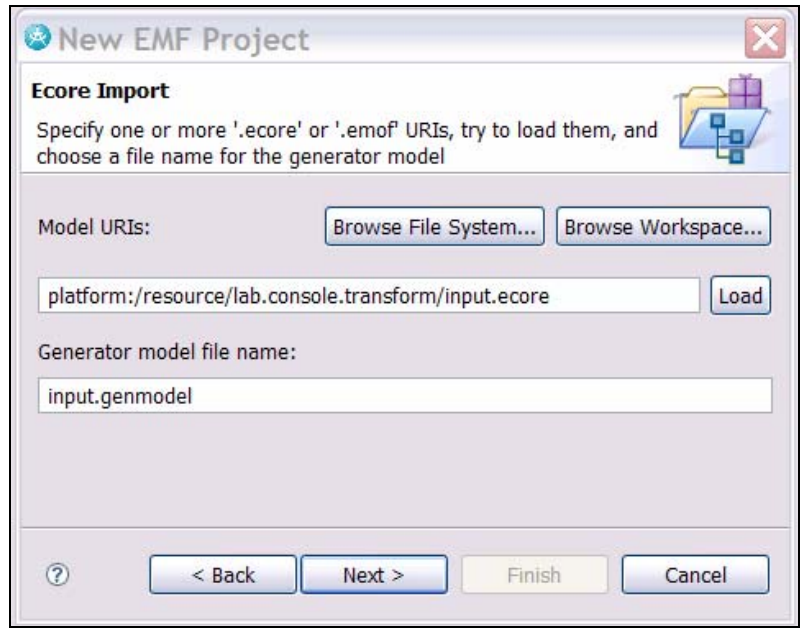


Figure 8-2: *Import the Ecore model*

6. Leave the defaults for the **Package Selection** and select **Finish**.

7. The file `input.genmodel` will display in the **editor**. Right-click the **Input node** and click **Generate Model Code**.

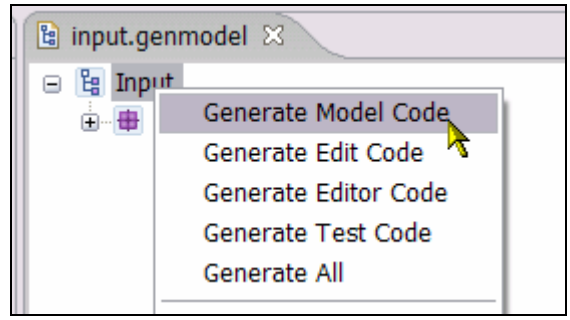


Figure 8-3: *Generate Model code*

8. Observe the packages and files created under the src directory of the model project.

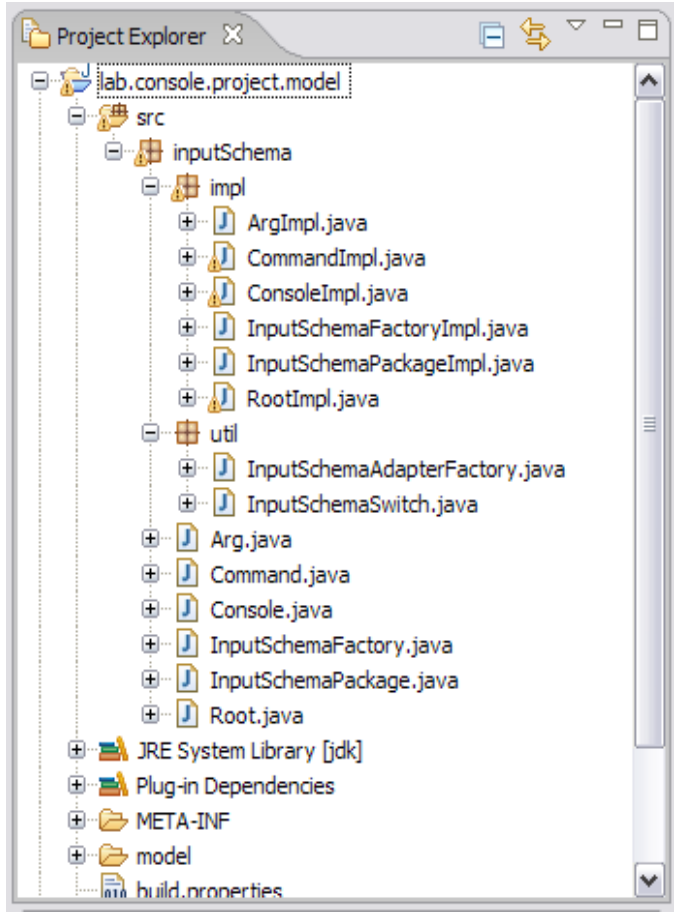


Figure 8-4: *Resulting files from the generation*

Task 3: Create a New Plug-in Project with Transformation Mapping

In this task, you will create a new Plug-in Transformation project named `lab.console.transform.frontend` to define the mapping from UML to the JET console transformation.

1. On the **File** menu, click **New > Project**
2. Replace `type filter text` with `Plug`
3. Select **Plug-in Project** and click **Next**.

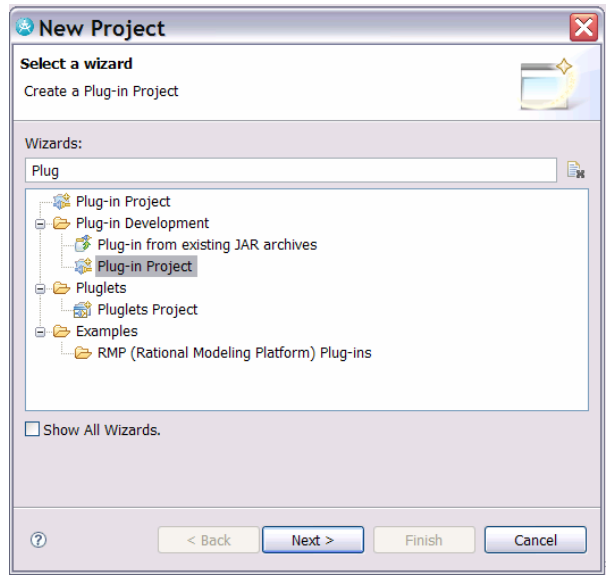


Figure 8-5: *Creating the plug-in project*

4. Name the project `lab.console.transform.frontend` and then click **Next**.
5. Review the **Plug-in Content** screen, leave all the defaults, and click **Next**.
6. On the Templates screen, select **Create a plug-in using one of the templates**.
7. Select Plug-in with Transformation Mapping and click **Next**.

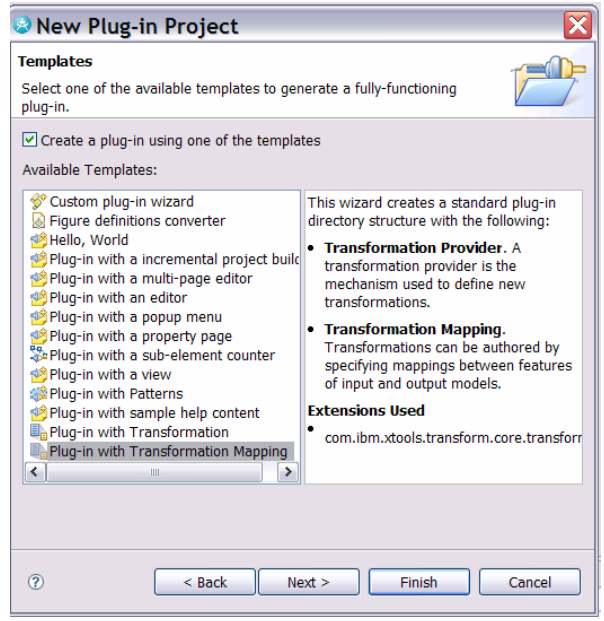


Figure 8-6: *Using the Transformation Mapping template*

8. On the New Transformation Mapping screen, click **Add Model** next to **Input models**.

9. On the Load Resources dialog, click **Browse Registered Packages**.
10. Replace the "*" with "*UML". Select the package `http://www.eclipse.org/uml2/2.0.0/UML`, then click **OK** twice. This selects the UML ecore model for the input model.

TIP: The mapping model uses ecore models as the common model format for mapping.

11. Click **Add Model** next to **Output models.** Click **Browse Workspace**, then select the file `input.ecore` from the `lab.console.transform.model` project from within the `model` folder.

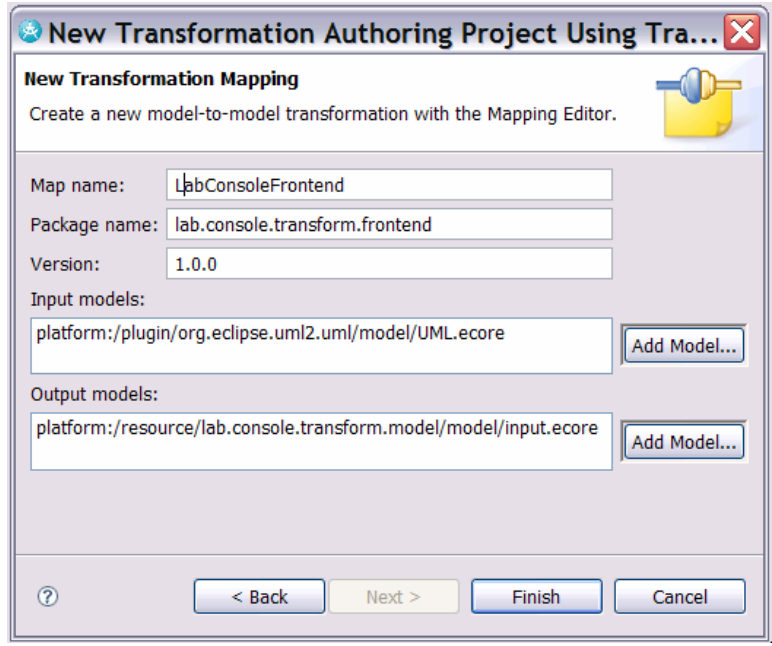


Figure 8-7: *Configuring the transformation project*

12. Enter the **Map name** as `LabConsoleFrontend`.
13. Click **Finish.** If asked to switch to the Plug-in Development perspective, select **No.**

Task 4: Create the Model to Root Mapping

In this task, you will create the first mapping to be used in the transformation. You will create a total of four mappings before you run the first version of the transformation.

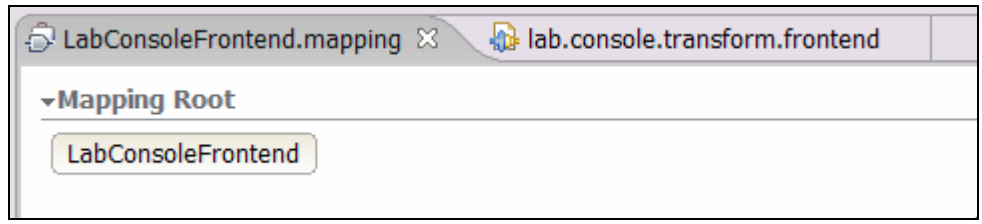


Figure 8-8: *Creating Model to Root mapping*

1. A file called `LabConsoleFrontend.mapping` is created and opened in the mapping editor**.**
2. Right-click the **LabConsoleFrontend** button and select **Create Map**. Name the map **ModelToRoot**.
3. The mapping editor toolbar displays with your new map.

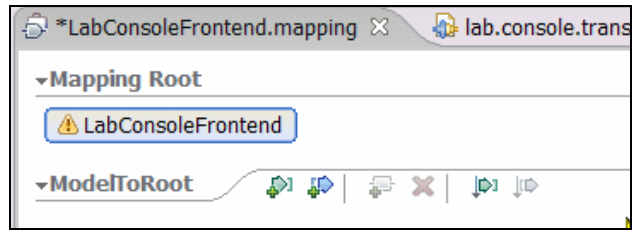


Figure 8-9: *The toolbar to be used when creating the mapping*

4. Click the leftmost button in the toolbar to add an input object.

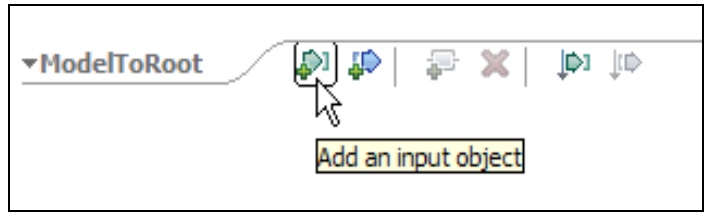


Figure 8-10: *Use the button on the left to create an input object*

5. When the **Add Input** screen displays, simply start typing the letters `mod` and the UML Model will be highlighted. Select **OK.**

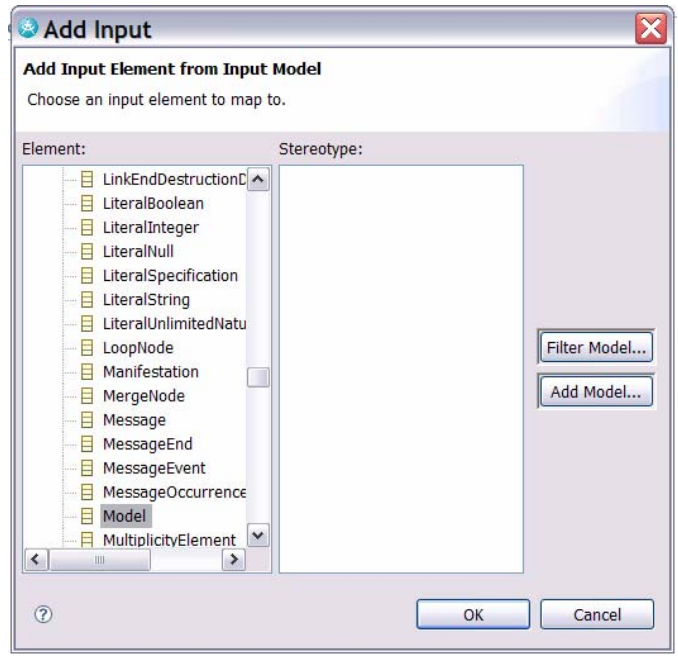


Figure 8-11: *Specifying the input object*

6. Click the second button from the left in the toolbar to add an output object.

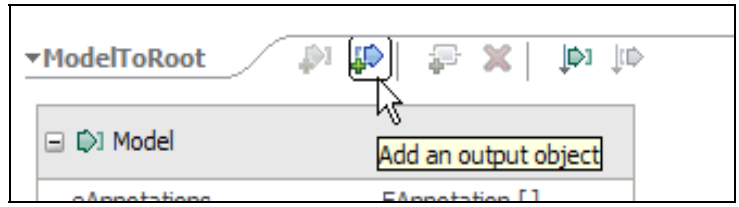


Figure 8-12: *Click the second button from the left to create an output object*

1. Select `Root` and click **OK**.

Now you are ready to define the transformation between the input and output elements. You want to map the `packagedElement` from the UML Model to the `console` element in the ecore model.

2. Hover the cursor over the `packagedElement` property of the input model until a handle appears. Select this handle and drag and drop it onto the `console` element of the target root. The result will be a transformation of type **Submap**, because the cardinality of these elements is greater than 1.

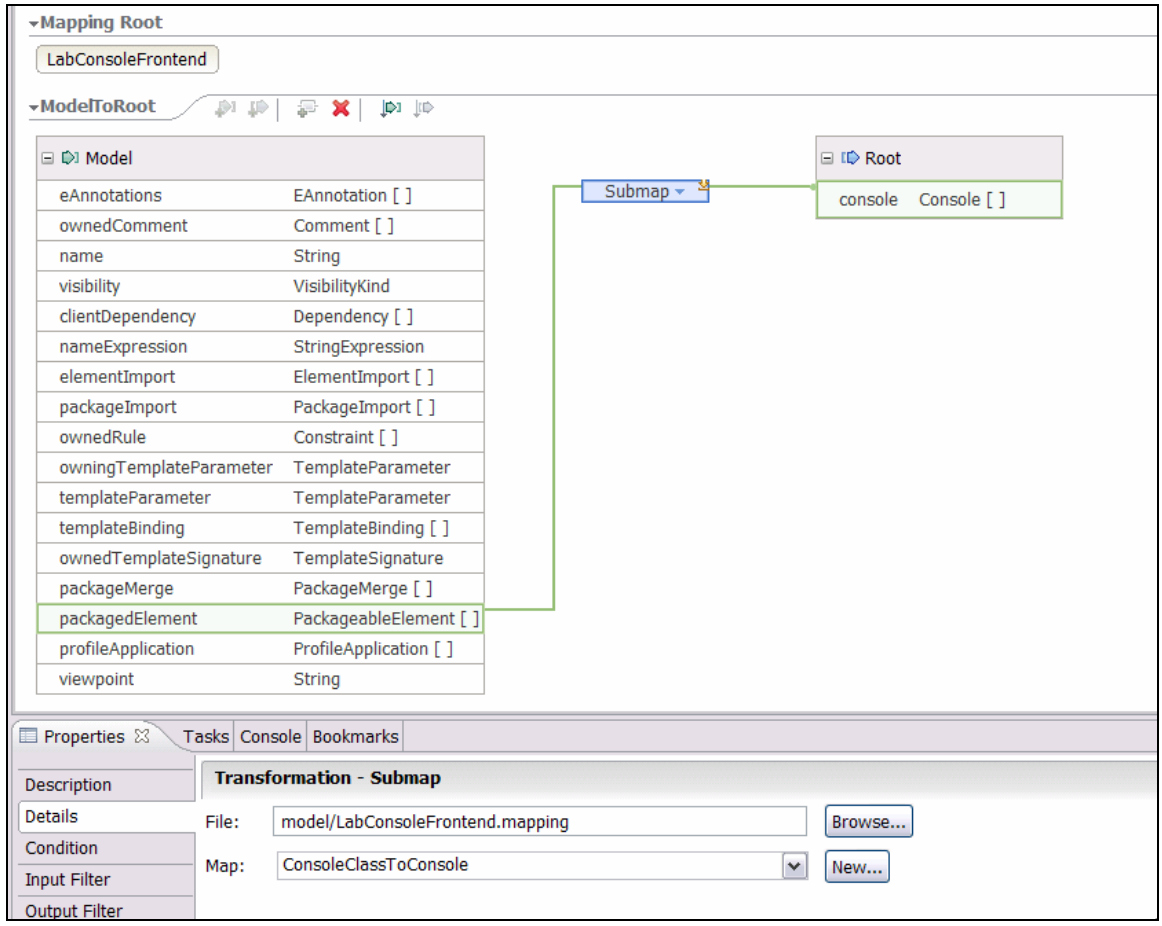


Figure 8-13: *Creating the mapping between the input and output objects*

8. On the **Details** tab of the Properties view, click **New** and name this new map `ConsoleClassToConsole`.
9. Enter **Ctrl-Shift-S** to save all of your work so far.

Task 5: Create the Console Class to Console Mapping

In this task, you will create the mapping that associates the class from the UML model to the console node in the output model.

1. In the Outline View, double-click the `ConsoleClassToConsole` mapping to open it in the mapping editor. Note that the input and output elements were selected for you when the mapping was created.
2. Select the input element and delete it. Set the input element to be a UML class. The output element should already be set to Console.
3. Create a transformation between the name of the input Class and the name of the output Console. Hover the cursor over the name property of the input class until a handle appears. Select this handle and drag and drop it onto the name element of the target console. The result will be a transformation of type Move. You could also think of it as a copy.

4. You want to map the package that the class is in to the package attribute of the console. In order to see the package attribute of the class, you need to change the filter in the mapping editor. Right-click the editor surface and select **Feature Filters > Advanced**.

5. Select the package attribute of the class and open the node so that you can select the package name and connect it to the package attribute of the console output element. You will leave it as a Move transformation.

TIP: Make the editor larger by double-clicking the tab of the editor.

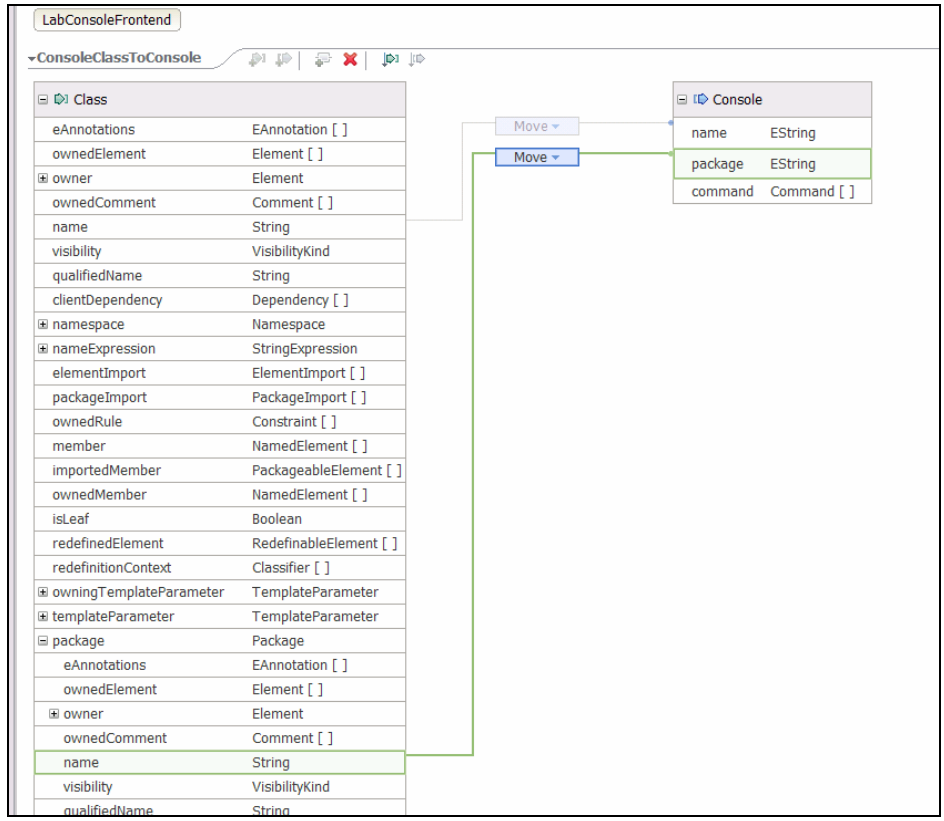


Figure 8-14: *The input object with Feature Filters set to Advanced*

6. Create a Submap from the ownedOperation of the UML class to the command of the Console. In the Properties View, **Detail** tab, create a new map and name it `OperationToCommand`.

TIP: Change the Feature Filter back to **Basic** in order to be able to make the connection in the editor.

7. Enter `Ctrl-Shift-S` to save all of your work so far.

Task 6: Create the Operation to Command Mapping

In this task, you will create the mapping that associates operations from the input UML class to the commands in the output console application.

1. In the Outline View, double-click the `OperationToCommand` mapping to open it in the mapping editor. Note

that the input and output elements were selected for you when the mapping was created.

2. Create a Move transformation between the name of the input Operation and the name of the output Command.
3. Create a transformation between the ownedComment of the input class and the help of the Console. Note that a Custom transformation was created. This is because the ownedComment is an array and the help is just a String. You need to add code to tell the transformation how to translate from the input to the output.
4. In the Properties View, **Detail** tab, add the following code from `C:\Workshop\Labs\Inputs\OwnedCommentToHelp.txt`

```
if(Operation_src.getOwnedComments().size() > 0)
{
   Command_tgt.setHelp(((Comment)Operation_src.getOwnedComments().get(0)).getBo
dy());
} else {
   Command_tgt.setHelp("");
}
```

TIP: Ensure that **Code** is set to **In-line**. Once the code has been entered click **Apply**.

Figure 8-15: *Adding custom code*

5. Create a Submap between the `ownedParameter` of the input Operation and the `arg` of the output Command. (Do you know what's coming next?) Create a new map for this called ParameterToArg.

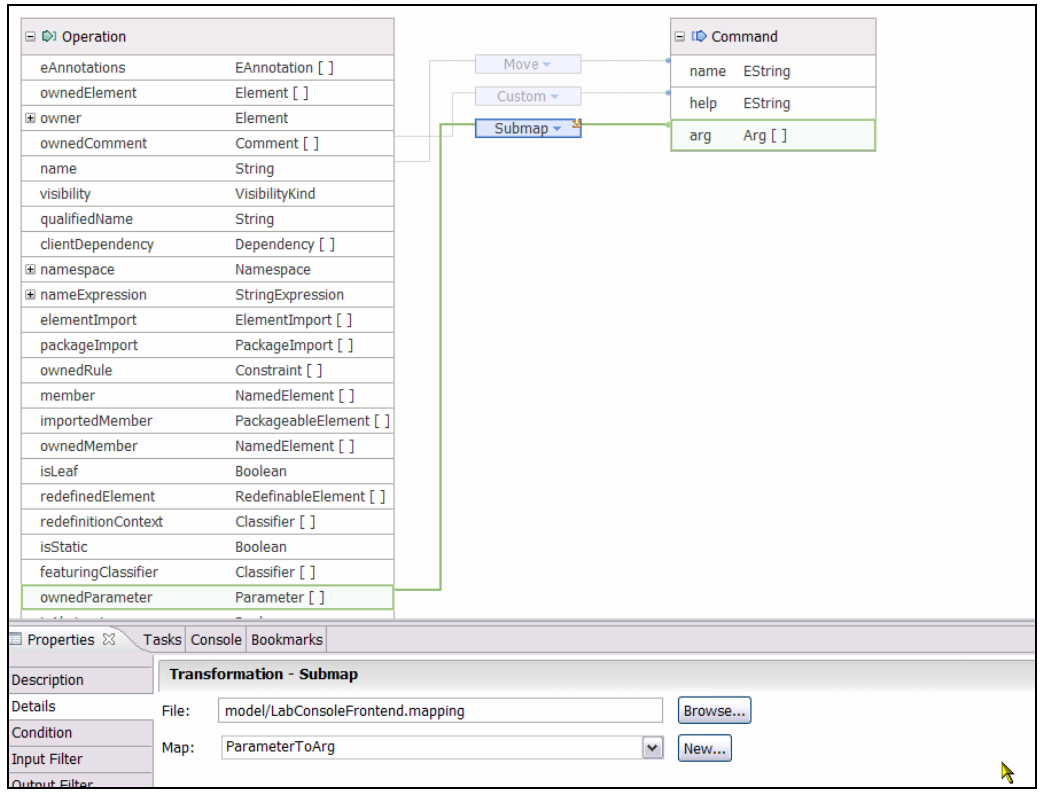


Figure 8-16: *Creating the submap*

6. Enter **Ctrl-Shift-S** to save all of your work so far.

Task 7: Create the Parameter to Arg Mapping

In this task, you will create the mapping that associates parameters from the input UML class operations to the arguments of the commands in the output console application.

1. Here is the last mapping. In the Outline View, double-click the ParameterToArg mapping to open it in the mapping editor.
2. Create a Move transformation from the Parameter name to the Arg name.
3. Create a Move transformation from the Parameter type name to the Arg type.

TIP: If you do not see the Parameter type as a node you can open, set the **Feature Filter** to **Advanced**.

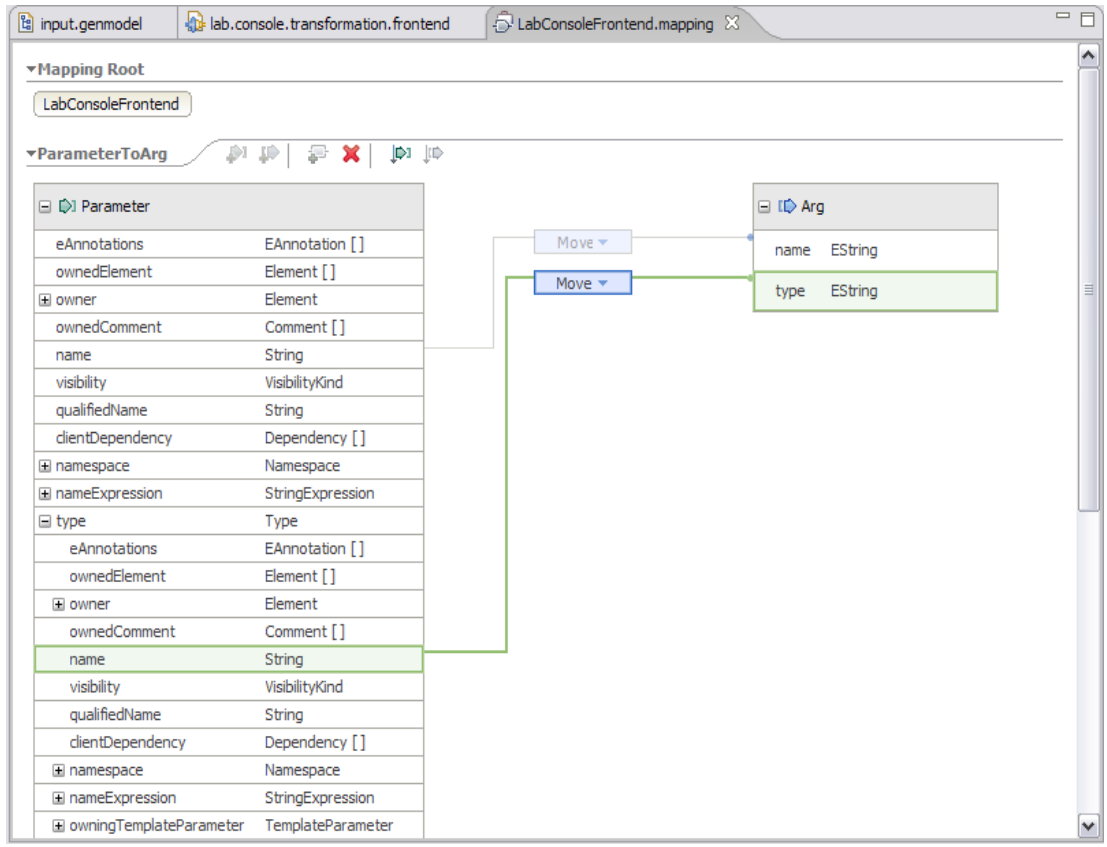


Figure 8-17: *Creating the move map*

4. Enter **Ctrl-Shift-S** to save all of your work so far.

Task 8: Generate the Transformation Code

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

1. Before you generate code, review the files that are in the project so far by opening the nodes of the `lab.console.transform.frontend` project in the **Project Explorer**. All of these were created when the project was created and as you have been editing the `.mapping` file.

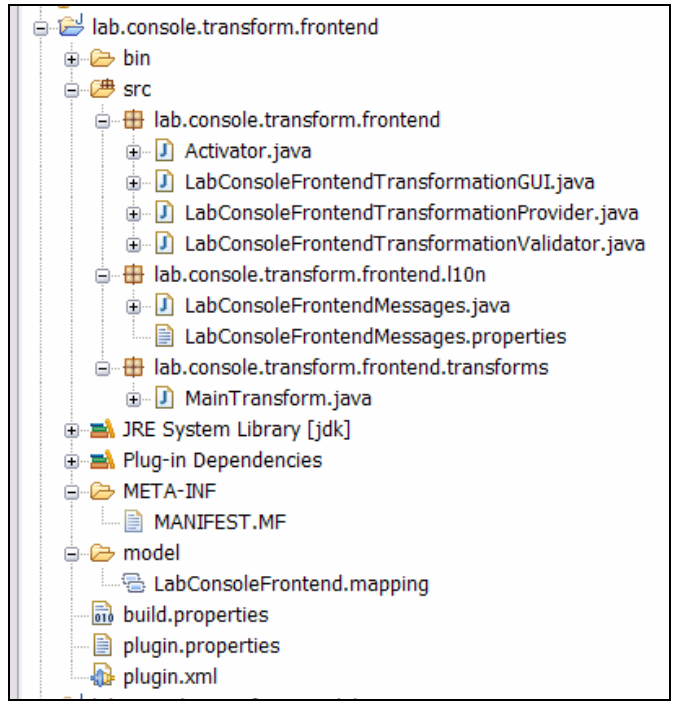


Figure 8-18: *Files as shown in the project explorer*

1. In the Mapping Editor, right-click to the right of the **LabConsoleFrontend** button and select **Generate transformation source code** from the pop-up menu.
2. To resolve the error in the file `OperationToCommandTransform.java`, double click this file and, in the editor, enter **ctrl-shift-o** to organize imports. Select `org.eclipse.uml2.uml.Comment`. Enter **Ctrl-Shift-S** to save all of your work and the error will be gone.
3. Review the transformation files that have been generated.

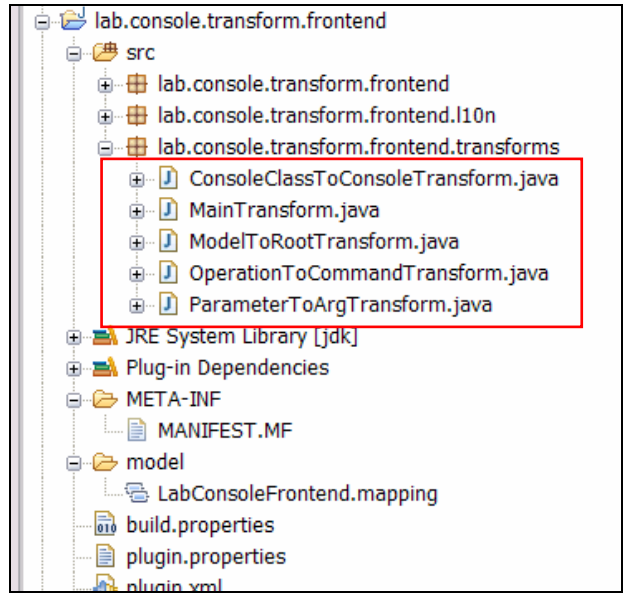


Figure 8-19: *The generated transformation files*

Task 9: Create a Custom Extractor

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

You could test this transformation now, but you would find two issues: 1) all classes would be mapped to console elements, and you only want to process those that have the keyword <<console>> applied, and 2) the transformation would only process classes at the root level of the model, and you want it to find classes that are nested in packages. To account for these requirements, you will implement a custom extractor.

1. The custom extractor is pre-cooked for you in the lab inputs, so in the Project Explorer select the folder `src\lab.console.transform.frontend.transforms`, right-click it and select **Import**. From the file system, import `C:\Workshop\Labs\Inputs\NestedPackageContentsExtractor.java`

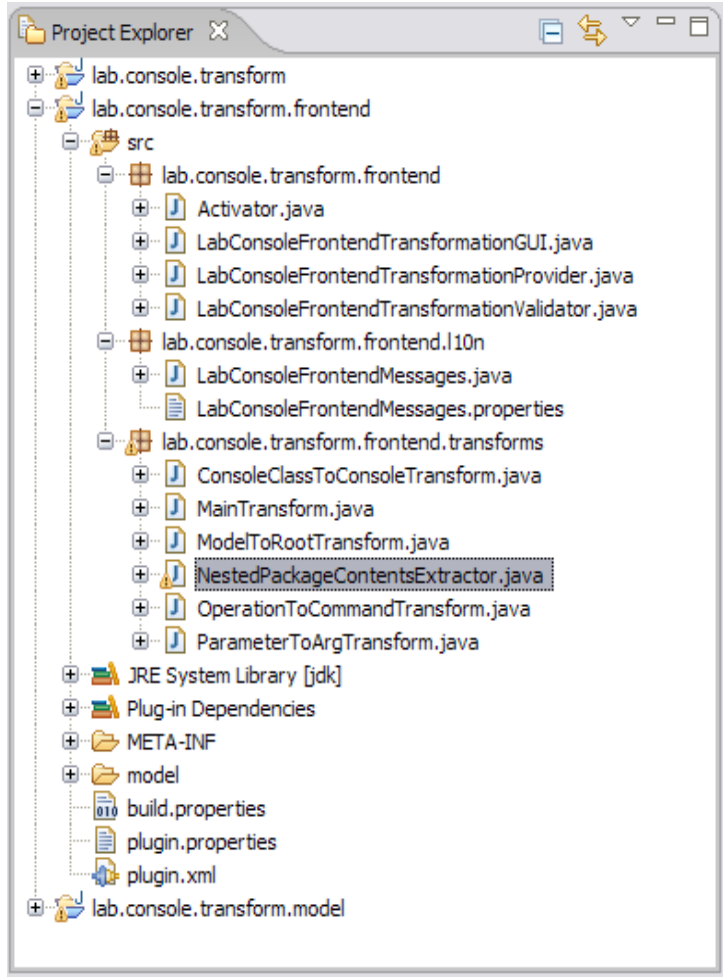


Figure 8-20: *The imported class in the Project Explorer*

2. Open the ModelToRoot mapping and select the Submap from packagedElement to Console.
    a. In the Properties view, on the **Custom Extractor** tab, select the check box 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 in-line).
    c. Select **Browse** and start entering the text for `NestedPackageContentExtractor` until you can select the class that you just imported.
    d. Click **OK**.
3. Enter `Ctrl-Shift-S` to save all of your work.
4. In the Mapping Editor, right click to the right of the **LabConsoleFrontend** button and select **Generate transformation source code** from the pop-up menu.

Task 10: Connect Transformation to JET

In this task, you will add the code that calls the JET transformation from the mapping transformation.

1. In the Project Explorer, in the `lab.console.transform.frontend` project under the src\lab.console.transform.frontend package, find and open the file LabConsoleFrontendTransformationProvider.java

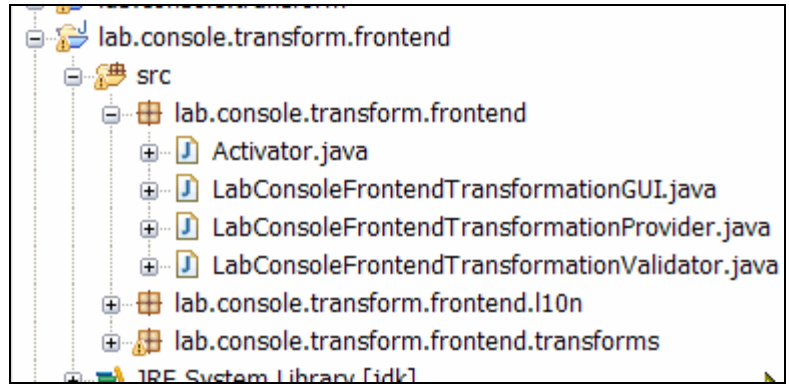


Figure 8-21: *Find the Transformation Provider class*

2. In the `createRootTransformation` method, replace the body of the method with this code:

```
return new RootTransformation(descriptor, new MainTransform()) {
    protected void addPostProcessingRules() {
          add(new JETRule("lab.console.transform"));
    }
};
```

```
/**
 * Creates a root transformation. You may add more rules to the transformation here
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @param transform The root transformation
 * @ !generated
 */
protected RootTransformation createRootTransformation(ITransformationDescriptor descriptor) {
    return new RootTransformation(descriptor, new MainTransform()) {
        protected void addPostProcessingRules() {
            add(new JETRule("lab.console.transform"));
        }
    };
}
```

Figure 8-22: *The updated createRootTransformation method*

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

4. Change the `@generated` tag in the method to `@!generated`.

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. Enter `Ctrl-Shift-S` to save all of your work.

Task 11: Configure Run-time Workbench

In this task, you will configure a Run-time workbench to use in testing the newly created transformation.

1. Switch to the Plug-in Development Perspective.
2. Select **Run** > **Run** from the main menu.
3. On the Run screen, select **Eclipse Application** and click the **New** button (leftmost on the toolbar).
4. Select the **Configuration** tab and set the **Configuration File** field to **Use an existing config.ini file as a template**. Leave the default location. *(Note: This step is critical, as the default Eclipse content option does not provide enough functionality to support a Rational Software Architect test.)*

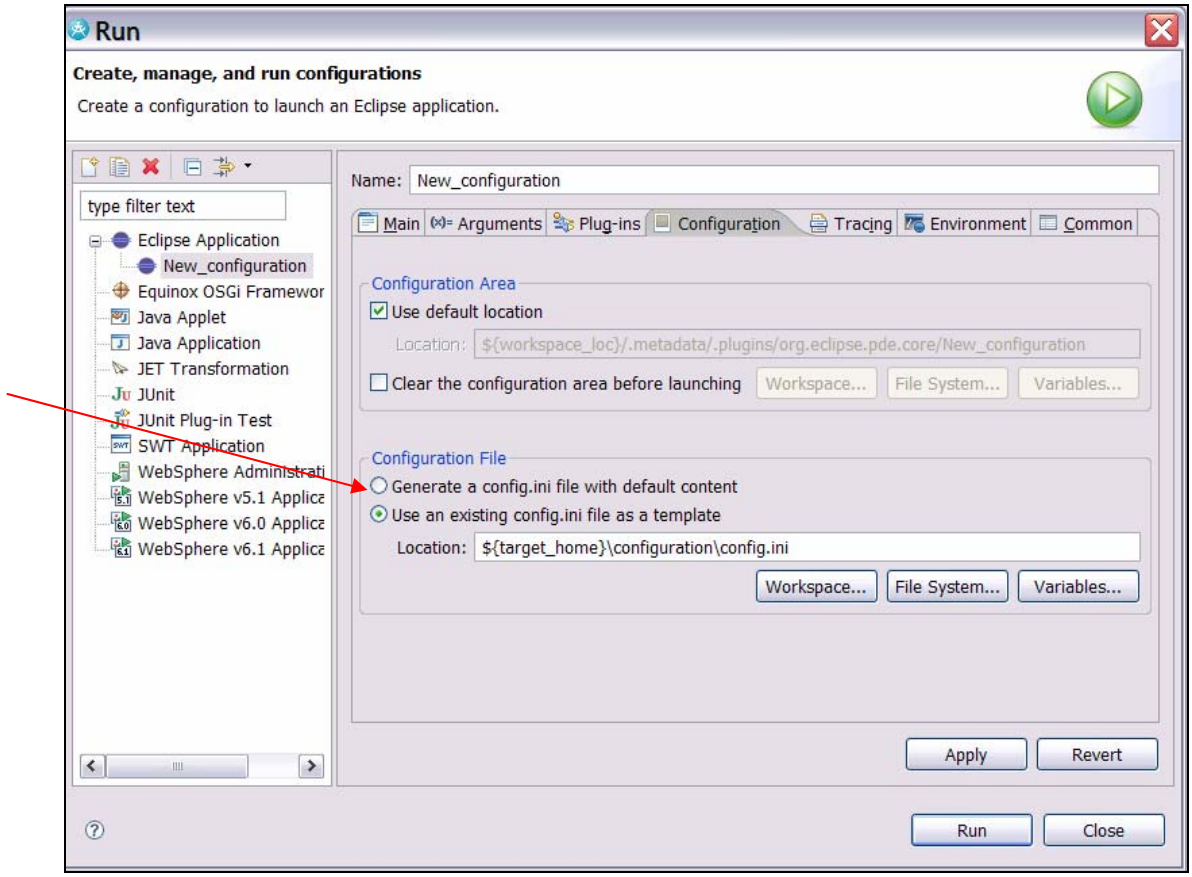


Figure 8-23: *Specifying the Configuration file*

5. Select **Apply**, then **Run**.

Next you will need to test the transformation in the Run-time workbench.

Task 12: Test the Transformation

In this task, you will test the newly created transformation in the Run-time workbench.

1. In the run-time workbench, close the Welcome screen.
2. Switch to the Modeling perspective in the Run-time workbench.
3. Import the project interchange file `C:\Workshop\Labs\Inputs\TestConsoleModel.zip` and select the project `TestConsoleModel`.
4. Review the elements in the test model.
5. In the project, open the CommandModel and the Target Model models.
6. Create a new transformation configuration of the LabConsoleFrontend called `myConsoleTest`. Click **Next**.

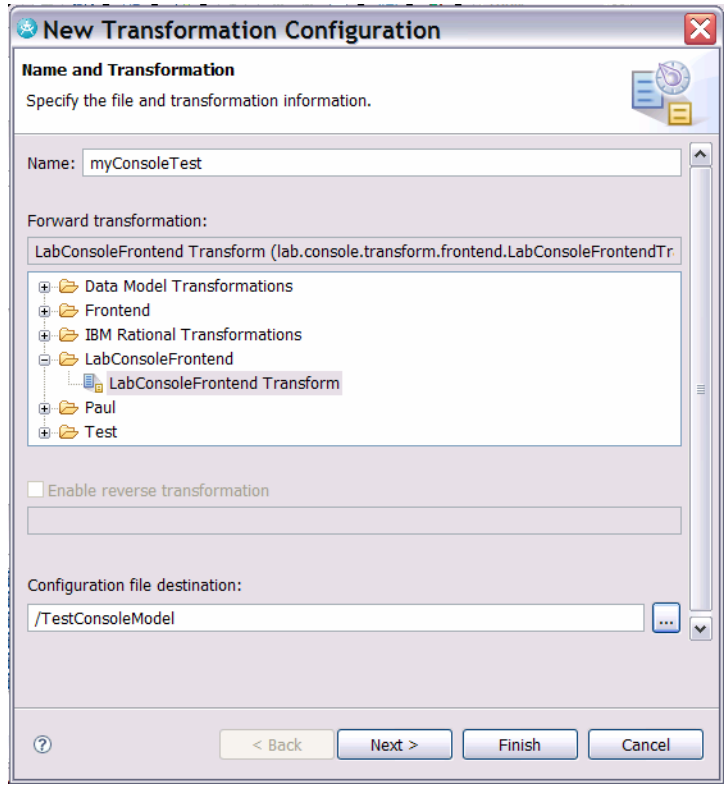


Figure 8-24: *Creating the transformation configuration*

7. Select the TestConsoleModel as the input model and TargetModel as the output model. Click **Finish**.
8. Locate the file **myConsoleTest.tc** in the Project Explorer. Right-click this file and select Transform >LabConsoleFrontend Transform.
9. As a result of the transformation execution, two new projects are created in the workspace.

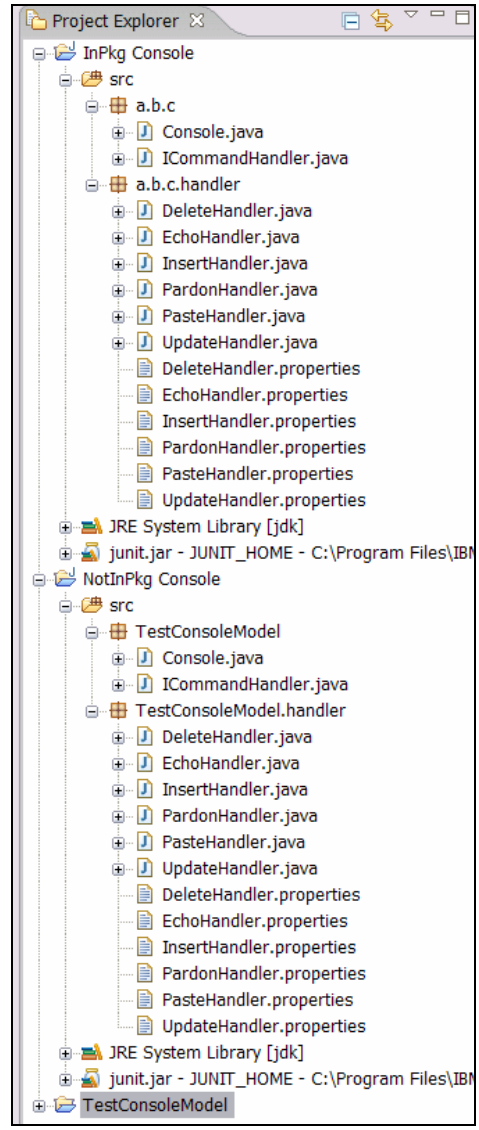


Figure 8-25: *New projects generated as result of the transformation*

3. Examine the contents of the projects and validate that the elements of the input UML model have been mapped to the transformed text elements.

Lab 9 – Create a UX Modeling Profile

Objectives

After completing this lab, you will be able to:

- Create a UML profile to be used for modeling User Experience
- Add a constraint to a profile
- Customize a profile with domain related icons
- Export and import projects
- Configure and use a run-time workbench for plug-in testing

Given

- `ScreenIcon.bmp` and `ScreenIcon.emf`: Images to be used as part of the profile
- `ProfileTestProject.zip`: A Project Interchange file that contains a simple model to be used when testing the profile
- `JavaClassNameConstraint`: A text file that contains code to be used in the constraint class.
- `UpdatedUXProfilePlug-in.zip`: A Project Interchange file that contains additions to the originally created profile

Scenario

In this portion of the workshop, you will create a UML Profile that will capture details related to User Experience (UX) modeling. The initial purpose for this profile is to generate Struts-based applications. However, an additional goal is to develop a profile that can be used for other user experience implementations, such as JSF. In addition, you will add to the richness of the profile by adding custom icons and a constraint.

Task 1: Create the Workspace

In this task, you will switch to a new workspace named `CreateUXProfileWorkspace` that you will create.

1. From the **File** menu, select **Switch Workspace**.

2. In the Workspace Launcher dialog, replace the displayed text with `C:\Workshop\StudentWork\CreateAUXProfileWorkspace` and click **OK**.
3. Close the Welcome screen.

Task 2: Create the Profile

1. Create a new UML Profile Project.
2. Click **File > New > Project**.
3. In the New Project dialog, replace type filter text with `UML Profile`. Select **UML Profile Project**, and then click **Next.**

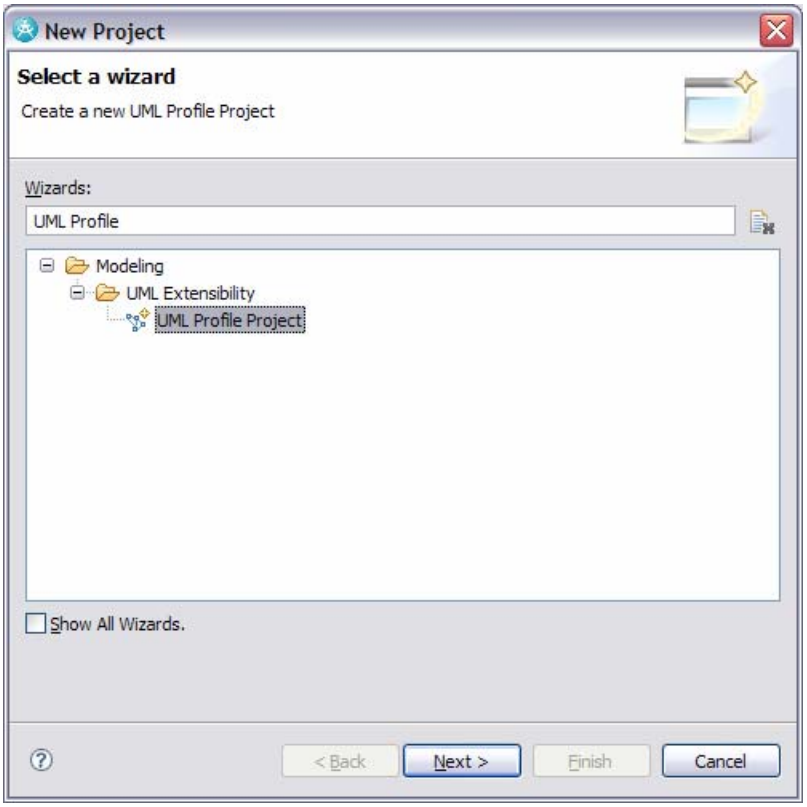


Figure 9-1: *UML Profile Project*

   a. Name the project `UXModeling Profile Project`. Click **Next**.
   b. Name the **Profile** and **File** `UXModeling`.

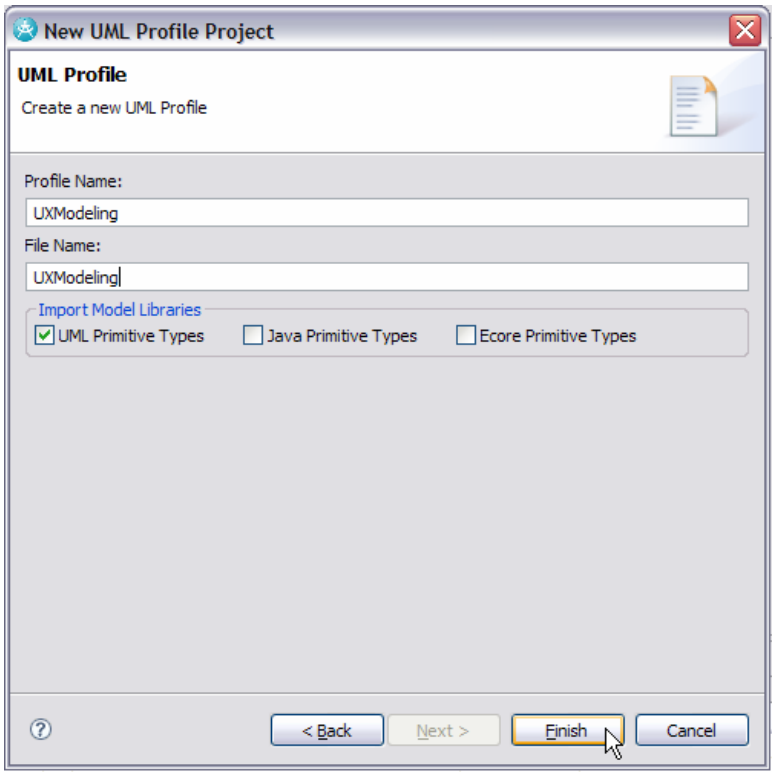


Figure 9-2: *Name the Profile*

4. Ensure that `UML Primitive Types` is selected to limit your profile to UML 2 types.
5. Click **Finish** to create the project. If you are asked to open the Modeling perspective, click **Yes**.

Task 3: Add Stereotypes and Properties to the Profile

As part of your team's effort, you will populate the profile with stereotypes, attributes, enumerations, and so on.

1. Add new Stereotypes to the profile.
   a. In the Project Explorer view, right-click the `UXModeling Profile` and click **Add UML > Stereotype**.

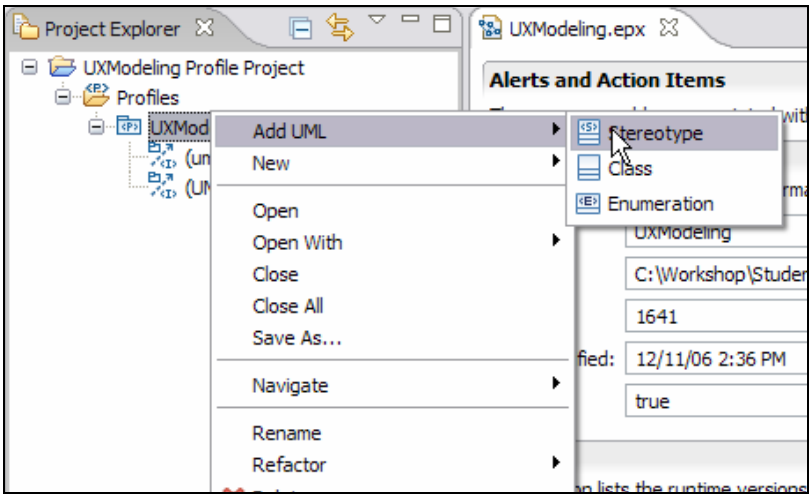


Figure 9-3: *Add a UML Stereotype*

b. Create three stereotypes named `display`, `input`, and `useraction`.

2. Add extensions to the `display` and `input` stereotypes

a. Add an extension to the `display` stereotype so that it will apply to **Property** (attributes).

TIP: To specify the extension for a stereotype, select the stereotype in the Project Explorer and then choose the **Extensions** tab within the Properties view.

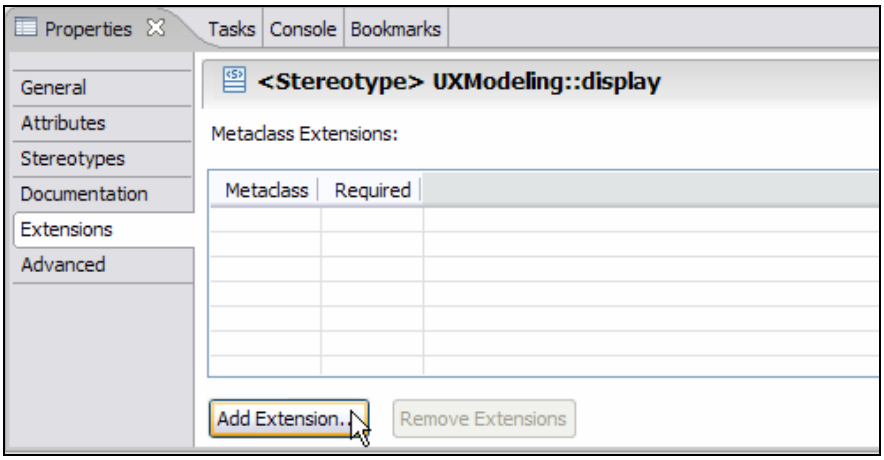


Figure 9-4: *Add an extension to the stereotype*

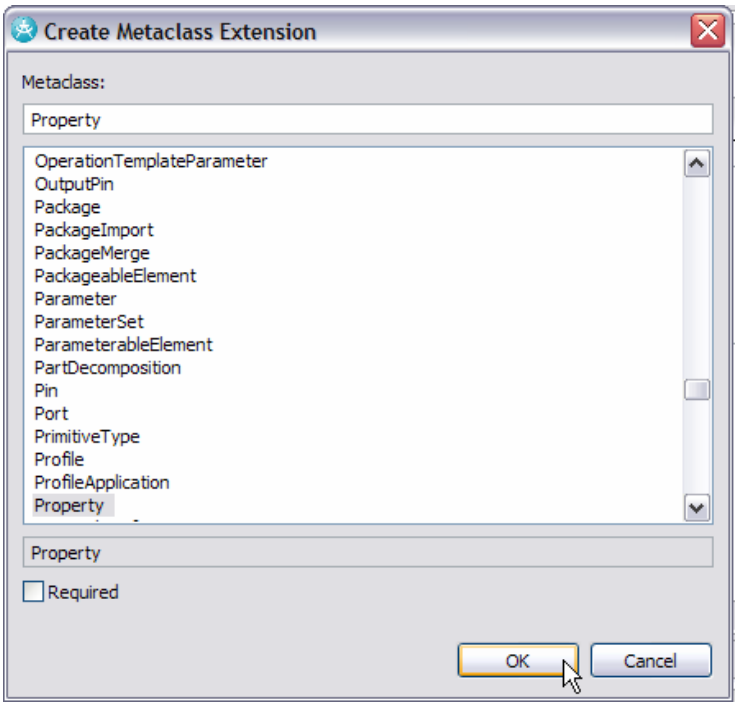


Figure 9-5: *Choose the extension for the stereotype (do not select **Required**)*

b. Add an attribute to the `display` stereotype. Its name will be `javabean` and it will be of Type `string`. When you name the attribute, append a colon, and a context assist window will open where you can select the type.

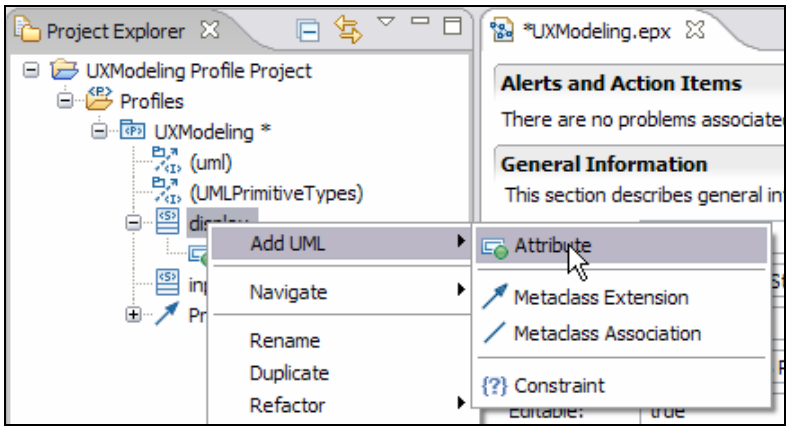


Figure 9-6: *Add Stereotype Attributes*

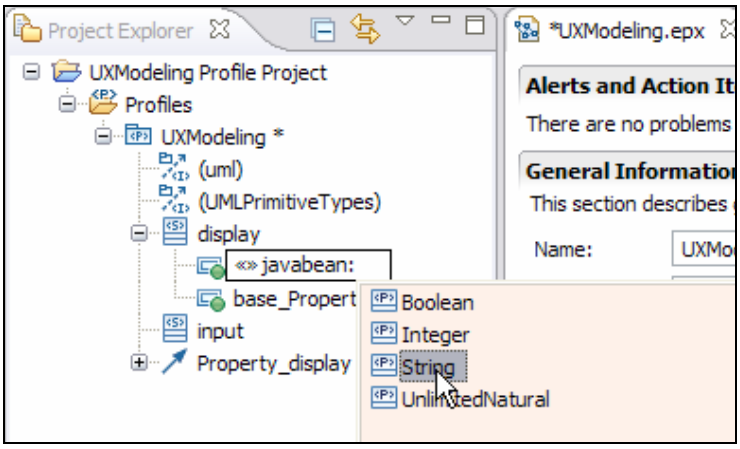


Figure 9-7: *Select the Attribute Type*

TIP: You can use the code assist (`Ctrl-Space`) feature in Project Explorer to select the element type.

c. Add an additional attribute, named `label`, to the `display` stereotype. It will also be of Type `string`.

d. Add an extension to the `input` stereotype so that it will apply to **Class**. Do not make it **Required**.

e. Add an attribute to the `input` stereotype named `javabean` of Type `string`.

3. Create an enumeration.

a. In the Project Explorer view, right-click the `UXModeling` profile folder and click **Add UML > Enumeration.**

b. Name the new enumeration `ActionKind`.

c. Right-click `ActionKind` and click **Add UML > Enumeration Literal**. Add `Reset` and `Submit` enumeration literals.

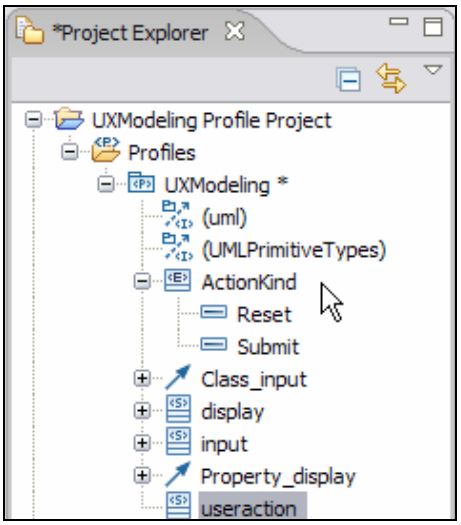


Figure 9-8: UML Enumeration

4. Select the `useraction` stereotype, and add an extension so it will apply to **Operation**. Do not make it **Required.**

a. For the `useraction` stereotype, navigate to the **Attributes** tab in the Properties view. Add the following attributes:

i. Name: `actionpath` Type: `String`

ii. Name: `javaclass` Type: `String`

iii. Name: `label` Type: `String`

iv. Name: `kind` Type: `Enumeration ActionKind`

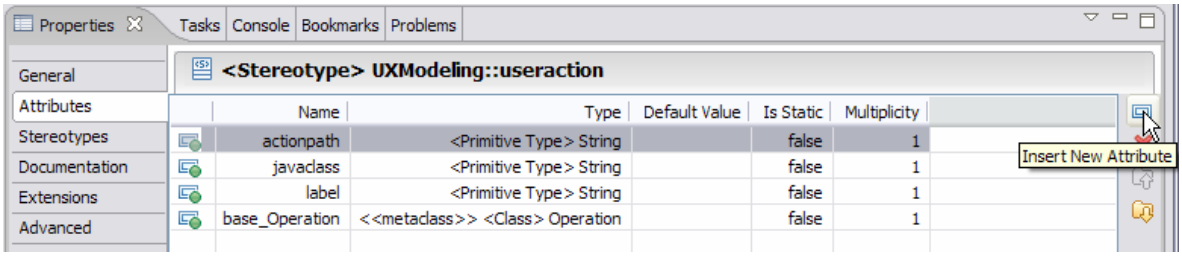


Figure 9-9: *Add Stereotype Attributes*

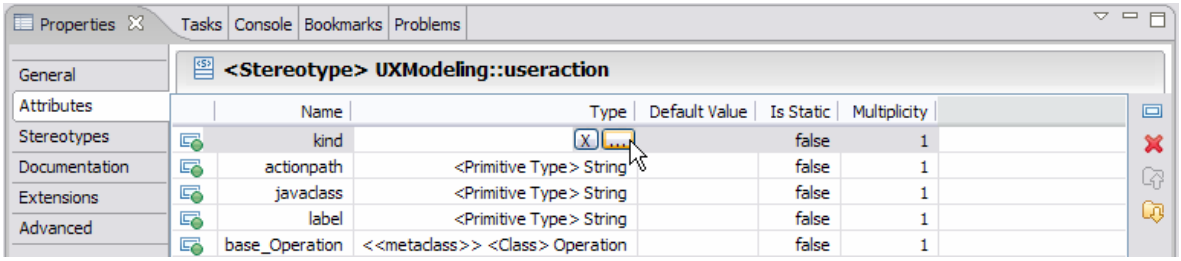


Figure 9-10: *Specify the Type for an Attribute*

5. Select **File > Save All**.

> ***TIP:*** The `Ctrl-Shift-S` keyboard shortcut will also **Save All**.

6. Review. The following table and Project Explorer snapshot identify the model elements added to your profile so far. For elements where documentation exists, add it via the element's Property View.

Name	Add UML >	Type	Extension	Owning Element	Documentation	Default Value
ActionKind	Enumeration					
Reset	Enumeration Literal			ActionKind		
Submit	Enumeration Literal			ActionKind		
display	Stereotype		Property			
javabean	Attribute	String		display	Bean class to which the «display» field belongs. Syntax = rootpackage.package1. ...packagen.ClassName .	
input	Stereotype		Class			
javabean	Attribute	String		input	(Mandatory) Syntax = rootpackage.package1. package2...packageN.b eanname	
useraction	Stereotype		Operation			
actionpath	Attribute	String		useraction		
javaclass	Attribute	String		useraction		
label	Attribute	String		useraction		
kind	Attribute	Enumeration ActionKind		useraction		

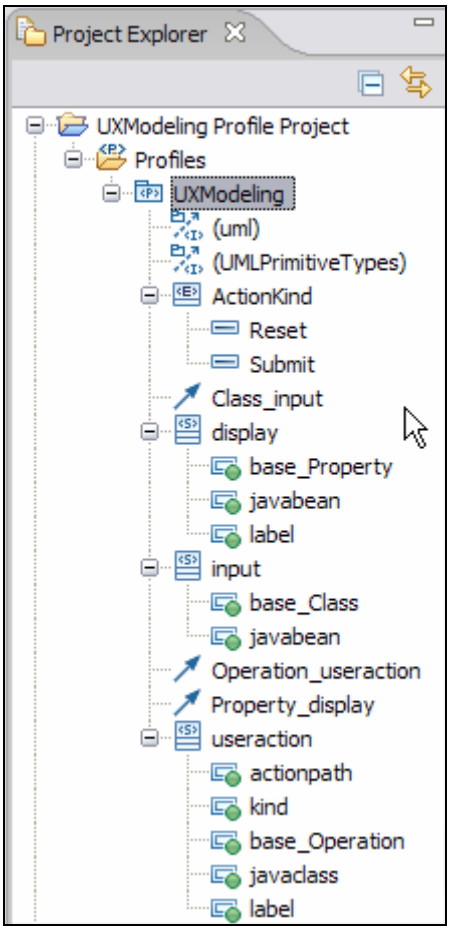


Figure 9-11: *Profile in Project Explorer*

7. **Save All**.

Task 4: Add Custom Icons

In this task, you will update the profile to use custom icons for the `input` stereotype you created. Custom icons can add to the usability of your profile, providing the end user with a visual cue.

1. In the Project Explorer, select the input stereotype node.
2. In the Properties view, select the **General** tab.
   a. Click the **Browse** button located to the right of the **Icon** field.

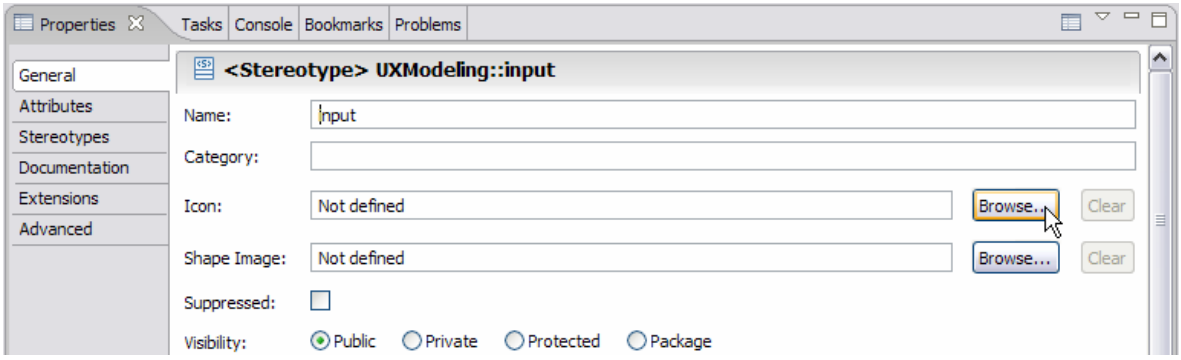


Figure 9-12: *General properties for the input stereotype*

b. Navigate to the `C:\StudentWork\Labs\Inputs` folder and select the `FormIcon.bmp` file. Click **Open**.

c. Click the **Browse** button located to the right of the **Shape Image** field.

d. Navigate to the `C:\StudentWork\Labs\Inputs` folder and select the `FormShape.emf` file. Click **Open**.

3. Select **File > Save All**.

Task 5: Add Profile to a Plug-in Project

In this task, you are going to add the Profile to a Plug-in project. Distributing the profile as just an .epx file is fine in very simple cases, but more often than not you will want to put the profile into a Plug-in.

1. Select **File > New > Project**.

TIP: If you are unable to find a type of project in the New Project dialog, select the **Show all Wizards** checkbox to see the complete list. The dialog is initially populated based on the roles and capabilities specified.

a. Replace type filter text with Plug-in. Then select **Plug-in Development > Plug-in Project**. Click **Next**.

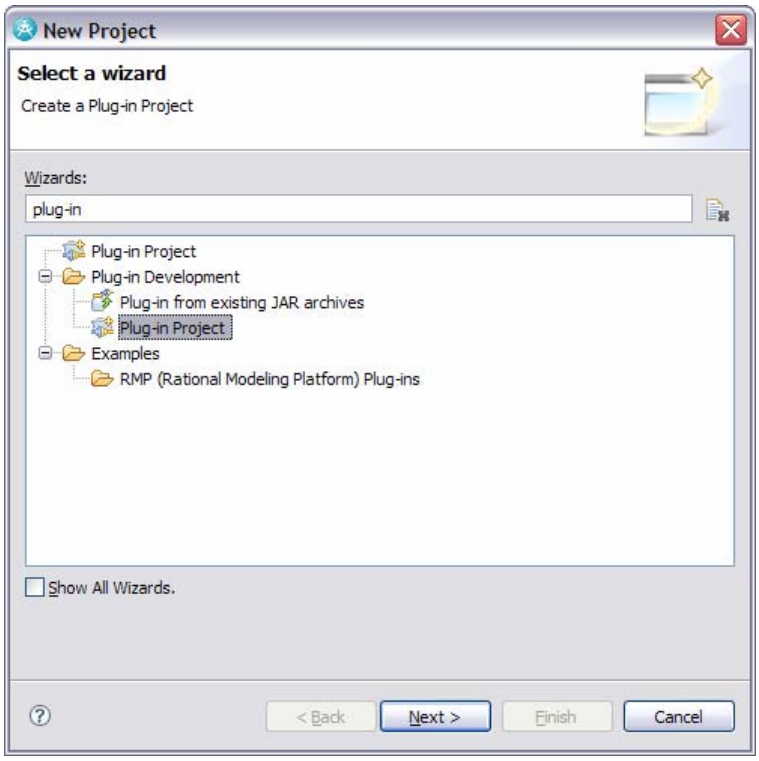


Figure 9-13: *Create the plug-in project*

TIP: Click **OK** if you are asked to enable Eclipse Plug-in Development Capabilities.

b. Specify `UXProfilePlug-in` as the **Project name**. Click **Next**.

c. Fill out the Plug-in Content dialog as follows:

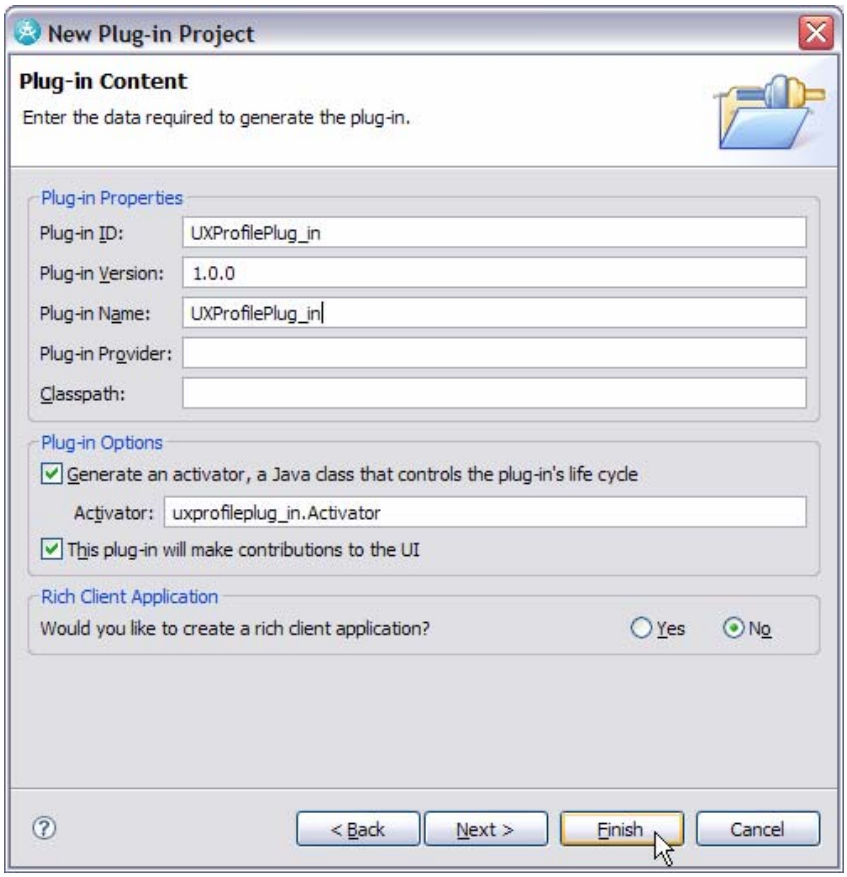


Figure 9-14: *Details for the plug-in project*

d. Click **Finish**.

2. Click **Yes** to switch to the Plug-in Development perspective.
3. Switch to the **Extensions** tab of the manifest editor.

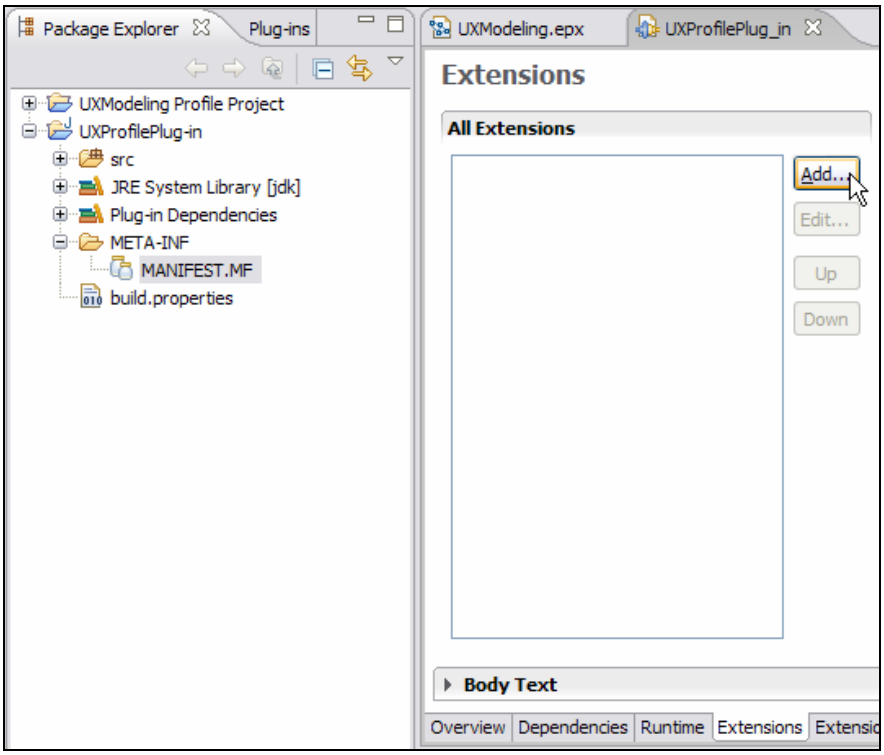


Figure 9-15: *The extensions tab of the plugin.xml file within the manifest editor*

2. Click **Add**.
3. Clear the **Show only extension points from the required plug-ins** option.
4. Select `com.ibm.xtools.uml.msl.UMLProfiles` from the **Available extension points** list. Click **Finish**.

> *TIP:* Use the **Extension Point filter** to quickly and easily find the extension point.

5. Click **Yes** to add the plug-in to the list of plug-in dependencies.
6. Specify `UXProfileID` as the **Id**.
7. Specify `UXProfileName` as the **Name**.

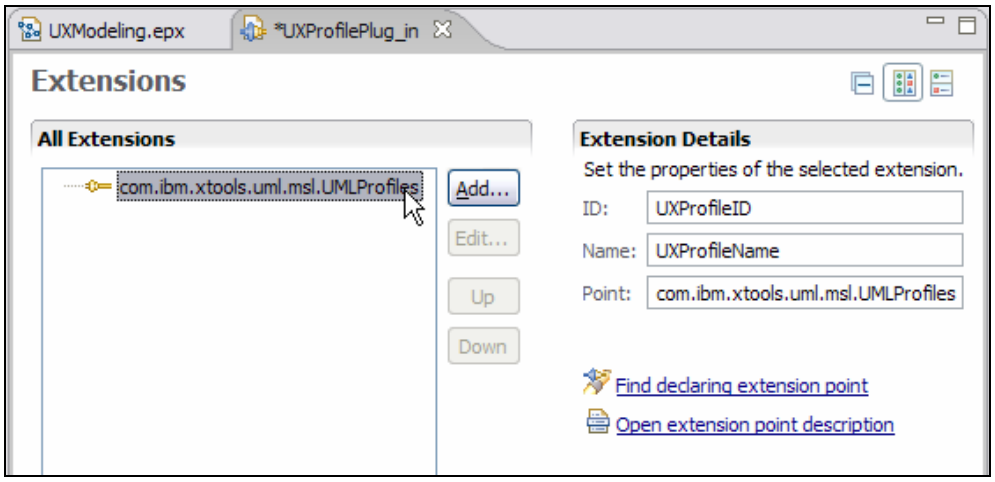


Figure 9-16: *UMLProfiles extension*

8. In **All Extensions**, right-click `com.ibm.xtools.uml.msl.UMLProfiles` and select **New > UMLProfile**.
9. Fill in the **Extension Element Details** as shown below:

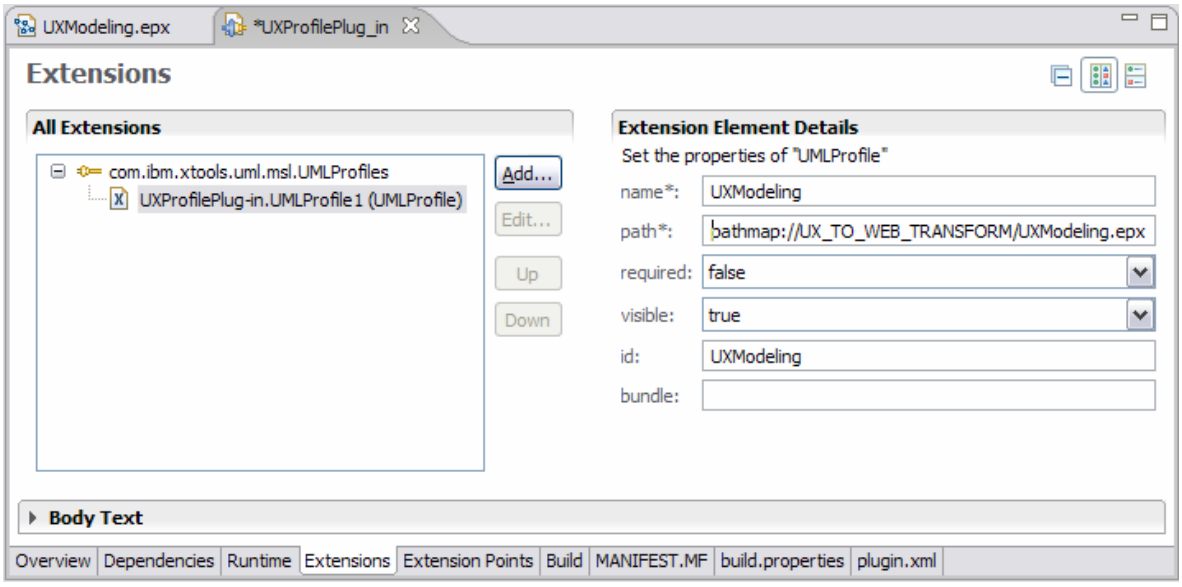


Figure 9-17: *Connecting the UXModeling profile to the extension*

2. Now define the pathmap under **All Extensions**:
   a. Click **Add**.
   b. Ensure that **Show only extension points from the required plug-ins** is not selected.
   c. Select `org.eclipse.gmf.runtime.emf.core.Pathmaps`

TIP: Type in the first part of the Extension Point filter name to automatically filter.

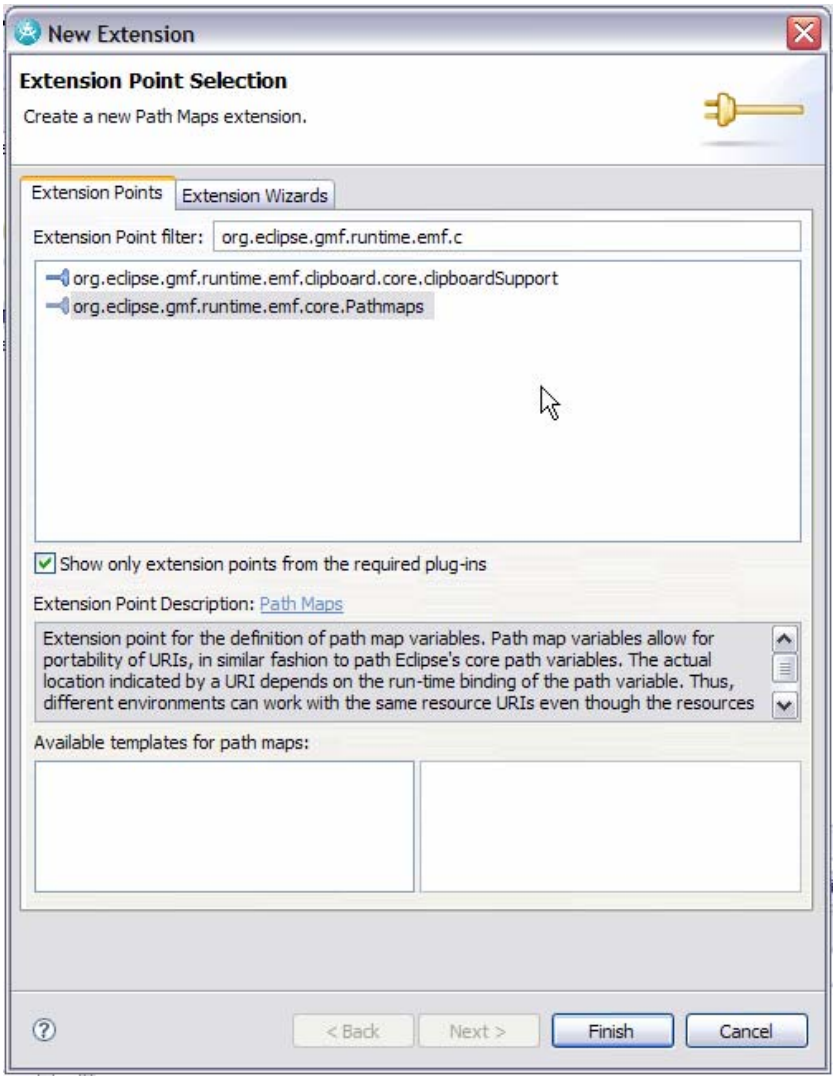


Figure 9-18: *Selecting the extension point for the Pathmap*

3. Click **Finish.**
4. Click **Yes** to add the plug-in to the list of plug-in dependencies.
   d. Enter the following details into the **Extension Details**:

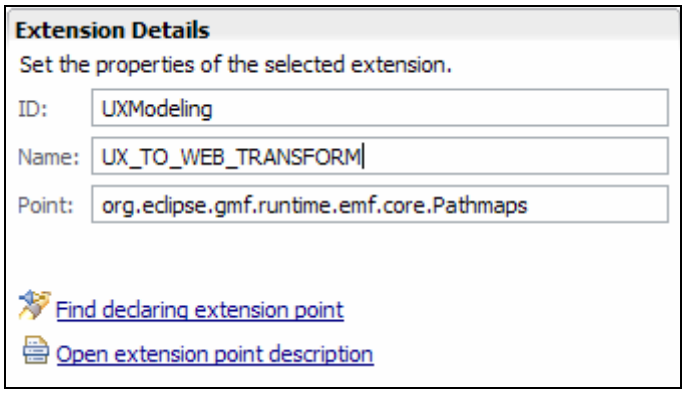


Figure 9-19: *Selecting the extension details Pathmap*

e. Save.

f. Right-click the new extension point and select **New > pathmap**.

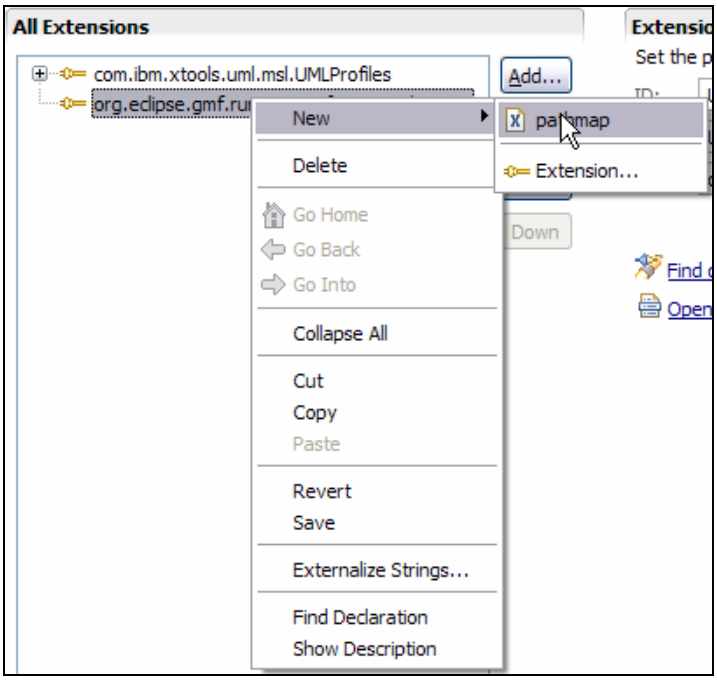


Figure 9-20: *Adding the pathmap*

g. Enter the following details into the **Extension Element Details**:

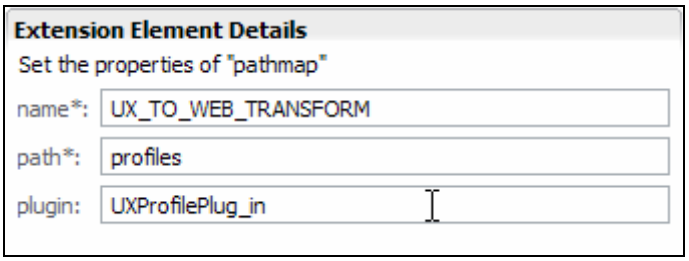


Figure 9-21: *Specifying the details for the pathmap element*

10. Save.

11. Review the source for the configuration by selecting the plugin.xml tab.

Now that you've set up the plug-in project, let's put the profile .epx file into the plug-in project.

12. In the Package Explorer, right-click `UXProfilePlug-in` and select **New > Folder**.

13. Specify `profiles` as the **Folder name**. Click **Finish**.

14. Copy the `UXModeling.epx` file from the `UXModeling` project to the newly created profiles folder within the `UXProfilePlug-in` project.

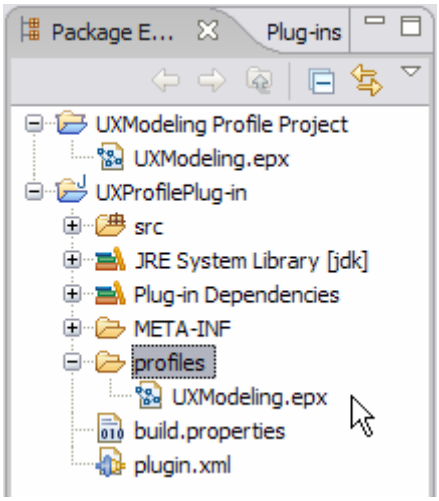


Figure 9-22: *Copying the UXModeling.epx file to the Plug-in*

Task 6: Add a Constraint

In this task, you will add a constraint to the profile to ensure that one of the properties specified by the profile is used properly. In this case, you want to make sure that the `javaclass` property for the `useraction` stereotype is not left blank.

1. Switch to the Modeling perspective.

a. In the Project Explorer, right-click the `UXModeling` project and select **Close**.

TIP: Close the original UML Profile project, because from this point forward you want to focus on the profile that you just copied into the plug-in project. By closing the original version, you reduce the risk of getting confused about which file and version you are working with.

b. Within the UXProfilePlug-in project, navigate to the `Profiles` folder.

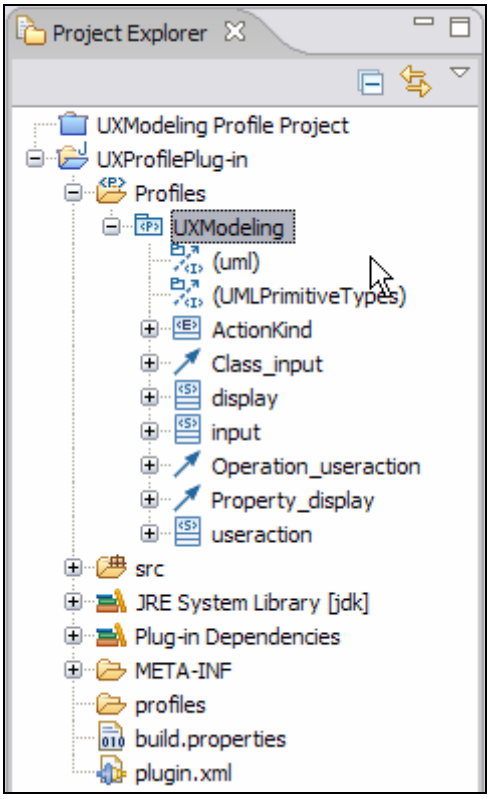


Figure 9-23: *UXModeling profile in the Modeling Perspective Project Explorer*

c. Choose the `UXModeling` profile to open the profile for editing.

d. Select the `useraction` stereotype node.

e. Right-click and select **Add UML > Constraint**.

f. Name the new constraint `JavaClassNameConstraint`.

g. In the Properties view, select the **General** tab.

h. Select `Java Class` as the **Language**.

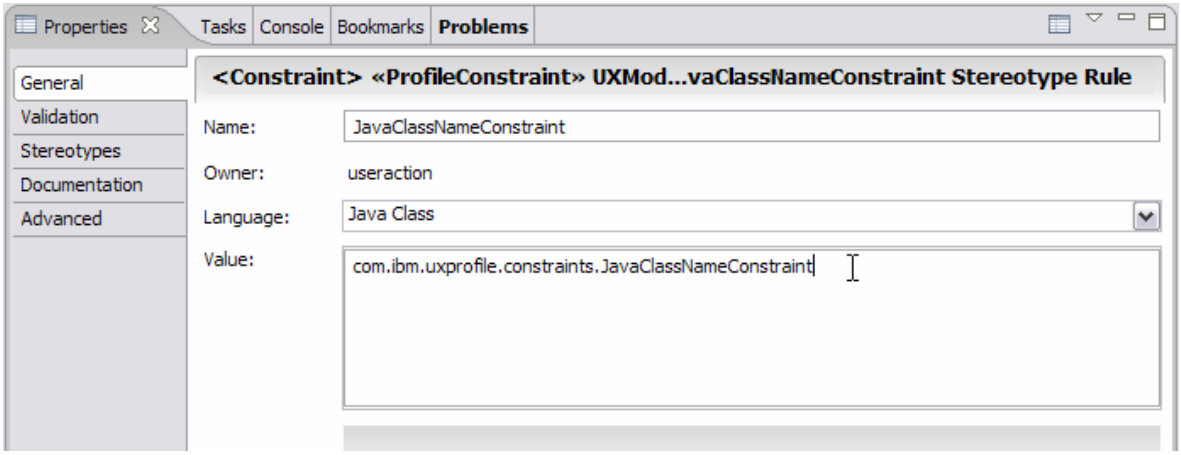


Figure 9-24: *Specifying the language for the constraint*

i. Specify the Value as:
`com.ibm.uxprofile.constraints.JavaClassNameConstraint`.

TIP: You are specifying that the constraint can be found in the specified java class. The constraint will eventually extend the AbstractModelConstraint class as found in the com.ibm.xtools.emf.validation package.

j. Select **File > Save All**.

Task 7: Create Constraint Class

In this task, you will create the java class that contains the logic associated with the constraint

1. Switch to the **Plug-in Development** perspective.

a. In the **Package Explorer**, expand the `UXProfilePlug-in` project node.

b. Open the `plugin.xml` file.

c. Switch to the **Dependencies** tab of the manifest editor.

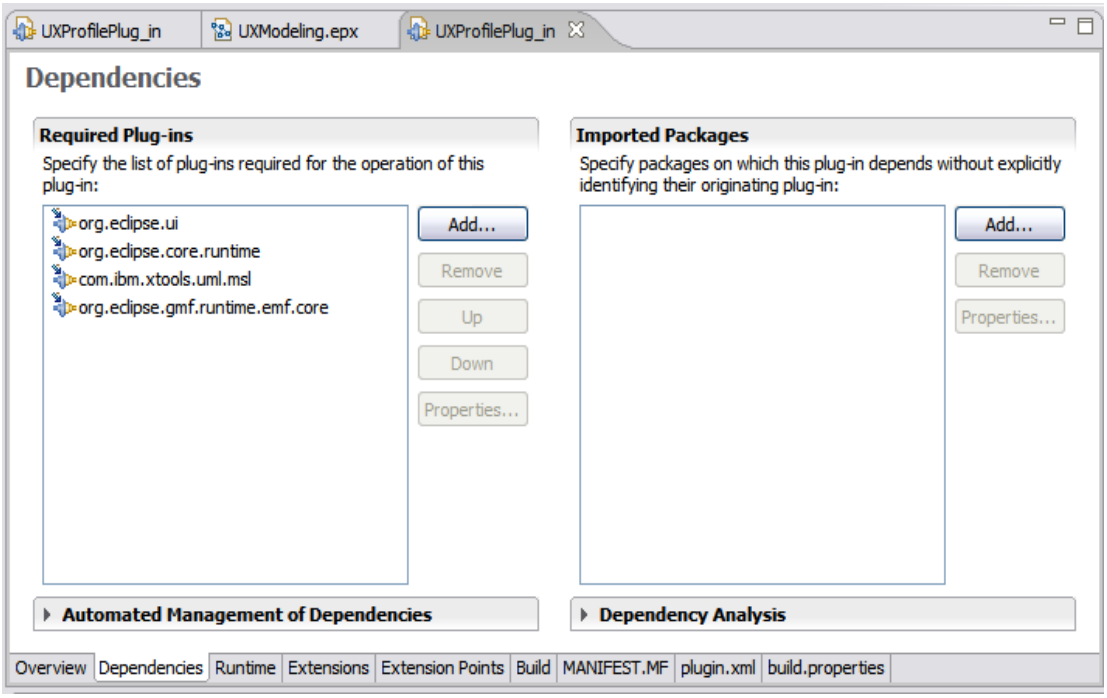


Figure 9-25: *Dependencies for the plug-in*

d. Click **Add**.

e. Select `org.eclipse.uml2.uml` and then click **OK**.

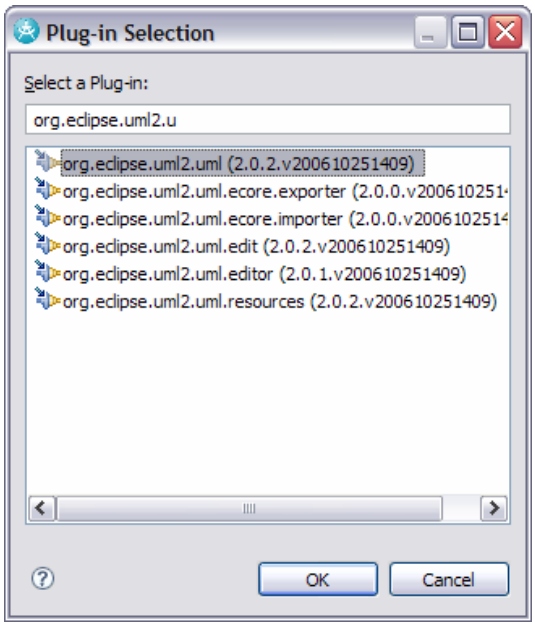


Figure 9-26: *Selecting the plug-in*

f. Save.

g. In the **Package Explorer**, right-click `src` and select **New > Package**. Name the package `com.ibm.uxprofile.constraints`. Click **Finish**.

h. Right-click the `com.ibm.uxprofile.constraints` package and select **New > Class**.

i. Fill in the New Java Class wizard dialog as shown below:

TIP: You can use the code assist (`Ctrl-Space`) feature to help select the Superclass.

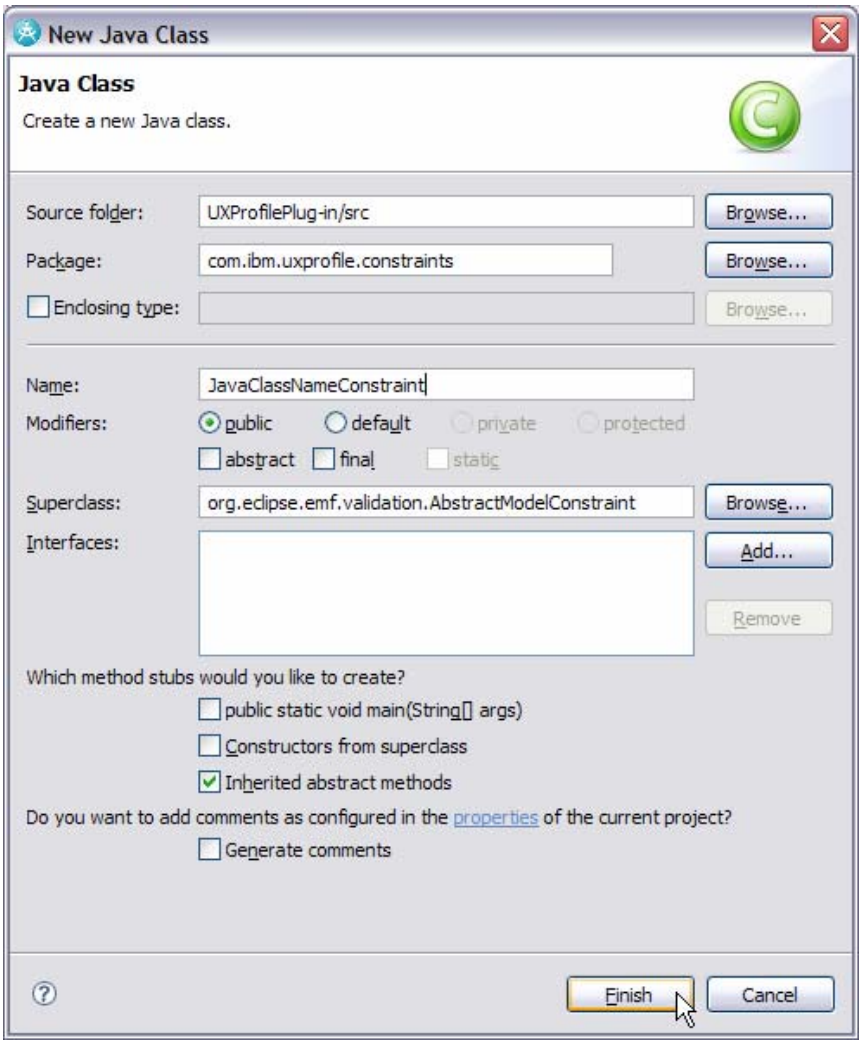


Figure 9-28: *Creating the constraint class*

j. Click **Finish**.

2. Save All.

3. Open the `JavaClassNameConstraint` class.

   a. Replace the existing code of the class with the contents of the `JavaClassNameConstraint.txt` file found in `C:\Workshop\Labs\Inputs.`
   b. Review the code.
4. Select **File > Save All**.

Task 8: Export the Project

In this task, you will export the UXProfilePlug-in project where it will be picked up for additional work by your team.

1. Select **File > Export**.
2. Select **Project Interchange** in the `Other` folder.
3. Click **Next**.
4. Click **Select All** and browse to `c:\Workshop\StudentWork` and save the file as `UXProfilePlug-in_V1.zip.`

Task 9: Import an Updated Version of the Project and Test

In this task, you will import an updated version of the `UXProfilePlug-in` project, modified by other members of your team. Additional stereotypes and enumerations have been added.

1. Switch to the **Plug-in Development** Perspective.
2. Select **File > Import**.
3. Select **Project Interchange** in the `Other` folder.
4. Click **Select All** and browse to `c:\Workshop\Labs\Inputs` and import the file `UpdatedUXProfilePlug-in.zip.`
5. Click **OK** at the Confirm Overwrite dialog.

Task 10: Configure Run-time Workbench

In this task, you will configure a Run-time workbench to use in testing the newly created profile.

1. Switch to the **Plug-in Development** Perspective.
2. Select **Run > Run** from the main menu.
3. On the Run screen, select **Eclipse Application** and click the **New** button (leftmost on the toolbar).
4. Name the configuration `UXProfilePlug-in.`
5. Select the **Configuration** tab and choose the **Use an existing config.ini file as a template** option. Leave the default location.

> *TIP:* This step is critical, because the default Eclipse content option does not provide enough functionality to support an Rational Software Architect test.

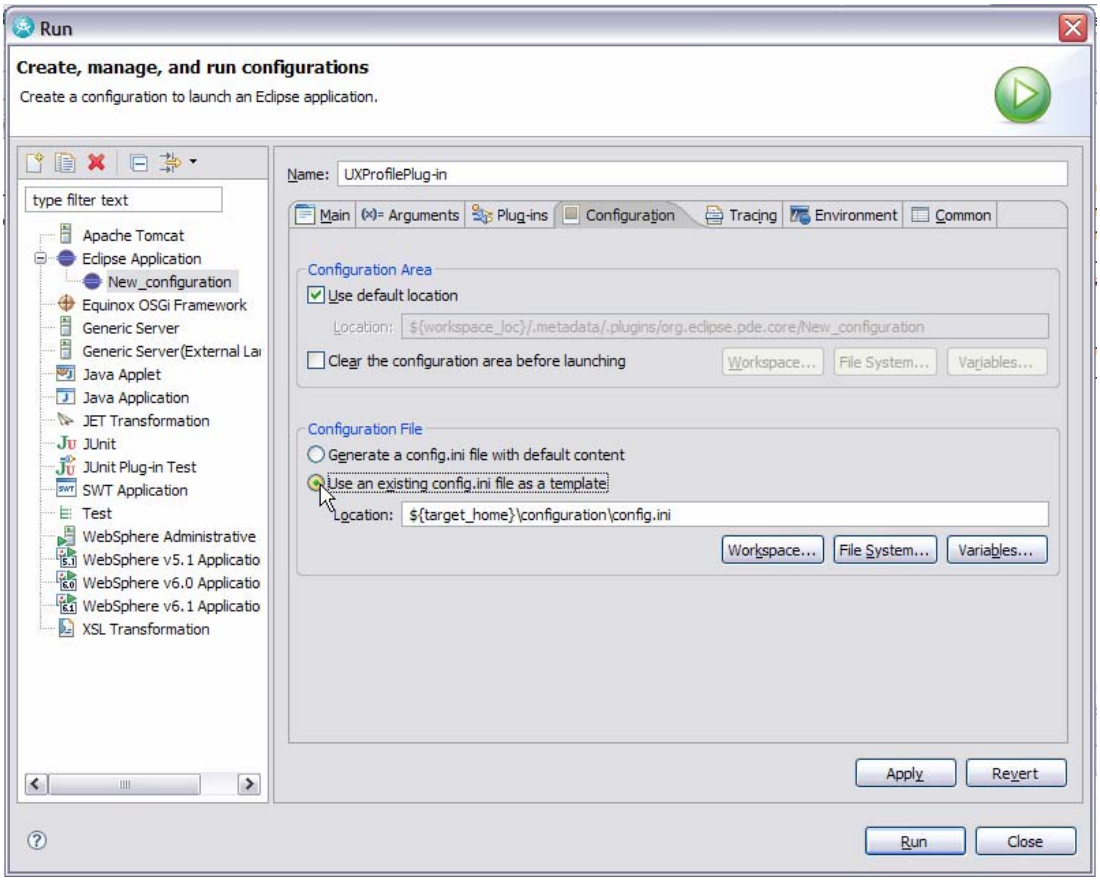


Figure 9-29: *Configuring the run-time workbench*

6. Select **Apply**, then **Run**.

Task 11: Test the Profile

In this task you will test the profile that you and your team created. A model has been partially built for the purposes of testing the profile. You'll complete the model and conduct the test. All of the testing will occur in the **Run-time Workbench**.

1. Close the Welcome screen.
2. Switch to the Modeling perspective in the Run-time workbench.
3. Select **File > Import**.
   a. Select **Project Interchange**. Click **Next**.
   b. Click **Browse** next to the **From zip file** field.
   c. Navigate to the `C:\Workshop\Labs\Inputs` folder and select `ProfileTestProject.zip`. Click **Open.**
   d. Click **Select All** then click **Finish**.

2. Apply the `UXModeling` profile to the test model.
   a. Open the `ProfileTest` model
   b. In the properties view, select the **Profiles Tab**
   c. Click **Add Profile**.
   d. Select `UXModeling`. Click **OK**.

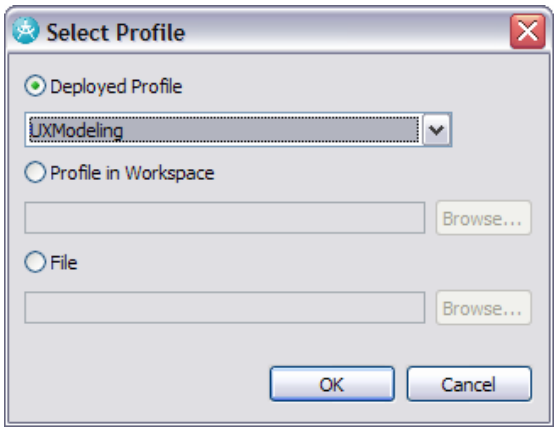


Figure 9-30: *Applying the profile*

   e. Click **OK**.

2. Open the `Main` diagram found within the `com.ibm.strutssample` package or, alternatively, work with the model elements by expanding the `ProfileTest` model.

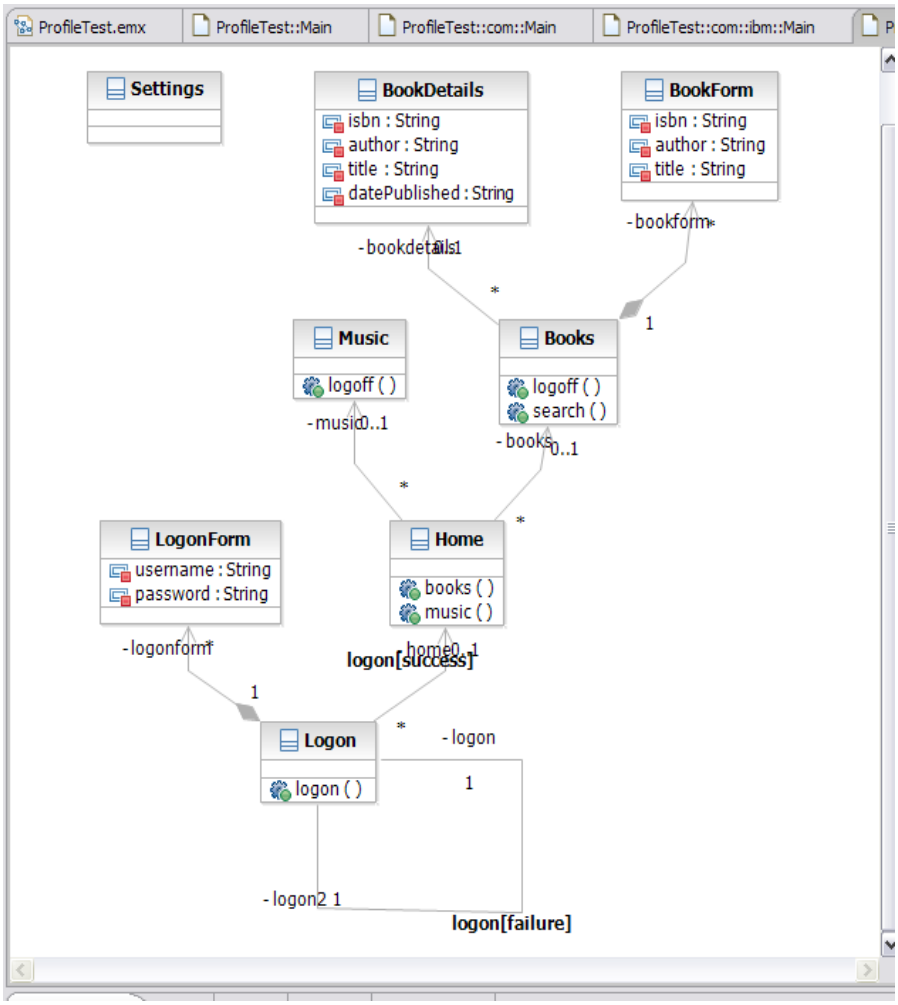


Figure 9-31: *Sample model elements from the* `com.ibm.strutssample` *diagram*

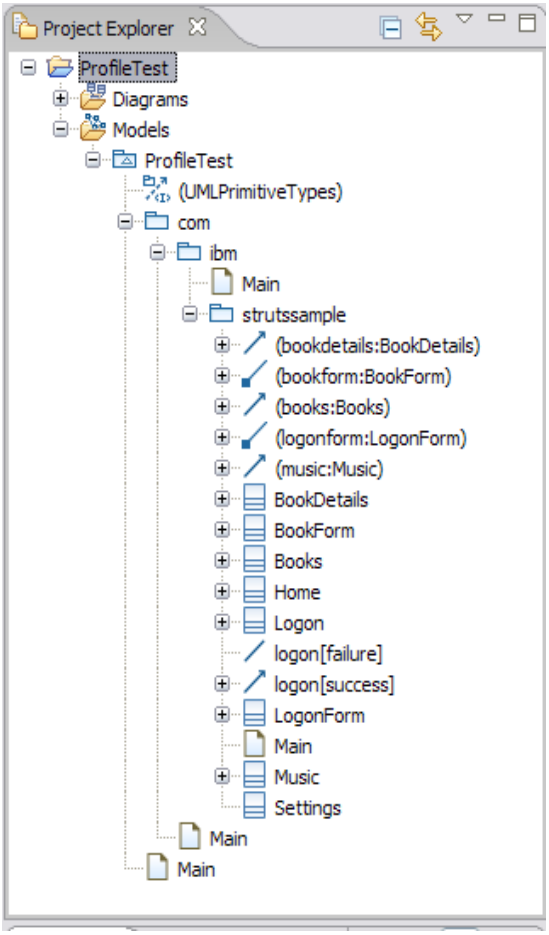


Figure 9-32: *Sample model elements viewed by expanding the ProfileTest model*

Your teammates have applied stereotypes to the `LogonForm, BookDetails, Home and Music` model elements.

3. Apply stereotypes to the `Logon, Home,` and `Music` model elements. Click the element and choose **Apply Stereotypes** from the **Stereotype** tab in the Properties view.

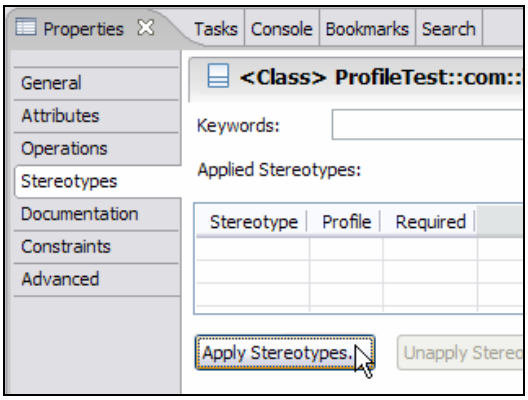


Figure 9-33: *Apply stereotypes to elements from the* ***Properties*** *view*

a. Apply stereotypes to the model elements (`Logon`, `Home`, `Screen`) resulting in a class diagram as shown below:

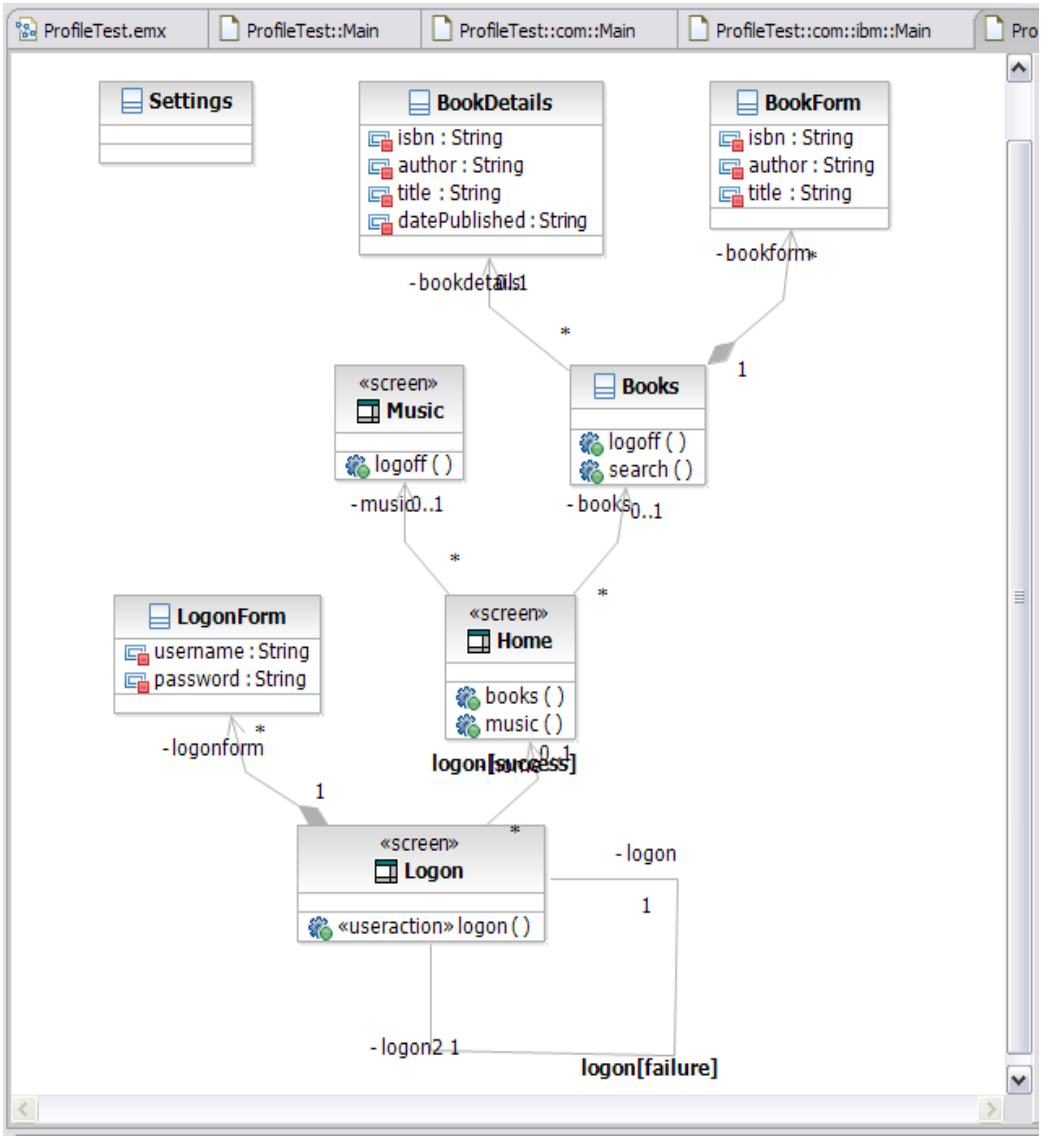


Figure 9-34: *Sample set of elements after applying the profile and adding stereotypes*

b. Select **File > Save All**.

Now let's test the constraint:

4. In the Project Explorer, select com under the **ProfileTest** model, right-click it, and select **Validate.**

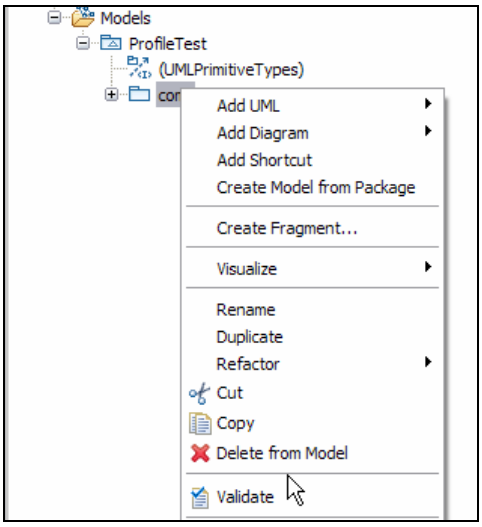


Figure 9-35: *Validate the model against the profile*

a. Note that in the Problems view, a validation error has appeared relating to the `«useraction»` elements.

b. Click the `«useraction»logon()` operation on the `«screen» Logon` class.

c. In the Properties view, switch to the **Advanced** tab.

d. Navigate to the `useraction` node, and then update the `javaclass` property to `com.ibm.test.Logon`.

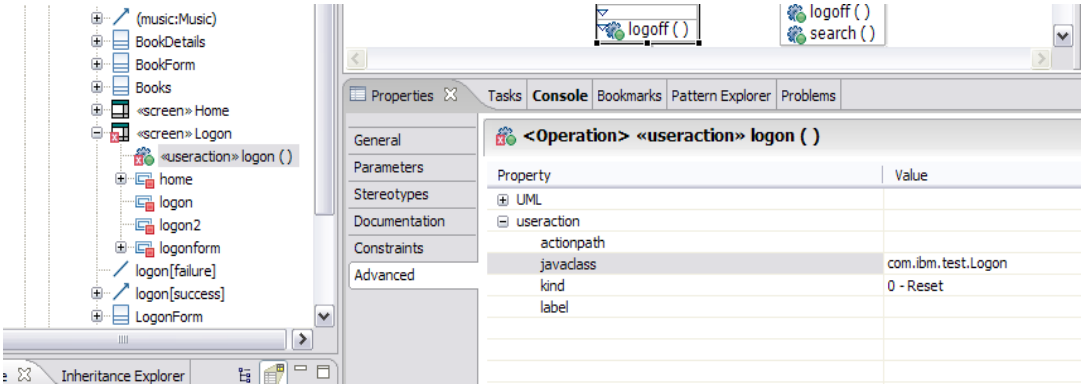


Figure 9-36: *Specifying a value for the javaclass property*

e. Select **File > Save All**.

f. Run the validation again. Note that there should now be no validation errors.

g. Close the Run-time workbench.

Task 12: Release the Profile

In this task you will release the profile, because you've tested it and are certain that this accurately

reflects the elements within this domain. Any profile changes from this point onwards can only be additive changes.

1. Switch to the Modeling perspective.
2. In the Project Explorer view, select the profile.
3. Right-click the `UXModeling` profile and select **Release**.
4. Provide `v1.0.0` as the version number. Click **OK**.

Task 13: Extra Challenges

If time permits during the course, or as a practice challenge for after the course, complete the following tasks.

1. Enhance the profile to include any additional stereotypes, properties, or enumerations that would make the profile more applicable to your organization.
2. Enhance the constraint so that it validates the javaclass property, checking that it has a valid java package and class name.

Lab 10 – Manually Create a Transformation

Objectives

After completing this lab, you will be able to:

- Author, run, and test a custom model-to-model transformation.

Given

The following lab artifacts can be found in the `Inputs` folder for this lab:

- `Code Fragment1.txt`
- `Code Fragment2.txt`
- `Import Statements1.txt`
- `Import Statements2.txt`
- `DEV498v7 Sample Config.launch`

Scenario

In this lab, your team needs to transform a number of source classes from one model to target interface and implementation classes in another model. There must be a realization relationship from the implementation class to the interface, and the implementation class needs copies of the source class operations, while the interface only needs copies of the public source class operations.

Instead of each team member manually performing the transformation, your task is to automate the process and make it available to the entire team.

To simplify the transformation authoring effort, you will use a plug-in template to produce the initial structure of the transformation. When defining the transformation configuration, you will define the rules to convert one type of source element into one or more target elements. You will then need to customize the transformation's behavior by modifying each rule's hot spot. After creating a test project, you will run and test the transformation.

In addition to the conversion rules, you will add a mechanism to traverse the source model elements and run a rule against a UML class that has a specific stereotype applied.

Task 1: Create the Workspace

In this task, you will switch to a new workspace named `M2MTransformationWorkspace` that you will create.

1. From the **File** menu, select **Switch Workspace**.
2. In the Workspace Launcher dialog, replace the displayed text with `C:\Workshop\StudentWork\M2MTransformationWorkspace` and click the **OK** button.
3. Close the **Welcome** screen.

Task 2: Create a New Plug-in Project

In this task, you will create a new Plug-in Transformation project named `MyTransformation` to simplify the authoring effort.

1. On the **File** menu, click **New > Project**
2. Enable **Show All Wizards**.
3. Replace type filter text with Plug-in. Select **Plug-in Project** and click **Next**.

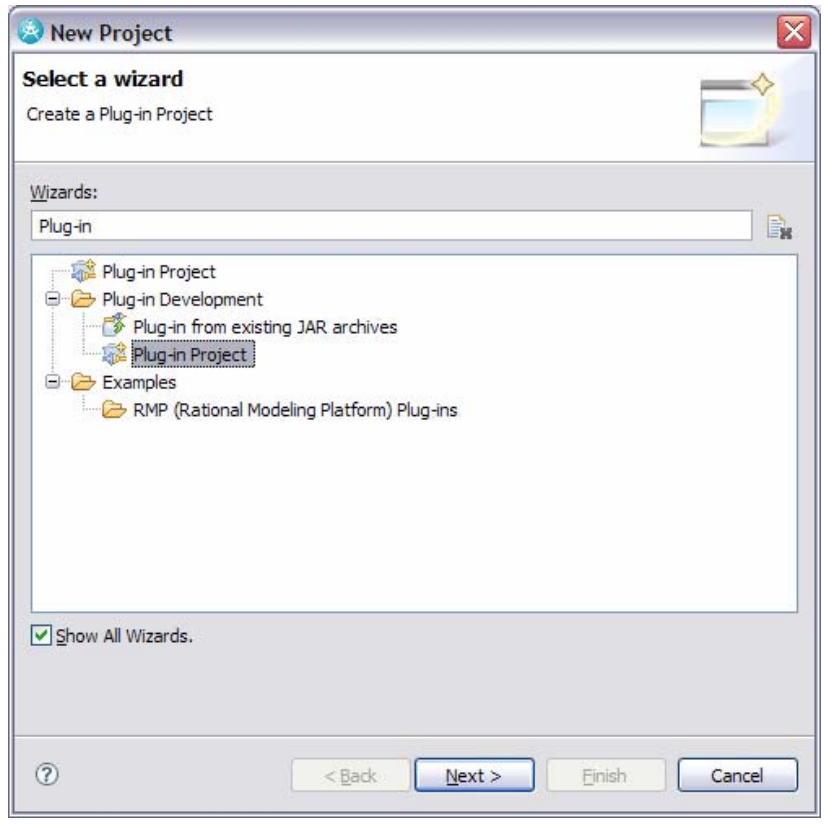


Figure 10-1: *Definition of Transformation Rules*

4. If prompted to confirm enablement of Eclipse Plug-In Development, click **OK**.
5. Name the project `com.ibm.myTransformation` and then click **Next**.
6. On the Plug-in Content screen, keep the defaults and click **Next.**
7. Select the **Create a plug-in using one of the templates** checkbox.
8. Choose **Plug-in with Transformation** and click **Next.**

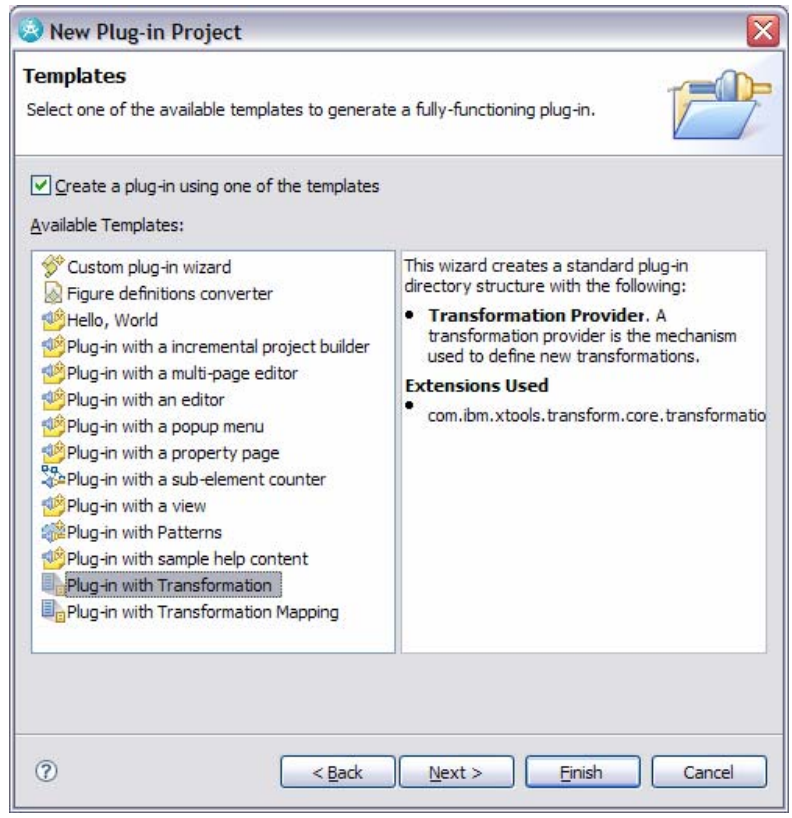


Figure 10-2: *Definition of Transformation Rules*

9. On the New Transformation Provider screen, keep the defaults and click **Next.**
10. On the New Transformation screen, select **UML2** to be the **Source Model Type** and the Target Model Type. Click **Next.**

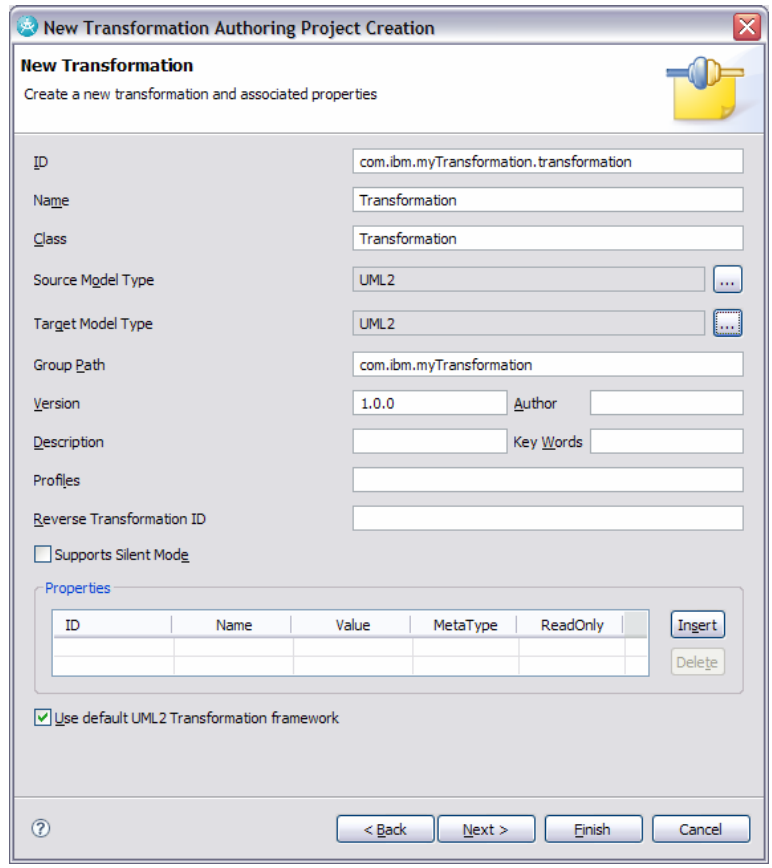


Figure 10-3: *Specify the Source Model Type and the Target Model Type*

11. On the **New Rule Definitions** screen, create new rules as indicated below:

- Click **Insert** to add a class rule. Select **Class** from the list box in the **UML Element Type** column, and enter `ClassRule` in the **Name** column.
- Click **Insert** to add an operation rule. Select Operation from the list box in the **UML Element Type** column, and enter `OperationRule` in the **Name** column.

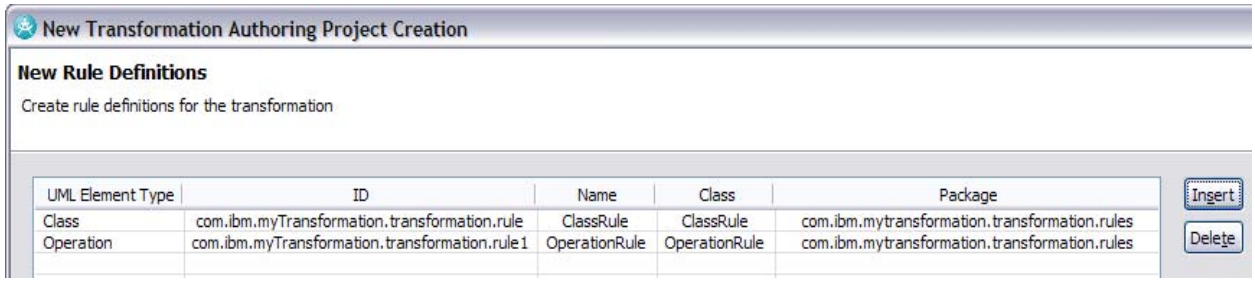


Figure 10-4: *Definition of Transformation Rules*

TIP: The order in which you specify the rules on this screen will impact the order in which they are listed in code. This order will then determine the order in which the rules are executed.

12. Click **Finish.** If prompted to switch to the **Plug-in Development** perspective, click **Yes**.

Task 3: Visualize the Transformation Structure

In this task, you will visualize the initial structure of the transformation.

1. Select the following elements in the **Package Explorer**.

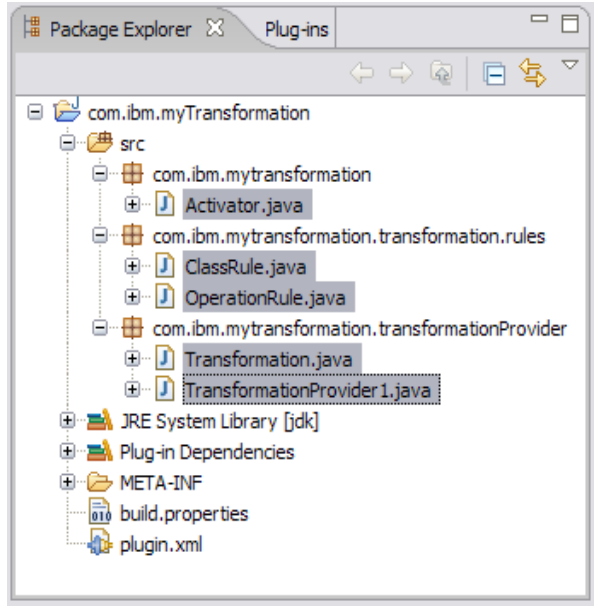


Figure 10-5: *Transformation Structure in the Model Explorer*

2. Right-click and click **Visualize > Add to New Diagram File > Class Diagram**. If asked, click **Yes** to enable Java Modeling activity.

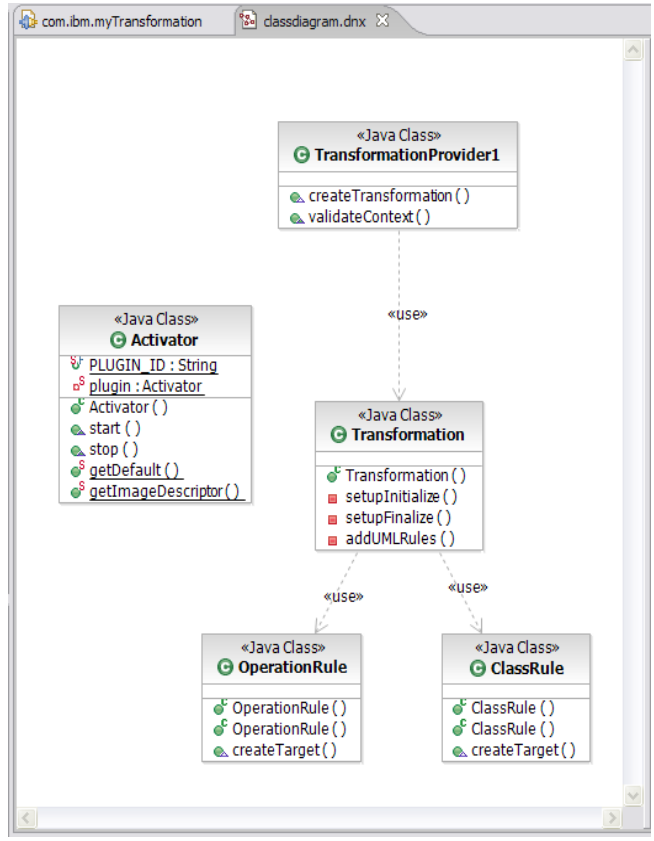


Figure 10-6: *Transformation Structure on a Class Diagram*

Task 4: Edit the Rules

In this task, we will add code to the rules for the pattern. This code will provide the behavior for the pattern.

1. Edit the class rule.

- In the Package Explorer, double-click `ClassRule.java` to open it in an editor.
- Locate the import statements at the top, delete them, and replace them with the contents of `Import Statements1.txt` located in the `C:\Workshop\Labs\Inputs` folder.
- Update the declaration of the class so that

```
public class ClassRule extends AbstractRule
```

becomes:

```
public class ClassRule extends ModelRule
```

- Locate the `createTarget` method, delete the body, and replace it with the contents of `Code Fragment1.txt` located in the `C:\Workshop\Labs\Inputs` folder.
- Review the code.

2. Edit the operation rule.

- In the Package Explorer, double-click `OperationRule.java` to open it in an editor.
- Locate the import statements at the top, delete them, and replace them with the contents of `Import Statements2.txt` located in the `C:\Workshop\Labs\Inputs` folder.
- Update the declaration of the class so that

```
public class OperationRule extends AbstractRule
```

  becomes:

```
public class OperationRule extends ModelRule
```

- Locate the `createTarget` method, delete the body, and replace it with the contents of `Code Fragment2.txt` located in the `C:\Workshop\Labs\Inputs` folder.
- Review the code.

3. From the **File** menu, select **Save All**.

Task 5: Configure Run-time Workbench

In this task, you will configure a Run-time workbench to use in testing the newly created transformation. There are two approaches that you can take when setting up your run-time workbench. The first approach is to spend time to create a custom list of plug-ins to have included within the run-time workbench. This can take some time to develop, but once created can significantly speed up the launching of the run-time workbench. The second approach is to accept the default list of plug-ins. This is quick to configure, but the run-time workbench will launch more slowly.

1. Set up a stripped down configuration for the runtime workbench. This will reduce workbench launch and debug times.

- Select **File > Import**.
- Select **File system**. Click **Next**.
- Click **Browse** and navigate to `Workshop\Labs\Inputs` and select `DEV498v7 Sample Config.launch`.
- Specify `MyTransformation` as the value for the **Into folder** field.

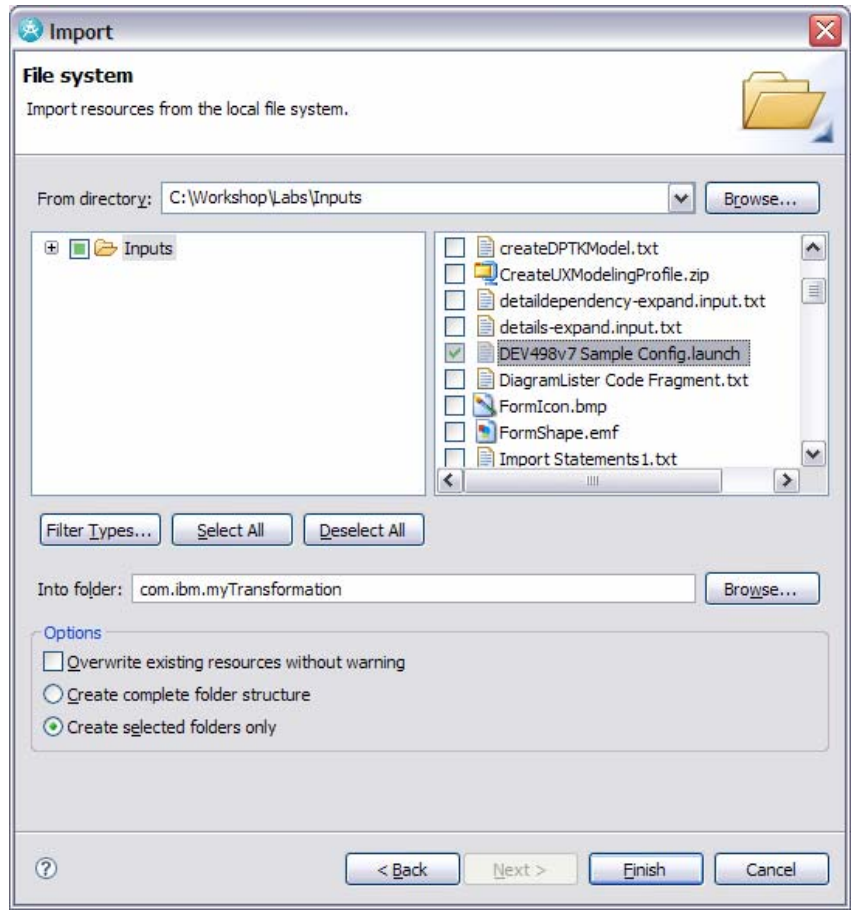


Figure 10-7: *Import the launch file*

- Click **Finish**.
- From the **Run** menu, select **Run**
- In the Create, manage, and run configurations dialog, select **DEV498v7 Sample Config** under Eclipse Application.

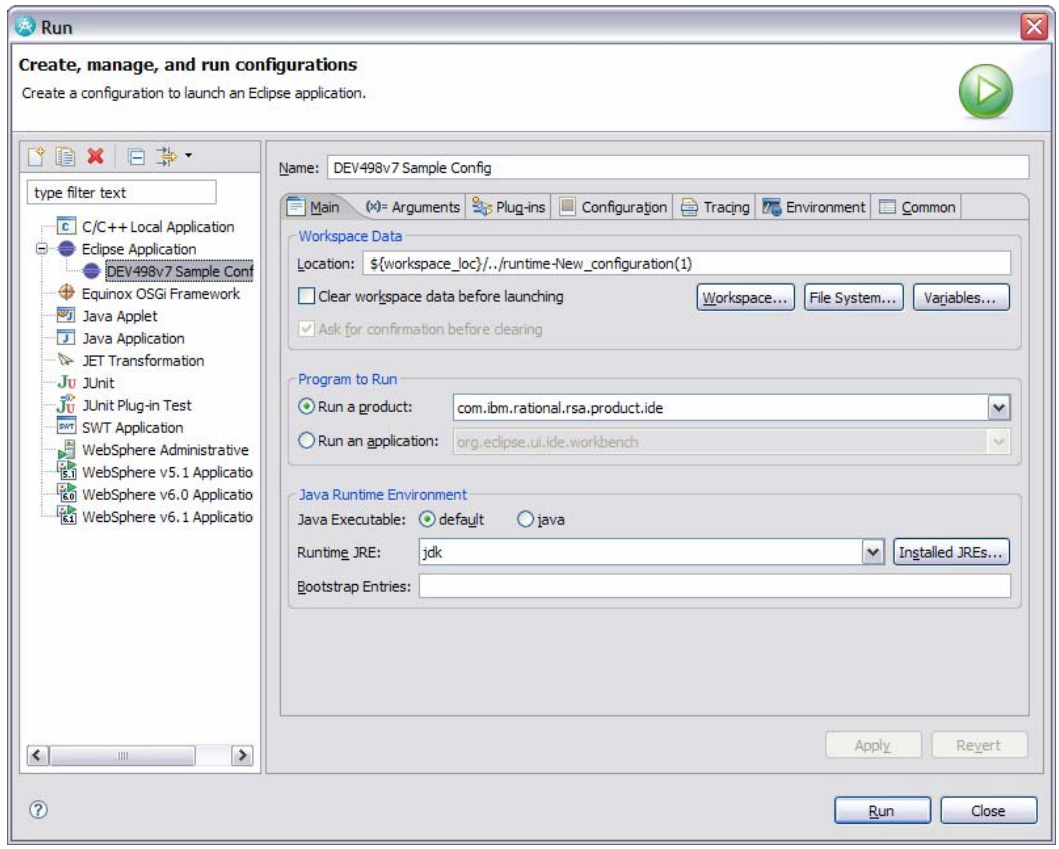


Figure 10-8: *Selected the sample configuration*

- Click **Run**.

2. **Optional**: If you would like to use a full configuration for the runtime workbench, follow these steps in place of Step 1.
   - From the **Run** menu, select **Run**
   - In the Create, manage, and run configurations dialog, select **Eclipse Application** and click the **New launch configuration** button.
   - Name the new configuration `Full Configuration`
   - Click **Apply**.

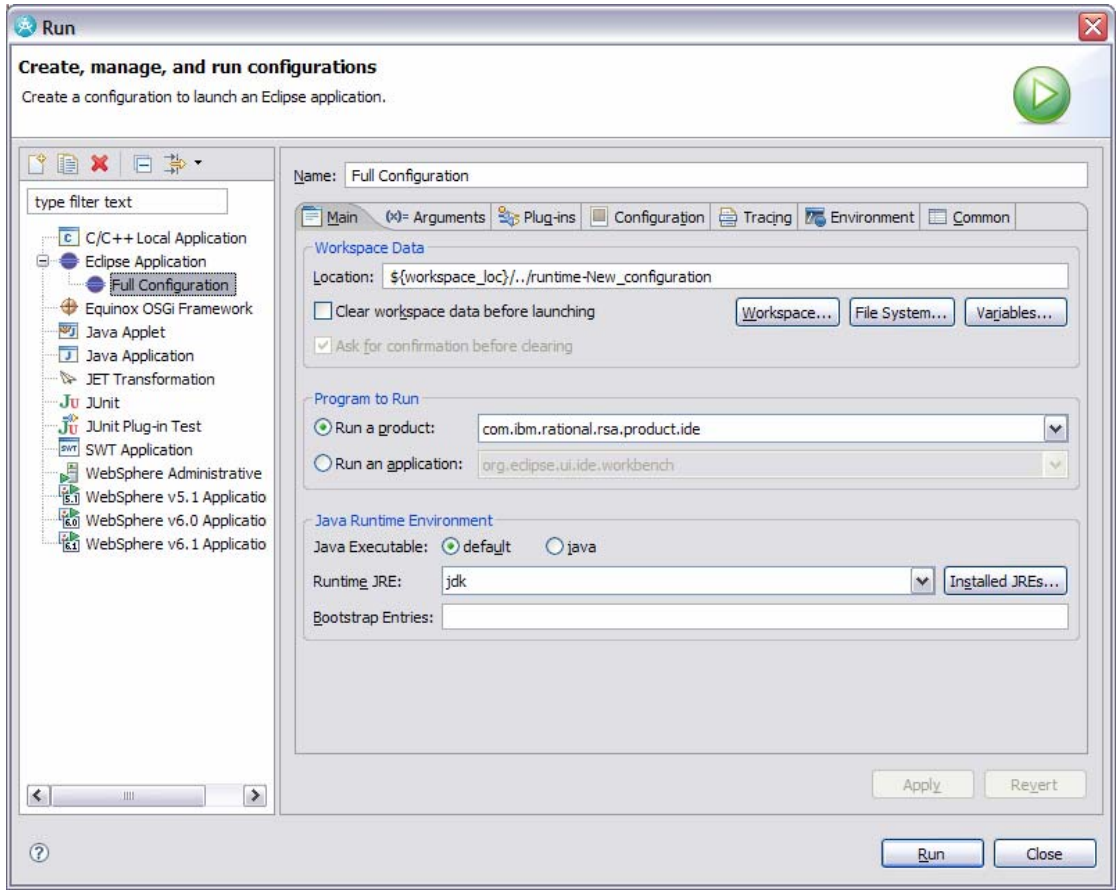


Figure 10-9: *Configuring a fill configuration runtime workbench*

- Select **Full Configuration** in the Configurations list and click **Run**. Because this is a full version, it will take several minutes to complete the launch of the Run-time Workbench.

TIP: Yes, you can create multiple configurations. When it comes time to test, you will need to select which configuration you would like to use for your test.

Task 6: Create a Test Project

In this task, you will use the run-time workbench to test the pattern that you've built.

1. Using the Run-time Workbench, create a test UML Modeling Project named `TransformationTest` based on the Blank Model template:

- Close the Welcome screen.
- From the **File** menu, click **New > Project**
- Replace type filter text with UML. Select **UML Project**, and click **Next.**

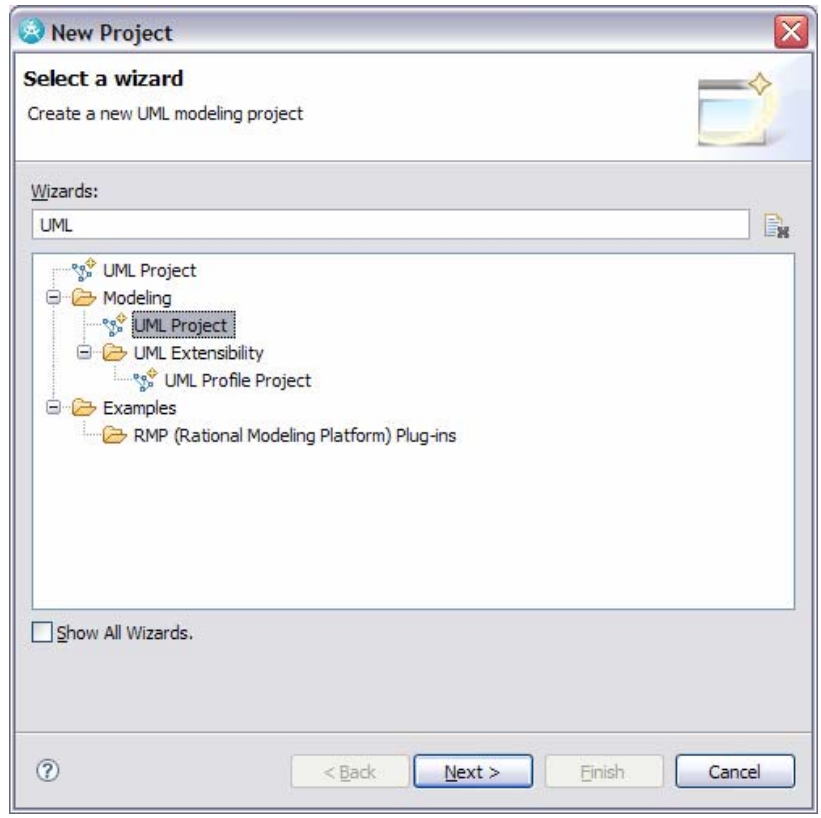


Figure 10-10: *Creating a new UML Project*

- Name the project `TransformationTest` and click **Next**.
- Under **Templates**, select **Blank Model**, change the file name to `SourceModel`, and click **Finish**.
- If prompted to switch to the Modeling Perspective, click **Yes**.

2. Create a class named `Employee` and add three private operations; **readEmail**, **answerPhone**, and **performWork**. Add one public operation **reportToManager (name:String)**.

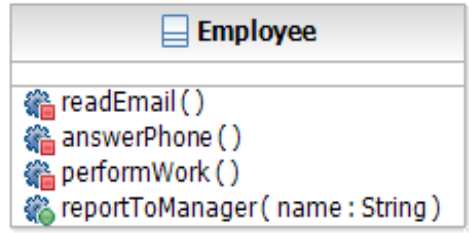


Figure 10-11: *Employee class*

3. To the `TransformationTest` project, add a new UML Model named `TargetModel` based on the Blank Model template.

- On the **Project Explorer**, select the `TransformationTest` project, right-click and select **New > UML**

Model.

- Click **Next**.
- Under **Templates**, select **Blank Model**, change the file name to `TargetModel` and click **Finish**.

Task 7: Run the Transformation

In this task, you will configure and run the transformation.

1. From the **Modeling** menu, select **Transform > New Configuration**.
2. Select `Transformation` from under `com.ibm.myTransformation` folder.
3. Name the configuration `MyTransformationConfiguration` and click **Next**.

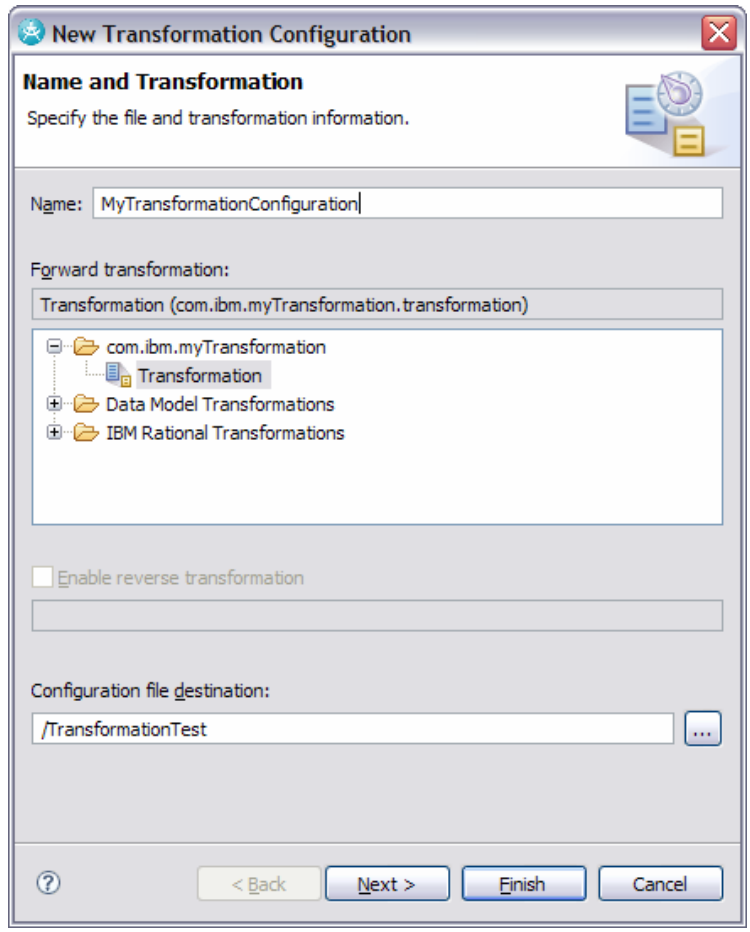


Figure 10-12: *Creating the new transformation configuration*

4. On the Source and Target screen, specify `SourceModel` as the **Selected source** and `TargetModel` as the **Selected target**.

TIP: Ensure that you select the model and not the model file. The easiest way to discern between the two is

that the model file has an emx extension.

5. Click **Finish**.
6. In the Project Explorer, right-click the `MyTransformationConfiguration.tc` file and click **Transform > Transformation**.
7. Explore the results in **TargetModel.**

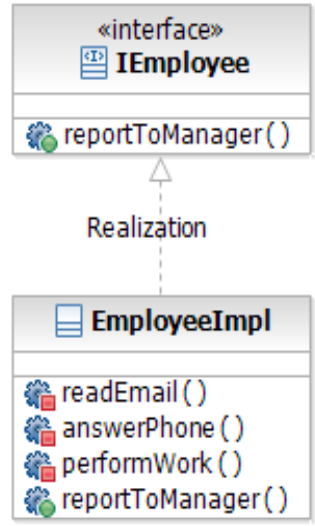


Figure 10-13: *Resulting elements in the TargetModel*

8. Optionally, you can add another public operation to the `Employee` class in `SourceModel`, for example **reportToManager (id:Integer)**, and re-run the transformation.

Task 8: Add a New Rule

In this task, we will add a new rule to the transformation. This rule will work with the properties (attributes) of a class – and its output will depend on keywords that have been applied.

1. Close the run-time workbench and switch back to the host workbench.
2. Add a new class named PropertyRule to MyTransformation.transformation.rules package:
   - In the Package Explorer, right-click on the `MyTransformation.transformation.rules` package and select **New > Class**.
   - Populate the New Java Class dialog as shown in the screen capture below:

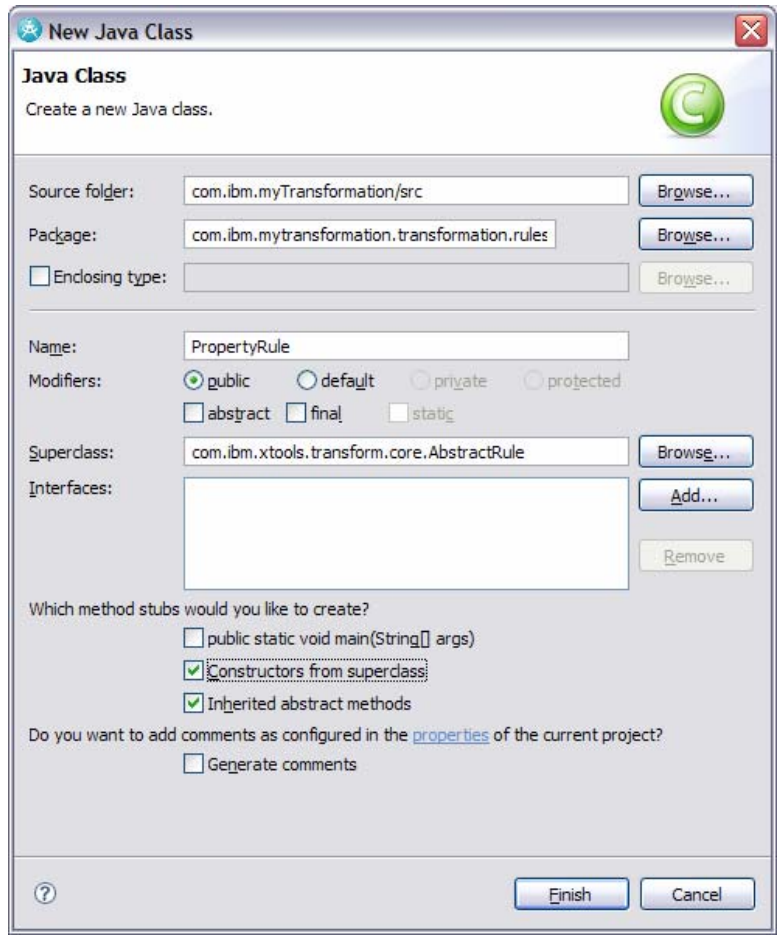


Figure 10-14: *Adding the new Property Rule*

TIP: Make sure that you have selected **Constructors from superclass**.

- Click **Finish**.

TIP: You may recall from earlier that we changed the Supeclass for our rules from AbstractRule to ModelRule. The reason for this change is that the ModelRule class provides support for modifying a target UML model. In the case of this rule, we will not be modifying a UML model, so we can keep AbstractRule as the Superclass.

3. Add code to the new rule:

- Update the code in the `createTarget` method so that we can tell when this method is called:

```
protected Object createTarget(ITransformContext arg0) throws Exception
{
        NamedElement element = (NamedElement) arg0.getSource();
        EList keywords = element.getKeywords();
```

```
            if(keywords.isEmpty())
            {
                  System.out.println(element.getName() + " FunnyProperty
Keyword has NOT been applied");
            }else if(keywords.contains("MyFunnyProperty")){
                  System.out.println(element.getName() + " FunnyProperty
Keyword has been applied");
            }
            return null;
      }
```

- Right-click in the editor for the class and click **Source > Organize Imports** to add required import statements.

4. Connect the new rule into the transformation:

- Add the following line to the end of the `addUMLRules(UMLKindTransform transform)` method of the `com.ibm.mytransformation.transformationProvider.Transformation` class.

```
transform.addByKind(UMLPackage.eINSTANCE.getProperty(), new
PropertyRule("MyTransformation.transformation.rule2", "PropertyRule"));
```

- Right-click in the editor for the class and click **Source > Organize Imports** to add any required import statements.
- Select **File > Save All**.

Task 9: Test the New Rule

In this task, we will test the new rule that we added to the transformation in the previous task. In this case we will launch the runtime workbench in Debug mode.

1. Launch a run-time instance of the workbench:

- Select **Run > Debug**.
- Select **DEV498v7 Sample Config** from the Configurations pane, and then click **Debug**.
- If you are prompted to switch to the Debug Perspective in the development workspace, click **OK**.

2. Test the updated transformation:

- In **Model Explorer** open the `SourceModel`.
- Create a new attribute named `Salary` on the `Employee` class. Add a **Keyword** to it named `MyFunnyProperty`.

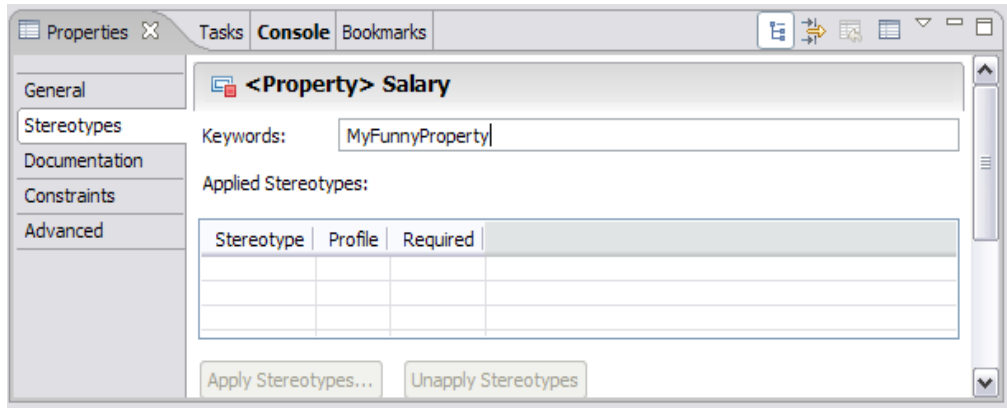


Figure 10-15: *Specifying a keyword on the Salary attribute*

- Create another attribute named `Paydate` on the `Employee` class. *Do not* specify a keyword on this attribute.
- Open the `TargetModel` model.
- Re-run the transformation configuration MyTransformationConfiguration. The transformation should produce output in the **Console** view within the host workbench.

TIP: If the console is not visible, click **Window > Show View > Other** , and then select **Basic > Console**.

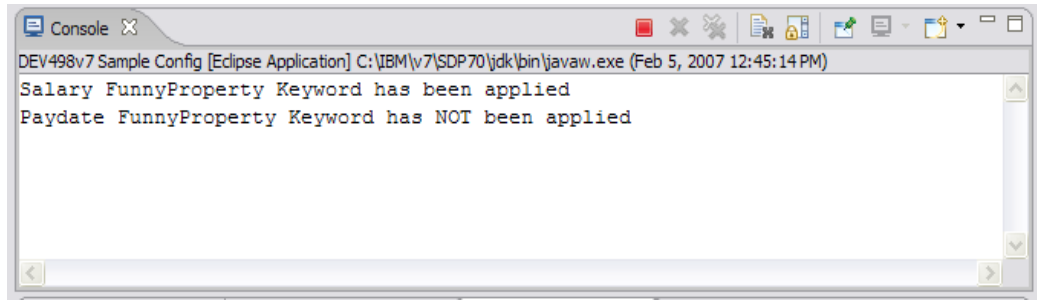


Figure 10-16: *Output written to the Console view within the Host workbench.*

- When you have finished testing and debugging the transformation, close the run-time workbench.

Task 10: Extra Challenges

If time permits during the course, or as a practice challenge for after the course, complete the following task.

1. Work in debug mode, set a breakpoint in the createTarget methods of the rules in the transformation. Run the transformation again and walk-through the code using the debugger.

Lab 11 – Create a Model to Model Transformation

Objectives

After completing this lab, you will be able to:

- Author, run, and test a custom model-to-model transformation.

Given

The following lab artifacts can be found in the `Inputs` folder for this lab:

- `OperationMapping.mapping`
- `Class2InterfaceCustomNameTransform.txt`
- `FindElementUtility.java`

Scenario

In this lab, your team needs to transform a number of source classes from one model, to target interface and implementation classes in another model. There must be a realization relationship from the implementation class to the interface, and the implementation class needs copies of the source class operations, while the interface only needs copies of the public source class operations.

Instead of each team member manually performing the transformation, your task is to automate the process and make it available to the entire team.

You will use the Transformation with Model mapping capabilities of Rational Software Architect to define how the source model elements will be mapped to the target model. Then you will generate and run the transformation from this model mapping.

Task 1: Create the Workspace

In this task, you will switch to a new workspace named `M2MTransformationWorkspace` that you will create.

1. From the **File** menu, select **Switch Workspace**.
2. In the Workspace Launcher dialog, replace the displayed text with `C:\Workshop\StudentWork\M2MTransformationWorkspace` and click the **OK** button.
3. Close the **Welcome** screen.

4. Switch to the Modeling perspective
5. Make sure that the XML Developer capability is enabled. Go to **Window > Preferences** and under **General > Capabilities**, select **XML Developer**.

Task 2: Create a New Plug-in Project with Transformation Mapping

In this task, you will create a new Plug-in Transformation project named `GeneralizeClasses` to simplify the authoring effort.

1. On the **File** menu, click **New > Project**
2. Replace `type filter text` with `Plug`
3. Select **Plug-in Project** and click **Next**.

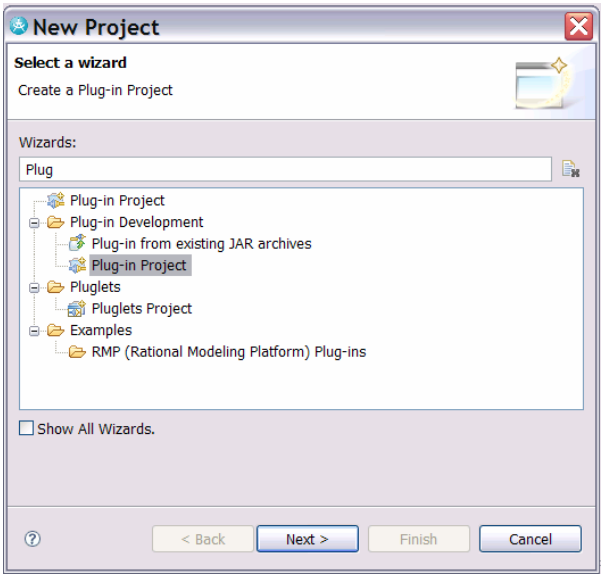


Figure 11-1: *Create a new Plug-in Project*

4. Name the project `Generalize Classes` and then click **Next**.
5. Review the **Plug-in Content** screen, leave all the defaults, and click **Next**.
6. On the Templates screen, select **Create a plug-in using one of the templates**
7. Select **Plug-in with Transformation Mapping** and click **Next**.

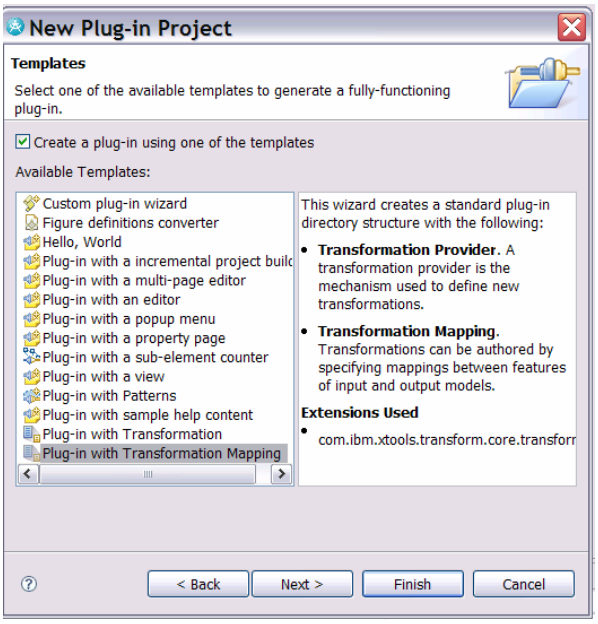


Figure 11-2: *Specify the template to use*

8. On the **New Transformation Mapping** screen, select the **Add Model** button next to **Input models.**
9. On the **Load Resources** dialog, select the **Browse Registered Packages** button.
10. Replace "*" with "*UML" and then select the package `http://www.eclipse.org/uml2/2.0.0/UML`, then **OK** twice. This selects the UML.ecore model for the input model.

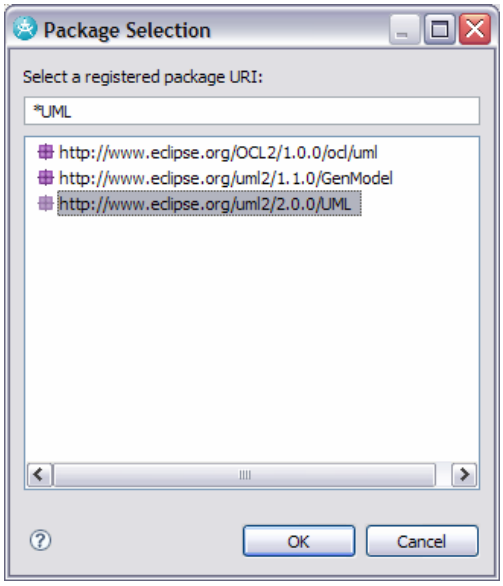


Figure 11-3: *Select the input model to use*

11. Repeat steps 8-10 to select UML.ecore for the **Output model**.

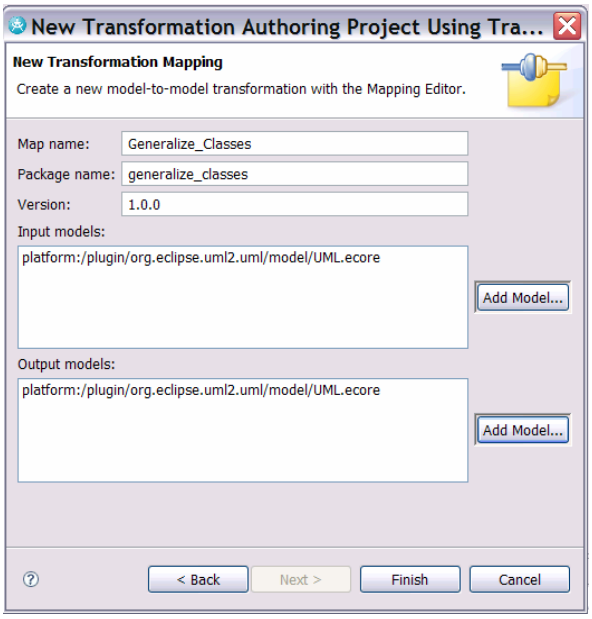


Figure 11-4: *Specifying the input and output models*

12. Click **Finish.** If asked to switch to the Plug-in Development perspective, select **No.**

Task 3: Create the Class to Class Mapping

In this task, you will create the first mapping to be used in the transformation. You will create a total of 4 mappings before you run the first version of the transformation.

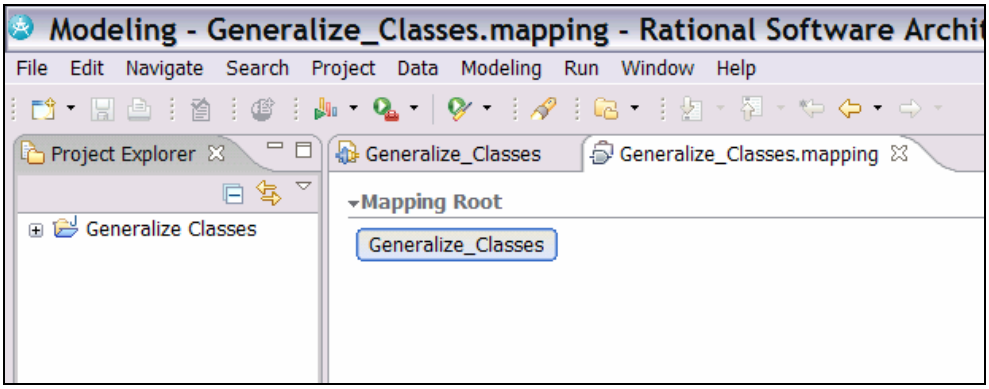


Figure 11-5: *The mapping editor*

1. A file called `Generalize_Classes.mapping` is created and opened in the mapping editor**.**

2. Right-click the **Generalize_Classes** button and select **Create Map**. Name the map **Class2Class**.

3. The mapping editor toolbar displays with your new map.

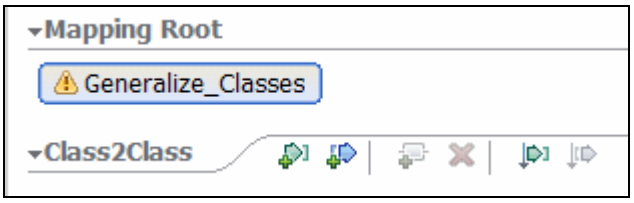


Figure 11-6: *Toolbar within the mapping editor*

4. Select the leftmost button to add an input element.
5. When the Add Input screen displays, simply start typing the letters `cla` and the UML Class will be highlighted. Select **OK.**

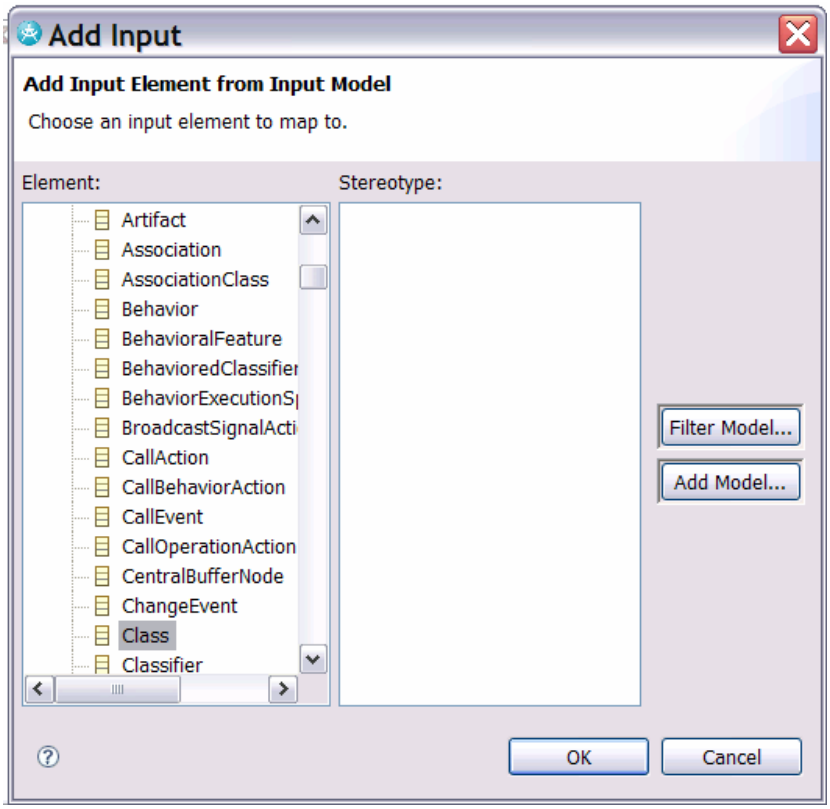


Figure 11-7: *Adding an Input element*

6. Select the **Add Output** button (located to the right of the **Add Input** button).
7. When the **Add Output** screen displays, simply start typing the letters `cla` and the UML Class will be highlighted. Select **OK.**

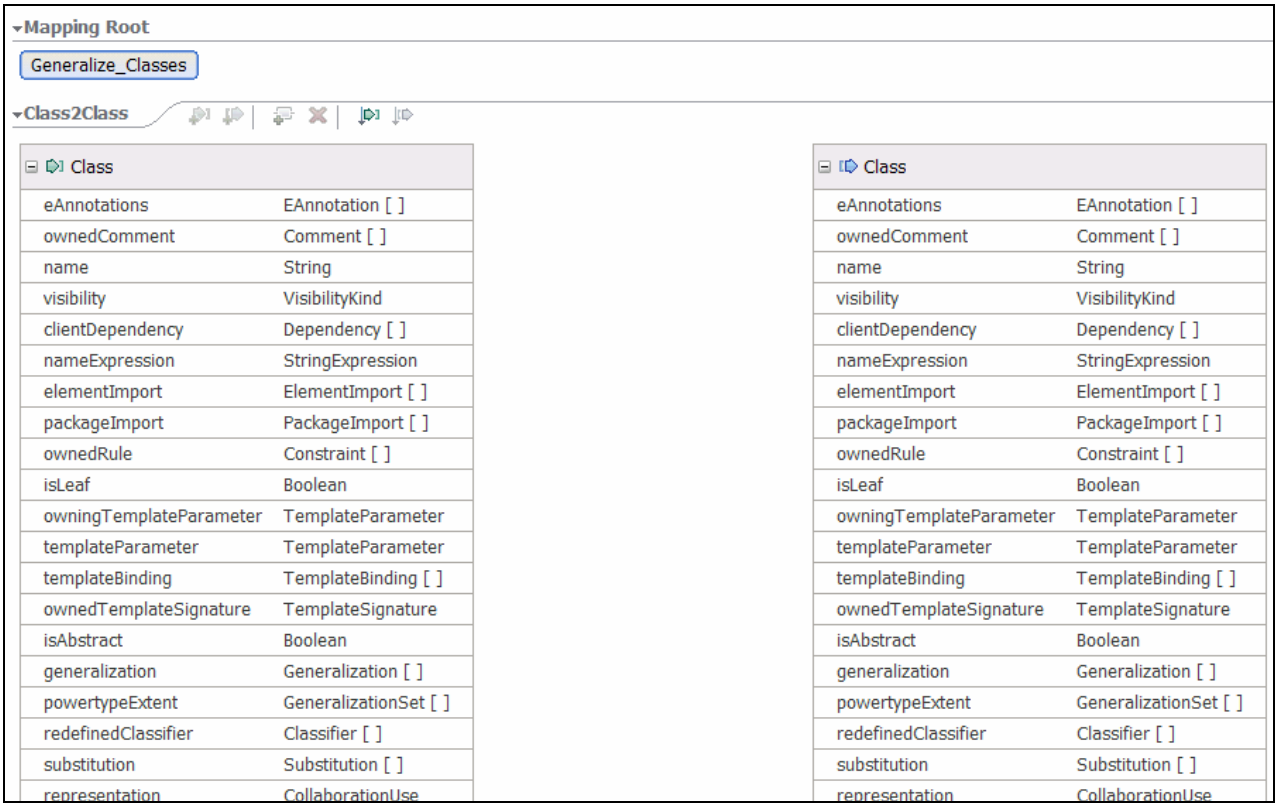


Figure 11-8: *Input and output elements added to the mapping*

Now you are ready to define the transformation between the input and output elements. For the first part of the exercise, you will simply create a new class in the target model with the same name as the class in the source model. You will come back and add the mapping of operations later in this exercise.

8. Hover the cursor over the name property of the input class until a handle appears. Select this handle and drag it onto the name element of the target class. The result will be a transformation of type Move. You could also think of it as a copy.

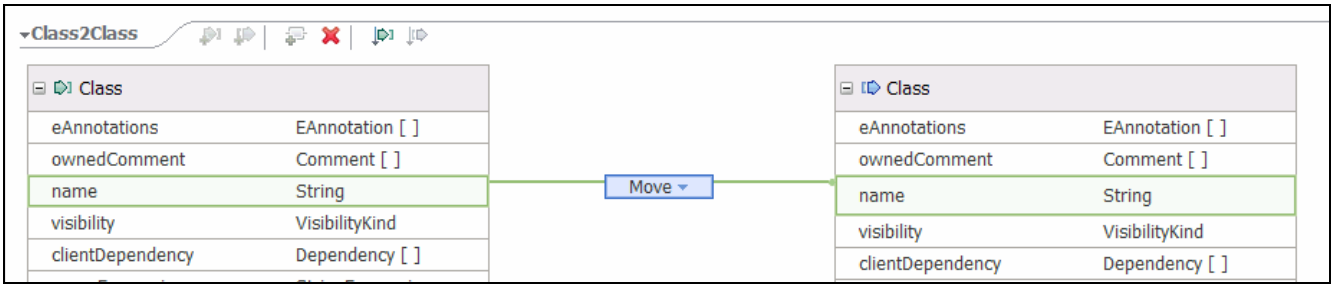


Figure 11-9: *Creating a Move transformation between the elements*

Task 4: Create the Class to Interface Mapping

1. Using the skills you learned in Task 3, create a new mapping in Generalize_Classes. Call this mapping `Class2Interface`.
2. Select the input element to be a UML Class, and the output element to be a UML Interface.
3. Create a transformation between the name of the input Class and the name of the output Interface.
4. Instead of a simple copy of the name, though, you want to rename the interface. Select the **Move** and use the down arrow to change it to **Custom**.
5. Make sure that **Custom transformation** is selected, and then select the **Details** tab in the **Properties** view.
6. In the Code: area, be sure that I**n-line** is selected and enter the following code:
   `Interface_tgt.setName("I"+Class_src.getName());`
7. As you enter code, try out the code completion with `Ctrl-Space`.

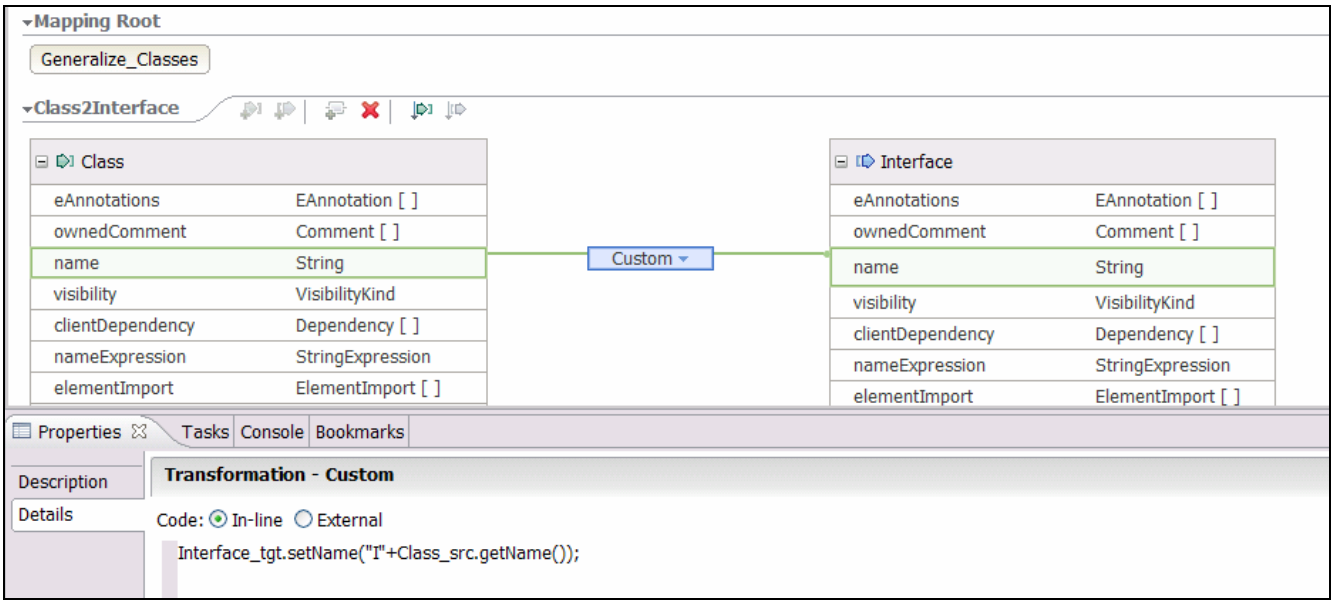


Figure 11-10: *Custom mapping between the elements*

8. Enter `Ctrl-Shift-S` to save all of your work so far.

Task 5: Create the Package to Package Mapping

1. Create a new mapping in Generalize_Classes. Call this mapping `Package2Package`.
2. Select the input element to be a UML **Package** and the output element to be a UML **Package**.
3. Create a transformation between the **name** of the input Package and the **name** of the output Package.
4. Create a transformation between the packagedElement of the input package and the packagedElement of the output Package. Because the packageElement is an array, the mapping tool will create a transformation of type **Submap**.

5. With the Submap transformation selected, in the Properties view, **Details** tab, make sure that the value for the Map is `Class2Class`.

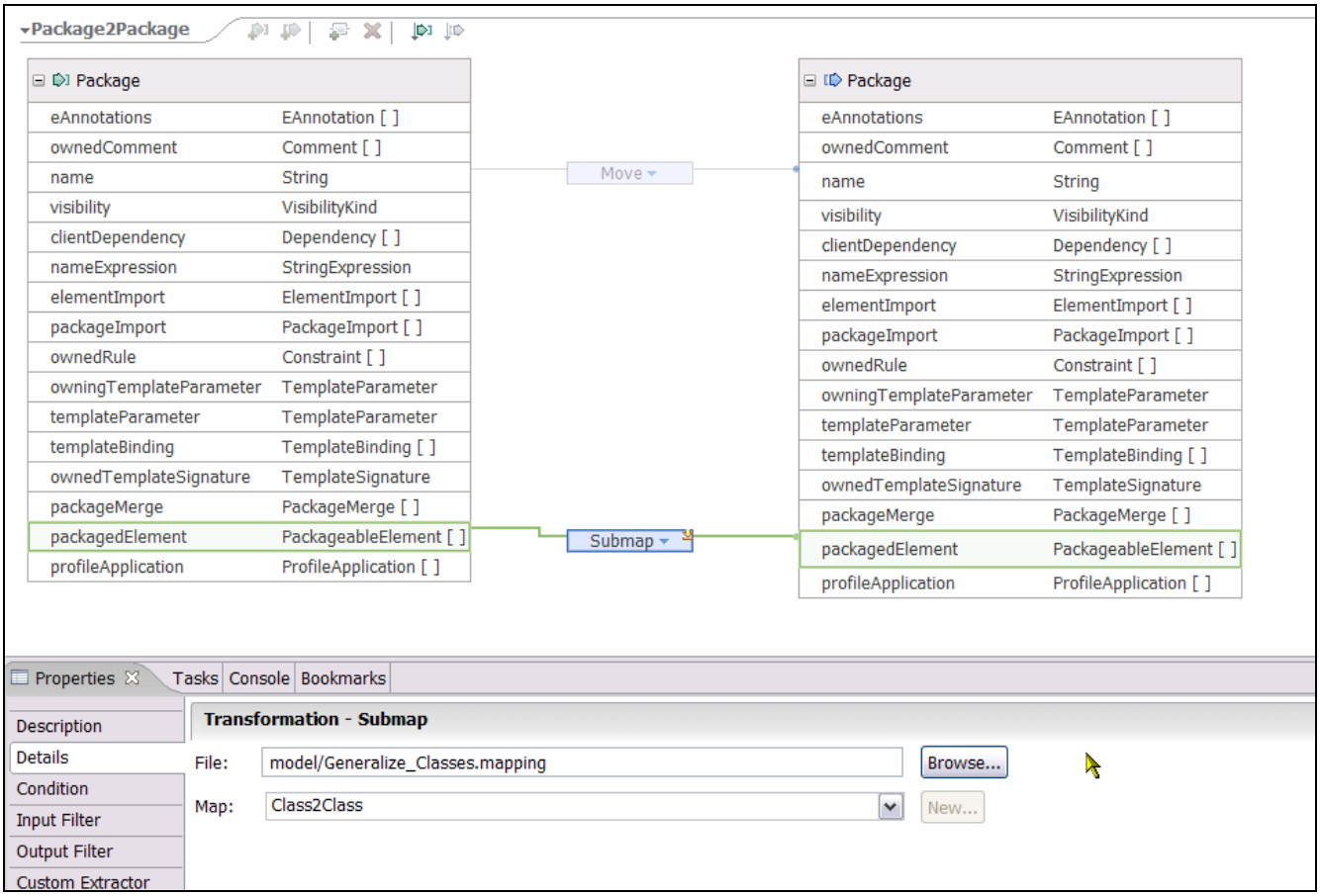


Figure 11-11: *A submap between the elements*

6. Create another Submap between the packagedElements, set its **Map** to `Class2Interface`.
7. Create one more Submap between the packagedElements, so that your transformation will handle nested packages, and set its **Map** to `Package2Package`.
8. You should now have three Submaps between the packageElements of the source and target.

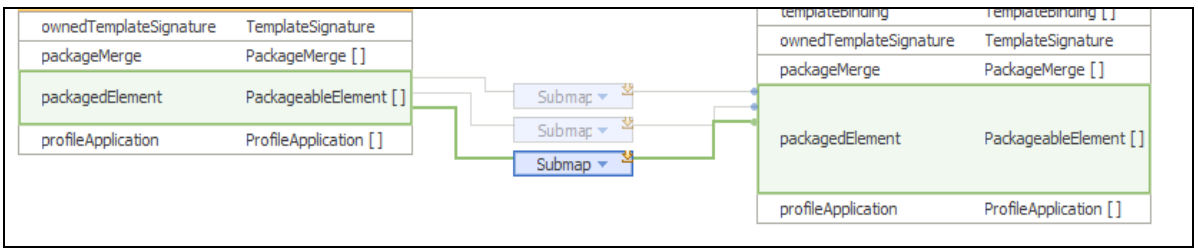


Figure 11-12: *The three submaps between packagedElement*

Task 6: Create the Model to Model Mapping

1. Create a new mapping in Generalize_Classes. Call this mapping `Model2Model`.
2. Select the input element to be a UML **Model** and the output element to be a UML **Model**.
3. Create a transformation between the name of the input Model and the name of the output Model.
4. Instead of a simple copy of the name, though, you want to rename the model. Select the **Move** and use the down arrow to change it to **Custom**.
5. Make sure that the Custom transformation is selected, then select the **Details** tab in the **Properties** view.
6. In the Code: area, be sure that in-line is selected and enter the following code:
```
Model_tgt.setName(Model_src.getName()+"TgtModel");
```
7. Add a **Submap** transformation from the source Model **packagedElement** to the target Model **packagedElement** and make sure its map is `Package2Package`.
8. In the **Outline** view, right-click on the Model2Model mapping and select **Execution Order > Move Up**. Repeat until the Model2Model mapping is at the top of the list. Repeat for each mapping until the list of mappings is in the following order:

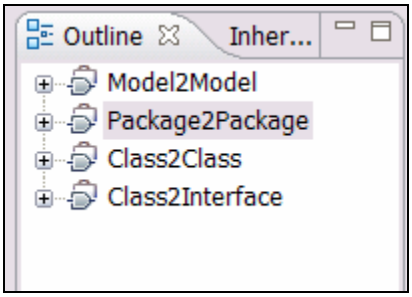


Figure 11-13: *The mappings in the Outline view*

Task 7: Generate the Transformation Code

1. Enter `Ctrl-Shift-S` to save all of your work so far.
2. Before you generate code, review the files that are in the project so far by opening the nodes of the project in the **Project Explorer**. All of these were created when the project was created and you have been editing the `.mapping` file.

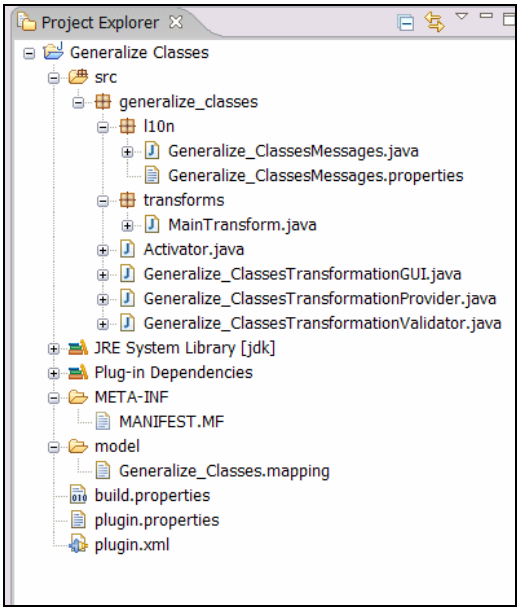


Figure 11-14: *Files in Project Explorer before generating code*

3. In the Mapping Editor, right-click the surface to the right of the **Generalize_Classes** button and click **Generate transformation source code** from the context menu.
4. Review the transformation files that have been generated.

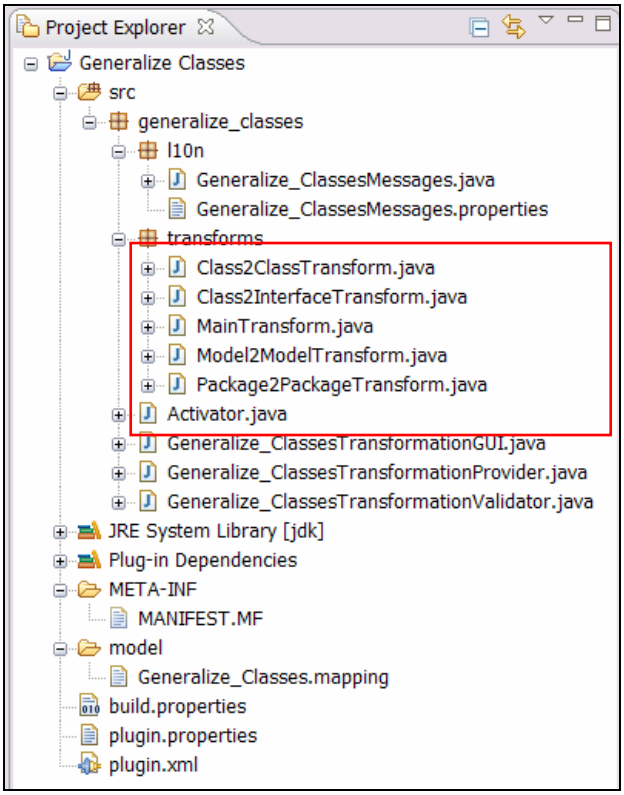


Figure 11-15: *Project Explorer after generating code*

5. Save your work.

Task 8: Configure Run-time Workbench

In this task, you will configure a Run-time workbench to use in testing the newly created transformation.

1. Switch to the Plug-in Development Perspective.
2. Select **Run** > **Run** from the main menu bar.
3. On the Run screen, select **Eclipse Application** and click the **New** button (leftmost on the toolbar).
4. Select the **Configuration** tab and set the **Configuration File** setting to **Use an existing config.ini file as a template**. Leave the default location.

TIP: This step is critical, as the default Eclipse content option does not provide enough functionality to support a Rational Software Architect test.

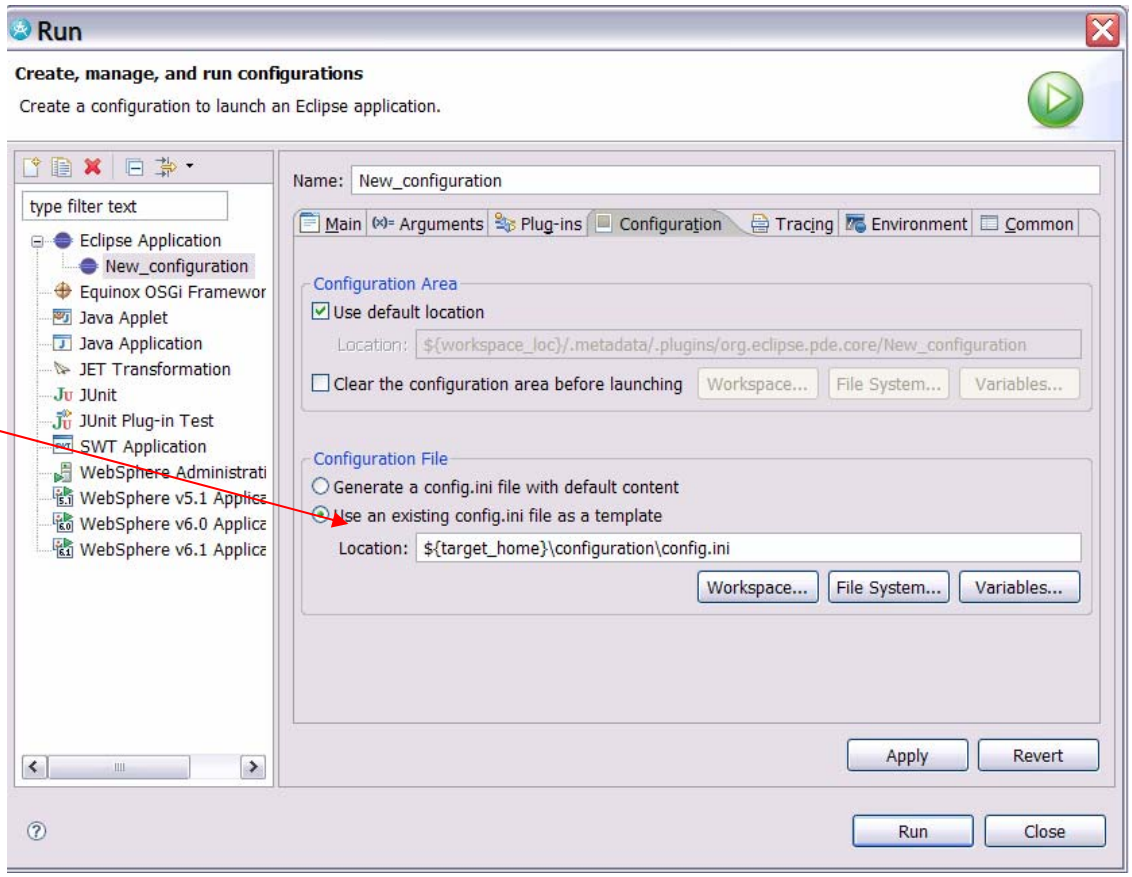


Figure 11-16: *Configuring a runtime configuration*

5. Select **Apply**, then **Run**.

Task 9: Create a Test Project

In this task, you will be using the Run-time Workbench to test the newly created transformation.

1. Close the Welcome screen.
2. Switch to the Modeling perspective.
3. Create a test UML Modeling Project named TransformationTest based on the Blank Model template:
   - From the **File** menu, click **New > Project**
   - Select **UML Project**, and click **Next.**
   - Name the project `TransformationTest`, keep the remaining defaults, and click **Next**.

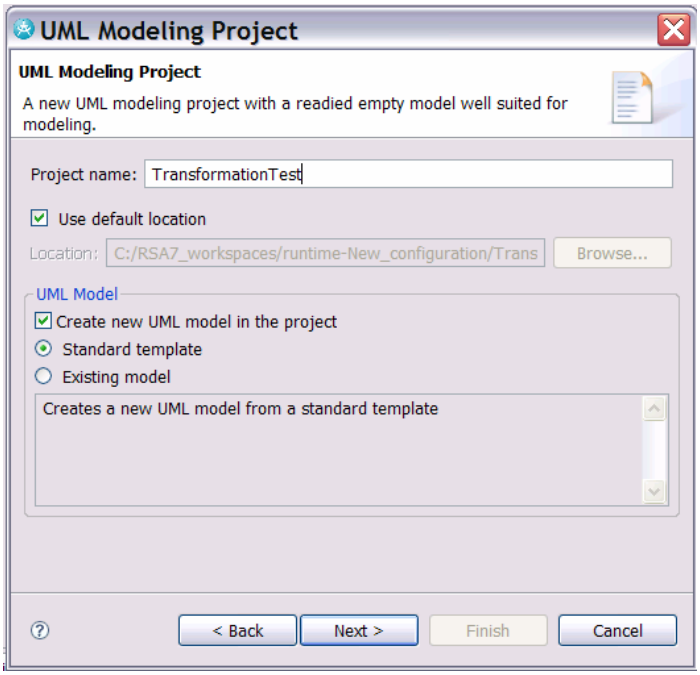


Figure 11-17: *Creating a modeling project*

- Under **Templates**, select **Blank Model**, change the file name to `SourceModel`, and click **Finish**.
- If prompted to switch to the Modeling Perspective, click **Yes**.

4. Create a package named `BusinessClasses`.
5. In the `BusinessClasses` package, create a class named `Employee` and add three private operations; **readEmail**, **answerPhone**, and **performWork**. Add one public operation **reportToManager (name:String)**.

   Note: to see signature, right-click class and select **Filter > Show Signature**.

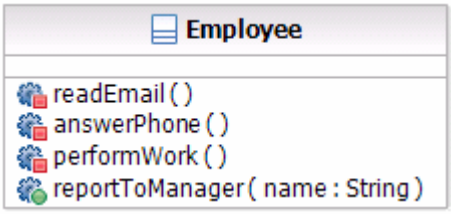


Figure 11-18: *Employee class*

6. To the `TransformationTest` project, add a new UML Model to be the target.
7. On the **Project Explorer**, select the `TransformationTest` project, right-click and select **New > UML Model**.
8. Select the Standard Template, then click **Next**.
9. Under **Templates**, select **Blank Model**, change the file name to `TargetModel` and click **Finish**.

Task 10: Run the Transformation

In this task, you will configure and run the transformation.

1. From the main menu bar, select **Modeling > Transform > New Configuration**.
2. Name the new configuration `FirstConfiguration` and select the **Generalize_Classes Transform**, then click **Next**.

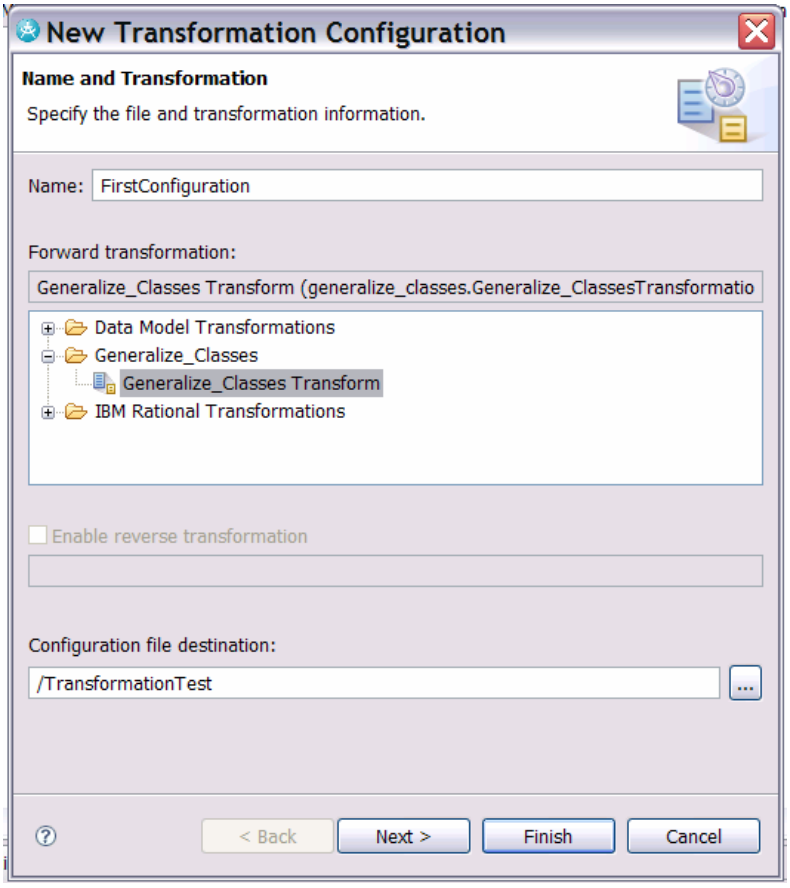


Figure 11-19: *Selecting the transformation*

3. On the New Transformation Configuration screen, select **SourceModel** as the Selected source and **TargetModel** as the Selected target. Click **Next**.

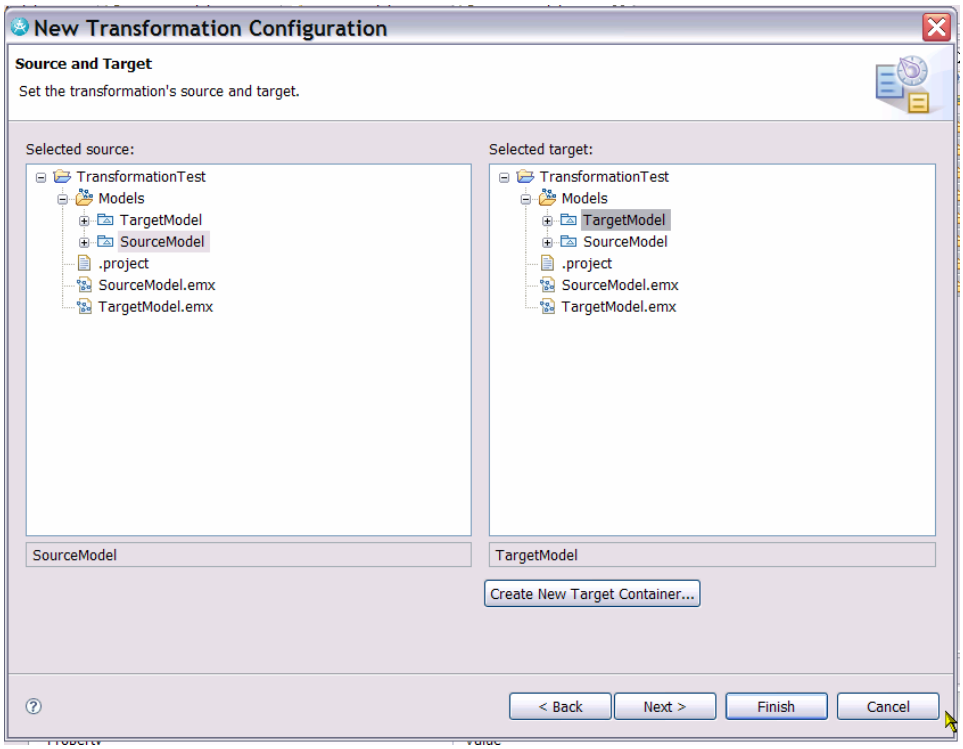


Figure 11-20: *Specify source and target*

4. Click **Next** through the next three screens, but leave the defaults and then click **Finish.**
5. This creates a `.tc` file in the project that contains the transformation configuration. Right-click this file and select **Transform > Generalize_Classes Transform.**
6. While the transformation is executing, you will be prompted that the target files will be updated with the automatic merge options. Click **OK.**
7. The model merge dialog will display so that you can validate the changes to the target model. Select the two changes as indicated in the following screenshot. Click **OK.**

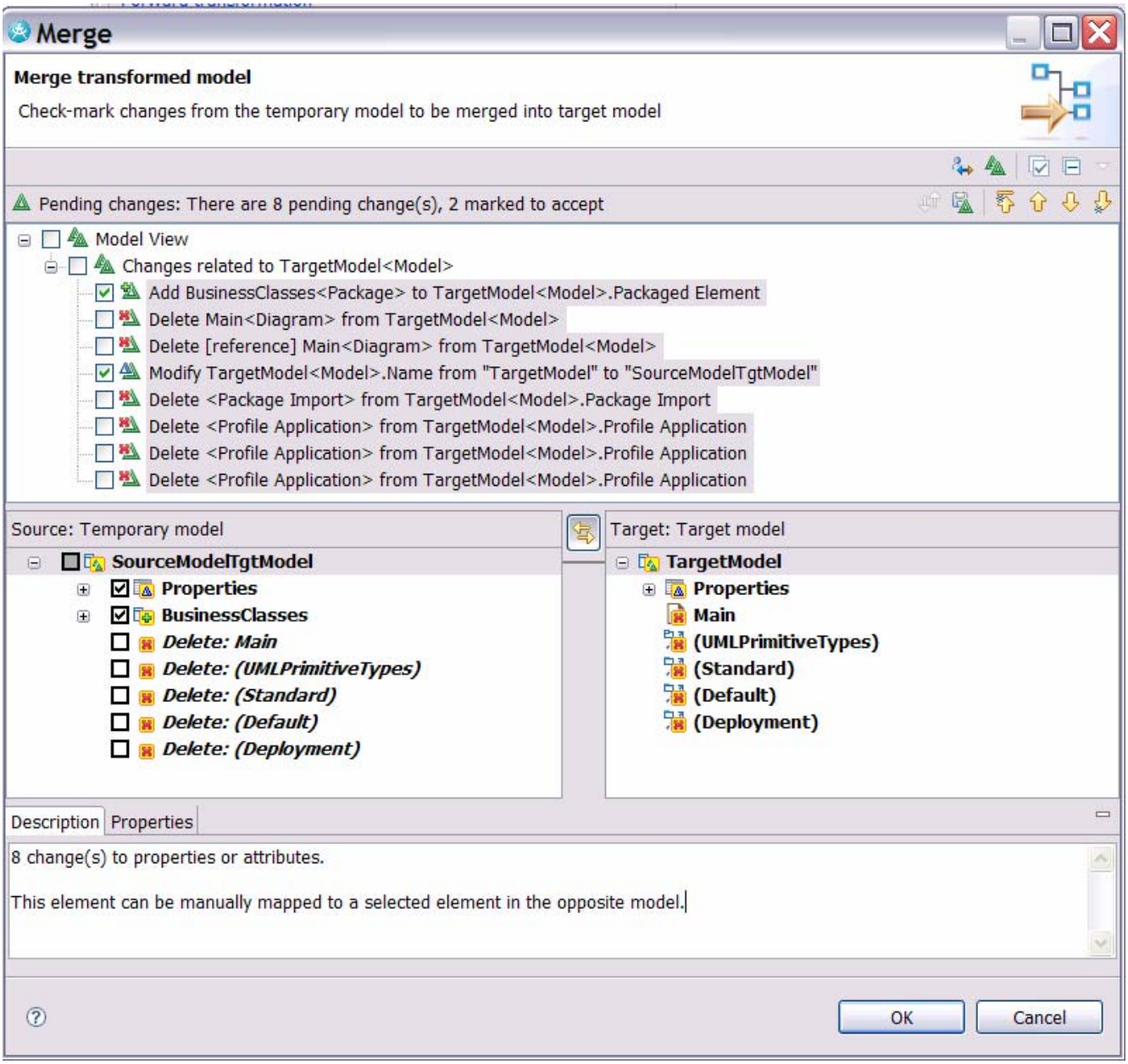


Figure 11-21: *Merging the models*

8. When prompted to accept changes from the file system, click **Yes.**
9. Explore the results in **TargetModel.**

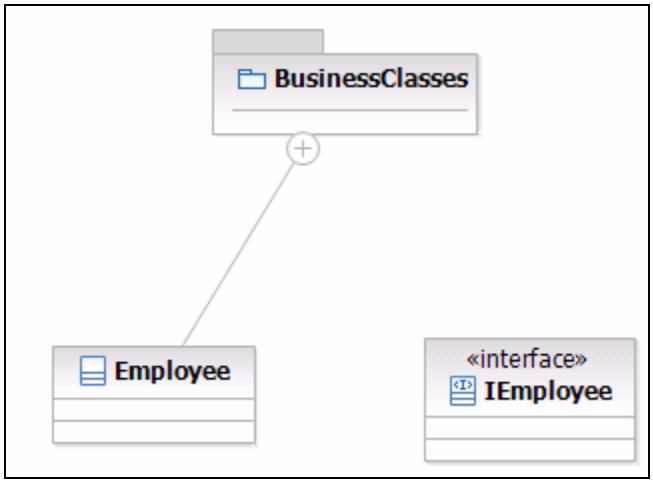


Figure 11-22: *Resulting elements in the TargetModel*

Task 11: Add New Mappings and a Relationship

You need to go back and complete the transformation by adding mappings to copy operations and adding the code to create the realization relationship between the class and the interface. You have "pre-cooked" those files to save time, and to demonstrate how you can re-use mappings across projects.

1. Close the run-time workbench and switch back to the host workbench.
2. Implement a mapping from another file.
   - Copy the file `OperationMapping.mapping` from `C:\Workshop\Labs\Inputs` into the `model` folder of the GeneralizeClasses project.
   - If not already open, right-click the Generalize_Classes.mapping file in the model folder and select **Open with > Mapping Editor**.
   - Double-click **Class2Class** in the Outline view.
   - Create a **Submap** from **ownedOperation** in the source Class to **ownedOperation** in the target Class.
   - On the **Details** tab of the **Properties** view, click **Browse** and select the file OperationMapping.mapping.
   - Repeat these steps, adding this submap to the **Class2Interface** transformation.
3. Add a condition to this Submap on the Class2Interface transformation so that that only public visibility operations are copied.
   - In the Properties view, select the **Input Filter** tab. Select the checkbox for **Filter Input Elements**.
   - In the Code: area, be sure that **In-line** is selected and enter the following code:

```
return ownedOperation_src.getVisibility()==VisibilityKind.PUBLIC_LITERAL;
```

TIP: As you enter code, try out the code completion with *Ctrl-Space*.

4. Add the code to create the Realization relationship from the implementation class to the interface in the

target model.

- Select the Custom transform on the mapping of the name of the source class to the name of the target interface. Copy the code from `Class2InterfaceCustomNameTransform.txt` into the code of this custom transform.

5. Save all.

6. Generate and clean up code.

- Generate the transformation code for the operation mapping. In the Mapping Editor with the OperationMapping.mapping file open, right-click the surface to the right of the **OperationMapping** button and select **Generate transformation source code** from the context menu.
- Re-generate the transformation code for the Generalize_Classes mapping. In the **Mapping Editor** with the Generalize_Classes.mapping file open, right-click the surface to the right of the **Generalize_Classes** button and select **Generate transformation source code** from the pop-up menu.
- In the Generalize Classes project, create a new package under the src directory called utilities. Copy FindElementUtility.java from C:\Workshop\Labs\Inputs into the utilities folder that you just created.
- There will be errors in `Class2InterfaceTransform.java` due to the fact that Class and Package are resolved to java.util rather than the uml versions needed. To correct this, add the following import statements:

```
import org.eclipse.uml2.uml.Class;

import org.eclipse.uml2.uml.Package;
```

- Organize imports in `Class2InterfaceTransform.java` using `Ctrl-Shift-o`.
- Save all.

7. Test the updated transformation

- Start the runtime workbench as before.
- Select FirstConfiguration.tc, right-click this file and select **Transform > Generalize_Classes Transform.** The results in the target model will look like this when the package and two classes are selected and dragged onto a diagram:

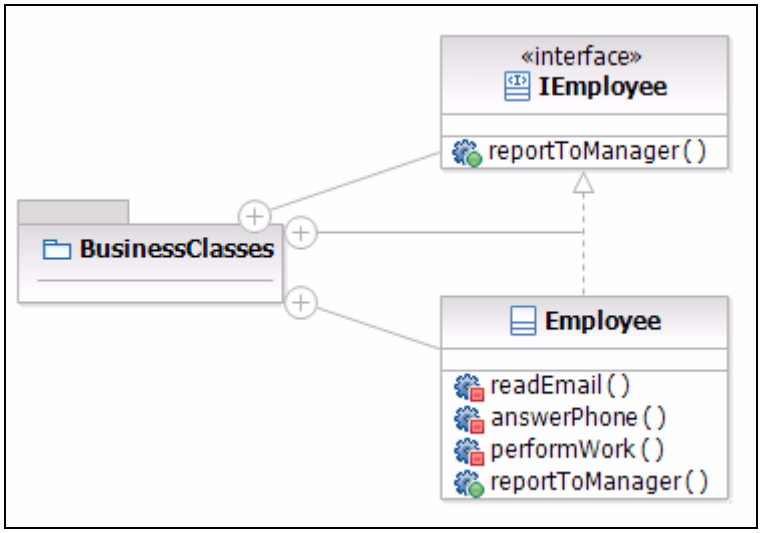


Figure 11-23: *Results in the target project*

- When you have finished testing and debugging the transformation, close the run-time workbench.

Tips and Troubleshooting

TIP: If you close the mapping editor and need to re-open it, right-click the *projectName*.mapping file in the models folder and select **Open with > Mapping Editor**. If the Mapping Editor does not display as an option, then make sure that you have enabled the XML Developer capability (Task 1, Step 5).

TIP: To use profiles, select the input and ouput profiles in addition to selecting the UML ECore model on the Create Project wizard. This will allow you to select the UML element as well as any stereotypes you want to map to and from.

TIP: When you create a new mapping transformation project using the New Project wizard it will add dependencies that are implied by the input and output models that you identify. So, for instance if you add the UML.ecore metamodel the wizard will add a dependency to that metamodels plugin.

If you later add another input or output metamodel you will need to add any new dependencies to your plugin.xml manually (dependencies are really in the manifest.mf file).

Or if you create (with the New Map wizard, not the New Map Project wizard) a map or copy a map to a non-mapping project, you will need to add necessary dependencies, nature, and builder to your plugin.xml and .project files.

TIP: If you want to map an abstract element (for example, the Type of a parameter) you will need to create a concrete mapping for each subtype you want handled. So for the Type of a parameter, create a Class-to-Class map, and a primitiveType-to-primitiveType map.

Lab 12 – Create the Master Detail Pattern

Objectives

After completing this lab, you will be able to:

- Create a pattern to be used in conjunction with the `UXModeling` profile.

Given

The following lab artifacts, a set of project interchange files, can be found in the `Inputs` folder for this lab:

- `details-expand.input.txt`
- `listscreen-expand.input.txt`
- `searchscreen-expand.input.txt`
- `detaildependency-expand.input.txt`
- `listdependency-expand.input.txt`
- `The project interchange file named CreateUXModelingProfile.zip.`

Scenario

In this portion of the workshop, you will create a Rational Software Architect Pattern that will support the creation of a Master-Detail relationship between a set of screens. The intent of the pattern will be to automate the creation of relationships between the classes involved in the pattern, and create classes that are needed to fill the roles within the Master-Detail collaboration.

This pattern will leverage the UXModeling profile that you created earlier. The pattern will be aware of the profile and its stereotypes, and will also apply some of the stereotypes to the pattern parameters.

Task 1: Create the Pattern Project

In this task you will create an implementation of a Master-Detail pattern.

1. Create a pattern project
   a. On the **File** menu, click **New > Project**.
   b. Make sure that **Show All Wizards** is selected.
   c. Replace `type filter text` with `plug`.
   d. Select **Plug-in Project**. Click **Next.**
   e. Name the project `Struts`, and then click **Next**.

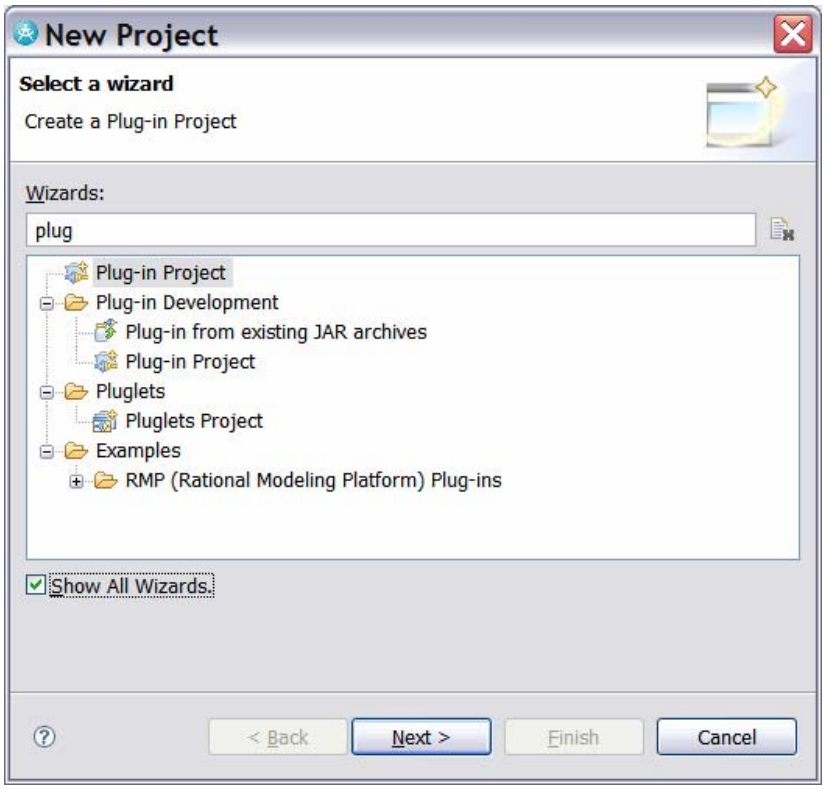


Figure 12-1: *Creating the plug-in project*

f. Click **Next.**

g. On the Templates page, select **Create a Plug-in using one of the templates**. Then choose **Plug-in with Patterns**, and click **Next.**

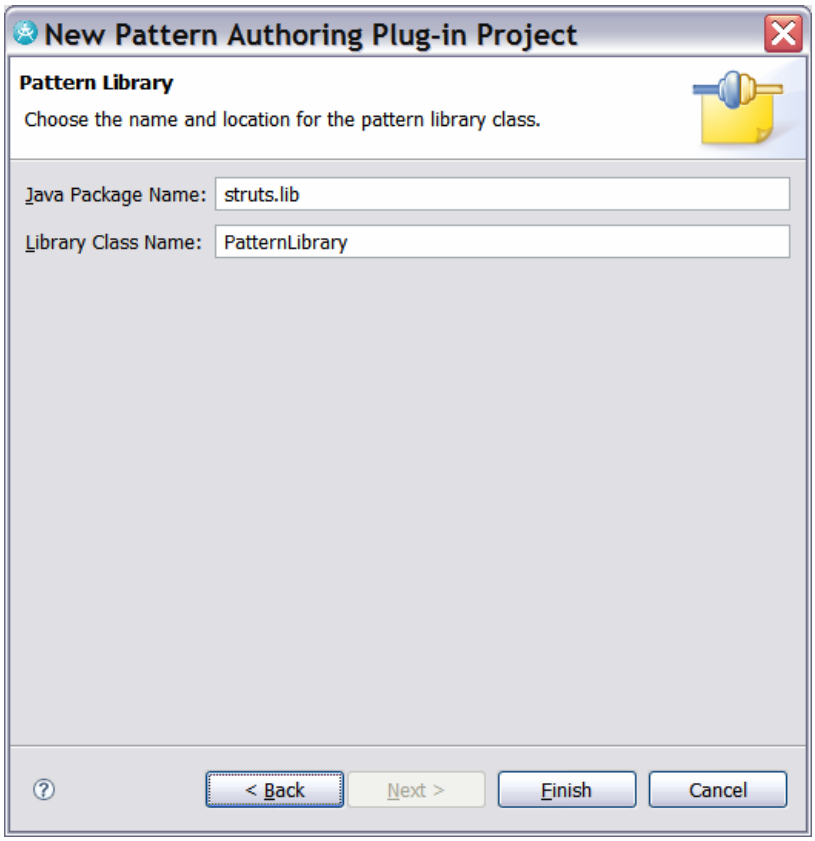


Figure 12-2: *Details for the Pattern Library*

h. Click **Finish**.

i. If asked, click **Yes** to change to the Plug-in Development perspective.

j. If asked, click **OK** to enable **Reusable Asset Management** capability.

2. Set up the pattern.

a. In the Plug-in perspective, bring the Pattern Authoring view to the front. If it is not open, then click **Windows > Show View > Other**. Replace `type filter text` with `Pattern`. Select **Pattern Authoring** and then click **OK**.

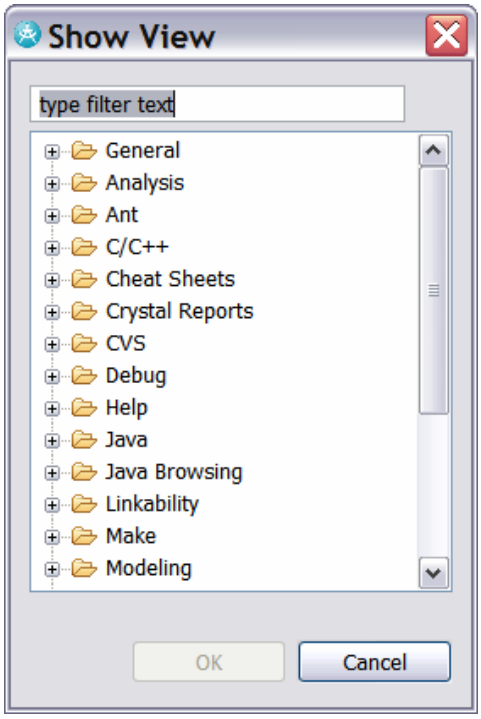


Figure 12-3: *Adding in the Pattern View*

b. In the Pattern Authoring view, right-click **Struts** and then click **New Pattern**.

k. In the New Pattern dialog, specify `Master-Detail` as the **Pattern Name**. The **Class Name** should be `MasterDetail`.

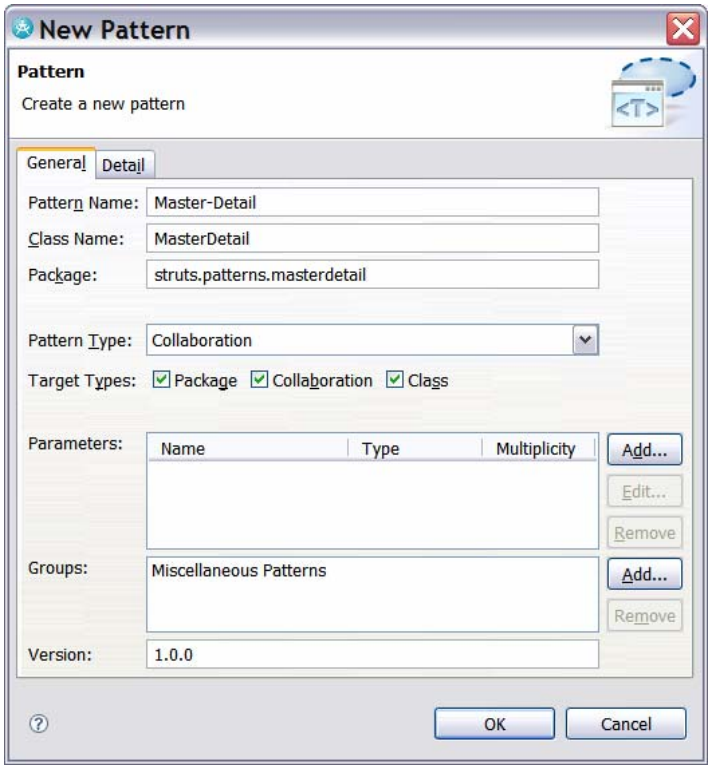


Figure 12-4: *Details for the new Pattern*

l. Add parameters to the pattern as follows:

- **Name:** Search Screen **Class Name:** SearchScreen **Type:** Class
- **Name:** List Screen **Class Name:** ListScreen **Type:** Class
- **Name:** Details Screen **Class Name:** DetailsScreen **Type:** Class

m. **Edit** the List Screen parameter. On the **Parameter Dependency** tab, set `Search Screen` as a **Client Parameter** and set `Details Screen` as a **Supplier Parameter**. Then click **OK**.

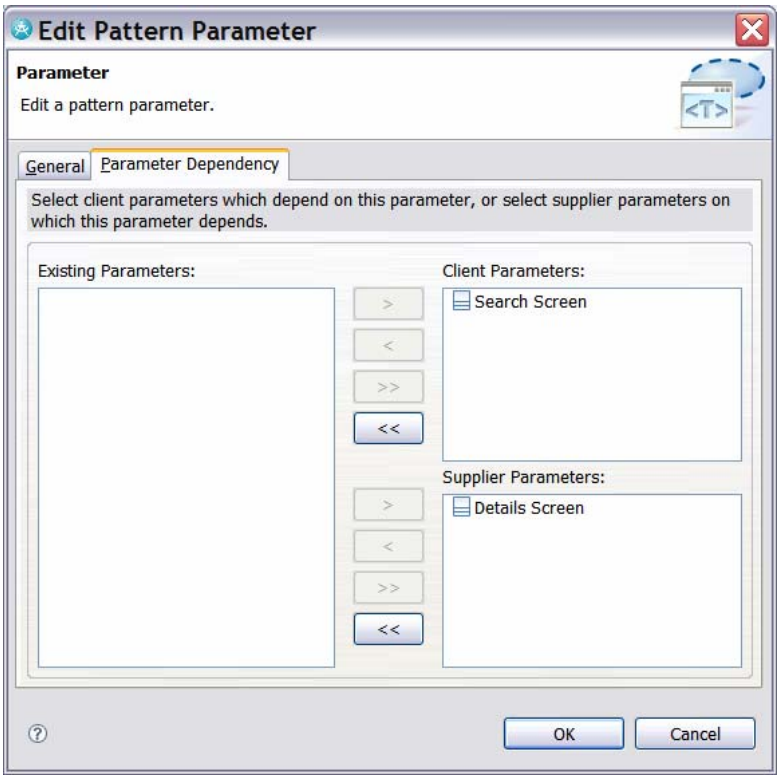


Figure 12-5: *Parameter Dependencies*

n. Remove the **Miscellaneous group** and add your own group called `My Struts Patterns`.

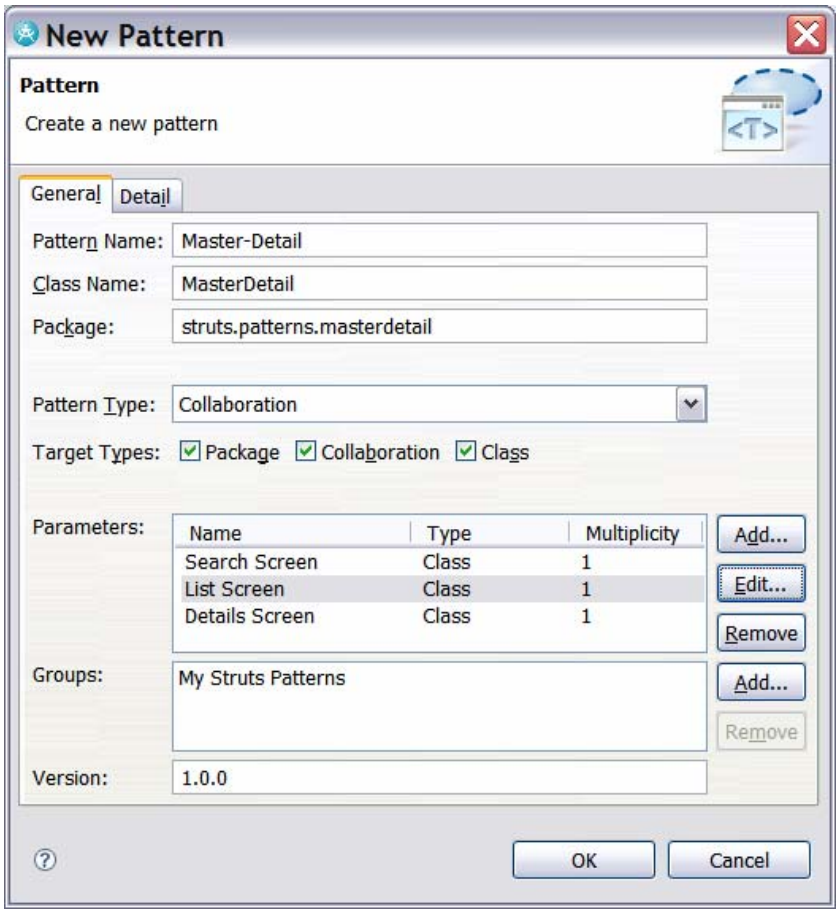


Figure 12-6: *Completed pattern specification*

o. Click **OK** to complete creating the pattern structure.

Task 2: Customize Expand Methods

In this task, you will add code to the Expand methods of the pattern to customize the behavior of the pattern.

1. Use the following code to replace the code found in the `public boolean expand(PatternParameterValue value)` of the `DetailsScreen` class:

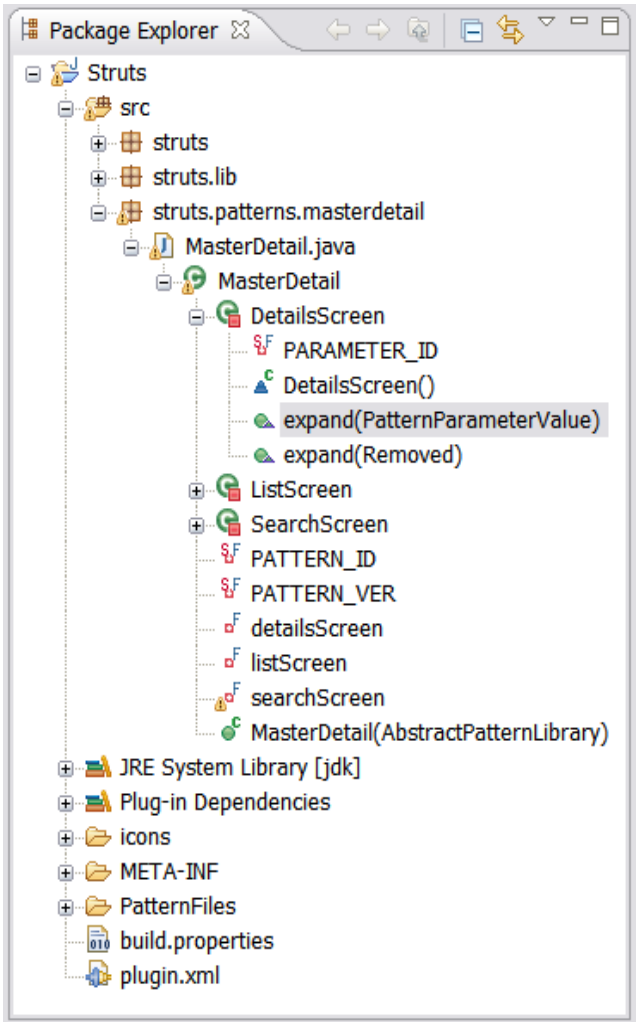


Figure 12-7: *Expand method for the DetailsScreen*

TIP: You can copy the following code from `C:\Workshop\Labs\Inputs\details-expand.input.txt`.

```
{
      Profile uxProfile = null;

      //add the <<screen>> stereotype to the class
      //first ensure that the profile has been applied to the model
      Class detailsClass = (Class)value.getValue();
      for (Iterator iter =
detailsClass.getModel().getProfileApplications().iterator();iter.hasNext(
);)
      {
            ProfileApplication profileAppl = (ProfileApplication)
iter.next();
            Profile profile = (Profile) profileAppl.getAppliedProfile();
```

```
            if(profile.getName().compareTo("UXModeling") == 0)
            {
                  uxProfile = profile;
                  break;
            }
      }
      if(uxProfile != null)
      {
            //since the profile has been applied to the model, we can add
the stereotype
            //to the class
            Stereotype screen =
detailsClass.getAppliedStereotype("UXModeling::screen");
            //if the stereotype has not been applied...
            if(screen == null)
            {
                  screen =
detailsClass.getApplicableStereotype("UXModeling::screen");
                  detailsClass.applyStereotype(screen);
            }
      }
      //add a display stereotype to each attribute for the class
      for(Iterator iter =
detailsClass.getOwnedAttributes().iterator();iter.hasNext();)
      {
            //add the stereotype to each attribute
            Property prop = (Property)iter.next();
            Stereotype display =
prop.getAppliedStereotype("UXModeling::display");
            //if the stereotype has not been applied...
            if (display == null)
            {
                  display =
prop.getApplicableStereotype("UXModeling::display");
                  prop.applyStereotype(display);
            }

      }
      return true;
}
```

2. Right-click in the editor and select **Source > Organize Imports**. When prompted to choose imports:
   - For **Iterator**, select `java.util.Iterator`.
   - For **Profile**, select `org.eclipse.uml2.uml.Profile`.
3. Add in an import statement to the class as follows:

   ```
   import org.eclipse.uml2.uml.Class;
   ```

4. Select **File > Save All**.
5. Use the following code to replace the code found in the `public boolean expand(PatternParameterValue value)` of the `ListScreen` class:

   TIP: The following code can be copied from `C:\Workshop\Labs\Inputs\listscreen-expand.input.txt`

```
{
```

```
      Profile uxProfile = null;
      //add the <<screen>> stereotype to the class
      //first ensure that the profile has been applied to the model
      Class listClass = (Class)value.getValue();
      for (Iterator iter =
listClass.getModel().getProfileApplications().iterator();iter.hasNext();)
      {

            ProfileApplication profileAppl = (ProfileApplication)
iter.next();
            Profile profile = (Profile) profileAppl.getAppliedProfile();

            if(profile.getName().compareTo("UXModeling") == 0)
            {
                  uxProfile = profile;
                  break;
            }
      }
      if(uxProfile != null)
      {
            //since the profile has been applied to the model, we can add
the stereotype
            //to the class
            Stereotype screen =
listClass.getAppliedStereotype("UXModeling::screen");
            //if the stereotype has not been applied...
            if(screen == null)
            {
                  screen =
listClass.getApplicableStereotype("UXModeling::screen");
                  listClass.applyStereotype(screen);
            }
      }
      //create an associated <<input>> class that will allow for entry of
search parameters
      //use {class}Form as the name of the input class

      //TODO : time permitting - add logic to ensure that the class does
not already exist
      String theResultsName = listClass.getName() + "Results";
      //now create a new class in the same package with theFormName
      Package theTargetPackage = listClass.getPackage();
      //add a relationship between {class} class and {class}Form class
      Class newClass =
(Class)theTargetPackage.createPackagedElement(theResultsName,
UMLPackage.eINSTANCE.getClass_());


      //add a stereotype to the new class
      Stereotype input =
newClass.getApplicableStereotype("UXModeling::list");
      newClass.applyStereotype(input);

      //add a composite relationship from the the input class to the
screen class
      AbstractPatternInstance instance = (AbstractPatternInstance)
value.getOwningInstance();

      instance.ensureDirectedAssociation(listClass, newClass,"creates
record list",AggregationKind.COMPOSITE_LITERAL,1,1);
```

```
        return true;
}
```

6. Right-click in the editor and select **Source > Organize Imports**. When prompted to choose imports:
   - For **AbstractPatternInstance**, select `com.ibm.xtools.patterns.framework.uml2.AbstractPatternInstance`
   - If asked, for **Class**, select `org.eclipse.uml2.uml.class`
   - If asked, for **Iterator**, select `java.util.Iterator`.
7. Add the following import statement to the class:
   - `import org.eclipse.uml2.uml.Package;`
8. Select **File > Save All**.
9. Use the following code to replace the code found in the public boolean expand(PatternParameterValue value) of the SearchScreen class:

TIP: The following code can be copied from `C:\Workshop\Labs\Inputs\searchscreen-expand.input.txt`.

```
{
        //this code checking for the profile should be genericized and added
        //to the utility class
        Profile uxProfile = null;

        //add the <<screen>> stereotype to the class
        //first ensure that the profile has been applied to the model
        Class searchClass = (Class)value.getValue();
        for (Iterator iter =
searchClass.getModel().getProfileApplications().iterator();iter.hasNext();)
        {

                ProfileApplication profileAppl = (ProfileApplication) iter.next();
                Profile profile = (Profile) profileAppl.getAppliedProfile();

                if(profile.getName().compareTo("UXModeling") == 0)
                {
                        uxProfile = profile;
                        break;
                }
        }
        if(uxProfile != null)
        {
                //since the profile has been applied to the model, we can add the
stereotype
                //to the class
                Stereotype screen =
searchClass.getAppliedStereotype("UXModeling::screen");
                //if the stereotype has not been applied
                if(screen == null)
                {
                        screen =
searchClass.getApplicableStereotype("UXModeling::screen");
                        searchClass.applyStereotype(screen);
                }

                //create an associated <<input>> class that will allow for entry of
search
                // parameters use {class}Form as the name of the input class
```

```
            //time permitting - add logic to ensure that the class does not
already
            //exist
            String theFormName = searchClass.getName() + "Form";
            //now create a new class in the same package with theFormName
            Package theTargetPackage = searchClass.getPackage();
            //add a relationship between {class} class and {class}Form class
            Class newClass =
(Class)theTargetPackage.createPackagedElement(theFormName,UMLPackage.eINSTANCE.ge
tClass_());


            //add a stereotype to the new class
            Stereotype input =
newClass.getApplicableStereotype("UXModeling::input");
            newClass.applyStereotype(input);

            //add a composite relationship from the the input class to the
screen class
            AbstractPatternInstance instance = (AbstractPatternInstance)
value.getOwningInstance();

            instance.ensureDirectedAssociation(searchClass,
newClass,"contained",AggregationKind.COMPOSITE_LITERAL,1,1);
      }

      return true;
}
```

10. Select **File > Save All**.

11. Review.

Task 3: Customize Update Methods

In this task, you will add code to the Update methods of the pattern to customize the behavior of the pattern in cases where there is a dependency between the pattern parameters.

1. Use the following code to replace the code found in the `public boolean update(PatternParameterValue value, PatternParameterValue dependencyValue)` of the `ListScreen.ListScreen_DetailsScreenDependency` class:

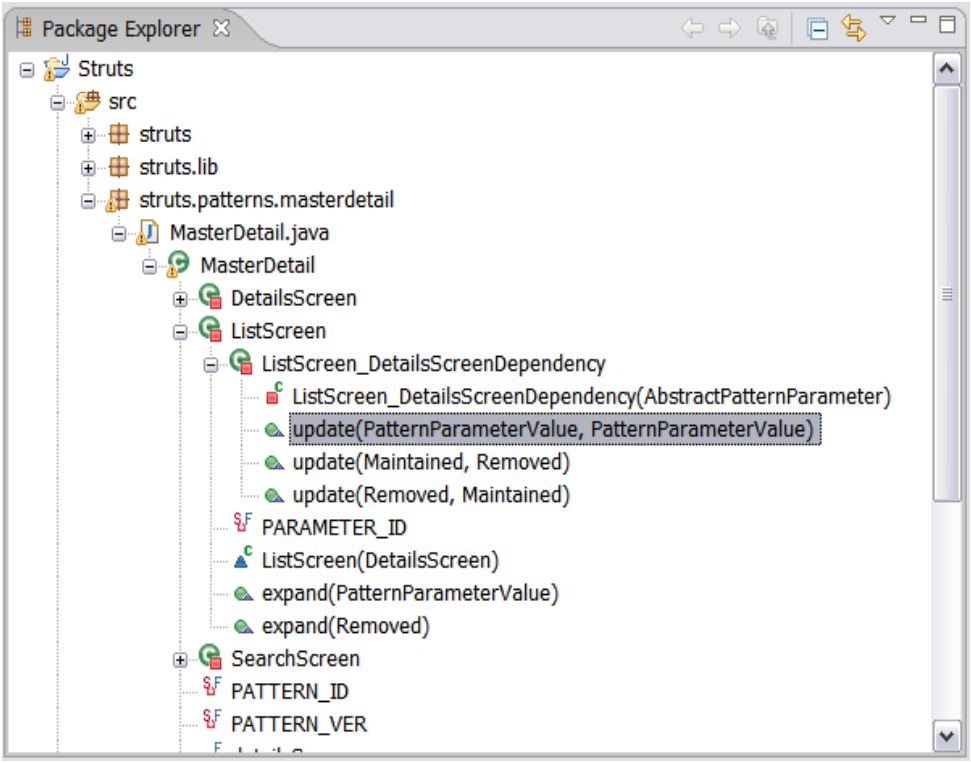


Figure 12-8: *Update method for the ListScreen.ListScreen_DetailsScreenDependency*

TIP: You can copy the following code from `C:\Workshop\Labs\Inputs\detaildependency-expand.input.txt`.

```
{
      //at this point we know the list screen and the details screen.
      //create a directed relationship between them.
      Class listClass = (Class)value.getValue();
      Class displayClass = (Class) dependencyValue.getValue();
      AbstractPatternInstance instance =
(AbstractPatternInstance)value.getOwningInstance();
      instance.ensureDirectedAssociation((Class)value.getValue(),"creates record
list", (Class)dependencyValue.getValue(), "displays list");

      return true;
}
```

1. Select **File > Save All**.
2. Use the following code to replace the code found in the public boolean update(PatternParameterValue value, PatternParameterValue dependencyValue) of the SearchScreen.SearchScreen_ListScreenDependency class:

TIP: You can copy the following code from `C:\Workshop\Labs\Inputs\listdependency-expand.input.txt`.

```
{
      //check if the association exists, if not then create it.
      Class listClass = (Class)value.getValue();
```

```
        Class searchClass = (Class) dependencyValue.getValue();

        AbstractPatternInstance instance = (AbstractPatternInstance)
value.getOwningInstance();
        instance.ensureDirectedAssociation((Class)value.getValue(),"resultscontaine
dBy", (Class)dependencyValue.getValue(), "generatesSearchCriteria");

        return true;
}
```

3. Select **File > Save All**.
4. Fix any compiler errors.
5. Select **File > Save All**.
6. Review.

Task 4: Test the Pattern

In this task, you will test the pattern that we've created. Note that the pattern depends on the UXModeling profile that we created earlier.

1. Import the project interchange file that contains the UXProfile:
   - Select **File > Import**.
   - Replace `type filter text` with `Project`. Select **Project Interchange** and then click **Next**.
   - Click **Browse** and navigate to `C:\Workshop\Labs\Inputs` and select `CreateUXModelingProfile.zip`.
   - Click **Select All**.
   - Click **Finish**.
2. Open the plugin.xml file from within the Struts project.
3. On the **Overview** tab, click the Launch an Eclipse Application link.

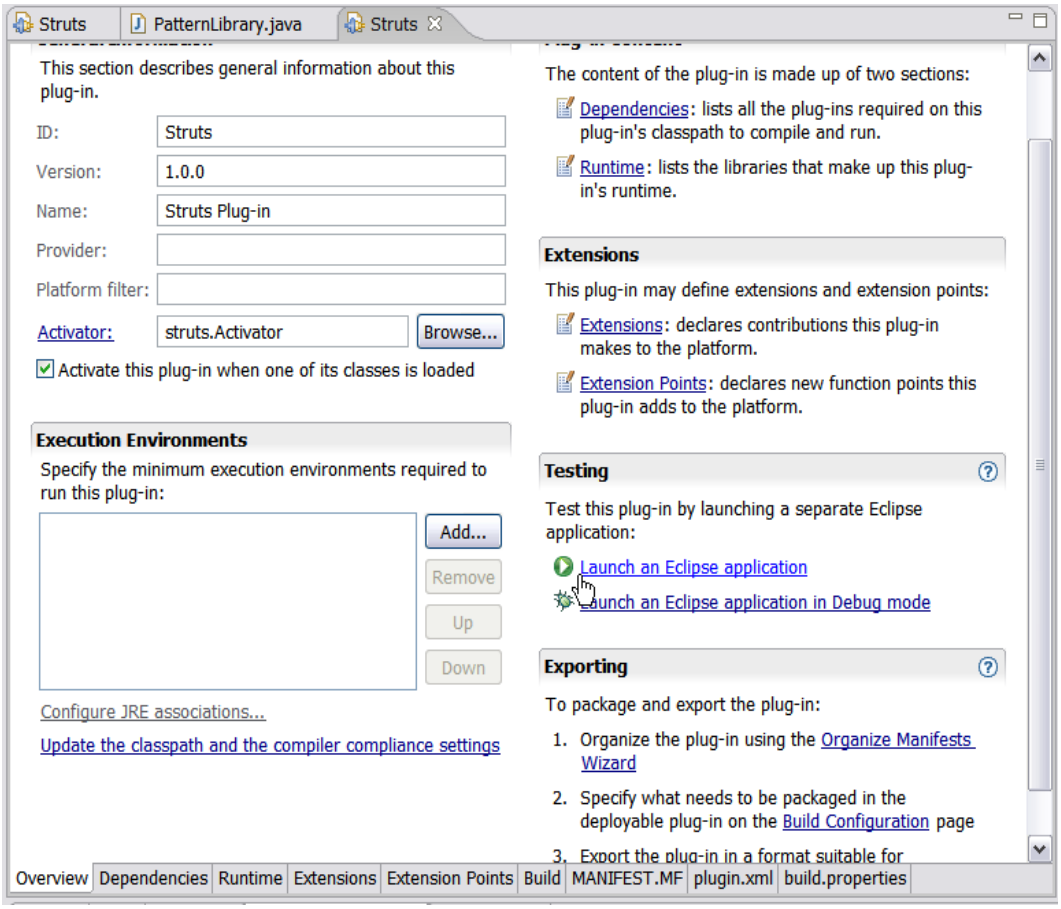


Figure 12-9: *Launching the runtime workbench*

The remaining steps are performed in the run-time workbench where we will test the pattern by applying it.

1. Create a new UML Model Project. Select **File > New Project**. Replace `type filter text` with `UML`. Select **UML Project**. Click **Next**.
2. Specify `PatternTest` as the **Project name**. Click **Next**.
3. Select **Blank Model** as the **Template**.
4. Specify `PatternTestModel` as the **File name**. Click **Finish**.
5. Apply the UXModeling profile to the model.

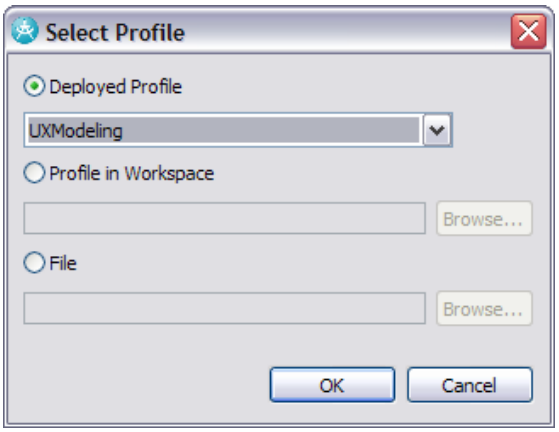


Figure 12-10: *Assigned the profile to the model*

6. Click **OK** when informed that the profile being applied has not yet been released.
7. Add the following classes to the model
   - `Music`
   - `MusicDetails`
   - `MusicList`
   - Add the following operation to the `Music` class
     - `logoff()`
   - Add the following attributes to the `MusicDetails` class:
     - `artist : String`
     - `recordingDate : String`
     - `genre : String`
     - `rating : String`
   - Add the following attributes to the `MusicList` class:
     - `artist : String`
     - `rating : String`
8. Apply stereotypes to the classes, attributes, and operations as shown in the diagram below:

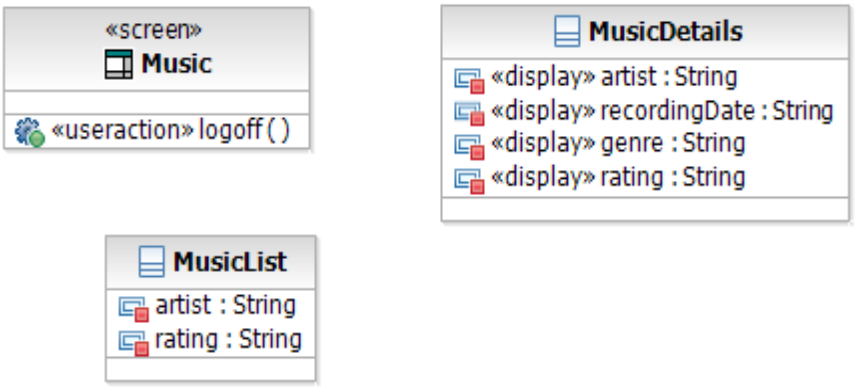


Figure 12-11: *Classes to use as parameters for the pattern*

4. Apply the `Master Detail` pattern using the classes shown above as parameters.
   - Add a new Class Diagram to the model. Name the diagram `Music-MasterDetail`.
   - Drag the `Master-Detail` Pattern from the pattern explorer to the `Music-MasterDetail` class diagram in the Diagram Editor.

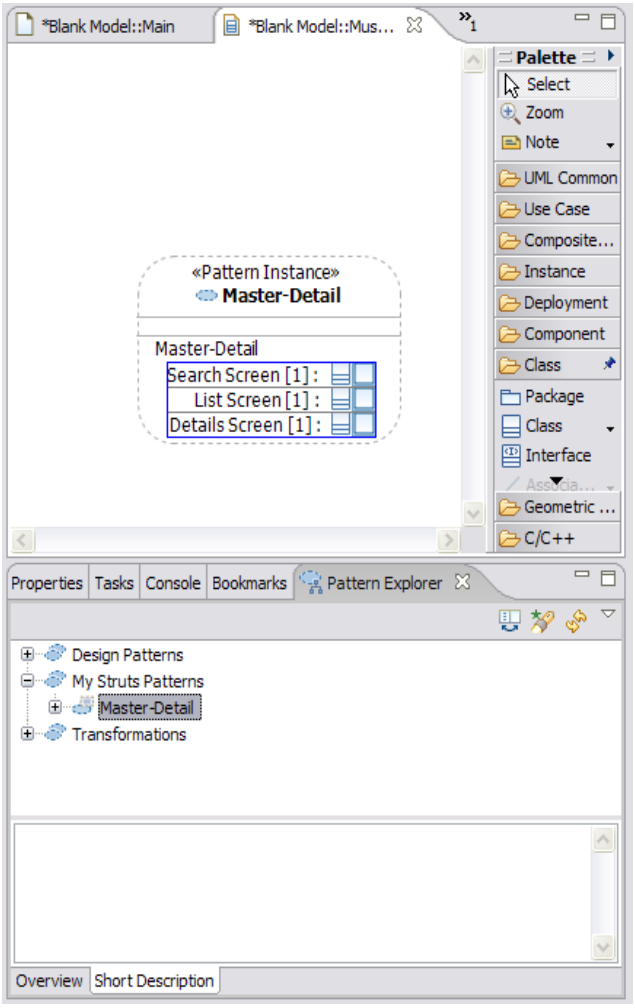


Figure 12-12: *Pattern instance within the class diagram*

- Drag the `Music` class from the Model Explorer to the `Search Screen` parameter of the `Master-Detail` pattern
- Drag the `MusicList` class from the Model Explorer to the `List Screen` parameter of the `Master-Detail` pattern
- Drag the `MusicDetails` class from the Model Explorer to the `Details Screen` parameter of the `Master-Detail` pattern.

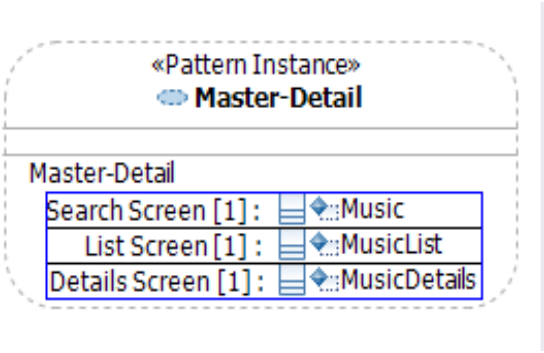
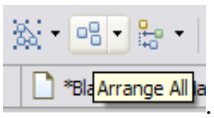


Figure 12-13: *Classes bound to the pattern*

- Drag the following classes from the Model Explorer to the `Music-MasterDetail` class diagram:
  i. `Music`
  ii. `MusicList`
  iii. `MusicDetails`
  iv. `MusicForm`
  v. `MusicListResults`

5. Within the class diagram, select all of the elements.
6. On the toolbar, click **Arrange All** 

.
7. The results should appear as follows:

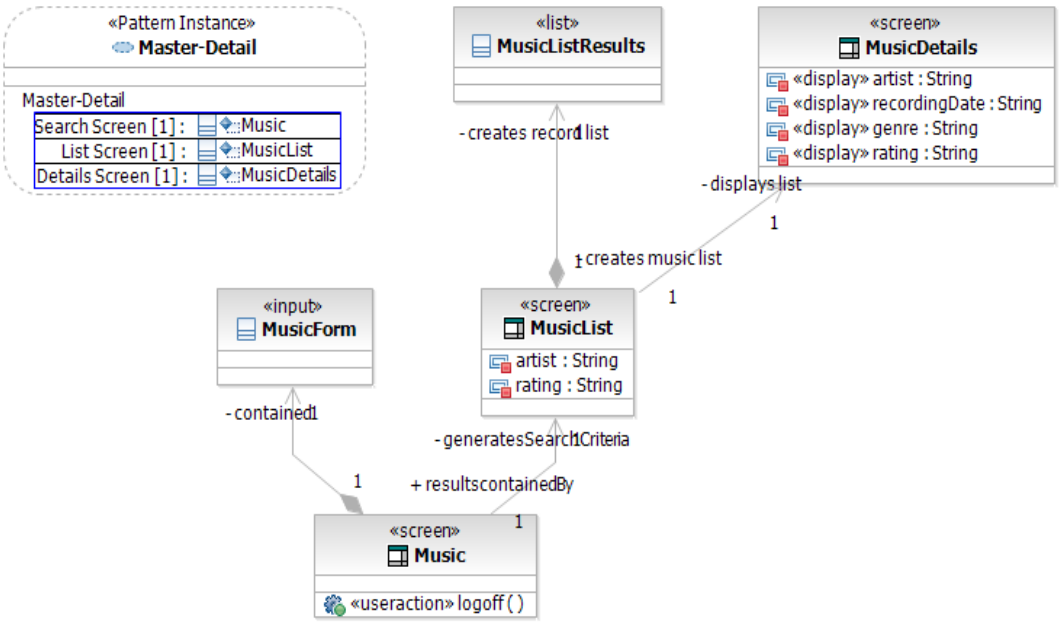


Figure 12-14: *Resulting classes as bound and generated by the pattern*

Task 5: Extra Challenges

If time permits during the class, or as a practice challenge for after the class, complete the following tasks.

1. Enhance the pattern so that any attributes in the List Screen parameter get moved from the List Screen to the List Results class that is created. In addition, each attribute in the new class should have a `«display»` stereotype applied.
2. Enhance the pattern so that any attributes in the Search Screen parameter get moved from the Search Screen to the {class}Form class that is created. In addition, each attribute in the new class should have a `«textfield»` stereotype applied.
3. Refactor and simplify the code.
4. Complete the pattern customization by coding the behavior that should occur when a parameter is removed from the pattern.

TIP: The necessary code should end up in the remaining Update and Expand methods.

Lab 13 – Create a Pluglet

Objectives

After completing this lab, you will be able to:

- Create and switch to a new workspace
- Customize a perspective
- Import and export shared projects using Project Interchange
- Create and test a simple pluglet

Given

The following lab artifacts can be found in the `Inputs` folder for this lab:

- A project interchange file that has a Pluglet project started (`PlugletProject.zip`)
- `DiagramLister Code Fragment.txt`

Scenario

In this lab, your team wants the capability to select a package in the Project Explorer and produce a listing of the package hierarchy, including any diagrams in each package. The team will use one of the extensibility features of IBM Rational Software Architect, known as a Pluglet. Another team member has partially implemented the pluglet, and it is being shared with you for completion.

You will start by creating a new workspace so that you will have a clean area in which to perform your development. Next you will need to configure a perspective, which allows you to control key aspects of the perspective (including available submenu options and actions sets associated with the toolbar and menu bar). Then, you will import the project to begin working on it.

Finally, you want to share your completed pluglet back with the other team members. Exporting your projects using Project Interchange will maintain the entire project structure and dependents.

Task 1: Create the Workspace

In this task, you will switch to a new workspace that you will create.

1. Start Rational Software Architect.
2. In the Workspace Launcher dialog, specify `C:\Workshop\StudentWork\CreateAPlugletWorkspace` as the **Workspace** directory, as shown below:

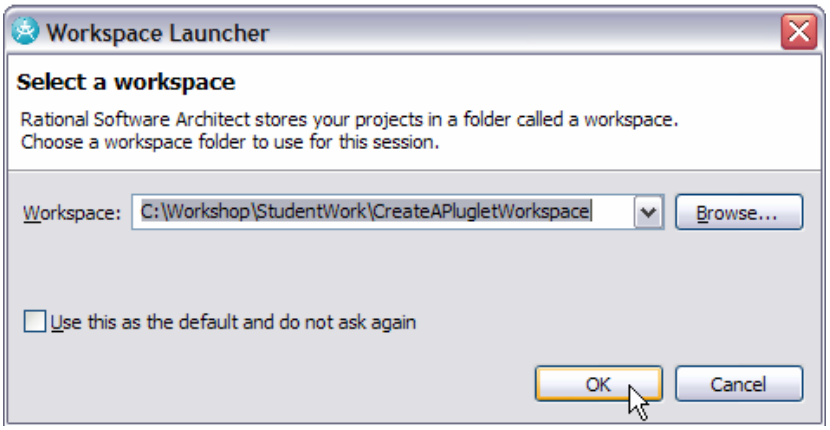


Figure 13-1: *Making a new Workspace Directory*

3. Click **OK**.
4. Close the Welcome screen.

Task 2: Configuring the Perspective

The steps in the task will guide you through activating pluglet projects and capabilities.

1. Ensure that you are in the **Modeling** perspective.
2. From the **Window** menu, select **Customize Perspective**.
3. In the Customize Perspective window, on the **Shortcuts** tab, be sure **New** is specified in the **Submenus** list.

4. In the **Shortcut Categories** list, select **Pluglets** to enable the pluglet projects and pluglet capabilities.

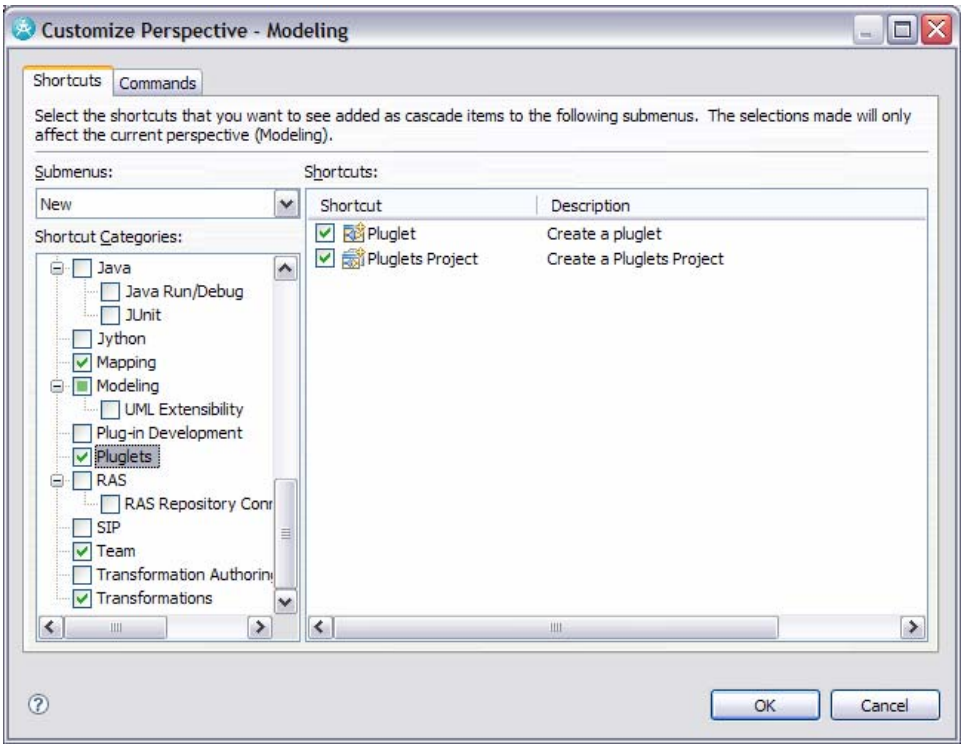


Figure 13-2: *Selecting Pluglets in the Shortcut Categories*

5. In the right pane, select the **Pluglet** and **Pluglets Project** check boxes.
6. Click the **Commands** tab. In the **Available command groups** list, make sure **Pluglets** and **Modeling** are selected.
7. Click **OK**.

Task 3: Import the Pluglet

You will import a project that contains a partially completed Pluglet.

1. From the **File** menu, select **Import**.
2. In the **Import** window, replace `type filter text` with `project`. Select **Project Interchange** and click **Next**.
3. In the Import Project Interchange Contents dialog, click **Browse** and navigate to `C:\Workshop\Labs\Inputs`.
4. Select `PlugletProject.zip` and click **Open**.

5. Click **Select All** to import all projects in the zip file.

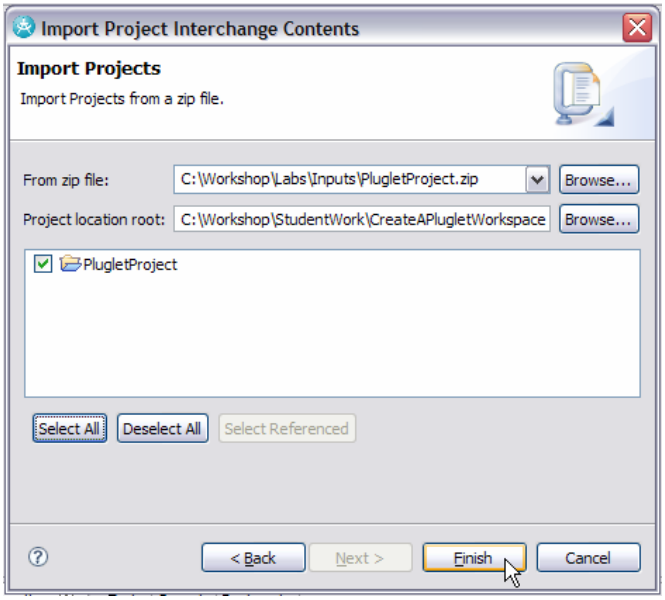


Figure 13-3: *Select projects to import*

6. Click **Finish**.

Task 4: Complete the Pluglet

The steps in the task will guide you through completing the pluglet.

1. In the Project Explorer, navigate to the **(default package)** and open the **DiagramLister** class.

1. Review the partially completed pluglet, in particular the plugletmain method.

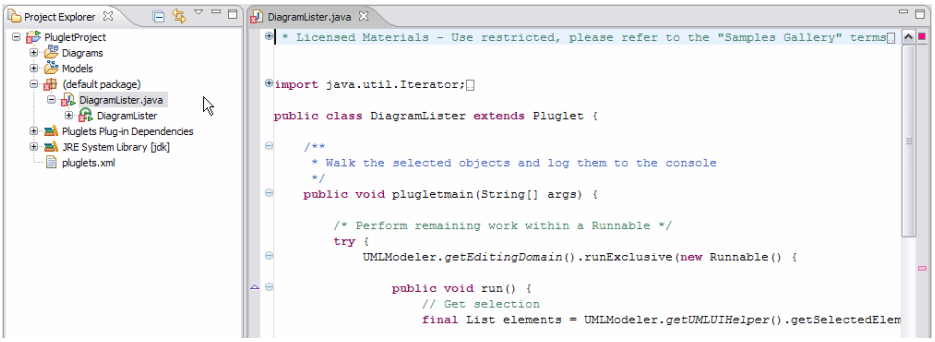


Figure 13-4: *plugletmain method in DiagramLister class*

3. Add the following method to the `DiagramLister` class (found in `C:\Workshop\Labs\Inputs\DiagramLister Code Fragment.txt`).

```
/**
* Recursively navigate thru a package and lists out all of the diagrams in that
* package and its children
* @param object The select object
*
*/
private void findDiagrams(List elements)
{
//get the UMLDiagramHelper - a helper for using UML 2.0 notation-based diagram
IUMLDiagramHelper diagramHelper = UMLModeler.getUMLDiagramHelper();
//            cycle thru selected element and its children
for (Iterator iter = elements.iterator(); iter.hasNext();) {
       Object object = iter.next();

       //ensure that it's a package - check for its children and go deeper
       if (object instanceof Package)
       {
              org.eclipse.uml2.uml.Package pack = (org.eclipse.uml2.uml.Package)
object;
              List diagrams = diagramHelper.getDiagrams(pack);
              out.println();
              out.println(pack.getName() + " package contains the following
diagrams:");

              for (Iterator iterd = diagrams.iterator(); iterd.hasNext();)
              {
                     Diagram diagram = (Diagram)iterd.next();
                     if(diagram != null)
                     {
                            out.println(diagram.getName() + " " + diagram.getType());
                     }
                     else
                     {
                            out.println("diagram was null");
                     }
              }
              //get the children for this package and send recursively search it for
more diagrams
              findDiagrams(pack.getNestedPackages());
       }
}
}
```

4. Press **Ctrl+S** to save the changes.

2. From the **Run** menu, select **Internal Tools > Internal Tools.**

3. Choose **Pluglet,** click **New** and enter `DiagramLister` as the configuration name.

4. Click **Browse Workspace**, select PlugletProject and DiagramLister.java, and then click **OK**.

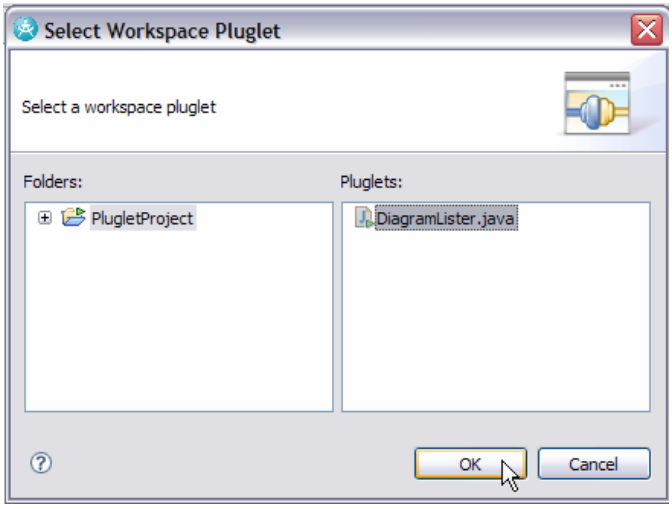


Figure 13-5: *Selecting the Pluglet*

5. Click **Apply**, and then click **Close**.

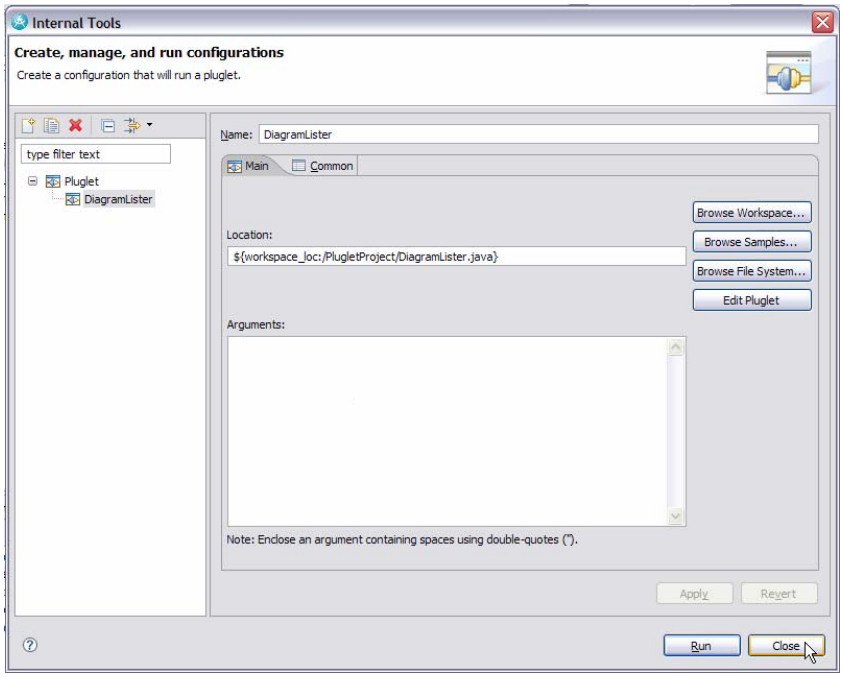


Figure 13-6: *Complete Run Configuration for the Pluglet*

Task 5: Run the Pluglet

This task will test the pluglet you just created.

1. In the Project Explorer, open the **ProfileTest** model. A Confirm Enablement dialog appears. Click **OK**.
2. Navigate to the **ProfileTest** model.
3. From the **Run** menu, select **Internal Tools > Internal Tools**.
4. In the **Configurations** pane, select `DiagramLister` and click **Run**.

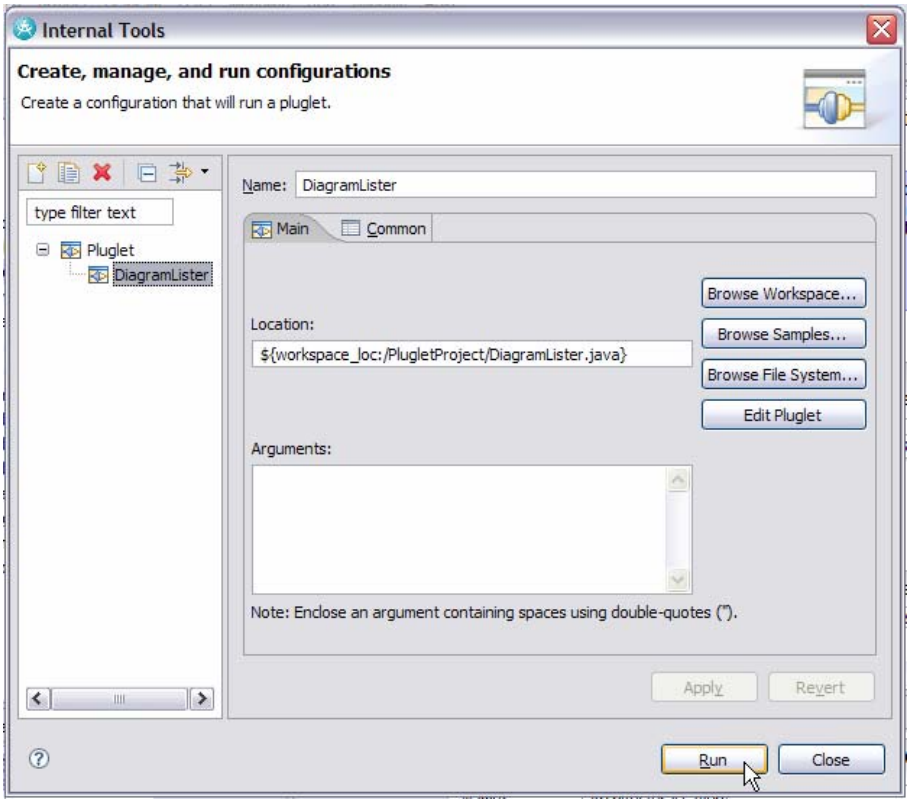


Figure 13-7: *Select the Pluglet Configuration to Run. The console will display a list of the packages (and sub-packages) along with the diagrams found within.*

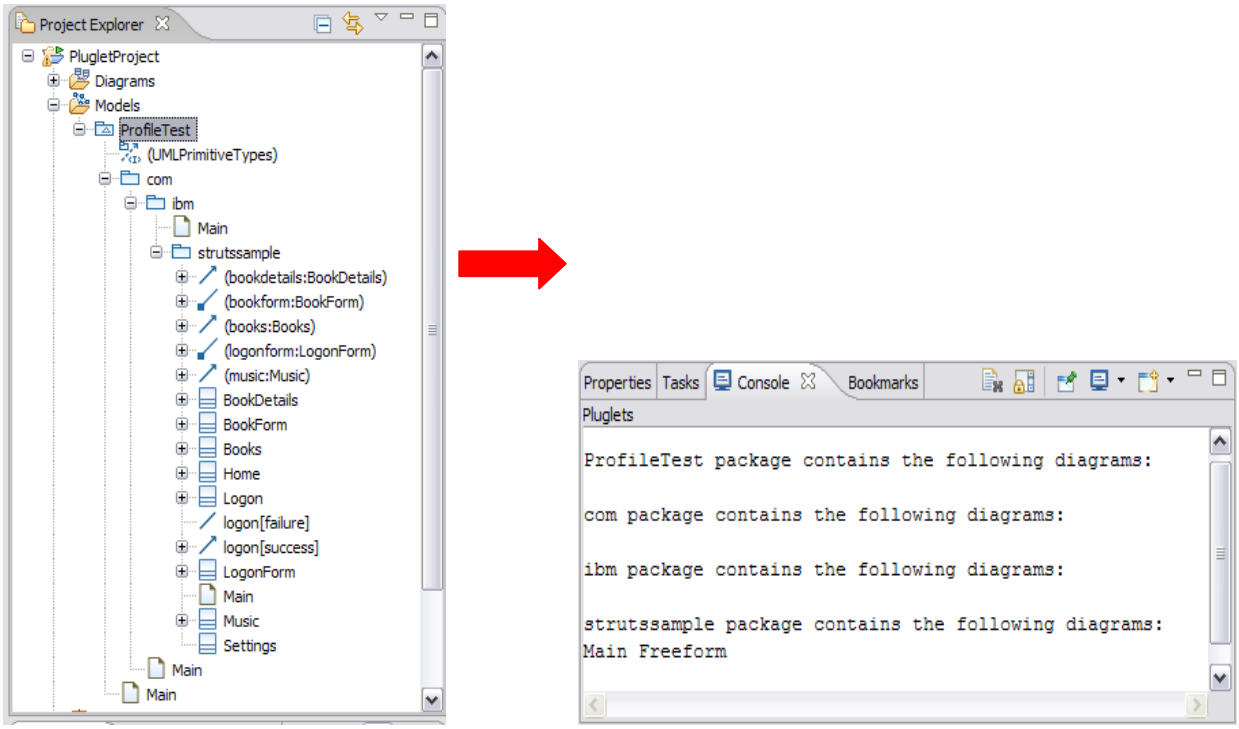


Figure 13-8: *Console after the Pluglet has been run*

TIP: Now that the system knows about the Pluglet, you can achieve subsequent runs of the pluglet by clicking **Run > Internal Tools > DiagramLister** while a package is selected in the Model Explorer.

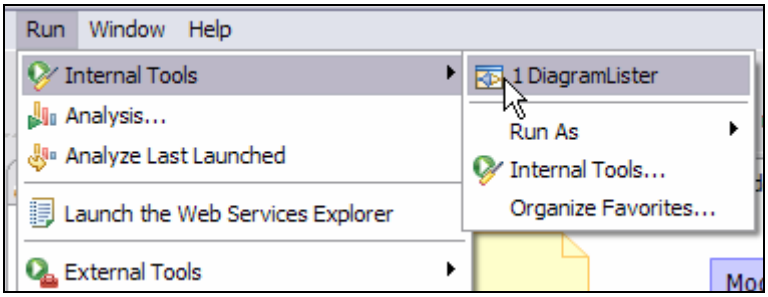


Figure 13-9: *Internal Tools menu after the Pluglet has been run once*

5. From the **File** menu, select **Save All** to save all the projects.

Task 6: Export the Pluglet

This task will allow sharing of the completed pluglet.

1. From the **File** menu, select **Export**.
2. In the Export window, select **Project Interchange** to export to a Zip format, and click **Next**.

3. In the Export Project Interchange Information window, click **Select All**.
4. Click **Browse** and navigate to the `C:\Workshop\Labs\StudentWork` directory.
5. Enter `CreateAPlugletLab` for the file name and click **Save**.
6. Click **Finish**.

Lab 14 – Create a UX Model Template

Objectives

After completing this lab, you will be able to:

- Create a UML Model Template that can be used in association with other Reusable Assets such as profiles, patterns, and transformations

Given

- `CreateUXModelingProfile.zip:` A project interchange file containing a plug-in project that hosts the UXModeling profile
- `UX Model Template Note.txt:` Instruction text included with the model template

Scenario

In this portion of the workshop, you will create a UML Model Template. This model template will provide a person with a starting structure for their modeling activities in support of capturing details related to UX modeling. In addition, you will add in some guidance on how to fill in the model using the elements within the template.

Task 1: Import the UXModeling Profile Plug-in Project

In this task, you will switch to, or create, a new workspace named `CreateAModelTemplateWorkspace`, and import the UXModeling Profile plugin project.

1. Start Rational Software Architect or select **Switch Workspace.**
2. In the Workspace Launcher dialog, replace the displayed text with `C:\Workshop\StudentWork\CreateAModelTemplateWorkspace` and click the **OK** button.
3. Close the Welcome screen.
4. Switch to the Modeling perspective.
5. Select **File > Import**.
6. Select **Project Interchange**. Click **Next**.
7. Click **Browse** next to the **From zip file** field.
8. Navigate to the `C:\Workshop\Labs\Inputs` folder and select `CreateUXModelingProfile.zip`. Click **Open.**
9. Click **Select All** and then click **Finish**.
10. Open the `UXProfilePlug-in` model.

Task 2: Create the Base Model

In this task, you will create the base model for the template. You create it much like any other model in Rational Software Architect; the major difference is in the intent. Rather than designing a software solution, you want to create a model that guides others in designing software solutions.

1. Create a new UML Project.
   a. Click **File > New > Project**.

b. In the New Project dialog, replace `type filter text` with `UML` and then select **UML Project**, and then click **Next.**

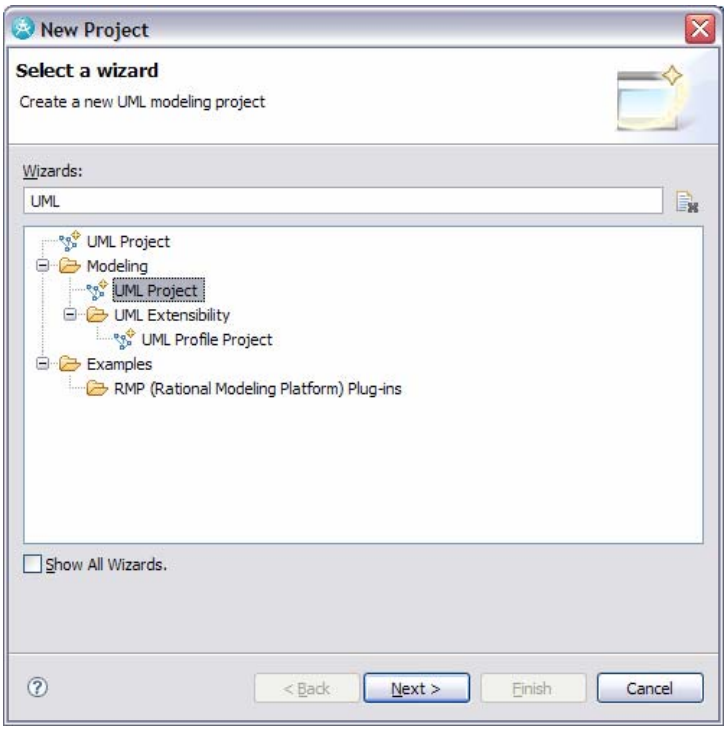


Figure 14-1: *Create a UML Project*

c. Name the project `UXModel Template Project`. Click **Next**.

d. From the **Templates** section, select `Blank Model`.

TIP: In this case you are starting with a Blank Model as you create your template. However, you can select one of the existing templates as the starting point for your own custom template.

e. Enter `UX Model` as the **File name**. Then click **Finish**.

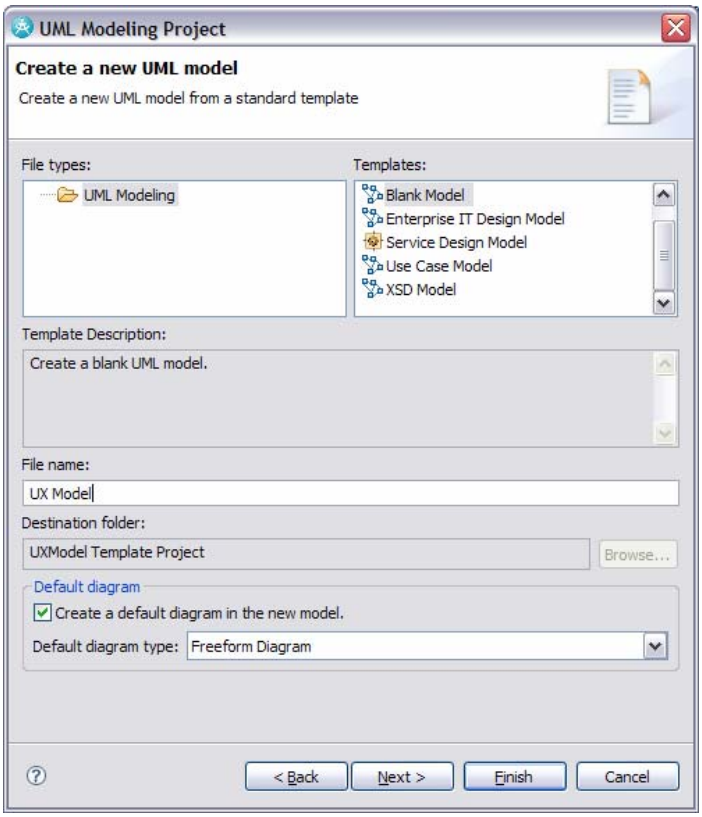


Figure 14-2: *Specify model to add to the project*

f. Select **File > Save All**.

TIP: The `Ctrl-Shift-S` keyboard shortcut will also **Save All**.

Task 3: Create Model Structure

In this task, you will create a set of model elements to be copied and reused as a template.

1. In Project Explorer, create the following package structure within the `UX Model` model:

TIP: Apply required stereotypes from the Properties view.

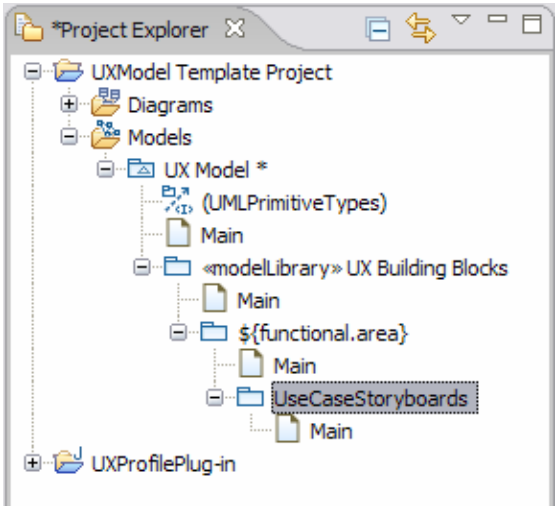


Figure 14-3: *Package structure for the template*

1. Within the UseCase Storyboards package, add a collaboration, a sequence diagram, and a class diagram:

   - Right-click `UseCase Storyboards` and select **Add Diagram > Sequence Diagram**. Note that this adds the containing collaboration for us automatically.

TIP: Work with the **Models** in Project Explorer to change model properties.

   - Rename the collaboration to `«use-case storyboard» ${use-case name}`. Note that `«use-case storyboard»` is a keyword, not a stereotype.

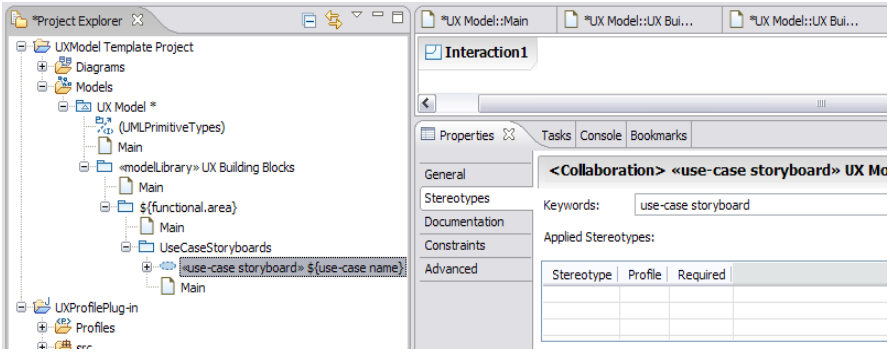


Figure 14-4: *Specifying the keyword*

TIP: Work with the **Diagrams** in Project Explorer to change diagram properties.

   - Rename the interaction `Basic Flow`.
   - Rename the sequence diagram `Basic Flow`.
   - Right-click the collaboration and select **Add Diagram > Class Diagram**.

- Rename the class diagram `Participants`.

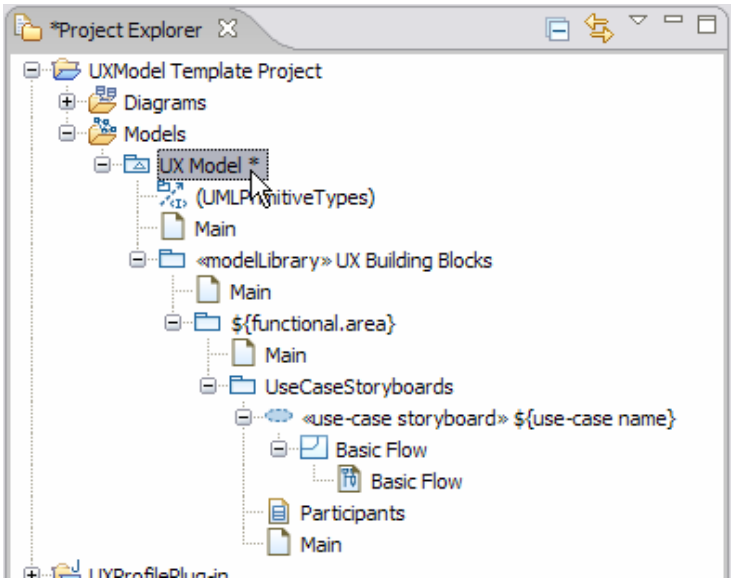


Figure 14-5: *Building blocks*

2. Right-click `UX Model` and select Add **UML > Package**. Name the package `UX Specification Viewpoints`.
3. Add two packages to the UX Specification Viewpoints package, and then name them and apply their **Stereotypes** as follows:
   - `«perspective» Screens`
   - `«perspective» Storyboards`
4. Select the freeform diagram, named `Main`, within the `«perspective» Screens` package and name it `Screens – Overview`.
5. Select the freeform diagram, named `Main`, within the `«perspective» Storyboards` package and name it `Storyboards – Overview`.
6. Select the freeform diagram, named `Main`, within the `UX Model` top level element and rename it `Template – Instructions`.
7. Delete all of the remaining default `Main` diagrams that have been created.

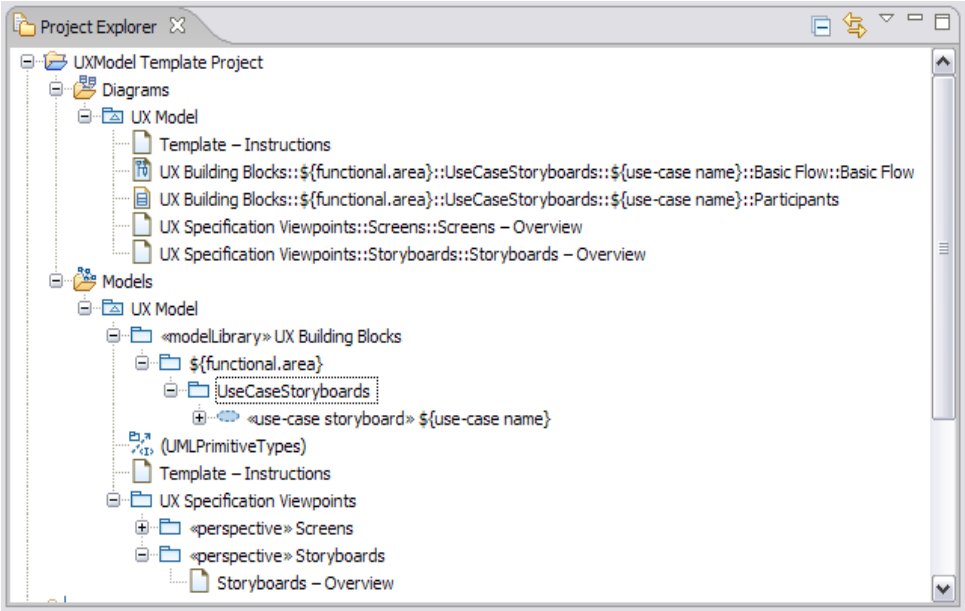


Figure 14-6: *Completed model template structure.*

8. **Save All**.

Task 4: Add Documentation

In this task, you'll add some brief documentation for the user.

1. Open the `Template – Instructions Diagram`.

1. Add two note elements to the diagram, then size and position them as shown in the screen capture below:

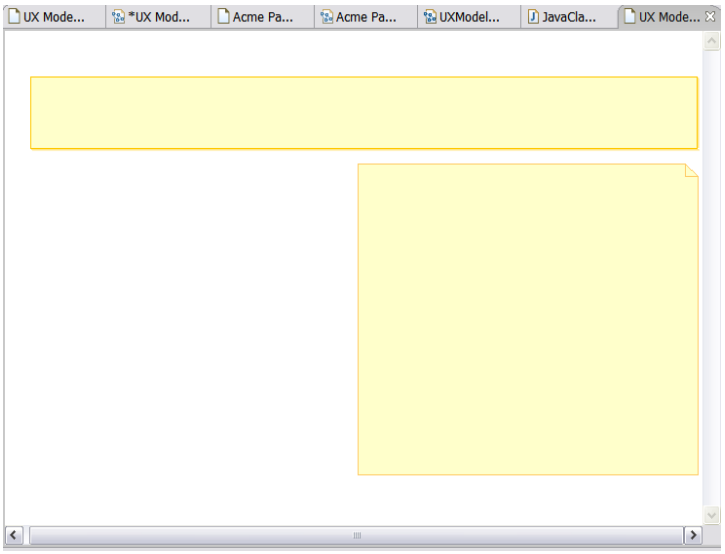


Figure 14-7: *Layout for the note elements.*

2. Add the following text to the top note:

   `UX Model Template`

3. Add the following text to the bottom note:

TIP: This text can be found at `C:\StudentWork\Labs\Inputs\ UX Model Template Note.txt`.

```
This model contains two main types of packages:


1.  A set of reusable packages and diagrams that should be used to set
up your model.  You will find these elements in the «modelLibrary» UX
Building Blocks package.


2.  A set of <<perspective>> packages that will contain diagrams that
will provide additional viewpoints on how the specified services are
composed, consumed and behave.  Additional <<perspective>> packages
should be added if new audiences or viewpoints need to be addressed.
No semantic elements should reside in these packages - just packages
and diagrams.


Users of this model can double-click the diagram links to navigate
through the main areas of the specification.  Update the links as
necessary based on any adjustments that you make to the model
structure.


WHEN YOU NO LONGER NEED THESE INSTRUCTIONS:

1.  Delete this note from the diagram.
```

4. Add links to the «perspective» diagrams:

- From the Project Explorer, drag the `Screens – Overview` diagram to the `Template – Instructions` diagram in Diagram Editor.
- From the Project Explorer, drag the `Storyboards – Overview` diagram to the `Template – Instructions` diagram in Diagram Editor.

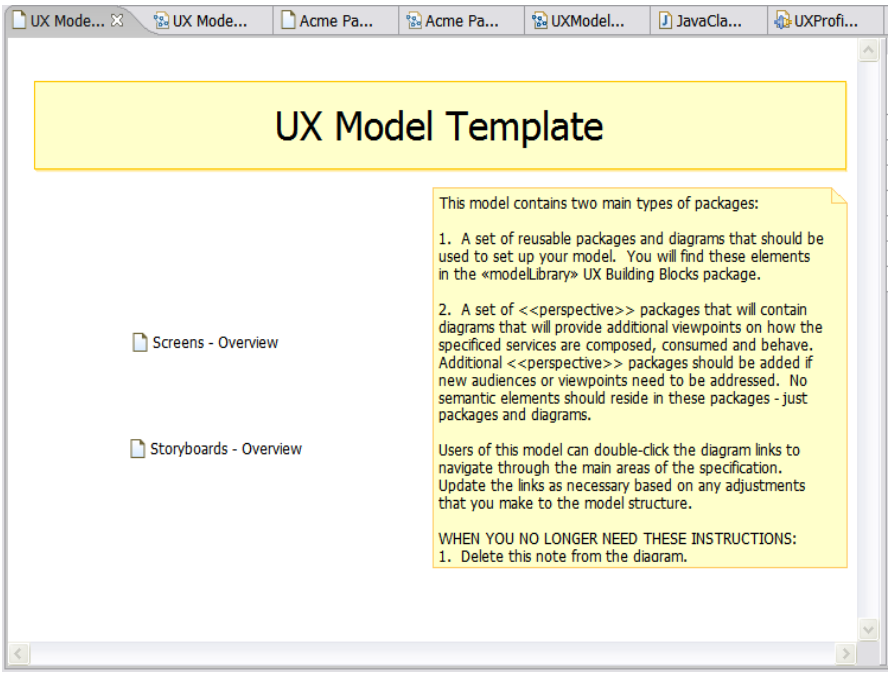


Figure 14-8: *Completed Template – Instructions diagram.*

5. **Save All**.

Task 5: Add the Model as a Template to the UXModeling Profile Plug-in

TIP: At this point, you have a model template that can be reused within your Workspace. For re-use elsewhere, this project can be **exported** as a **Project Interchange** and then **imported** to another Workspace.

In this task you'll add the model as a template to the UXModeling Profile plug-in.

1. Switch to the Plug-in Development perspective.
1. Select the **UXModelingPlug-in** project.
2. Add a folder to the project and name it `modeltemplate`.
3. Click **File > New > Other** and choose **UML Model.**
4. Click **Next** and select the **Existing Model** radio button.
5. Click **Next** and **Browse** for the **model file** and **Destination folder** shown. Enter `UX Model Template` as the **File name.**

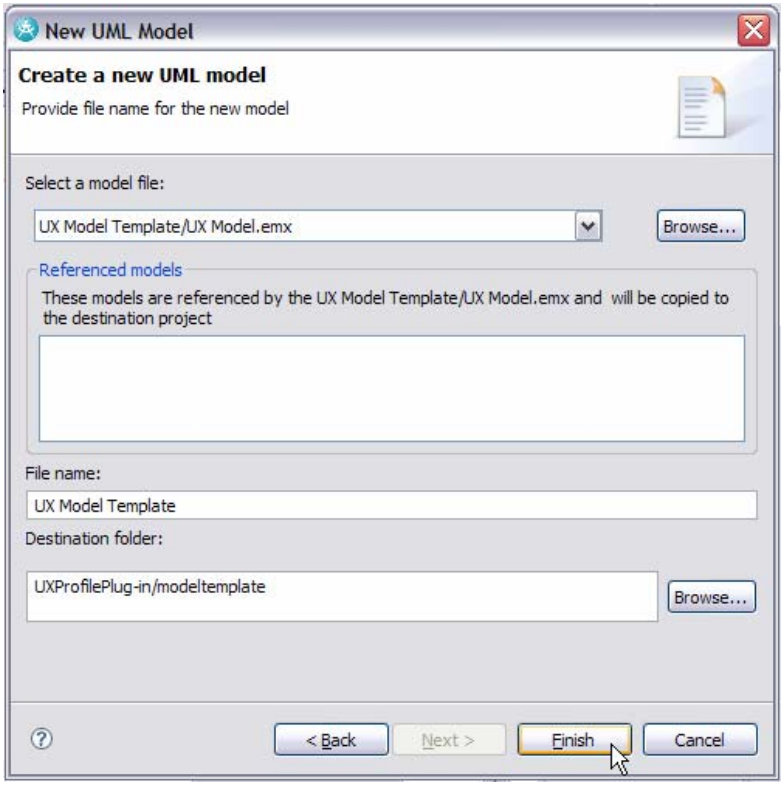


Figure 14-9: Completed UML Model Creation dialog.

6. Click **Finish**. Choose **OK** if a Java Modeling enablement dialog appears.
7. Close the UXModel Template Project project. You will work strictly with the model template added to the UXModelingPlug-in project.
8. Open the plugin.xml file associated with the UXModelingPlug-in project.
9. Select the **build** tab.
10. In **Binary Build**, select the box corresponding to the `modeltemplate` folder.

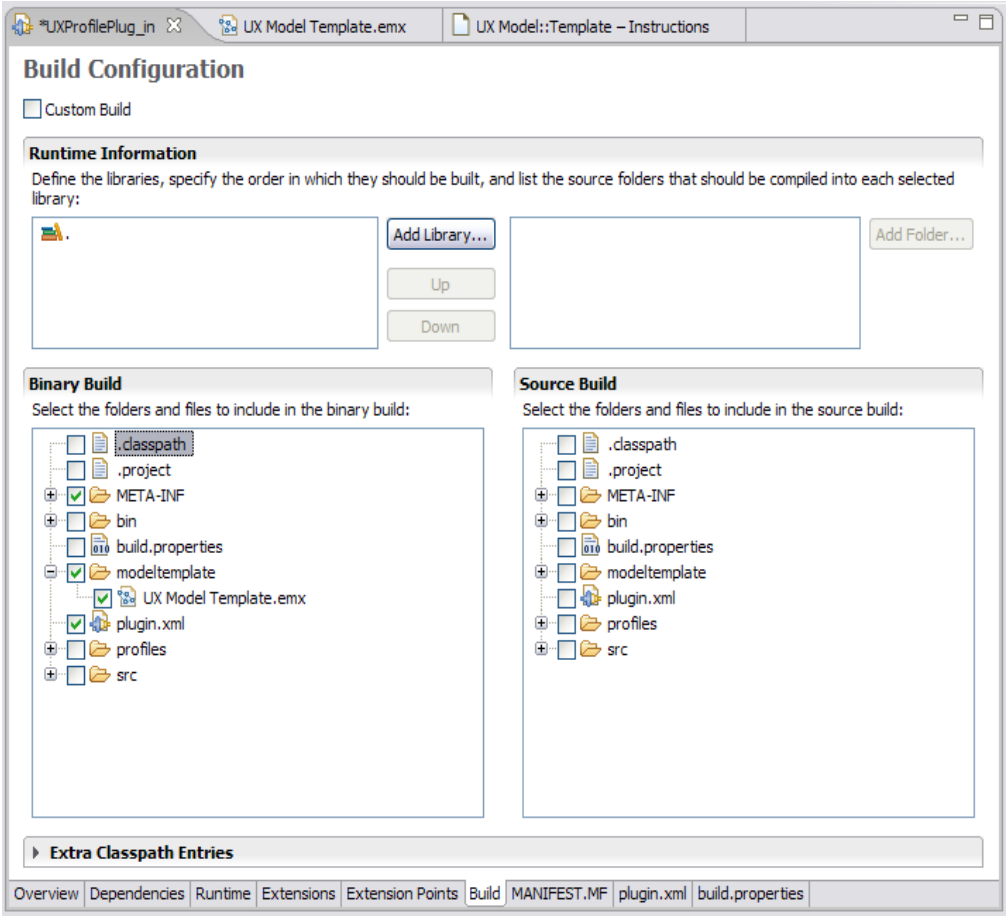


Figure 14-10: *Build tab contents after selecting modeltemplate folder.*

11. **Save All**.

Task 6: Apply Profile to the Model Template

In this task, you will apply the `UXModeling` profile to your model template. This way, when someone uses the model template, the profile will already be applied for them. Configure a Run-time workbench to use in applying the profile to your model template.

1. Open the plugin.xml file.

1. Select the **Overview** tab of the `plugin.xml` file in the manifest editor.

2. Click **Launch an Eclipse application**.

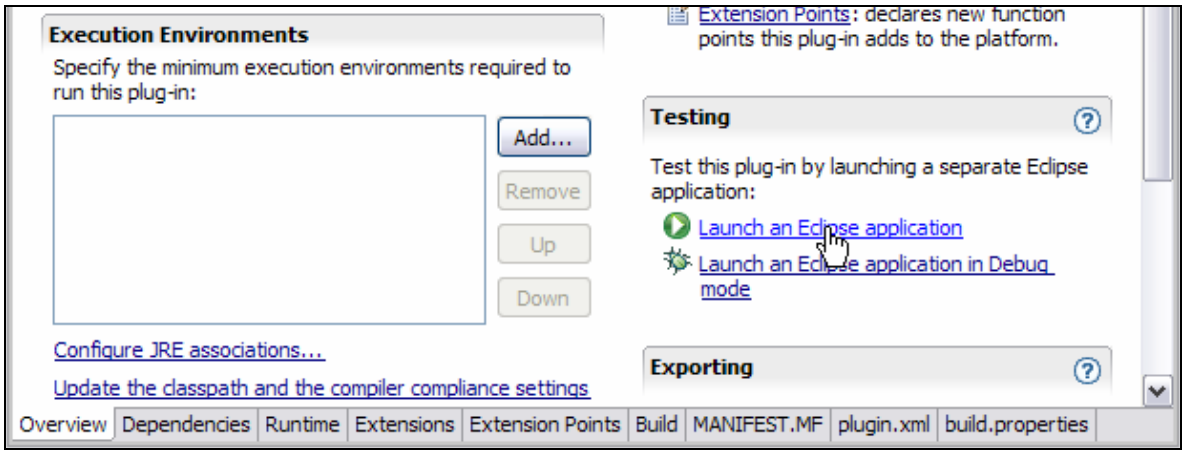


Figure 14-11: *Launching a Run-time Workbench configuration.*

3. Close the Welcome screen if it appears.
4. Switch to the Modeling perspective in the Run-time workbench.
5. Create a new UML Project, named `Test`, and add a blank model to the project. `Blank Model` is fine for the **File name.**
6. Delete the model from the project.
7. Select **File > Import**.
8. Select **File system**. Click **Next**.
9. Click **Browse** and navigate to `C:\Workshop\StudentWork\CreateAModelTemplateWorkspace\UXProfilePlug-in`. Click **OK**.
10. Select `UX Model Template.emx`. Ensure that the **Into folder** matches the name of the UML Project created previously.

Figure 14-12: *Importing the model file for the template.*

11. Click **Finish**.
12. Double-click `UX Model Template.emx` to open the model.
13. Open the model in the Project Explorer view.
14. In the Properties view, select the **Profiles** tab
15. Click **Add Profile.**
16. Select the `UXModeling` profile. Click **OK**.

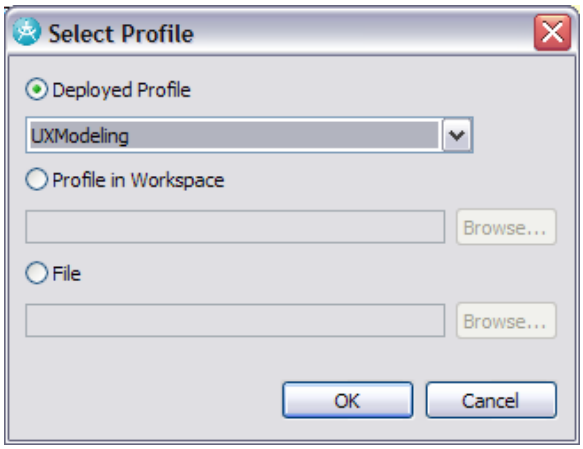


Figure 14-13: *Specifying the profile.*

17. **Save All**.
18. Close the runtime workbench.
19. Switch to the host workbench.
20. Delete the existing copy of the model template, `UX Model Template.emx`, found in `UXProfilePlug-in`.
21. Select **File > Import**.
22. Select **File system**. Click **Next**.
23. Click **Browse** and navigate to `C:\Workshop\StudentWork\runtime-EclipseApplication\Test`. Click **OK**.
24. Select `UX Model Template.emx`. Ensure that the **Into folder** is set to `UXProfilePlug-in`.

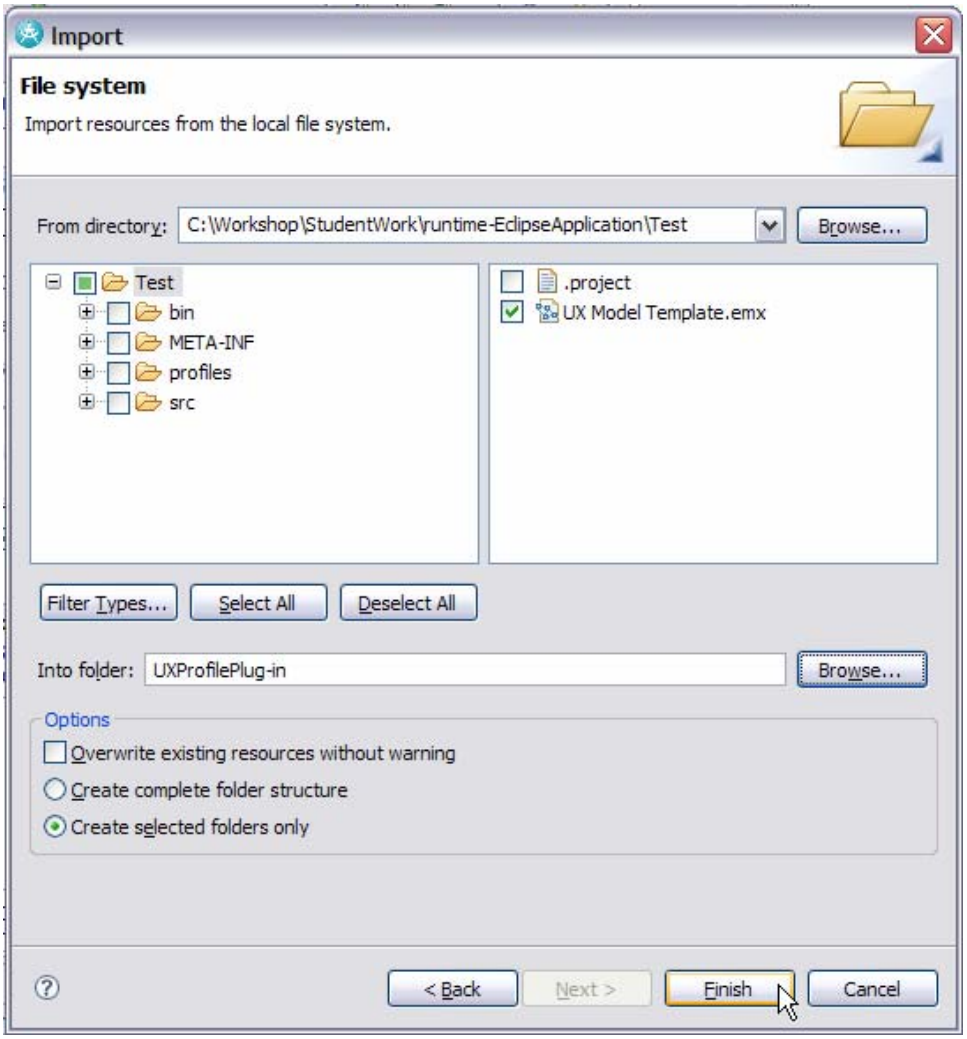


Figure 14-14: *Importing the template back into the plug-in project.*

25. Click **Finish**.

TIP: To double-check that you have a valid reference from the model template to the profile, you can open the emx file in a text editor and confirm that the pathmap is being used.

TIP: Model templates can be contributed via plug-ins by using the `com.ibm.xtools.modeler.ui.wizards.template` extension point. By contributing in this way, the user will no longer need to find the location of the template on disk. Instead, the newly registered template will show up in the Creation wizard with the other templates (for instance, Analysis, EJB, WSDL, Use Case, Blank, and so on).

Lab 15 – Package Reusable Assets

Objectives

After completing this lab, you will be able to:

- Package a RAS asset that contains a profile, pattern, model template, and a transformation
- Import the RAS assets

Given

- A project interchange file, `UXPackaging.zip`, which contains the reusable assets that we are going to package and deploy.
- `UXTransformationTest.zip`

Scenario

In this portion of the workshop, you will create a RAS asset that contains the reusable assets that you have created during the course, including a profile, pattern, model template, and transformation. Once you have packaged these artifacts as RAS assets, you will test the import of the assets in Rational Software Architect.

Task 1: Create the Workspace

In this task, you will switch to a new workspace named `PackagingWorkspace` that you will create.

1. From the **File** menu, select **Switch Workspace**.
2. In the Workspace Launcher window, replace the displayed text with `C:\Workshop\StudentWork\PackagingWorkspace` and click the **OK** button.
3. Close the Welcome screen.

Task 2: Create a RAS Repository

In this task you will create a Repository that will be used to manage RAS assets.

1. Switch to the RAS (Reusable Assets) perspective.
2. Set up a local repository:
   - If necessary, open the RAS Asset Explorer by clicking **Window > Show View > Other > RAS > Asset Explorer.**

- In the Asset Explorer, add a new **Local Repository**  and click **Next**.

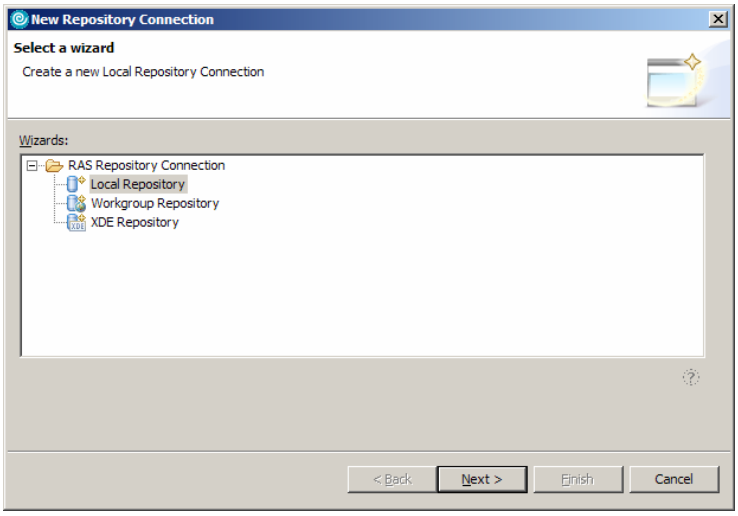


Figure 15-1: *New Repository Connection Dialog*

3. Accept the default **Repository Name** and **Repository Location**. Then click **Finish**.

Task 3: Import Reusable Assets

In this task, you will import the reusable assets that we want to package.

1. Switch to the Plug-in Development perspective.
2. Select **File > Import**.
3. Select `Project Interchange`. Click **Next**.
4. Click **Browse** and select `UXPackaging.zip` from the `C:\Workshop\Labs\Inputs` directory. Click **Select All**. Then click **Finish**.
5. Quickly review the artifacts as shown within the Package Explorer view.

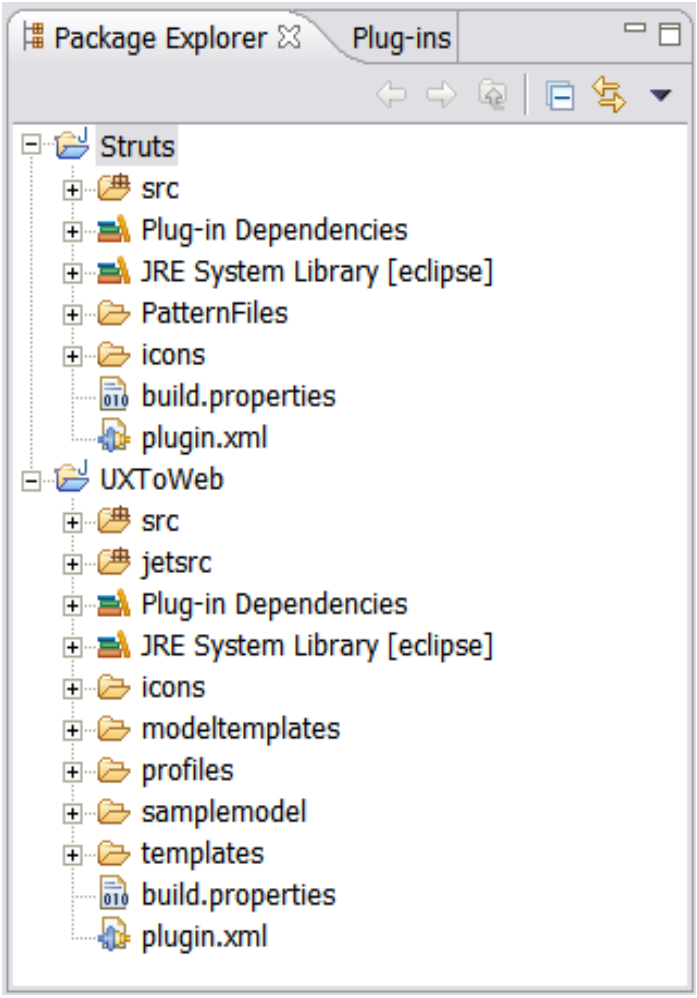


Figure 15-2: *Imported elements within the Package Explorer*

Task 4: Create a Feature

In this task, you will create an Eclipse Feature that will be associated with the plug-in which contains the reusable asset that we've built.

1. Select **File > New > Project**.
2. Select **Feature Project**. Click **Next**.

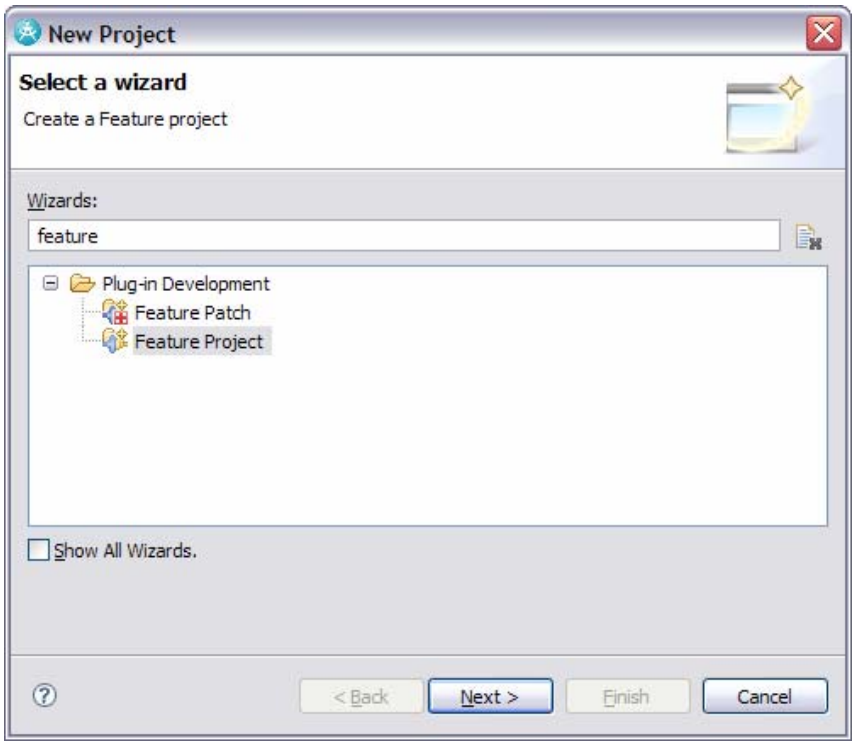


Figure 15-3: *Create a new feature project*

3. Enter `com.ibm.workshop.ux.feature` as the **Project name** and accept the defaults on the Feature Properties dialog. Click **Next**.
4. Select `UXToWeb (1.0.0)` and `Struts (1.0.0)` as the **Referenced Plug-ins**. Click **Finish**.

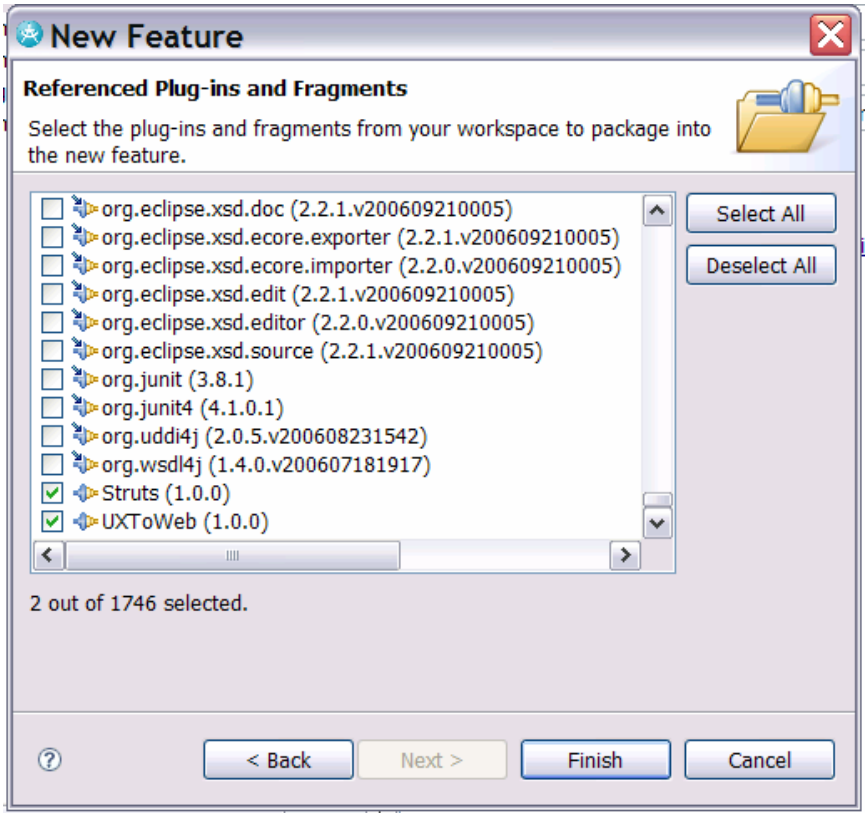


Figure 15-4: *Select the plug-ins that the feature should reference*

TIP: The `feature.xml` file is opened by default in the manifest editor. When distributing your own assets, you will want to enter details on the **Information** tab corresponding to a description of the asset, copyright information, and licensing details.

5. Select **File > Save All**.

Task 5: Deploy as a RAS Asset

In this task you will package up the feature and associated plug-in project as a RAS asset.

1. Open the `plugin.xml` file for the `UXToWeb` plug-in project.
2. In the manifest editor, switch to the **Build** tab.
3. Confirm that the **Binary Build** section matches that shown below:

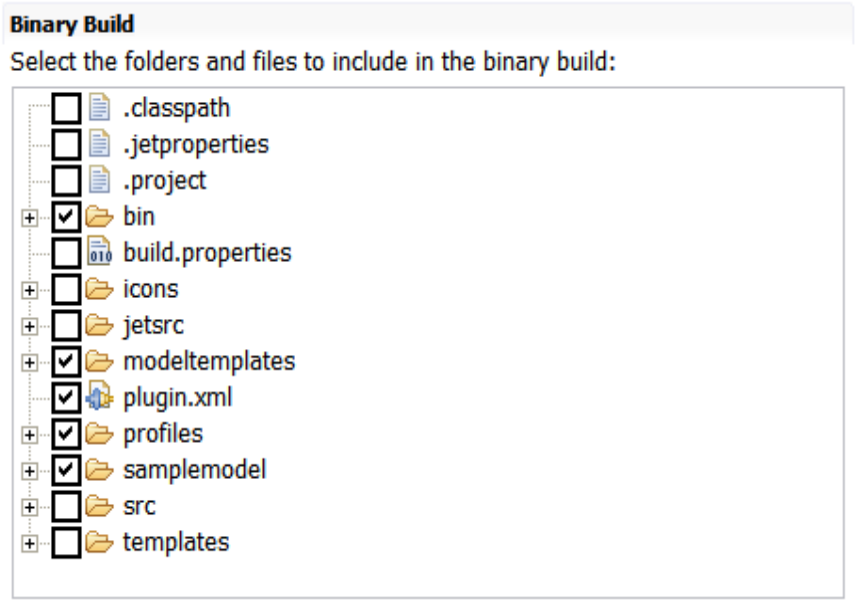


Figure 15-5: *Binary Build section of the Build tab within the plugin.xml file for the UXToWeb project*

4. Select **File > Save All**.
5. Open the `plugin.xml` file for the `Struts` plug-in project.
6. Confirm that the **Binary Build** section matches that shown below:

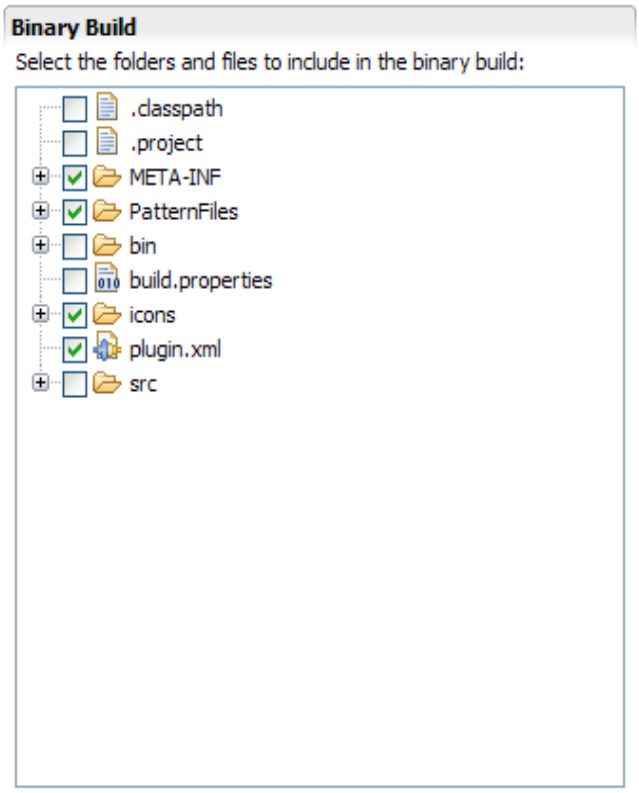


Figure 15-6: *Binary Build section of the Build tab within the plugin.xml file for the Struts project*

7. Select **File > Save All**.
8. On the **File** menu, click **Export**.
9. Select **RAS Asset** and then click **Next**.
10. In the **Destination** field, select **Repository**. Select **My Local Repository** from the **Repository** menu.

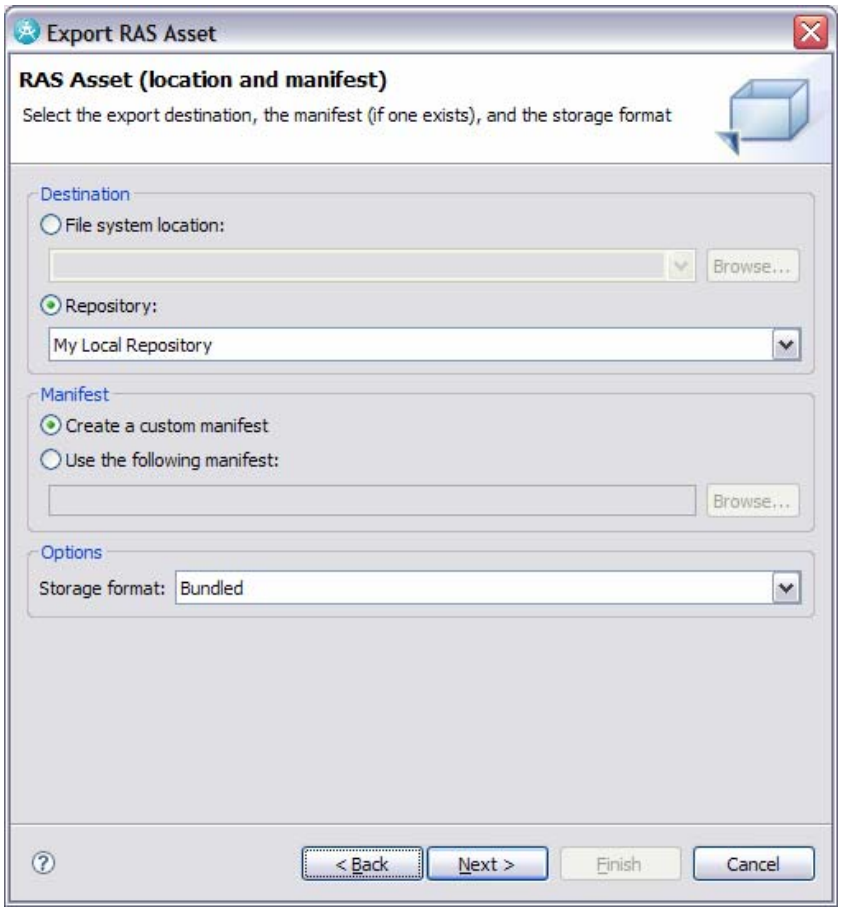


Figure 15-7: *Setting Location and Manifest for RAS Asset*

11. Click **Next**.

12. Enter a description and name for the asset. Click **Next**.

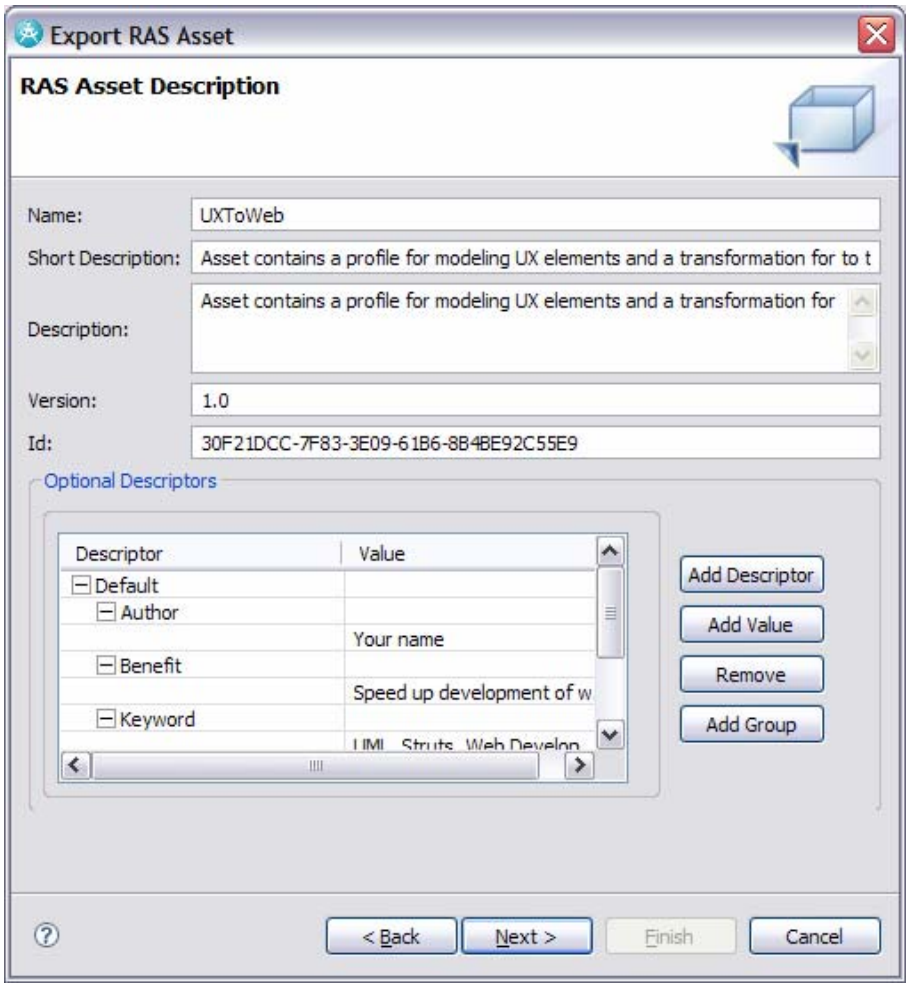


Figure 15-8: *Description for the RAS asset*

13. Choose `com.ibm.workshop.ux.feature` as the resource to export, and ensure that **Export as a deployable feature, fragment or plug-in** is selected.

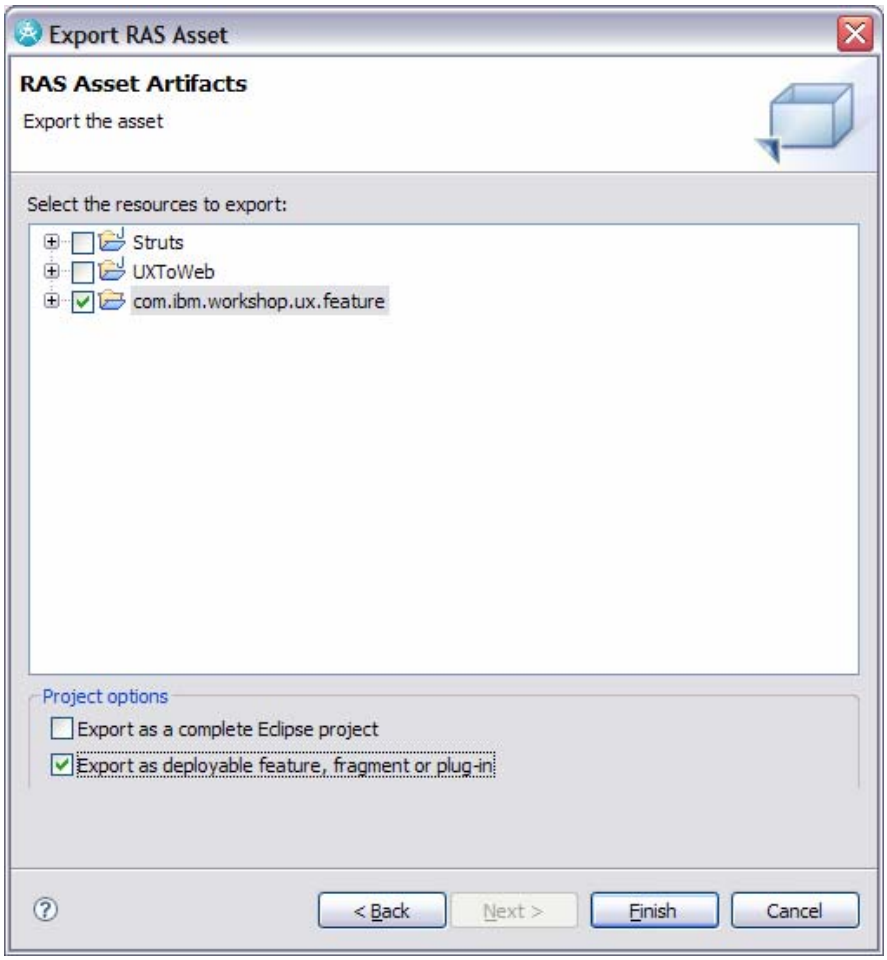


Figure 15-9: *Description for the RAS asset*

14. Click **Finish**.
15. Click **OK** on the Export was successful dialog.

TIP: You can ignore the displayed warnings, as they just point out that RAS is not familiar with some of the file extensions used. Click **OK** to dismiss the warnings.

Task 6: Import the RAS Asset

In this task, you will import the RAS asset that contains the reusable assets. Perform a quick test once you have imported the asset.

1. Switch to the Reusable Asset perspective.
2. Import:
   - Right-click inside the Asset Explorer and click **Refresh**.

- Select the `UXToWeb` asset, right-click and choose **Import**.

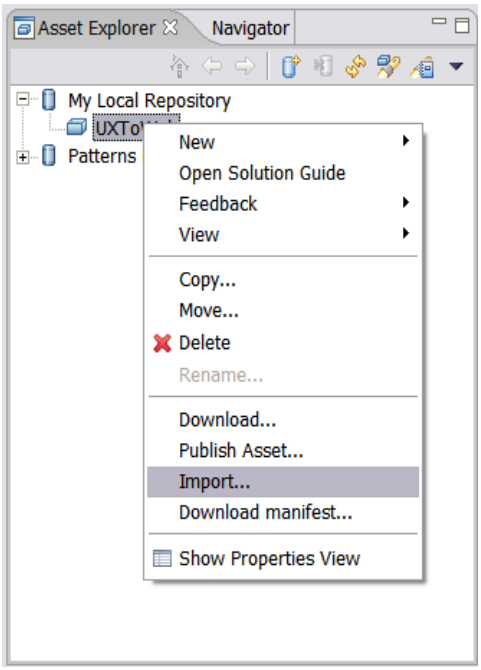


Figure 15-10: *Import the RAS asset*

- Click **OK**, when told about the plug-in that it will install.
- Click **Next** to confirm the asset being imported.
- Accept the terms of the license agreement. Click **Finish**.
- Click **OK** when presented with the **Import Results**.
- Click **Yes** if prompted to restart Rational Software Architect.

Task 7: Verify the install of the RAS Asset

In this task, you will verify that the reusable assets that were contained within the RAS package were installed.

1. Switch to the Modeling perspective.
2. Confirm that the assets were installed:
    - Select **Modeling > Transform > Configure Transformations**.
    - Ensure that `UXToWeb Plug-in` is available within the `UXToWeb` folder. Then click **Close**.

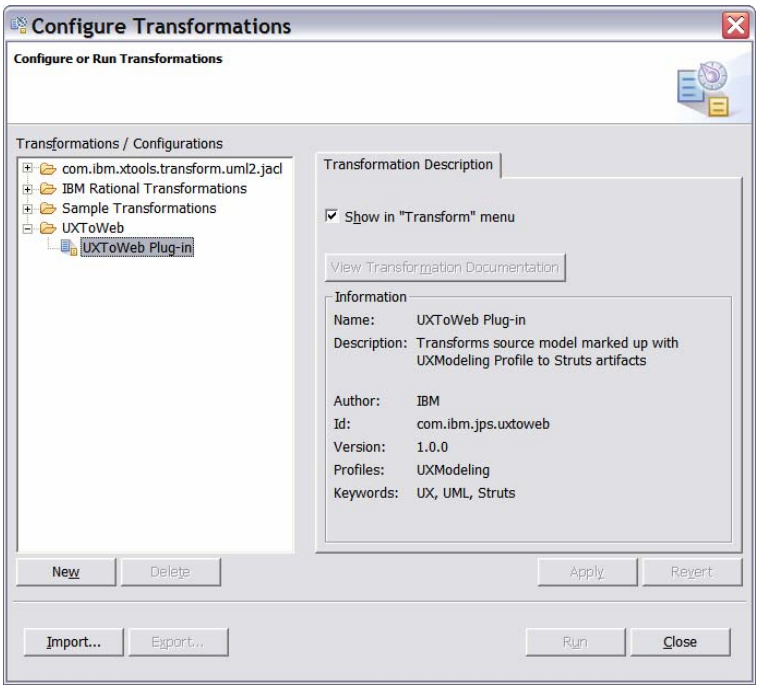


Figure 15-11: *UXToWeb Transformation listed in the Configure Transformations dialog*

- Open the **Pattern Explorer**.
- Ensure that `Master Detail` exists within the `My Struts Patterns` folder.

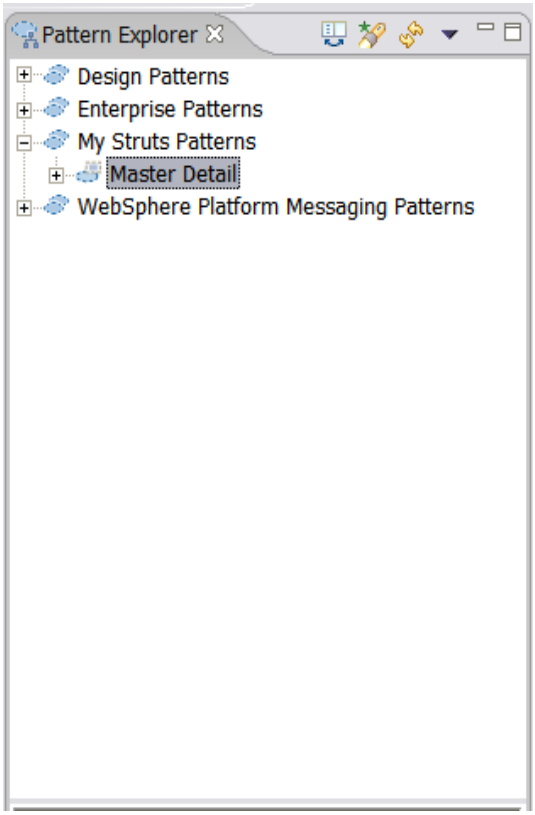


Figure 15-12: *Master Detail pattern in the Pattern Explorer*

- Select **File > New > UML Model**.
- Confirm that `UX Model Template` is available in the **Templates** list.

Task 8: Test the RAS Asset

In this task, you will use a sample model to test the asset.

1. Select **File > Import**.
2. Select **Project Interchange**.
3. Click **Browse** and navigate to `C:\Workshop\Labs\Inputs` and select `UXTransformationTest.zip`.
4. Click **Select All**.
5. Click **Finish**.
6. Within the `UXTestModel`, navigate to the `com.ibm.strutssample` package and open the `Main` diagram.
7. Review the stereotypes on the model elements to ensure that they match those shown in the screen capture below:

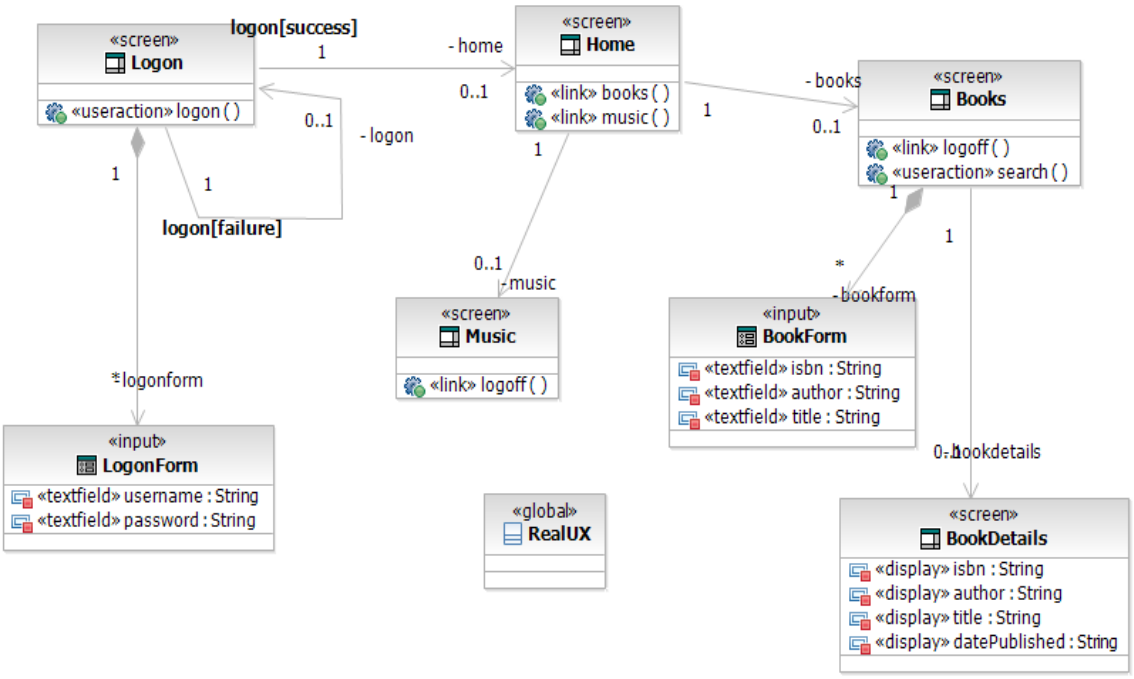


Figure 15-13: *Class diagram depicting elements in test model*

1. Apply the pattern:
   - Open the `UXTestModel` model
   - Add the following classes to the com.ibm.strutssample package:
     - `MusicDetails`
     - `MusicList`
   - Add the following attributes to the MusicDetails class:
     - `artist : String`
     - `recordingDate : String`
     - `genre : String`
     - `rating : String`
   - Add the following attributes to the MusicList class:
     - `artist : String`
     - `rating : String`
2. The classes we will use with the pattern are `Music`, `MusicDetails`, and `MusicList`:

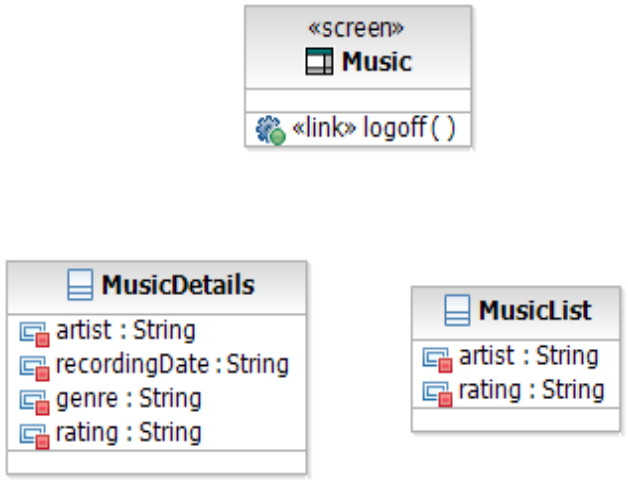


Figure 15-14: *Classes to use as parameters for the pattern*

3. Apply the `Master Detail` pattern using the classes shown above as parameters.
   - Add a new Class Diagram to the `com.ibm.strutssample` package. Name the diagram `Music-MasterDetail`.
   - Drag the `Master-Detail` Pattern from the **Pattern Explorer** and drop it on the `Music-MasterDetail` class diagram within the **Diagram Editor**.

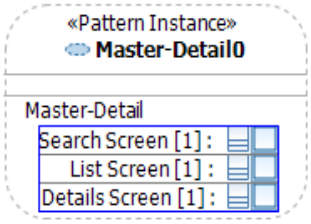


Figure 15-15: *Pattern instance on class diagram*

   - Drag the `Music` class from the **Model Explorer** to the `Search Screen` parameter of the `Master-Detail` pattern
   - Drag the `MusicList` class from the **Model Explorer** to the `List Screen` parameter of the `Master-Detail` pattern
   - Drag the `MusicDetails` class from the **Model Explorer** to the `Details Screen` parameter of the `Master-Detail` pattern.

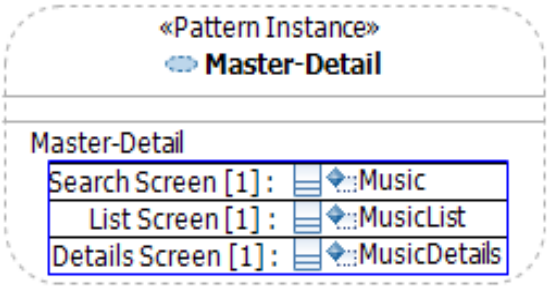


Figure 15-16: *Classes bound to the pattern*

- Drag the following classes from the **Model Explorer** to the `Music-MasterDetail` class diagram:

  i. `Music`
  ii. `MusicList`
  iii. `MusicDetails`
  iv. `MusicForm`
  v. `MusicListResults`

4. The results should appear as follows:

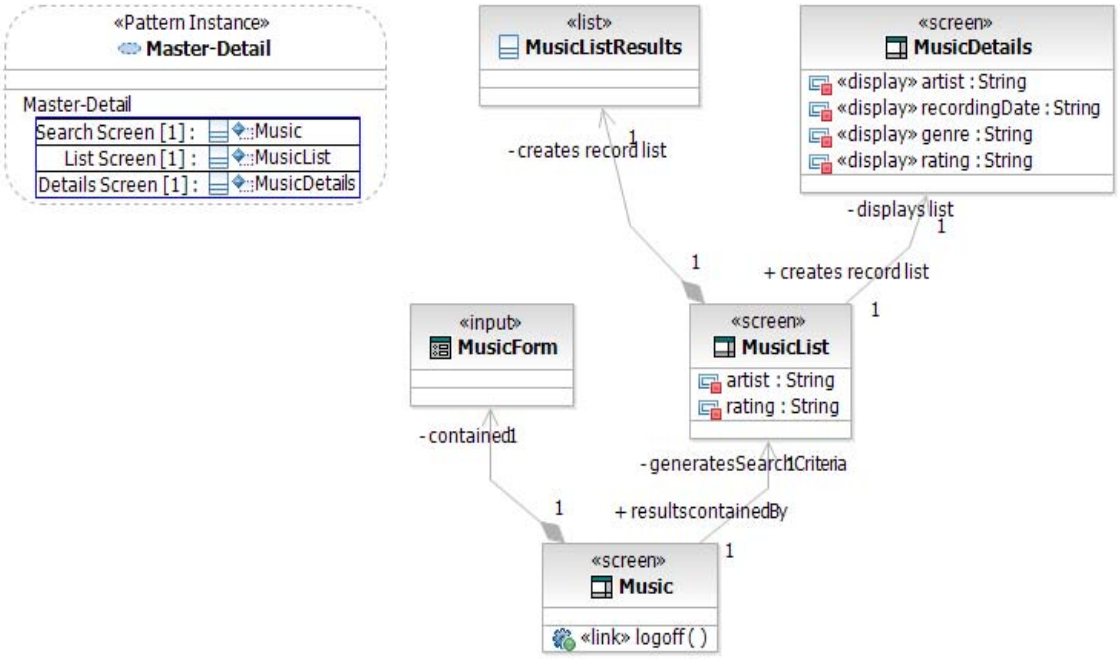


Figure 15-17: *Pattern instance, parameters and generated elements*

Lab 16: Running a GMF Editor

Run Pre-built GMF generated Editor for the Console's Input XML File

Objectives

After completing this lab, you will be able to:

- Understand what a GMF editor can look like and how it behaves

Given

This lab is based on the ongoing Console transformation example. All of the projects that are used are imported into an empty workspace.

Scenario

In the EMF Lab, you built an EMF API for the XML file used as an input for the JET Console transformation. You also built a simple non-graphical editor.

In this lab, you use a GMF-generated editor to edit the Console transformation's input file. The next lab walks through the steps to run a pre-built GMF editor.

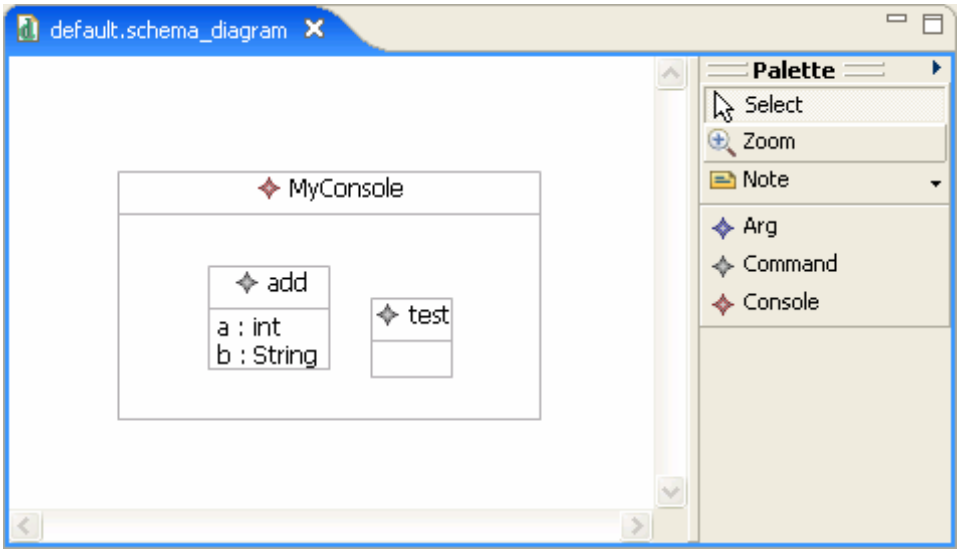


Figure 16-1: *A specialized Console editor*

Task 1: Create and Prepare the Workspace

You will load the pre-built editor projects into an empty Workspace.

1. Open Rational Software Architect with a new workspace for this lab, such as "c:\GMF Demo Workspace".
2. Open the Preferences window (select menu **Window > Preferences**). Expand the **General** option and select **Capabilities**. Find **Development** (or **Eclipse Developer**) in the **Capabilities** list and make sure that the checkbox is selected. If the checkbox is empty or is filled in with a square, click it until you see a check mark. This enables all of the **Eclipse Developer** capabilities, which includes EMF.

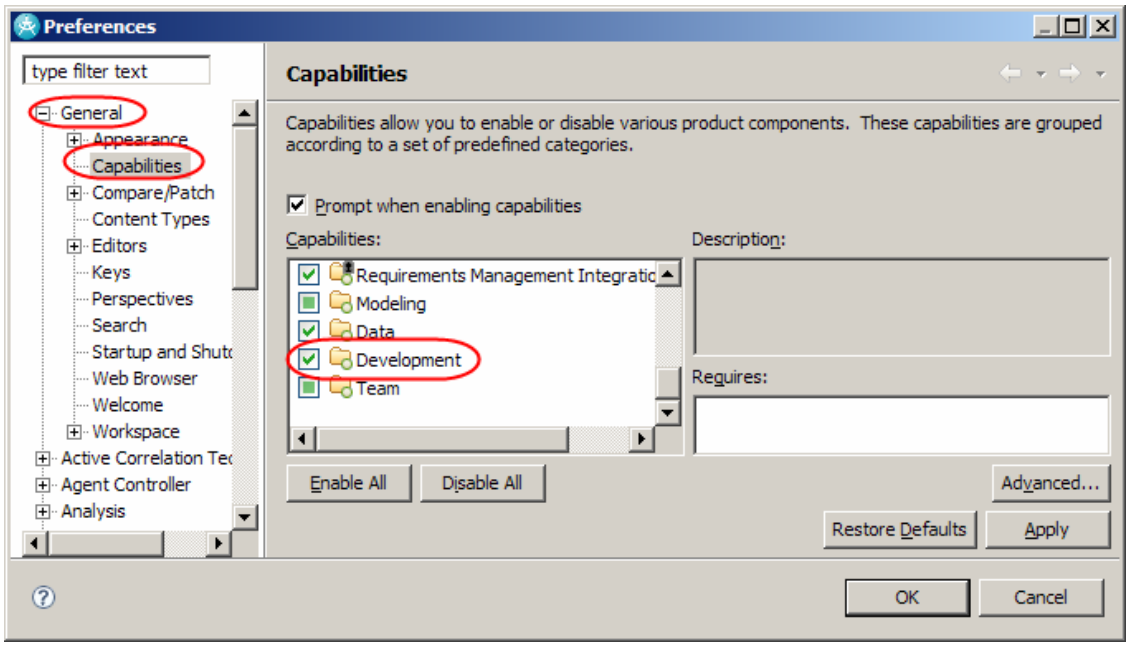


Figure 16-2: *Enabling the Eclipse Developer capabilities*

3. Click **OK** when you are done.
4. Import all of the projects from the Project Interchange file `GmfSolutionPI.zip`.

Task 2: Run the Editor

In this task, you will run the generated editor.

1. In Navigator or Package Explorer, right-click the project named `lab.console.input.diagram` and select **Run As > Eclipse Application**. Then wait for a new instance of Rational Software Architect to launch.
2. In the new instance of Rational Software Architect, create a simple project named console.diagram.test.

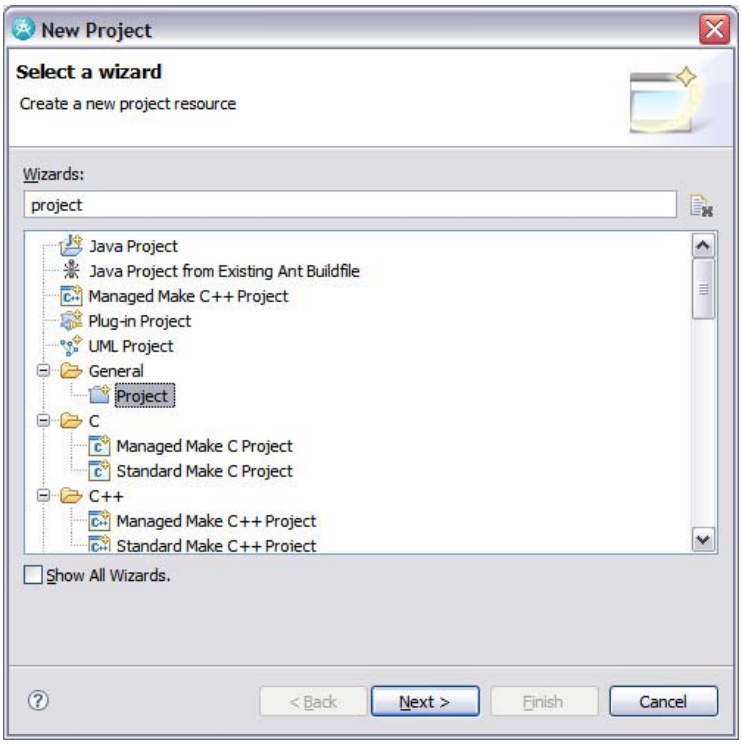


Figure 16-3: *Creating a simple Project*

3. Right click the new project name and select **New > Other**. Select the Input Diagram wizard and click **Next**.
4. Accept the **default** of `default` and click **Finish**.
5. `default.input_diagram` should be opened in an editor that looks like the following.

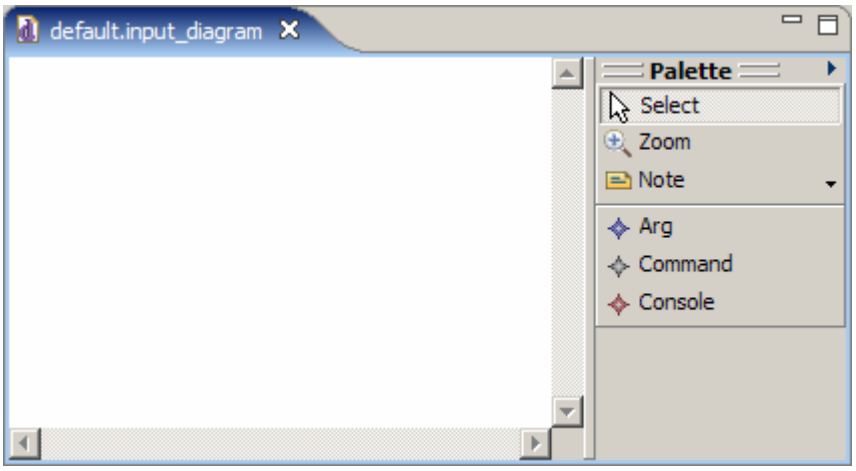


Figure 16-4: *Viewing default.input_diagram in the editor*

6. To add a new Console, click Console in the **Palette** and then click the drawing surface. Name the new Console `MyConsole`. Open up the properties for `MyConsole` and set the **Package** to `my.console`.

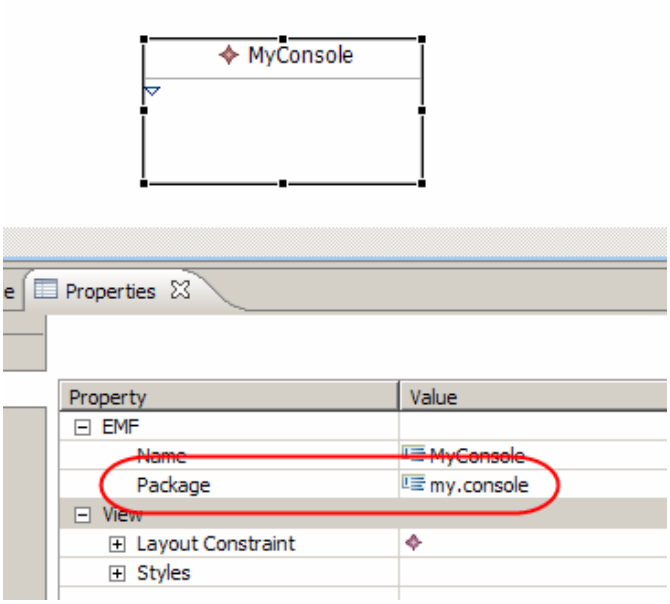


Figure 16-5: *Setting the Package property for MyConsole*

7. In the Diagram editor, expand the node for `MyConsole` so that there is room to work within the compartment.

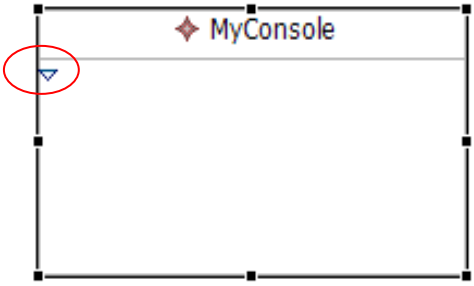


Figure 16-6: *Expand the compartment within MyConsole*

8. To add a child Command, click **Command** in the **Palette** and then click in the compartment in `MyConsole`. Name the Command `echo`.

9. Click **Arg** in the Palette and then in the compartment inside of echo Command to add an argument. Give the new Arg a label of `text:String`. Open the **Properties** of the **Arg** and you should see that the **Name** is `text` and the **Type** is `String`.

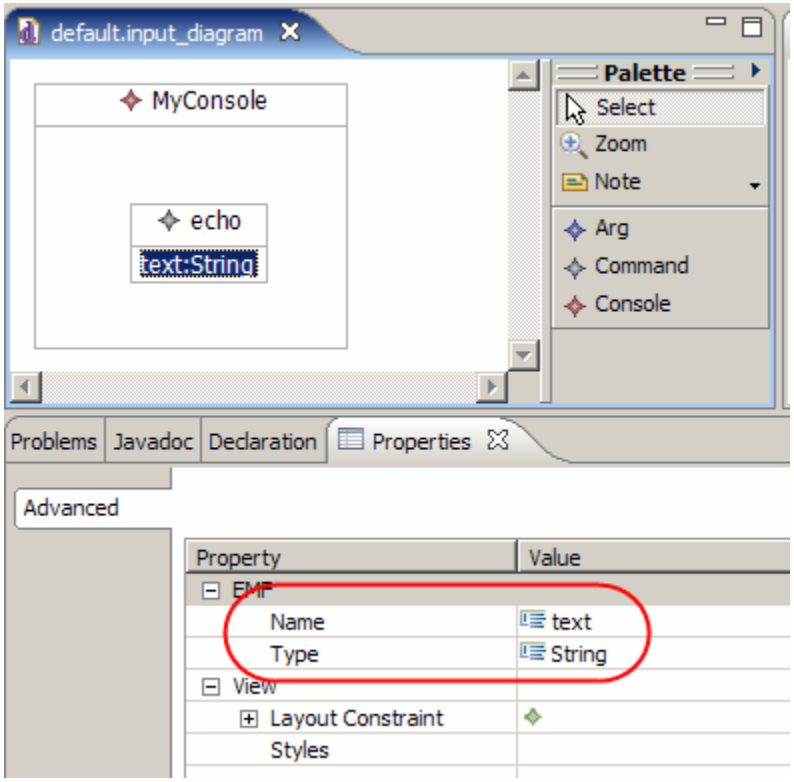


Figure 16-7: *Viewing the properties of the Arg element.*

10. Add any other **Consoles**, **Commands**, and **Args** that you want.
11. In order to test the transformation, save and close your diagram.
12. It's easier to test the existing transformation if the file has an XML extension, so rename `default.input` to `default.input.xml`.
13. Right-click `default.input.xml` and select **Run As > Input for JET Transformation**. In the Properties page that appears, select `lab.console.transform` as the **ID**. Then click **OK** to run the transformation.

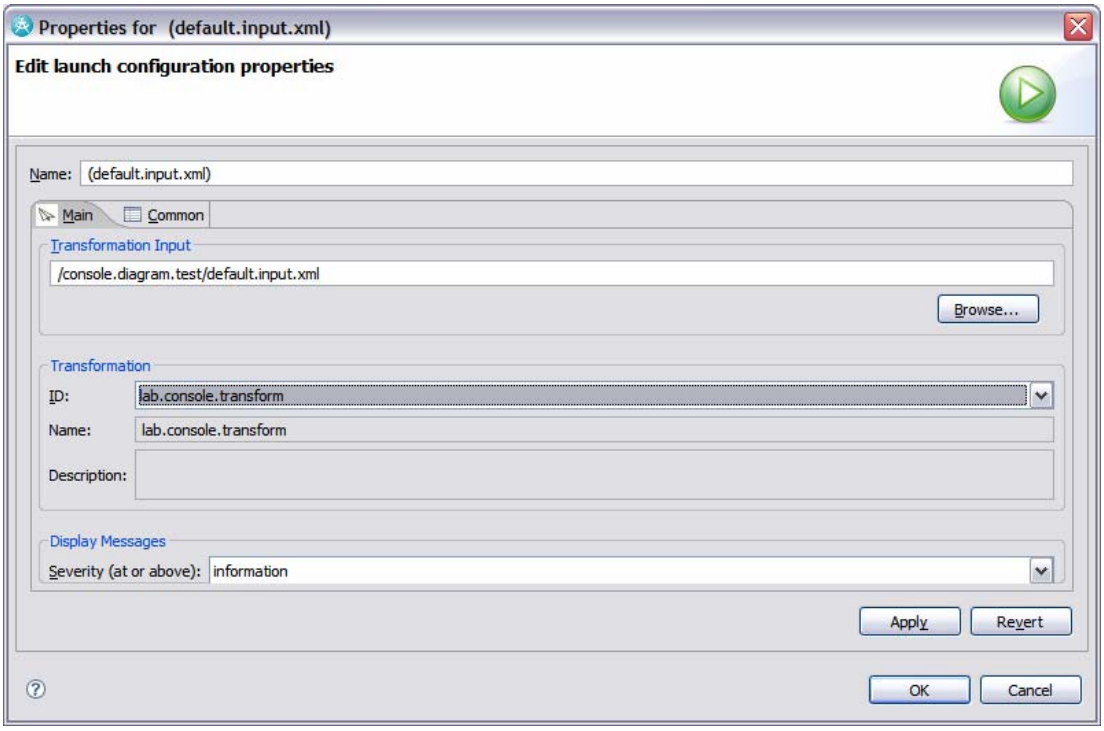


Figure 16-8: *Selecting the transformation to run.*

14. The project `MyConsole Console` (and any other consoles in that you defined) are generated.

15. Close the second instance of Rational Software Architect when you are done testing.

Lab 17: Building a GMF Editor

Build a GMF Editor for the Console's Input XML File

Objectives

After completing this lab, you will be able to:

- Create a custom Graphical Editor using GMF to edit an XML file.

Given

- This lab continues at the end of the EMF Lab.

Scenario

In the EMF Lab, you built an EMF API for the XML file used as an input for the JET Console transformation. You also built a simple non-graphical editor.

In this lab, you use GMF to build a graphical editor for the Console input file. The result will look like the following.

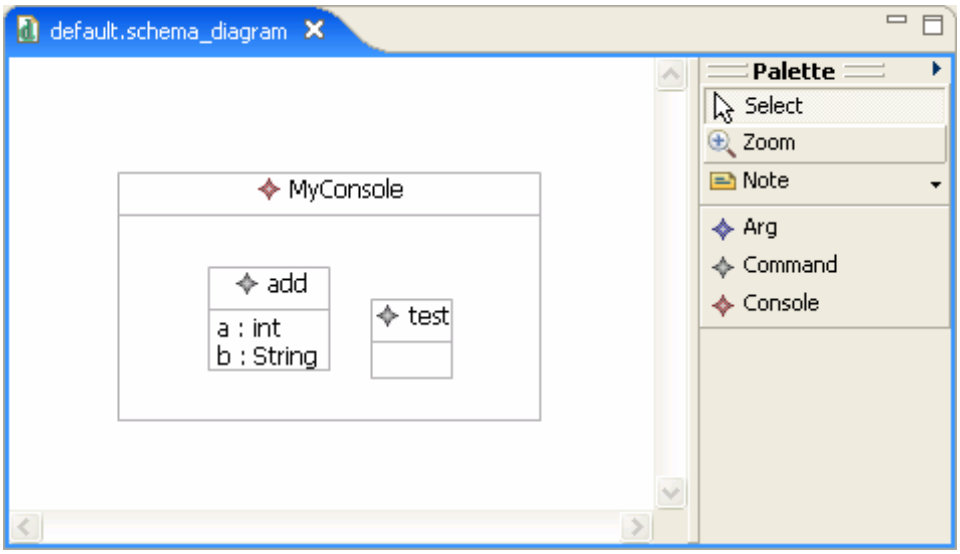


Figure 17-1: *A view of the completed GMF editor for the console example*

Task 1: Create and Prepare the Workspace

If you decide to use the results of the EMF Console lab, simply open that Workspace and skip the rest of this task. Otherwise, you will create a new workspace and import existing projects into it in this task.

1. Open Rational Software Architect with a new workspace for this lab, such as "c:\GMF Lab Workspace".
2. Open the Preferences window (select menu **Window > Preferences**). Expand the **General** option and select **Capabilities**. Find **Development** (or **Eclipse Developer**) in the **Capabilities** list and make sure that the checkbox is selected. If the checkbox is empty or is filled in with a square, click it until you see a check mark. This enables all of the Eclipse Developer capabilities, which includes EMF.

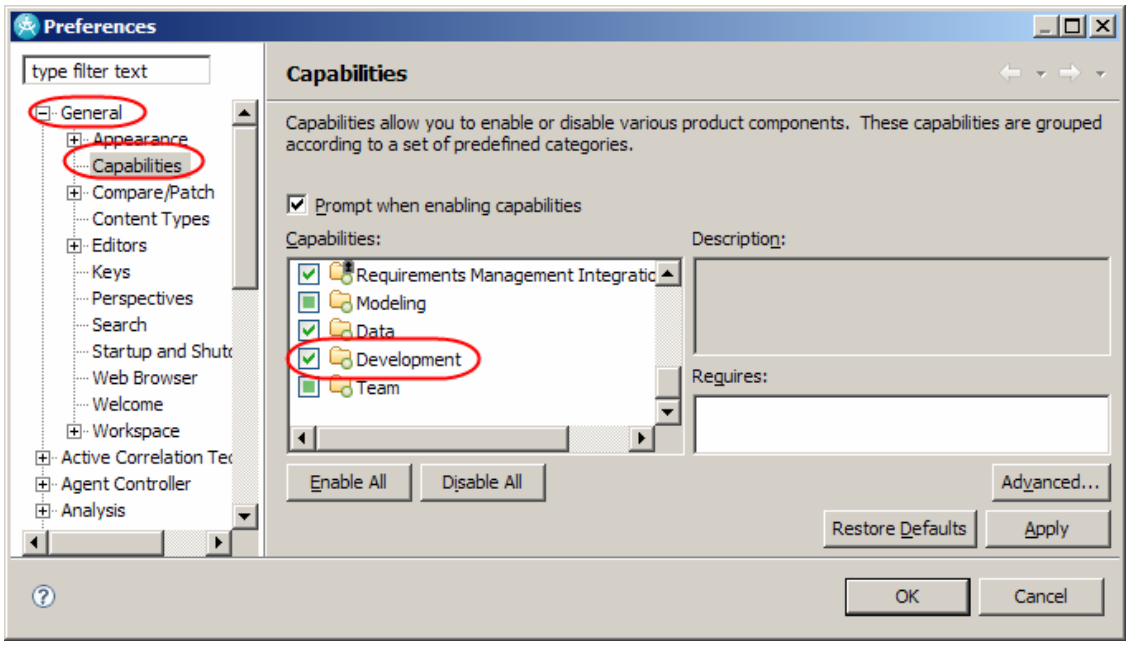


Figure 17-2: *Enabling the Eclipse Developer capabilities*

3. Click **OK** when you are done.
4. Import all of the projects from the Project Interchange file `EMFLabSolutionPI.zip`.

Task 2: Create GMFGraph

A GMFGraph Model is a model file (with the extension GMFGraph) which defines the graphical elements of a GMF editor. For example, it defines how nodes and relationships are drawn.

1. Within the `lab.console.input` project, right-click `model\Input.ecore` and select **New > Other**. Select the GMFGraph Simple Model wizard and select **Next**. Note, do NOT select the **GMFGraph Model**.

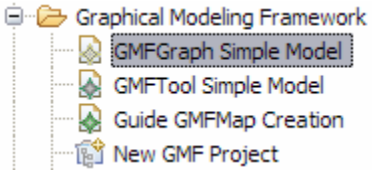


Figure 17-3: *Selecting GMFGraph Simple Model*

2. A default filename of `Input.gmfgraph` should already be filled in, so click **Next**.
3. The `Input.ecore` file that you right-clicked should already be highlighted as the input Domain Model, so click **Next**.
4. Set the Graphical Definition page options as shown below. In particular, the **Diagram element** should be set to `Root`. It is the element in the model that corresponds to the entire diagram. In the **Domain model elements to process** grid, the first checkbox column indicates which Classes in the model will be drawn as nodes in the generated diagram editor. In this example, `Consoles` and `Commands` will be drawn as nodes. The second column indicates which classes and relationships will be drawn as links. In this example, you aren't using any links, so none are checked. The final column indicates what labels are needed for nodes and links. You do want a label for `Arg` elements, but it isn't a node label, so you will manually add it shortly.

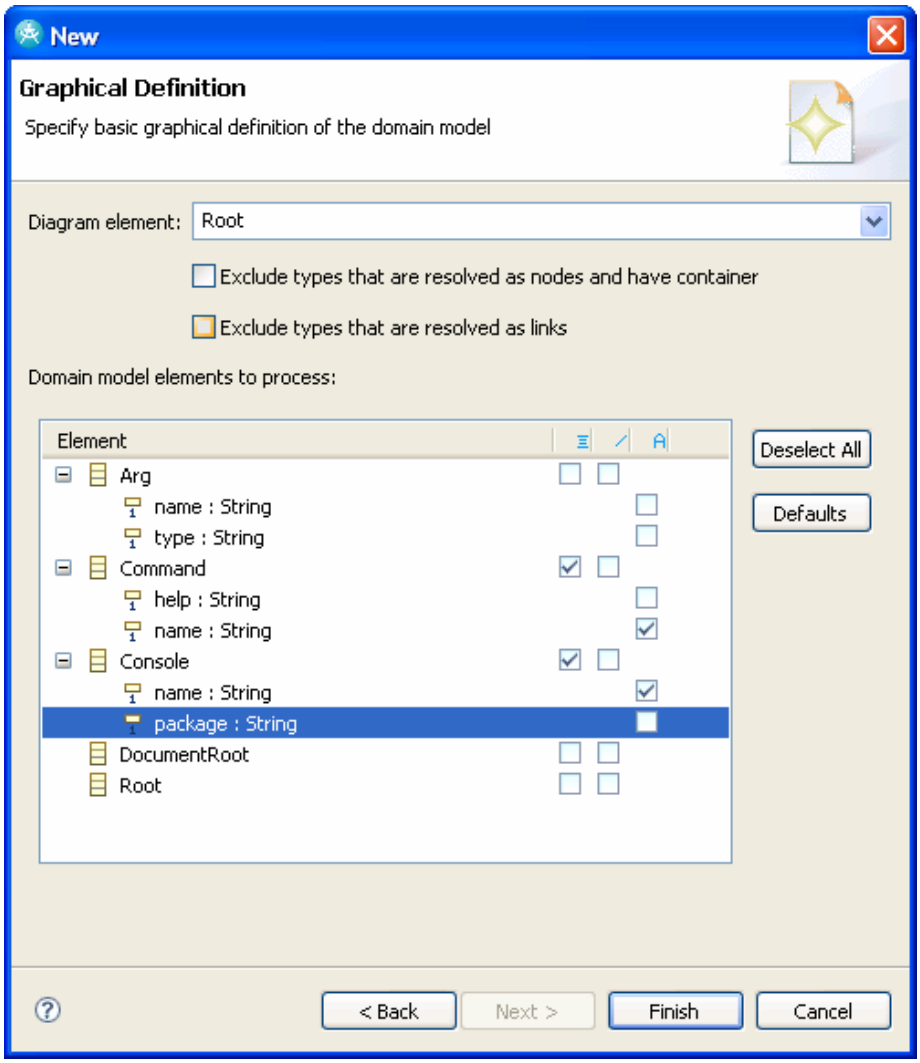


Figure 17-4: *Graphical Definition Wizard Settings*

5. Finally, click **Finish**. The new file `Input.gmfgraph` is created and opened.

Task 3: Refine the Generated GMFGraph

In this task, you will fine tune the code generation settings and generate the code. The following illustration shows some of the graphical elements that you need.

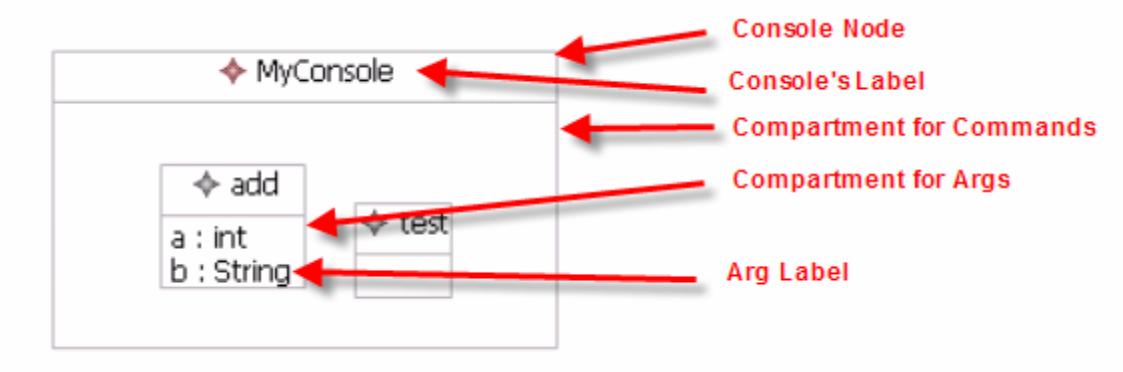


Figure 17-5: *The graphical elements you will need in your editor*

The wizard created a node and a label for the `Console` and `Command` nodes. You need to create a label for the `Arg` elements and compartments for the `Command` and `Arg` elements.

1. Make sure that the file `Input.gmfgraph` is open. You should see an editor like the one pictured below. If you just see a text file, go back to Task 1 and make sure that your workspace has the **Development** capabilities turned on.

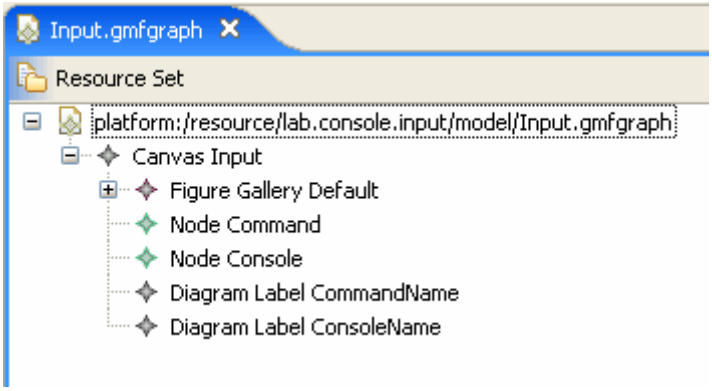


Figure 17-6: *The input.gmfgraph in its editor*

2. The **Figures Gallery** defines low level graphical elements, such as square nodes, elliptical nodes and so on. You need to add a label figure for the Argument label. Right-click **Figure Gallery Default** and select **New Child > Label**.

3. Right-click the newly added label (which is nested under **Figure Gallery Default**) and select **Show Properties View**. In the Properties view, set the name of the label to `ArgLabelFigure`. You should now see the following:

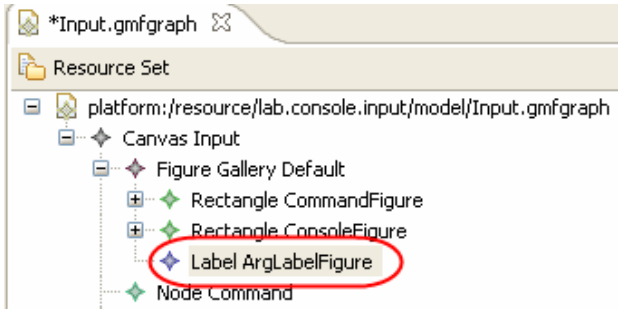


Figure 17-7: *The updated Label element*

4. The elements that are in the **Canvas Input** node are higher-level logical graphical constructs that reference the lower level (physical definition) **Figure Gallery** elements. You need to add a logical `Argument Label` that references the physical `ArgLabelFigure`. Right-click **Canvas Input** and select **New Child > Labels Diagram Label**. In the Properties view for the new label, set **Element Icon** to `false`, because you don't want an icon for the arguments. Set the **Figure** to `Label ArgLabelFigure`, which is the link to the low level/physical label from the gallery. Set the **Name** to `ArgLabel`.

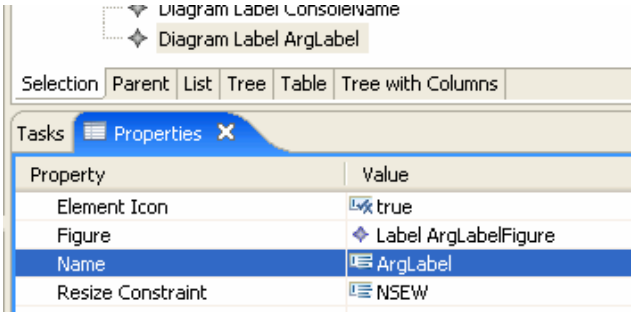


Figure 17-8: *Name is updated to ArgLabel*

5. Next, you need to define the compartment within the `Console` node which holds `Commands`. Right-click **Canvas Input** and select **New Child > Compartment**. Set **Collapsible** to `true` which means that the compartment can be collapsed and expanded. Set **Figure** to `Rectangle ConsoleFigure`, which is the figure node which will contain this compartment. Set the **Name** to `CommandCompartment`. Leave **Needs Title** set to `false`.

6. Likewise, add another compartment definition for the `Argument` compartment within the `Command` node. Right-click **Canvas Input** and select **New Child > Compartment**. Set **Collapsible** to `true`, **Figure** to `Rectangle ConsoleFigure`, and **Name** to `ArgCompartment`.

7. Save and close the GMFGraph editor.

Task 4: Create GMFTool

A GMFTool Model is a model file (with the extension GMFTool) which defines the tools that are available in the GMF editor. Tools include menus, context menus and the toolbar palette. The GMFTool wizard creates a default toolbar palette.

1. Within the `lab.console.input` project, right-click `model\Input.ecore` and select **New > Other**. Select the **GMFTool Simple Model** wizard and select **Next**. Note, do NOT select the **GMFTool Model**.

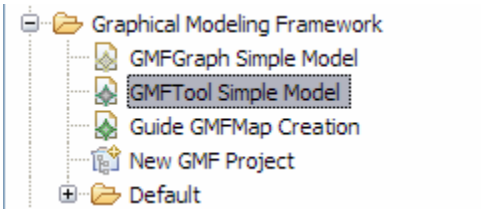


Figure 17-9: *Selecting GFMTool Simple Model*

2. A default filename of `Input.gmftool` should already be filled in, so click **Next**.

3. The `Input.ecore` file that you right-clicked should already be highlighted as the input Domain Model, so click **Next**.

4. Set the Tooling Definition page as shown below. In particular, the D**iagram element** should be set to `Root`. It is the element in the model that corresponds to the entire diagram. In the **Domain model elements to process** grid, the first checkbox column indicates which Classes in the model need Node tools. In this example, `Consoles`, `Commands` and `Args` need node tools. The second column indicates which classes and relationships need link tools. In this example, you aren't using any links, so none are selected. The final column is not actually used for defining Tools.

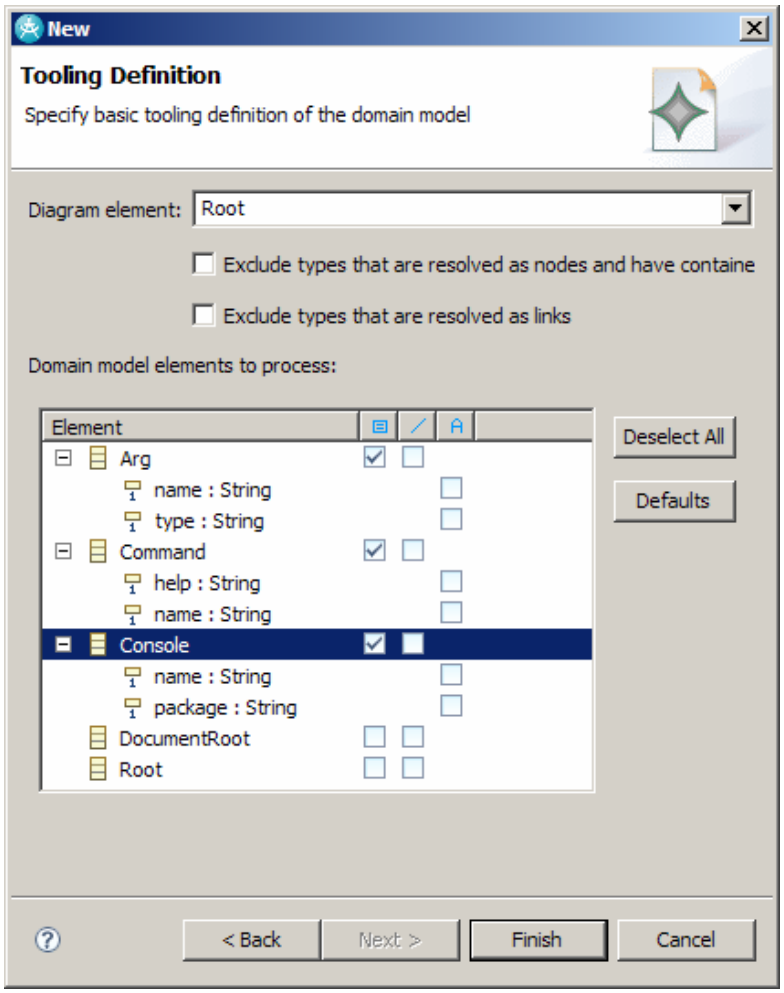


Figure 17-10: *Tooling Definition Wizard Settings*

5. Finally, click **Finish**. The new file `Input.gmfgraph` is created and opened.

Task 5: Create GMFMap

A GMFMap Model is a model file (with the extension GMFMap) which maps all the other GMF related files together. Specifically, it maps the graphical elements (from GMFGraph) to the corresponding domain data (ecore) and tools (GMFTool).

1. Within the `lab.console.input` project, right click `model\Input.ecore` and select **New** > **Other**. Select the `Guide GMFMap Creation` wizard and select **Next**. Note, do NOT select the `GMFMap Model`.

Figure 17-11: *Selecting Guide GMFMap Creation*

2. A default filename of `Input.gmfmap` should already be filled in, so click **Next**.

3. The names of the Domain Model, Graphical Definition and Tooling Definition files should already be filled in as shown below. Click the top right **Load** button (for the Domain Model), then the one below that (for the Graphical Definition) and then the last one (for the Tooling Definition) in order to load the various files into the wizard. Then click **Next**.

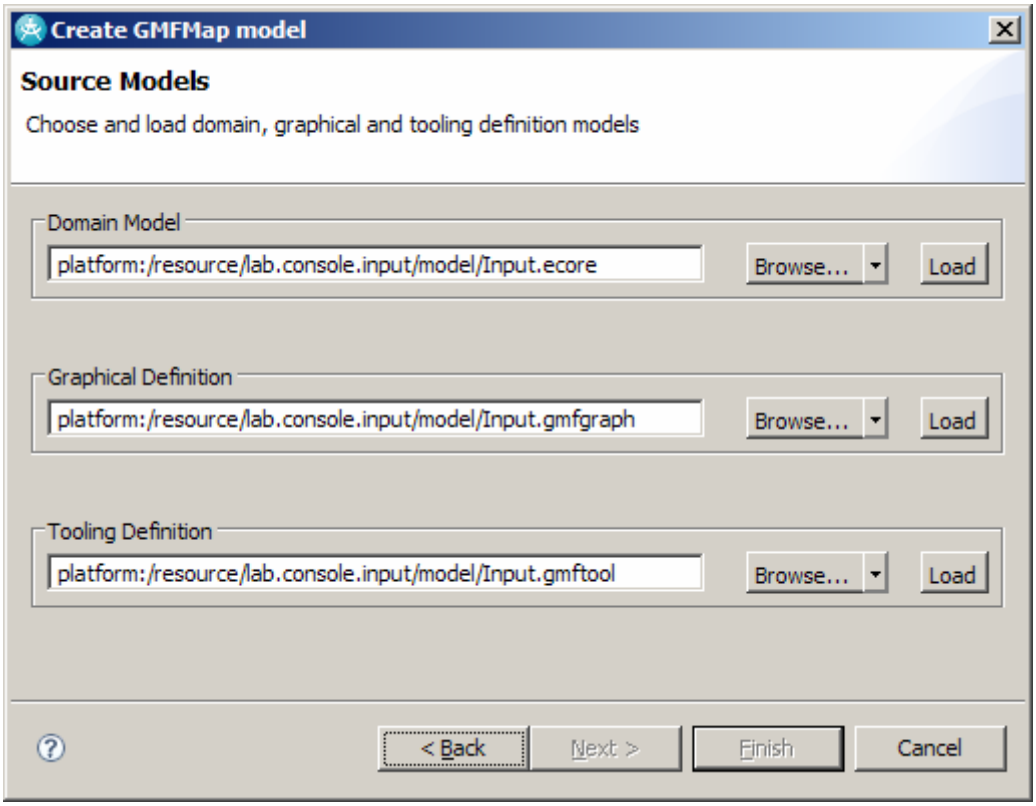


Figure 17-12: *The GMFMap creation wizard*

4. Specify that the **Diagram Root Element** is `Root` and click **Next**.

5. On the Mapping screen, you see a list of tentative Nodes and Links. To change a Node into a link, select it and click the **As Link** → button. To change a Link into a node, select it and click the **As Node** ← button. To remove an element, select in and click the **Remove** button. In this case, you only want to see the root level nodes. Specifically, select the extra **Links** and **Nodes** and remove them from the lists by clicking on **Remove**. As a result, `Console` is the only **Node** and there are no **Links**.

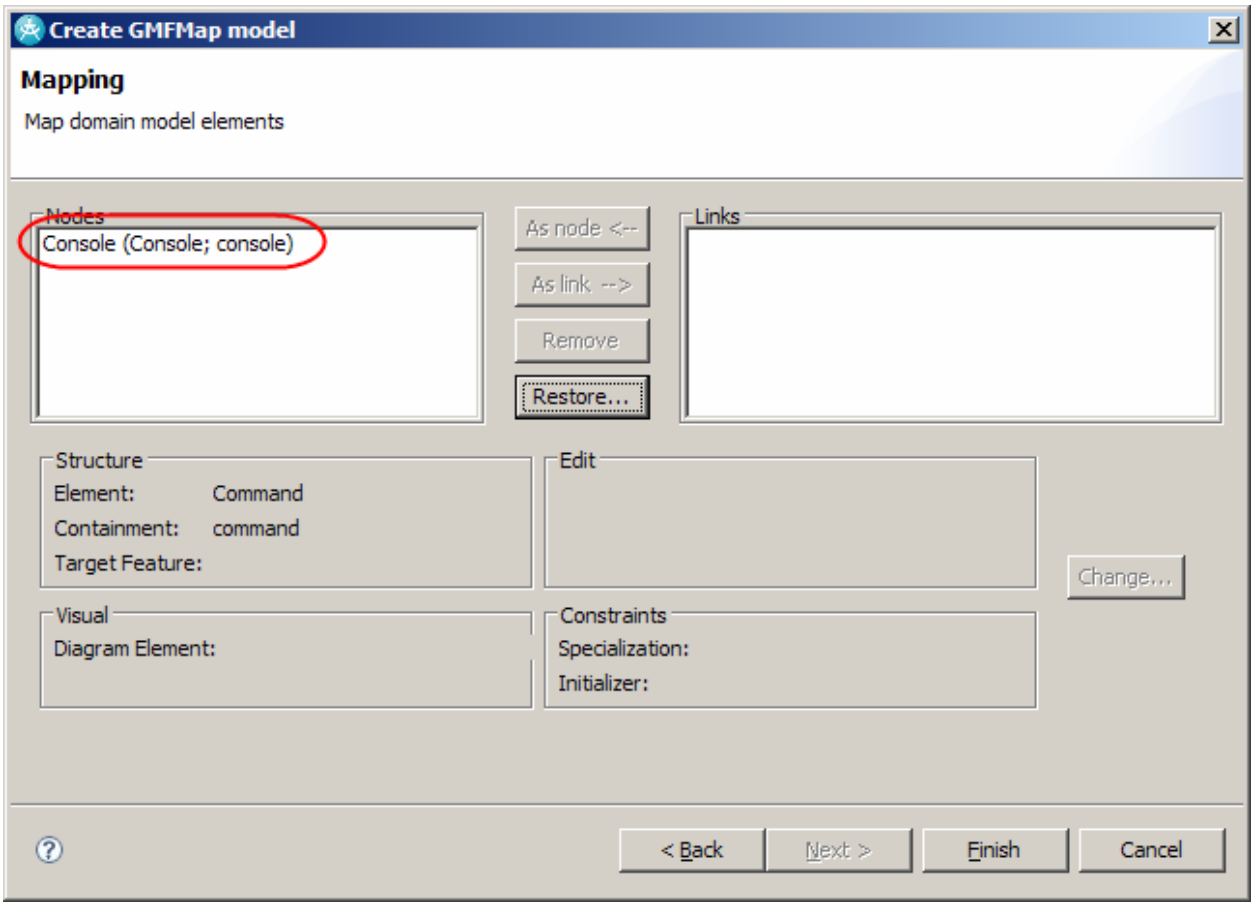


Figure 17-13: *When specifying the Mapping, ensure that console is the only node*

6. Click **Finish**. The new file `Input.gmfmap` is created and opened.

Task 6: Refine the Generated GMFMap

GMFGraph ties together the graphical elements, tooling elements and domain model elements together. In particular, it is the final definition of the nodes, link, labels and compartments. In addition, the graphical compartments are defined in the GMFGraph, but GMFMap defines the hierarchical structure of the compartments.

1. Make sure that the file `Input.gmfmap` is open.
2. Expand **Input.gmfmap**, then **Mapping**, then **Top Node Reference** so that you can see and select the Console **Node Mapping**.

TIP: This defines a node for Consoles linking it to the graphical, domain, and tooling definitions.

3. Review the **Properties** view for the `Console` **Node Mapping** and make sure that the **Domain meta information > Element** is `EClass Console`, the **Visual representation > Diagram Node** is set to `Node Console` and that the **Visual representation > Tool** is set to `Creation Tool Console`.

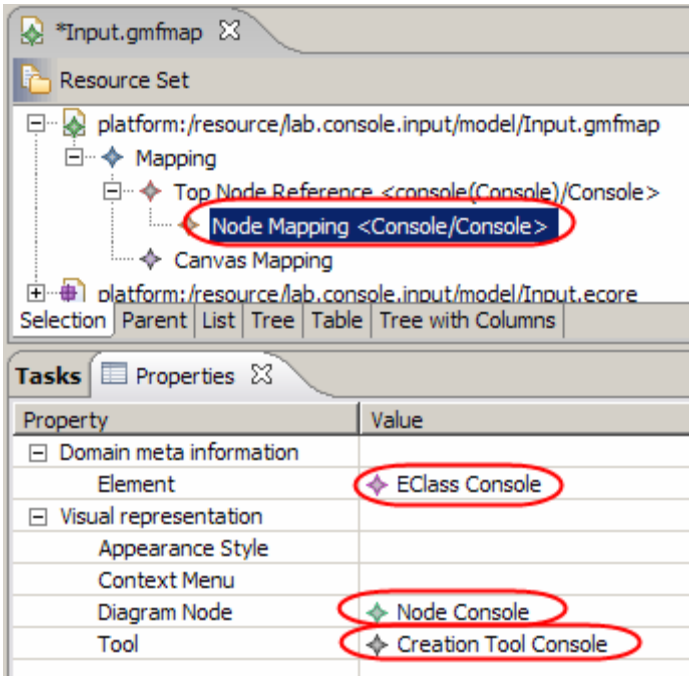


Figure 17-14: *Ensure that the properties are set as shown*

4. Next, you need to add the label for the Console Node. Right-click the `Console's Node Mapping` and select **New Child > Label Mapping**. In the Properties, set the **Diagram Label** to `Diagram Label ConsoleName` and set the **Features** to `EAttribute name` (using the popup dialog box from pressing '' button).

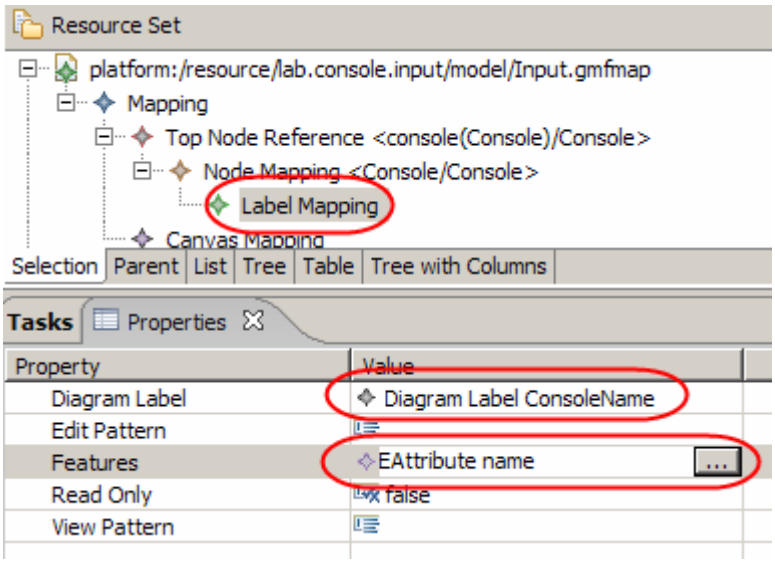


Figure 17-15: *The updated set of properties*

5. Next, you need to add the compartment to the Console node. Right-click the `Console's Node Mapping` and select **New Child > Compartment Mapping**. In the Properties, set **Visual representation > Compartment** to `Compartment CommandCompartment`.

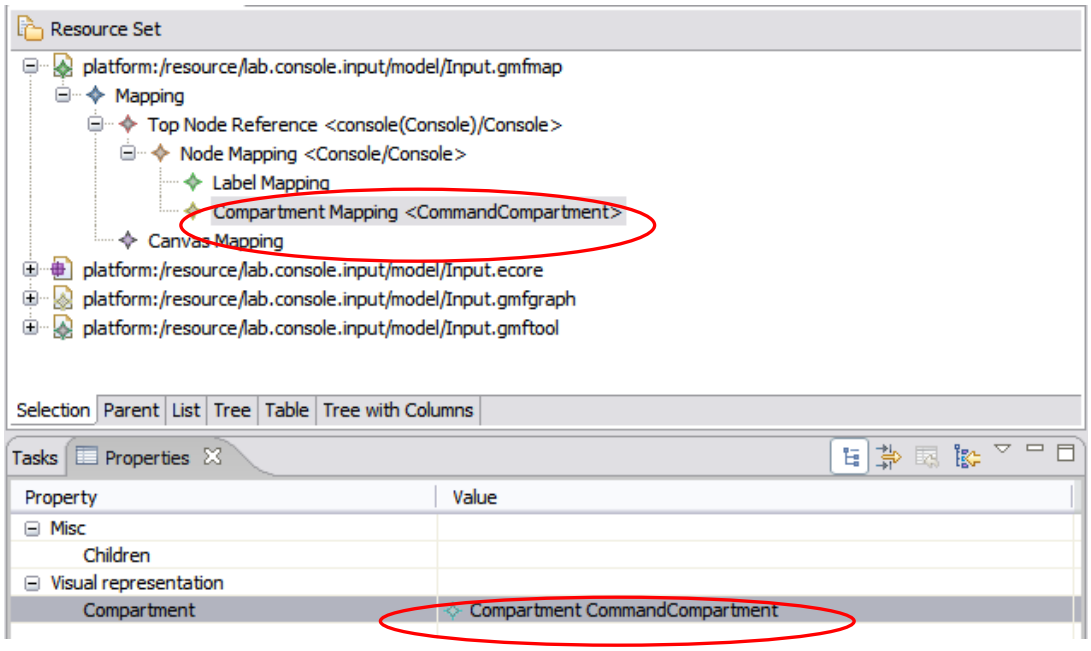


Figure 17-16: *The updated set of properties*

6. Next, add the nodes that can appear within the **CommandCompartment**. Right-click the `Console's Node Mapping` and select **New Child > Child Reference**. In the **Properties** view, set **Compartment** to `Compartment Mapping <CommandCompartment>`, which indicates which compartment this new child is in (the Command Compartment). Set **Containment Feature** to `EReference command`, which is the `Input.ecore` defined containment element of Console's which contain the nested (Command) elements.

7. To complete the nested Command node definition, right-click the new `Child Reference` and select **New Child > Node Mapping**. In the Properties, set **Domain meta information > Element** to `EClass Command`, **Diagram Node** to `Node Command`, and **Tool** to `Creation Tool Command`.

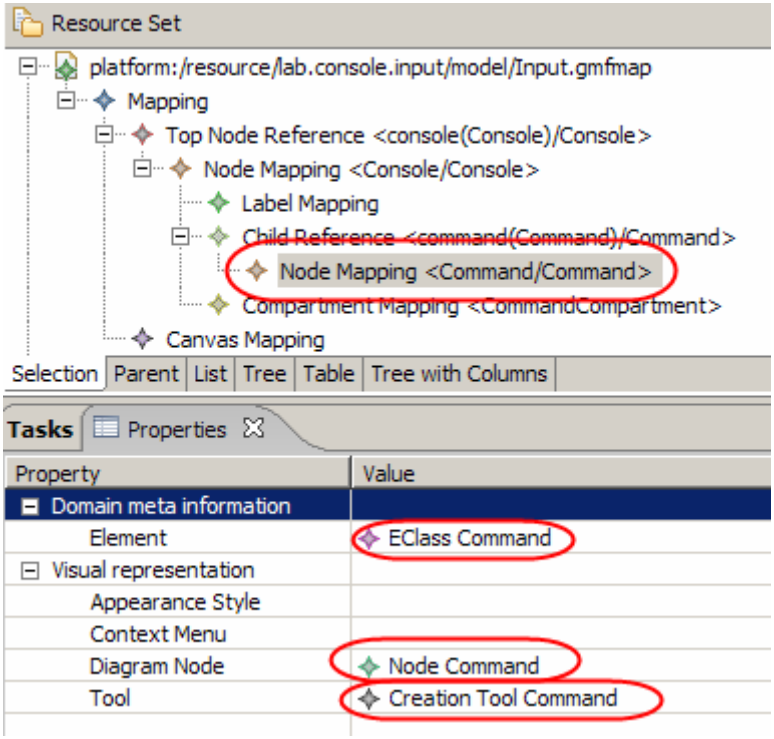


Figure 17-17: *The updated set of properties*

8. Add a **Label Mapping** to the **Node Mapping <Command/Command>** node by right-clicking it and adding a child **Label Mapping**. Set its **Diagram Label** to `Diagram Label CommandName` and **Features** to `EAttribute Name`.

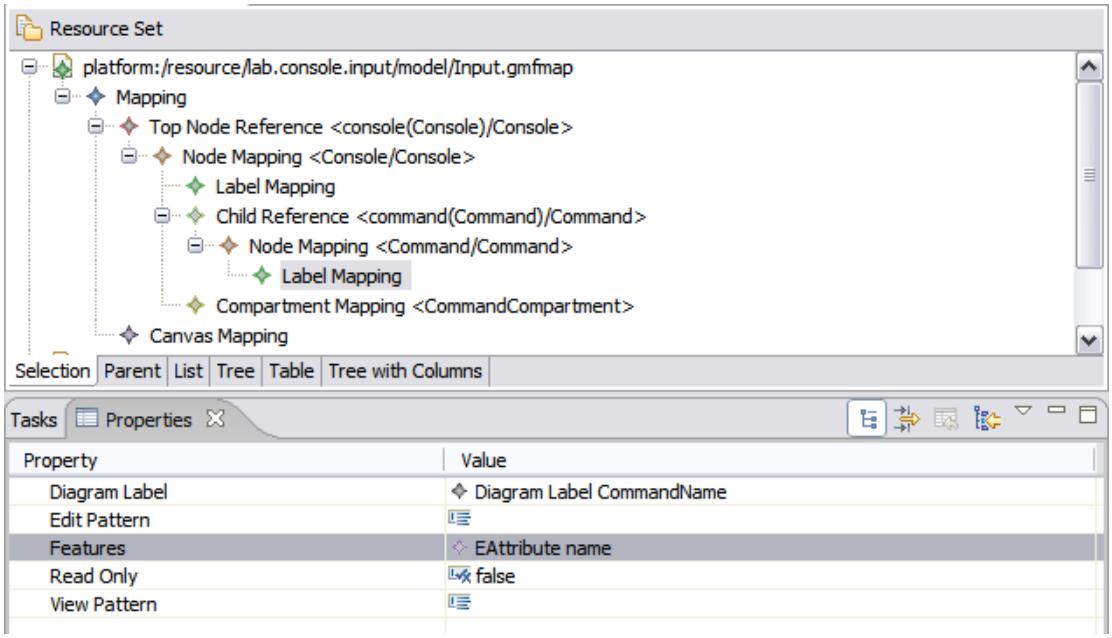


Figure 17-18: *The updated set of properties*

9. Add a **Compartment Mapping** child to the **Node Mapping <Command/Command>** Node setting its **Compartment** property to `Compartment ArgCompartment`.

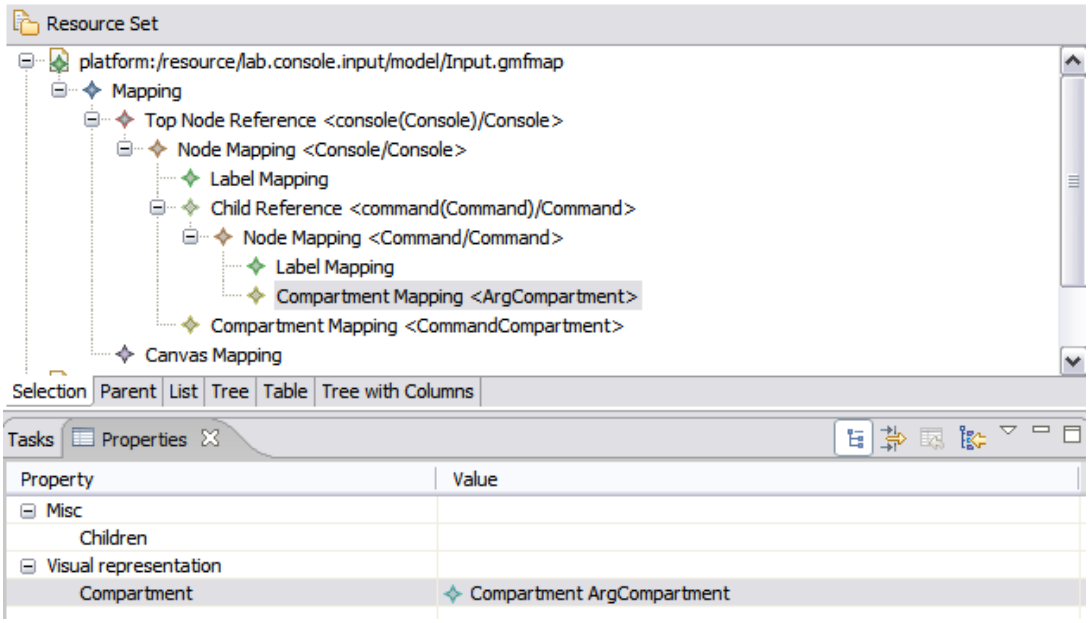


Figure 17-19: *The updated set of properties*

10. Add a new 'Child Reference' child to the **Node Mapping <Command/Command>** Node. For its properties, set the **Compartment** to `Compartment Mapping <ArgCompartment>` and set the **Containment Feature** to `EReference arg`.

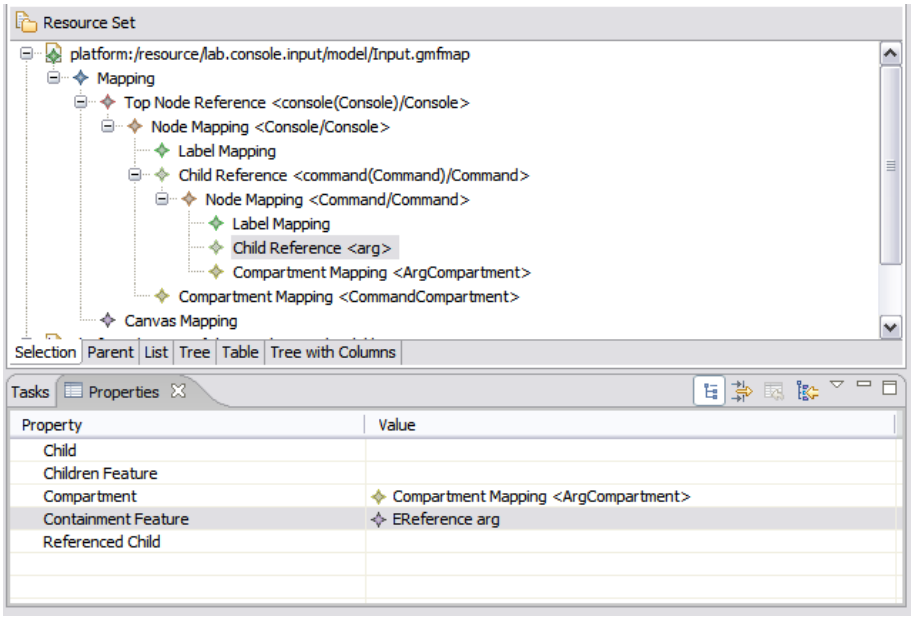


Figure 17-20: *The updated set of properties*

11. Right-click the new `Child Reference <arg>` and select **New Child > Node Mapping**. For the properties, set **Element** to `EClass Arg`, **Diagram Node** to `Diagram Label ArgLabel`, and **Tool** to `Creation Tool Arg`. Note how you set the Diagram Node to a label instead of a node.

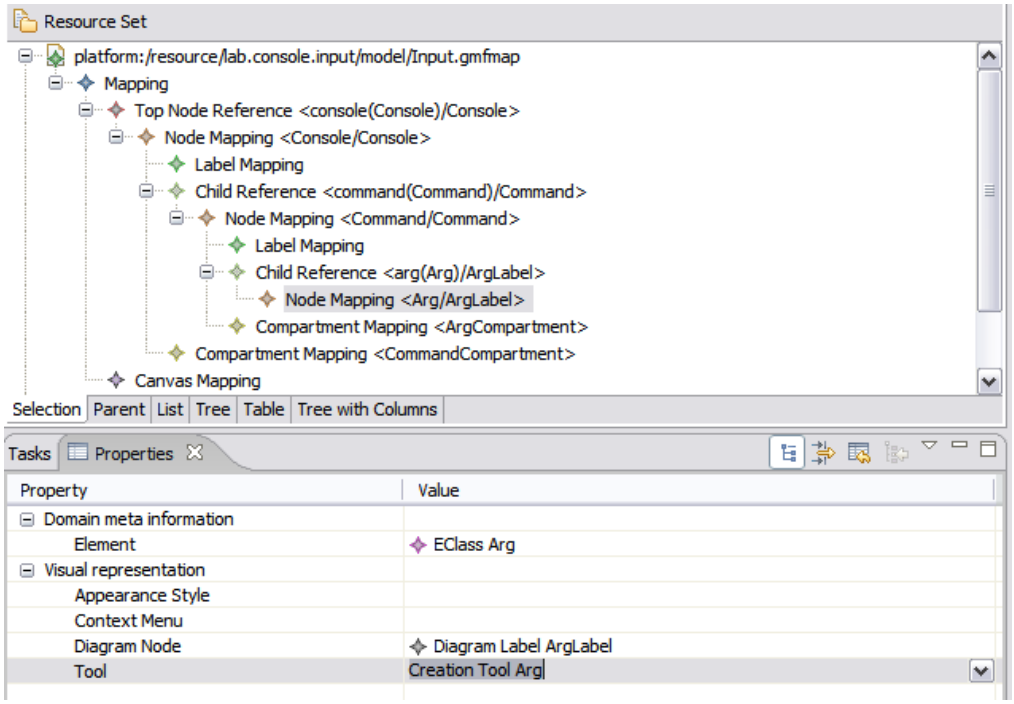


Figure 17-21: *The updated set of properties*

12. Add a **Label Mapping** child to the new **Node Mapping <Arg/ArgLabel>** node. This label will be structured differently then previously defined labels, because you want to show the name and the data type of the argument in the label, such as "arg0:String'. For its properties, set the 'Diagram Label' to '**Diagram Label ArgLabel**'. Set the Features to '**EAttribute name**' AND '**EAttribute type**' (in that order). Set the 'View Pattern' to '**{0}:{1}**' and set the 'Edit Pattern to '**{0}:{1}**'. In the edit and view patterns, any instance of {0} represents the first feature (which is name), any instance of {1} represents the second feature (which is type) and so on.

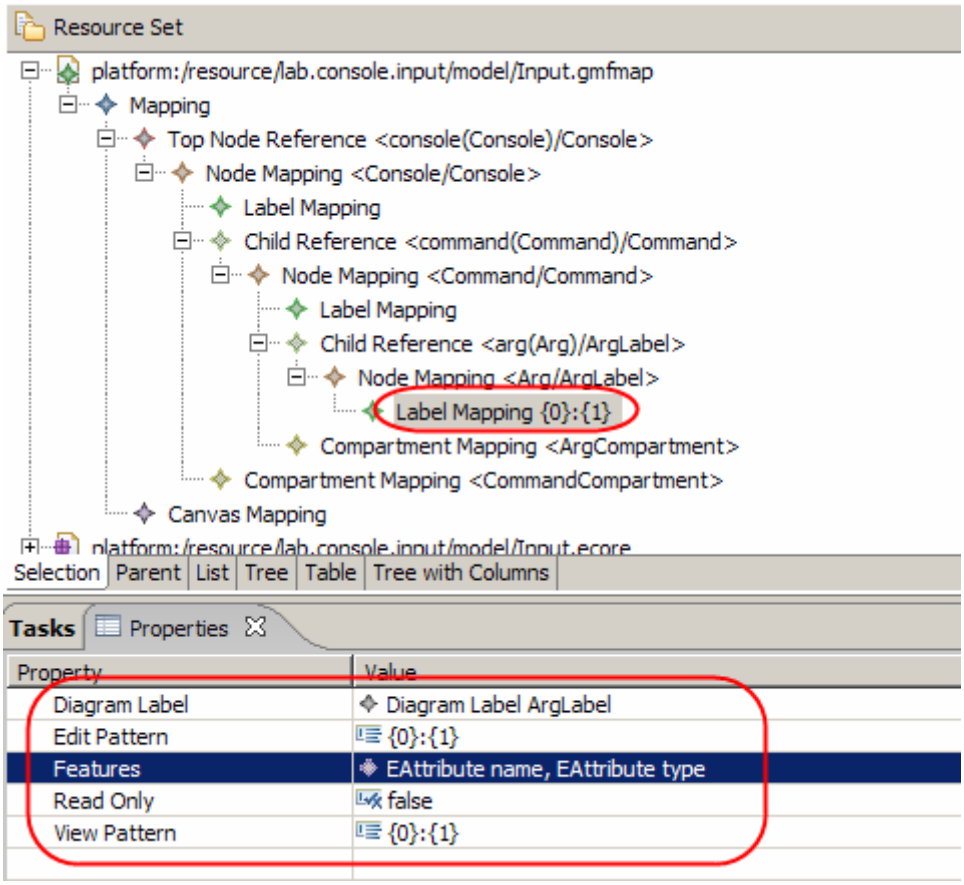


Figure 17-22: *The updated set of properties*

13. All of the nodes, labels, and compartments are defined.

14. Select **File > Save All**.

15. Close the `Input.GMFMap`.

Task 7: Create GMFGen

The GMFGen file contains code generation settings for the various GMF files.

1. Within the `lab.console.input` project, right-click `model\Input.gmfmap` and select **Create generator model**. Accept the default name of `Input.gmfgen` and click **OK**.

2. If you are prompted for the location of the genmodel file, select the `input.genmodel` file from the project.

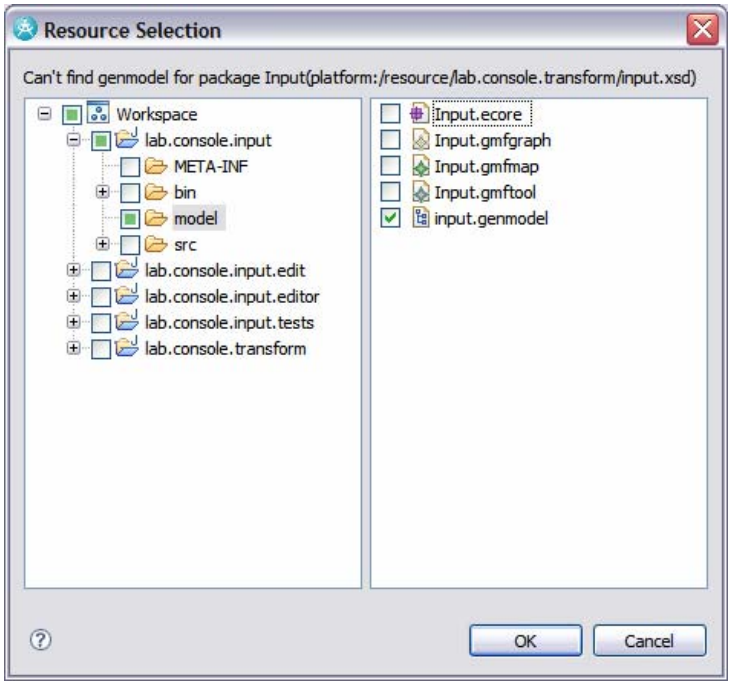


Figure 17-23: *Specifying the genmodel file*

3. If you are prompted to use `IMapMode`, select **Yes**.
4. A new file named `Input.gmfgen` should be created in the model directory.

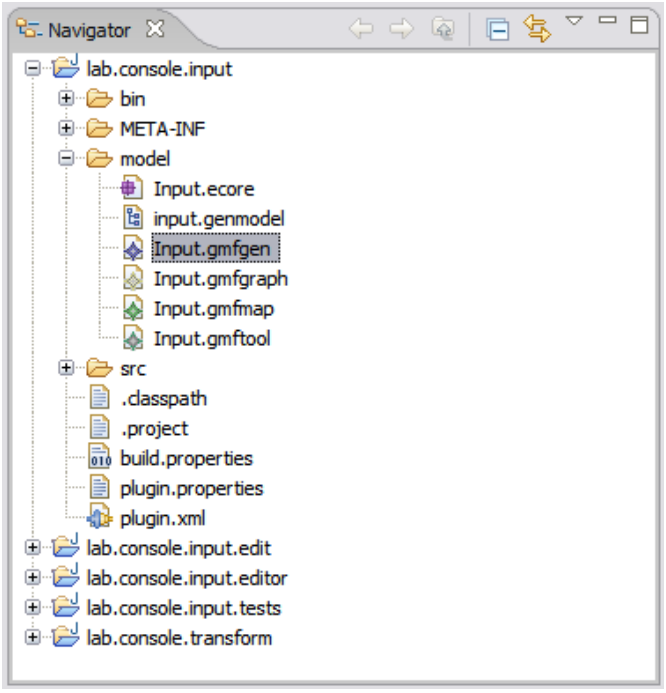


Figure 17-24: *Update view of the files in the project*

Task 8: Refine the Generated GMFGen

By modifying `Input.gmfgen` you can change some of the behavior in the generated diagram editor.

1. Open `lab.console.input/model/Input.gmfgen`.
2. The default generated editor does NOT enable diagram printing. In order to enable diagram printing, expand and find **Gen Editor Generator lab.console.Input.diagram / GenDiagram RootEditPart / Gen Plugin Input Plugin**. In the Properties view for the **Gen Plugin Input Plugin**, set **Printing Enabled** to `true`.

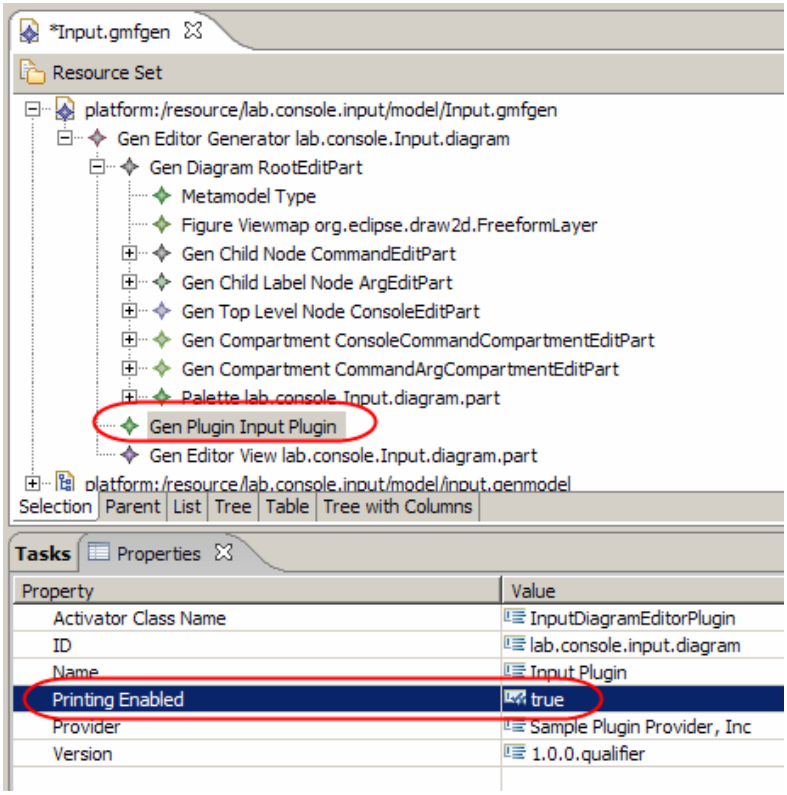


Figure 17-25: *Ensure that Printing Enabled is set to true*

3. Compartments in the generated diagrams can use a **List Layout** style or a **Freeform Layout**. If it is **List Layout** style, then the child elements are displayed in a vertical list. In **Freeform Layout** style, the user can position the child nodes anywhere in the compartment. You want **Freeform Layout** for the Command Compartment and **List Layout** style for the Arg Compartment. Find the **Gen Compartment** entries as illustrated before. Make sure that **List Layout** is `false` for **Gen Compartment ConsoleCommandCompartmentEditPart** and `true` for **Gen Compartment CommandArgCompartmentEditPart**.

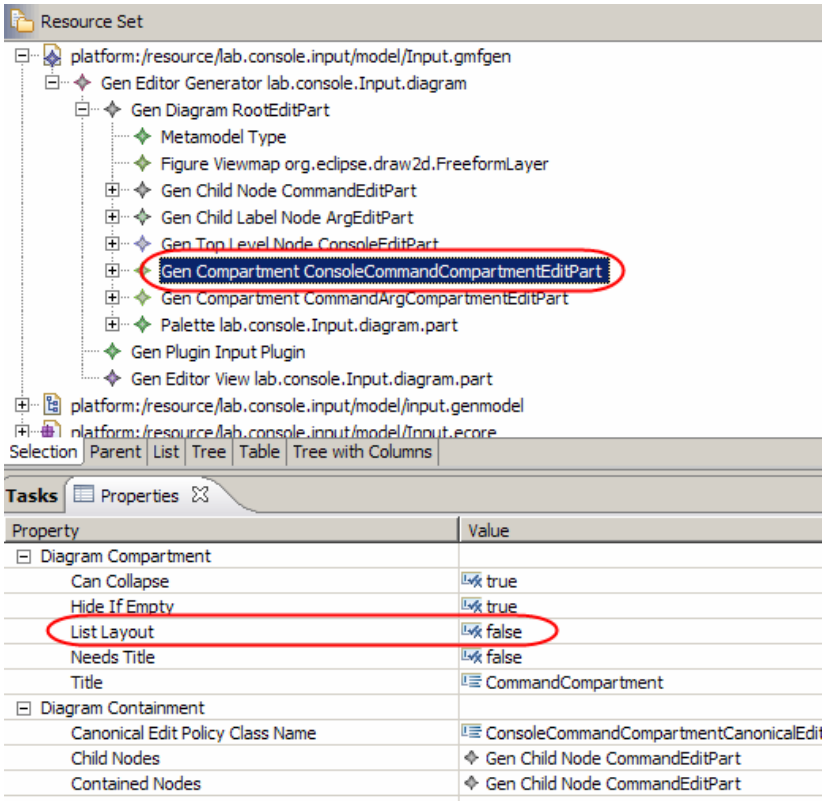


Figure 17-26: *Set List Layout to false*

4. Select **File > Save All**.
5. Close the `Input.gmfgen` file.

Task 9: Generate the Graphical Editor

All the pieces are finally in place to generate the Graphical Editor's plugin and source code.

1. Within the `lab.console.input` project, right-click `model\Input.gmfgen` and select **Generate diagram code**. A plug-in project named `lab.console.input.diagram` should be created/updated. It contains the graphical editor.
2. Click **OK**.

Task 10: Refine the generated code

The code generated by the GMF generator is designed to work with base Eclipse. Rational Software Architect leverages and extends the capabilities of basic GMF. In more advanced scenarios, it is possible to leverage the additional power and capabilities of Rational Software Architect in your GMF based diagrams. However, there is one minor incompatibility in using basic GMF-generated editors with Rational Software Architect, which is very easily corrected.

Note that if you do not do this task, all text labels that are in the resulting editor will not work correctly. For example, if you attempt to change the name of a Console in the editor, it will fail to change and give an error message.

1. In the generated diagram editor plugin, which is `lab.console.input.diagram`, open up `plugin.xml`. Select the `plugin.xml` tab to view the source code for `plugin.xml`. Search for the string 'parserProviders'. Change the nested element that says `<Priority name="Lowest"/>` to `<Priority name="Low"/>`. In other words, change the priority from Lowest to Low. Then save and close `plugin.xml`.

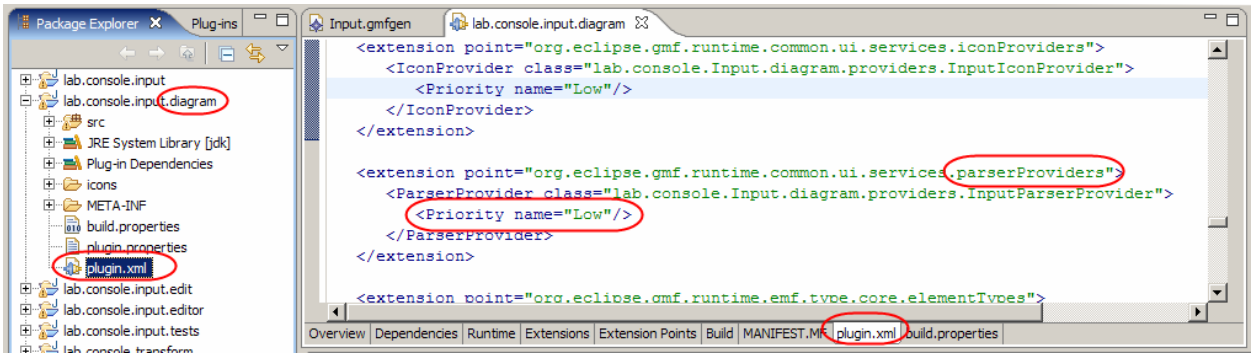


Figure 17-27: *The updating plugin.xml file*

Task 11: Test the Generated Editor

In this task, you will test the generated editor.

1. In Navigator or Package Explorer, right-click the project named `lab.console.input.diagram` and select **Run As > Eclipse Application**. Then wait for a new instance of Rational Software Architect to launch.
2. In the run-time workbench, create a simple project named **console.diagram.test**.

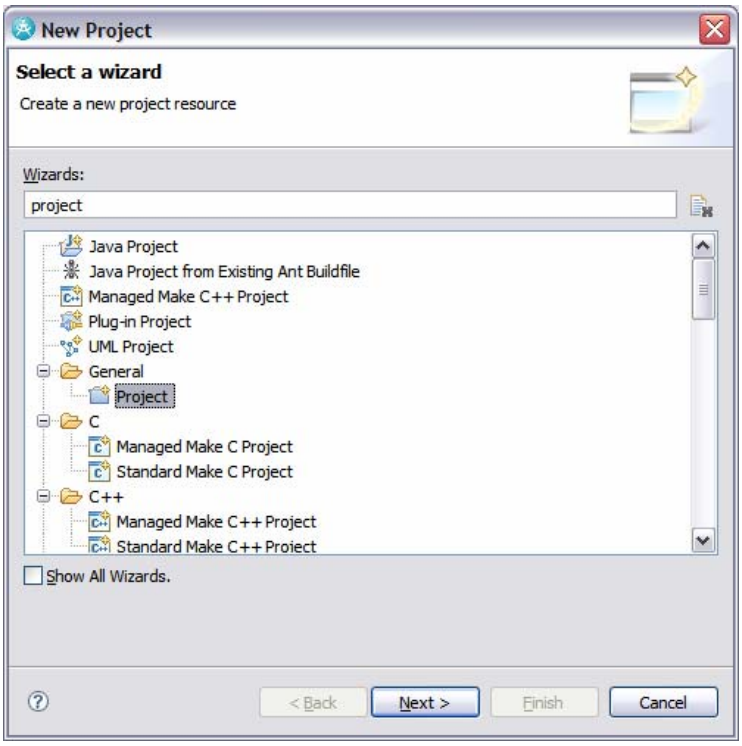


Figure 17-28: *Creating a simple Project*

3. Right-click the new project name and select **New > Other**. Select the Input Diagram wizard and then click **Next**.
4. Accept the default of `default` and click **Finish**.
5. `default.input_diagram` should be opened in an editor that looks like the following.

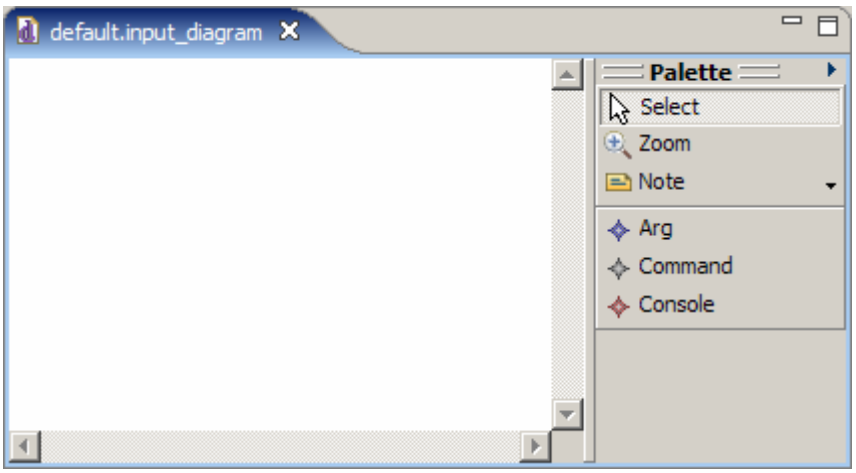


Figure 17-29: *The default.input_diagram in the editor*

6. To add a new Console, click **Console** in the **Palette** and then click the drawing surface. Name the new Console `MyConsole`. Open up the properties for `MyConsole` and set the **Package** to `my.console`.

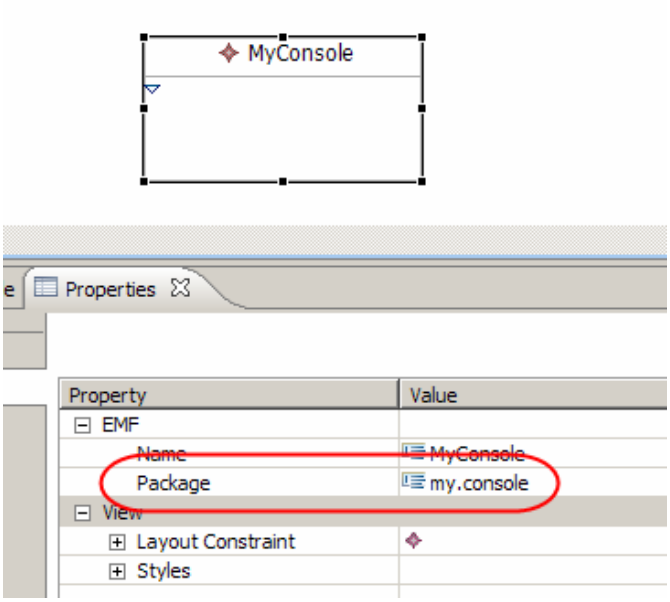


Figure 17-30: *Specifying the Package for the Console*

7. In the Diagram editor, expand the node for `MyConsole` so that there is room to work within the compartment.

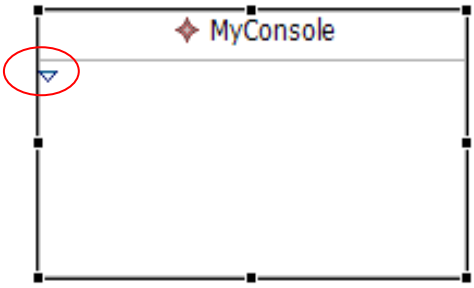


Figure 17-31: *Expand the compartment within MyConsole*

8. To add a child Command, click **Command** in the **Palette** and then click in the compartment in `MyConsole`. Name the Command `echo`.

9. Click **Arg** in the Palette and then in the compartment inside of echo Command to add an argument. Give the new Arg a label of `text:String`. Open the **Properties** of the Arg and you should see that the **Name** is `text` and the **Type** is `String`.

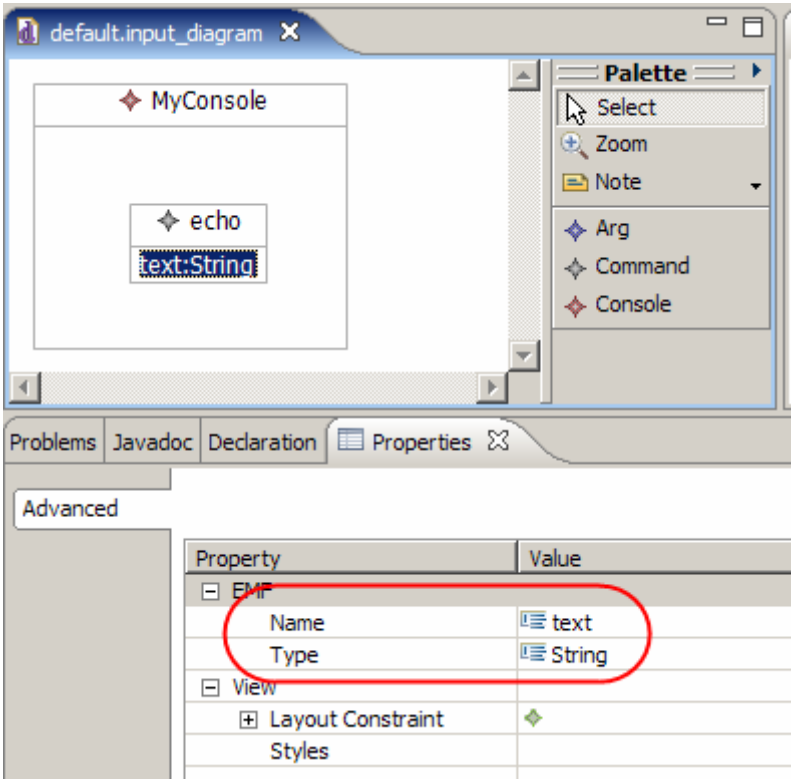


Figure 17-32: *Viewing the properties of the Arg element.*

10. Add any other **Consoles**, **Commands**, and **Args** that you want.
11. In order to test the transformation, save and close your diagram.
12. It's easier to test the existing transformation if the file has an XML extension, so rename `default.input` to `default.input.xml`.
13. Right-click `default.input.xml` and select **Run As > Input for JET Transformation**. In the Properties page that appears, select `lab.console.transform` as the **ID**. Then click **OK** to run the transformation.

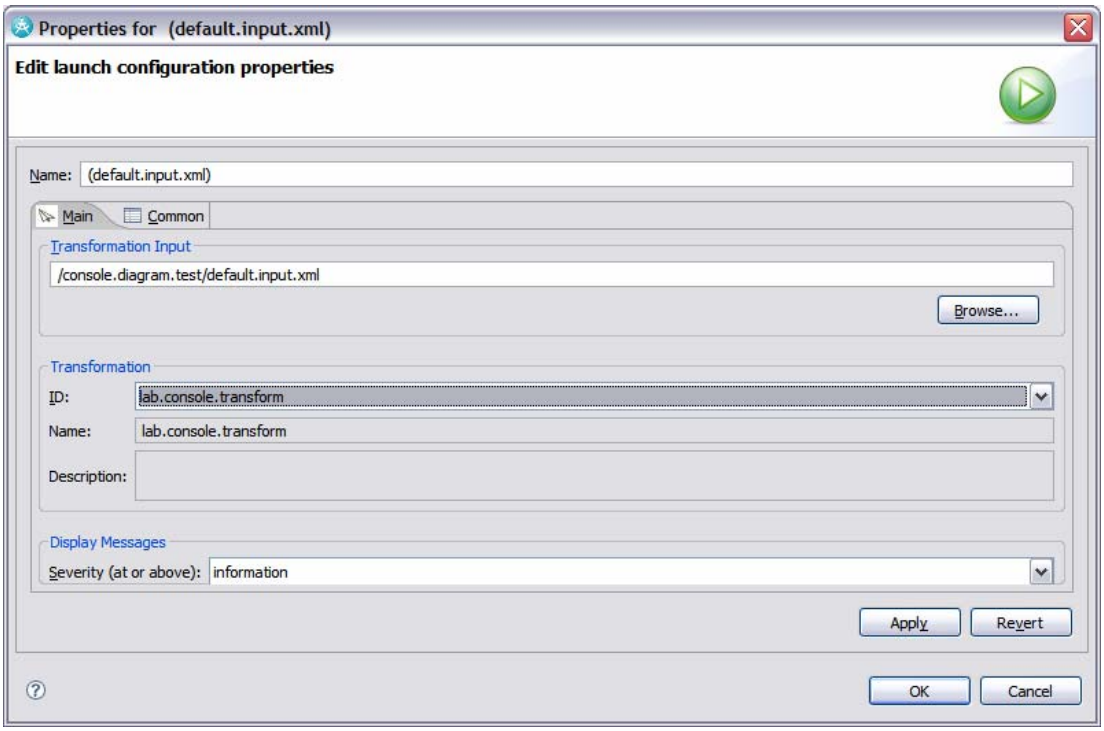


Figure 17-33: *Selecting the transformation to run.*

14. The project `MyConsole Console` (and any other consoles that you defined) are generated.

15. Close the second instance of Rational Software Architect when you are done testing.