



## Eclipse Transformation Training Exercise Workbook #1

# 1 Plug-in Exercises

## 1.1 What this exercise is about

In this exercise you will extend the Workbench

At the end of this exercise you should be able to

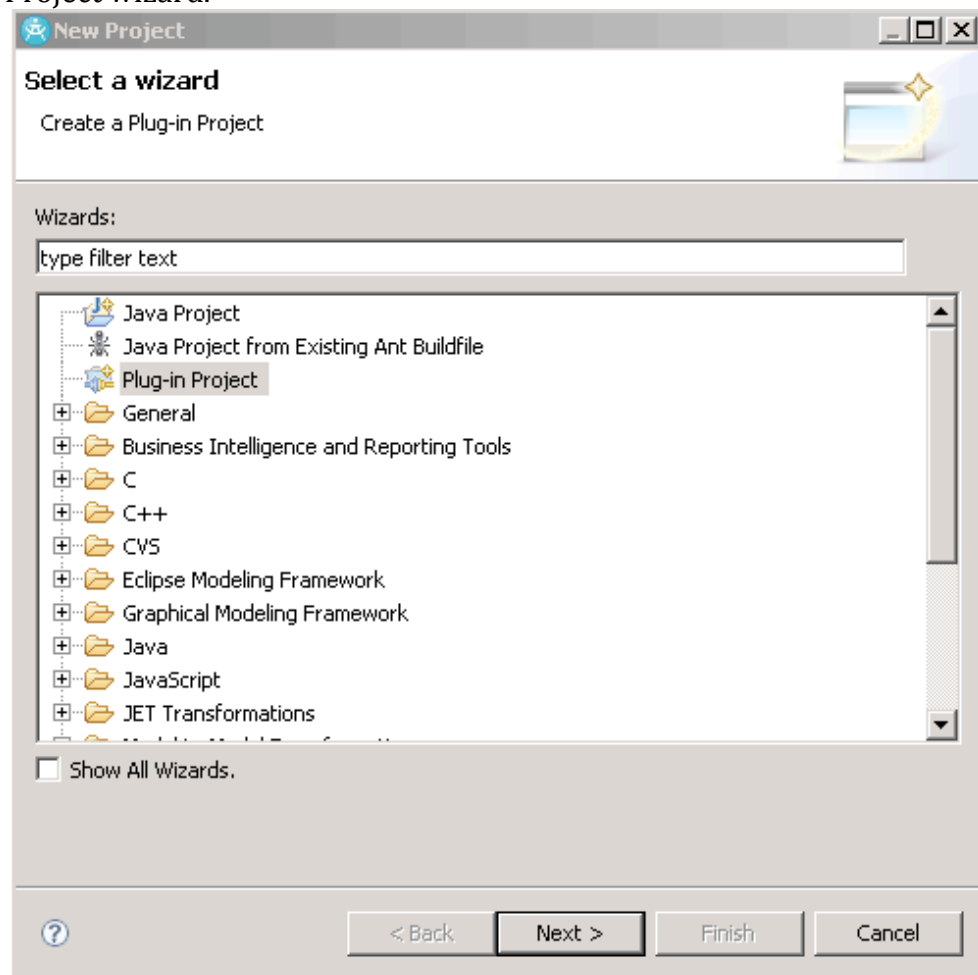
- Create Eclipse Plug-in Projects
- Add menus to the Workbench

## 1.2 Create a new project

The first step in creating our plug-in is to create a plug-in project using the Eclipse New Project wizard. We will create an empty Eclipse Plug-in project that contributes to the Workbench UI.

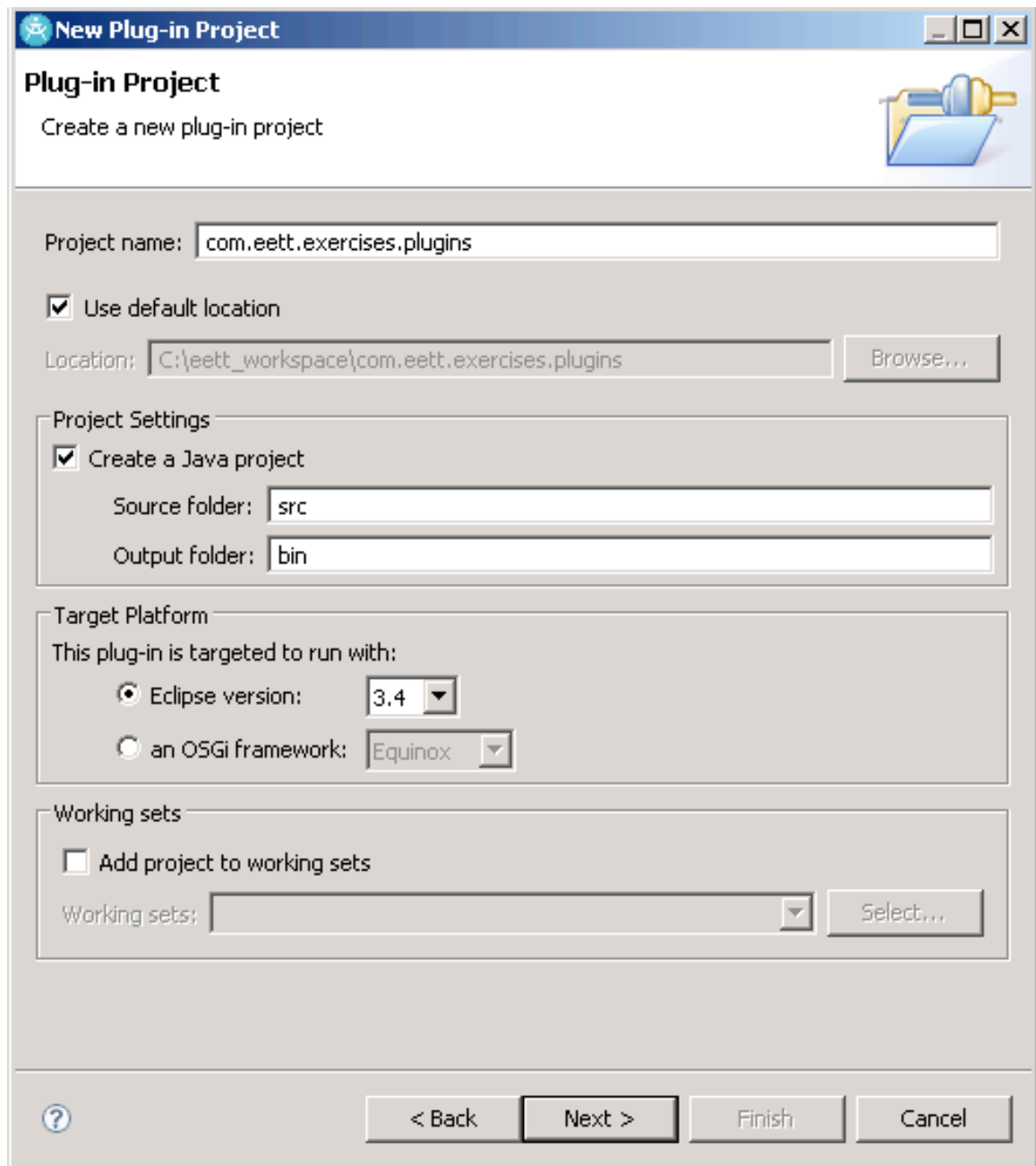
From the File menu New → Project...

Select Plug-in Project from the list of possible projects to create in the New Project wizard.



Click the Next button.

Enter the name of the project; com.eett.exercises.plugins. Keep Use default location and Create a Java project checked. Retain the defaults for the other properties. Click on Next >.



**New Plug-in Project**

**Plug-in Project**  
Create a new plug-in project

Project name:

☒ Use default location

Location:

**Project Settings**

☒ Create a Java project

Source folder:

Output folder:

**Target Platform**  
This plug-in is targeted to run with:

☒ Eclipse version:  ▼

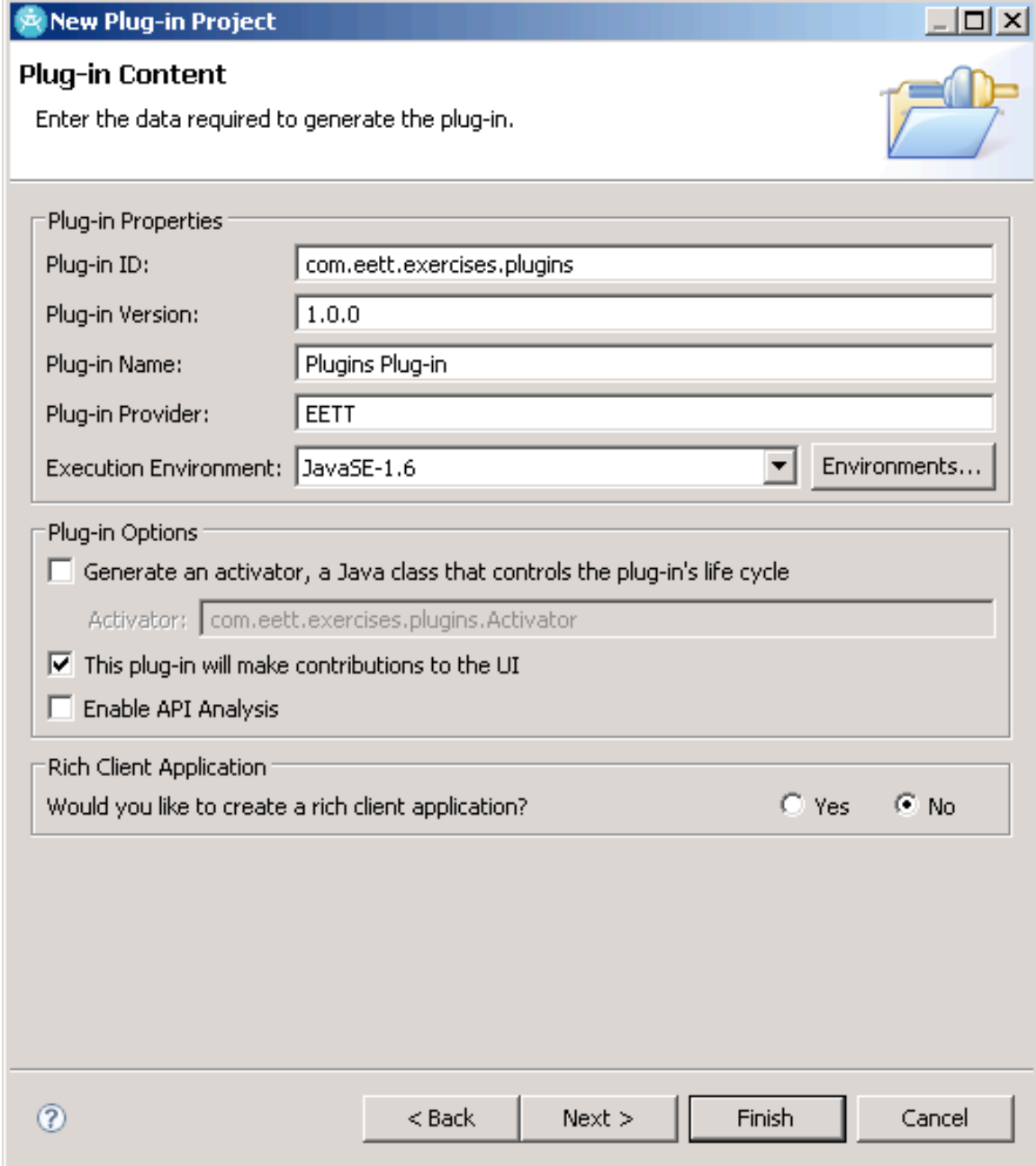
☐ an OSGi framework:  ▼

**Working sets**

☐ Add project to working sets

Working sets:  ▼

Specify the Plug-in Content. Keep the Plug-in ID, Plug-in Version, Plug-in Provider and Execution Environment defaults. Change the Plug-in Name to EETT Plug-in Exercises. Uncheck Generate an activator and make sure that This plug-in will make contributions to the UI is checked. Click the Next > button.



The screenshot shows the 'New Plug-in Project' dialog box with the 'Plug-in Content' tab selected. The dialog has a title bar with a question mark icon and standard window controls. The main area is divided into three sections: 'Plug-in Properties', 'Plug-in Options', and 'Rich Client Application'. In the 'Plug-in Properties' section, the 'Plug-in ID' is 'com.eett.exercises.plugins', 'Plug-in Version' is '1.0.0', 'Plug-in Name' is 'Plugins Plug-in', 'Plug-in Provider' is 'EETT', and 'Execution Environment' is 'JavaSE-1.6'. In the 'Plug-in Options' section, the checkbox 'Generate an activator, a Java class that controls the plug-in's life cycle' is unchecked, and the 'Activator' field contains 'com.eett.exercises.plugins.Activator'. The checkbox 'This plug-in will make contributions to the UI' is checked, and 'Enable API Analysis' is unchecked. In the 'Rich Client Application' section, the question 'Would you like to create a rich client application?' has 'No' selected. At the bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

**New Plug-in Project**

**Plug-in Content**  
Enter the data required to generate the plug-in.

**Plug-in Properties**

Plug-in ID:

Plug-in Version:

Plug-in Name:

Plug-in Provider:

Execution Environment:

**Plug-in Options**

☐ Generate an activator, a Java class that controls the plug-in's life cycle

Activator:

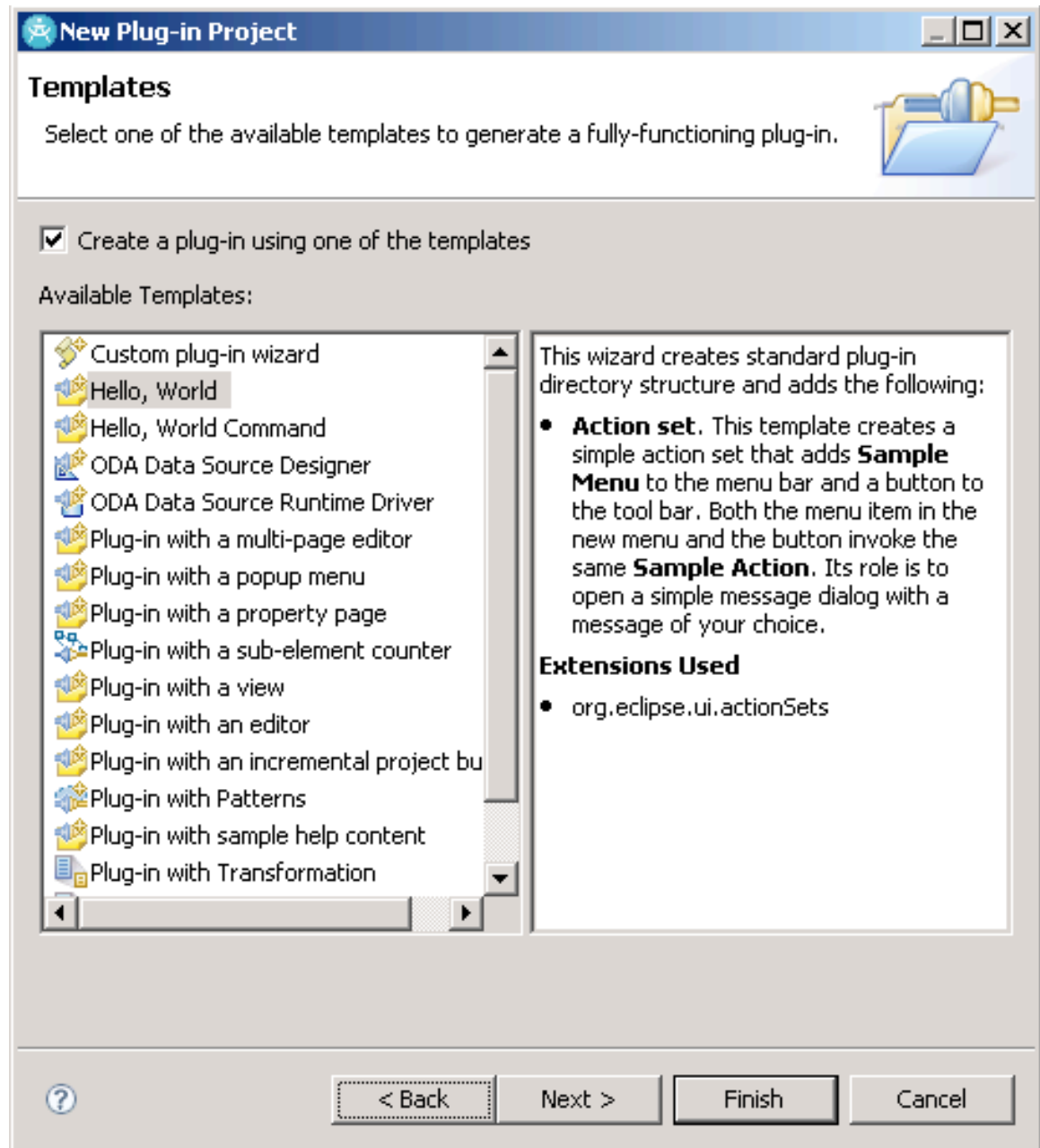
☒ This plug-in will make contributions to the UI

☐ Enable API Analysis

**Rich Client Application**

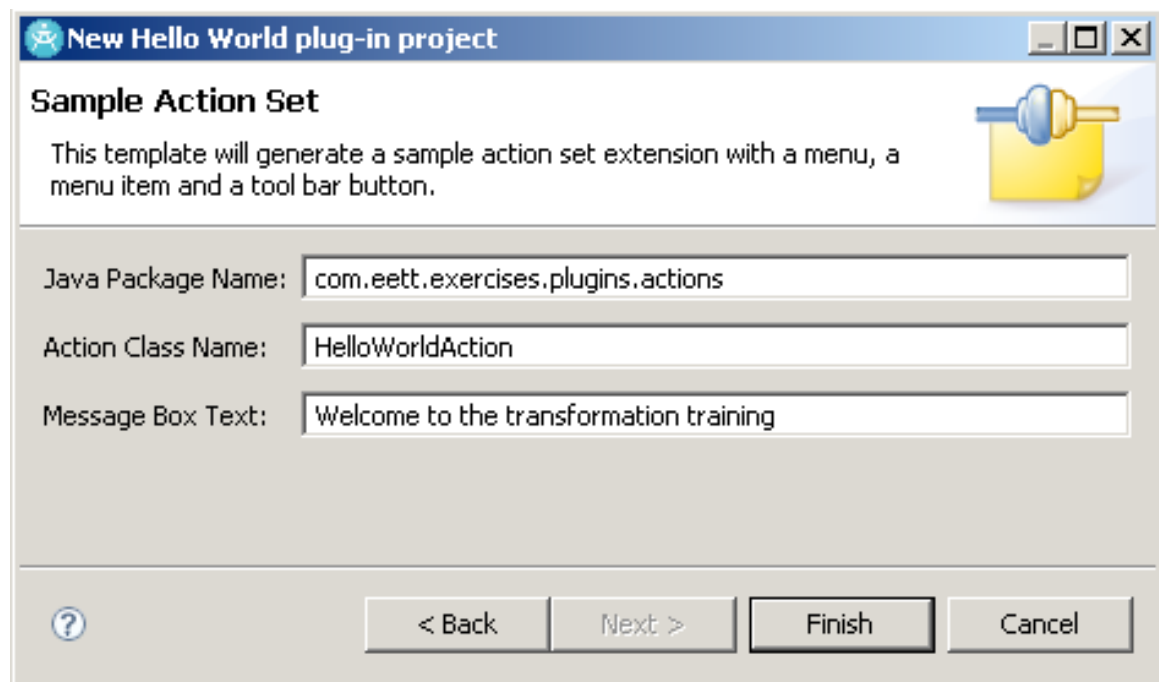
Would you like to create a rich client application? ☐ Yes ☒ No

On the Templates page check the 'Create a plug-in using one of the templates' and select the 'Hello, World' option. Click Next >.



By using a Template most of the code necessary to create a menu and an Action to handle selecting the menu is generated for you.

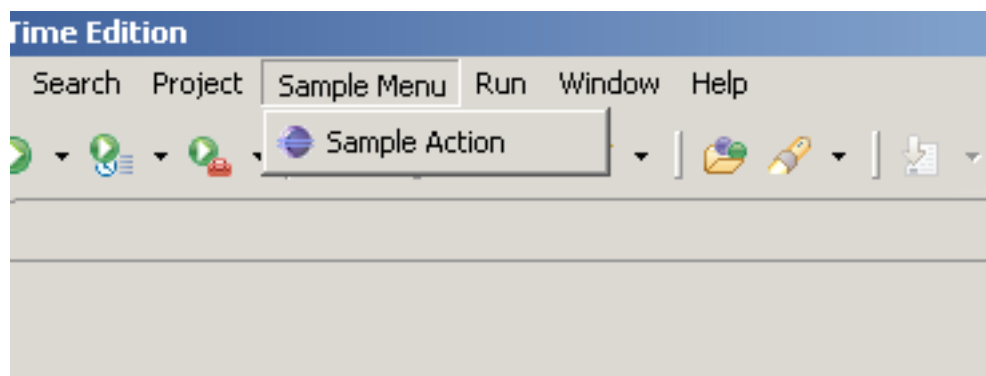
In the Sample Action Set page you can configure the code that will be generated. Change the Action Class Name to HelloWorldAction and the Message Box Text to Welcome to the transformation training. Click Finish.

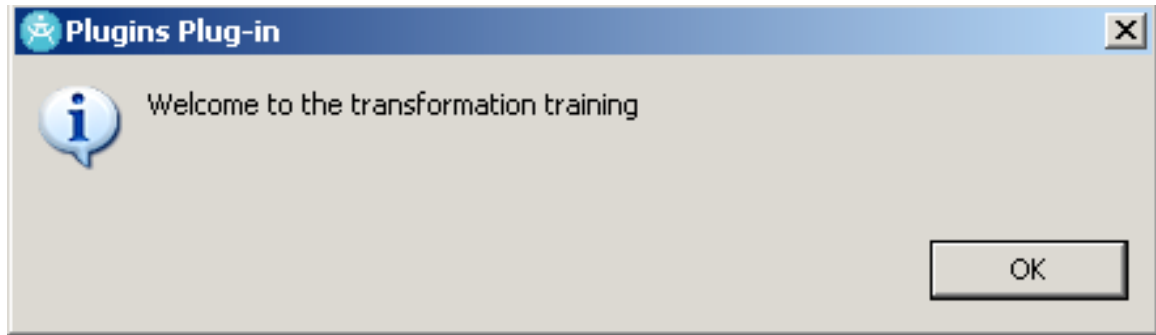


When prompted to switch to the Plug-in Development perspective click Yes.

### 1.3 Test the plug-in

At this point we can launch a Runtime Workbench and see our plug-in in action. To view the default that is generated select Run As → Eclipse Application from the project's context menu.



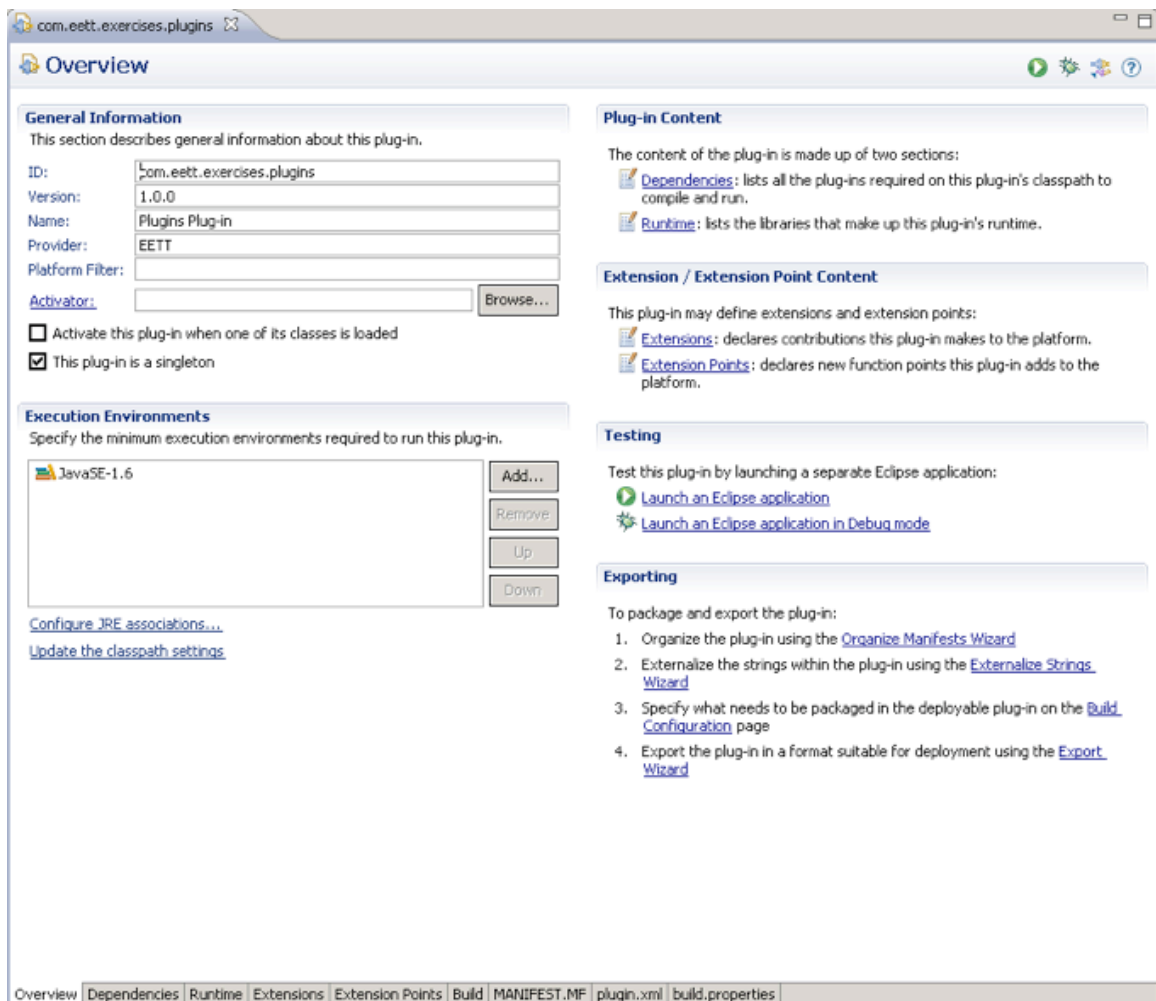


*This is an Eclipse version of the classic Hello World application. We will explore in more detail and augment the default that was created for us.*

#### 1.4 Define the plug-in manifest files

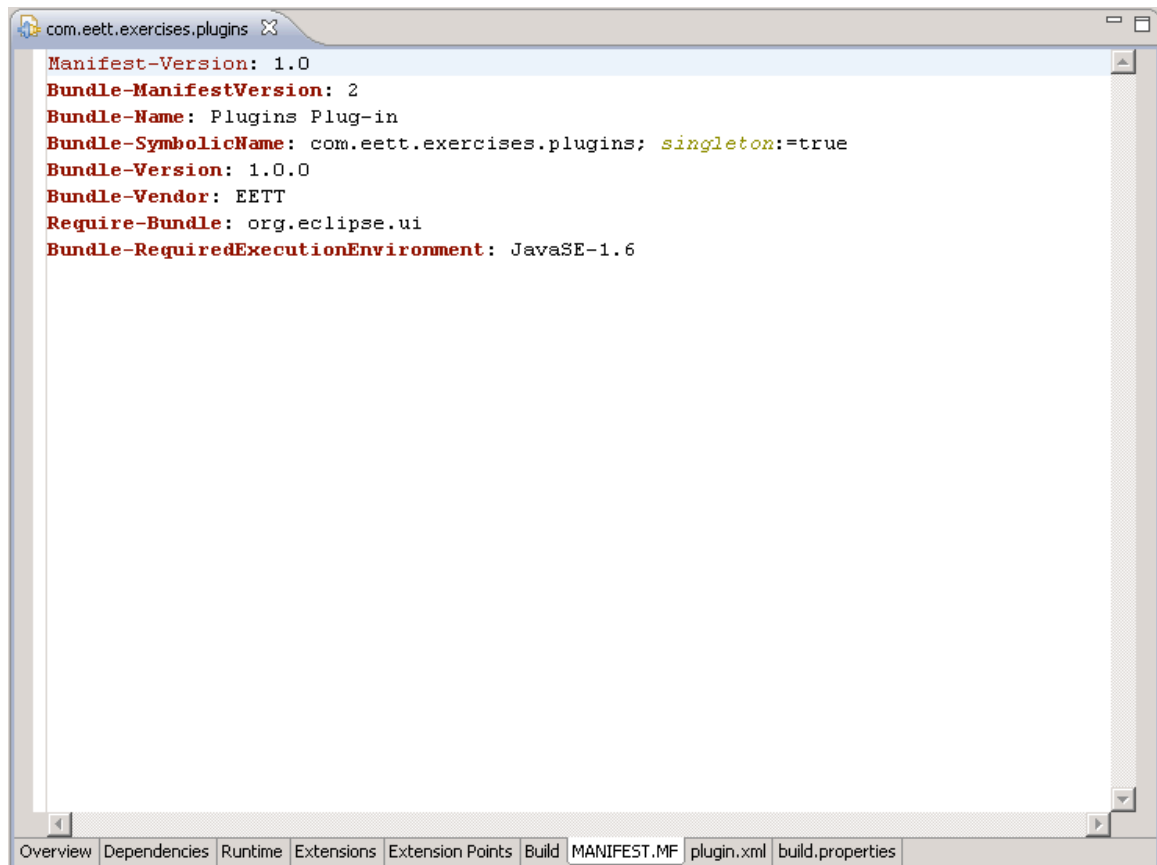
The project wizard that we used in the previous steps created the plug-in manifest files for us. The plugin.xml contains our extensions to the Eclipse workbench and META-INF/MANIFEST.MF describes the plug-in and its dependencies.

Open the MANIFEST.MF file in the Plug-in Manifest Editor and explore the values using the Overview, Dependencies and Runtime pages.



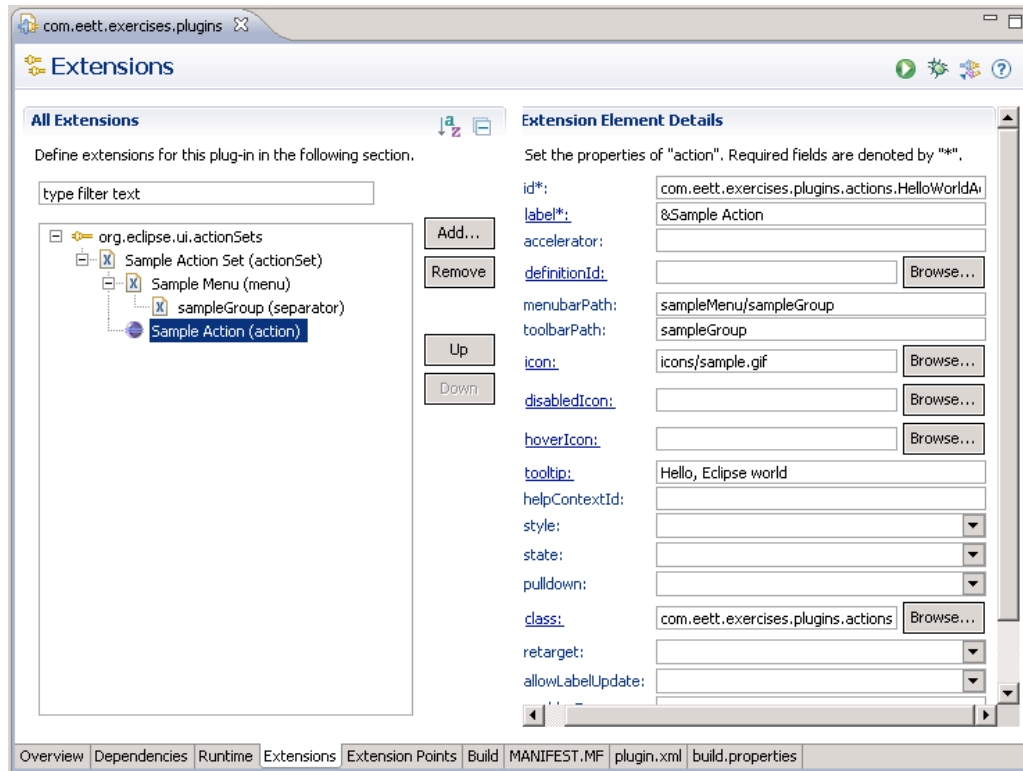
## 8 | ZeligsoftEclipse TransformationTraining Exercise Workbook #1

You can also change the MANIFEST.MF manually using the MANIFEST.MF page in the editor. We do not need to change this file at this point.



Open the plugin.xml using the Plug-in Manifest Editor. The current file should look like the following. We will change some of these values in order to customize the defaults that were generated by the wizard.





(xml content on next page)

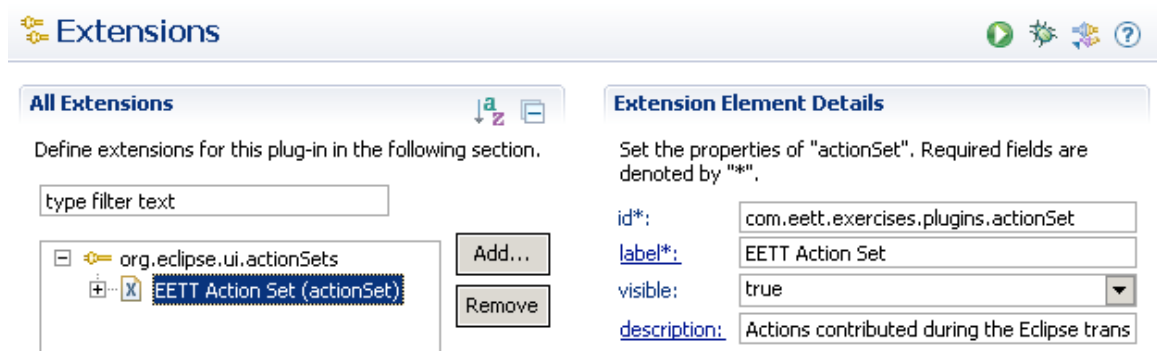
```

<plugin>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Sample Action Set"
      visible="true"
      id="com.eett.exercises.plugins.actionSet">
      <menu
        label="Sample &Menu"
        id="sampleMenu">
        <separator
          name="sampleGroup">
        </separator>
      </menu>
      <action
        label="&Sample Action"
        icon="icons/sample.gif"
        class="com.eett.exercises.plugins.actions.HelloWorldAction"
        tooltip="Hello, Eclipse world"
        menubarPath="sampleMenu/sampleGroup"
        toolbarPath="sampleGroup"
        id="com.eett.exercises.plugins.actions.HelloWorldAction">
      </action>
    </actionSet>
  </extension>
</plugin>

```

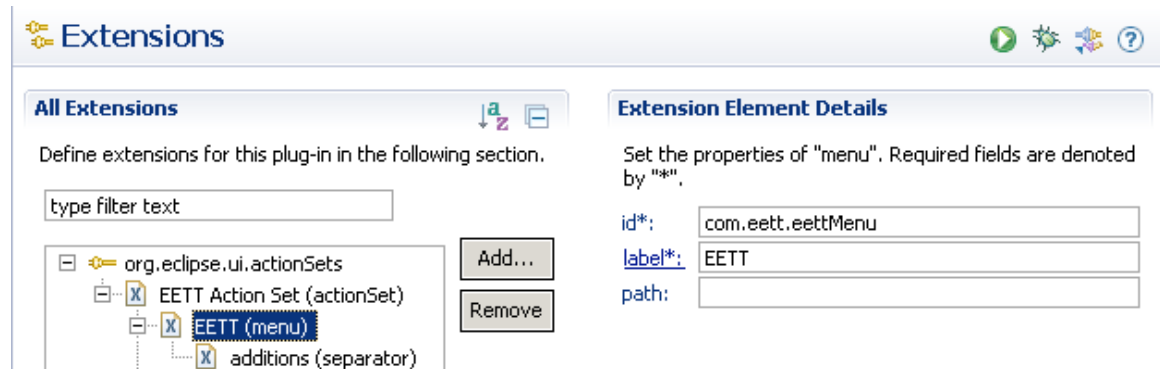
When we ran the plug-in that was generated the menu item names were not very descriptive or customized. We will change the names of the menus from Sample... to something more specific to our exercises. We will reuse the main menu item that is contributed to the workbench so we need to ensure that its name is applicable and that its id is intuitive.

Start by changing the label of the actionSet to EETT Action Set, which is the label used by the Workbench to represent this action set to the user. We will also set the description field to 'Actions contributed during the Eclipse transformation training'. This change will show up in the perspective configuration where you can show and hide action sets.

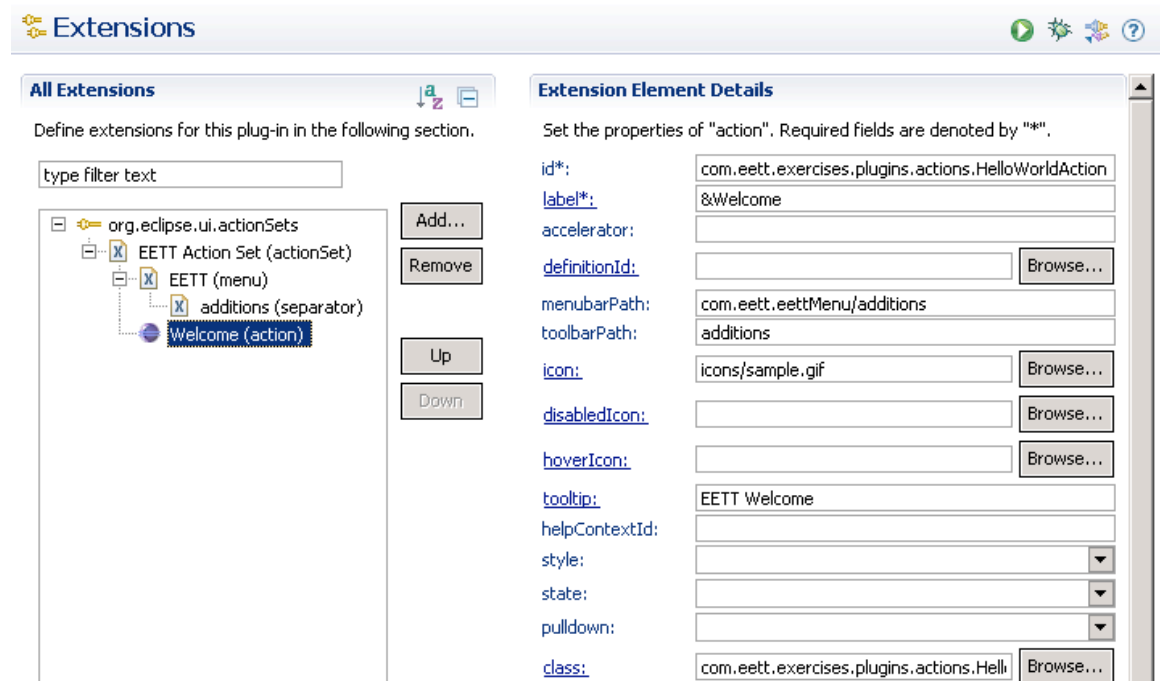


Change the menu label to EETT, which will be the label used in the main menu of the workbench. Also, change the id of the menu to com.eett.eettMenu, this id is used to refer to the menu to add actions.

Change the menu's separator to have the name additions rather than sampleGroup. This separator is used as part of the path to add actions to the EETT menu.

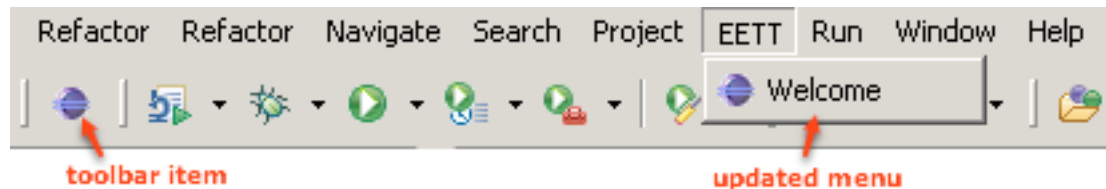


We can now update the action to reflect the changes we have made to the menu and action set. We will start by changing the label to Welcome to better reflect what the action does. In the previous steps we changed the id of the menu and the name of its separator so we have to update the path menubarPath to reflect this; change it to com.eett.eettMenu/additions. We can also change the tooltip to EETT Welcome.

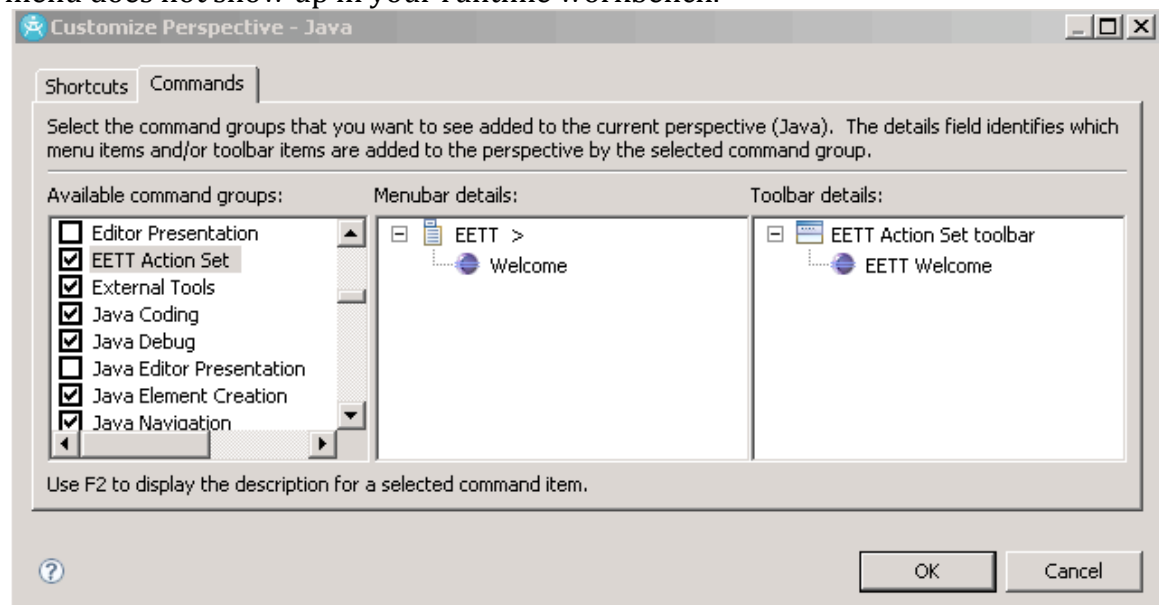


The plugin.xml file should now resemble

```
<plugin>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="EETT Action Set"
      visible="true"
      id="com.eett.exercises.plugins.actionSet">
      <menu
        label="EETT"
        id="com.eett.eettMenu">
        <separator
          name="additions">
        </separator>
        </menu>
      <action
        label="Welcome"
        icon="icons/sample.gif"
        class="com.eett.exercises.plugins.actions.HelloWorldAction"
        tooltip="EETT Welcome"
        menubarPath="com.eett.eettMenu/additions"
        toolbarPath="additions"
        id="com.eett.exercises.plugins.actions.HelloWorldAction">
      </action>
    </actionSet>
  </extension>
</plugin>
```



To view our action set name changes we can open the Customize Perspective dialog; Window → Customize Perspective... This is also a good place to look if the menu does not show up in your runtime workbench.



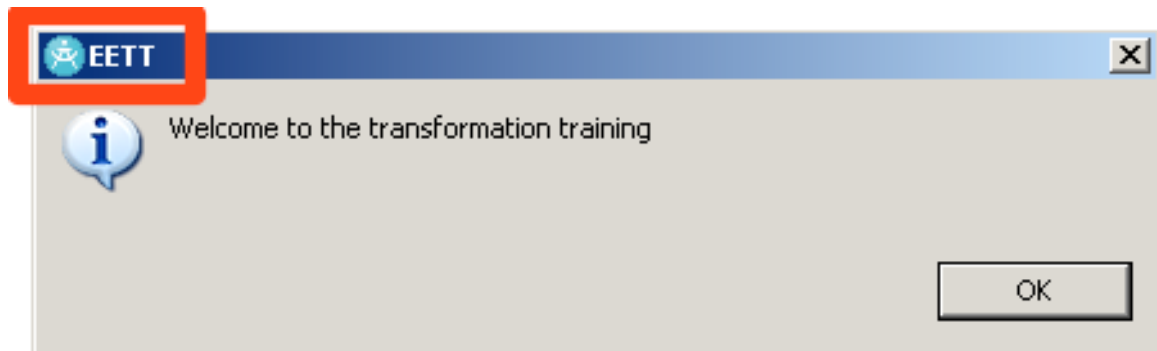
### 1.5 Implement the action delegate

This part of the exercise will modify the default action delegate that was generated by the wizard. Switch to the Java perspective and open the HelloWorldAction.java file. The code is documented, take a few minutes to familiarize yourself with it.

We are going to change the title for the dialog box. In the run method change the second parameter of `MessageDialog.openInformation` to "EETT". *Note that it is best practice to externalize strings such as this one so that they can be localized.* Your run method should now resemble.

```
public void run(IAction action) {  
    MessageDialog.openInformation(  
        window.getShell(),  
        "EETT",  
        "Welcome to the transformation training");  
}
```

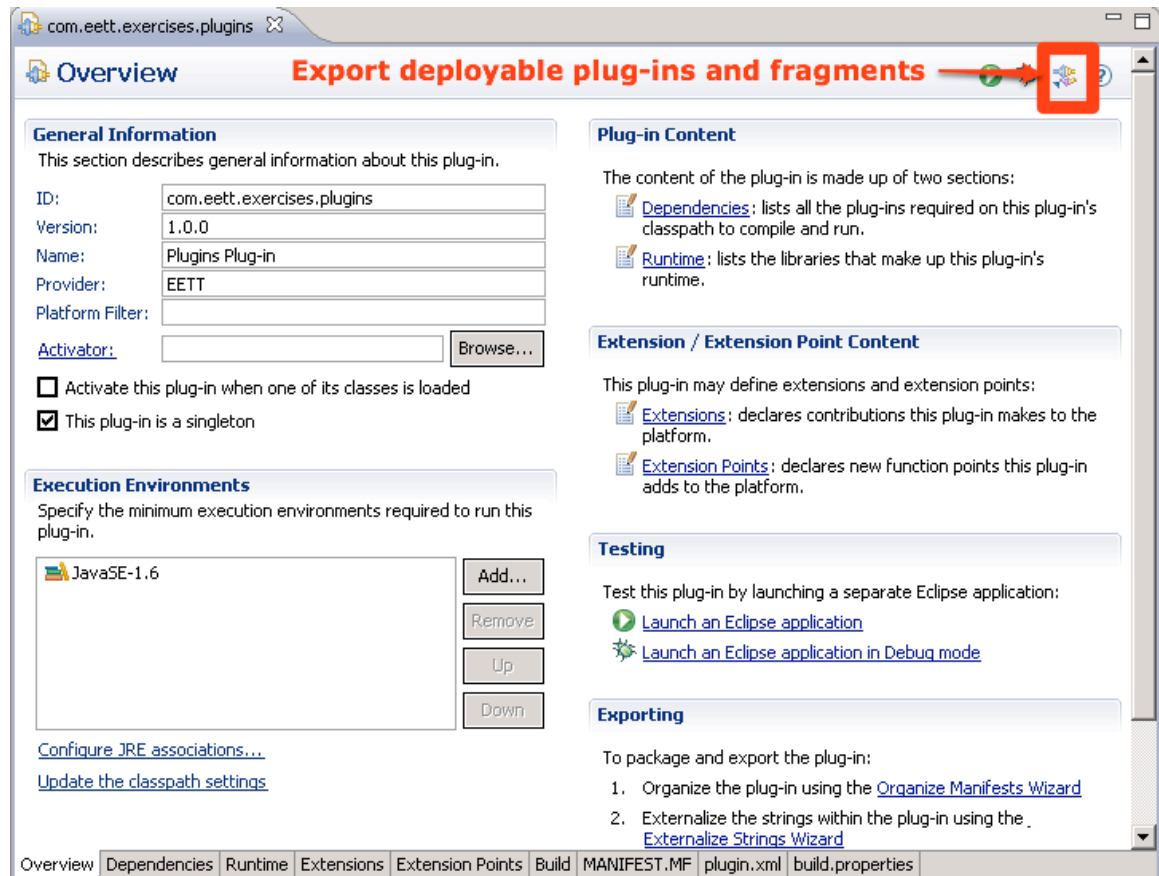
Save your changes and launch the runtime workbench to ensure that your changes were made. It should now look like.



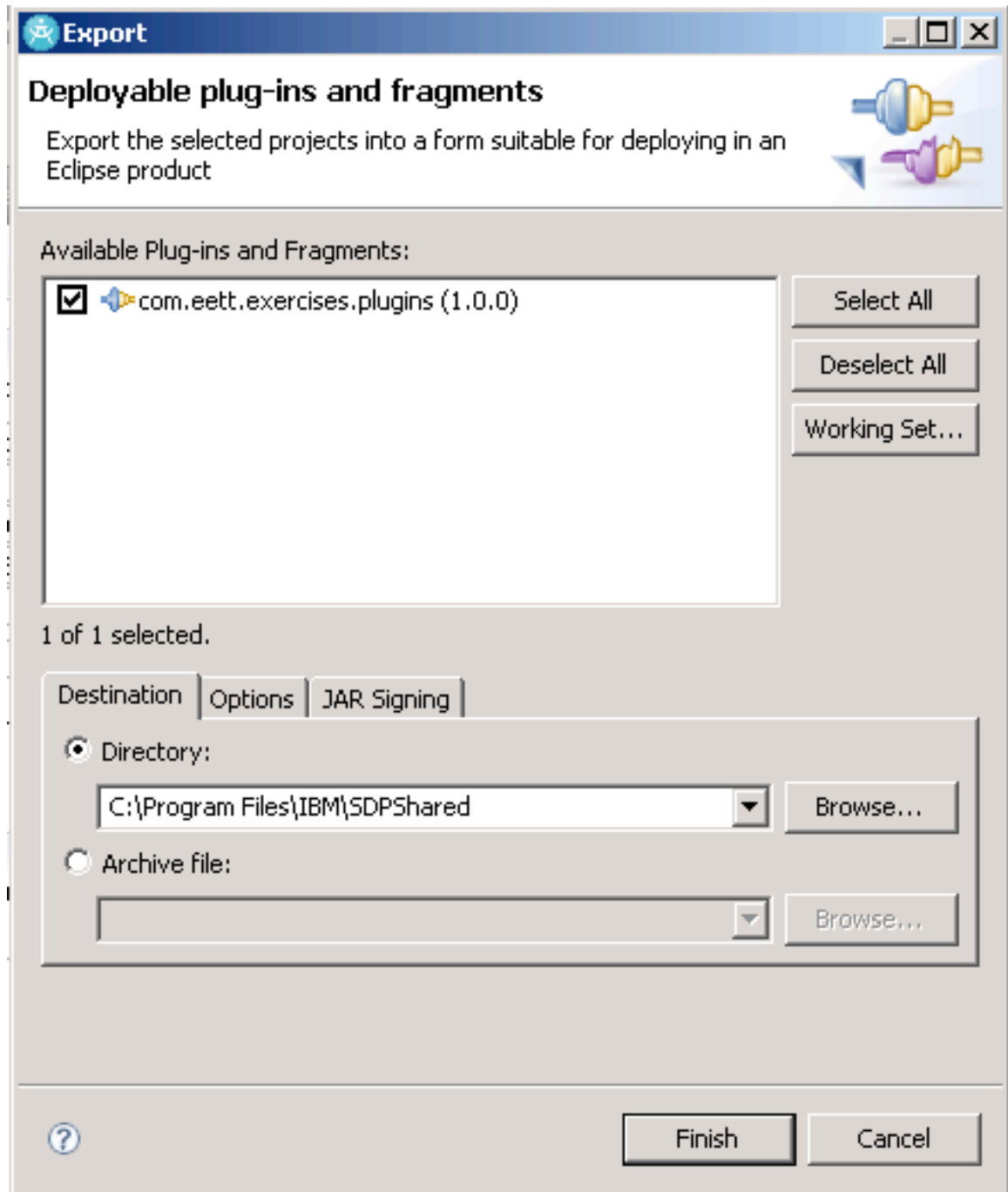
## 1.6 Publish the plug-in

We now have a plug-in and we can publish it to our installation so that it is available without running the workbench.

Open the Plug-in Manifest Editor for our plug-in and switch to the Overview page. In the top right-hand corner is a button to export the plug-in. Click on the button.



In the dialog make sure your plug-in is selected in the list of Available Plug-ins and specify Destination Directory as a dropins directory in your Eclipse install. With RSA-RTE this will likely be SDP directory of your install. Leave the other settings to their defaults.



Click the Finish button and restart your workbench. When the workbench restarts it will have the EETT menu just like in the runtime workbench. Test to make sure that it behaves the same.

Through out the course we will add additional actions to the EETT menu.

## 2 Eclipse Modeling Framework Exercises

In this exercise a metamodel for the structural part of ROOM will be developed in order to gain a better understanding of EMF. The exercise will start by creating an Ecore model of the ROOM language. This model is used to create dynamic instances and to generate the code for the metamodel and a default tree based editor. There will be activities that show how to work with the generated code in order to implement the derived attributes, implement operations and override code that is generated by EMF.

The last two activities of the exercise will practice working with transactions and querying a model.

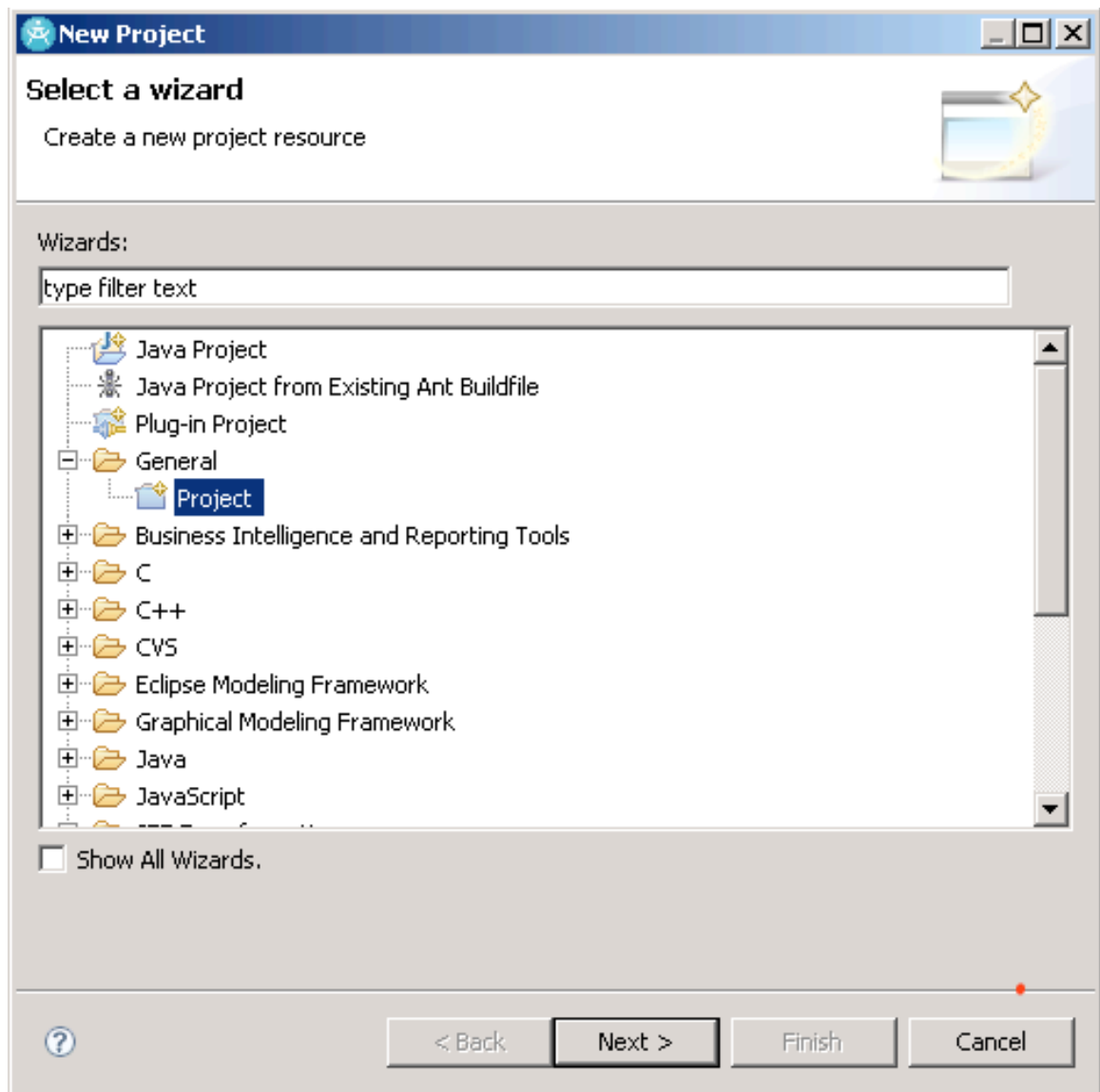
### 2.1 Building an Ecore Model

Create a new project basic project.

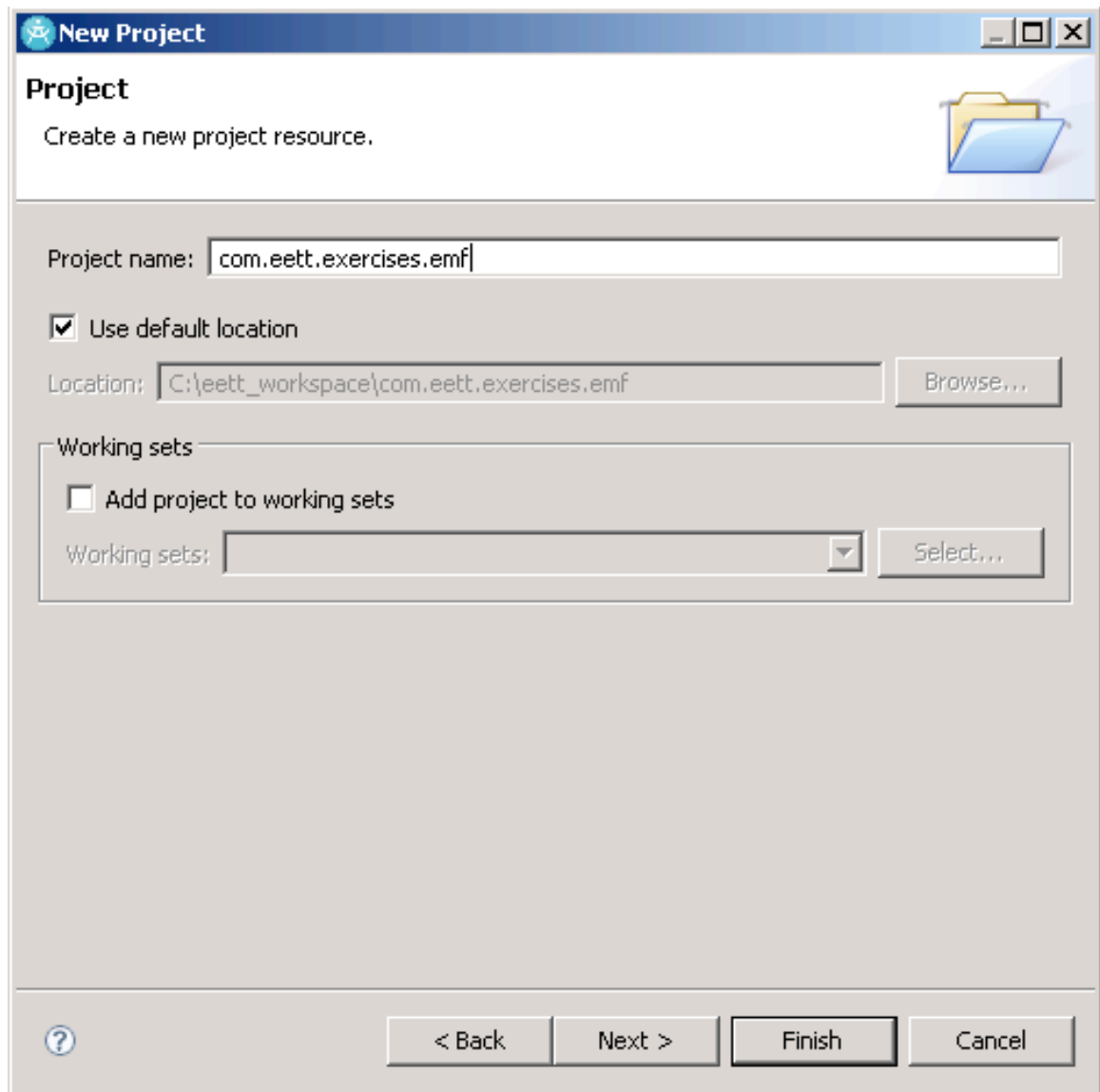
From the main menu select File → New → Project...

In the New Project Wizard select Project under the General group, and click the Next > button.





Provide a Project Name, com.eett.exercises.emf and click the Finish button

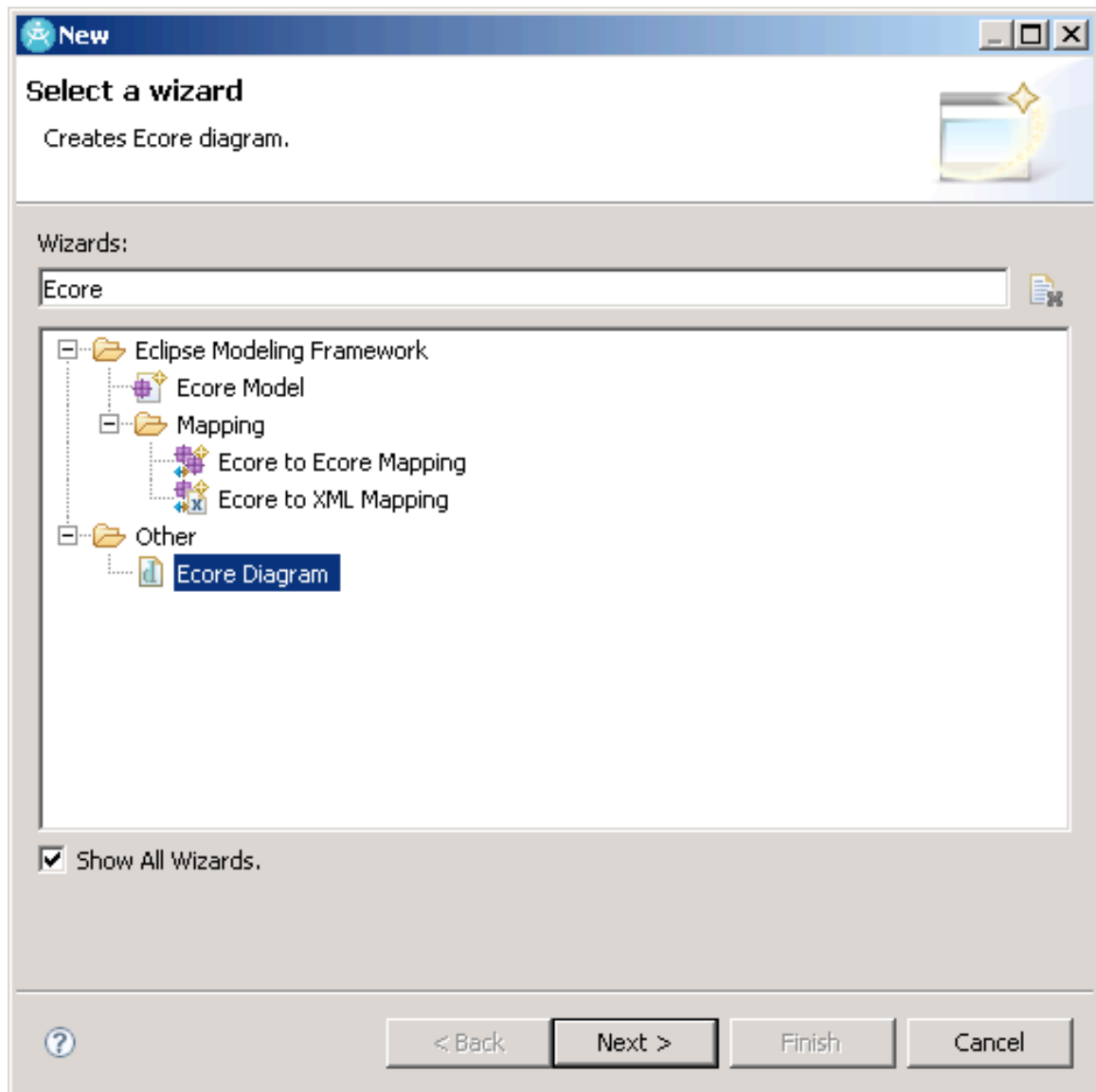


Create a new EMF model.

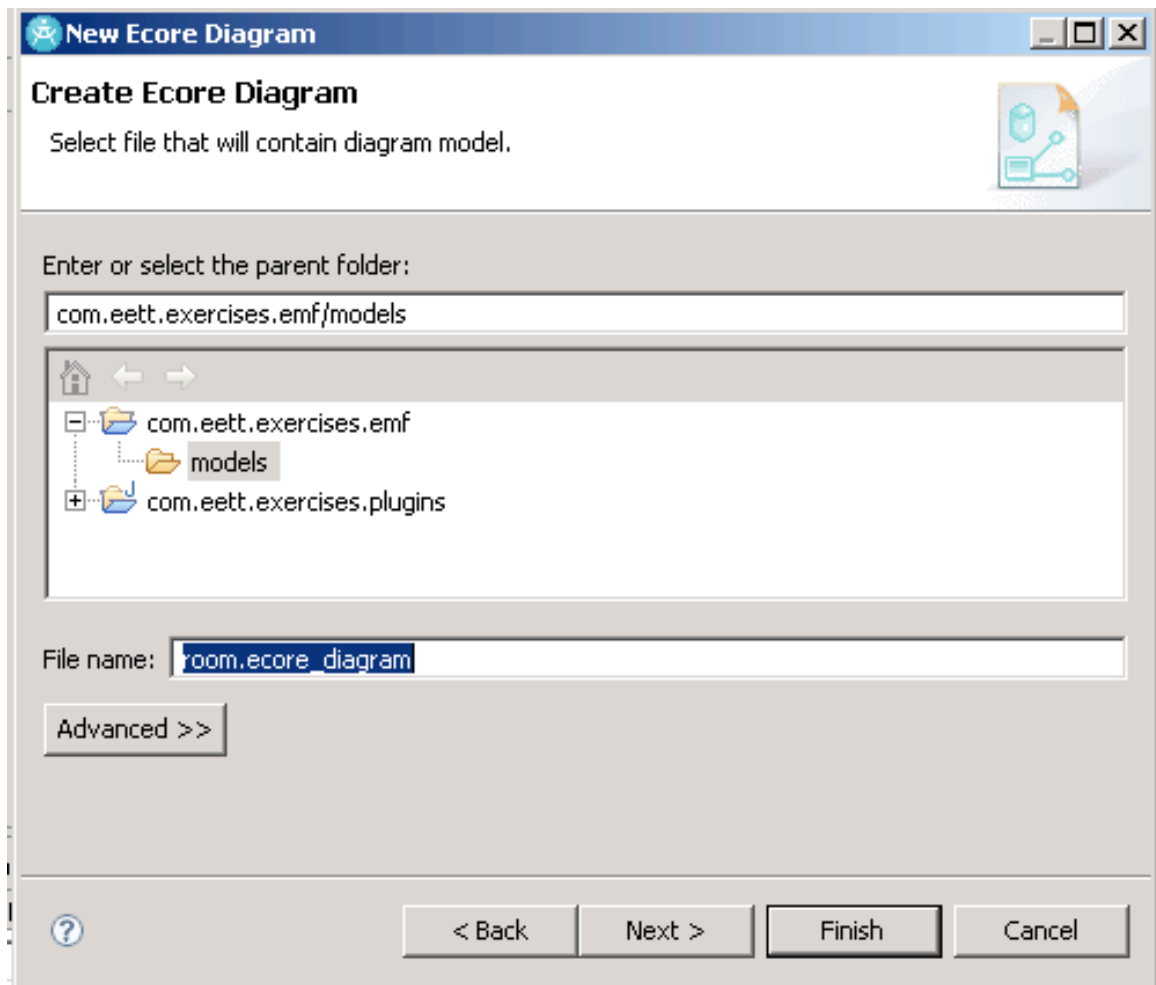
Add a models folder to the com.eett.exercises.emf project, to store our model artifacts

Select the models folder and choose File → New → Other...

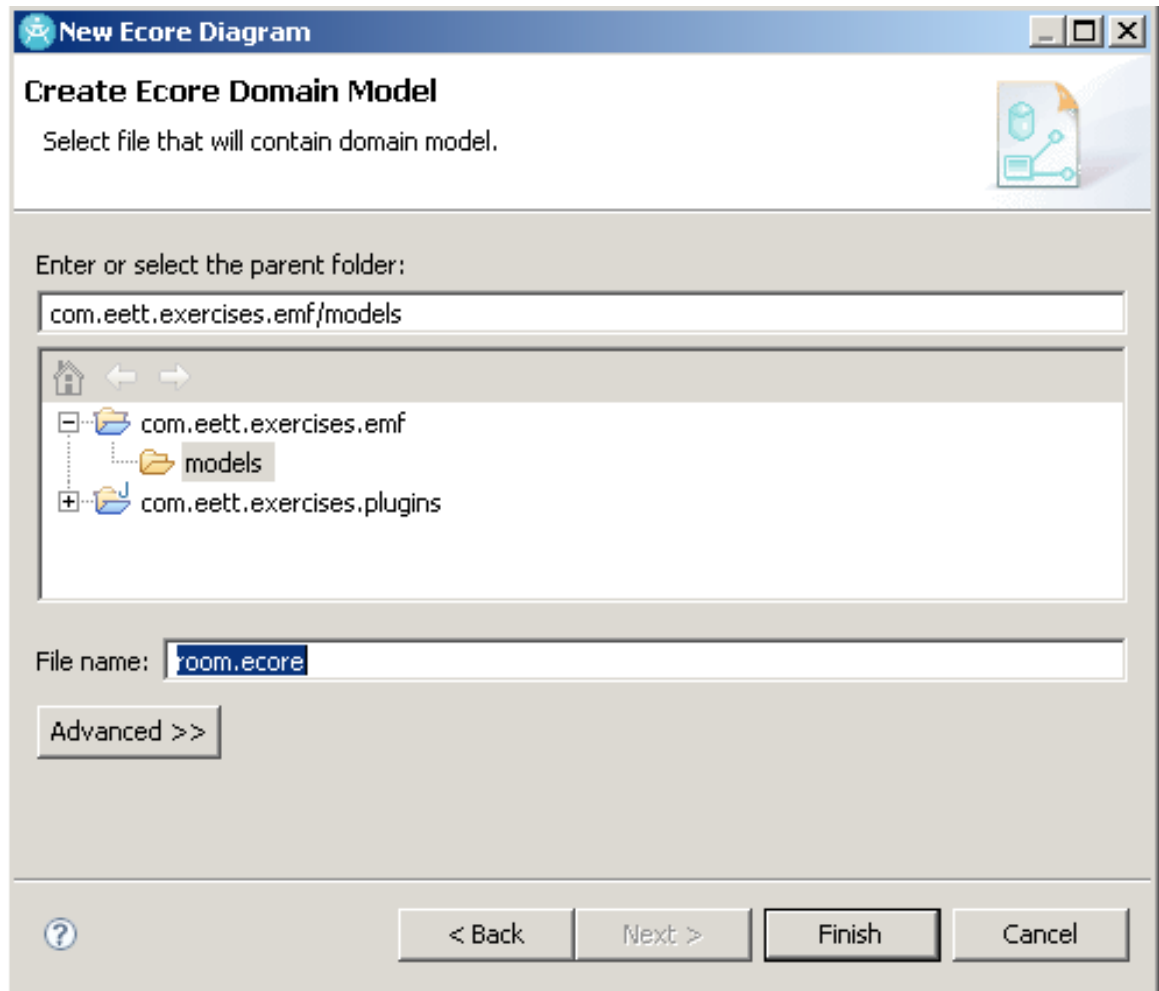
In the New wizard select Ecore Diagram from Other (you may have to check Show All Wizards) and click the Next > button



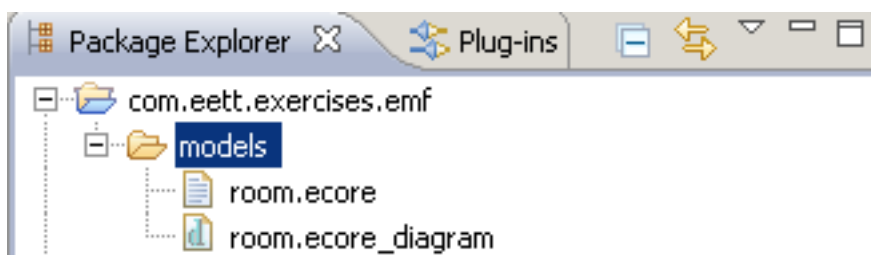
Set the name of the Ecore Diagram to `room.ecore_diagram` and click the Next > button



Set the name of the Ecore model to room.ecore and click the Finish button



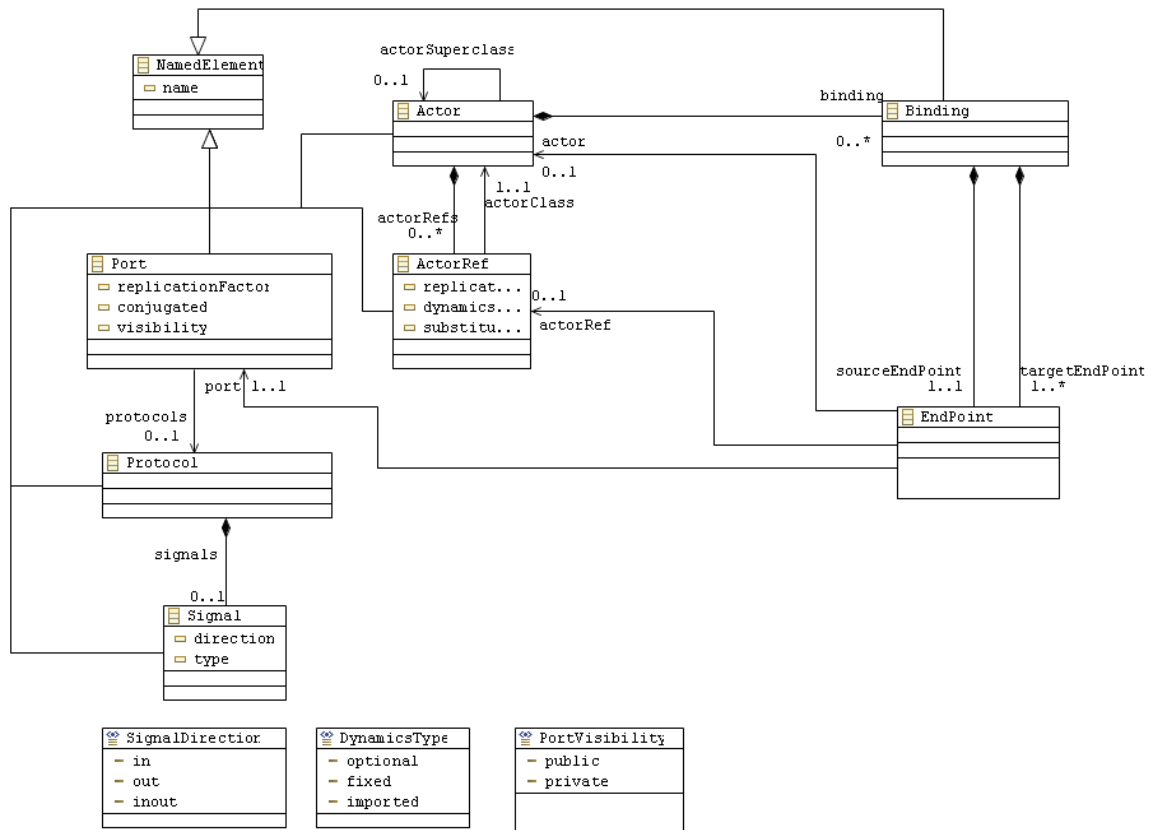
Your project should now look like



Open the Ecore diagram and set the properties in the property grid to the following, this configures the properties of the root ePackage. To access the properties right-click in the diagram editor and select Show Properties View.

ROOM		
Core	Property	Value
	Name	ROOM
	Ns Prefix	room
	Ns URI	http://www.eett.com/room/2008
Rulers & Grid		
Appearance		

In the diagram editor build a model that looks like the following



## NamedElement (Abstract EClass)

Attributes

Name	Type	Lower bound	Upper bound
name	EString	0	1

References

Name	Type	Containment	Opposite	Lower bound	Upper bound

## Actor (EClass) extends NamedElement

Attributes

Name	Type	Lower bound	Upper bound

References

Name	Type	Containment	Opposite	Lower bound	Upper bound
actorSuperClass	Actor	false		0	1
bindings	Binding	true	actor	0	-1
actorRefs	ActorRef	true	container	0	-1
ports	Port	true		0	-1

## ActorRef (EClass)

Attributes

Name	Type	Lower bound	Upper bound	Default
replicationFactor	EInt	0	1	
dynamicsType	DynamicsType	0	1	fixed
substitutable	EBoolean	0	1	false

References

Name	Type	Containment	Opposite	Lower bound	Upper bound
actorClass	Actor	false		1	1
container	Actor	false	actorRefs	1	1

**Port (EClass)**

## Attributes

Name	Type	Lower bound	Upper bound	Default
replicationFactor	EInt	0	1	1
conjugated	EBoolean	0	1	false
visibility	PortVisibility	0	1	public

## References

Name	Type	Containment	Opposite	Lower bound	Upper bound
protocol	Protocol	false		0	1

**Protocol (EClass)**

## Attributes

Name	Type	Lower bound	Upper bound

## References

Name	Type	Containment	Opposite	Lower bound	Upper bound
signals	Signal	true		0	-1

**Signal (EClass)**

## Attributes

Name	Type	Lower bound	Upper bound	Default
direction	SignalDirection	0	1	in
type	EString	0	1	

## References

Name	Type	Containment	Opposite	Lower bound	Upper bound

**SignalDirection (EEnum)**

Name	Value
IN	0
OUT	1
INOUT	2

**DynamicsType (EEnum)**

Name	Type
optional	0
fixed	1
imported	2



### PortVisibility(EEnum)

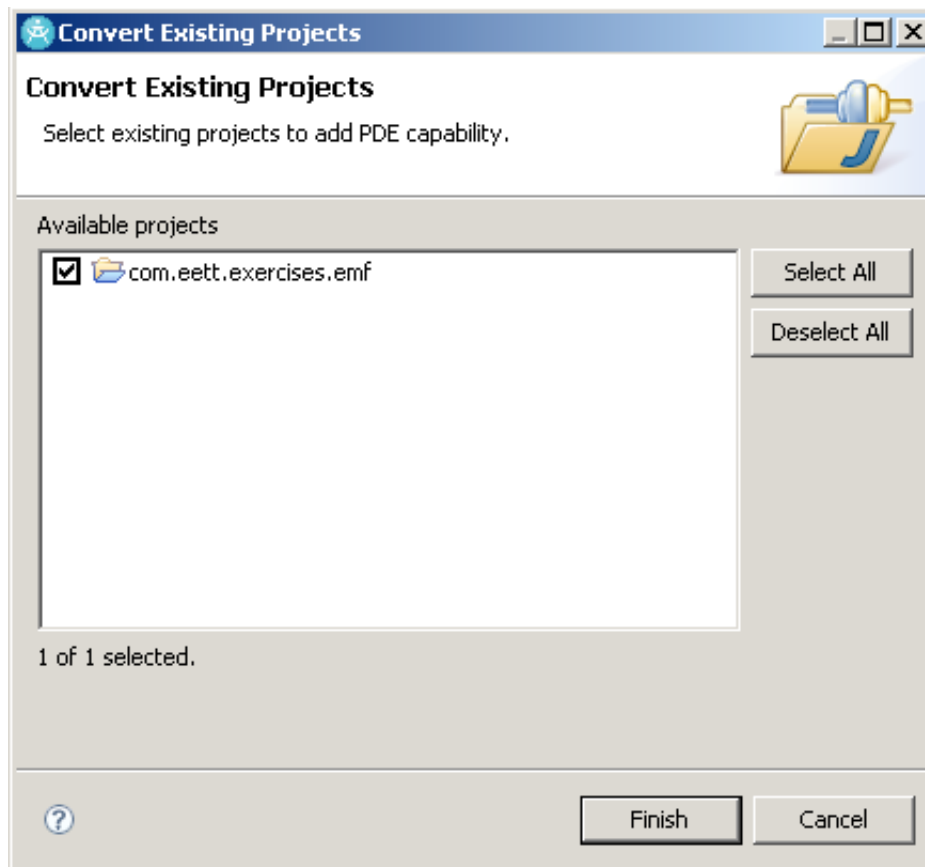
Name	Type
public	0
private	1

When finished creating the model validate it to ensure that there are no errors. To validate select the root package and Sample Ecore Editor → Validate from the main menu.

We have built a metamodel for a subset of the ROOM language, we can now look at creating the generator model and creating models using the metamodel.

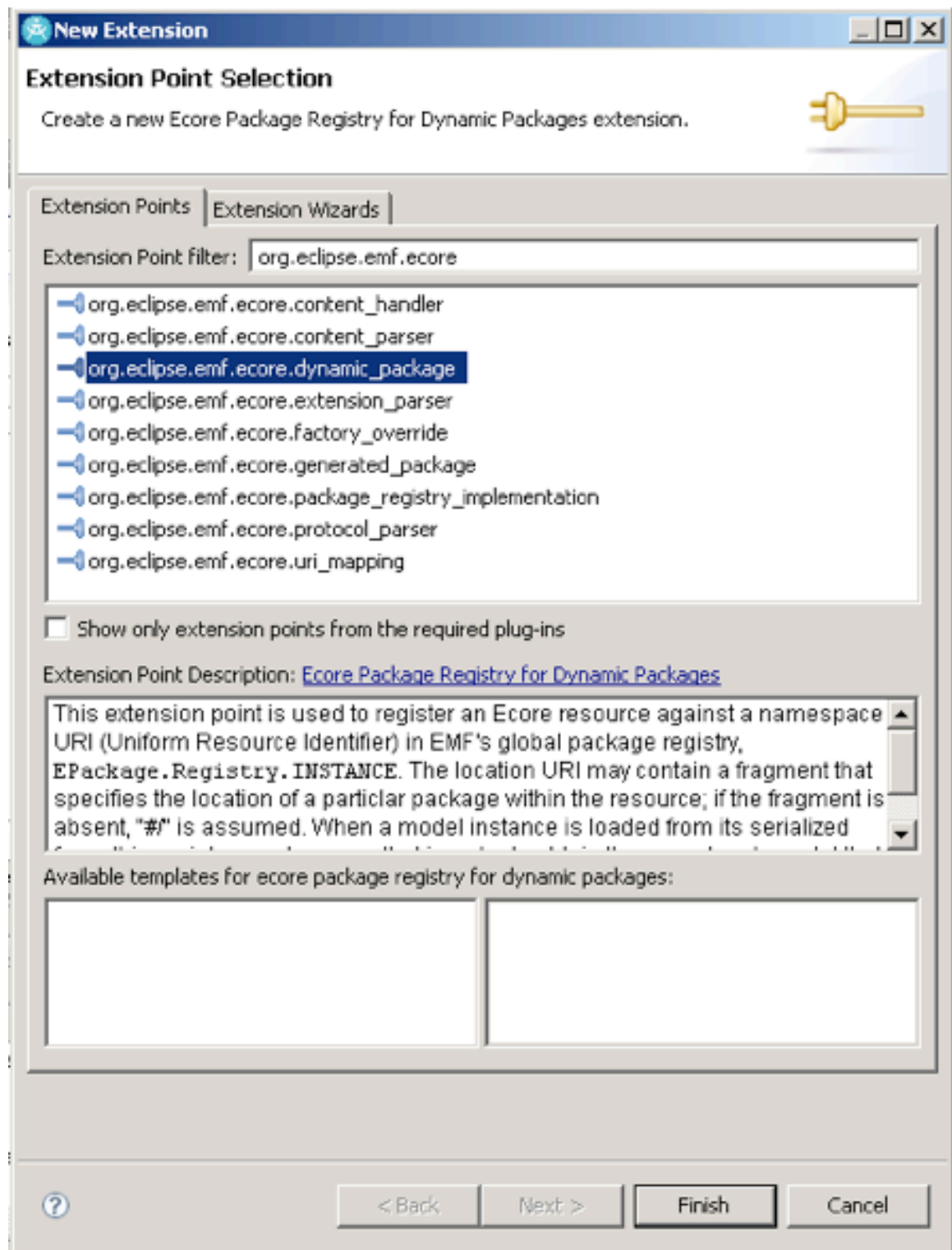
Publish the Model so that it can be used dynamically

We will first convert our com.eett.exercises.emf project to a plugin project. Select the project and choose PDE Tools → Convert Projects to Plug-in Projects... Make sure that com.eett.exercises.emf is checked and click Finish

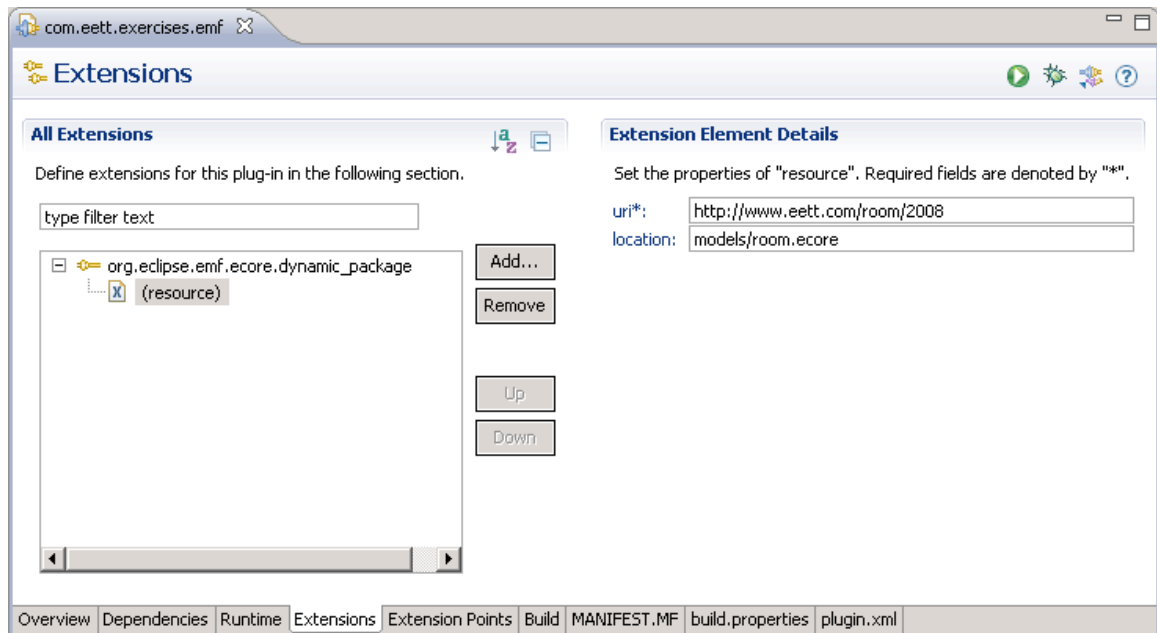


Open the META-INF/MANIFEST.MF in the Plug-in Manifest Editor and change the Name of the plug-in to be EETT EMF Models and the provider to EETT. Save your changes and switch to the Extensions page of the editor.

On the Extensions page click the Add... button. In the Extension Point Selection dialog make sure that Show only extension points from the required plug-ins is unchecked. Then select the com.eclipse.emf.ecore.dynamic\_package extension point. Click the Finish button.



Set the extension points URI field to be the NS URI of the room.ecore model and the location to the location of the room.ecore model. Save your changes.



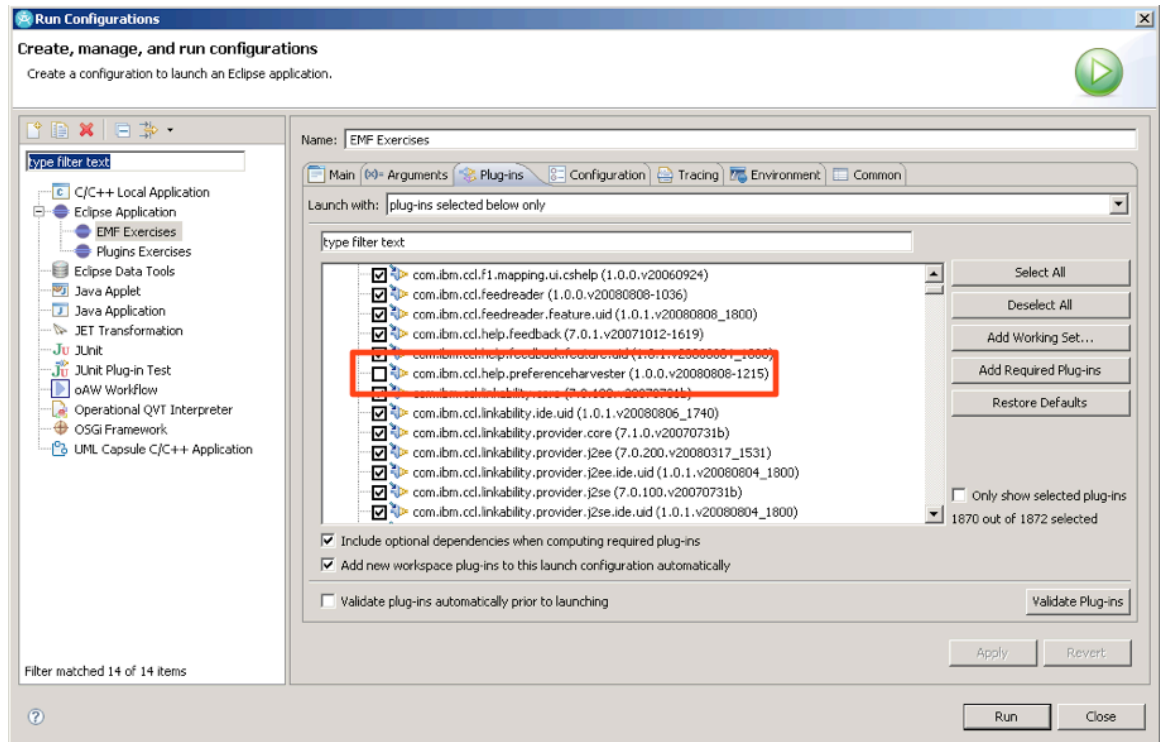
## 2.2 Creating a Dynamic Instance

Once the metamodel has been created it is possible to create instances of the ROOM metamodel that was defined in the previous exercises. The reflective API of EMF allows a generic editor to be provided to create what are called dynamic instances of the model. This is as opposed to generating the implementation of the metamodel and using either the generated editor or building a custom editor.

In the EMF Ecore editor select the Actor element, and select Create Dynamic Instance from the context menu.

Open the Run Configuration dialog (Run → Run Configurations...). Create a new Eclipse Application configuration, EMF Exercises, and switch to the Plug-ins page. Change the Launch with field to 'plug-ins selected below only'.

In the list of plug-ins uncheck `com.ibm.ccl.help.preferenceharvester`, and click Apply followed by Run.



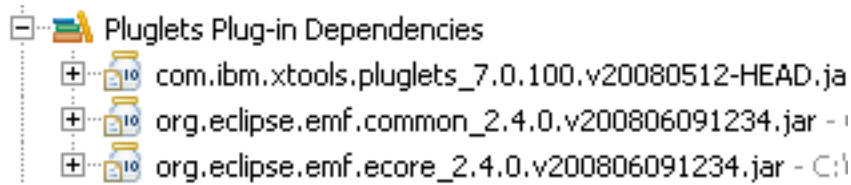
This should launch a runtime workbench, which we will use to programmatically work with our ROOM metamodel. To do this we will use Pluglets, which are an IBM Modeling Platform specific technology for extending the workbench. A pluglet provides script like capabilities, to perform tasks without having to build full plug-ins.

In the runtime workbench create a new Pluglets project, named `com.eett.exercises.emf.pluglets`. Don't worry about the other settings in the project wizard.

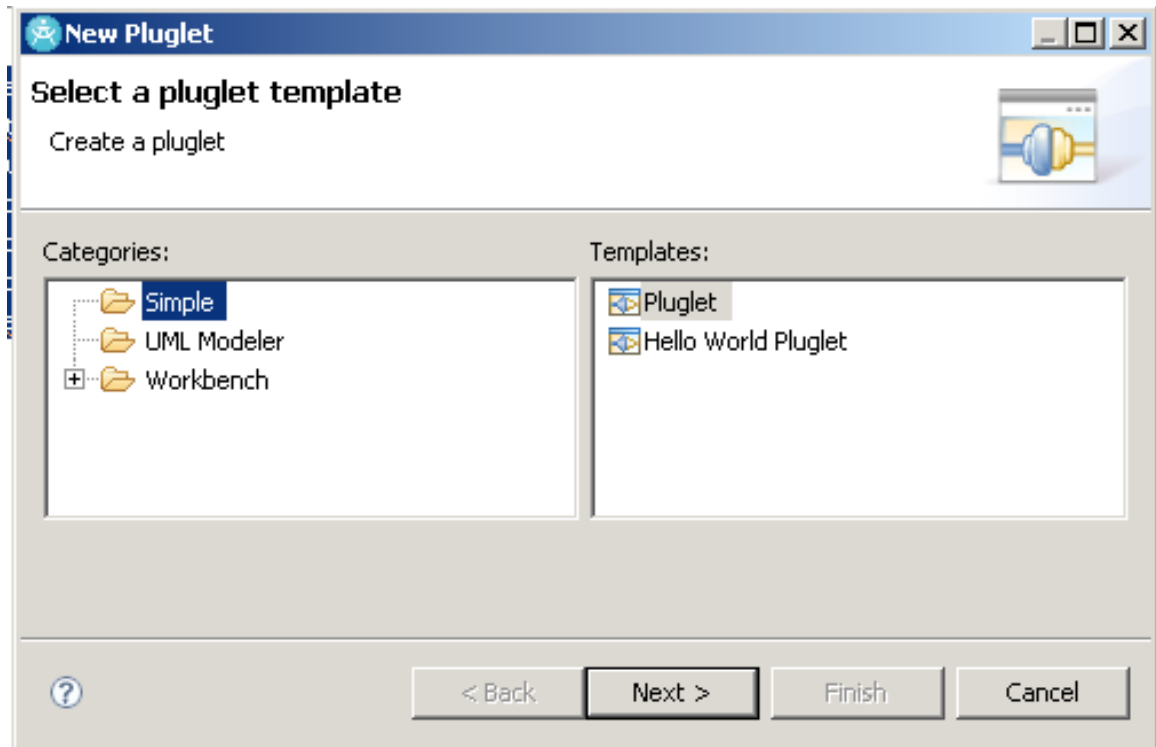
The project will contain a `pluglet.xml` file which is used to capture dependencies on plug-ins. We going to use EMF in this exercise, therefore we need to add a dependency on `org.eclipse.emf.ecore`. Open `pluglet.xml` with a text editor and modify it to look like the following and save the file

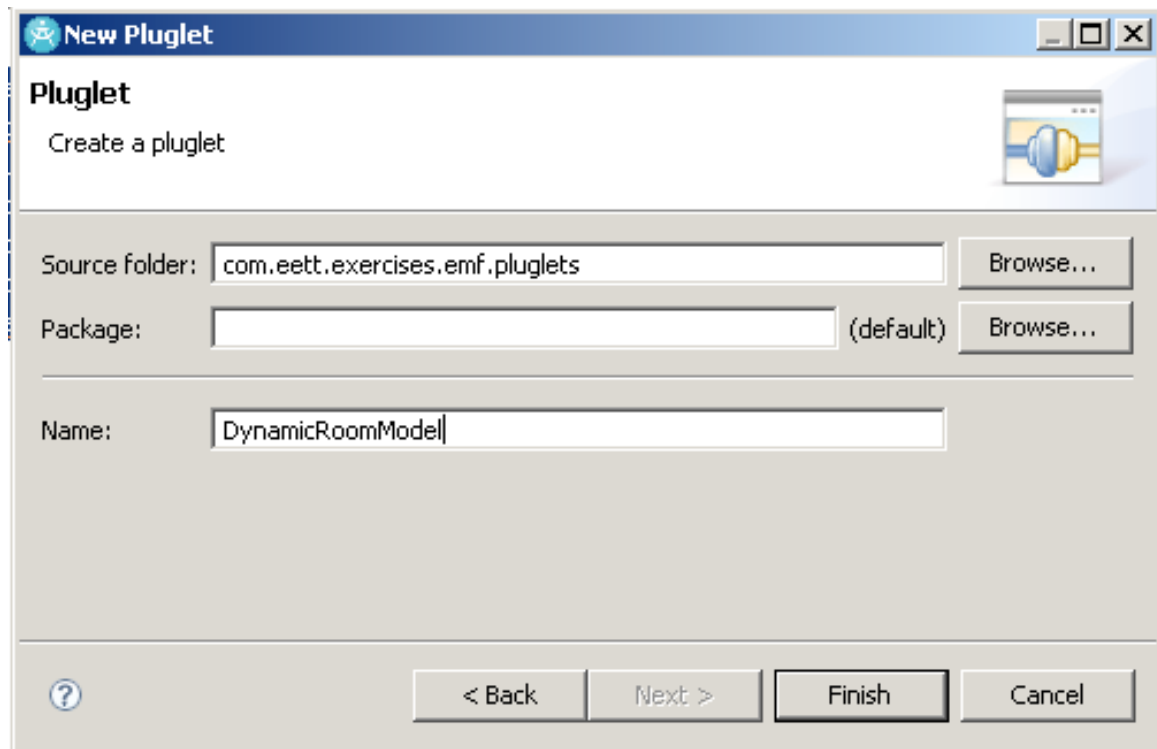
```
<?xml version="1.0" encoding="UTF-8"?>
<pluglets>
  <require>
    <import plugin="com.ibm.xtools.pluglets"/>
    <import plugin="org.eclipse.emf.ecore"/>
  </require>
</pluglets>
```

The Pluglets Plug-in Dependencies should now include `org.eclipse.emf.ecore` and `org.eclipse.emf.common`.



Now create a new Pluglet (File → New → Pluglet) in the project. Select the Simple → Pluglet template. Click the Next > button. Set the name of the Pluglet to DynamicROOMModel and click Finish.





The image shows a 'New Pluglet' dialog box in a software application. The title bar is blue with a pluglet icon and the text 'New Pluglet'. The main area is white with the title 'Pluglet' and the subtitle 'Create a pluglet'. There is a pluglet icon in the top right corner. The dialog has three input fields: 'Source folder:' with the text 'com.eett.exercises.emf.pluglets' and a 'Browse...' button; 'Package:' with a text box and '(default)' and a 'Browse...' button; and 'Name:' with the text 'DynamicRoomModel'. At the bottom, there is a help icon, a '< Back' button, a 'Next >' button, a 'Finish' button, and a 'Cancel' button.

**New Pluglet**

**Pluglet**  
Create a pluglet

Source folder:

Package:  (default)

Name:

Open the DynamicRoomModel.java file and enter the code described below. This code creates two Actor objects, AbstractDyeingController and DyeingController. They are stored in separate resources, abstractDyeingController.xmi and dyeingController.xmi respectively. We then set the actorSuperclass of DyeingController to be AbstractDyeingController.

```
import java.io.IOException;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.EStructuralFeature;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;

import com.ibm.xttools.pluginlets.Pluginlet;

public class DynamicRoomModel extends Pluginlet {
    private static final String MODELS_FOLDER
        = "platform:/resource/com.eett.exercises.emf.pluginlets/models/";

    public void pluginletmain(String[] args) {

        // Retrieved the EPackage for the Room metamodel that we
        // registered
        // It is retrieved using the string we specified as the URI when
        // we
        // registered it in the dynamic_package extension point
        EPackage room
            = EPackage.Registry.INSTANCE
                .getEPackage("http://www.eett.com/room/2008");

        if(room != null) {
            // Create a ResourceSet so that we can create Resources
            // to store the model elements we create in
            ResourceSet resourceSet = new ResourceSetImpl();

            // Create the Resources to store the objects that we create in
            Resource controllerResource
                = resourceSet.createResource(URI.createURI(
                    MODELS_FOLDER + "dyeingController.xmi"));
            Resource abstractControllerResource
                = resourceSet.createResource(URI.createURI(
                    MODELS_FOLDER + "abstractDyeingController.xmi"));

            // In order to be able to create an Actor object we need
            // have its EClass which we can get from the EPackage
            EClass actorEClass = (EClass) room.getEClassifier("Actor");

            // Similarly, to set the features of an Actor we
            // need the EStructuralFeature objects for them which we
            // can get from the EClass
            EStructuralFeature actorNameAttribute
                = actorEClass.getEStructuralFeature("name");
            EStructuralFeature actorSuperclass
                = actorEClass.getEStructuralFeature("actorSuperclass");

            // Create the AbstractDyeingController Actor and add it
```

```

// to the Resource we created for it
EObject abstractController =
    room.getEFactoryInstance().create(actorEClass);
abstractController
    .eSet(actorNameAttribute, "AbstractDyeingController");
abstractControllerResource.getContents()
    .add(abstractController);

// Create the DyeingController Actor and add it
// to the Resource we created for it
EObject controller =
    room.getEFactoryInstance().create(actorEClass);
controller
    .eSet(actorNameAttribute, "DyeingController");
controllerResource.getContents().add(controller);

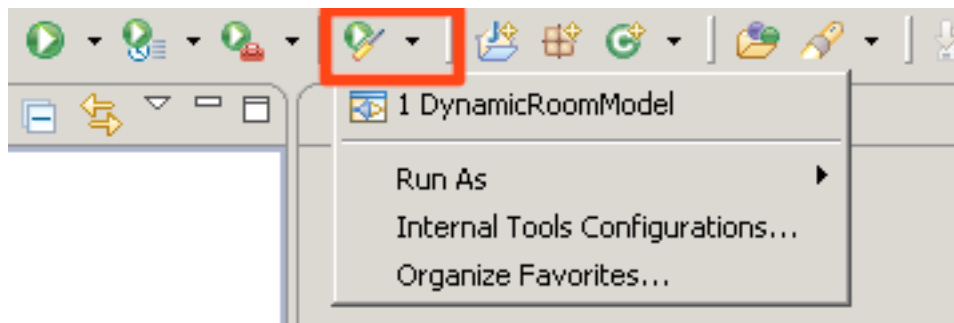
// Set the actorSuperclass feature of the dyeingController
// to be the abstractController
controller.eSet(actorSuperclass, abstractController);

// Save and unload the Resources we created and added
// contents to
try {
    abstractControllerResource.save(null);
    controllerResource.save(null);
    abstractControllerResource.unload();
    controllerResource.unload();
} catch (IOException e) {
    out.println("Could not save the resources.");
}

}
else {
    out.println("Could not get the room EPackage");
}
}
}

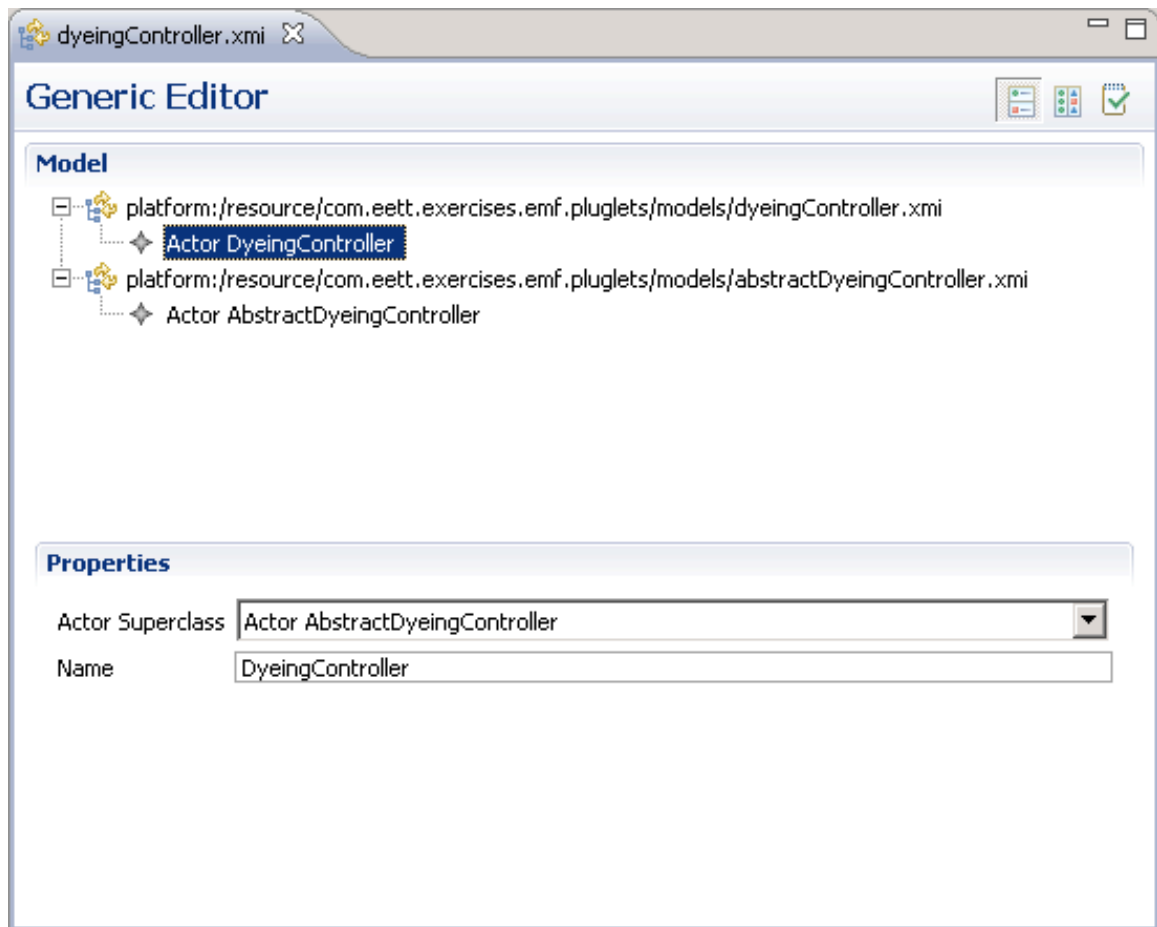
```

Run the Pluglet by selecting Run → Internal Tools → DynamicRoomModel or from the toolbar as shown below.





Running the Pluglet should create a new folder models in your project with two files. Open the dyeingController.xmi file with the Generic Editor. Notice how it also loaded the abstractDyeingController.xmi because there is a cross-reference to it (actorSuperclass of DyeingController).

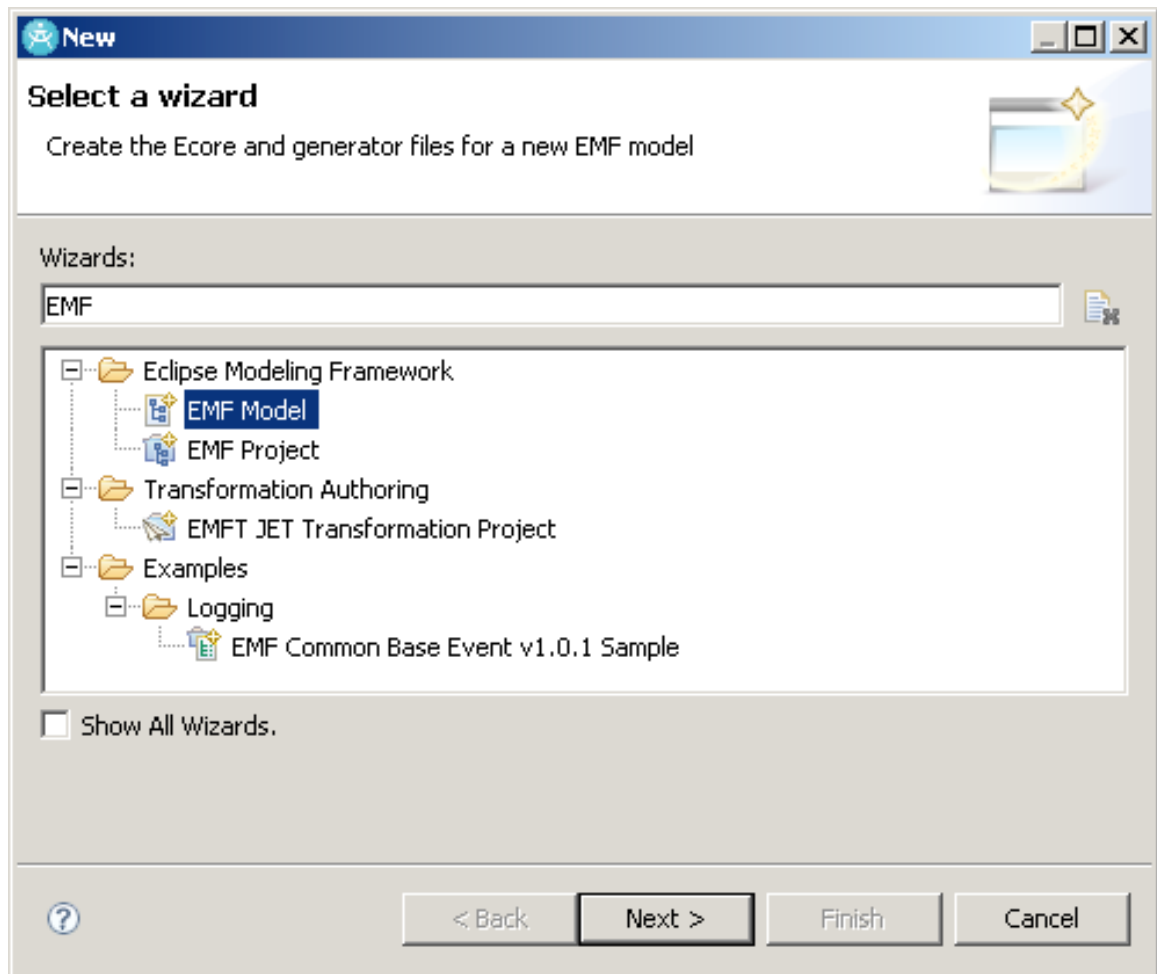


### 2.3 Generating code for the Ecore Model

The Ecore model created in the previous exercise specifies the concepts, relationships and capabilities within our ROOM domain. In order to be able to generate code for this model we need to create a generator model that decorates the metamodel you defined with additional information to help with generation.

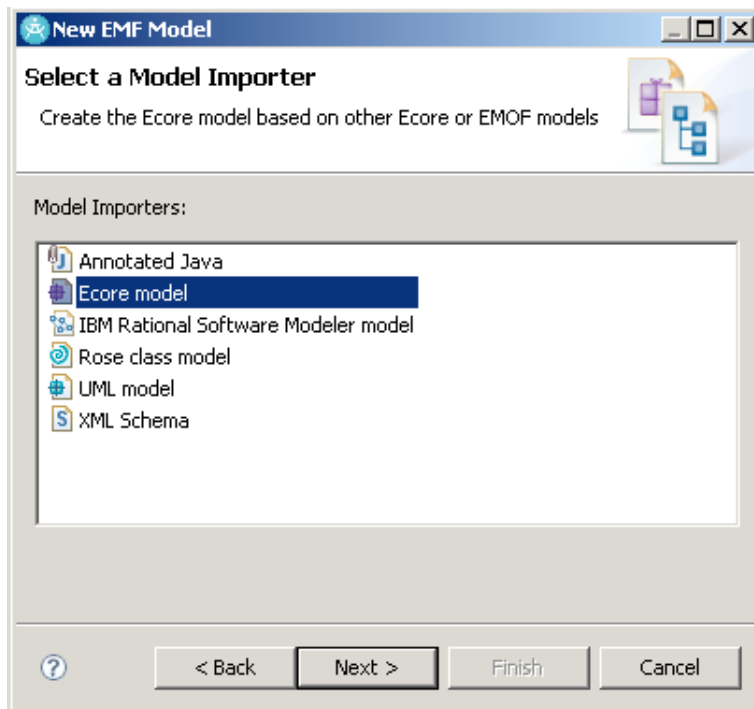
On the models folder in the com.eett.exercises.emf project right-click and select New → Other...

In the New wizard select EMF Model under the Eclipse Modeling Framework folder and click Next >

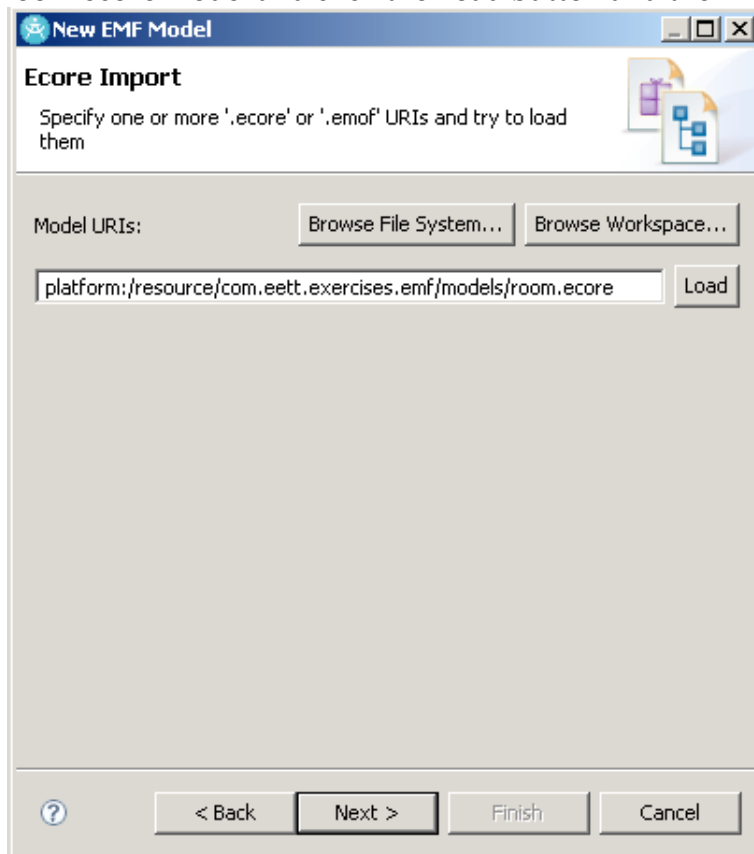


Set the file name to room.genmodel and click the Next > button

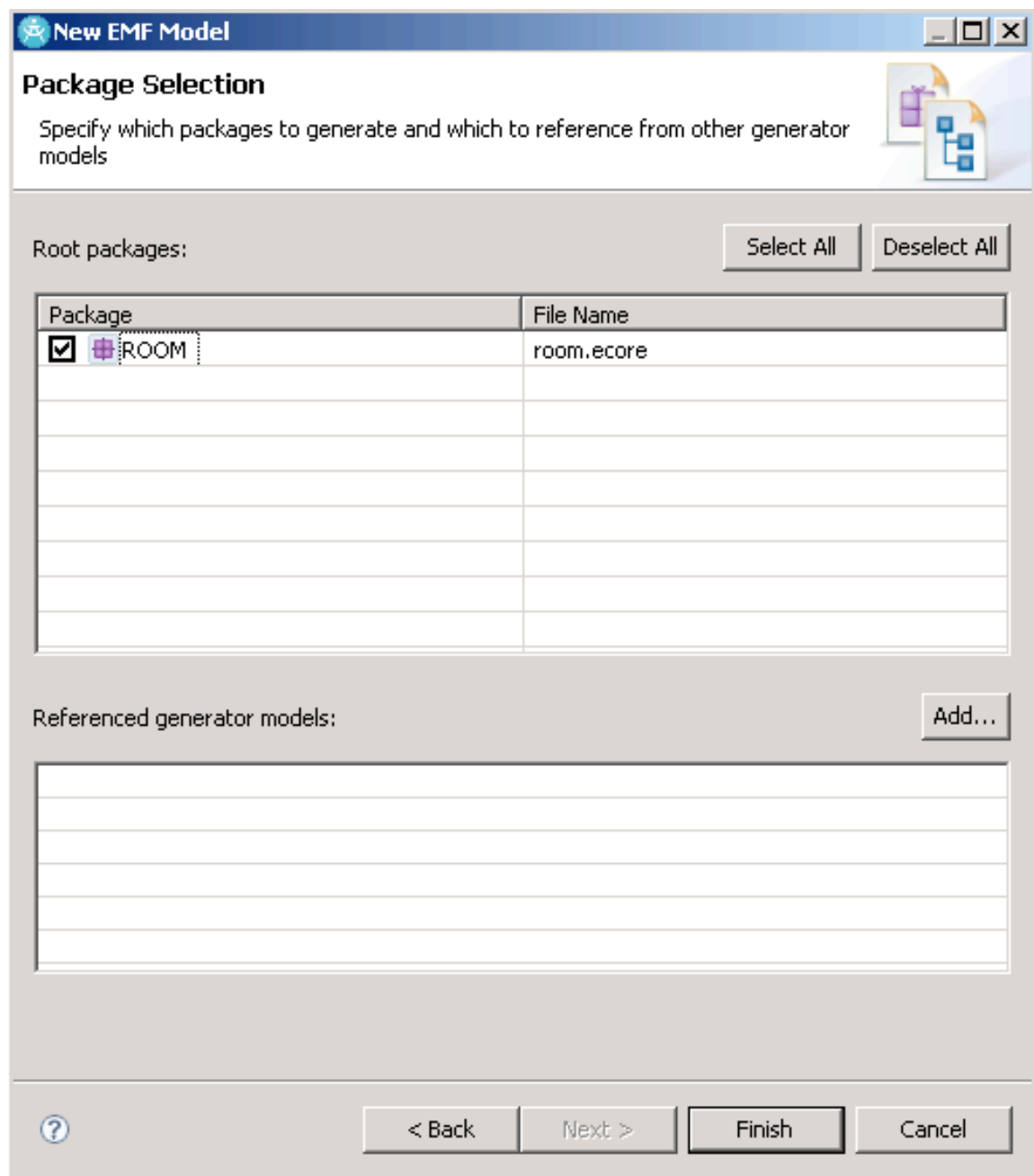
A Model Importer must be selected to create our generator model, select the Ecore Model option and click Next >



On the Ecore Import page use the Browse Workspace to locate and select the room.ecore model and click the Load button and then Next >



The following dialog should appear, select the Finish button



The dialog box is titled "New EMF Model" and contains a "Package Selection" section. It instructs the user to "Specify which packages to generate and which to reference from other generator models".

**Root packages:**

Buttons: **Select All** **Deselect All**

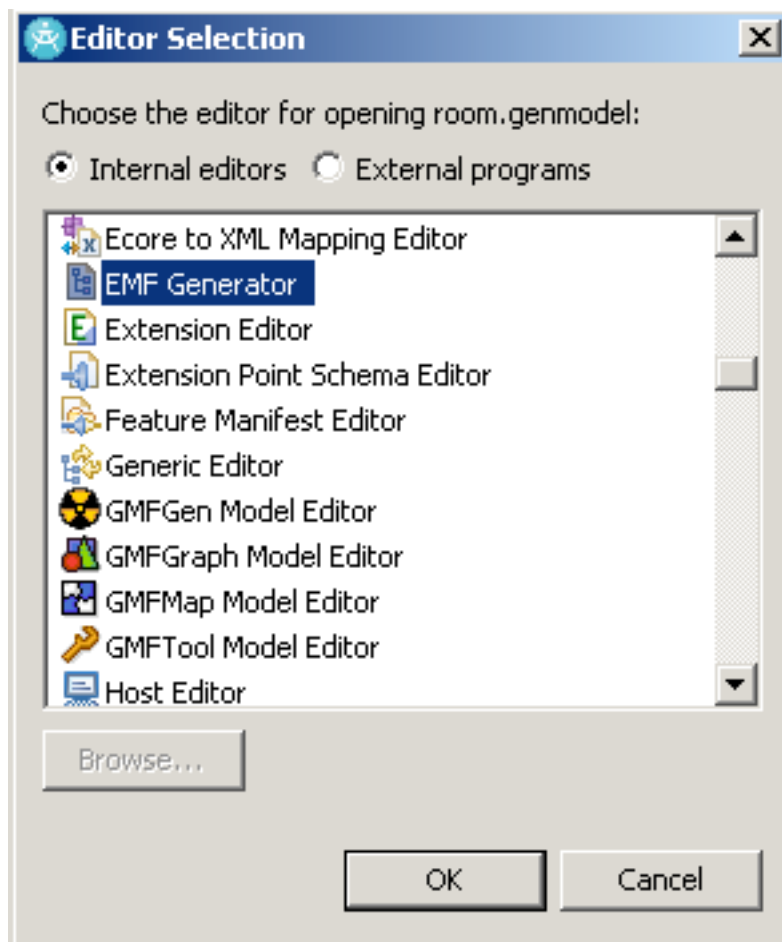
Package	File Name
<input checked="" type="checkbox"/> ROOM	room.ecore

**Referenced generator models:**

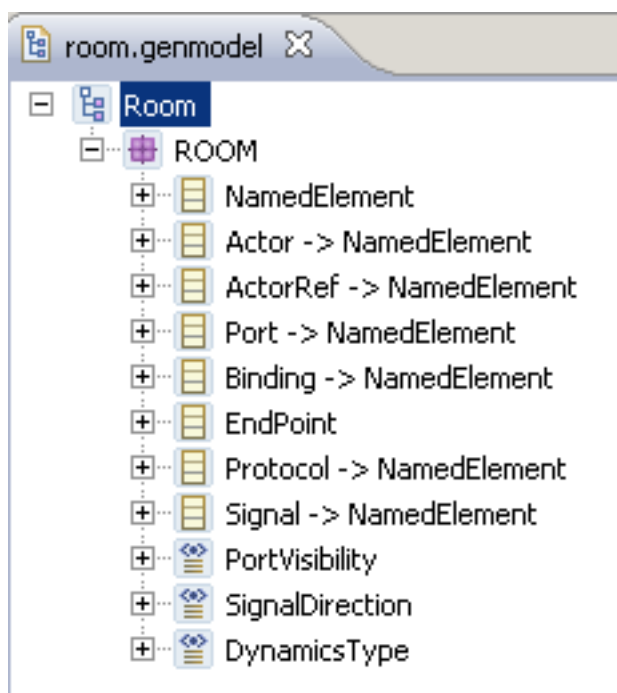
Button: **Add...**

Buttons: **< Back** **Next >** **Finish** **Cancel**

Open the new room.genmodel with the EMF Generator editor,



The model should look like the following

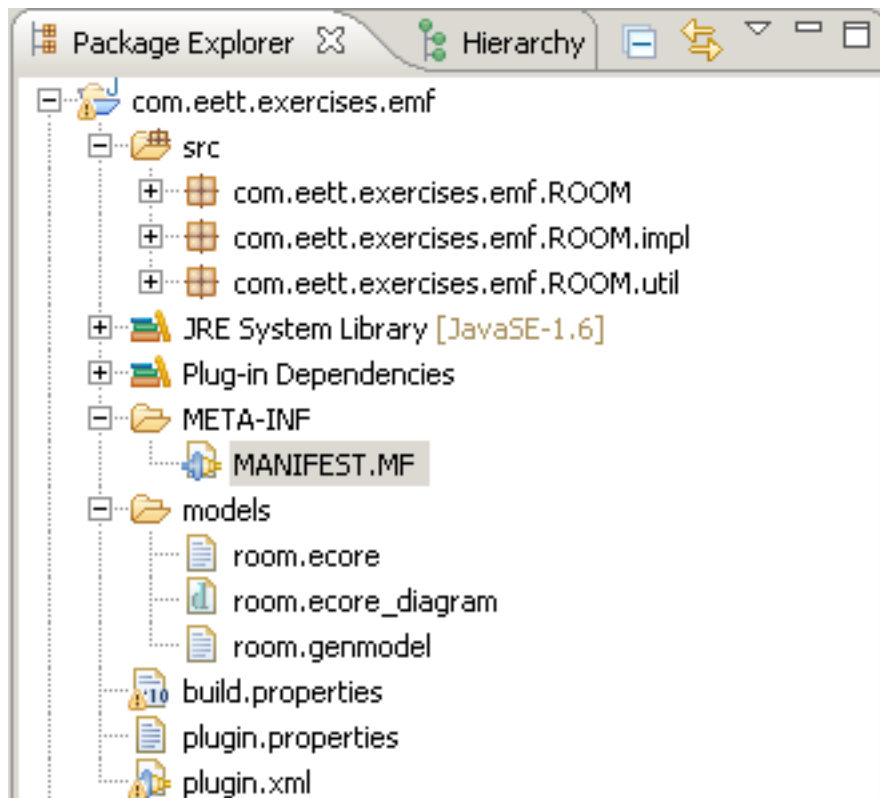


Explore the generator model observing the additional properties that have been added to the elements from our Ecore model. At this point the only property that we need to change is Base Package on the root EPackage ROOM. Change this property to `com.eett.exercises.emf`, this is the package that the generated code will be generated into.

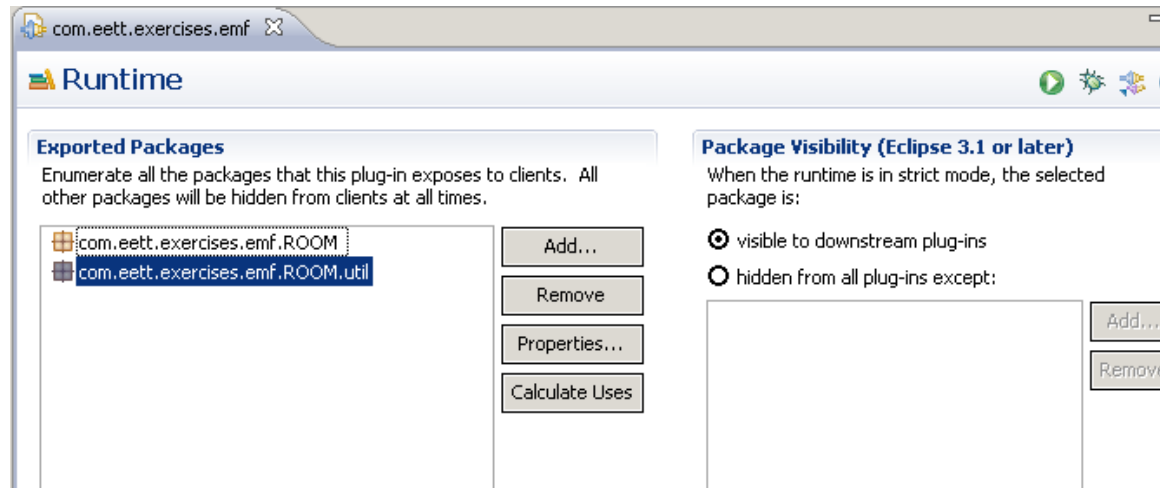
Property	Value
<b>All</b>	
Base Package	<code>com.eett.exercises.emf</code>
Prefix	<code>ROOM</code>
+ Ecore	
+ Edit	
+ Editor	
+ Model	
+ Package Suffixes	
+ Tests	

With the generator model open right-click on the root package element and select Generate All, this will generate three new plug-ins (`com.eett.exercises.emf.edit`, `com.eett.exercises.emf.editor`, and `com.eett.exercises.emf.test`) and add source to the `com.eett.exercises.emf` project that the model is in. It will also modify the `com.eett.exercises.emf` project adding the plug-in and Java nature to the project.

The `com.eett.exercises.emf` project should now resemble

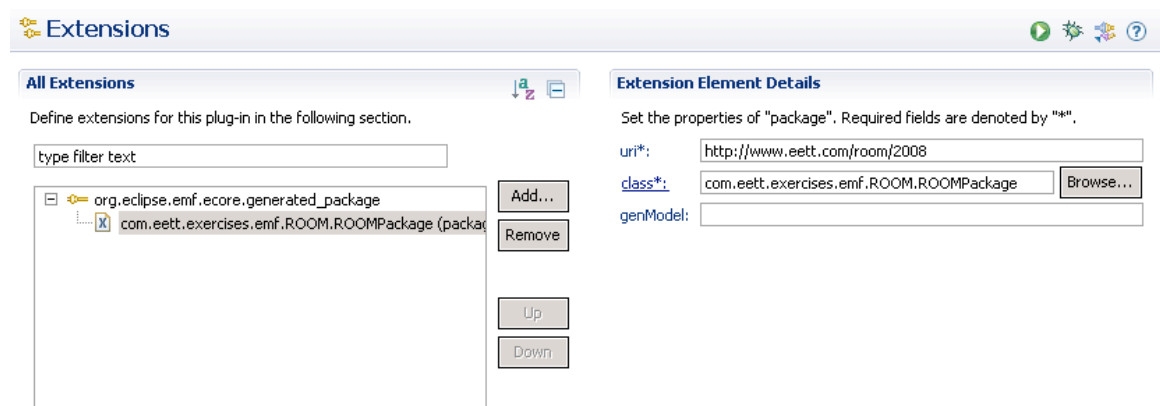


If there are errors in the three other projects we need to update the plug-in manifest for the com.eett.exercises.emf. Open the MANIFEST.MF file in the Plug-in Editor and change to the Runtime page. We need to export the com.eett.exercises.emf.ROOM and com.eett.exercises.emf.ROOM.util packages that were generated because the .edit, .editor and .test plug-ins are dependent on them.



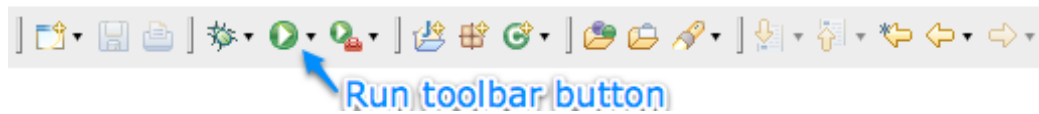
```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Emf
Bundle-SymbolicName: com.eett.exercises.emf;singleton:=true
Bundle-Version: 1.0.0
Require-Bundle: org.eclipse.emf.ecore
Export-Package: com.eett.exercises.emf.ROOM,
               com.eett.exercises.emf.ROOM.util
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

We need to change the registration of the metamodel to point to the generated package. Comment out the org.eclipse.emf.ecore.dynamic\_package extension in plugin.xml and add a new extension org.eclipse.emf.ecore.generated\_package. Set the URI to be <http://www.eett.com/room/2008> and the class to com.eett.exercises.emf.ROOM.ROOMPackage.

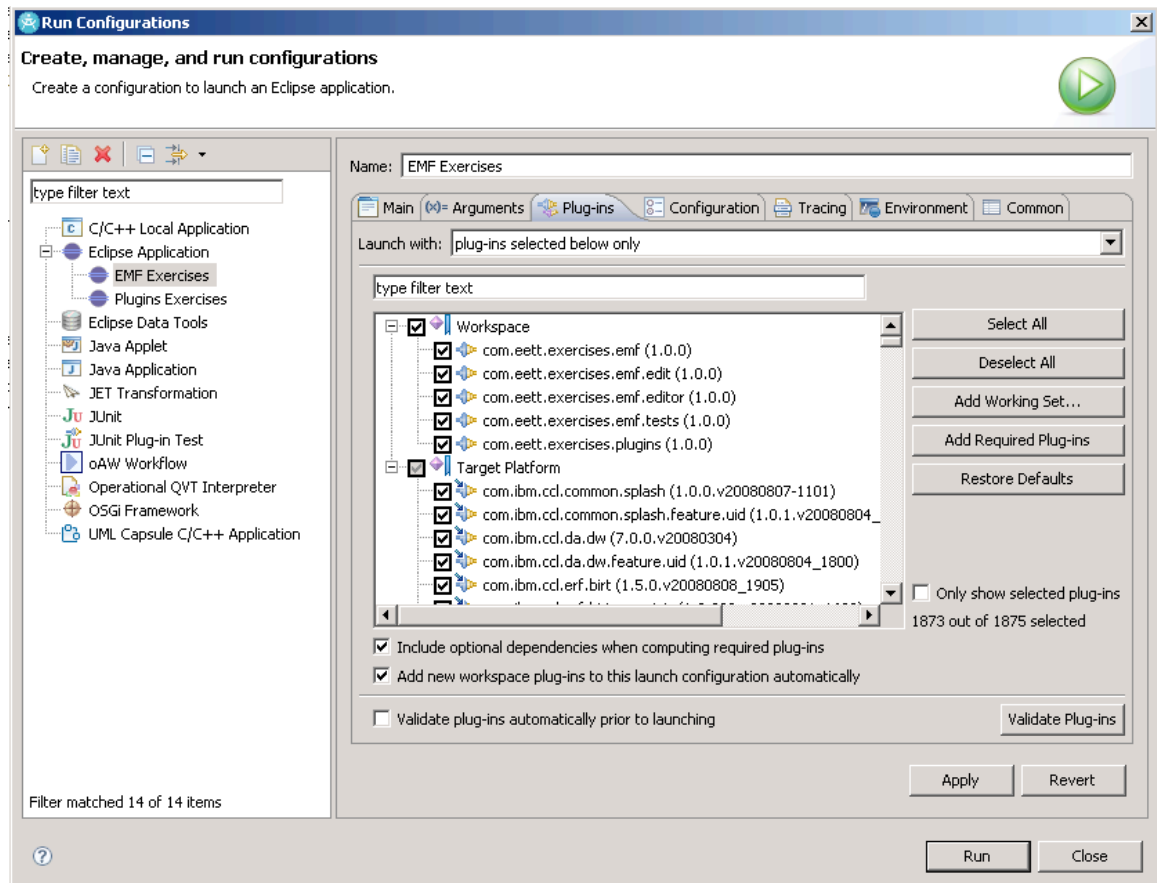


For now we won't worry about the code that was generated, instead we will create a model using the generated model editor.

Switch to the Java perspective and open the Run Configurations dialog by selecting Run → Run Configurations... from the main menu or by using the toolbar.



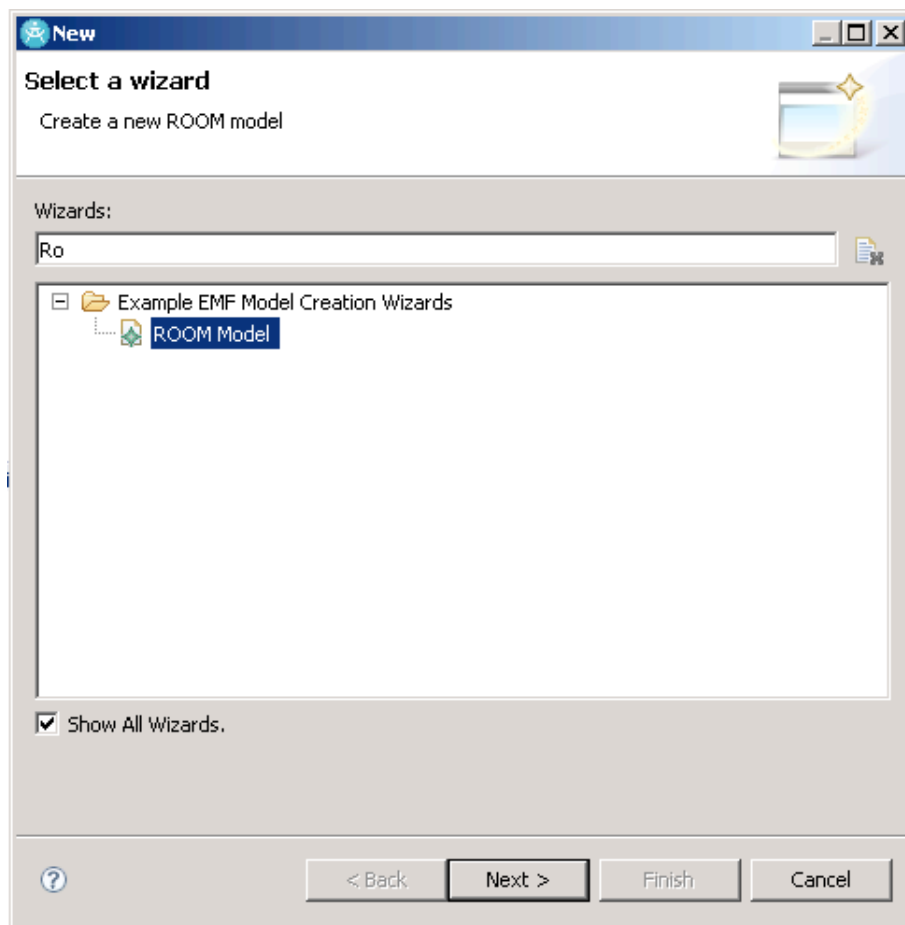
Select EMF Exercises Run Configuration by selecting the Eclipse Application entry on the left and make sure that our new plug-ins are included.



Now click on the Apply button followed by the Run button, this should launch a Runtime Workbench that will enable you to use the editor that was generated for the ROOM metamodel.



To create a ROOM model select the models folder in our com.eett.exercises.emf.pluginlets project and File → New → Other... In the New wizard select the ROOM Model wizard and click Next >.



Set the name of the model to DyeingController.room and click Next >. In the ROOM Model wizard set the Model Object to Actor and click Finish.

This will create a new resource with an Actor object in it and open it in the ROOM Editor, which is customized to the ROOM metamodel. Notice that the shortcoming of our current metamodel is that we have no top container element allowing us to create other elements of our model in the same resource.

Exit the runtime workbench and modify the metamodel to include a Model EClass that can contain Actors and Protocols. Reload the generator model (in the EMF Generator editor, Generator → Reload... from the main menu.) Regenerate the model and re-launch the runtime workbench to be able to create a ROOM Model with Model as the Model Object.

### Possible improvements

- Change the Wizard category of the ROOM Model in the wizard
- By changing the \_UI\_Wizard\_category property in the plugin.properties file
- Add a perspective where the ROOM Model is listed in the New Menu with having to go through the Select a wizard

## 2.4 Implement a derived attribute and operation

In this exercise we will add a derived attribute and an operation into our model and then provide the logic for them. An Actor has a reference to Port and a port has different attributes, one of which is visibility. We will consider all public Ports to be interface ports, so we can derive the list of interface ports from the 'ports' existing ports reference. Similarly, we may want to have an operation on an Actor for creating an interface port, given a name.

Open the room.ecore model in the Sample Ecore Model Editor and modify the Actor EClass adding

## 2.5 Querying our model

In this exercise you will create a plug-in that extends the ROOM Model Editor with a query that will find all Actor's that do not have a actorSuperclass. In order to do this we will add an action to the model editor that invokes the query we will write.

Start by creating a new plug-in project with

ID	com.eett.exercises.emf.query,
Name	EETT EMF Query Exercise
Provider	EETT
Version	1.0.0
This plug-in is a singleton	<input checked="" type="checkbox"/>

The screenshot shows the Eclipse Plug-in Editor interface. At the top, the browser tab is labeled 'com.eett.exercises.emf.query'. Below it is the 'Overview' section. Under 'General Information', there is a description: 'This section describes general information about this plug-in.' followed by fields for ID (com.eett.exercises.emf.query), Version (1.0.0), Name (EETT EMF Query Exercise), Provider (EETT), Platform Filter, and Activator (with a 'Browse...' button). Two checkboxes are present: 'Activate this plug-in when one of its classes is loaded' (unchecked) and 'This plug-in is a singleton' (checked). The 'Execution Environments' section has a description: 'Specify the minimum execution environments required to run this plug-in.' and a list containing 'JavaSE-1.6' with buttons for 'Add...', 'Remove', 'Up', and 'Down'. Below this are links for 'Configure JRE associations...' and 'Update the classpath settings'. At the bottom, a series of tabs are visible: Overview, Dependencies, Runtime, Extensions, Extension Points, Build, and MANIFEST.MF.

Switch to the Dependencies page of the Plug-in Editor and add the following dependencies

- org.eclipse.ui
- org.eclipse.core.runtime
- org.eclipse.emf.ecore
- org.eclipse.emf.query
- com.eett.exercises.emf
- com.eett.exercises.emf.editor

In order for the query to be accessible from the ROOM Editor we need to add an extension to the editor. Switch to the Extensions page of the Plug-in Editor and click the Add... button in the All Extensions section.

From the Extension Point Selection dialog select the org.eclipse.ui.editorActions extension point (you may have to uncheck the 'Show only extension points from the required plug-ins' option if you haven't all the dependencies listed above.)

If an editorContribution isn't automatically added for you, select the extension point and New → editorContribution from its context menu. Set the details for the editorContribution to:

<b>id</b>	com.eett.exercises.emf.query.editorContribution
<b>targetID</b>	com.eett.exercises.emf.ROOM.presentation.ROOMEditorID

The targetID field specifies the identifier that was generated for the ROOM editor, this is necessary for the workbench to know the specific editor that we are contributing to. A list of available editors can be found by clicking the Browse... button beside the field.

Add a menu to the editorContribution through its context menu and set its details as shown in the following table. The path field identifies the menu that we are contributing to in the ROOM editor, this id is defined in the ROOM editor plug-in that was generated from the ROOM EMF model. Add a separator to this menu called additions.

<b>id</b>	com.eett.exercises.emf.queryMenuID
<b>label</b>	&Query
<b>path</b>	com.eett.exercises.emf.ROOMMenuID/additions

Add an action to the editorContribution through its context menu and set its details as shown in the following table. We are adding this action to the menu that we defined in the previous step, as referenced in the menubarPath. In the next step we will create the class we have defined in the class field.

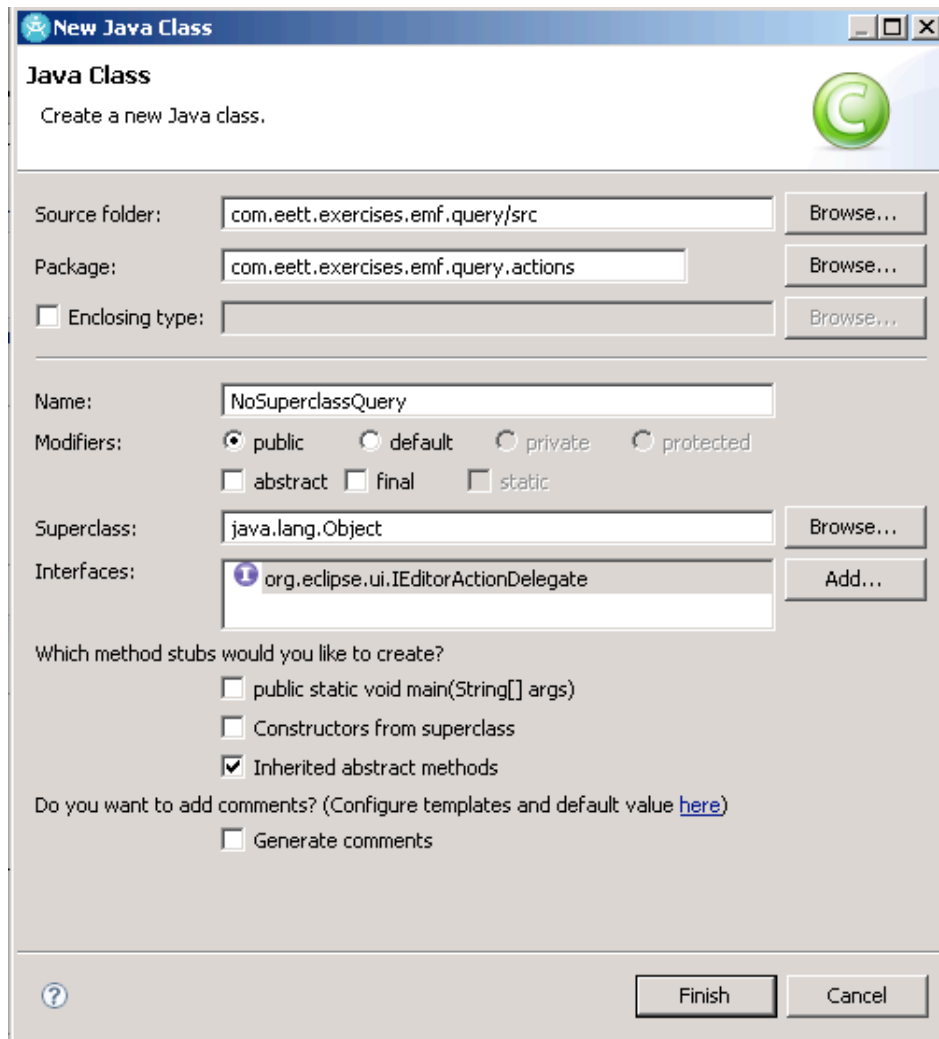
<b>id</b>	com.eett.exercises.emf.query.NoSuperclassQuery
<b>label</b>	Actors with No Superclass
<b>class</b>	com.eett.exercises.emf.query.actions.NoSuperclassQuery
<b>menubarPath</b>	com.eett.exercises.emf.ROOMMenuID/ com.eett.exercises.emf.queryMenuID/additions

We do not want the action to be enabled for anything so we will add an enablement to our action to control when it can be invoked. Add an enablement with an objectClass through the action's context menu and set the name to org.eclipse.emf.ecore.EObject which will restrict the enablement to only objects of type EObject.

The plugin.xml page should now resemble:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
    point="org.eclipse.ui.editorActions">
    <editorContribution
      id="com.eett.exercises.emf.query.editorContribution"
      targetID="com.eett.exercises.emf.ROOM.presentation.ROOMEditorID">
      <menu
        id="com.eett.exercises.emf.queryMenuID"
        label="&Query"
        path="com.eett.exercises.emf.ROOMMenuID/additions">
        <separator name="additions" />
      </menu>
      <action
        class="com.eett.exercises.emf.query.actions.NoSuperclassQuery"
        id="com.eett.exercises.emf.query.NoSuperclassQuery"
        label="Actors with No Superclass"
        menubarPath="com.eett.exercises.emf.ROOMMenuID/
com.eett.exercises.emf.queryMenuID/additions"
        style="push">
        <enablement>
          <objectClass name="org.eclipse.emf.ecore.EObject"></objectClass>
        </enablement>
      </action>
    </editorContribution>
  </extension>
</plugin>
```

We will now create the handler for our query action in the src folder of the plugin add a Class using the wizard (New → Class). Set the name to NoSuperclassQuery and the package to com.eett.exercises.emf.query.actions. The handler must implement the IEditorActionDelegate.



In the newly created class there are two methods that we are interested in implementing `setActiveEditor` and `run`, we will also add an additional operation to perform the actual query.

The `setActiveEditor` will allow us to capture the current editor and to access the selected model element to use as the source for our query. To support the method add an editor field to the class of type `ROOMEditor`. The behavior of `setActiveEditor` follows.

```
@Override
public void setActiveEditor(IAction action, IEditorPart targetEditor)
{
    if(editor instanceof ROOMEditor) {
        editor = (ROOMEditor) targetEditor;
    }
}
```

The run action will invoke the query that we will write and then select the objects returned by the query in the editor. The behavior of run follows.

```
@Override
public void run(IAction action) {
    Collection<EObject> selectedObjects
        = new ArrayList<EObject>();

    // get the selected object from the editor
    if(editor != null
        && editor.getSelection() instanceof
IStructuredSelection){

        IStructuredSelection structuredSelection =
            (IStructuredSelection) editor.getSelection();
        for(Object next : structuredSelection.toList()){
            if(next instanceof EObject){
                selectedObjects.add((EObject) next);
            }
        }

        // execute the query
        Collection<EObject> result = performQuery(selectedObjects);

        // select the query results in the editor
        if(!result.isEmpty()){
            editor.setSelectionToViewer(result);
        }
    }
}
```

Finally, we write a query that takes the selected element in the editor and find all the Actors that do not have the actorSuperclass reference set.

```
private Collection<EObject> performQuery
(Collection<EObject> selectedObjects) {

    // create a condition that tests whether an Actor's
    // actorSuperclass feature to see if it is null
    EObjectCondition condition = new
EObjectReferenceValueCondition(
    ROOMPackage.eINSTANCE.getActor_ActorSuperclass(),
    EObjectInstanceCondition.IS_NULL);

    SELECT query = new SELECT(
        new FROM(selectedObjects),
        new WHERE(condition));

    return query.execute();
}
```

This completes the creation of the plug-in that will perform the query. It needs to be tested in a runtime workbench. Create a runtime workbench including this plug-in and the ones that it requires. Open an existing ROOM model or create a new one using the ROOM Editor, ensure that the ROOM Editor menu contains a query sub-menu.





## 3 UML Module Exercises

### 3.1 UML Model Exercise

#### 3.1.1 What this exercise is about

In this exercise, you will create a UML model of a dyeing system. The system that consists of a dye tank that has a fill valve and a drain valve, the tank has a high sensor to determine when it is filled to its maximum and low sensor to determine when it is getting low on dye. A controller receives signals from the sensors in order to open and close the appropriate valves.

At the end of this exercise, you should be able to:

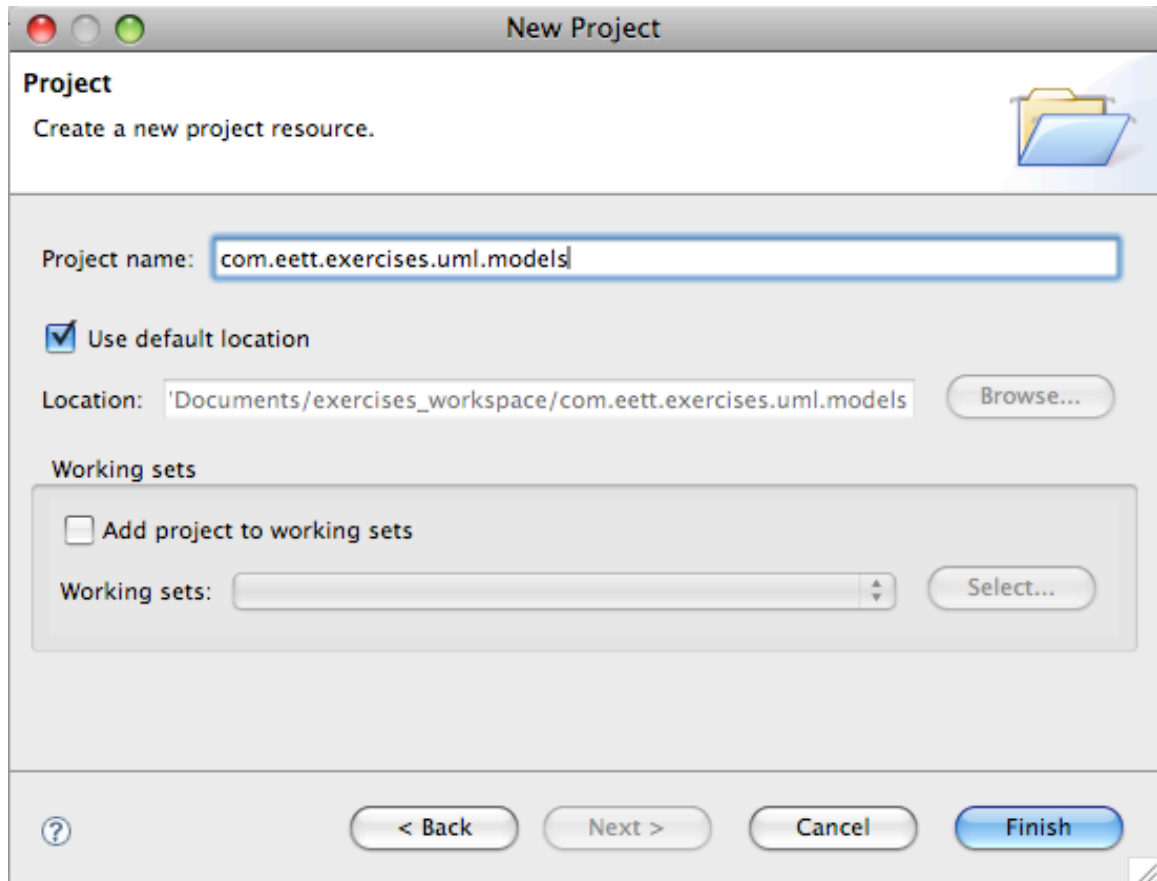
Create a UML model using the UML Model Editor, and

Programmatically create UML model elements

#### 3.1.2 Exercise Instructions

Start by creating a project that will contain the models that we create in this exercise and others.

1. Create a new basic Eclipse project
  1. Select File → New → Project... → Project
  2. Set the Project name to com.eett.exercises.uml.models
  3. Keep Use default location checked
  4. Click Finish >

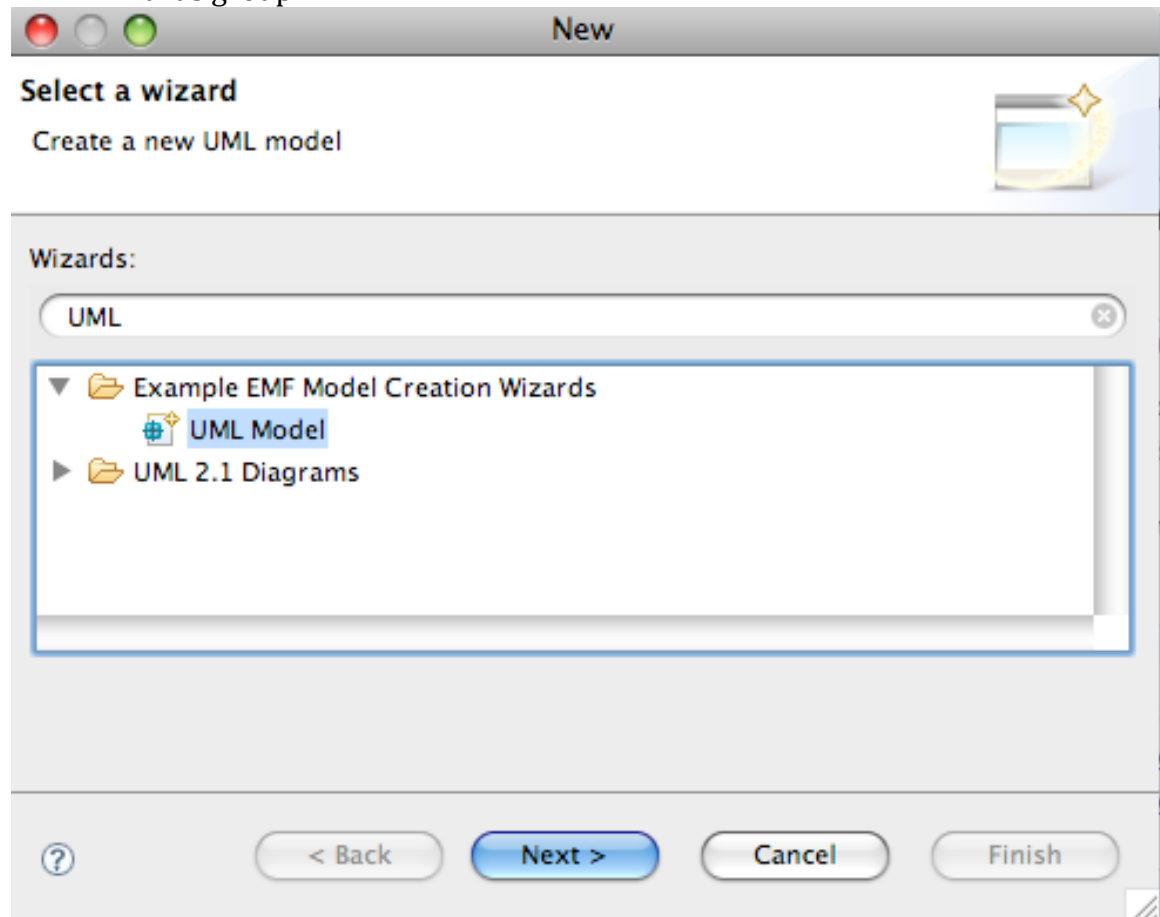


---

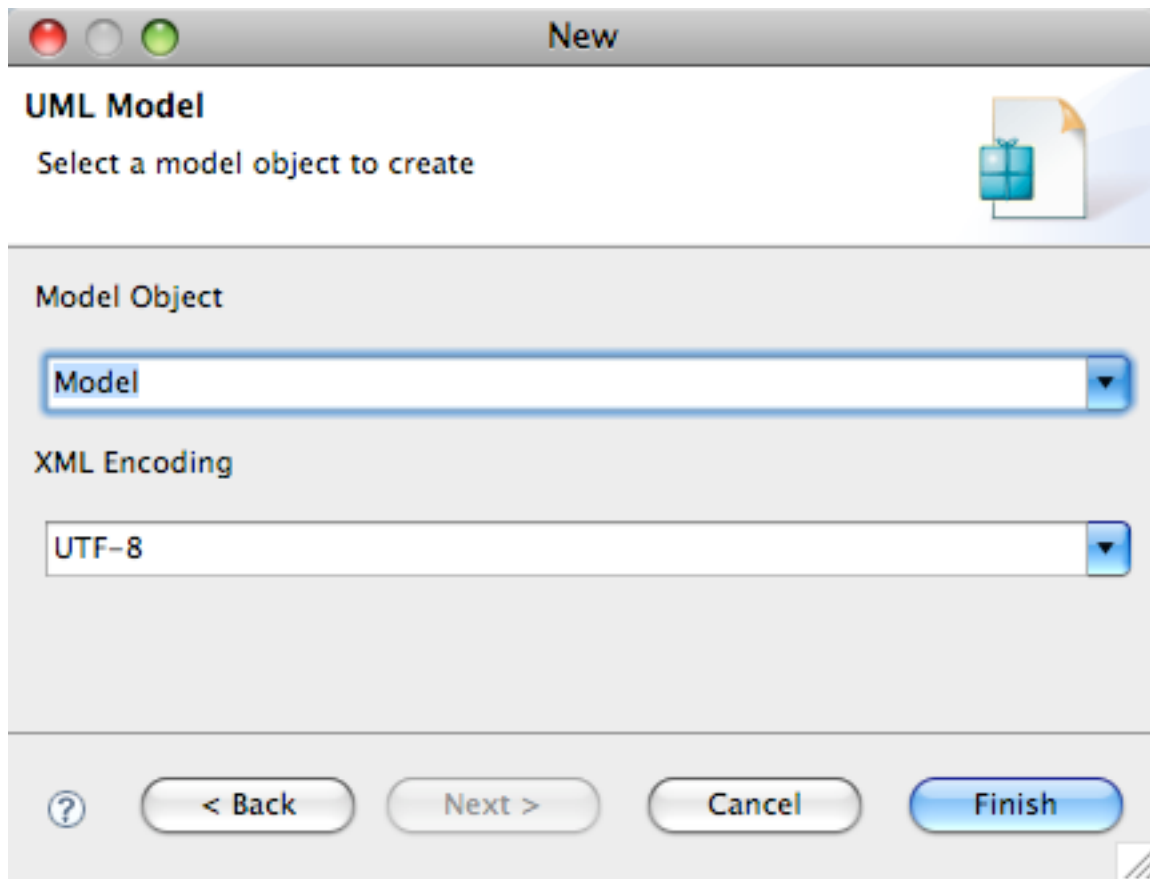
*You should now have an empty project in your workspace*

---

2. Use the project context menu we will add a folder to store our models in.
  1. Select the project and New → Folder from its context menu.
  2. Set the Folder name to models and click Finish
3. We will now create a UML model in our project and populate it with our dyeing system model
  1. Select the models folder and New → Other... from its context menu.
  2. In the New wizard select UML Model in the Example EMF Model Creation Wizards group.



3. Set the File Name to dyeingSystem.uml and click Next >
4. Set the Model Object field to Model and leave the XML Encoding field to the default value and click Finish



*The project in your workspace should now contain a `dyeingSystem.uml` file, that you can open using the UML Model Editor.*

## 3.2 UML Keywords Exercise

### 3.2.1 What this exercise is about

In this exercise, you will create a UML model using the UML Model Editor and tag the elements in the model using key words. You will also extend the UML model editor with actions to automatically add specific keywords to model elements. At the end of this exercise, you should be able to:

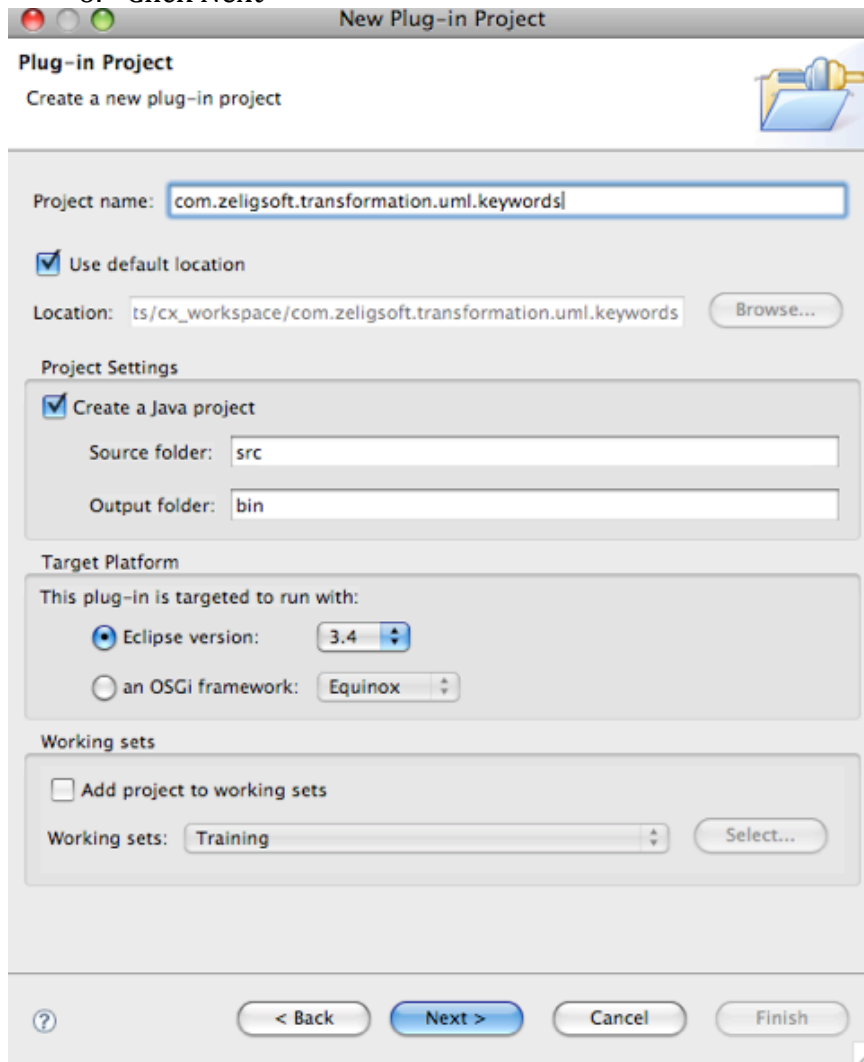
- Set key words in a UML model using the UML Model Editor
- Add actions to the UML Model Editor to tag elements with key words

### 3.2.2 Exercise Instructions

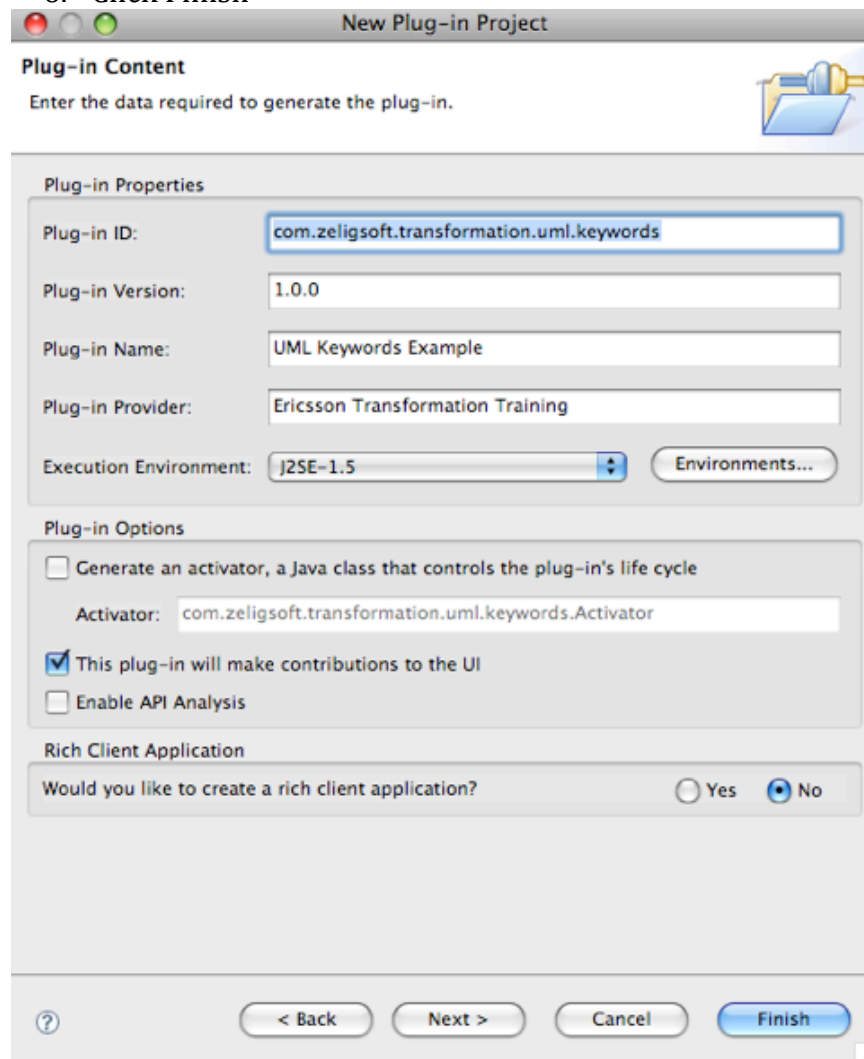
Start by creating a project with the name `com.eett.exercises.uml.keywords` in the workspace.

4. Switch to the Plug-in Development perspective, if not already there.
  1. Select Window → Open Perspective → Other... → Plug-in Development

5. Create a new Plug-in project as we will be adding actions to an editor
  1. Select File → New → Project... → Plug-in Project
  2. Set the Project name to com.eett.exercises.uml.keywords
  3. Keep Use default checked
  4. Keep Create a Java project checked
  5. Make sure that the Eclipse version is set to 3.4
  6. Click Next >

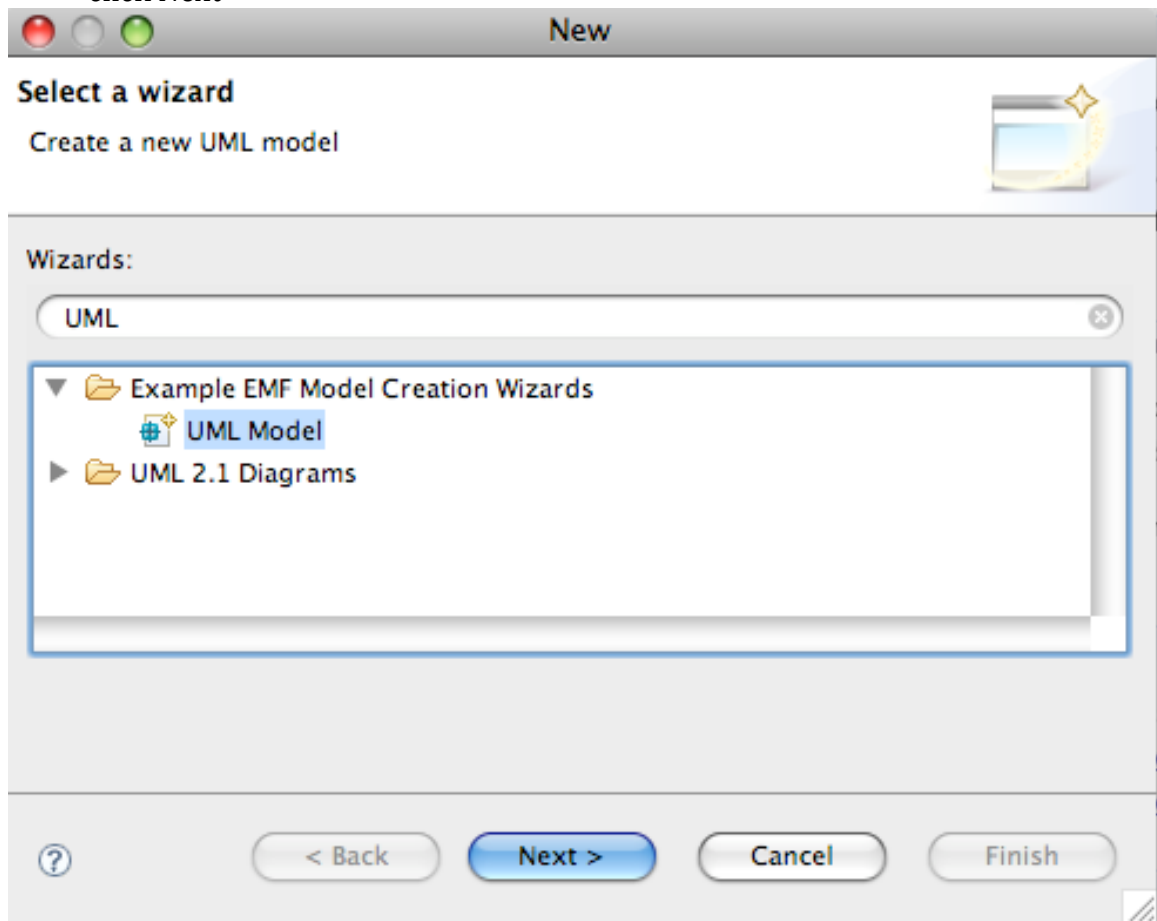


6. Configure the Plug-in Content dialog
  1. Keep the defaults for Plug-in ID and Plug-in Version
  2. Set the Plug-in Name to UML Keywords Example
  3. Set the Plug-in Provider to EETT
  4. Make sure that "This plug-in will make contributions to the UI" is the only Plug-in Option checked
  5. Make sure that Would you like to create a rich client application? Is set to No
  6. Click Finish

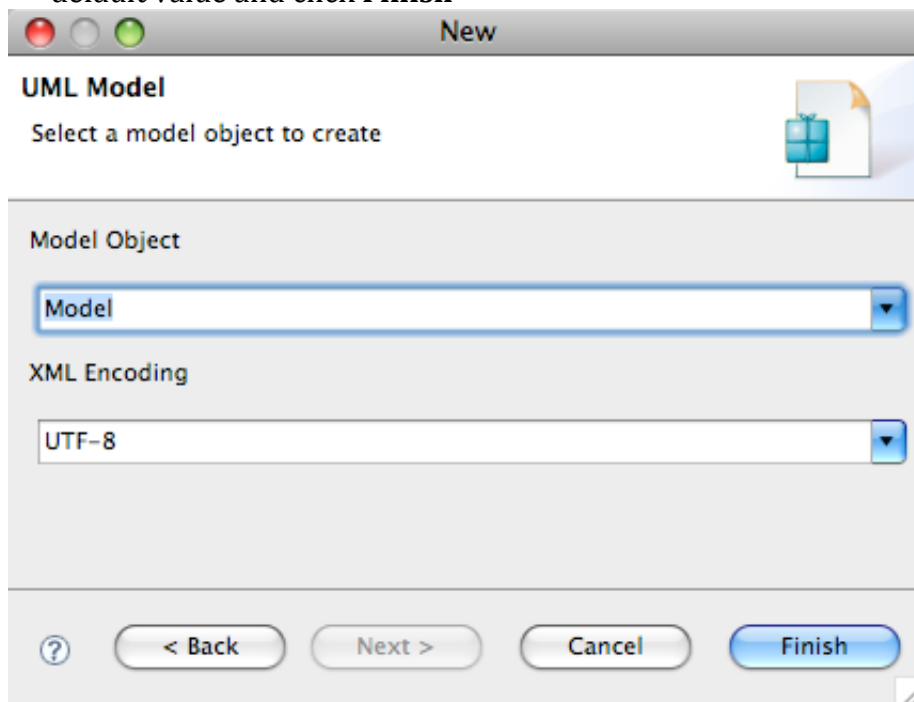


7. Create a UML model named dyeingSystem.uml in which we will build a UML model of a Dyeing system that consists of a dye tank that has a fill valve and a drain valve, the tank has a high sensor to determine when it is filled to its maximum and low sensor to determine when it is getting low on dye. A controller receives signals from the sensors in order to open and close the appropriate valves.
  1. Select the project created in the last step and New → Folder from the context menu
  2. Change the name of the folder to models

3. Select the models folder and File → New → Other... → UML Model and click Next >



7. Set the File Name to dyeingSystem.uml and click **Next >**
8. Set the Model Object field to Model and leave the XML Encoding field to the default value and click **Finish**



### 3.3 UML Profiles Exercise

#### 3.3.1 What this exercise is about

In this exercise, you will create a Profile defining a ROOM like model. It will allow us to define, in UML, models that use the ROOM concepts. This includes for example identifying classes as ROOMActors and ports as ROOMPorts.

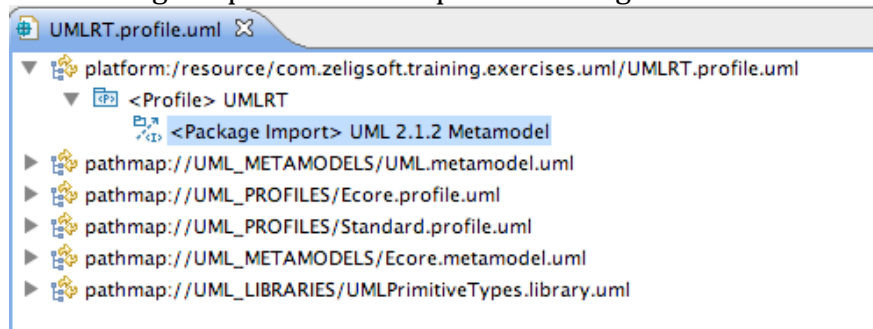
#### 3.3.2 What you should be able to do

At the end of this exercise, you should be able to:

- Define a UML profile using the UML Model Editor
- Register and publish the profile to make it available at run-time
- Apply and use the profile in a UML model
- Add actions to the UML Model Editor to create elements with the stereotypes applied

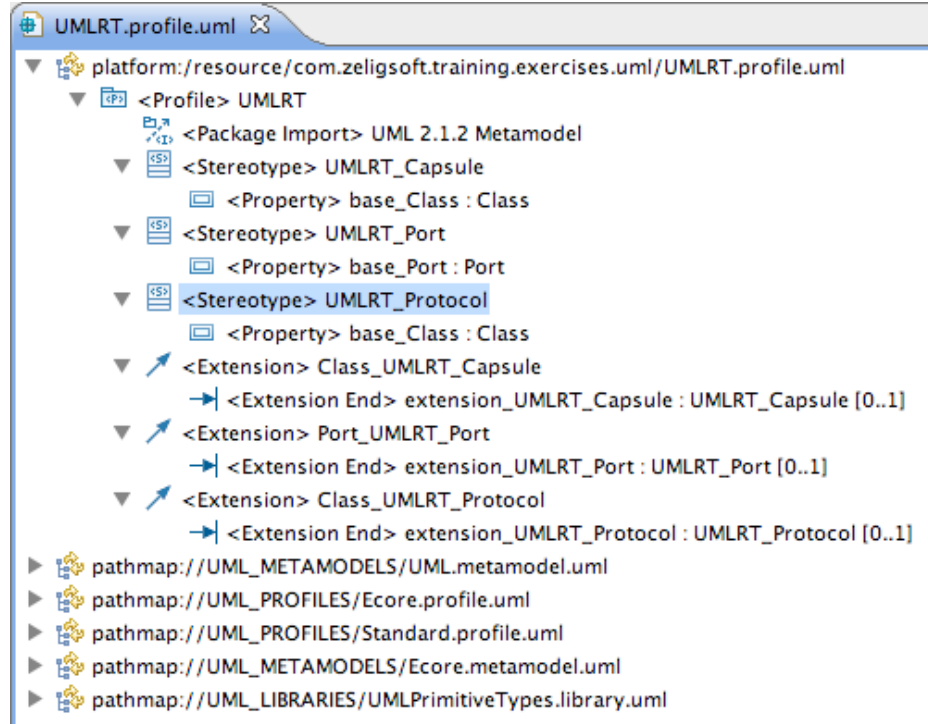
#### 3.3.3 Exercise instructions

1. Select the **com.eett.exercises.uml** project and then choose File | New | Other...
2. Select UML Model in the Example EMF Model Creation Wizards folder and click the Next > button
3. Enter UMLRT.profile.uml as the file name and click the Next > button
4. Select Profile as the Model Object and click the Finish button
5. Open the new model file with the UML Model Editor and change the name of the Profile element to UMLRT
6. Load Resource, pathmap://UML\_METAMODELS/UML.metamodel.uml
7. Add a Package Import and set Imported Package to uml

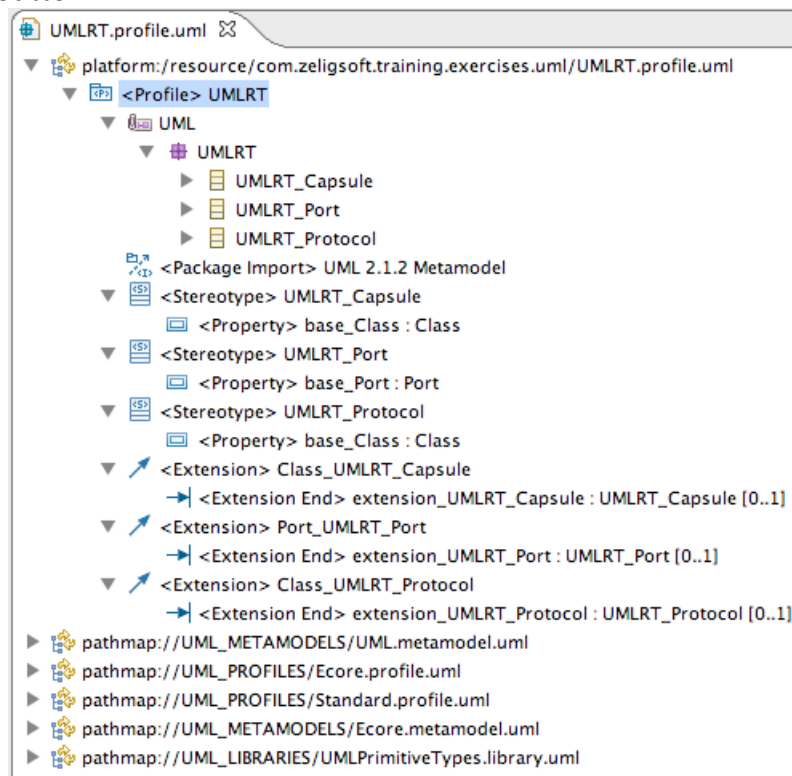


8. In the Profile create Stereotypes
  - a. UMLRT\_Capsule
  - b. UMLRT\_Port
  - c. UMLRT\_Protocol

9. For each of the Stereotypes add a metaclass extension, select the Stereotype and select UML Editor | Stereotype | Create Extension ... from the main menu and select the appropriate metaclass
  - a. UMLRT\_Capsule – uml::Class
  - b. UMLRT\_Port – uml::Port
  - c. UMLRT\_Port – uml::Port



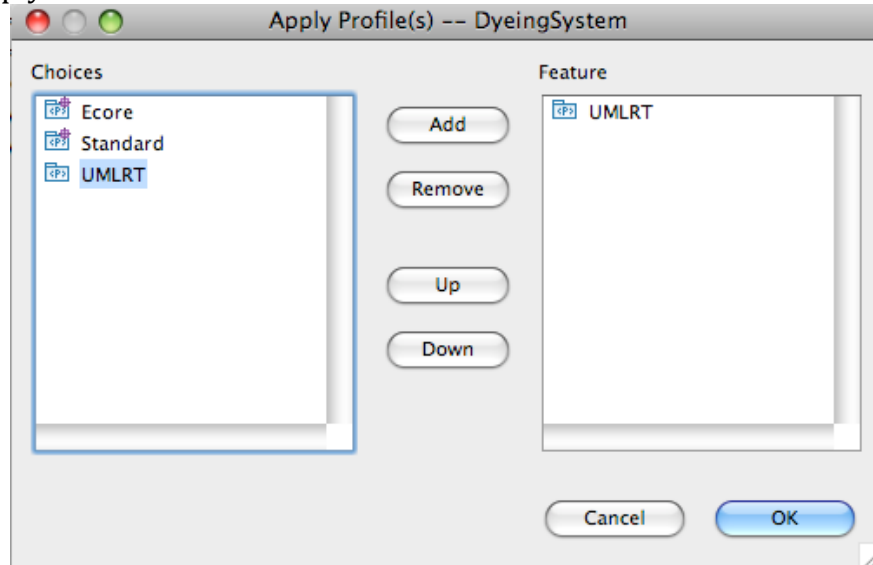
10. Make sure the UMLRT Profile element is selected in the model and then select UML Editor | Profile | Define, keeping the default options click the Ok button





### 3.4 Applying the Profile to a Model

1. Open the DyeingSystem.uml model with the UML Editor
2. Select UML Editor | Load Resource from the main menu
3. Enter UMLRT.profile.uml as the resource location and click the Ok button
4. Select the UMLRT Model element in the editor and UML Editor | Package | Apply Profile ...



5. Select Valve Class in the model and UML Editor | Element | Apply Stereotype ... and choose the UMLRT\_Capsule Stereotype
  - a. Repeat for DyeingSystemController
  - b. Repeat for DyeingSystem
6. Select the Ports on each of the elements and apply the UMLRT\_Port stereotype
7. Select the ValveControl Class and apply the UMLRT\_Protocol stereotype

## 4 Transformation Exercises

### 4.1 M2M Setup

In order to execute transformations from the ROOM Editor we need to add an extension to the editor. Create a new Plug-in project.

<b>ID</b>	com.eett.exercises.m2m.ui
<b>Version</b>	1.0.0
<b>Name</b>	EETT Transformations UI Exercise
<b>Provider</b>	EETT

Add the following dependencies.

- org.eclipse.core.runtime
- org.eclipse.ui
- com.eett.exercises.emf
- com.eett.exercises.emf.editor

Switch to the Extensions page of the Plug-in Editor and click the Add... button in the All Extensions section.

From the Extension Point Selection dialog select the org.eclipse.ui.editorActions extension point (you may have to uncheck the 'Show only extension points from the required plug-ins' option if you haven't all the dependencies listed above.)

If an editorContribution isn't automatically added for you, select the extension point and New → editorContribution from its context menu. Set the details for the editorContribution to:

<b>id</b>	com.eett.exercises.emf.query.editorContribution
<b>targetID</b>	com.eett.exercises.emf.ROOM.presentation.ROOMEditorID

The targetID field specifies the identifier that was generated for the ROOM editor, this is necessary for the workbench to know the specific editor that we are contributing to. A list of available editors can be found by clicking the Browse... button beside the field.

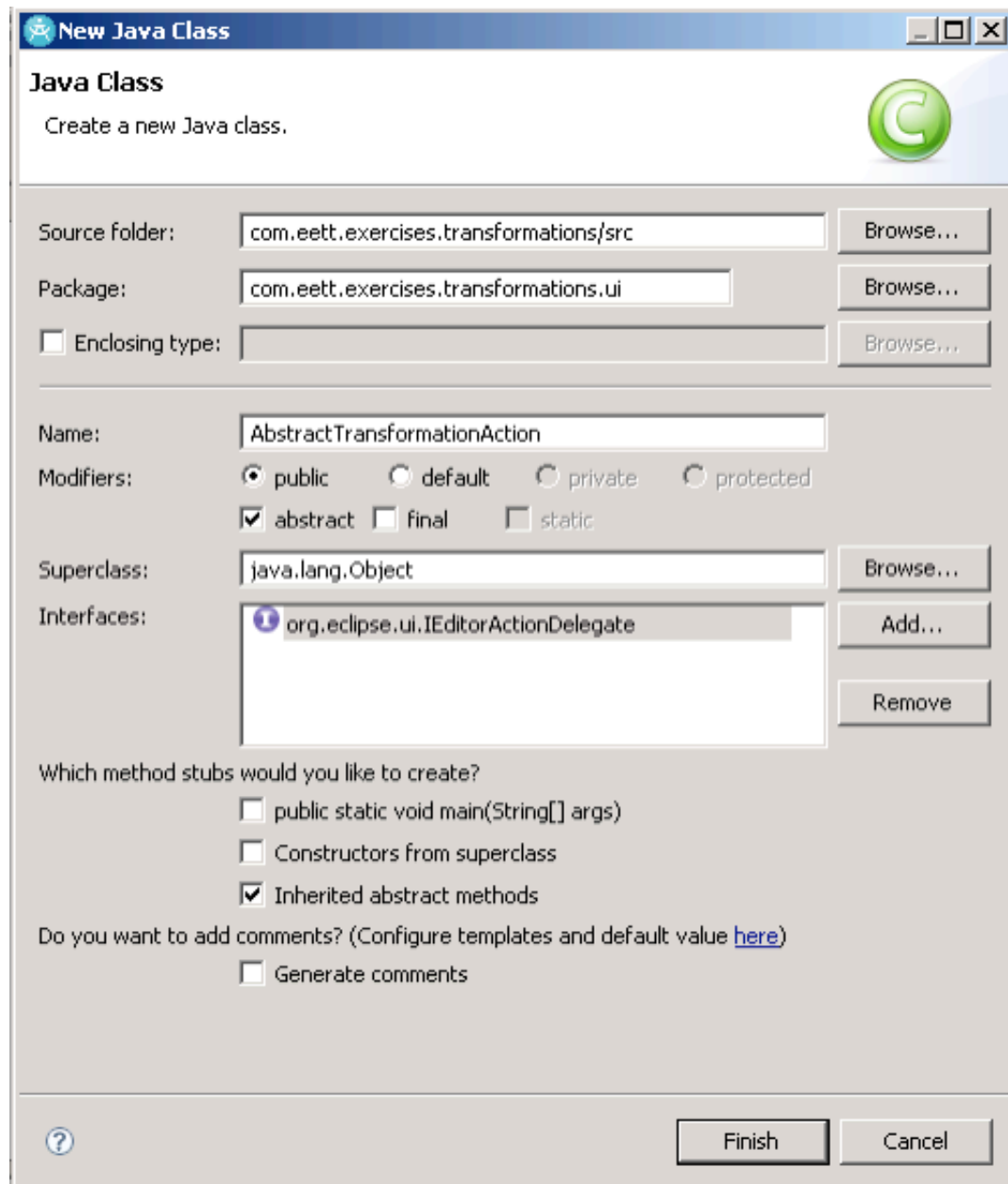
Add a menu to the editorContribution through its context menu and set its details as shown in the following table. The path field identifies the menu that we are contributing to in the ROOM editor, this id is defined in the ROOM editor plug-in that was generated from the ROOM EMF model. Add a separator to this menu called additions.

<b>id</b>	com.eett.exercises.transformations.MenuID
<b>label</b>	&Transform
<b>path</b>	com.eett.exercises.emf.ROOMMenuID/additions

The plugin.xml page should now resemble:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension point="org.eclipse.ui.editorActions">
    <editorContribution
      id="com.eett.exercises.transformation.editorContribution"
      targetID="com.eett.exercises.emf.ROOM.presentation.ROOMEditorID">
      <menu
        id="com.eett.exercises.transformation.MenuID"
        label="&Query"
        path="com.eett.exercises.emf.ROOMMenuID/additions">
        <separator name="additions" />
      </menu>
    </editorContribution>
  </extension>
</plugin>
```

We will now create an abstract handler for our transformation actions in the src folder of the plugin add an abstract Class using the wizard (New → Class). Set the name to AbstractTransformationAction and the package to com.eett.exercises.transformations.actions. The handler must implement the IEditorActionDelegate.



The setActiveEditor method will allow us to capture the current editor and to access the selected model element to use as the source for our transformation. To support the method add an editor field to the class of type ROOMEditor. The behavior of setActiveEditor follows.

```
@Override
public void setActiveEditor(IAction action, IEditorPart targetEditor)
{
    if(editor instanceof ROOMEditor) {
        editor = (ROOMEditor) targetEditor;
    }
}
```

We will add an additional method to the class to return the selected object. If there is more than one object selected it will pick the first one in the list.

```
// Return the object that is currently selected in the active editor
// if more than one is selected return the first one
protected EObject getSelectedObject() {
    if(editor != null
        && editor.getSelection() instanceof IStructuredSelection)
    {
        IStructuredSelection structuredSelection =
            (IStructuredSelection) editor.getSelection();
        for(Object next : structuredSelection.toList()){
            if(next instanceof EObject){
                return (EObject) next;
            }
        }
        return null;
    }
}
```

## 4.2 M2M with Java

In this exercise we will create a model-to-model mapping from our ROOM model to a simplified model of the Java programming language. This requires the java.ecore modeled that will be provided. Before doing the exercises generate and register the model. The remainder of the exercise assumes that you have done this.

Create a new Plug-in Project.

ID	com.eett.exercises.m2m.java
Version	1.0.0
Name	EETT M2M Java Exercise
Provider	EETT

Dependencies

org.eclipse.emf.java

com.eett.exercises.emf

Add an action to the ROOM Editor in the com.eett.exercises.transformations plug-in. Add an action to the editorContribution through its context menu and set its details as shown in the following table. We are adding this action to the menu that we defined in the previous step, as referenced in the menubarPath. In the next step we will create the class we have defined in the class field.

<b>id</b>	com.eett.exercises.transformation.java.transform
<b>label</b>	Java Transform
<b>class</b>	com.eett.exercises.transformations.ui.actions.ROOM2JavaAction
<b>menubarPath</b>	com.eett.exercises.emf.ROOMMenuID/ com.eett.exercises.transformation.MenuID /additions

We do not want the action to be enabled for anything so we will add an enablement to our action to control when it can be invoked. Add an enablement with an objectClass through the action's context menu and set the name to

com.eett.exercises.emf.ROOM.Model, which will restrict the enablement to only objects of type Model. Add a new handler for this action that specializes the AbstractTransformationAction we created in the previous exercise.

In the com.eett.exercises.m2m.java plug-in create a class ROOM2JavaTransformation in the com.eett.exercises.m2m.java package. This is the class we will use as the entry point for our transformation. This class will have a protected constructor and a single method for now, transform that takes a ROOM Model and creates Java JModel.

```
/**
 * A class that is the entry point of a M2M transformation
 * written in Java. Its purpose is to transform a
 * com.eett.exercises.emf.ROOM.Model into a
 * org.eclipse.emf.java.JModel.
 *
 * @author Zeligsoft
 */
public class ROOM2JavaTransform {
    /**
     * We only want one of these transform entry point classes
     * created so we manage the creation of the instance and
     * make it accessible through this static field.
     */
    public static final ROOM2JavaTransform INSTANCE =
        new ROOM2JavaTransform();

    /**
     * Constructor is protected since we do not
     * allow others to create instances.
     */
    protected ROOM2JavaTransform() {

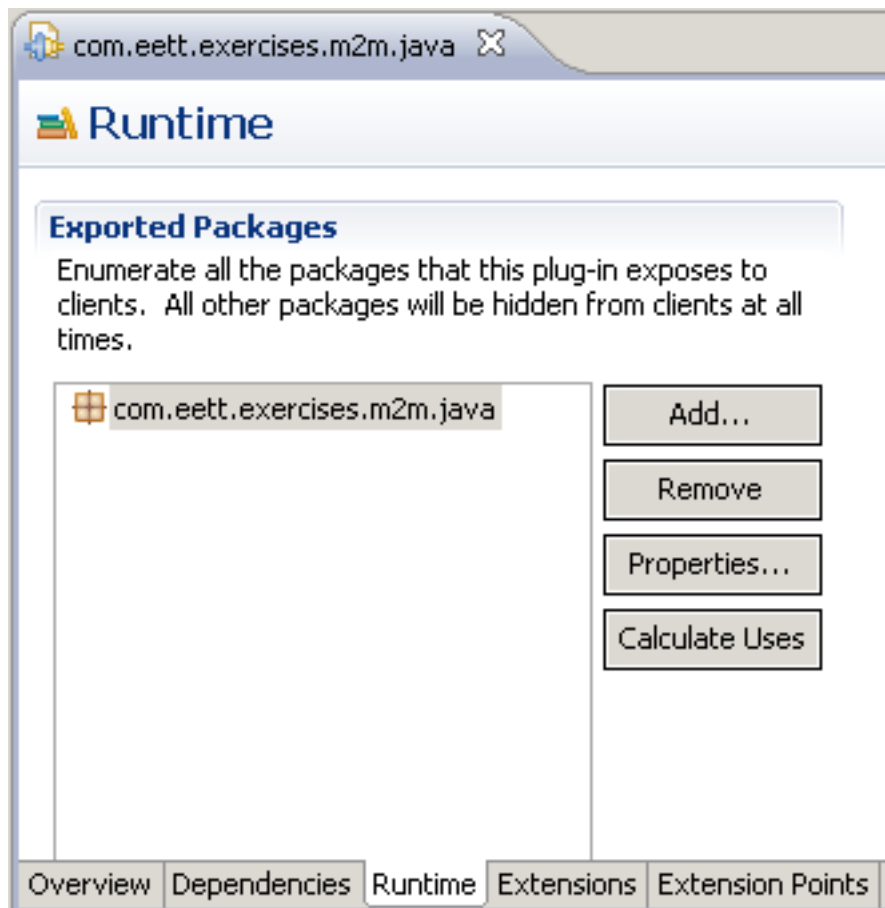
    }

    /**
     * A transformation written in Java that takes as input
     * a ROOM model and transforms it into a Java model.
     */
    public JModel transform(Model roomModel) {
        JModel target = JavaFactory.eINSTANCE.createJModel();

        target.setName(roomModel.getName());

        return target;
    }
}
```

The package containing this class must be exported, in order for you to access it.



We now need to invoke this transformation from the action we have created. This means that the `com.eett.exercises.transforms` plug-in needs a dependency on the Java M2M plug-in. Then we modify the run method in the action to be the following.

TODO

### 4.3 M2M with QVT

- Create the project
- Add the dependencies to the metamodels
- Define the transformation and entry point
- Add a mapping rule for Actor to JClass
- Executing the transformation
- Generating a trace model
- Augment the transformation to include protocols
- Re-run the transformation

### 4.4 M2T with xPand

- Create the project
- Add the dependencies to the metamodels
- Define the transformation and entry point
- Add a mapping rule for JClass
- Build the workflow for testing
- Execute the transformation
- Add an xTend extension

Augment the transformation to include JFields and JMethods  
Re-run the transformation

#### **4.5 M2T with JET**



## 5 Validation Exercises

### 5.1 Working with Validation Framework

This exercise will perform model validation on our ROOM metamodel using the EMF validation framework. More specifically, it will create an EValidator implementation that delegates to the validation framework, to provide user-demand “batch mode” validation from an EMF editor.

### 5.2 Working with Validation in RSA-RTE