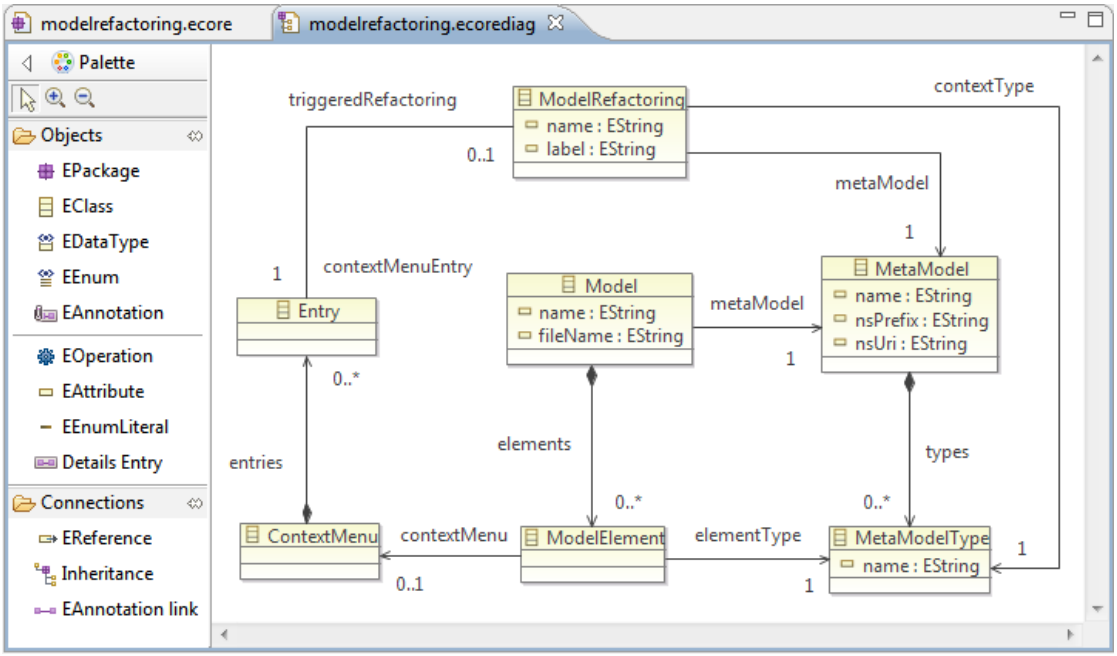
How to generate new EMF model refactorings using Java code

Thorsten Arendt

January 14, 2011

This manual presents the application of an EMF model refactoring using **EMF Refactor**. More precisely, we demonstrate the model refactoring **Move EAttribute** for Ecore models. Please note, that EMF Refactor can be used for refactorings of any models whose meta model is based on EMF Ecore.

Let's take a look to the following Ecore diagram presenting a first model concerning EMF model refactorings in an early stage of the EMF Refactor development process.

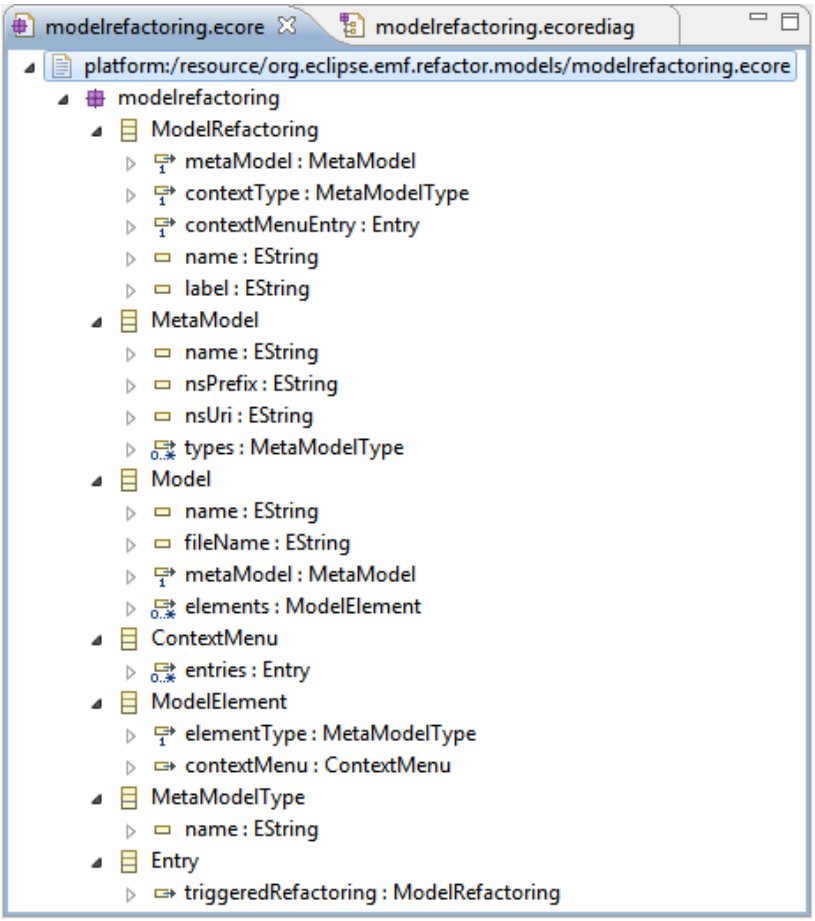


A `ModelRefactoring` has a name and conforms to a `MetaModel` that is specified by name, namespace prefix, and namespace URI. Furthermore, it has a label that should be shown as an `Entry` in the `ContextMenu` of an arbitrary `ModelElement`. A `ModelElement` belongs to a `Model` that is specified by a name and stored in a file with a specific name. Furthermore, a `Model` conforms to a `MetaModel` and each `ModelElement` is typed

over a specific `MetaModelType` belonging to the corresponding `MetaModel`. Besides the afore mentioned attributes, each `ModelRefactoring` is related to a `MetaModelType` representing the type of the contextual element the refactoring can be applied on.

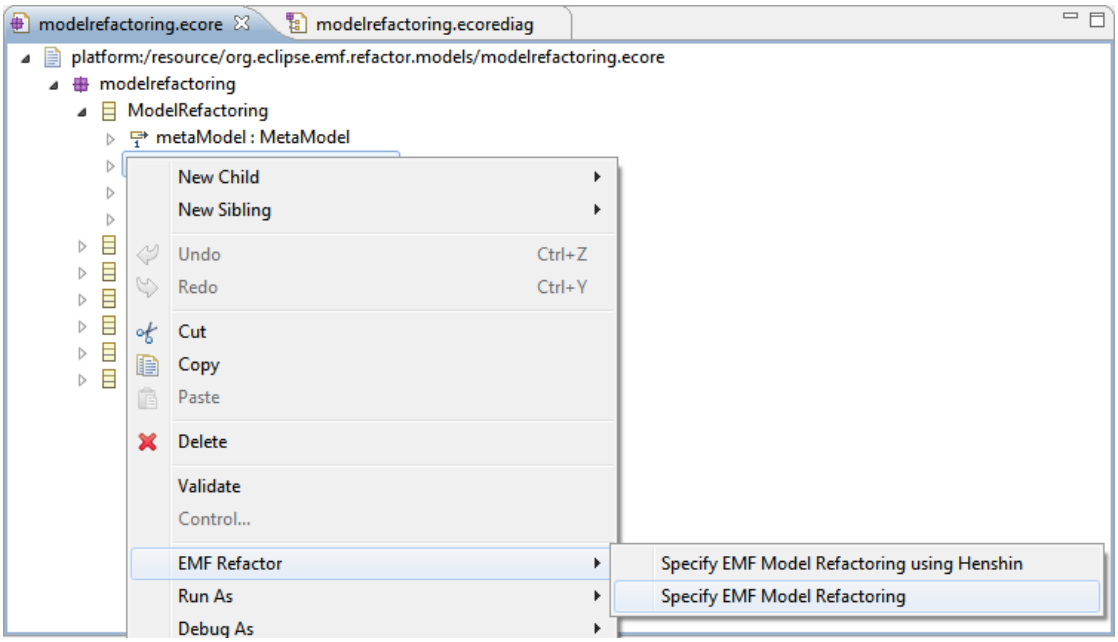
During software design it became questionable whether attribute `label` of class `ModelRefactoring` could be better placed in class `Entry`. So, model refactoring **Move EAttribute** is the next task to be performed.

Since **EMF Refactor** can be used on arbitrary EMF based models the generation of a specific refactoring is mainly triggered from within the EMF instance editor. The next figure shows the example model from above using this tree-based editor.

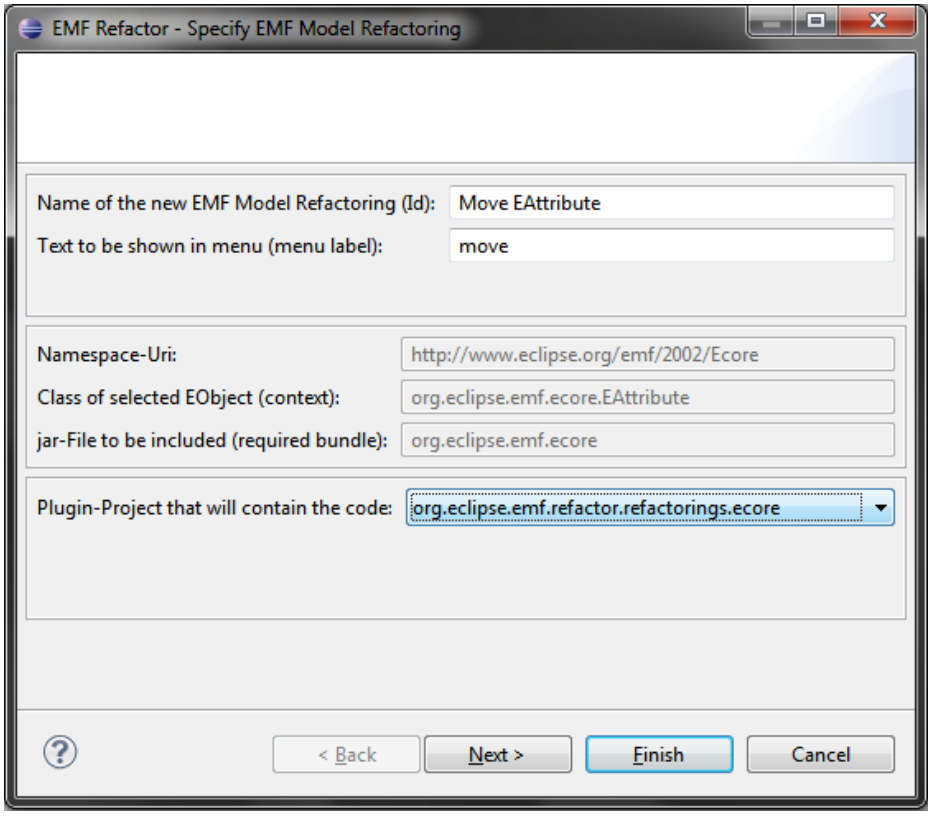


EMF model refactoring **Move EAttribute** can be specified in the following way: First, it has to be checked whether the contextual `EAttribute` is not marked as ID of the containing class, and whether this class has at least one referenced class. If these (initial) checks pass the user has to put in the name of the class the attribute has to be moved to. Then, it has to be checked whether the containing class has a referenced class with the specified name, and whether this class does not already owns an attribute with the same name as the contextual attribute. If these (final) checks pass the contextual attribute can finally be moved to the specified class.

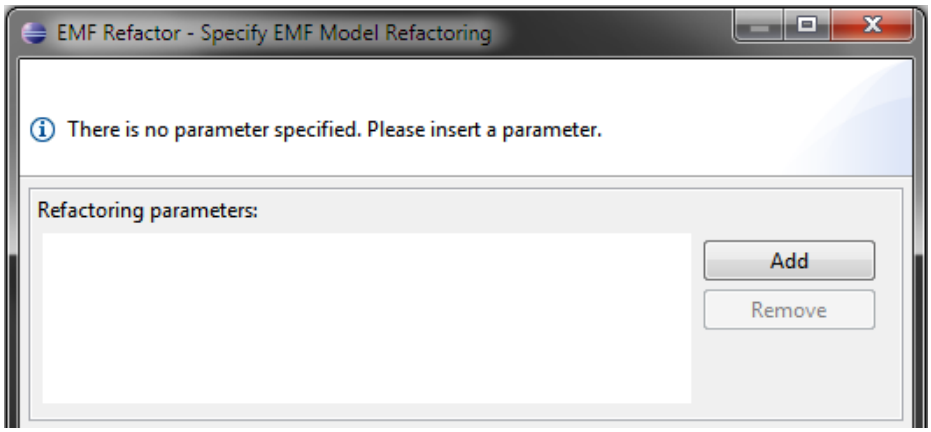
The refactoring specification process can be triggered from within the context menu of a certain model element in the tree-based EMF instance editor. The next figure shows the context menu of an arbitrary `EAttribute` representing the contextual type of our example EMF model refactoring **Move EAttribute**. Here, we select entry `Specify EMF Model Refactoring`.



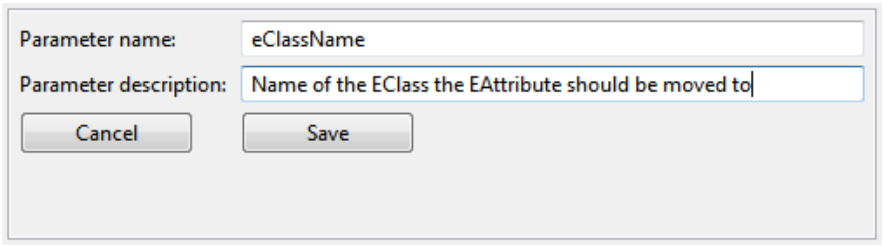
In the first page of the upcoming refactoring generation dialog three refactoring specifics have to be given (see following figure). First, you have to type in the name of the new refactoring. This name also serves as id of the new refactoring. Then, the text of the label has to be specified concerning the context menu entry when triggering the refactoring application. Finally, an Eclipse plug-in project has to be selected in which the corresponding refactoring Java code should be generated to. Further specifics concerning the contextual model element type are set automatically.



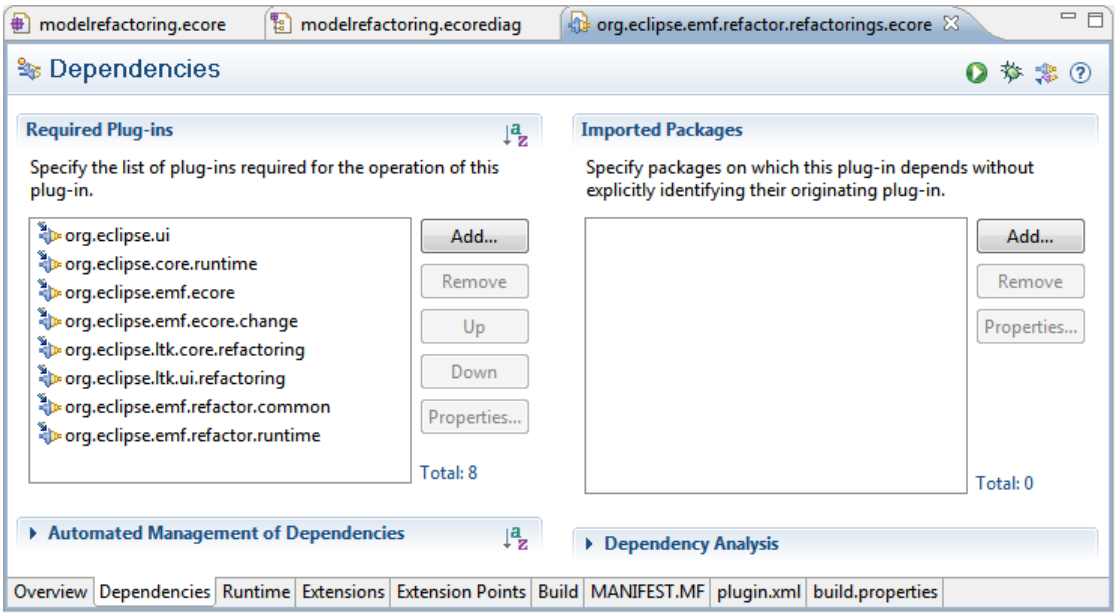
The second page of the refactoring generation dialog specifies the parameters of the corresponding model refactoring. In the upper part of this page you can add new parameters respectively remove existing parameters (see following figure).



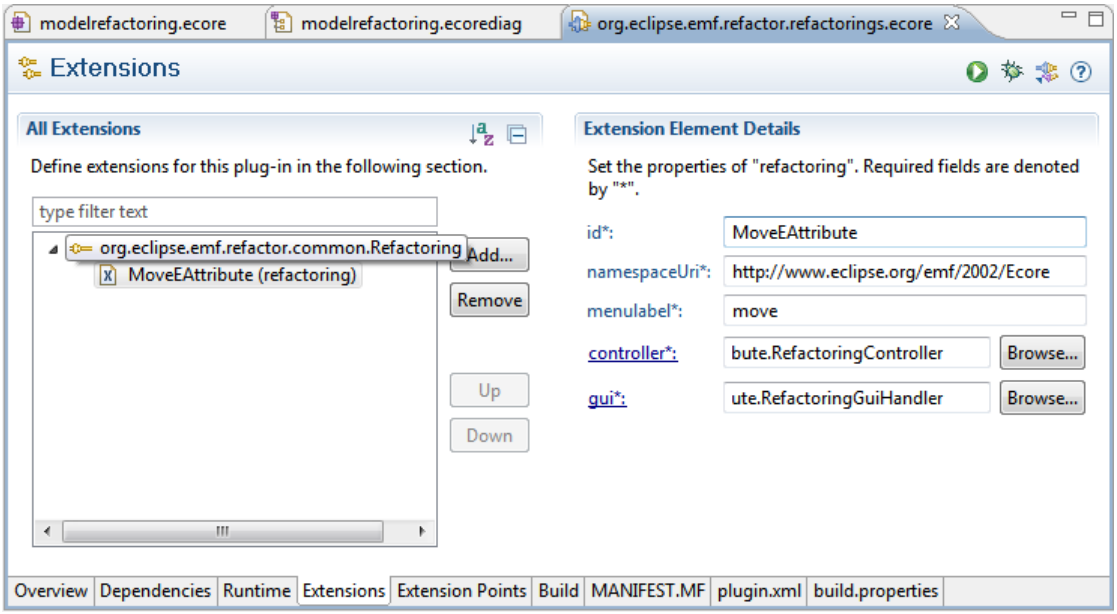
Our example refactoring **Move EAttribute** has one single parameter: the name of the class the attribute has to be moved to. So, we add this parameter as shown in the following figure. Besides the name of the parameter, `eClassName`, you can put in a parameter description that will be used later on in the specific refactoring parameter input dialog.



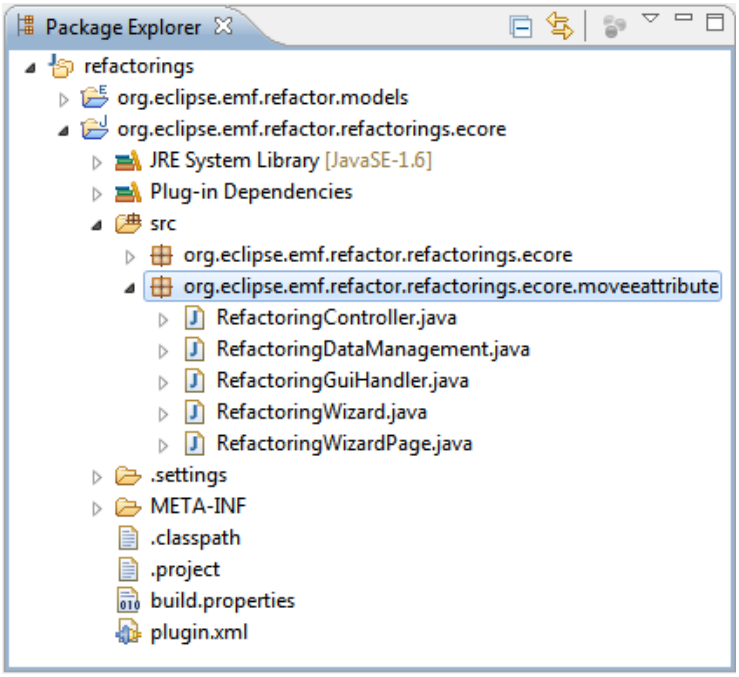
After finishing the refactoring generation dialog, **EMF Refactor** adds some additional information to the selected Eclipse plug-in project. First, **EMF Refactor** adds additionally required plug-in dependencies like shown in the following figure.



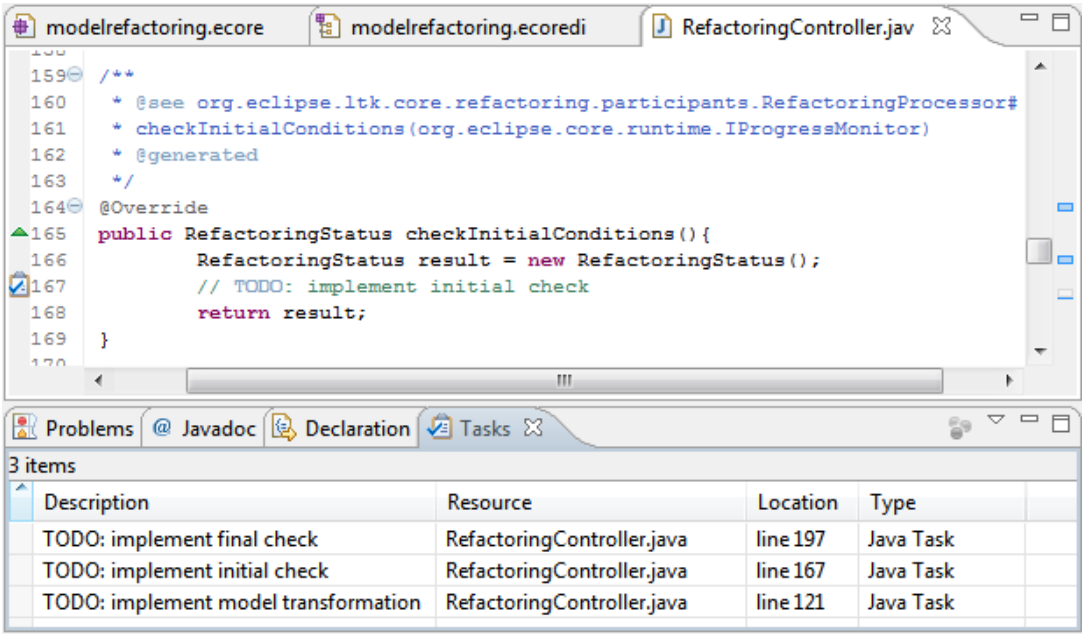
To register the new EMF model refactoring the selected Eclipse plug-in project has to serve a specific extension point, `org.eclipse.emf.refactor.common.Refactoring`, defined by **EMF Refactor**. Besides the given refactoring specifics `id`, `namespaceUri` and `menulabel` additional references to two Java classes are needed. The following figure shows the generated extension point serving for our example refactoring **Move EAttribute**.



EMF Refactor generates altogether five refactoring specific Java classes as shown in the following figure. These classes are needed by the application module of **EMF Refactor** to execute the specified refactoring. Furthermore, a specific package is created containing the generated Java classes.



Since the application module of **EMF Refactor** uses the Eclipse Language Toolkit (LTK) technology, a refactoring requires up to three parts, either implemented in Java or using model transformation specifications. In this manual we present the implementation using Java code. The parts of a refactoring specification reflect a primary application check for a selected refactoring without input parameters (initial check), a second one with parameters (final check) and the proper refactoring execution. Therefore, the generated code contains three parts indicating those parts of the code that have to be completed (see following figure).



The following code snippet shows the Java implementation of the initial check of the example refactoring **Move EAttribute**. In lines 168 to 170 the contextual `EAttribute` instance is obtained by the refactoring specific data management object. Then, it is checked whether this attribute is marked as ID of the containing class (line 171) and a detailed error description is added (lines 172 to 174) if so. Line 177 checks whether the containing class has no referenced classes followed by an appropriate error description (lines 178 and 179).

modelrefactoring.ecore | modelrefactoring.ecorediag | RefactoringController.java

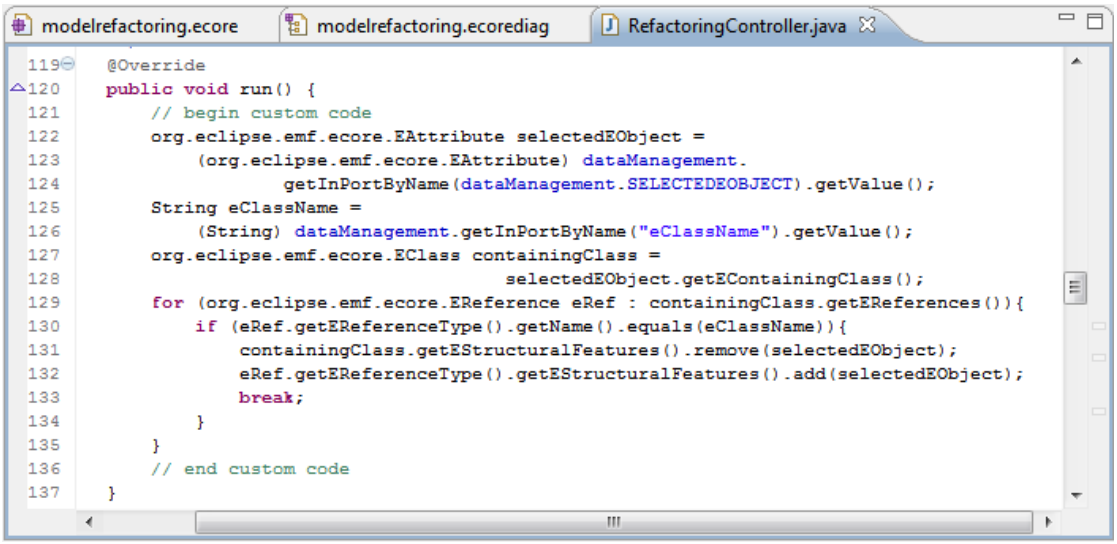
```
158  /**
159   * @see org.eclipse.ltk.core.refactoring.participants.RefactoringProcessor#
160   * checkInitialConditions(org.eclipse.core.runtime.IProgressMonitor)
161   * @generated
162   */
163  @Override
164  public RefactoringStatus checkInitialConditions(){
165          RefactoringStatus result = new RefactoringStatus();
166          // begin custom code
167          // initial check 1: the id value of the EAttribute must be 'false'
168          org.eclipse.emf.ecore.EAttribute selectedEObject =
169              (org.eclipse.emf.ecore.EAttribute) dataManagement.
170                      getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
171          if (selectedEObject.isID()){
172              result.addFatalError("The selected EAttribute cannot be moved " +
173                              "since it is used for identification purposes " +
174                              "(its id value is set to 'true')!");
175          }
176          // initial check 2: the containing EClass must have at least one EReference
177          if (selectedEObject.getEContainingClass().getEReferences().isEmpty()) {
178              result.addFatalError("The selected EAttribute cannot be moved " +
179                      "since its containing EClass does not have any EReferences!");
180          }
181          // end custom code
182          return result;
183  }
```

The following code snippet shows the Java implementation of the final check of the example refactoring **Move EAttribute**. In lines 197 to 201 the contextual `EAttribute` instance and the entered class name are obtained by the refactoring specific data management object. Then, the corresponding class is obtained (lines 203 to 208). If there is no such class (line 209) a detailed error description is added (lines 210 and 211). If there is such a class it is checked whether this class already owns an attribute with the same name as the contextual attribute followed by an appropriate error description (lines 215 and 223).

modelrefactoring.ecore | modelrefactoring.ecorediag | RefactoringController.java

```
191 @Override
192 public RefactoringStatus checkFinalConditions(){
193         RefactoringStatus result = new RefactoringStatus();
194         // begin custom code
195         // final check 1: the containing EClass must be referencing an EClass
196         // with the specified name
197         org.eclipse.emf.ecore.EAttribute selectedEObject =
198             (org.eclipse.emf.ecore.EAttribute) dataManagement.
199                     getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
200         String eClassName =
201             (String) dataManagement.getInPortByName("eClassName").getValue();
202         org.eclipse.emf.ecore.EClass referencedEClass = null;
203         for (org.eclipse.emf.ecore.EReference eRef :
204                         selectedEObject.getEContainingClass().getEReferences()){
205             if (eRef.getEReferenceType().getName().equals(eClassName)){
206                 referencedEClass = eRef.getEReferenceType();
207             }
208         }
209         if (null == referencedEClass) {
210             result.addFatalError("The containing EClass is not referencing " +
211                                  "an EClass named '" + eClassName + "'!");
212         }
213         // final check 2: the referenced EClass must not have an
214         // EStructuralFeature with the same name as the selected EAttribute
215         else {
216             for (org.eclipse.emf.ecore.EStructuralFeature eStrFeat :
217                                  referencedEClass.getEAllStructuralFeatures()){
218                 if (eStrFeat.getName().equals(selectedEObject.getName())){
219                     result.addFatalError("The referenced EClass already owns " +
220                     "an EStructuralFeature named '" + selectedEObject.getName() + "'!");
221                 }
222             }
223         }
224         // end custom code
225         return result;
226 }
```

The last figure in this manual shows the implemented model transformation of the example refactoring **Move EAttribute**. Again, in lines 122 to 128 the contextual `EAttribute` instance and the entered class name are obtained by the refactoring specific data management object. In line 131 the contextual attribute is removed from its previous containing class and line 132 inserts the contextual attribute in the corresponding referenced class. In summary, the contextual attribute is moved to the specified class.



Now, the newly specified refactoring **Move EAttribute** can be applied, either by deploying the Eclipse plug-in project or by starting the Eclipse runtime environment.

- END -