

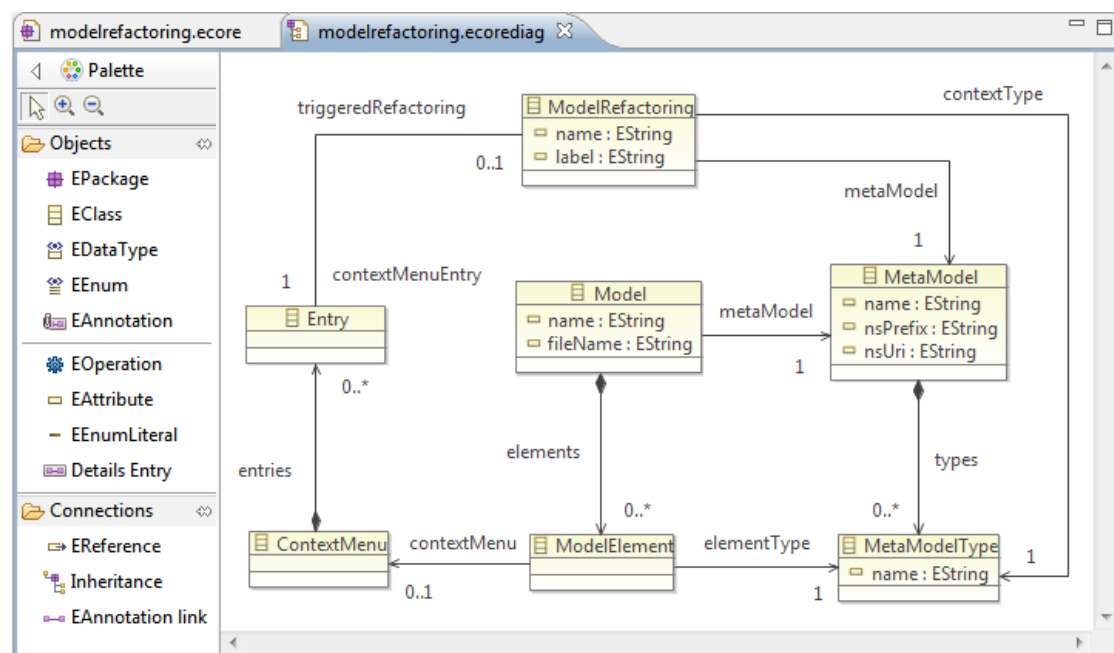
How to generate new EMF model refactorings using Henshin transformations

Thorsten Arendt

January 15, 2011

This manual presents the specification of an EMF model refactoring in **EMF Refactor** using EMF model transformations formulated in henshin. 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.

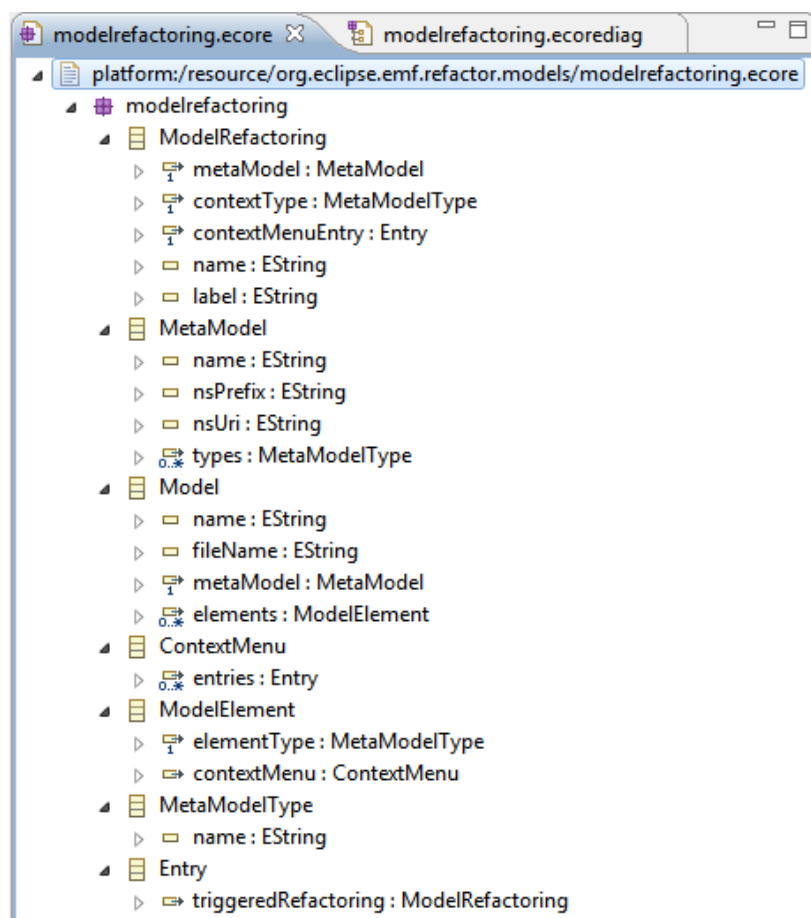


A **ModelRefactoring** has a name and conforms to a **MetaModel** that is specified by name, namespace prefix, and namespace URI. Furthermore, it has a label that should be shown as an **Entry** in the **ContextMenu** of an arbitrary **ModelElement**. A **ModelElement** belongs to a **Model** that is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



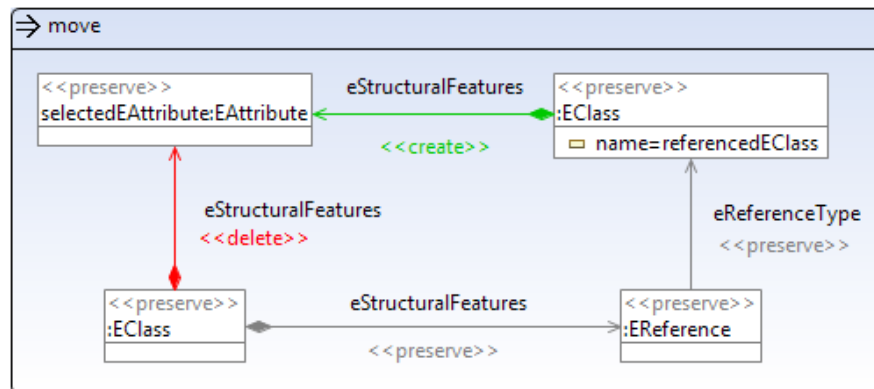
EMF model refactoring **Move EAttribute** can be specified in the following way: First, it has to be checked whether the contextual `EAttribute` is not marked as ID of the containing class, and whether this class has at least one referenced class. If these (initial) checks pass the user has to put in 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.

Before triggering the code generation process in **EMF Refactor** we specify the corresponding EMF model transformations using henshin. Henshin¹ is a new approach for inplace transformations of EMF models and uses pattern-based rules which can be structured into nested transformation units with well-defined operational semantics.

Each part of a EMF model refactoring (initial check, final check, and the proper model transformation) has to be specified by a henshin transformation unit named `mainUnit` to be executed by **EMF Refactor**. Then, this transformation unit can reference the corresponding henshin rules. Besides refactoring specific parameters, each main unit must have a parameter named `selectedEObject` representing the contextual model element the refactoring should be applied on.

The following figure shows the henshin rule `move` specifying the movement of the contextual `EAttribute` from its containing `EClass` to a referenced `EClass`. This rule is contained in a henshin `SequentialUnit` named `mainUnit` to be executed. Refactoring **Move EAttribute** (i.e. the corresponding main unit) has one parameter, `eClassName`, representing the name of the class the attribute has to be moved to. The value of this parameter is passed to rule parameter `referencedEClass` and the value of parameter `selectedEObject` is passed to rule parameter `selectedEAttribute`.

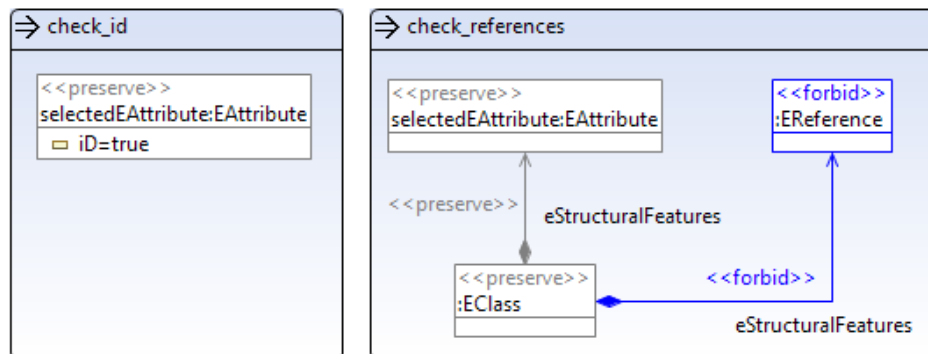


This rule uses the abstract syntax of EMF Ecore. It specifies the selected `EAttribute` (in the upper left corner) that is contained in an `EClass` (in the lower left corner). This class also has an `EReference` to another `EClass` (in the upper right corner) with the specified name given by parameter `referencedEClass`. The containment relationship between the containing class and the contextual attribute has to be removed (represented by tags `<<delete>>`) whereas a new one between the referenced class and the contextual attribute has to be created. All other elements remain unchanged (represented by tags `<<preserve>>`).

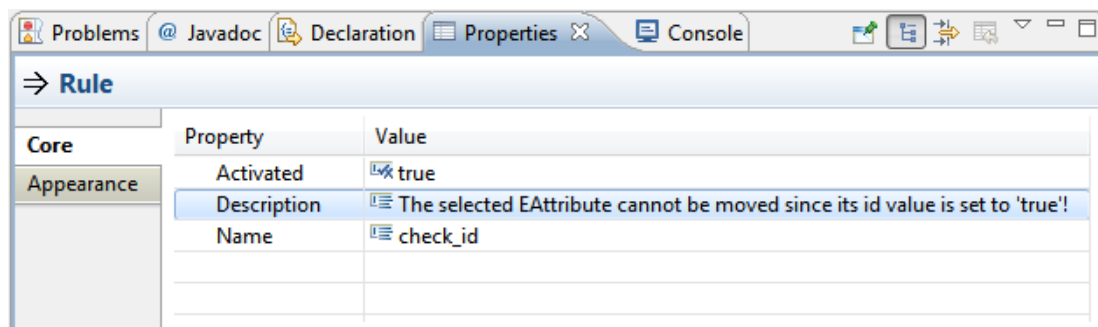
In **EMF Refactor**, initial and final precondition checks can also be specified using henshin transformations. Here, each conflicting situation is defined by a rule pattern using the abstract syntax of the underlying modeling language. These rules must be

¹<http://www.eclipse.org/modeling/emft/henshin/>

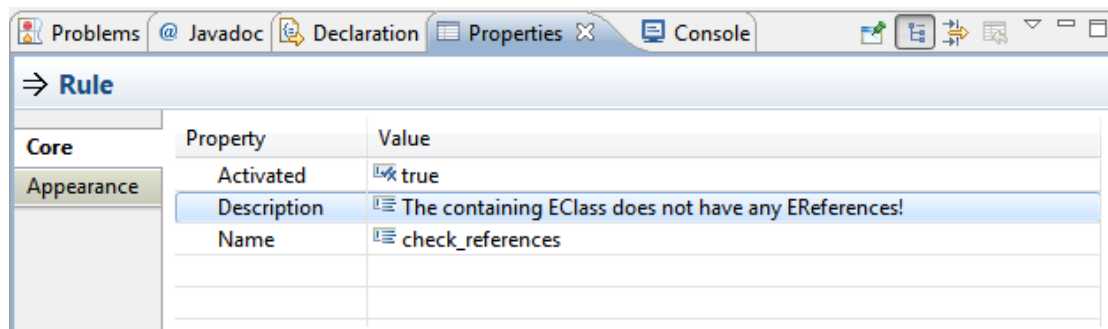
included in a henshin unit following the same conventions as the execution unit (see above). The following figure shows both henshin rules specifying the initial checks of refactoring **Move EAttribute**.



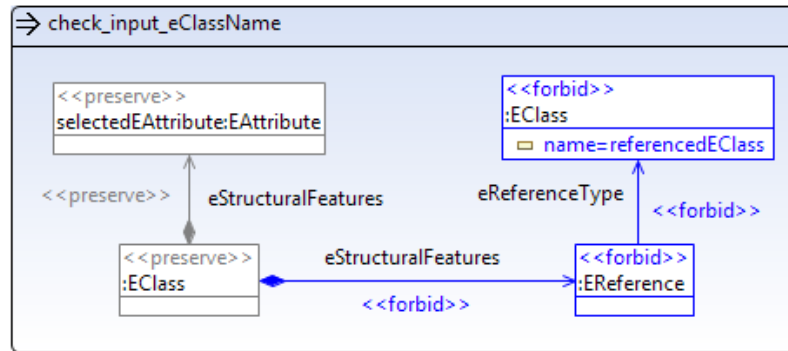
Rule `check_id` checks whether the selected attribute is marked as ID of the containing class. Rule `check_references` checks whether the containing class has no referenced classes. The absence of a referenced class is modeled using tags `<<forbid>>`. These rules are contained in a henshin `IndependentUnit` to be executed. If rule `check_id` can be applied, **EMF Refactor** uses its description value to provide a detailed error message (see following figure).



The following figure shows the corresponding description (respectively error message) of rule `check_references`.



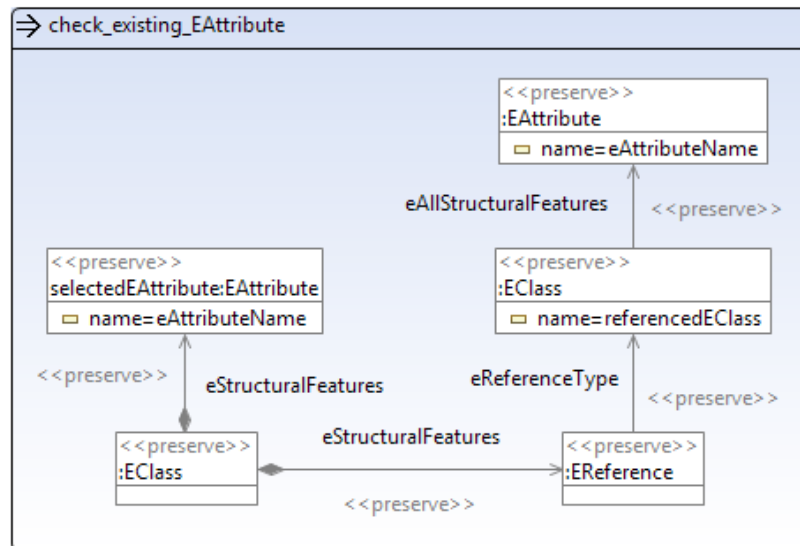
As mentioned above, there are two final conditions that have to be checked. First, there must be a class with the user specified name that is referenced by the containing class of the contextual attribute. The rule pattern for the absence of such a class is shown in the following figure. Again, the value of unit parameter eClassName is passed to rule parameter referencedEClass for this purpose.



The following figure shows the corresponding description (respectively error message) of rule `check_input_eClassName`.

⇒ Rule		
Core	Property	Value
Appearance	Activated	<input checked="" type="checkbox"/> true
	Description	The containing EClass is not referencing an EClass with the specified name!
	Name	check_input_eClassName

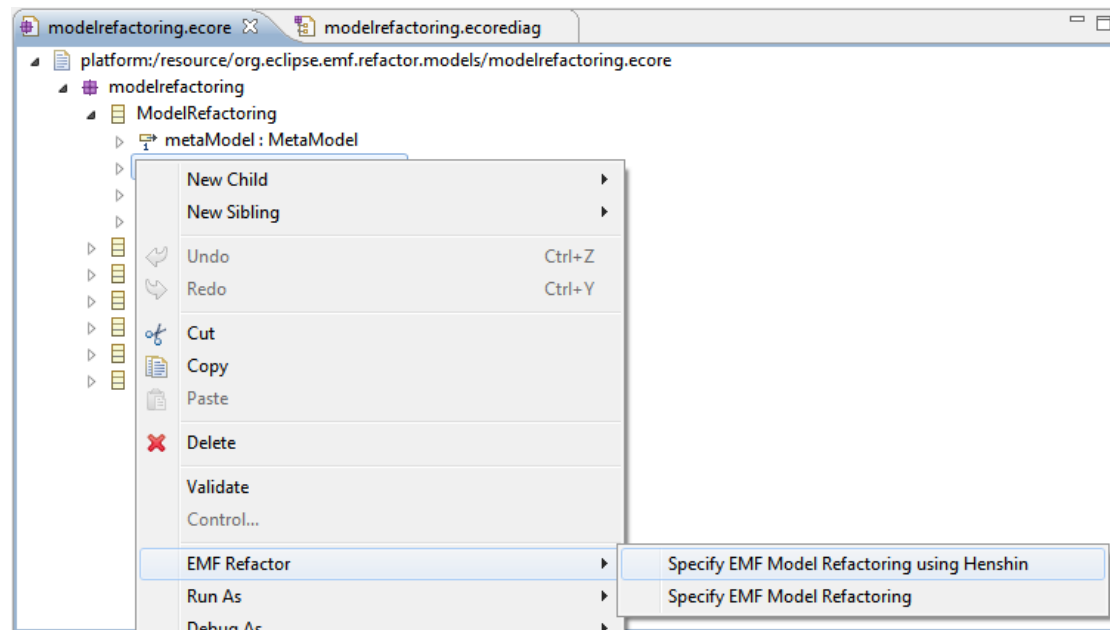
The second (and last) final precondition that has to be checked is specified by rule `check_existing_EAttribute` as shown in the following figure. Besides the already known parameters `selectedEAttribute` and `referencedEClass`, this rule has another parameter, `eAttributeName`. When selecting the contextual attribute, this parameter is set to the attribute's name. Then, the rule checks whether the referenced class (with the user given name) already owns an attribute respectively reference with the same name as the contextual attribute using parameter `eAttributeName`.



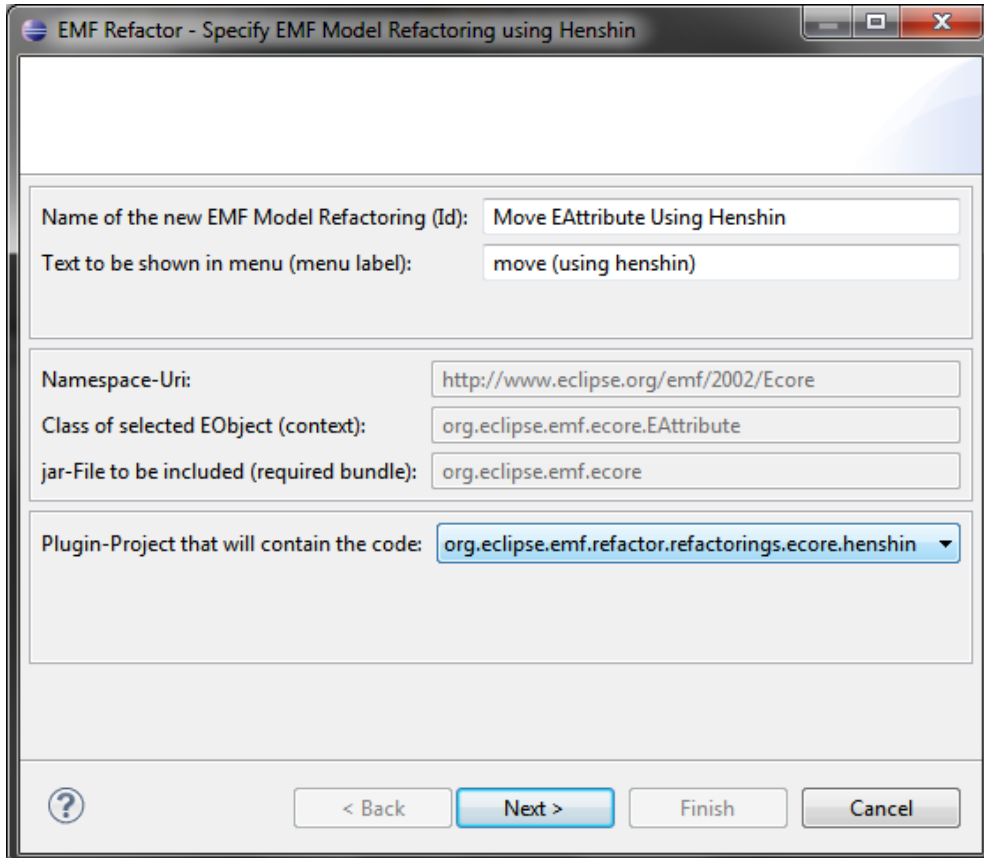
The following figure shows the corresponding description (respectively error message) of rule `check_existing_EAttribute`.

⇒ Rule		
Core	Property	Value
Appearance	Activated	<input checked="" type="checkbox"/> true
	Description	The referenced EClass already owns an EStructuralFeature with the same name!
	Name	check_existing_EAttribute

After constructing the necessary henshin transformations we can start the code generation process of **EMF Refactor**. This 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 using Henshin**.



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.

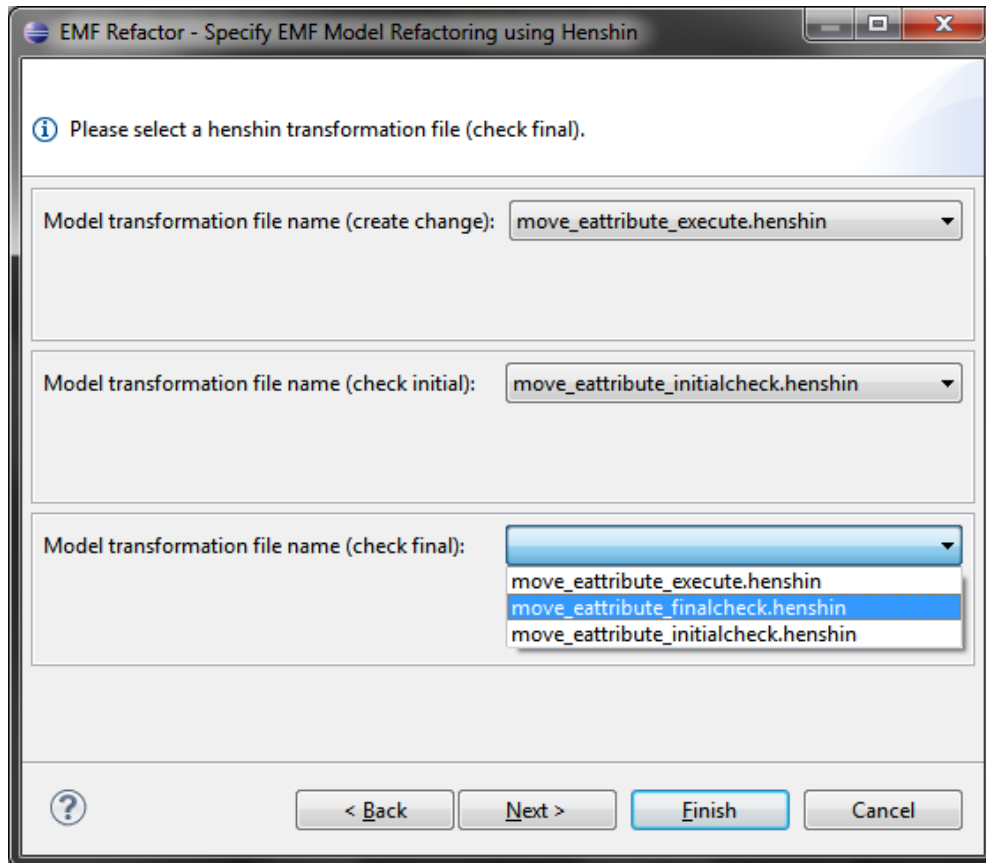


The image shows a dialog box titled "EMF Refactor - Specify EMF Model Refactoring using Henshin". It contains several input fields and a dropdown menu. The fields are: "Name of the new EMF Model Refactoring (Id):" with the value "Move EAttribute Using Henshin", "Text to be shown in menu (menu label):" with the value "move (using henshin)", "Namespace-Uri:" with the value "http://www.eclipse.org/emf/2002/Ecore", "Class of selected EObject (context):" with the value "org.eclipse.emf.ecore.EAttribute", and "jar-File to be included (required bundle):" with the value "org.eclipse.emf.ecore". The "Plugin-Project that will contain the code:" dropdown menu is set to "org.eclipse.emf.refactor.refactorings.ecore.henshin". At the bottom, there is a help icon (question mark) and four buttons: "< Back", "Next >", "Finish", and "Cancel".

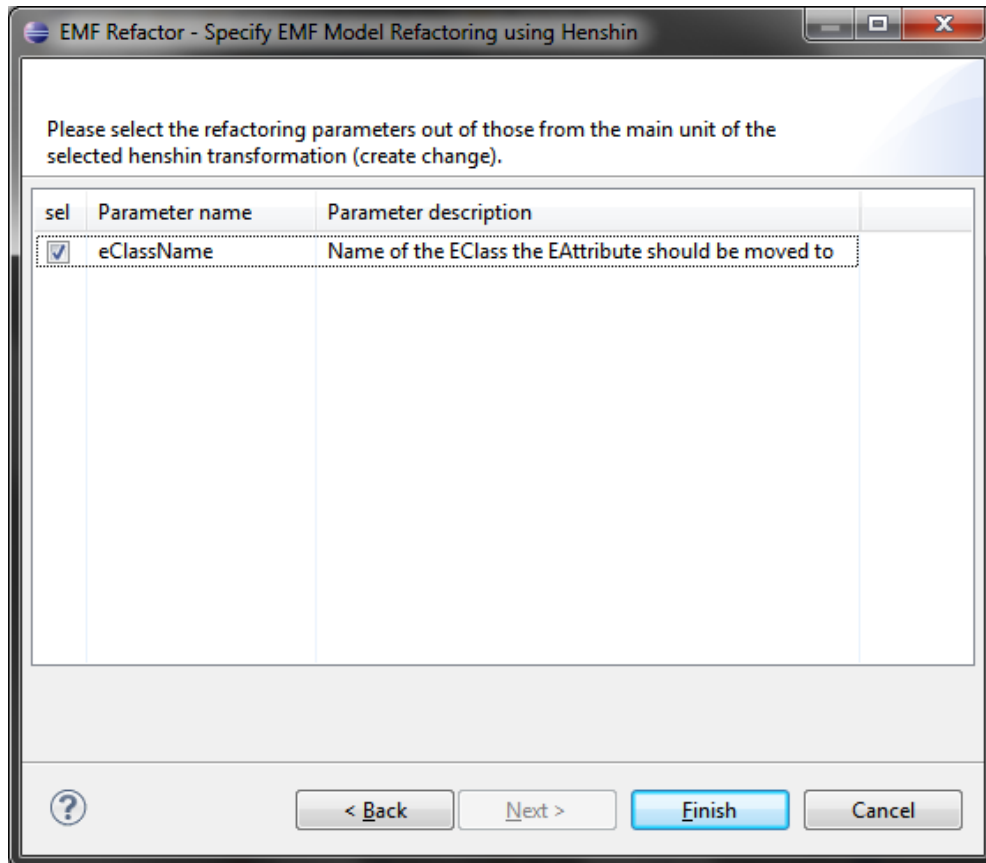
Name of the new EMF Model Refactoring (Id):	Move EAttribute Using Henshin
Text to be shown in menu (menu label):	move (using henshin)
Namespace-Uri:	http://www.eclipse.org/emf/2002/Ecore
Class of selected EObject (context):	org.eclipse.emf.ecore.EAttribute
jar-File to be included (required bundle):	org.eclipse.emf.ecore
Plugin-Project that will contain the code:	org.eclipse.emf.refactor.refactorings.ecore.henshin

< Back Next > Finish Cancel

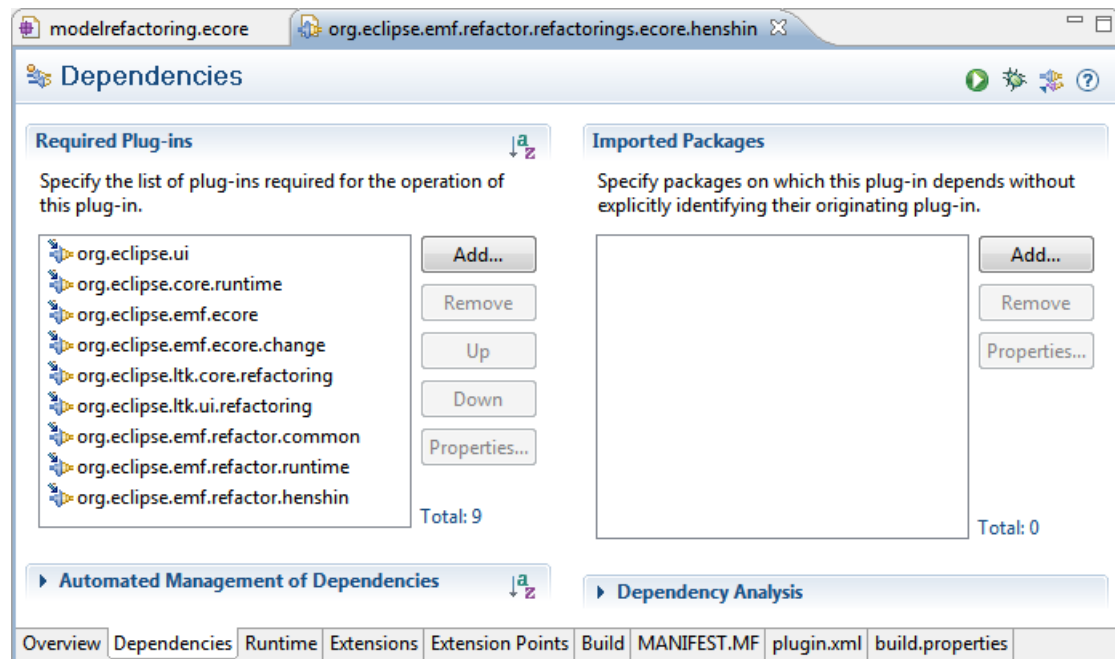
In the second page of the refactoring generation dialog the henshin transformation files that specify the three parts of the refactoring has to be selected. These files must be available in a folder named transformation in the plug-in project that has been selected in the previous dialog page. The following figure shows the selection for our example refactoring **Move EAttribute**.



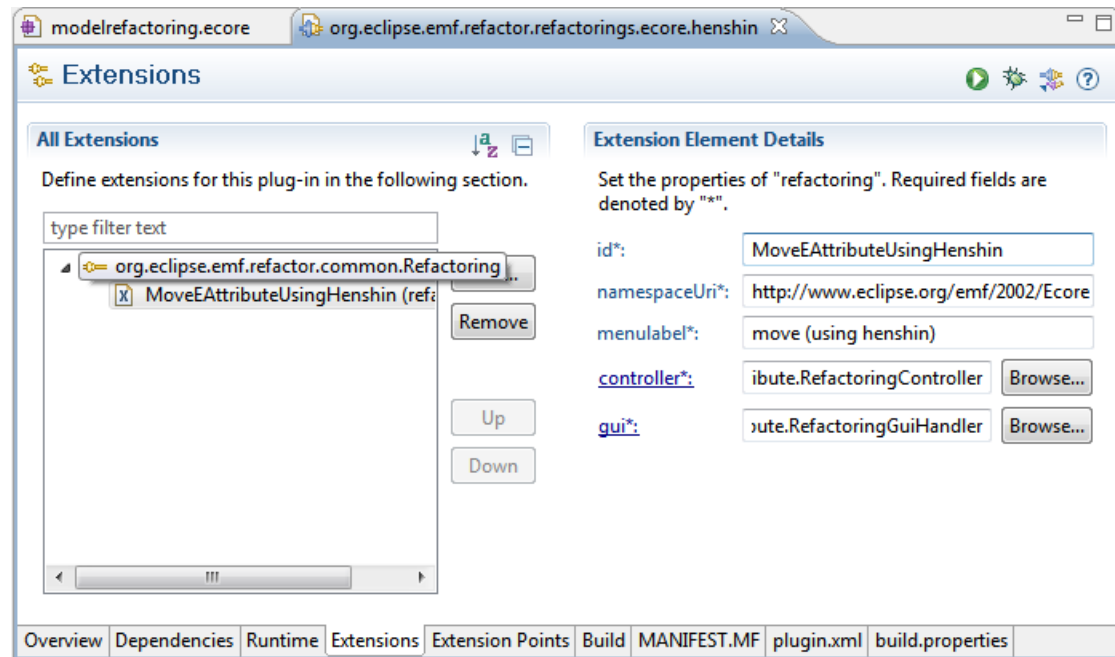
The third page of the refactoring generation dialog specifies the parameters of the corresponding model refactoring. It shows the parameters of the main transformation unit of the corresponding execution henshin file selected in the previous dialog page (except for the contextual parameter `selectedEObject`). For refactoring **Move EAttribute** we select parameter `eClassName` as shown in the following figure.



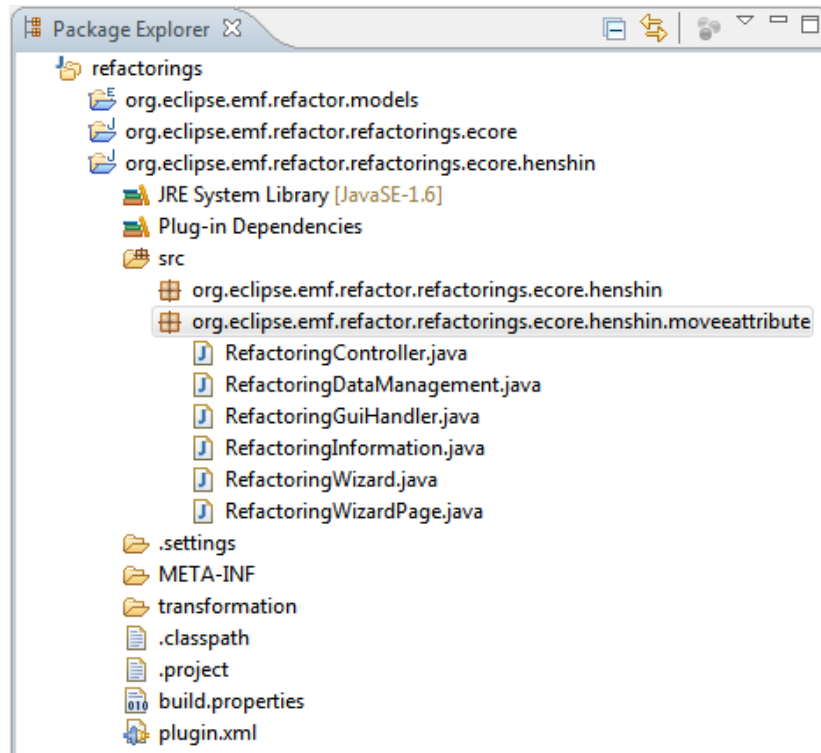
After finishing the refactoring generation dialog, **EMF Refactor** adds some additional information to the selected Eclipse plug-in project. First, **EMF Refactor** adds additionally required plug-in dependencies like shown in the following figure.



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 **MoveEAttribute** (using `henshin`).



EMF Refactor generates altogether six 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.



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 -