

# Cross Language Refactoring for Eclipse Plug-ins

Martin Kempf    Reto Kleeb    Michael Klenk    Prof. Peter Sommerlad

IFS Institute for Software at HSR Rapperswil  
Oberseestr. 10, CH-8640 Rapperswil, Switzerland

martin.kempf@gmail.com, reto@techbase.ch, michael.klenk@hsr.ch, peter.sommerlad@hsr.ch

## Abstract

This article presents our research on how a cross-language refactoring could be implemented in an Eclipse Plugin.

A non-Java language running on the Java virtual machine JVM interacts with Java code. Refactorings in either language might break the code written in the other. To keep the code synchronized, cross-language Refactoring is needed. In this article we describe how such a cross-language feature can be implemented. Which parts of the Java Development Toolkit JDT of Eclipse you can use and what has to be written at your own.

The examples are shown for the language Groovy for which we implemented refactoring support for the Eclipse Plugin (GRE-Refactoring), but the basic architecture can also be used for others like Scala.

**Categories and Subject Descriptors** D [2]: 6—Programming Environments

**General Terms** Design, Languages, Experimentation

**Keywords** Cross Language Refactoring, Groovy, Eclipse, JDT

## 1. Why do we need Cross Language Refactorings?

Groovy can seamlessly interact with Java and vice versa. Nearly everything is possible. A Groovy class extending a Java class, calling a Java method from Groovy, implementing a Groovy interface in Java and so on. Based on this close relationship the wish appeared to consider both, Java and Groovy, when a refactoring is performed in Eclipse. Imagine you are writing a test infrastructure with Groovy because of its dynamic behavior, but the application is written in Java. After a refactoring in the application – something that hap-

pens quite often – the test infrastructure is broken. The calls form the tests to the application can not be executed anymore because a method has for example been renamed. Furthermore you don't get to see the error before runtime. The compiler does not warn you about a method that does not exists. Or even worse, after the rename the test now calls a different method. To avoid these time-consuming consequences, a cross language refactoring is required. This paper will give an overview over the use cases which must be considered. It shows possible solution-concepts as well as a mapping of the solutions to the use cases.

## 2. Pragmatic Approach Evaluation

To realize the cross language refactoring as simply as possible, we had the idea to use the JDT refactoring to change affected Groovy elements in a Java file. This idea does not work because the Java model recognizes the references to Groovy elements only on the binary level. And the Java refactoring does only work for (Java) source code. Due to the same reason the JDT does not offer the context menu item Refactor → Rename... . From Dr. Dirk Bäumer's<sup>1</sup> point of view a hack to make the menu available is not possible, without changing the JDT.

On the other side, we had the idea of executing the Groovy refactoring after a JDT refactoring to check if Groovy code is affected and execute the changes if so. This should work with a so called "rename participant".

To summarise: Renaming Groovy elements in Java code and starting a refactoring from the Groovy element in Java code is the problem. Both problems have their origin in the fact that Groovy references are seen from the JDT at binary level. But if there are no references from Java to Groovy, a cross language refactoring is fully possible without big changes.

## 3. Solution Concepts

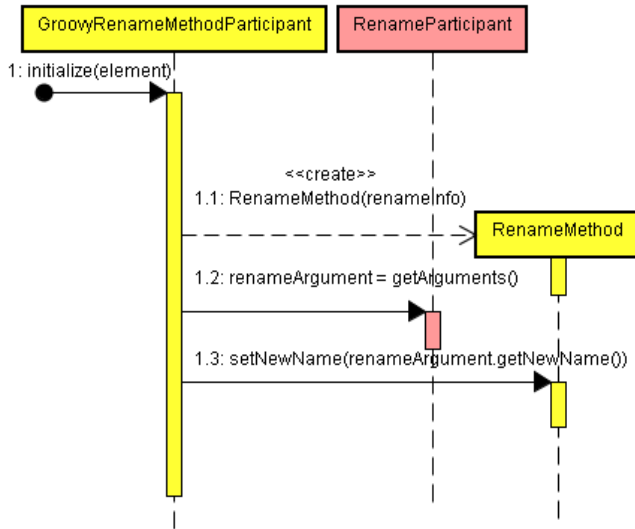
### 3.1 Using the JDT Rename Participants

A rename participant can be used to start additional refactorings when a Java refactoring is proceeding. This mechanism is for example also used to change class references in the plugin.xml when a class is renamed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WRT'08, October 19, 2008, Nashville, Tennessee, USA.  
Copyright © 2008 ACM 978-1-60558-339-6/08/10...\$5.00

<sup>1</sup>Dirk Bäumer is a member of the Eclipse architecture team



**Figure 1.** Cooperation of rename participant with the Groovy refactoring

In this paper, we will show an example how such a cross-language refactoring could be implemented for the refactoring rename method.

### Extension Point

```

<extension point="org.eclipse.ltk.core.refactoring.renameParticipants">
  <renameParticipant
    class="GroovyRenameMethodParticipant"
    id="ui.GroovyRenameParticipant"
    name="GroovyRename">
    <enablement>
      <with variable="element">
        <instanceof
          value="org.eclipse.jdt.core.IMethod">
        </instanceof>
      </with>
    </enablement>
  </renameParticipant>
</extension>
  
```

### The Rename Participant

The class to implement in the sequence diagram below is the GroovyRenameMethodParticipant and the refactoring class itself. The return value of the method boolean initialize(Object element) decides whether the rename participant is participating or not. Furthermore the diagram shows how the implemented Groovy refactoring could cooperate with the rename participant (RenameParticipant). See Figure 1. After this initialization phase the participant gets requests from its refactoring processor to check the conditions and to create its change object. These requests are simply delegated to the RenameMethod object which performs the refactoring for the Groovy code.

### 3.2 Java Search Engine

To find Groovy or Java elements in Java source code, the Java search engine can be used. With this engine it is possible to define where, what and how a Java element can be found. It also finds references in the class file. Therefore a search for the Java encoding of a Groovy method will return references to Groovy in Java code (JavaSearchEngine).

#### SearchPattern

A search pattern defines the element that is supposed to be found. The pattern is created with the following static method from the class SearchPattern

```

SearchPattern createPattern(String stringPattern,
  int searchFor, int limitTo, int matchRule)
  
```

The parameter stringPattern varies depending on the Java element to search, which is specified with the parameter searchFor. Of course, the more specific the search pattern is, the less elements get found. This has to be considered when the pattern is not complete due to type inference reasons. The details about how the stringPattern must be set up, can be found in the class SearchPattern of the JDT.

#### Java Search Scope

The Java search scope defines where an element is supposed to be found, and can therefore be used to limit the search scope. Most of the methods to create a search scope already need a Java element. But since the search engine is used inside a Groovy refactoring, to even find a Java element, the first search scope will always have to be the whole workspace. It might be faster to first find the class on which e.g. the method call is executed, then create a hierarchy scope, and start a second search after the method inside the limiting hierarchy scope.

#### Search Requestor

The search requester is used to collect the search results. Every time the search engine finds an element matching the pattern in the scope, the method acceptSearchMatch (SearchMatch match) is called. Using the parameter match, the Java element can be extracted, analyzed and if usable put into a collection.

#### Search Sample

The following code sample shows the search for a method declaration inside a workspace scope.

```

String pattern = "MyClass.MyMethod() void";

//create search pattern
SearchPattern searchPattern = SearchPattern.createPattern(pattern,
  IJavaSearchConstants.METHOD,
  IJavaSearchConstants.DECLARATIONS,
  SearchPattern.R_EXACT_MATCH);
  
```

```

SearchEngine searchEngine =
    new SearchEngine();
IJavaSearchScope scope =
    SearchEngine.createWorkspaceScope();

final List<IMethod> methodList =
    new ArrayList<IMethod>();
SearchRequestor requestor= new SearchRequestor() {
    public void acceptSearchMatch(SearchMatch match) {
        Object element = match.getElement();
        if (match.getElement() instanceof IMethod) {
            methodList.add((IMethod)element);
        }
    }
};
searchEngine.search(searchPattern,
    new SearchParticipant[] { SearchEngine.
    getDefaultSearchParticipant()},
    scope, requestor, new NullProgressMonitor());

```

### 3.3 Starting a JDT Refactoring Programmatically

The JDT offers the possibility to call a refactoring programmatically. The process comprises of the following steps (RefactoringContrib):

1. Get the refactoring contribution of the refactoring you want to start.
2. Get the descriptor of the refactoring you want to start.
3. Parametrize the descriptor.
4. Create the refactoring using the descriptor.
5. Call the three main methods (checkInitialConditions, checkFinalConditions, createChange) of the refactoring.

#### Sample Listing

The code sample shows the programmatic start of the refactoring “Rename Method”. The code is a continuation of the search code sample.

```

RefactoringContribution contribution =
    RefactoringCore.getRefactoringContribution(
    IJavaRefactorings.RENAME_METHOD);

RenameJavaElementDescriptor descriptor =
    (RenameJavaElementDescriptor) contribution.
    createDescriptor();

//set the Java element to refactor
descriptor.setJavaElement(methodList.get(0));
//new method name from user input
descriptor.setNewName(newMethodName);
//refactor also the references
descriptor.setUpdateReferences(true);

RefactoringStatus status = new RefactoringStatus();
state = descriptor.validateDescriptor();

Refactoring renameMethod = descriptor.

```

```

    createRefactoring(state);
state.merge(renameMethod.
    checkInitialConditions(pm));
state.merge(renameMethod.
    checkFinalConditions(pm));
return renameMethod.createChange(pm);

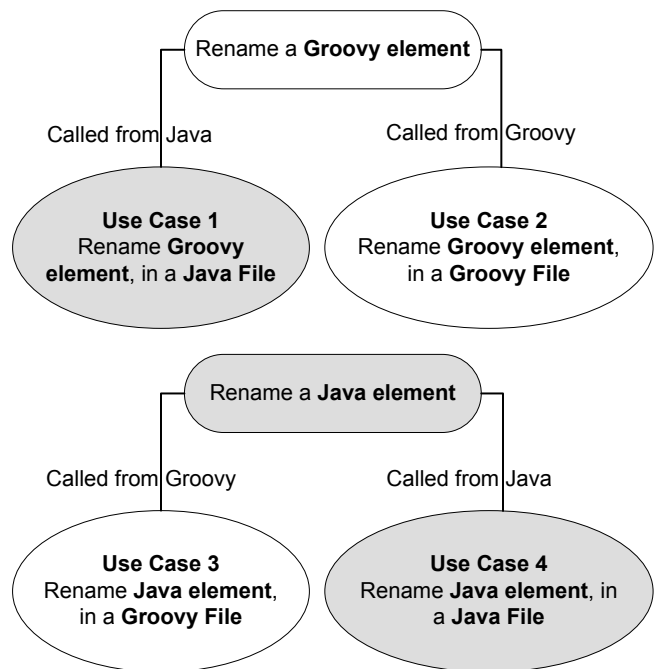
```

### 3.4 Rename Binary Groovy Elements in Java

As already mentioned, Groovy elements in Java source code are only available as binary elements but with the Java search engine it is possible to find out if the element is referenced by Java code. Renaming a found binary element is not supported by the JDT. Having this would make the cross language refactoring complete.

A possible solution is to first search for a Groovy element using the search engine. If a Groovy element is found in Java source it is clear, that the element must be refactored also in Java source. The binary elements can not provide proper source information. To get them, the text search engine of Eclipse could be used with a search pattern. The described approach is not tested, but one thing is clear: the Groovy refactoring must also generate the text edits to rename the element in the Java source.

## 4. Use Case - Solution Mapping



**Figure 2.** Use case diagram of the possible cross language refactorings. (Use cases for the JDT are marked gray)

The postconditions are the same for all the use cases shown in Figure 2:

- All occurrences of the Groovy element in Java and Groovy files are renamed.

- All occurrences of the Java element in Java and Groovy files are renamed.

#### 4.1 Use Case 1: Rename Groovy element used in Java

Use Case: The user is working with a Java file and selects a Groovy element to rename.

##### Solution Steps

Provide a new context menu Refactoring → Groovy Rename... . This menu entry has its own action from which the following steps are started:

1. Evaluate the element considering the selection information.
2. Start the corresponding Groovy refactoring with the evaluated information in the first step.
3. According to the information received, refactor the Groovy code.
4. Create a `SearchPattern`, to find references in Java source using the Java search engine.
5. Generate edits for found references in the Java file with the Groovy refactoring.

#### 4.2 Use Case 2: Rename Groovy element used in Groovy

Use Case: The user is working with a Groovy file and selects a Groovy element to be renamed.

##### Solution Steps

1. Get the selected Groovy element in the Groovy source.
2. Create a `SearchPattern`, to find references in Java source using the Java search engine.
3. Generate edits for found references in the Java file with the Groovy refactoring.

#### 4.3 Use Case 3: Rename Java element used in Groovy

Use Case: The user is working with a Groovy file and selects a Java element to be renamed.

##### Solution Steps

1. Get the selected Java element in the Groovy AST. The Java element is represented like a Groovy element in the AST (the representation of Java elements in the AST of your target language may differ).
2. Find the declaration of the element with the Java search engine.
3. Start the JDT rename refactoring programmatically with the found declaration.
4. Use the rename participant to refactor the code in Groovy.

Alternatively, the last point is not needed. Instead, the refactoring of Groovy code is started normally and not by the

rename participant. But since the participant is used anyways in use case 4, the solution with the rename participant is smarter.

#### 4.4 Use Case 4: Rename Java element used in Java

Use Case: The user is working with a Java file and selects a Java element to be renamed.

##### Solution Steps

1. Use the rename participant to start the corresponding Groovy refactoring.
2. Convert the Java element, received in parameter `element` when initializing the rename participant, into a pattern that can be used by the language specific refactoring.
3. Start the Groovy refactoring with the created Groovy pattern.

## 5. Conclusions

This paper offers a good base of ideas of how to realize the cross language refactoring. But it will definitely be challenging to cover all of the possible situations. Despite the fact that the JDT offers some ingenious possibilities to interact with, the integration of all the required components is still cumbersome. To sum up, the following issues are not solved yet:

- Evaluate a Groovy element in a Java file. (UC: 1, Step: 1).
- Rename Groovy elements in a Java file. (UC: 1, Step: 5 as well as UC: 2, Step: 3)

For the use case three and four, refactoring of the Java code, the described solution does work completely and the cross language refactoring could be implemented using these approaches.

## References

- [JavaSearchEngine] Using the Java search engine (JDT Plug-in Developer Guide), Eclipse documentation: [http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt\\_api\\_search.htm](http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_api_search.htm), 2008
- [RenameParticipant] Rename Participants (Platform Plug-in Developer Guide), Eclipse documentation: <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/>, 2008 Search for Rename Participant and choose the extension point description
- [RefactoringContrib] Refactoring Contribution (JDT API), Eclipse Corner Article: <http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html>, 2008
- [GRE-Refactoring] Refactoring Support for the Groovy-Eclipse Plugin, University of Applied Sciences (HSR) in Rapperswil, Switzerland. <http://groovy.ifs.hsr.ch>