Refactoring Support for the Groovy-Eclipse Plug-in

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 refactoring plug-in for the Groovy-Eclipse Plug-in.
Refactoring is a very important technique for every software engineer to ensure the healthiness of his code and a cornerstone of agile software development. In our project we introduce refactoring support for *Groovy-Eclipse*, with six automated refactorings as well as a source code formatter.
Since Java and Groovy are that closely related we also analyzed and documented the options to introduce cross-language refactorings between Java and Groovy.

Categories and Subject Descriptors D [2]: 7—Distribution, Maintenance, and Enhancement

General Terms Design, Languages

Keywords Refactoring, Groovy, Eclipse

1. Motivation

It should be the goal of every software engineer to produce software that not only works as expected but also is easy to understand and to maintain. Writing good software is not something that one could learn by taking a few classes and putting the GoF (DesPatterns) book under his pillow.

One of the skills that make a good software developer is the ability to maintain and simplify existing code. And that is when various tools, Integrated development environments (IDEs) in particular, become handy. One feature that is offered by most IDEs nowadays is refactoring: All the various refactorings have the goal to improve the quality of the existing code without changing its behavior, make it easier to maintain, understand and test.

Groovy is a modern but as of now not yet widely known language, that runs inside the Java Virtual Machine (JVM). It is dynamically typed and includes all the neat features that other languages with dynamic type systems like Ruby or Python offer.

There are already various plugins for the most common IDEs and text editors that support Groovy. The plugin we used as a base for our refactorings is *Groovy-Eclipse* – the plugin that brings Groovy support to the Eclipse platform. When we started with our project the plugin did not contain any kind of refactoring support.

2. Goal

Our personal goal is to improve the functionality of the Groovy-Eclipse plugin in general. Since there are a lot of features someone could work on, we only focused on the refactorings that were requested by the community. Besides the lack of automated refactorings another feature was missed by Groovy-Eclipse users: A solution to maintain a consistent style of the source code.
Therefore the goal of our project is to introduce support for automated refactorings as well as a Groovy source code formatter.
We created a first code base that would allow subsequent projects to benefit from our results. This base could be used to introduce more refactorings or other plugin features. Furthermore we wanted to document the difficulties and pitfalls we came across. Considering the fact that Groovy is closely related to Java, there was also the idea to create cross-language refactorings. Unfortunately the integration of the Eclipse Java Development Tools (JDT) turned out to be more difficult than expected. The results of the analysis are documented and provide an overview of the different subjects that need to be considered for a successful integration of an external plugin into the JDT refactorings.

3. Implemented Refactorings

During the term project and our bachelor thesis we were able to introduce the following refactorings as well as a source code formatter:

- Extract Method

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

- Inline Method
- Rename Class
- Rename Method
- Rename Field
- Rename Local Variables

4. Example: Refactoring Extract Method

The basic idea behind this refactoring is to extract a selected block of statements and move them into a new method. This new method will then be called from the position in the source where the statements used to be. Local variables used in the selected statements will be passed to the new method. If one of the variables will be assigned to a new value this variable will be returned. This fact combined with the restriction of Groovy that a method can only have one return value leads to the limitation that only groups of statements with no more than one assignment to a variable can be extracted.

```
def method (){
  def myNum = 0
  myNum++ // selected statement
  println myNum
}
```

Result after performing Extract Method:

```
def method (){
  def myNum = 0
  myNum = increment (myNum)
  println myNum
}
def increment (val) {
  val++
  return val
}
```

5. Example: Code Formatter

The feature that was missed the most by users of Groovy-Eclipse was a formatter: A tool to maintain source code so that it stays human readable and consistent. A common approach is to indent the code lines according to the nesting of the code statements: Similar constructs should be written following a convention to make sure everyone recognizes them as fast as possible.
A sample code snipped after a hacking session,

```
def race_start ()
{
3.times
{
print "GO "
}}
```

will be formated like this:

```
def race_start () {
  3.times { print "GO " }
}
```

6. Results

We started this project by digging into the code of the existing plugin as well as the code and the grammar of the Groovy language itself. With a deeper understanding and the support of the developer mailing list, we discovered the spots that were important for our project and we started to understand how the plugin and the Groovy-core work together.

After a while we found more and more situations where information in the AST (mostly positions) were simply wrong. To be able to replace, and insert as well as to correctly find the important snippets of code inside a source file, we had to fix these errors. We did successfully change and test the relevant spots in the grammar as well as in certain other parts of the Groovy-core.

Soon after we started with the Bachelor thesis, we obtained committer rights and were able to directly commit our changes into the official repository. This was not only a great personal success but also extremely handy to speed up the integration of the core changes that we did depend on. By the end of our Bachelor thesis, we integrated the majority of the core changes into the current stable release of the Groovy-Core, and furthermore we integrated the first four refactorings (Extract Method, Inline Method, Rename Local Variable, Rename Method) as well as our source code formatter into the developer branch of the official Groovy-Eclipse repository.

In addition to merging back our changes into the official repositories, we were also able to improve the build and test situation of the Groovy-Eclipse plugin. When we first started working with the plugin there was no continuous build, the unit tests were not properly organized, and some of them where even failing. We decided to invest the extra time to fix the tests and provide a complete continuous integration solution on a server hosted at the University of Applied Sciences (HSR)[1] in Rapperswil, Switzerland.

The full report of our project is accessible on our project server hosted at HSR University of Applied Sciences in Rapperswil, Switzerland.
http://sifsstud4.hsr.ch/groovy-refactoring.pdf

References

[DesPatterns] Design Patterns. Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph E. Johnson, Addison-Wesley Longman, 1995

[1] http://www.hsr.ch