HSR -- University of Applied Sciences Rapperswil

Institute for Software

Crosslanguage Refactoring between Java and Groovy

Bachelor Thesis: Spring Term 2009

Stefan Reinhard
guetux@infosky.ch

Stefan Sidler
stefansidler@gmx.ch

Supervised by Prof. Peter Sommerlad

Version: June 12, 2009

Abstract

The *Groovy Plug-in* for the *Eclipse IDE* features a number of automated refactorings, that were realized in a bachelor thesis at the University of Applied Sciences Rapperswil. Although Groovy and Java code can be used mutually in projects, the already implemented refactorings are limited to Groovy code.

In this subsequent bachelor thesis the *Groovy Eclipse Plug-in* was extended by *Crosslanguage Refactorings*: If either a Groovy or a Java element is renamed, both languages are respected throughout the complete refactoring process. Our achieved goal was to support all possible rename actions, no matter in which language the element to be refactored was defined.

As a result, the *Groovy Eclipse Plug-in* offers a higher level of integration with the *Java Development Tools* and more productivity for Groovy developers. Due to automated testing over complete development process, our solution is ready for production use and will be submitted to the official plug-in maintainers.

Management Summary

Introduction

The goal for every good software developer should be, to write not only functional code, it should also be readable. To achieve this, it is required to change and improve the code a few times during development. It is important, that this task, called refactoring - changing the structure without changing its functionality - does not need too much time from the developer. So nowadays most of the *Integrated Development Environments* (IDEs) have automated features for the most common refactorings included.

Groovy is a modern, dynamically typed programming language, based on the Java Virtual Machine (JVM). It is closely related to Java and projects can be mixed with both, Groovy and Java code. Given Java components can be accessed in Groovy and additional functionality can be added. Or, of course the other way around. For a Java developer, it's really easy to start programming Groovy and he can use his already in Java written classes.

In a former Bachelor thesis, a team from the *University of Applied Sciences (HSR)*[1] in Rapperswil, Switzerland, added the refactoring functionality to the Groovy plug-in in Eclipse. Unfortunately this plug-in was just able to alter Groovy code. In a mixed project this is not really handy for a Groovy developer.

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

Approach

We were both Eclipse users since a long time, but none of us ever saw behind the scenes. So first we both had to dig ourselves into the Eclipse framework and into the already existing Groovy plug-in. In the Bachelor thesis of our predecessors [KKK08] the new functionality was theoretically analyzed. Based on this, we split the crosslanguage feature into four different scenarios.

- Local Java refactoring: refactor a Java element, started from the Java editor
- Remote Java refactoring: refactor a Java element, started from the Groovy editor
- Local Groovy refactoring: refactor a Groovy element, started from the Groovy editor
- Remote Groovy refactoring: refactor a Groovy element, started from the Java editor

For the last two scenarios, there were some unsolved problems on how to detect a Groovy element in Java, and rename Groovy elements in Java. We started with the other two scenarios, and tried find a solution to solve these problems in the meantime. This plan worked quite well, but it took us much more time to understand the framework as expected.

Results

At the end of this Bachelor thesis, we were able to expand the current plug-in with all used functions to rename a Groovy or Java element in both languages. Our private goal, to push the Groovy language is achieved as well, and we look forward to integrate our code into the official *Groovy-Eclipse Plug-in*. Hopefully, this will help a lot of Groovy developers, to improve their code.

During this thesis we came more common with dynamically typed languages, as Groovy. We learned the power it offers, as well as the threats it opens.

Both of us never worked in such a big Framework, as Eclipse before. It was a good experience, and showed us once more, how important it is to write easy readable code.

Outlook

At the end of this Bachelor thesis, we reached a level where our code can be integrated into the official *Groovy-Eclipse* repository. When this task is done, all Groovy developers can benefit from our work, and easily rename their elements. But even with this improvement, there are still a lot of open issues in the plug-in itself which have to be closed.

At the moment, the plug-in is not capable of working with dependencies between different projects. So as our refactorings. If this will change, the refactorings will have to be extended too.

Bachelor Thesis job definition

Students:

- Stefan Reinhard
- Stefan Sidler

Supervisor:

- Prof. Peter Sommerlad

Tutor:

- Michael Klenk

Duration of thesis:

- Start: 16.02.09
- End: 12.06.09

Job Description:

The target of this bachelor thesis is to introduce cross-language refactorings between the Java and Groovy programming language for the Groovy eclipse plug-in.

The Groovy language is directly designed for the Java Virtual Machine and does integrate very well with existing Java solutions. As the Groovy compiler generates native Java bytecode, it's no problem to use both languages in one and the same project. For this reason, Refactorings should work in both languages. Here's an example: If a Java method is renamed, which is used in Java and Groovy code, the refactoring should rename all references in both languages.

The Groovy refactoring plug-in for Eclipse was developed as bachelor thesis by M. Kempf, R. Kleeb and M. Klenk at the University of Applied Sciences Rapperswil. It's meanwhile integrated into the offical Groovy Eclipse plug-in and works very well, but misses the interaction with Java refactorings. On base of this thesis, the crosslanguage refactoring plug-in has to be implemented and tested. The main goal is to support all possible situations, in which a refactoring of a Java or Groovy element can be started, and then refactor all occurences of the given element in both languages.

Optional additions to the existing plug-in functionality may be the following:

- Import organization, so that only referenced classes are imported.
- Improved "Hyperlink to Type", if it's possible to get the source of the type.
- Search function for all references to a given variable.

The target of the bachelor thesis is to integrate the results into the official project and to extend the functionality of the Groovy Eclipse plug-in.

[illegible]

Supervisor Peter Sommerlad

Vereinbarung

1. Gegenstand der Vereinbarung
Mit dieser Vereinbarung werden die Rechte über die Verwendung und die Weiterentwicklung der Ergebnisse der Bachelorarbeit „Cross-Language Refactoring for the Groovy Eclipse Plug-in“ von Stefan Reinhard und Stefan Sidler unter der Betreuung von Prof. Peter Sommerlad geregelt.

2. Urheberrecht
Die Urheberrechte stehen den Studenten zu.

3. Verwendung
Die Ergebnisse der Arbeit dürfen sowohl von den Studenten wie von der HSR nach Abschluss der Arbeit verwendet und weiter entwickelt werden .

Beilage/n:
keine

Rapperswil, den 11.03.09 .... [illegible]
Stefan Reinhard

Rapperswil, den 11.03.09 .... [illegible]
Stefan Sidler

Rapperswil, den 11.03.09 .... [illegible]
Der Betreuer / die Betreuerin der Bachelorarbeit

Rapperswil, den 11.3.09 .... [illegible]
Der Studiengangleiter / die Studiengangleiterin

Disclaimer

We ensure with this disclaimer that this thesis was created by ourselves. We did not use any additional resources then the ones mentioned in the bibliography in the appendix.

The figures we used for this thesis were created by ourselves or are attached with a reference to the original author. Each copied text phrase has an attached reference to the original text. This thesis was not given, in this or in any similar form, to an examination board.

Contents

1 Introduction to the Bachelor Thesis

1.1 Refactor

Refactoring is a progressive way for structure changes in program code. Or as Martin Fowler [Fow99] said:

> *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."*

The amount of refactorings ranges from complex structure changes to simple renames of an identifier. During deployment, complex structure changes don't have to be done frequently, whereas renaming of some identifiers is quite usual. But also this simple renaming is not that simple. Manually search for all occurrences of the identifiers name, and replace it, is really unhandy. If there are two different classes with a field with the same name, both field get renamed and the whole project does not work anymore.

Automated Refactorings

To avoid the problem above, common IDEs have built-in automated refactorings, which know the structure of your software project. This refactorings can determine which code has to be renamed, and which don't. This is a great convenience for the developer. He has to rename an element just once, and all references will update automatically.

1.2 Groovy

Groovy is a dynamically typed programming language, such as Phyton or Ruby. As distinct from the other languages, it is based on the JVM and is almost fully compatible to Java. Certainly, valid Java code is valid Groovy code too. For a Java developer it's easy to learn. It is even possible to mix up a project with Java and Groovy code.

Dynamically Typed Languages

Dynamically typed means, an element can be whatever it wants to be. It's not chained to a type during deployment and can even change its kind during runtime. But more about the advantages of Groovy and dynamically typed languages in chapter 2 on page 5.

1.3 Eclipse

Eclipse is one of the most widely used IDEs for Java with automated refactoring support. Thank to its open architecture, its possible to add features through additional plug-ins. So for the Groovy language.

1.4 Groovy-Eclipse Refactorings

In a former Bachelor thesis [KKK08], our predecessors were able to add a code formatter and even various automated refactorings to the *Groovy-Eclipse Plug-in*. These refactorings work great and were added directly into the official plug-in. But unfortunately they function just within Groovy code. On a mixed project, it is possible to reference a Java element from Groovy code, or vice versa.

1.5 Crosslanguage Refactorings

For example: A class, written in Java, has a reference to an element, written in Groovy and the definition in Groovy gets renamed. When the reference in Java will remain the same, it points now out into the nonentity. Actually, this is not satisfying for any developer. A rename refactoring in a mixed project has to handle both languages.

Dynamically typed languages are a tough problem to solve for every automated refactoring. How can the IDE know to which class a field belongs, when there are no information about?

For automated *crosslanguage* refactorings, these and other problems were to solve. It needs a good and clear architecture inside of the refactoring. We decided to use the given refactorings, as good as it gets, and don't reinvent the wheel again. This opened a few new issues, like how to prevent an infinite loop with Groovy and Java refactorings?

With our solution, we solved all accrued problems and can present a ingenious built and minutely tested extension for the *Groovy-Eclipse plug-in*.

1.6 Implemented Refactorings

This section presents the new features for the crosslanguage refactoring, that we have built for the *Groovy-Eclipse* plug-in. The following descriptions and examples gives just a short overview about all implemented refactorings. For a more detailed description about the challenges, please read the chapters 4 to 7.

1.6.1 Rename Field

The *Rename Field* refactoring is used to rename the definition of a given field, and update all its references.

The example shows a common crosslanguage rename refactoring, where a field gets renamed to an obviously better name. All other occurrences, in both languages, of the element gets renamed too. This action can be started from Java or Groovy as well.

Java

```
public class Java {
  public static void main(String[] args) {
    Building building = new Building();
    building.numberOfParts = 3;
  }
}
```

Groovy

```
class Building {

  public int numberOfParts
  public String address
}
```

Java (refactored)

```
public class Java {
  public static void main(String[] args) {
    Building building = new Building();
    building.numberOfFlats = 3;
  }
}
```

Groovy (refactored)

```
class Building {

  public int numberOfFlats
  public String address
}
```

1.6.2 Rename Class

The *Rename Class* refactoring is used to rename a classname into a more accurate name into the project.

The following example shows a simple rename class crosslanguage refactoring. All occurrences of the classname get renamed, to make sure the functionality remains the same.

Java

```
public class Java {
  public static void main(String[] args) {
    Vehicle bobsCar = new Vehicle();
  }
}
```

Groovy

```
class Vehicle {
  def driver
  def guest
}
```

Java (refactored)

```
public class Java {
  public static void main(String[] args) {
    Car bobsCar = new Car();
  }
}
```

Groovy (refactored)

```
class Car {
  def driver
  def guest
}
```

1.6.3 Rename Method

The *Rename Method* refactoring is very similar to the other crosslanguage refactorings. It renames the name of a method, and update all references.

In the following example, the name of the function `walkFast()` gets renamed to the more meaningful name `run()`. This task is common in agile software developing, as the name of a method should say what it does.

Java

```java
public class Java {
  public static void main(String[] args) {
    Human paul = new Human();
    paul.walkFast();
  }
}
```

Groovy

```groovy
class Human {
  def walkFast() {
    println "running"
  }
}
```

Java (refactored)

```java
public class Java {
  public static void main(String[] args) {
    Human paul = new Human();
    paul.run();
  }
}
```

Groovy (refactored)

```groovy
class Human {
  def run() {
    println "running"
  }
}
```

2 The Groovy Programming Language

According to the TIOBE Programming Community Index [TI], Java is the world's most popular programming language. The reason for this is probably that it was the first technique to offer dynamic website content (Applets) as well as having a flat learning curve due to its clearly arranged and yet very powerful syntax are considered to be the reason for this success. Thanks to its big community Java has a lot of valuable support to offer. This includes thousands of active forums, libraries and books. But Java suffers from a number of symptoms of old age. Starting with its first release in 1995, it has gained a lot of syntactical improvements but also kept some legacy issues. Along comes Groovy! A new, dynamic and agile programming language, exclusively designed for the JVM.

But what has Groovy got to offer? Upon first contact with a new programming language, most developers react in the same way: "Why the heck should I care? I can do everything I want and need with MY own language!".

This statement is certainly not completely unjustified, but if everyone would resent improvement, we would not have seen the rise of the digital age in the last decade (Imagine a web application written in assembler). The evolution of high-level programming languages has continued steadily since their first appearance in the mid fifties.

Let's get back on the subject: Why Groovy? Because it is elegant, expressive and Java people will love it. Groovy is, to be precise, the second language after Java standardized by the Java Community Process [JSR241] and is almost fully compatible to Java. Valid Java code is valid Groovy code. And both languages interact with each other without hardly any restriction. However, daily tasks are completed much more swiftly and efficiently in Groovy. Due to its powerful syntax, one is able to do the same thing, needing only half the amount of code as in Java and has the complete strength and all the abilities of the Java world at his fingertips.

This chapter will take the reader on a walk trough the magic land of Groovy and it's inhabitants. To those who are new to Groovy, get ready for an exciting adventure which will expand your Java thinking mind. And to all those already familiar with Groovy, a brief review of the advantages using it as your all-purpose glue in the Java universe is awaiting you.

2.1 Dynamic typing and duck typing

Java is a classic statically typed language with strong type checking. This simply means that any field, variable, parameter, return value and so on has to be strictly typed. In cases where type flexibility is needed, polymorphism and generics help along the way. When incompatible types get mixed up, like `String` and `Integer`, there's either an error shown at compile time or an unmistakable exception appears during run time. Additionally, in Java there is a distinction between primitive types like `int` and real objects. This is all no big news, but what are the consequences of static typing in the end?

Beginning with the advantages, static typing surely leads to clarity for the developer as well as for the compiler. The developer always knows what type he's dealing with and what it's capable of. If he uses a modern IDE, auto-completion is probably one of his best friends. Further the compiler is able to prove if the intention of the developer resulted in type-valid code. If not, he clearly states in which statement the types do not match. So static typing does help the developer to write valid code before runtime and makes code easier to debug.

The downside on the other hand is that types always *have* to be declared and thus blow up codesize. Here's an example in a pseudo Java method:

Listing 2.1: Type declarations needed in Java

```java
public SomeLongClassName handle(AnOtherWayToLongName param) {
   UsedAsIntermediateResult temp = param.getData();
   ResultingValueClass result = createResulting(temp);
   result.setSomething("literal");
   return result;
}
```

Ok, this example is maybe really a little bit extravagant. However, the equal pseudo method written in Groovy uses not even half the width:

Listing 2.2: Dynamic typing in Groovy

```groovy
def handle(param) {
   def temp = param.data
   def result = createResulting(temp)
   result.something = "literal"
   return result
}
```

Is not the Groovy version easier to skim on the first look?

The benefit of dynamic typing is not only saving keystrokes for lazy developers. It also allows more flexibility when code is often changed and reduces the necessity for refactorings. The keyword here is duck typing:

> *"If a bird walks like a duck, swims like a duck and quacks like a duck, then it probably is a duck."*

This slightly altered quote from poet *James Whitcomb Riley* applied to type systems means the following: Don't check if it *IS* a duck, check whether if its *ABLE* to walk like a duck, quack like a duck etc. Even more strictly speaking, don't limit the things you can do with an object to it's type, rather to it's capabilities. Staying on the bird example, a frog can quack too, but not all animals do. When trying to reflect this with inheritance, you end up with something like `Animals` and `QuackingAnimals`. In Groovy, this is not necessary:

Listing 2.3: Duck typing example

```groovy
class Duck {
   def quack() { println "Quack, quack" }
}

class Frog {
   def quack() { println "Quaaaaaak" }
}

def pond = [new Duck(), new Frog()]
pond.each { animal -> animal.quack() }
```

The example above creates a pond list, home of a nice duck and a neat frog. We ask each of them to `quack()` for us, and that's exactly what they do, because they're aware of how to `quack()` in their very own way. But attention is required: If we add a `Fish` to our pond, we have a new inhabitant who's only able to `bubble()` and would call a method that does not exist. Trying anyway will result in a `MissingMethodException`, unless we check if the method is actually there:

Listing 2.4: Duck typing example (continued)

```groovy
// Assuming Duck and Frog exist

class Fish {
   def bubble() { println "blubb, blubb" }
}

def pond = [new Duck(), new Frog(), new Fish()]
pond.each { animal ->
   if (animal.metaClass.respondsTo(animal, "quack") {
      animal.quack()
   }
}
```

So we set the implicit requirement that all our pond animals better ought to know how to `quack()`. And we as developers are the ones who are responsible for them do so. The `respondsTo()` check in the second example is only required if we don't know what kind of animals actually live in our pond. When we know there's only a frog and a duck, we're not required to build up a complex class structure only to ask them to quack a little.

This could be called “design by capability” instead of “design by contract”. We get more flexibility and need to do less ceremony by the small price of loosing the indication that we probably do no harm. It’s probably because of this why according to [TI] dynamic language gained a lot more attention in the last few years:

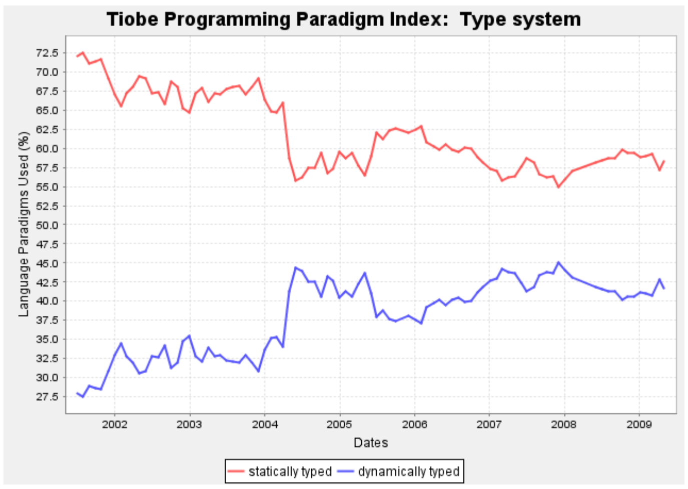


Figure 2.1: Usage of different type system paradigms over the last few years

The developer has more responsibilities with dynamic languages as there is no all time complaining compiler to hold his hands. Luckily there’s a way to handle this responsibility: Unit tests. They get an even more essential role when dealing with dynamic languages. Or as Venkat says in “Programming Groovy”[Ven08]:

> *“Programming with dynamic typing without having the discipline of unit testing is playing with wildfire.”*

2.2 Starting to groove

Because the Groovy syntax is pretty similar to Java, it's quite easy to learn for Java developers. However, there are some differences and many improvements. In this section we'll look at a bunch of those. Some of the most notable differences are:

- One of first thing most Java folks notice are the missing semicolons. They're indeed optional as long as one statement is kept on one line of course.
- All `return` statements are optional, it's always the value of the last statement that's being returned.
- The default modifier for classes and methods in Groovy is `public`. The `protected` modifier is still available though.
- Assertions are activated by default.
- Method brackets can be omitted.
- Everything in Groovy is an object. You still can declare primitive types, but they're mapped to objects automatically.
- And, most notably, there's no need to specify types. Just use the `def` keyword instead. Method parameters require no type at all.

Here's a very basic example to sum this all up. These classes do exactly the same thing. Also notice the direct access to `println` in Groovy added for convenience reasons.

Listing 2.5: Simple Java class

```java
public class JavaAdd {
  public static void main(String[] args) {
    Java inst = new JavaAdd();
    System.out.println(inst.add(11, 31));
  }

  public int add(int a, int b) {
    return a + b;
  }
}
```

Listing 2.6: Simple Groovy class

```groovy
class GroovyAdd {
   static void main(args) {
      def inst = new GroovyAdd()
      println inst.add(11, 31)
   }

   def add(a, b) {
      a + b
   }
}
```

2.2.1 Groovy Scripts

Groovy code does not necessarily have to be defined in a class with a main method to be ran. With so called Groovy scripts it's possible to write some statements directly into a `.groovy` file and execute it. A Groovy script is instantly compiled to bytecode and run by the JVM. What the compiler actually does is putting all statements into a `GroovyClass` and run it. All other classes defined within the same script are loaded too. If classes not defined within the same Scripts are used, they get dynamically loaded by the `GroovyClassloader`. Therefore, the "Hello World" example, as required by the gods of computer science, looks in Groovy like this:

```groovy
println "Hello World"
```

2.2.2 Closures

Code blocks are known in nearly any programming language and represent a pile of statements and a scope. The interesting part with closures begins with their flexibility of usage. In Groovy, code blocks are treated as first-class citizens by closures. A closure could be relaxedly defined as functional object that take any number of parameters, may return a value and use all variables from their surrounding scope. Because they're handled like objects, they can be passed around and called anywhere in the code. The name closure derives from the fact that they may be bound to variables in the surrounding scope. For that reason, closures in Groovy are always directly initialized. There's no way to "define" a closure and instantiate it later in any way. The concept of closures replaces the need for anonymous inner types as known from Java an provides much more versatility at the same time. Enough talking, here are some examples:

Listing 2.7: Closure examples

```groovy
// Define a closure that returns the current time as String
def date = { new Date().toString() }
assert date() == new Date().toString()

// A closure with two parameters
def twoParam = { name, message -> name + " says " + message }
assert twoParam("Groovy", "hi") == "Groovy says hi"

// Without explicit parameters, there's always one parameter named it
def defaultParam = { "Hi " + it }
assert defaultParam("Groovy!") == "Hi Groovy!"

// A closure passed to a self defined method
def invoke(closure) {
   closure(3)
}
assert invoke { it * 2 }  == 6

// There are many predefined methods taking closures
// Also notice that this closure does access a variable from the scope
def sum = 0
5.times { sum += 2 }
assert sum == 10
```

2.2.3 Strings and GStrings

String handling is enhanced in Groovy with single quoted, slashy and multiline strings as well as `GStrings`. The latest can be used for lazy evaluated expressions within a string.

Listing 2.8: GString usage

```
// A GString evaluating a variable and a expression
def name = "Racecar", fast = true
def sentence = "Our $name is ${fast ? 'very fast' : 'rather slow'}!"
assert sentence == "Our Racecar is very fast!"

// Only double quoted strings are evaluated
assert "$name" == "Racecar"
assert '$name' == "\$name"

// Strings also have additional methods
assert name.reverse() == "racecaR"

// A multiline string
def letter =  """I made this letter longer than usual
 because I lack the time to make it short."""
```

2.2.4 Collections

Groovy features a native syntax for lists and maps including ranges. A lot of other additional features simplifies daily tasks with collections:

Listing 2.9: Native syntax with lists and maps

```
// Define a list with some numbers
def numbers = [4, 8, 15, 16, 23, 42]

// Array style access
assert numbers[2] == 15

// Negative indexes begin at the end
assert numbers[-2] == 23

// Using ranges to get sublists
assert numbers[(1..3)] == [8, 15, 16]
assert numbers[(1..<3)] == [8, 15] // Exclusive

// Define a map of magical creatures
def creatures = [houseelf:"Dobby", owl:"Hedwig", phoenix:"Fawkes"]

// Property style access
assert creatures.owl == "Hedwig"

// Iterate over a map
creatures.each { key, value -> println "Name:$value, species:$key" }

// Operate on each element of a list with the star-dot operator
assert [5, 6, 6] == creatures.values()*.size()
```

2.2.5 Metaprogramming

In Groovy, a class is not a static template, that applies for every instance. Objects can be extended dynamically at runtime and use comprehensive meta programming paradigms. Groovy supports also compile time meta-programming since version 1.6 trough so called AST-Transformations. Here are some examples:

Listing 2.10: Using some meta-programming capabilities of Groovy

```
// Intercepting method dispatch
class AllMethodAcceptor {
   def defined() { "This method is defined" }
   def methodMissing(String name, args) {
      def params = args.collect {
         it.class.simpleName
      }.join(", ");
      return "Method $name($params) is not defined"
   }
}

def acceptor = new AllMethodAcceptor();
assert acceptor.defined() == "This method is defined"
assert acceptor.foo() == "Method foo() is not defined"
assert acceptor.bar(42) == "Method bar(Integer) is not defined"

// Using ExpandoMetaClass to expand classes
Integer.metaClass.isTheAnswer = {
   delegate==21? "only half" :
      delegate==42 ? "yes!" : "not at all"
}

assert 23.isTheAnswer() == "not at all"
assert 21.isTheAnswer() == "only half"
assert 42.isTheAnswer() == "yes!"

// Using AST Transformations to mixin methods
class VerticalStarter {
    def start() { "starting vertically" }
}

@Mixin(VerticalStarter)
class Aircraft {
    def fly() { "over the clouds..."}
}

def aircraft = new Aircraft();
assert aircraft.start() == "starting vertically"
assert aircraft.fly() == "over the clouds..."
```

There are lots of other possibilities and tricks using metaprogramming, that can't be explained in detail here. For more information about the metaprogramming capabilities refer to the online documentation [Groovy] or one of the many cool books about Groovy, like [GinA], [Ven08] or [JNS08].

2.2.6 Builder

To get a feeling for what metaprogramming can be used, builders are a good example. The idea behind builders is to use a domain specific language (DSL) to create structured trees. XML or user interfaces are classical examples of structured trees, but there are also lots of other scenarios. This example shows the generation of XML using the `MarkupBuilder` class:

Listing 2.11: XML Builder usage

```groovy
def writer = new java.io.StringWriter()
def xml = new groovy.xml.MarkupBuilder(writer)
def languages = ["Java", "Groovy", "Scala"]

xml.languages() {
   languages.each {
      name(it)
   }
}

def expected = """<languages>
  <name>Java</name>
  <name>Groovy</name>
  <name>Scala</name>
</languages>"""
assert writer.toString() == expected
```

It's also possible to self define builders or even create complete domain specific languages with Groovy.

2.3 Groovy and Java integration

One of the prior design goals of Groovy was a close integration with Java code. As a result, Groovy classes compile to standard Java bytecode that's in nearly in any situation fully usable from Java. This is a big difference to other dynamic languages that were ported onto the JVM. Most of those require some sort of proxies or bridges. Because of this, Groovy is compatible to Java and vice versa. For example a class defined in Java can extend a class defined in Groovy or the other way around.

However, there are a few things that require attention. If Groovy code runs in the standard JVM, `groovy.jar` always has to be loaded into the classpath. The same thing applies when Java code depends on Groovy classes. When launching Groovy code directly with the `groovy` command, this step is done automatically. GroovyScripts can be called programmatically from Java, but they have to be loaded with a `ScriptEngineManager`. Further, because Groovy has a mightier syntax, there are scenarios that require some special acrobatic tricks in Java. For instance all dynamic Groovy fields and methods appear to be or return `Object` for Java. Here's an example of calling a Closure from Java:

Java

```java
public class Caller {
  public static void main(String[] args) {
    Object obj = ClosureHolder.getCallMe();
    Closure callMe = (Closure)obj;
    callMe.call();
  }
}
```

Groovy

```groovy
class ClosureHolder {
   static def callMe = {
         println "Wheee, i'm from Groovy!"
   }
}
```

3 Architectural Overview

The following chapter provides an overview of the architecture for all `Crosslanguage Refactorings` between Groovy and Java.

3.1 Refactoring Participants

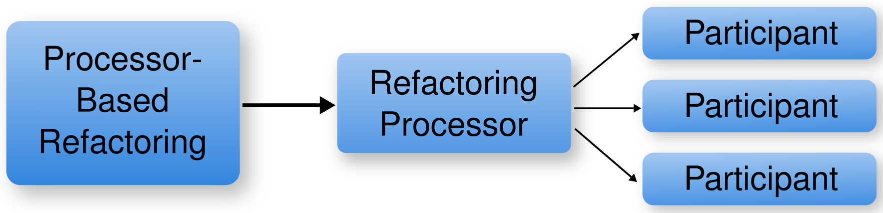


Figure 3.1: A split refactoring with a processor and 0..n participants

The complete architecture of the *Crosslanguage Refactoring Plug-In* is inspired by the processor based refactoring architecture of Eclipse. A refactoring according to this conception is split into one refactoring processor and zero to many refactoring participants. The refactoring processor holds the main implementation of the refactoring, for example the renaming of a Java class. It is further responsible to satisfy all refactoring participants. This involves the following steps for each participant:

1. Load the participant and check if he wants to contribute in the current refactoring.
2. When all user inputs are gathered and the refactoring is ready to be applied, check the status of each participant.
3. If the status is free of errors, collect all changes made by the participants. These changes then become an integral part of the whole refactoring.

3.2 Invocation and refactoring direction

Through the refactoring participant infrastructure, it's possible to become informed whenever a refactoring is launched and additions can be contributed to the refactoring. This extensibility matches the needs of our *Crosslanguage Refactoring Plug-in* perfectly. We decided to realize all refactoring extensions with refactoring participants. The only alternative would have been to make the refactorings themselfs aware of how to update the related code in the other language. Because we had no direct access to the JDT refactorings, this decision was quite easy. As a result of the participant based architecture, there are only two base cases we have to consider: A refactoring in Java or Groovy happens and the other language has to be updated as well. It's either on of those.

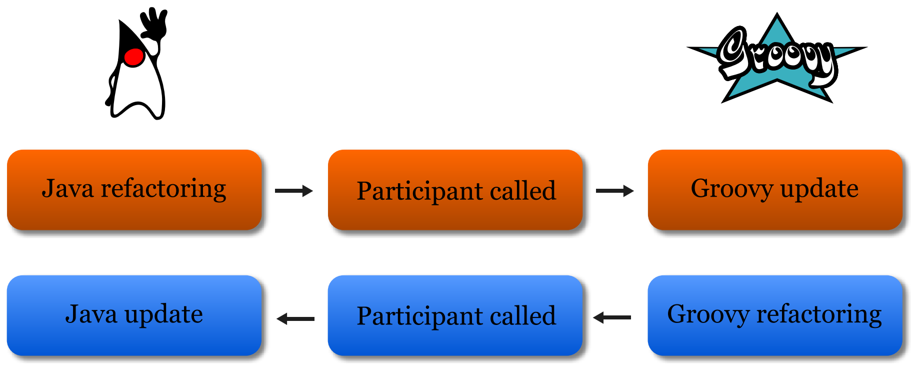


Figure 3.2: Workflow directions of Groovy and Java Refactorings

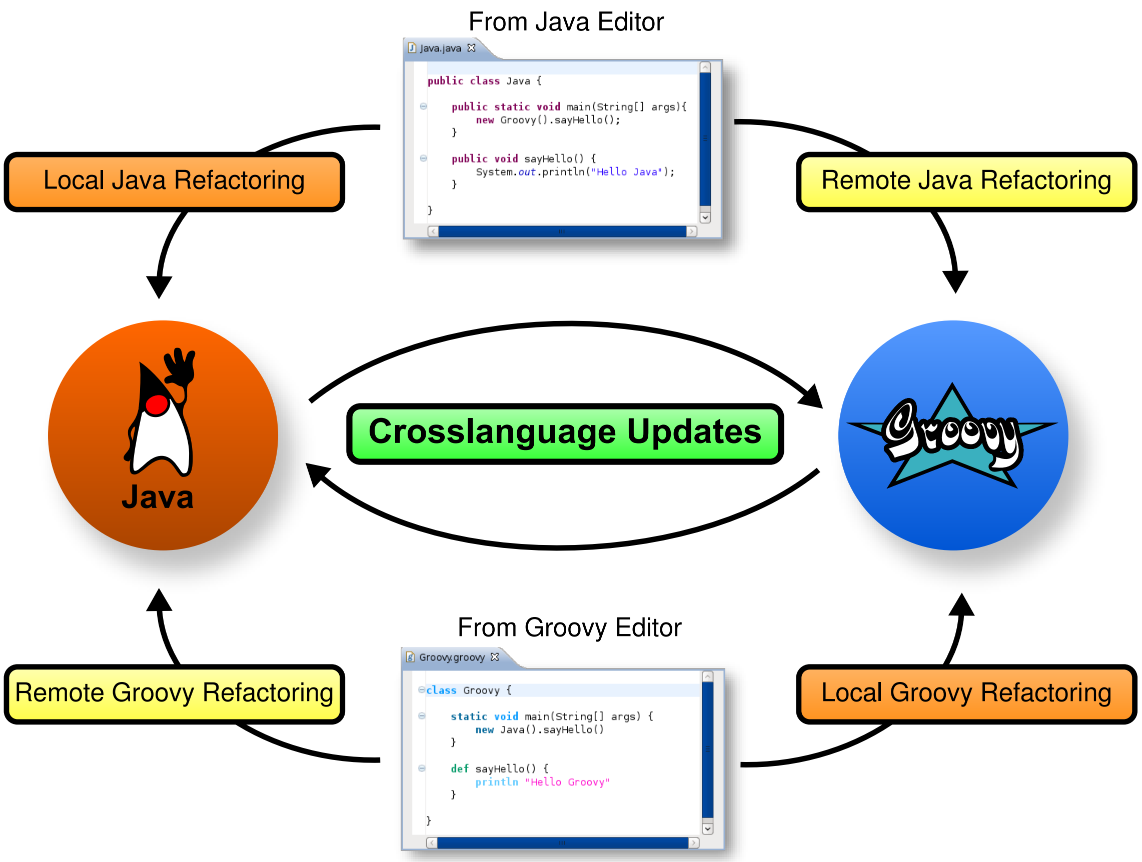
3.3 Refactoring scenarios

Already our predecessors [KKK08], saw the importance of a *crosslanguage refactoring* and wanted to implement it. They found a few problems, and time was running, so they decided to delay this part out of their thesis. They wrote a special chapter about the problems found, and even some fixing ideas.

A closer look to the *Crosslanguage Refactorings* shows, there basically are two different kind of elements (Java and Groovy) to refactor. These two different kinds need two completely different routines to refactor. But that's not all. To facilitate refactoring usage, one should be able not only to start it on a declaration, but from any reference. A reference can be everywhere, in a Java and in a Groovy file. So every kind of refactorings have to be able to start from both, Java and Groovy editors. So at the end, we have four different use cases. The splitting in four scenarios was already defined from our predecessors in the former bachelor thesis [KKK08]. The name of each use case was derived from the source of the refactoring to it's destination. For example "Rename a Groovy element from Java" and a number. However, these numbers and names proved to be a bit misleading and confusing during our bachelor thesis. Therefore, we decided to introduce our own terminology for each use case. We differentiate between local and remote refactorings.

- **Local** refactorings stay in the same language, like renaming a Java element in the Java editor.
- **Remote** refactoring work on references, for example renaming the same Java element from a Groovy editor where the specific element is used.

The following diagram shows an overview over these four use cases:



```java
public class Java {

    public static void main(String[] args){
        new Groovy().sayHello();
    }

    public void sayHello() {
        System.out.println("Hello Java");
    }

}
```

```groovy
class Groovy {

    static void main(String[] args) {
        new Java().sayHello()
    }

    def sayHello() {
        println "Hello Groovy"
    }

}
```

Figure 3.3: Use Case diagram of the crosslanguage refactoring

- **Chapter 4 (Local Java Refactoring):** A normal Java refactoring on a Java element, started from a Java editor that will also update references in Groovy.
- **Chapter 5 (Remote Java Refactoring):** Refactor a Java element out of a Groovy editor
- **Chapter 6 (Local Groovy Refactoring):** A normal Groovy refactoring on a Groovy element, started from a Groovy editor that will also update references in Java
- **Chapter 7 (Remote Groovy Refactoring):** Refactor a Groovy element out of a Java editor

3.4 Package Structure

The main part of our developed solution resides in the sub package of the Groovy refactoring, at `org.codehaus.groovy.eclipse.refactoring.core.jdtIntegration`. There was also a lot of existing code that needed to be heavily altered like the Refactoring Dispatch, but the core classes are all located here.

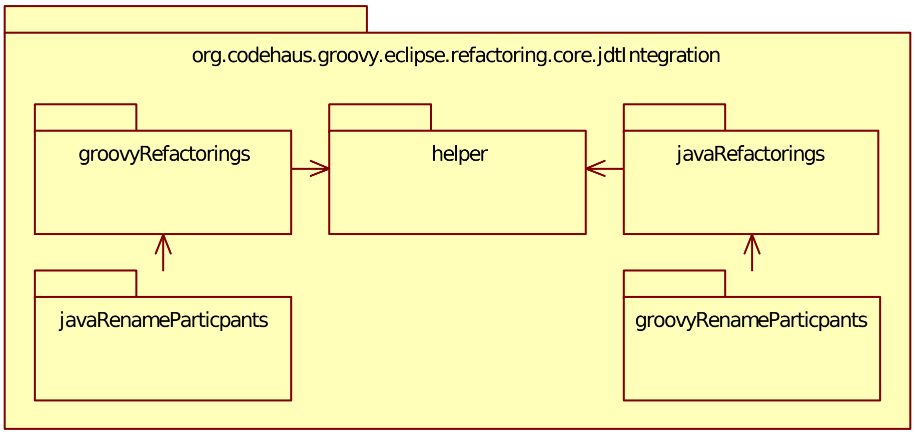


Figure 3.4: The package structure of the participants

The subpackages are segmented according to their responsibilities. Following is an overview of what each package is in charge of:

- **javaRenameParticipants** holds the participants to catch a Java rename refactoring an launch an appropriate refactoring on the Groovy side if necessary. The changes of the Groovy refactoring will be returned to the superior Java Refactoring
- **groovyRefactorings** contains the so called Refactoring-converters. Based on a Java element from the Eclipse Java model, these Converters create a Groovy refactoring that is able to refactor an element in Groovy code. The package is essentially used for Local Java Refactorings when groovy sources must be updated and for Remote Groovy Refactorings launched from a Java editor.
- **helper** is a shared utility package offering various service classes used by both Groovy and Java refactoring packages.
- **javaRefactorings** are simple AST based Refactorings to update a refactored Groovy element within Java code. For example if a Groovy method is renamed, these refactorings will look up all invocations of the renamed method an refactor them to the new name.
- **groovyRenameParticipants** keep track of all rename refactorings affecting Groovy code. When a refactoring is executed, the participants will launch the corresponding Java update refactoring.

3.5 Java Search

To search a Java or Groovy element in the source code, the *Java Search* can be used. It just needs a few parameters, which are:

- The **`SearchEngine`** is the entry point of the JDT search API.
- The **`SearchPattern`** will get explained in chapter 3.5.1 SearchPattern.
- The **`SearchScope`** defines where the searched element is supposed to be. Mostly this is an already defined Java element, but can also be the whole workspace.
- The **`SearchRequestor`** collects the search results. Everytime an element matches the pattern and the scope, the `acceptSearchMatch(SearchMatch match)` function is called. With the parameter match, the element can be analyzed and if it is usable, it can be added to the result collection.

Listing 3.1: Example of a search

```java
SearchEngine engine = new SearchEngine();

IJavaSearchScope scope = SearchEngine . createWorkspaceScope ();

final List<IField> results = new LinkedList<IField>();

SearchRequestor requestor = new SearchRequestor() {
  public void acceptSearchMatch(SearchMatch match) throws CoreException {
    Object element = match.getElement();
    if (element instanceof IField) {
      results.add((IField)element);
    }
  }
};

SearchParticipant[] participants = new SearchParticipant[] {
    SearchEngine.getDefaultSearchParticipant() };

engine.search(pattern, participants, scope, requestor,
    new NullProgressMonitor());
```

3.5.1 SearchPattern

With a search pattern object, it's easy to define the element, that is supposed to be found. It gets created via the static method `createPattern` from the class `SearchPattern()`. Obviously the more precise the pattern is, the less elements will be found. The details about how to initialize a `SearchPattern` can be found in the JDT documentation.

Listing 3.2: Defines a specific search pattern

```
private List<IType> searchAllJavaFields() throws CoreException {
  SearchPattern pattern = SearchPattern.createPattern(
      renameClassNode.getName(),          // stringPattern to search for
      IJavaSearchConstants.TYPE,          // searching for a type
      IJavaSearchConstants.DECLARATIONS,  // search limits, just declarations
      SearchPattern.R_EXACT_MATCH);       // match rule

  JavaModelSearch search = new JavaModelSearch(project, pattern);
   return search.searchAll(IType.class);
}
```

3.5.2 JavaModelSearch

The `Java Search` is used in a lot of situations along all the different `Crosslanguage Refactoring` scenarios. So we encapsulated it in a helper class called `JavaModelSearch`. It just needs a `SearchPattern` and a `IJavaProject` object as the `Search Scope` to search.

The `JavaModelSearch` class has just two public methods.

- `searchAll` searches for all elements, and returns a list with the result.
- `searchFirst` returns the first element of the `searchAll` method.

As arguments for both methods, it is necessary to give a class-object from the type of elements you're searching.

Listing 3.3: Search example for a specific search pattern

```
  JavaModelSearch search = new JavaModelSearch(project, pattern);
   return search.searchAll(IType.class);
```

4 Local Java Refactoring

4.1 Introductory example

This chapter describes the implementation of *Local Java Refactorings*: When a Java element gets renamed from the usual Eclipse Java editor and related Groovy code has to be updated. For example we have a Java class `ShiftIncrement` which has a value that can be directly accessed and incremented by one left shift. This class is used from a small Groovy test script. (Please note that this example is kept simple for the purpose of clarity.)

Java

```
public class ShiftIncrement {

   public int value;

   public void increment() {
      value = value << 1;
   }
}
```

Groovy

```
def counter = new ShiftIncrement()
counter.value = 2
3.times { counter.increment() }
assert counter.value == 16
```

Now if either the `ShiftIncrement` class, the `value` field or the `increment()` method get's refactored, the *Crosslanguage Refactoring Plug-in* will have to take care of the Groovy script being updated correctly. Let's assume that the `increment()` method should be renamed to `inc()`. Notice that the `counter` variable in the Groovy script is dynamically typed, so we don't know if the object we're invoking the method on, is defined in Java or in Groovy. The result after the refactoring process should be:

Java

```
public class ShiftIncrement {

   public int value;

   public void inc() {
      value = value << 1;
   }
}
```

Groovy

```
def counter = new ShiftIncrement()
counter.value = 2
3.times { counter.inc() }
assert counter.value == 16
```

4.2 Use Case

The user renames a Java element in a mixed project. If the refactored Java element has references in Groovy, these references have to be updated as well. In this use case, the user starts a rename refactoring from the JDT editor.

Preconditions

The Project has to be in a clean state. That means, all sources need to be compilable without build errors.

Postconditions

All occurrences of the Java element in Java and Groovy files are renamed and the project is in a clean state again, without build errors.

Basic course of events

1. Start the JDT rename refactoring from a Java editor.
2. After all the Java elements are renamed, a rename participant starts the corresponding Groovy refactoring.
3. The existing Groovy refactoring updates all the Groovy references.

Alternative paths

- **If a reference in Groovy is dynamically typed**, the refactoring can't automatically identify them. In this case, the user can select all the candidates he wants to rename. It's the users own responsibility, to take care about which candidates he want to rename.

4.3 Eclipse Extension Point

Eclipse is fundamentally based on the plug-in concept and therefore able to provide a high level of flexibility and extensibility. The platform itself is actually free of any programming language specific parts. All supported languages and tools are loaded as independent sets of modules. For Java this is the JDT Plug-in and to get C++ support, the CDT Plug-In is the way to go. To provide such a high modularity, a reliable interface to connect these features is required.

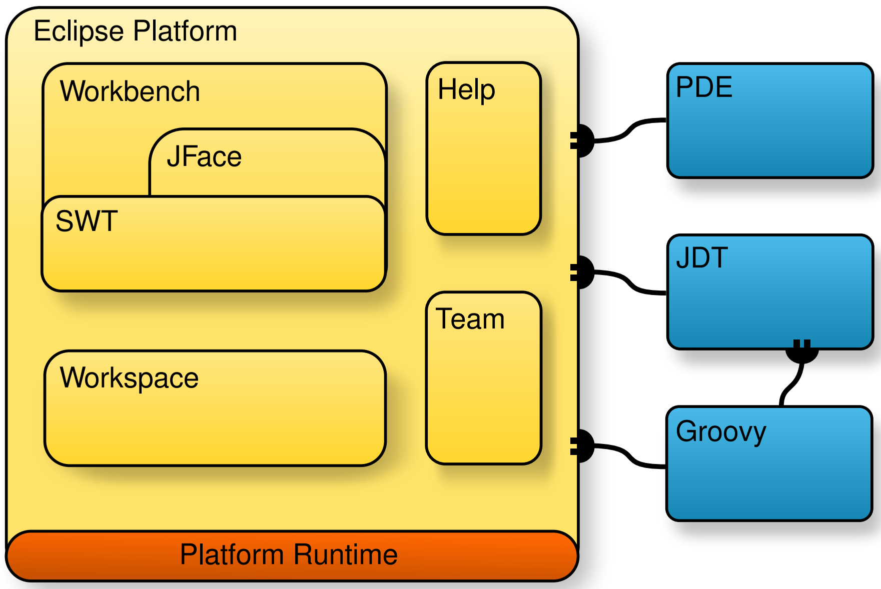


Figure 4.1: Plug-in based architecture of Eclipse, taken from the *Eclipse Foundation*

In terms of Eclipse, this is solved over the extension point API. By using pre-existing extension points, plug-ins can be linked with others or the runtime itself. Extension points really are comparable to sockets, where the plug-ins can, plug themselfs in. A plug-in can also define it's own extension points and open itself for extension this way. This is probably one of the reasons why Eclipse has so many attractive plug-ins to offer and is one of the most used IDE's.

Like most configuration tasks of Eclipse plug-ins, registering at an extension point happens trough the plugin.xml. For the *Crosslanguage Refactoring Plug-in*, we mainly used the rename participant extension point offered by the JDT refactorings. This extension point is used to register new rename participants for the JDT Refactorings. To differentiate the various kinds of rename refactorings, a type checking enablement can be defined. This works like a classical instanceof, as the XML tag suggest.

Listing 4.1: Extension point configuration for the field rename participant

```
<extension point="org.eclipse.ltk.core.refactoring.renameParticipants">
  <renameParticipant class="org.codehaus.groovy....FieldRenameParticipant"
    id="groovy.eclipse.refactoring.FieldRenameParticipant" name="FieldRename">
    <enablement>
      <with variable="element">
        <instanceof value="org.eclipse.jdt.core.IField" />
      </with>
    </enablement>
  </renameParticipant>
</extension>
```

4.4 Participant implementation

With the refactoring participants now properly registered to their corresponding extension points, they're now able to provide some additional changes for their parent refactorings. In our case this would be to take care of all necessary updates in Groovy that arise because of the renamed Java element. Therefor we have to check if the Java element is used somewhere in Groovy code of the same project. If any references were found, those will have to be refactored. If no references are around in Groovy, the participant should stay calm. Our first prototype to do so, was based on the idea to traverse all Groovy files and look up any references.

For this, we implemented some simple AST Visitors who could detect and collect such references. When the first referencing node was detected, it was possible a Groovy refactoring on that node and declare that the participant would be active for the current refactoring.

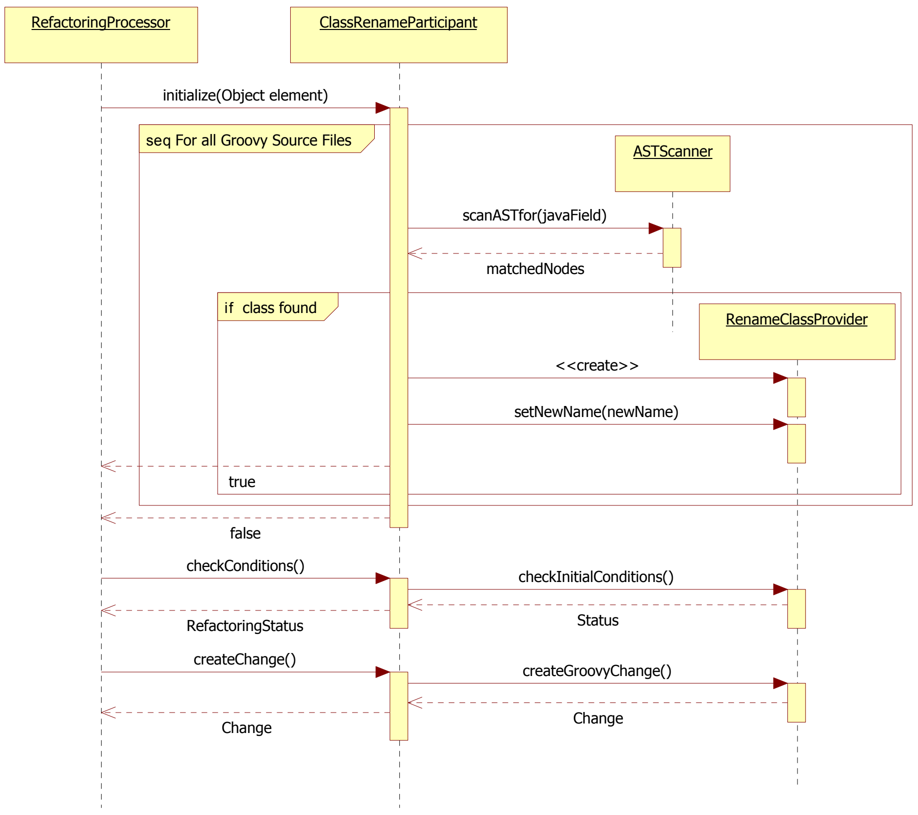


Figure 4.2: Participant flow as prototyped

This basically does the job, but there's one drawback with this solution: To find a reference, all Groovy files from the same project have to be visited. Then, once the Groovy refactoring is initialized, they'll have to be visited again to lookup all required renames. So we visit each Groovy file in the project twice, which can be very extensive in large projects. To avoid this situation, we decided to push the decision whether a refactoring is required or not, down to the refactoring provider itself. The refactoring provider already has the necessary information present, everything it needs to do is count all considered renames. If there is any rename, he has obviously some work to do. Therefore we implemented a `hasCandidates()` method for all rename providers and were now closer to the final design of our rename participants. The remaining tasks of particpants were to initialize a refactoring provider for the renamed Java element, set the new name passed with the arguments of the parent refactoring and check if there are any candidates to rename. That way, the participants are kept lightweight and free of unnecessary complexity.

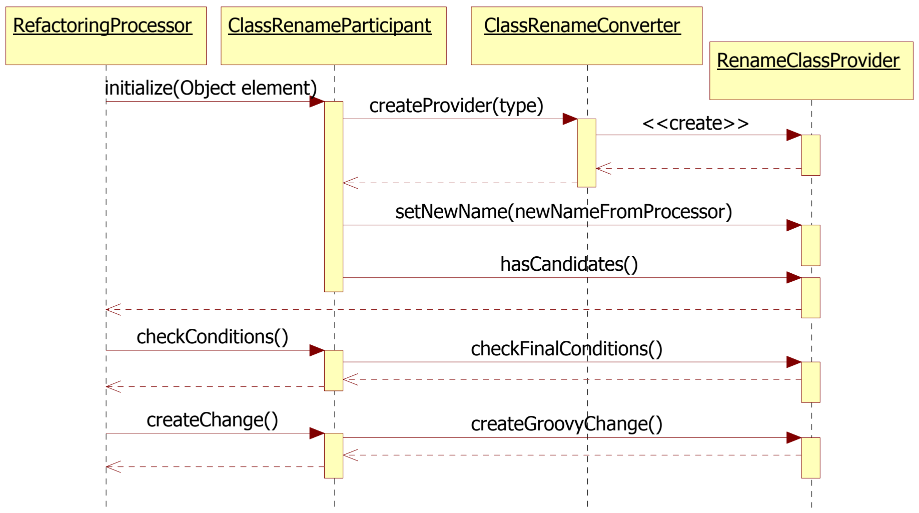


Figure 4.3: Participant flow as implemented

4.5 Refactoring Converter

The remaining complexity consisted in the initialization of a Groovy refactoring for a `IJavaElement`. The implementation of all Groovy refactoring providers are intended to be initialized with either an `ASTNode` or a pattern class[1]. We decided to detach this responsibility to so called refactoring converters, as can be seen in fig 4.3. Given a specific Java element, i.e. a class, field or method, the job of a converter is to create an appropriate Groovy refactoring provider to rename all references to that element.

The first challenge was that all present Groovy refactoring providers required a user selection to be instantiated which we didn't have. After a closer look at the existing code we discovered, that the selection wasn't actually used anywhere. The reason why it still needed to be passed in was simply overstressed inheritance: The abstract superclass of all refactoring providers, *RefactoringProvider*, requested a selection, although it was used only by Extract Method and Inline Method refactorings . So we made that parameter optional by adding an additional constructors who left out the user selection.

The next step was to prepare the refactoring element, in other words the element indicating what should be reanmed. This varied for each element to be refactored and needed the following steps:

- **Class** Acquire the fully qualified name from the Java type and create a `ClassNode` (Groovy's AST node to represent all kind of types).
- **Field** Create a new `FieldPattern` by determining the declaring class and the name of the field.
- **Method** First get a list of all parameters and then create a `MethodNode` (Groovy's AST node to represent method declarations) with the same name, return type, parameters and modifiers. Then determine the declaring class and with the rebuilt method node, create a `MethodPattern`.

A tricky part with fields and methods was, that the declaring class also has to obtain the same type hierarchy as the Java representative. To achieve this we introduced the helper class `HierarchyBuilder` which recursively ascends the supertypes of a class, until `java.lang.Object` is reached. With each ascent, the type hierarchy gets extended by the newly visited superclass. This way the hierarchy gets properly built and the prepared element fulfills all requirements of the refactoring to be properly renamed in Groovy.

[1]Either `FieldPattern` or `MethodPattern`

4.6 Ambiguous Candidate Selection

Dealing with ambiguities

It lies in the nature of dynamic languages such as Groovy and any other dynamic language, that there may be situations where a refactoring can't cleary determine the necessary editing steps without some help.

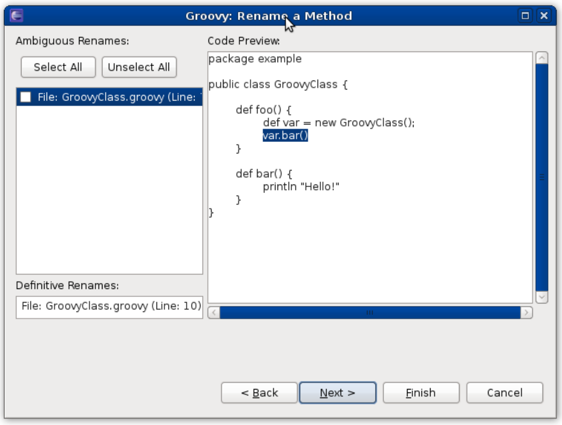
Whenever types can not be determined, a refactoring may have no clue how to handle such a statement. Imagine the following situation, where the method `JavaClass.bar()` gets renamed:

```
JavaClass foo = new JavaClass();
foo.bar()
```

It's obvious that the second call has to be updated, as `foo` is of type `JavaClass`. But what if the type can't be determined?

```
def foo = RandomFactory.createWhatSoEver();
foo.bar()
```

There's no chance to tell what type `foo` will actually be of, but it might be `JavaClass`, so should we refactor the second statement or not?. There are basically two known options to address this problem: Type inference or additional user input. Type inference means to determine the type of a variable trough the inner logic of the code. This technique is commonly known from functional programming languages. In our example, it still would be impossible to infer the type of `foo` if `RandomFactory` really returns a random object, thus need some additional user input and that's exactly what our predecessors [KKK08] did. They implemented a selection dialog where all ambiguous candidates are listed and can be previewed. The refactoring operator then decides which elements should be renamed and selects them.



```
package example

public class GroovyClass {

	def foo() {
		def var = new GroovyClass();
		var.bar()
	}

	def bar() {
		println "Hello!"
	}
}
```

Figure 4.4: Selection of ambiguous candidates for Groovy refactoring

Implementation in Crosslanguage Refactorings

This problem is also present in the *Crosslanguage Refactoring Plug-in*. If a Java element that is used in a dynamic manner within Groovy gets refactored, the candidate selection dialog should pop up as well. The point is that the ambiguous candidate selection dialog is implemented as `RefactoringWizardPage` and only included in the Groovy refactorings. When a Java element is refactored, a JDT internal refactoring wizard is executed of course and this wizard does not assume that participants of the underlying refactoring need additional wizard pages to gather any user input. We researched a lot in the Eclipse documentation and asked the newsgroups for help, but neither of those could give us a detailed answer. It seemed like dynamically extending refactoring wizards trough refactoring participants isn't allowed in the refactoring architecture of Eclipse. This partly does make sense because it would brake the layer principle: Refactoring implementations shouldn't deal with the UI stuff. Still, we needed a solution to fix this problem. There were basically two options to go:

- When a Java refactoring is activated, launch a separate Groovy refactoring wizard and handle both steps independently. This would have the additional benefit that the Groovy refactoring could be canceled. However, the participant would then be downgraded to a simple refactoring update listener. The preview of a Java refactoring would show only the java refactoring steps and undo would consider only those. To us, this didn't feel like the integral flow one would expect while refactoring.
- The alternative would be to pop up the selection dialog as self-contained window if necessary during a refactoring. Self-contained means on top of the current refactoring wizard. This isn't the best solution either, but at least the Groovy refactoring then remains a integral part of the java refactoring and contributes changes to it.

We decided to strive for the second way. Now the thrilling question was how to get the ambiguous selection dialog displayed without being included into a refactoring wizard. We even struggled with displaying a simple message dialog from a refactoring participant at all. The problem was that participants always run in separate threads so they can do no harm to their parent refactorings. But to launch a new window we need the shared Eclipse workbench shell, and this can not be a accessed outside of a GUI thread.

Listing 4.2: Display a message window outside of a GUI thread

```java
PlatformUI.getWorkbench().getDisplay().syncExec( new Runnable() {
   public void run() {
      IWorkbenchWindow window = workbench.getActiveWorkbenchWindow();
      Shell shell = window.getShell();
      MessageDialog.openInformation(shell, "Info",
         "Ambiguous candidates detected!");
   }
});
```

To be able to access the Eclipse workbench shell, we need to tell SWT to launch a new GUI-thread as shown in listing 4.2. From this thread, we can now launch the ambiguous candidate selection dialog. Thanks to J.-P. Pellet from the JDT newsgroup for this hint!

The remaining question was how to present the dialog. As a `RefactoringWizardPage`, the selection dialog is tied pretty closely to a `RefactoringWizard`. Running the selction in a normal wizard without the refactoring functionality didn't work because it requires the parent wizard to be a refactoring wizard. Extracting the wizard page elements to a single, self sustainable dialog would have been a possibility, but then we would have ended up with two different classes basically doing exactly the same thing. This is generally never a good idea when trying to keep maintainability in mind.

Our solution for this problem is finally quite simple. We decided to use just a usual `RefactoringWizard` that doesn't perform any changes. This way it behaves quite like a dummy wizard with additional functions such as previewing changes, but doesn't apply them to any source files. To prevent it from performing changes, we only needed to overwrite the `performFinish()` method. Now why should this help us any further? What matters is the refactoring provider in the background, that gets configured by the ambiguous selection dialog. After the dummy wizard is finished, the refactoring provider knows which ambiguous candidates should be renamed. The rest of the refactoring participant remains as before. It only has to check if the refactoring provider has any ambiguous candidates. If this is the case, it launches the dummy refactoring wizard action and continues after the selection is complete. This behaviour was realized in the superclass of all Java rename participants, so it's available for all participants.

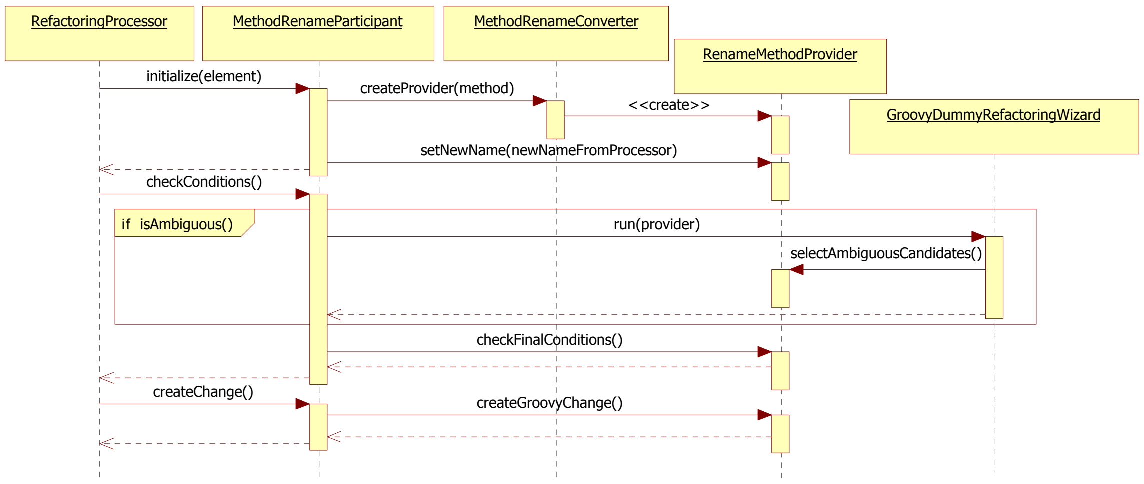


Figure 4.5: Complete workflow of a `MethodRenameParticipant`

4.7 Implementation Review

The following class diagram shows the most important classes discussed in this chapter:

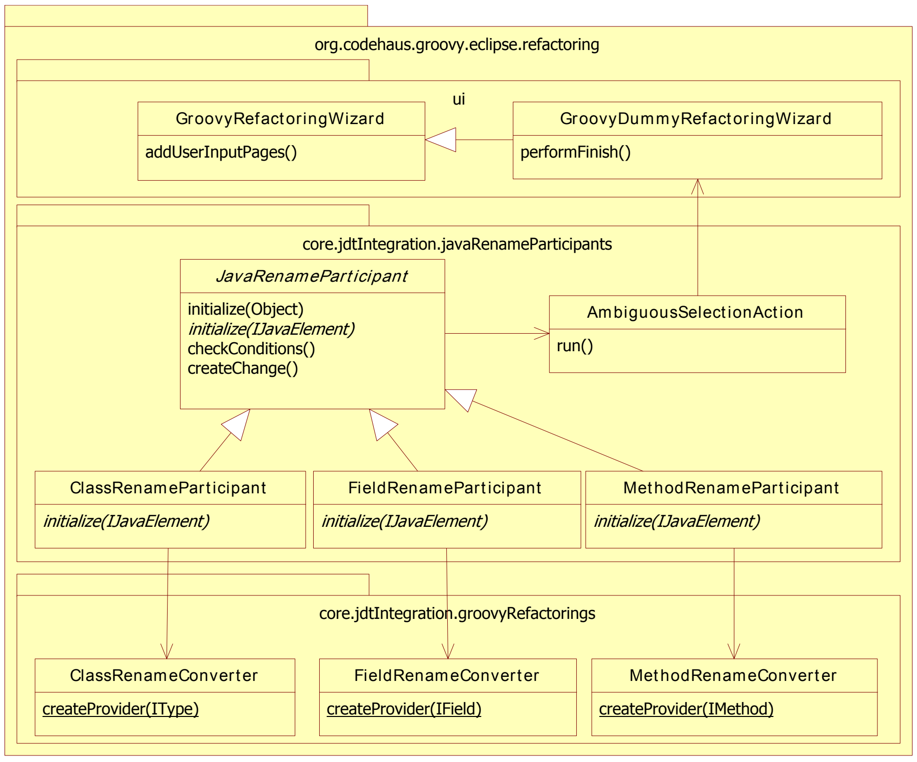


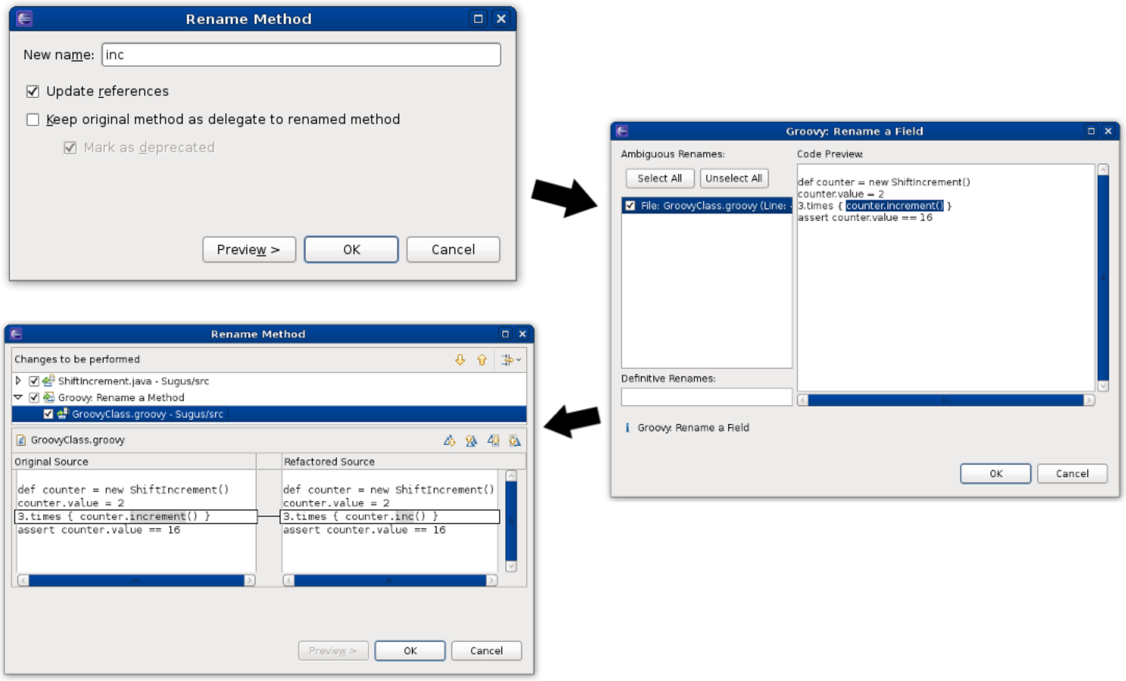
Figure 4.6: Overview of involved classes in local java refactorings

The responsibilities of the classes are as follows:

- **JavaRenameParticipant** is a general class with shared tasks common to all rename participants. It hold's the participants `initialize()` method to check if the Java refactoring should update references or not. Then the template method for initializing the participants implementation is called and the refactoring provider is acquired. Further the condition checking and change creation is delegated to the refactoring provider. When the refactoring is ambiguous, an `AmbiguousSelectionAction` is launched.
- **Participant Implementations** check the elements to be refactored and call the specific converter to create a refactoring provider for that element.
- **Refactoring Converters** process the elements from the Java model to refactor it in Groovy.
- **AmbiguousSelectionAction** launches a dummy wizard to present ambiguous candidates to the refactoring operator who then selects all those to be renamed.

4.8 In Action

In reference to the introductory example from section 4.1, we're now ready to refactor the `increment()` method to `inc()` in Java. As we remember, in Groovy we have a dynamically typed variable to our Java class `ShiftIncrement` calling the `increment()` method. Because the variable is dynamically typed, we cannot be sure that this call should be renamed too. The following figure shows the refactoring as seen by the user:


Rename Method

New name: inc

☑ Update references

☐ Keep original method as delegate to renamed method

☑ Mark as deprecated

Groovy: Rename a Field

Ambiguous Renames:

☑ File: GroovyClass.groovy (Line:

Code Preview:

```
def counter = new ShiftIncrement()
counter.value = 2
3.times { counter.increment() }
assert counter.value == 16
```

Definitive Renames:

Groovy: Rename a Field

Rename Method

Changes to be performed

☑ ShiftIncrement.java - Sugus/src

☑ Groovy: Rename a Method

☑ GroovyClass.groovy - Sugus/src

GroovyClass.groovy

Original Source

```
def counter = new ShiftIncrement()
counter.value = 2
3.times { counter.increment() }
assert counter.value == 16
```

Refactored Source

```
def counter = new ShiftIncrement()
counter.value = 2
3.times { counter.inc() }
assert counter.value == 16
```

Figure 4.7: Example screens of a local Java refactoring

If there are other `increment()` methods without any paremeters used in our Groovy code, they would appear in the candidate selection dialog too.

It's further possible to the refactor Java supertypes of Groovy classes and all derived elements, such as an interface from a method, will be renamed.

4.9 Unsolved Problems

One issue we couldn't solve during our term project is an error message appearing when refactoring inherited Java methods. If a method in Groovy is derived from a Java interface, abstract or normal class, there's an annoying warning message shown, if the method is refactored. Here's an example:

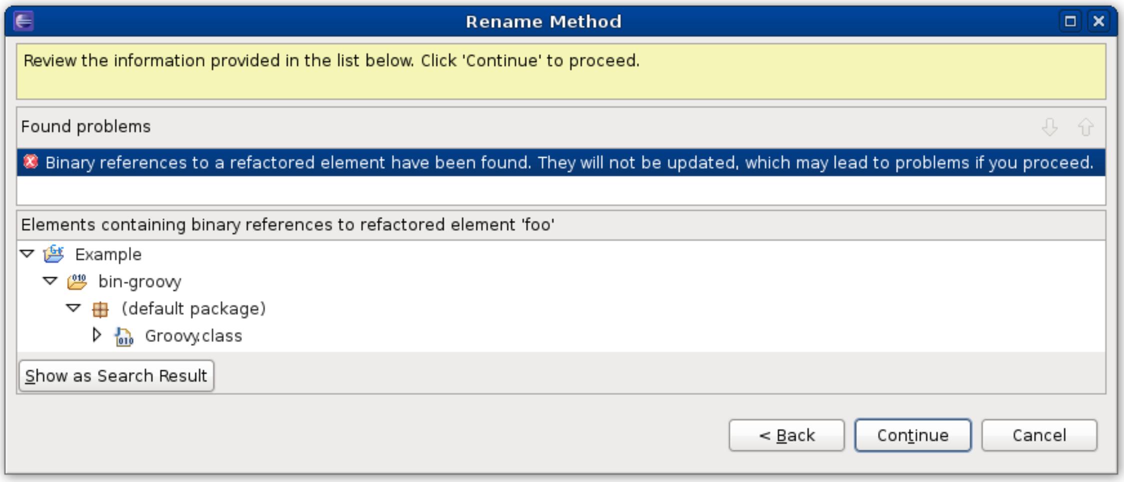
Java

```java
abstract class Java {
   public void foo() {
      System.out.print("Hello");
   }
}
```

Groovy

```groovy
class Groovy extends Java {
   public void foo() {
      super.foo()
      println " ${getClass().name}!"
   }
}
```

If the method `foo()` is refactored in Java, the overriding method in Groovy should be renamed too. Our rename participant would be able to do so. The implementation of the JDT refactoring always looks for any binary class file that overrides the method to be renamed. The Groovy class is only visible as class file for the Java refactoring, so it assumes there is no source for it available and shows the following error message:


Rename Method

Review the information provided in the list below. Click 'Continue' to proceed.

Found problems

Binary references to a refactored element have been found. They will not be updated, which may lead to problems if you proceed.

Elements containing binary references to refactored element 'foo'

- Example
  - bin-groovy
    - (default package)
      - Groovy.class

Figure 4.8: Warning message appearing when refactoring an inherited method

If we continue now, everything works as expected. This warning message was introduced in Eclipse 3.4 and is hard-coded in the JDT refactorings. Hard-coded means there is no way to gently disable it in a config file or in a programmatical way. Even though it only appears when refactoring inherited methods, it's still very present and disturbing.

4.10 Inline editing vs. refactoring wizards

Eclipse offers the possibility for Java refactorings to choose, if a developer wants to have a rename wizard or directly enter the new name inline into the editor. Both of these choices work great with the *Groovy Plug-in*, but if there are ambiguous candidates found, the ambiguous candidate selection window pops up, no matter which option is activated. This is the desired behaviour, because we need additional user input in this case.

4.11 Further Ideas

Here are some starting points how this use case could be further improved:

- **Preference page** The Groovy participants of our solution are always active. Maybe a user wants to deactivate them to temporarily work isolated in the Java domain.
- **Interproject compatibility** Because of the whole Groovy Plug-In not being capable of working with dependencies between different projects yet, the refactoring participants only consider Groovy code from the same project. This could be extended when the plug-in gets inter-project compatible.

5 Remote Java Refactoring

5.1 Introductory example

A *Remote Java Refactoring* means refactoring a Java element such as a type, method or field from Groovy. There is a strong relation to *Local Java Refactorings*, except that the starting point is a different one. However, developers used to rename refactorings, rely on them to work whenever they operate on a reference to a element from the same project. Why navigate to the mentioned element, when we can rename it right where we use it? Well, the biggest challange to do so is to be sure if we deal with something defined in Groovy or in Java. Imagine the following scenario:

Java

```
public class Dog {
   public void yell() {
      System.out.println("Wuff!");
   }
}
```

Groovy

```
class ComicFox {
   def yell() {
      println "Chunky bacon!"
   }
}

Dog bo = new Dog()
bo.yell()

def presidentsDog = bo
presidentsDog.yell()
```

Here we have class `Dog` defined in Java and a Groovy script where a class `ComicFox` is defined and the Java `Dog` is used. Both animals can `yell()` [1] at us, although they aren't related in any way. Now if the `yell()` method of `Dog` should be renamed to `bark()` or something, we have two cases in the Groovy script where methods have to be updated.

The first variable is statically typed to `Dog` whereas the second one is a dynamically typed reference to the same `Dog`. In the first case, we can figure out that we're operating on a Java type, but what about the second case? From the compilers point of view, it could also be the case, that we want to hear a `ComicFox` yelling, because seeing a dynamically typed variable there is no telling.

[1]`ComicFox`es actually do yell like that, see `http://poignantguide.net`

The result of renaming `yell()` to `bark()` should look like this:

Java

```java
public class Dog {
   public void bark() {
      System.out.println("Wuff!");
   }
}
```

Groovy

```groovy
class ComicFox {
   def yell() {
      println "Chunky bacon!"
   }
}

Dog bo = new Dog()
bo.bark()

def presidentsDog = bo
presidentsDog.bark()
```

5.2 Use Case

The user is working in a mixed project, within a Groovy file and renames an element, defined in Java. The declaration in Java has to be renamed, as well as all references in Java and Groovy.

Preconditions

The Project has to be in a clean state. That means, all sources need to be compilable without build errors.

Postconditions

The element and all references are renamed and the project is in a clean state again, without build errors.

Basic course of events

1. The User starts the rename refactoring from a Groovy editor.
2. The refactoring seaches all possible candidates from the selected reference and shows a list to the user.
3. If the user choose a Java element out of the list a JDT refactoring on this element is started.
4. After all the Java elements are renamed, a rename participant starts the corresponding Groovy refactoring.
5. The existing Groovy refactoring updates all references in Groovy.

Alternative paths

- **If the user chooses a Groovy element** out of the list of possible candidates, the scenario in chapter 6 (Local Groovy Refactoring) is started.
- **If a reference in Groovy is dynamically typed**, the refactoring can't automatically identify them. In this case, the user can select all the candidates he wants to rename. It's the users own responsibility, to take care which candidates he wants to rename.

5.3 Detecting Java elements within Groovy code

As Java and Groovy classes produce nearly equal bytecode, how can a class defined in Groovy be distinguished from a one that's defined in Java? Simply by the fact that all Groovy classes implement the `groovy.lang.GroovyObject` interface. At runtime, this can be approved by a simple `instanceof` check:

Listing 5.1: Identifing Java types within Groovy code

```
def obj = { println it } // A Closure
assert (obj instanceof GroovyObject) == true

obj = "Hello ${this}" // A GString
assert (obj instanceof GroovyObject) == true

obj = "Eat more bananas!" // A normal Java String
assert (obj instanceof GroovyObject) == false

obj = new XmlSlurper() // A Groovy utility class defined in Java
assert (obj instanceof GroovyObject) == false
```

This works for any Groovy object. The main challenge of this use case is to find a reliable way to detect Java elements within Groovy code. So, does this detection help us to find Java objects used in Groovy? Or expressed otherwise: How can we predict that when working on a Groovy project within Eclipse, the selected element is actually defined in a Java file?

In case of the Groovy refactorings for Eclipse, we can't perform such checks like in runtime, because there is no runtime, or at least a simulation. But there is another opportunity: The Abstract Syntax Tree[2].

When implementing refactorings, the first step in the chain of events is mostly to find the piece of code that should be refactored. In the *Groovy Eclipse Plug-in*, a refactoring is currently always launched form the source editor. All rename refactorings of Groovy expect the selection to point at either a declaration or a reference to a class, method, field or local variable. After the rename refactoring is launched, the first step is to get the Abstract Syntax Tree of the edited document and to search the corresponding `ASTNode` of the selection. With the `ASTNode` found, a refactoring dispatcher then decides, which specific refactoring should be executed. This dispatching process is discussed more detailed in the next section. The point for now is that each rename refactoring is initialized with a subkind of `ASTNode` at a particular time.

The question is now if we can, based on the AST, determine if the selected element is defined in Java or Groovy. Indeed we can, because interfaces are represented in the AST as well. A Groovy class is represented as `ClassNode` in the AST, and this class has even a direct method `isDerivedFromGroovyObject()`.

[2]For more information on how Groovy ASTs work, please refer to the thesis report from our predecessors [KKK08] and the javadoc of the package org.codehaus.groovy.ast

5.4 Prototype: Enhanced rename dispatcher

Our first idea on how to implement *Remote Java Refactorings* was: “If we can find classes defined in Java within Groovy code, the refactorings can’t be that hard to implement, can they?”. When the selection is pointing to a `ClassNode`, we can directly verify whether we deal with a Groovy class or not. And with fields and methods we would have to grab the classes where they are declared and check the declaring class if it comes from Groovy or Java. This section describes the prototypes we created to prove this assumption.

5.4.1 Existing rename dispatcher

When programmers use rename refactorings, they expect that the appropriate refactoring is executed automatically. More specifically, if we select a method to be refactored, we don’t want to be asked if we currently try to refactor a local variable or a method or what so ever. To support this behaviour, we need an automatic dispatch of refactorings. For this reason, the *Groovy Eclipse Refactoring Plug-in* features the class `RenameDispatcher`, which automatically searches for the applicable refactoring based on a text selection within a Groovy file, it works as follows:

1. Get the selected `ASTNode` by parsing the AST using the visitor pattern.
2. Analyze the `ASTNode` and determine it’s actual type (using `instanceof` checks). In case of fields and methods, create a pattern to rename the given element.
3. Initialize and return an accurate refactoring to rename the given element.

This is only a simplified description, for a detailed implementation look at section 2.5.2 in [KKK08].

5.4.2 Generic extended rename dispatcher

As the existing rename dispatcher worked only for Groovy elements, the first plan was to extend the `RenameDispatcher` class so that Java refactorings would be created to. The return value from the dispatcher would then have to be changed from `GroovyRefactoring` to the more universal `Refactoring` type. That way, the action which launches the dispatcher and opens the `RefactoringWizard` would only require minimal extension.

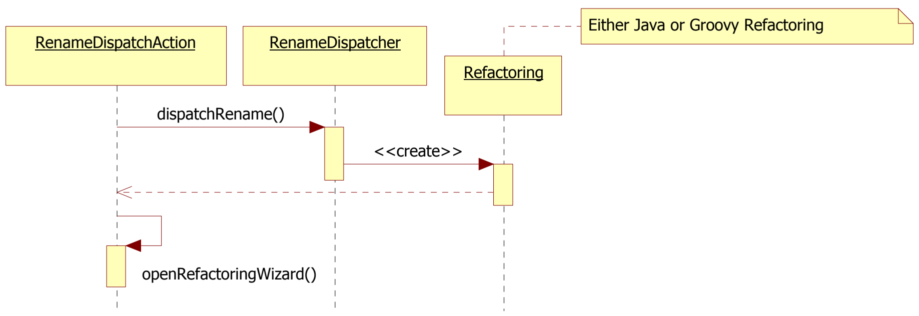


Figure 5.1: First idea for enhancing the rename dispatcher

As we began to implement this solution, we realized pretty fast that there were some drawbacks. A Java refactoring itself can be instantiated without a problem as described in chapter 8.5. But we also need to gather a new name from the user and this can only be achieved by launching a refactoring wizard. There are, of course, refactoring wizards implemented within the JDT, but the access to those is discouraged because they're considered as internal implementations. This way, the JDT plug-in tries to preserve the right to change them whenever needed and declares to potential plug-in vendors that they should not be used directly.

Further we would need to keep apart the two different kinds of refactorings for Groovy and Java elements because each refactoring requires it's individually initialised refactoring wizard. This would mean that the advantage of generalized refactorings is gone and we need to dispatch again.

5.4.3 Launching a JDT refactoring wizard programmatically

The previous section has shown that the refactoring wizards from the JDT plug-in should not be used directly. However, there is an alternative option to launch a Java refactoring wizard programmatically. The class `RenameSupport` is an implementation of the facade pattern according to [GoF95] to offer the refactoring functionality of the JDT-plug-in. A refactoring can either be directly applied or a refactoring wizard may be launched. Here's an example of launching a refactoring wizard for a Java field:

```
IField field = /* The field to be renamed, acquired from the Java model */
Shell shell = /* The parent Eclipse SWT shell */

RenameSupport fieldRename = RenameSupport.create(field, "newFieldName",
            RenameSupport.UPDATE_REFERENCES |
            RenameSupport.UPDATE_GETTER_METHOD |
            RenameSupport.UPDATE_SETTER_METHOD);
fieldRename.openDialog(shell);
```

5.4.4 Specialized extended rename dispatcher

With the `RenameSupport` class, we found an acceptable way to launch a refactoring of a Java element used in the Groovy domain. To employ this class in the whole refactoring process, the dispatching between Groovy and Java refactorings needed to be clearly separated. The `RenameSupport` class is not related with refactorings directly, its purpose is only to encapsulate them. This required the dispatch to be divided into two individual steps:

1. Check if the selected element is actually defined in Java. If so, launch a Java refactoring wizard on that element.
2. If the selected element can not be identified as Java origin, assume it's a Groovy element and run a Groovy refactoring wizard on it.

The implementation of these steps looks like this:

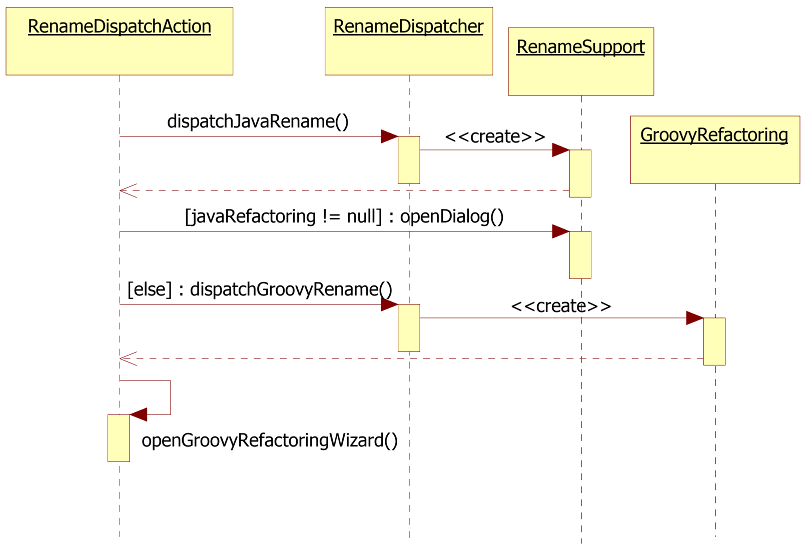


Figure 5.2: Second idea for the enhanced rename dispatcher

5.4.5 Why both of these solutions didn't work

Both the general as well as the specialized rename dispatcher solution had one big problem in common: They didn't work with dynamically typed variables. As described in the introduction to this section, when the refactoring was launched on a method invocation or a field access, we checked if the declaring class was defined in Java. Of course, no declaring class could be found when the corresponding variable was dynamically typed. To get back to the introductory example from section 5.1, the method call `presidentsDog.yell()` could never be used to rename the method that was eventually defined in Java. This was completely inacceptable, because working with dynamically typed variables and methods in Groovy is the standard and not the exception.

5.5 Pre-refactoring candidate collection

5.5.1 Description

During the development of our prototypes we realized that the dynamic nature of Groovy requires a whole different approach to support crosslanguage refactorings. Even if it's possible to detect objects defined in Java at runtime, the same awareness can't be guaranteed when working with sources and abstract syntax trees. Maybe one could argue that we should have known this before, but we think that getting into a dead end within a project is not that bad as long as you learn something from your faults.

Therefore, we had to look for a different approach to support remote refactorings of Java from Groovy code. Because there is no simple way to detect Java elements when used in a dynamic context, it was relatively clear that we would have to look up all potential matches of the element to be refactored. More precisely speaking, when a method `foo()` called on a dynamic variable should be renamed, we'll need to search for all Java classes defining such a method. Only Java classes? No, because the method could also be defined Groovy. We just don't know.

The *Groovy Refactoring Plug-in* currently handles this situation with the already known ambiguous candidate selection dialog as described in chapter 4.6. The refactoring operator is then supposed to select the declaration and all occurences he wants to rename in this dialog. It's actually even possible to refactor multiple declarations of methods when they all appear as candidates in the list. Unfortunatley it's not possible to integrate Java candidates in this dialog, becuase we need to launch a completly independent Java refactoring.

Our idea to solve this problem was collect all possible refactoring candidates in Groovy and Java prior to execute the actual refactoring. If we have multiple candidates, the user should select which element he wants to rename. The following flowchart visualizes the required steps:

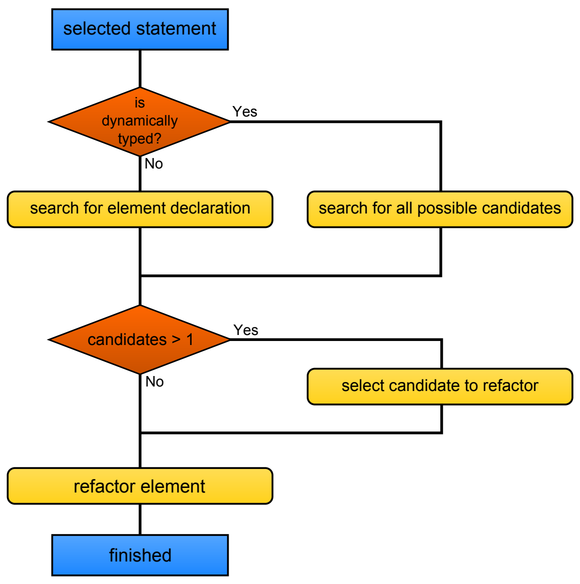

selected statement
is dynamically typed?
Yes
No
search for element declaration
search for all possible candidates
candidates > 1
Yes
No
select candidate to refactor
refactor element
finished

Figure 5.3: Candidate collection from Groovy and Java

For the behaviour of the pre-refactoring candidate , the following requirements should be fulfilled:

- All possible Groovy and Java elements are considered.
- The search for candidates leads only to valid elements, that *can* be refactored.
- Multiple candidates are expected to be seen only, when a refactoring is launched on a statement within a dynamically typed context.
- Refactorings called on statements within a statically typed context or on declarations should directly refactor the element declaration.
- In a situation where multiple candidates would be possible, but only one direct declaration is found, the refactoring should directly refactor this declaration.
- If multiple candidates are found, the the refactoring operator can choose the declaration that should be refactored.

To sum this all up, the behaviour of the current refactorings should not be changed in a major way. We basically just map all unknown refactoring sources to a declaration. If a refactoring is launched and there's only one element that makes sense to be refcatored, the refactoring should be executed directly, no matter whether the element is defined in Java or Groovy. Hence, if there are multiple possiblities, the Refactoring operator is asked to specify, which declaration he wants to refactor.

This solution gives us the chance to support all possible remote Java refactorings from Groovy. There is only some additional user input that has to be gathered. This fact is acceptable in consideration of the type-information we loose when working with dynamic languages.

5.5.2 Implementation

The implementation of the pre-refactoring candidate collection required a rather large structural change in the refactoring dispatch action. Previously, only the Groovy dispatch was engaged and then the refactoring wizard was launched. If refactoring candidates from two different languages are to be collected and individually dispatched, this requires some additional logic.

The refactoring dispatch process is now spread into different classes and looks as follows:

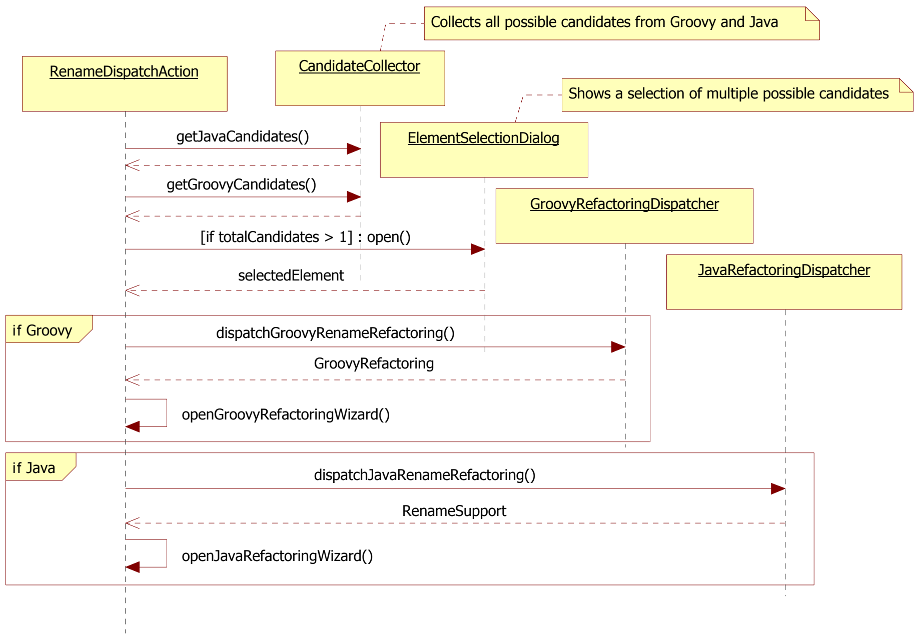

Collects all possible candidates from Groovy and Java
RenameDispatchAction
CandidateCollector
Shows a selection of multiple possible candidates
ElementSelectionDialog
GroovyRefactoringDispatcher
JavaRefactoringDispatcher
getJavaCandidates()
getGroovyCandidates()
[if totalCandidates > 1] : open()
selectedElement
if Groovy
dispatchGroovyRenameRefactoring()
GroovyRefactoring
openGroovyRefactoringWizard()
if Java
dispatchJavaRenameRefactoring()
RenameSupport
openJavaRefactoringWizard()

Figure 5.4: Collecting all refactoring candidates and open individual refactoring

All classes except the `RenameDispatchAction` are new and have individual responsibilies. Here's a description of the involved classes:

1. The **CandidateCollector** extracts the selected `ASTNode` from a Groovy file and assembles individual lists of refactoring candidates from Java or Groovy.
2. The **ElementSelectionDialog** is launched when multiple candidates were found and shows them separated by language. The refactoring operator then selects one candidate that is returned to the `RenameDispatchAction`.
3. The **RenameDispatchers** are now individual for each language and return either a `GroovyRefactoring` or a `RenameSupport` in case of Java refactorings. Thus Java refactorings are still initialized the same way as in the prototype.

After we have recieved a definitive candidate to refactor, the appropriate refactoring wizard is launched and the job of the dispatcher is finished.

5.5.3 Search patterns

The only elments that actually can have multiple candidates are methods and fields. The search of those was realized with AST visitors in terms of Groovy and with a finegrained search in the Java model. In both cases the search pattern for those elements depends on whether dynamic typing is involved or not:

- **without dynamic typing** the pattern does contain the fully qualified class in which the element to be refactored was declared
- **with dynamic typing** the pattern looks only for elements with the same name

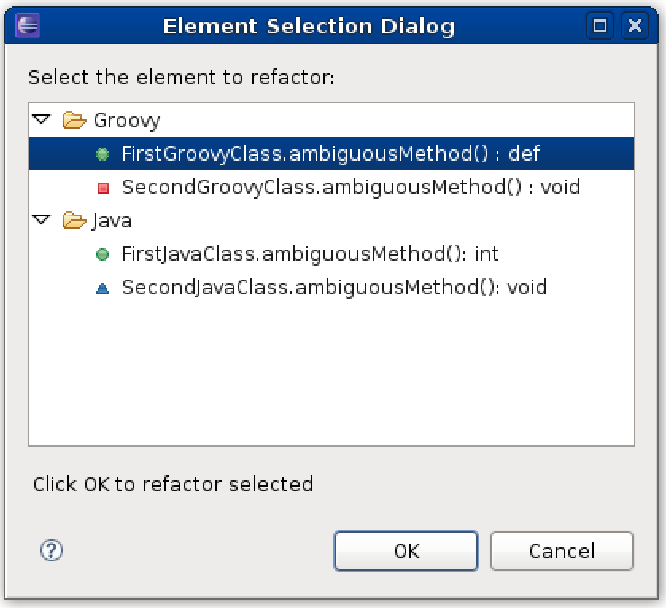
In case of methods, the pattern further compares the number of parameters. A complete comparison of method signatures is not reasonable, because in Groovy, parameters can also be dynamically typed or even rearanged with named parameters.

5.6 Element selection dialog

In cases where multiple candidates are considered to be refactored, we need to present those candidates to the refactoring operator. For this purpose we had to create a simple element selection dialog that shows all candidates separated by each language.

Our first intention was to self implement an SWT based control. A basic dialog with a simple list to select the candidates was fastly realized. However, the hassle with GUI development lies often in the details so it was with this self built frame. Make it look like a usual Eclipse dialog and resizable was already annoying enough. Further there was no clean distinction between GUI and data, so we looked for a simpler solution.

Eclipse already has numerous predefined dialogs for selection[3], so we decided to use one of these. Specifically, we extended the `ElementTreeSelectionDialog`, an expandable tree based selection. We had make sure the tree is always fully expanded when the window pops up. Further we had to provide a content provider and a label provider. The content provider holds the tree structure of all displayed elements and the label provider serves a textual description and a image for each element. These classes are all relatively simply structured and thus not further explained here. Here's an example of the resulting dialog:


Element Selection Dialog

Select the element to refactor:

Groovy
FirstGroovyClass.ambiguousMethod() : def
SecondGroovyClass.ambiguousMethod() : void
Java
FirstJavaClass.ambiguousMethod(): int
SecondJavaClass.ambiguousMethod(): void

Click OK to refactor selected

Figure 5.5: Selection window for ambiguous refactoring candidates

This dialog is initialized with an instance of the `RenameCandidates` class, a generic collection of candidates from Groovy and Java. After the selection is finished, the selected element can be gathered form the dialog and the refactoring process is started.

[3]Examples at http://blog.eclipse-tips.com/2008/07/selection-dialogs-in-eclipse.html

5.7 Implementation Review

A complete overview of our implementation for remote Java refactorings can be seen in the following class diagramm:

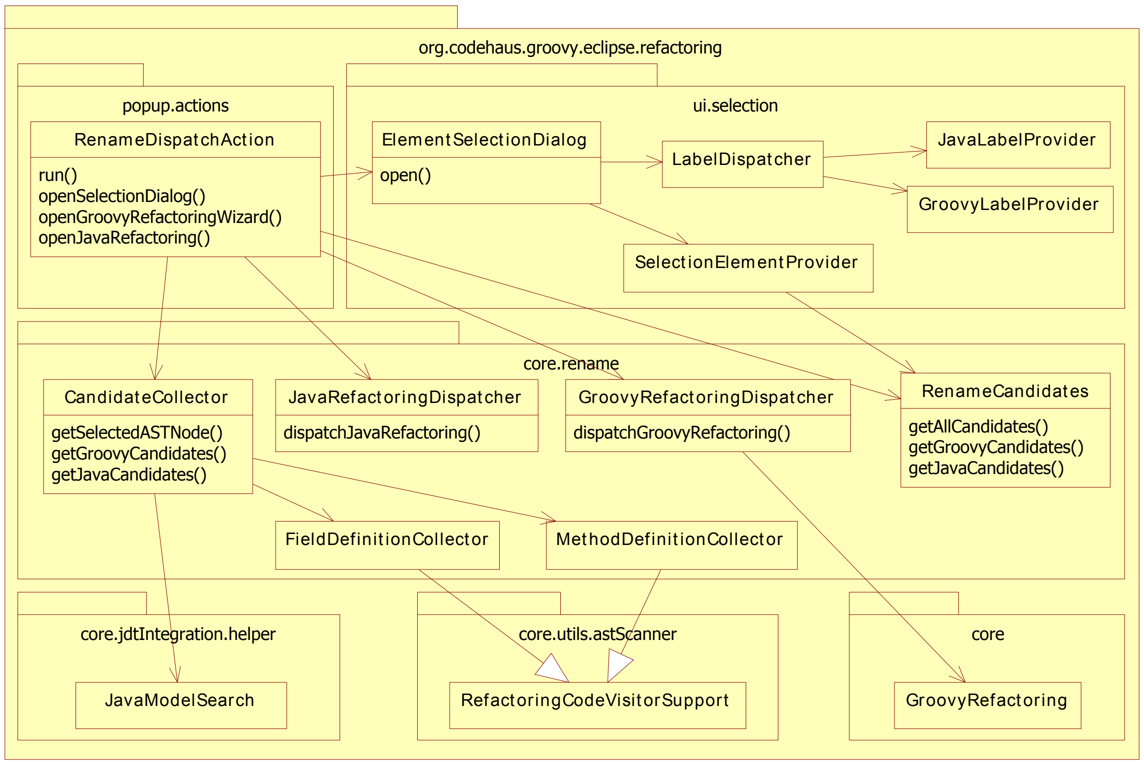
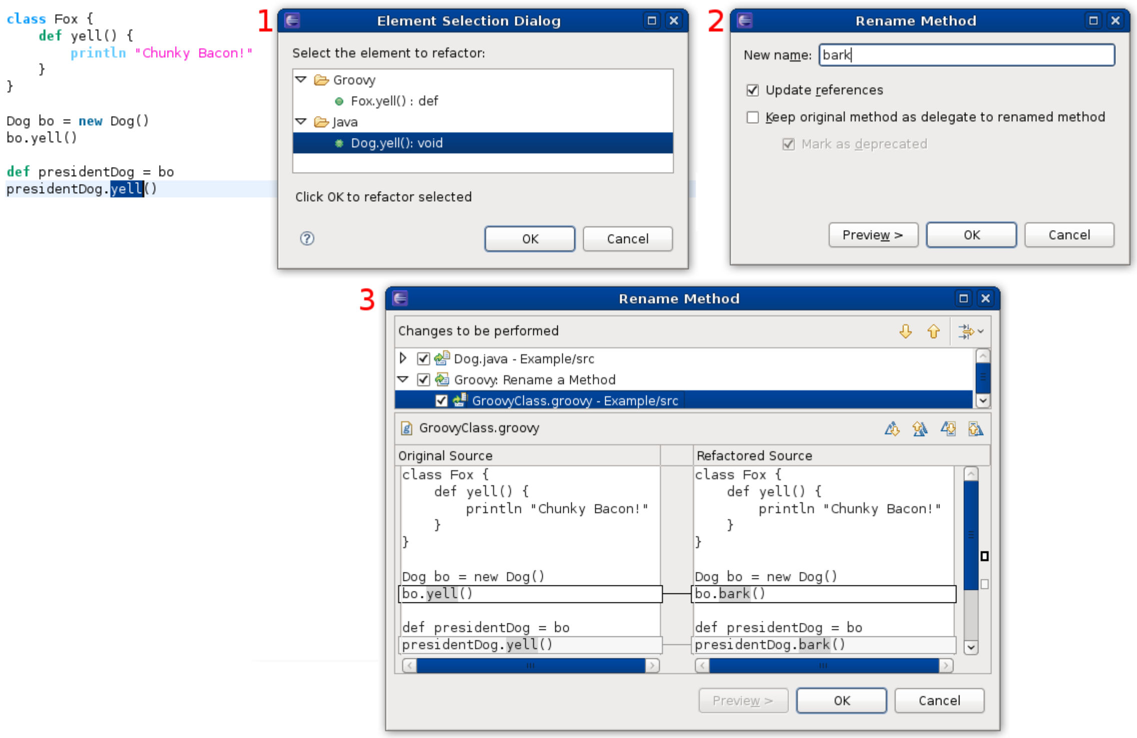


Figure 5.6: Overview of involved classes in remote java refactorings

5.8 In Action

The following screenshot show a remote Java refactoring in action:



```
class Fox {
    def yell() {
        println "Chunky Bacon!"
    }
}

Dog bo = new Dog()
bo.yell()

def presidentDog = bo
presidentDog.yell()
```

Original Source:

```
class Fox {
    def yell() {
        println "Chunky Bacon!"
    }
}

Dog bo = new Dog()
bo.yell()

def presidentDog = bo
presidentDog.yell()
```

Refactored Source:

```
class Fox {
    def yell() {
        println "Chunky Bacon!"
    }
}

Dog bo = new Dog()
bo.bark()

def presidentDog = bo
presidentDog.bark()
```

Figure 5.7: Example of a local java refactoring

We have the same scenario as descriped in chapter 5.1:Introductory example. The class `Dog` is defined in Java and the method `yell()` is called once on the statically typed variable and once on a dynamically typed one. It invokes a rename refactoring on the first call, window number 2 is opened directly. Otherwise with the dynamically typed variable, where we first have to deal with a coexisiting method defined in Groovy. We select the Java class and the refactoring is launched as if it directly was startet on the declaration. But note that in this case, because we launched the refactoring from the only ambiguous call to `bark()` in Groovy, there's no other ambiguous selection dialog popping up (the selected statement is always considered as definitive candidate). If we would have called the refactoring on the declaration in Java, we would have been forced to select ambigous candidates later.

5.9 Further Ideas

There's one thing that is not possible with our extension of the *Groovy Refactoring Plug-in* any more: Renaming multiple declarations. This may not seem as a big drawback on the first sight, but in terms of Groovy it actually would be useful. Because of duck typing, we may have situations where something like inheritance is used without a specifcally declared relation between classes (See chapter 2.1 on page 6). We can't refactor such related methods no longer in one step, because we force the refactoring operator to select one declaration. To support this feature, the declarations in the candidate selection dialog should be checkable or something like that. The refactoring should then first gather the new name and then rename all specified declarations. However, in Groovy this is possible to do within one single refactoring step. In terms of Java, this means to launch multiple refactorings and batch process each one. This could result in overlapping refactoring changes and is therefore dangerous in some situations. For this reason, we didn't implement multiple refactorings. Still, they would be quite usefull in some situations.

6 Local Groovy Refactoring

6.1 Introductory example

Groovy is a mighty tool to simplify the life of a Java developer. It fits simple daily tasks such as parsing XML as well as big projects. The tight integration with the JVM allows to use Groovy nearly everywhere where Java is suspected to be the only adequate option. This includes a huge number of existing Java frameworks and libraries such as Servlets, Spring, Hibernate and lots of others. For this reason, refactorings in the Groovy domain should also be reflected in connected Java code as well.

This is exactly what this chapter is about. For example let's take a simple Groovy bean that is used within Java code. If we now refactor a field of this bean, wouldn't it be pleasant if the generated getter and setter methods in Java are updated as well?

Java

```
public class PersonUsage {
  public static void main(String[] args) {
    Person stefan = new Person();
    stefan.setName("Stefan");
    stefan.setTotalYears(23);
    System.out.println(stefan);

    Person dude = new Person();
    dude.setTotalYears(42);
    dude.setName("His Dudeness");
    System.out.println(dude);
  }
}
```

Groovy

```
class Person {
  private static int lastId = 0

  def final id = lastId++
  def name
  def totalYears

  String toString() {
    "${id}: ${name} (${age})"
  }
}
```

Java (refactored)

```
public class PersonUsage {
  public static void main(String[] args) {
    Person stefan = new Person();
    stefan.setName("Stefan");
    stefan.setAge(23);
    System.out.println(stefan);

    Person dude = new Person();
    dude.setAge(42);
    dude.setName("His Dudeness");
    System.out.println(dude);
  }
}
```

Groovy (refactored)

```
class Person {
  private static int lastId = 0

  def final id = lastId++
  def name
  def age

  String toString() {
    "${id}: ${name} (${age})"
  }
}
```

6.2 Use Case

The user works on a mixed project and renames a Groovy element from the declaration, or a reference. If the renamed element is referenced from some Java code, this code has to be updated too. In this use case, the user starts a rename refactoring from the Groovy editor.

Preconditions

The Project has to be in a clean state. That means, all sources need to be compilable without build errors.

Postconditions

The element and all references are renamed and the project is in a clean state again, without build errors.

Basic course of events

1. The User starts the rename refactoring from a Groovy editor.
2. The refactoring seaches all possible candidates from the selected reference and shows a list to the user.
3. If the user choose a Groovy element out of the list a usual Groovy refactoring on this element is started.
4. After all the Groovy elements are renamed, a rename participant starts the corresponding JDT refactoring.
5. The JDT refactoring updates all references in Java.

Alternative paths

- **If the user chooses a Java element** out of the list with possible candidates, the scenario in chapter 5 (Remote Java Refactoring) is started.
- **If a reference in Groovy is dynamically typed**, the refactoring can't automatically identify them. In this case, the user can select all the candidates he wants to rename. It's the users own responsibility, to take care about which candidates he wants to rename.

6.3 Updating binary references in Java

To support local Groovy refactorings, we need a clean way to update Java source code referencing to Groovy elements. On the first occasion this sounds pretty simple to realize: There are numerous Java refactorings available in the JDT and all we need to do, is to invoke those. Unfortunately it's not that easy. All JDT refactorings are exclusively designed for source files in a Java workspace. The restriction to source files makes sense, because there's almost nobody expecting a binary class or something in a jar file to be refactorable. Binary elements are just seen as given and if you need to change something for your project, you hope to get the sourcecode somewhere.

Now comes the problem: Groovy and Java in Eclipse are only connected trough binary references. If a Groovy class is written, the compiler generates the bytecode and Java sees that a "new" binary class file has been added to the project. Now if we try to refactor something used from a Groovy class from Java, the JDT thinks we want to refactor a binary element and immediately stops the refactoring. This also applies for programmatically started refactorings as described in chapter 8.5 on page 75. As far a we could find out, this affects all commonly known refactorings in the JDT and cannot be bypassed or configured somehow. Also the mailing list couldn't help us any further.

Because of the mentioned problems, we had to look for a different way to refactor binary references in Java. We had three main ideas:

- **Copy the JDT refactorings:** As the JDT source is available, we could import the complete refactorings into our project and modify them, until binary elements can be renamed without problems.
- **Error-marker search:** After a Groovy refactoring has been performed, we could look for all new error-markers within Java code and correct these if possible. This can only be done post-refactoring and thus allows no preview.
- **AST Modification:** We could also implement our own AST-based refactorings and use those for updating Java code. This may required a complex implementation.

We decided for the third option because the other two didn't seam to be well-fitting solutions to us: Copying a huge bunch of code to change one simple line is never a good idea. Further we don't even need the complete functionality of the JDT refactorings. The second solution is like playing pyromaniac firefighters: repair something after you intentionally broke it. An maybe there's even more broken afterwards. So the third option was the only reasonable solution.

6.4 Groovy Refactoring Participants

Before we could start to implement the additional refactorings we had to solve an other issue. The existing Groovy refactorings were all realized as standalone refactorings and offered no support for refactoring participants.

In the language toolkit of Eclipse, there are two similar classes to support refactoring implementations. These are the pure `Refactoring` itself without participation support and the extended version named `ProcessorBasedRefactoring`. The Groovy refactorings are all based on the first one and thus did not offer native support for refactoring participants. So the logical solution seemed to be a migration of the Groovy refactorings to processor based refactorings. This was also our first intention, but the internal structure of `ProcessorBasedRefactoring` is much more complex compared to simple refactorings. The Groovy refactorings themselfs also rely on a largely enhanced refactoring architecture. As a result, the effort to migrate the Groovy Refactoring Plug-in to `ProcessorBasedRefactorings` would require huge architectural changes. The biggest benefit of this migration would be that Groovy refactorings then could offer extension points to register refactoring participants. However, we had to abandon this solution because there were already some delays in our project schedule and it was not a requirement for us to implement Groovy participants over extension points.

Instead, we extended the existing Groovy refactorings with a simplified support for participants. Our so called `ProcessorBasedRefactorings` light:

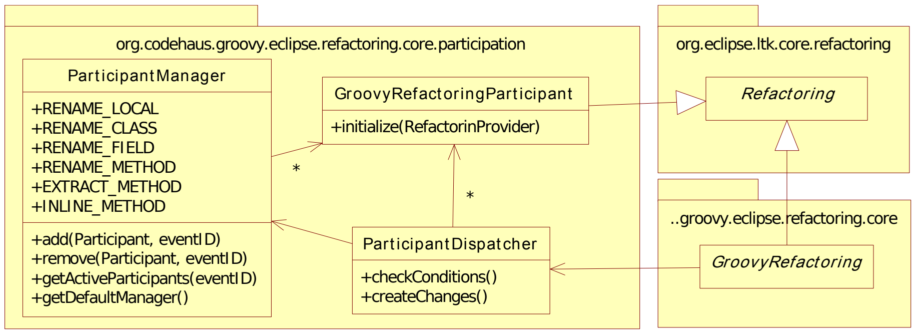

org.codehaus.groovy.eclipse.refactoring.core.participation
ParticipantManager
+RENAME_LOCAL
+RENAME_CLASS
+RENAME_FIELD
+RENAME_METHOD
+EXTRACT_METHOD
+INLINE_METHOD
+add(Participant, eventID)
+remove(Participant, eventID)
+getActiveParticipants(eventID)
+getDefaultManager()
GroovyRefactoringParticipant
+initialize(RefactorinProvider)
*
*
ParticipantDispatcher
+checkConditions()
+createChanges()
org.eclipse.ltk.core.refactoring
Refactoring
..groovy.eclipse.refactoring.core
GroovyRefactoring

Figure 6.1: Participation support for Groovy refactorings

The implementation consists of three simple classes:

- A **GroovyRefactoringParticipant** is a usual refactoring extended by an initialize method to declare if the participant is willing to contribute to the current refactoring.
- The **ParticipantManager** is a registration point for refactoring participants. There is a binary flag for each implemented refactoring. Using this, a participant can be registered to one or multiple refactoring events.
- The **ParticipantDispatcher** is associated with a participant manager and safely delegates condition-checking and change creation for each registered participant.

6.5 Java Update Refactorings

6.5.1 AST modification procedure

With the help of the implemented participant support, it's possible to add additional change tasks to Groovy refactorings. So the next step was to implement the refactorings to update renamed Groovy elements in Java sources. The term refactoring is maybe a little bit overstressed at this point, because we only need to lookup referencing elements and rename all found occurrences. We still use the name refactoring because it's implemented as one.

The Eclipse JDT Plug-in offers a very powerful abstract syntax tree facility with support to parse, traverse, edit and rewrite Java source files. A very recommendable introduction on handling JDT ASTs can be found in [AST06]. The rest of this chapters uses terms and definitions explained in this article.

So, what's the job of Java update refactoring when a Groovy refactoring is performed? Roughly speaking, it has to:

1. Find all java files containing references to the renamed Groovy element.
2. Create an AST for all those files.
3. Traverse the AST and collect all occurrences of the renamed subject.
4. Replace each occurrence with the new name as specified in the parent refactoring.
5. After all occurrences are replaced, collect the required edits and return those changes to the parent Groovy refactoring.

Surprisingly, this enumeration nearly exactly describes how we implemented the Java update refactorings. To be specific, here's a more detailed but still abstract description of how each task from the list above was realized:

1. Finding all referencing Java files is accomplished by searching the Java model for a specific pattern. When a search pattern is configured to search only for references, the returned values are always types or methods containing a reference to the searched element. From these elements, we can get the files of definition.
2. We generate the AST including all bindings using the `ASTParser`.
3. The AST gets traversed by an `ASTVisitor` and resolves each eligible binding depending on the element that is going to be refactored . The binding is then compared with the element. If they match, the AST node is identified as a reference and added to the collection of all found occurrences.
4. For all found occurrences, the class `ASTRewrite` generates the necessary textual edits.
5. These edits then get bundled into a `Change` object that will be a part of the refactoring to be applied.

This is still a high level description of the necessary steps to refactor Java code from AST modifications. But the procedure remains the same for all implemented Java update refactorings.

6.5.2 Implementation

As described in the previous section, there are a lot of shared tasks among all Java update refactorings. In respect of this fact, most of the functionality is realized in a universally used abstract refactoring. However, there are ultimately two context sensitive tasks that differ for each refactoring:

- The **search pattern** to find references in the Java model.
- The **ASTVisitor** to identify references and collect them.

It's the responsibility of each refactoring implementation to create these two elements and pass them back via template methods `getSearchPattern()` and `getCollector()`. The Java search pattern only needs to be correctly adjusted so that all references to the element to be refactored are found. The collector on the other hand, is an AST visitor that must be able to retain all found references. For this reason the class `SimpleNameCollector` was introduced. A `SimpleName` is the smallest possible element to be extracted from a statement in the Java AST. This is also the canonical element that finally represents the text we need to replace.

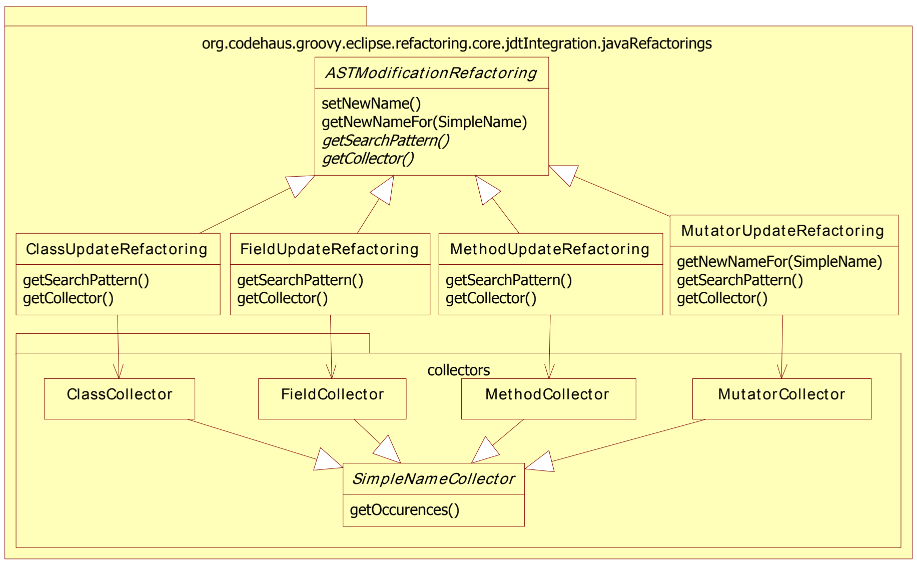


Figure 6.2: Simplyfied class diagram of Java update refactoring classes

Each `SimpleName` returned by the collector will then be replaced during the refactoring process.

There is also one special case: Groovy fields that generate synthetical getter/setter methods, also known as mutators. When such a field is renamed, we have to refactor the mutator methods in Java instead of field access statements. This case is covered by the `MutatorUpdateRefactoring`.

6.6 Implementation Review

The implemented participant architecture in combination with the Java update refactorings offers a fully fledged solution to support local Groovy refactorings. Anyhow, we're still missing the link between those modules. To interconnect both elements, the last thing to do was to implement the Groovy refactoring participants and call the appropriate Java refactorings. The following sequence diagram shows the new workflow of a Groovy refactoring with the added participant:

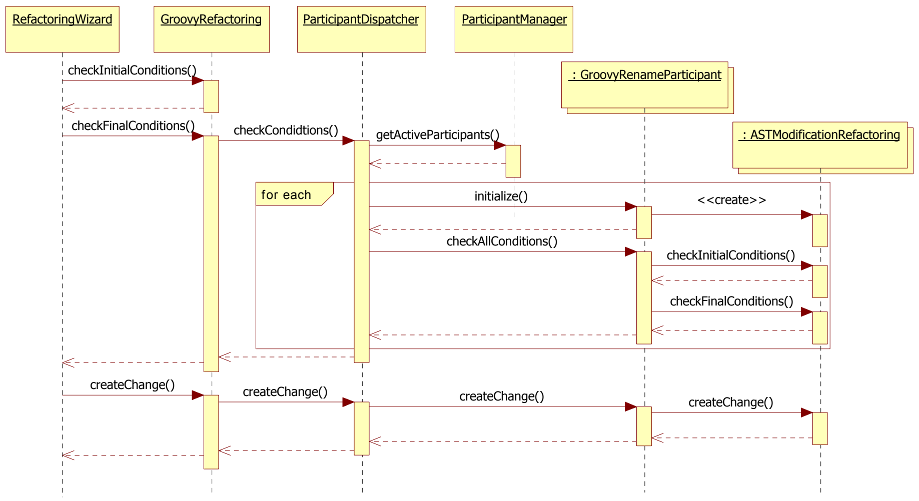


Figure 6.3: New workflow of Groovy refactorings including participants

To register additional participants, they need to be loaded when the plug-in is activated. To register a participant for a specific refactoring event, it has to be added to the default `ParticipantManager`:

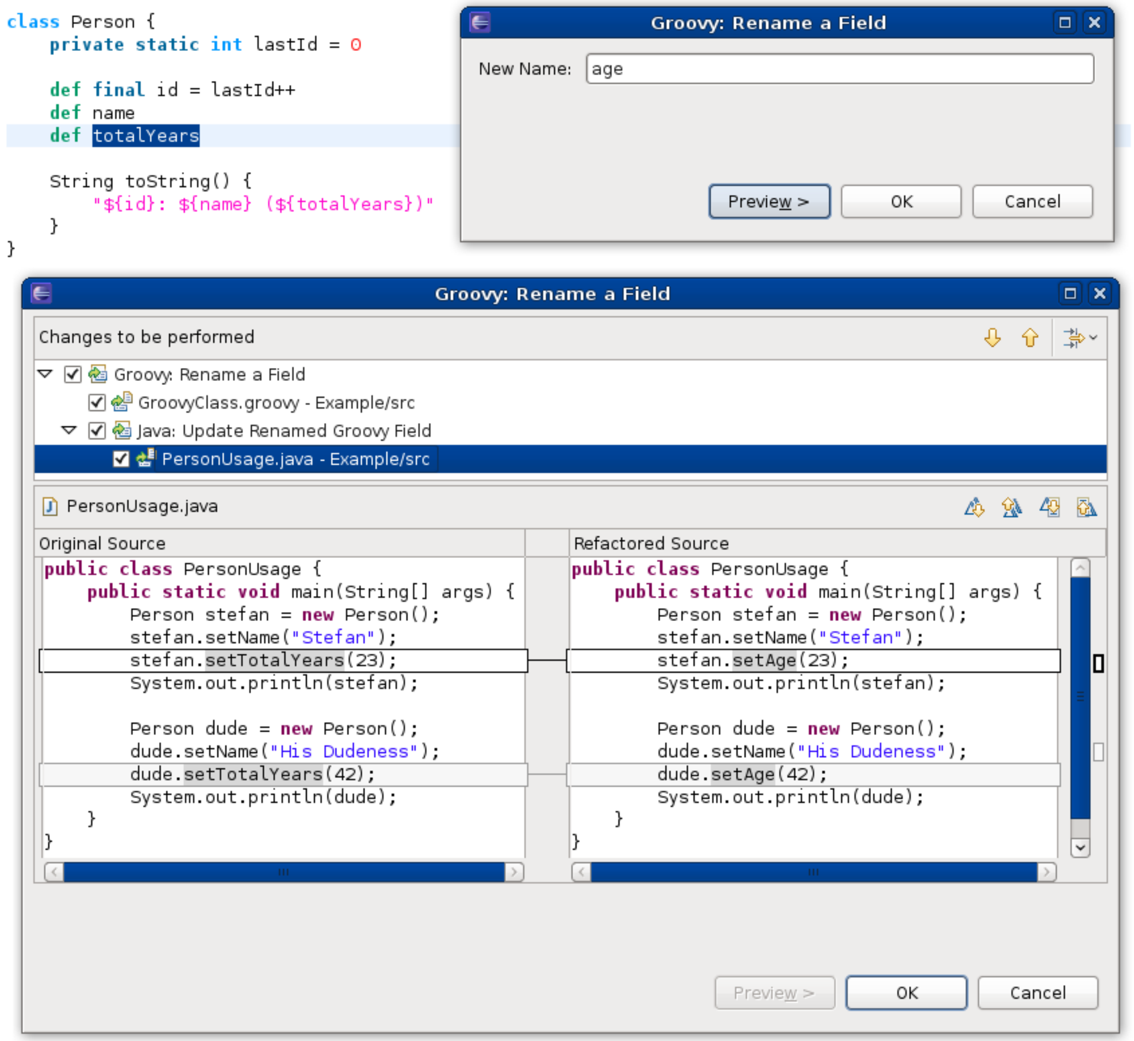
Listing 6.1: Participant registration

```
ParticipantManager pm = ParticipantManager.getDefaultManager();
pm.add(ClassRenameParticipant.class, ParticipantManager.RENAME_CLASS);
pm.add(FieldRenameParticipant.class, ParticipantManager.RENAME_FIELD);
pm.add(MethodRenameParticipant.class, ParticipantManager.RENAME_METHOD);
```

Over all, the design of this solution is pretty similar to the refactoring participants as seen in chapter 4: Local Java Refactoring. This was our intention and helps to keep the code maintainable. Further it's now possible to extend the Groovy refactorings with other additional refactoring participants.

6.7 In Action

The following screenshots shows the *Crosslanguage Groovy Refactoring* in action. The code base is the same as in chapter 6.1: Introductory example.



```
class Person {
    private static int lastId = 0

    def final id = lastId++
    def name
    def totalYears

    String toString() {
        "${id}: ${name} (${totalYears})"
    }
}
```

Groovy: Rename a Field

New Name: age

Groovy: Rename a Field

Changes to be performed

- Groovy: Rename a Field
  - GroovyClass.groovy - Example/src
  - Java: Update Renamed Groovy Field
    - PersonUsage.java - Example/src

PersonUsage.java

Original Source

```
public class PersonUsage {
    public static void main(String[] args) {
        Person stefan = new Person();
        stefan.setName("Stefan");
        stefan.setTotalYears(23);
        System.out.println(stefan);

        Person dude = new Person();
        dude.setName("His Dudeness");
        dude.setTotalYears(42);
        System.out.println(dude);
    }
}
```

Refactored Source

```
public class PersonUsage {
    public static void main(String[] args) {
        Person stefan = new Person();
        stefan.setName("Stefan");
        stefan.setAge(23);
        System.out.println(stefan);

        Person dude = new Person();
        dude.setName("His Dudeness");
        dude.setAge(42);
        System.out.println(dude);
    }
}
```

Figure 6.4: A Local Groovy Refactoring in action

6.8 Restrictions

The basic rename scenarios when Groovy code is used from Java are covered with this refactoring. However, it has to be mentioned that Groovy has a lot more power than conventional Java. This may lead to situations where the resulting class files are no longer conveniently usable from Java. Especially everything using meta programming stuff tricks . For example builders or expanded classes using `ExpandoMetaClass` are not accessible without reasonable efforts. This is not very surprisingly as Java was never ment to do so. Notice that in these situations even the Groovy refactorings mostly fail because there's simply now way to detect what should be refactored.

6.9 Further Ideas

- **Class name duplicate checks:** Currently there's no warning message when a Java class is renamed to an already used name by a Groovy class in the same package. This would lead to unpredictable situations and thus should be avoided.
- **Derived method warning:** Within the JDT, there's a warning message shown when a derived method from a superclass is about to be renamed. If the refactoring operator still decides to rename it, he accepts that this may result in semantic changes of the code. In the *Groovy Refactoring Plug-in*, there's currently no such message.

7 Remote Groovy Refactoring

7.1 Introductory example

As written in the last chapter, Groovy is even useful for Java developers. With the tight integration into the JVM, a Java developer can souce some parts of his software out into Groovy, to simplify them. So refactorings in the Groovy domain should also be able to start from references in Java.

In the following example, we have two bikes in a Java class and want to rename the field ”*r*” into the more obvious name “*rearWheel*”. Started from the property `harley.getR`, the definition of the field in Groovy and all references get renamed. The second bike, the Kawasaki isn't realy interessted in this operation, because it never uses this field or its getter, so nothing gets renamed here.

```
public class Java {
   public static void main(String[] args) {
      Bike harley = new Bike();
      harley.getR().spin();
      harley.getF().spin();

      Bike kawasaki = new Bike();
      kawasaki.spinAll();
   }
}
```

```
class Bike {
   Wheel r = new Wheel()
   Wheel f = new Wheel()
   def wheels = [r,f]

   void spinAll(){
      wheels.each{ it.spin() }
   }
}

class Wheel{
   def spin(){
      println "Spinning wheel"
   }
}
```

After the refactoring, the declaration of the used element `r` is renamed, so does all references.

```java
public class Java {
   public static void main(String[] args) {
      Bike harley = new Bike();
      harley.getRearWheel().spin();
      harley.getF().spin();

      Bike kawasaki = new Bike();
      kawasaki.spinAll();
   }
}
```

```groovy
class Bike {
   Wheel rearWheel = new Wheel()
   Wheel f = new Wheel()
   def wheels = [rearWheel,f]

   void spinAll(){
      wheels.each{ it.spin() }
   }
}

class Wheel{
   def spin(){
      println "Spinning wheel"
   }
}
```

7.2 Use Case

The user renames a Groovy element starting from a reference in a Java file. The declaration in Groovy has to be renamed, as well as all references in Groovy and Java.

Preconditions

The Project has to be in a clean state. That means, all sources need to be compilable without build errors.

Postconditions

The element and all references are renamed and the project is in a clean state again, without build errors.

Basic course of events

1. Start a Groovy refactoring from the special menu entry in the JDT editor.
2. The refactoring searches all possible candidates from the selected reference and shows a list to the user.
3. If the user choose a Groovy element out of the list a usual Groovy refactoring on this element is started.
4. After all the Groovy elements are renamed, a rename participant starts the corresponding JDT refactoring.
5. The JDT refactoring updates all references in Java.

Alternative paths

- **If the user chooses a Java element** out of the list with possible candidates, the scenario in chapter 4 (Local Java Refactoring) is started.
- **If a reference in Groovy is dynamically typed**, the refactoring can't automatically identify them. In this case, the user can select all the candidates he wants to rename. It's the users own responsibility, to take care of which candidates he want to rename.

7.3 New Menu entry

The biggest challenge in this use case is the fact, a JDT refactoring does not allow to rename binary references. Even the menu entry in the context menu is not visible. Our predecessors [KKK08] also discussed this problem with Dr. Dirk Bäumer [1], and had just one solution. A new menu entry. Due to the fact, the menu entry is invisible, if a binary reference is selected, this new menu entry replaces the usual JDT entry. On the users end, nothing changes if the selected element is defined in Groovy or in Java.

In the Eclipse framework, menu entry's are based on actions and commands, which both are defined through various extension points. So that new functionality can be easily added. For more information about commands and actions please read chapter 6 of the book *eclipse Plug-ins* [EPi08]. We defined our actions in the same extension point, as the Groovy refactoring [KKK08], so when the refactoring plug-in is loaded, all the menu entries will load at the same time.

For menu entries, the Eclipse framework offers two different APIs. The common action API and, since Eclipse 3.3, the command API. Inspired by the other menu entries attaching to this extension point, we used the action API, for our elements. We had to add two new actions: the refactoring menu in the menu bar, and the context menu.

7.3.1 Refactor menu entry

In the refactoring menu, in the menu bar of the JDT editor, all actions are visible all the time, no matter what kind of element is selected. So we had to add a completely new menu entry, to the already existing menu, which is shown all the time too.

To add this menu entry in the refactoring menu, we used a new `editorContribution`. For the definition, in which part of the menu the action should be, we had to define the menu part again, with the same identifier as it is already defined in the JDT menu. This is the only way to add a new menu entry to an already defined menu. The action is linked to the `RenameInJavaAction` action listener, which starts the groovy refactoring .

Listing 7.1: Extension point configuration for the menu entry

```
<editorContribution
    id="org.codehaus.groovy.eclipse.refactoring.editorContributionJava"
    targetID="org.eclipse.jdt.ui.CompilationUnitEditor">
   <menu
      label="Refac&amp;tor"
      path="edit"
      id="org.eclipse.jdt.ui.refactoring.menu">
    <separator name="reorgGroup"/>
  </menu>
    <action
        class="org.codehaus.groovy.eclipse.refactoring.popup.actions.RenameInJavaAction"
        definitionId="org.codehaus.groovy.eclipse.refactoring.command.renameGroovy"
        id="org.codehaus.groovy.eclipse.refactoring.renameGroovy"
        label="Rename Groovy Element..."
        menubarPath="org.eclipse.jdt.ui.refactoring.menu/reorgGroup">
```

[1] At that time, Dirk Bäumer was a member of the Eclipse architecture team

```
    </action>
</editorContribution>
```

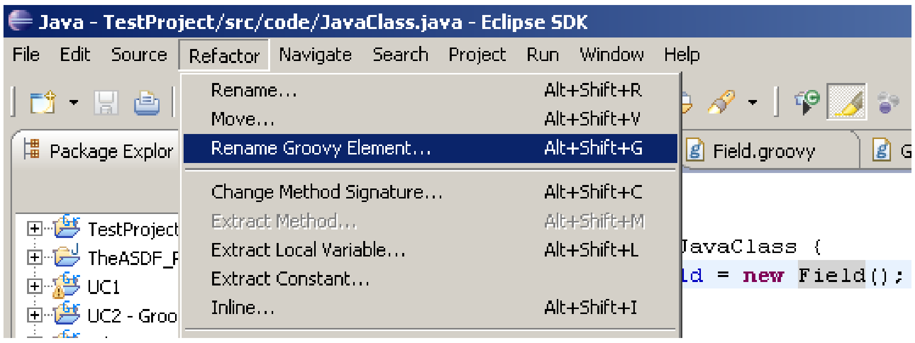

Java - TestProject/src/code/JavaClass.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Rename... Alt+Shift+R
Move... Alt+Shift+V
Rename Groovy Element... Alt+Shift+G
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Inline... Alt+Shift+I
Package Explor
TestProject
TheASDF_F
UC1
UC2 - Groo
Field.groovy

Figure 7.1: Screenshot of the refactoring menu entry

7.3.2 Context menu entry

If a context menu gets visible, it contains just those elements, that can actually interact with the selected element. If a reference to a Groovy element is selected, the JDT rename refactoring action isn't visible. So it should be possible, to replace this action. But the way it looks, it is impossible to add a new menu entry to the existing `Refactoring` submenu. We tried a few ways, but the new menu entry was never shown. So we decided to add a new menu entry just to the context menu itself. This isn't really a user-friendly way, but not even the Eclipse newsgroups contained clues how to solve this problem.

In the Eclipse framework, the context menu is called `popup menu`. To add a new entry, we added a new `viewerContributon` to the extension point. With the `targetID`, this contribution is shown in the Java editor. The action is quiet the same as for the 7.3.1 refactor menu entry, just with a different `menubarPath`.

Listing 7.2: Extension point configuration for the context menu entry

```
<viewerContribution
    id="org.codehaus.groovy.eclipse.refactoring.javaContribution"
    targetID="#CompilationUnitEditorContext">
  <action
      class="org.codehaus.groovy.eclipse.refactoring.popup.actions.RenameInJavaAction"
      definitionId="org.codehaus.groovy.eclipse.refactoring.command.renameGroovy"
      id="org.codehaus.groovy.eclipse.refactoring.renameJavaAction"
      label="Rename Groovy Element"
      menubarPath="org.eclipse.jdt.ui.refactoring.menu">
  </action>
</viewerContribution>
```

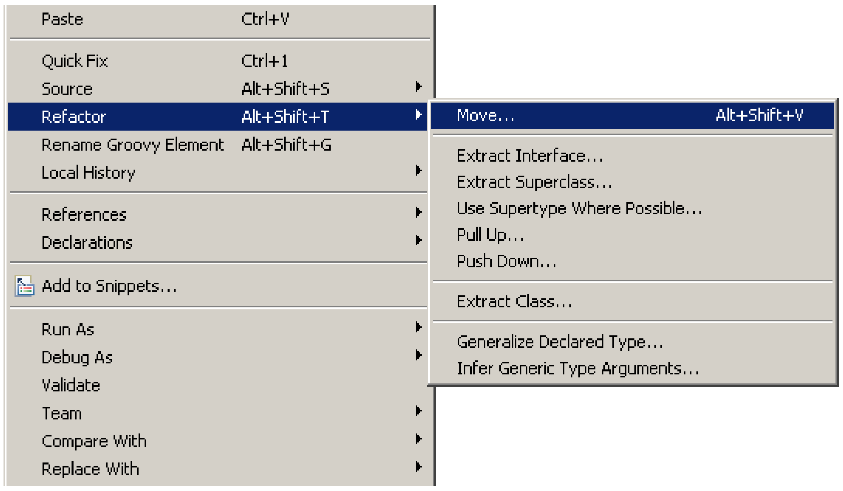


Figure 7.2: Context menu with selection on a Groovy element

The linked action class implements the `IEditorActionDelegate` interface, with a `selectionChanged` method, which is called each time, a selection changes. This method takes the selected element from the editor and checks, if it is a Groovy element. If this is the case, the menu entry is enabled, if not it will be disabled. Unfortunately, the Eclipse framework uses lazy loading for all plug-ins. So until the *Groovy Refactoring Plug-in* is called the first time, this new menu entry is enabled all the time. But if someone uses the menu item, Eclipse loads the plug-in and a message pops up which says, that the chosen operation isn't enabled. From then on, the plug-in is loaded and the menu item is enabled only if it needs to be.

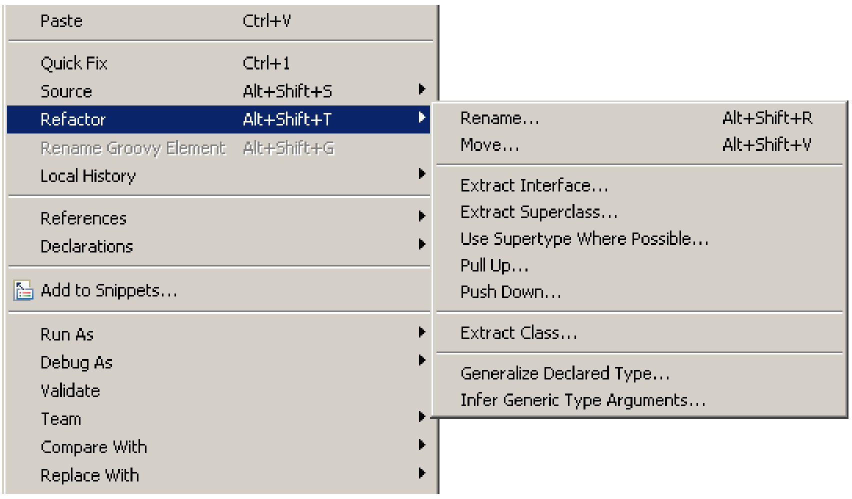
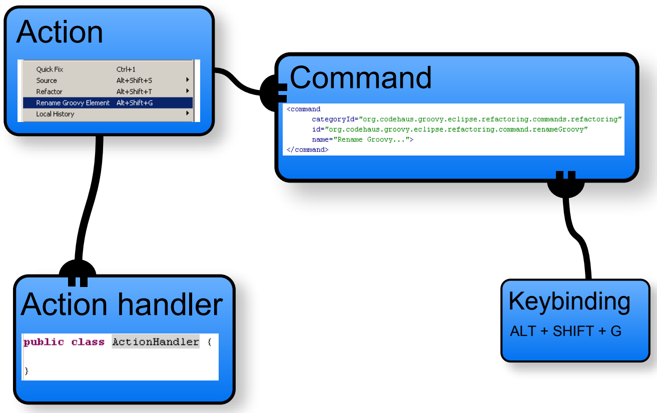


Figure 7.3: Context menu with selection on a Java element

7.3.3 Adding a key binding

In the Eclipse framework, a key binding belongs to a command. If you want to add a keybinding to one or more actions, you have to add a new command and link them from all the actions you want to.



```
<command
      categoryId="org.codehaus.groovy.eclipse.refactoring.commands.refactoring"
      id="org.codehaus.groovy.eclipse.refactoring.command.renameGroovy"
      name="Rename Groovy...">
</command>
```

```
public class ActionHandler {

}
```

Figure 7.4: Overview - from the Action to the Keybindings

A Key binding should consist at least one modifier key, and one other key. In Eclipse, the modifier keys have an abstract name, a platform-independent way to represent the keys. For example, the abstract name “M1” is for the COMMAND key on a MacOS X, or the CTRL key on most other platforms. The other keys are generally specified simply as the ASCII character, in uppercase.

For the Groovy refactoring out of a Java editor, we implemented the keybinding *ALT+SHIFT+G*, or with abstract names *M3+M2+G*.

Listing 7.3: Extension point configuration for the keybinding

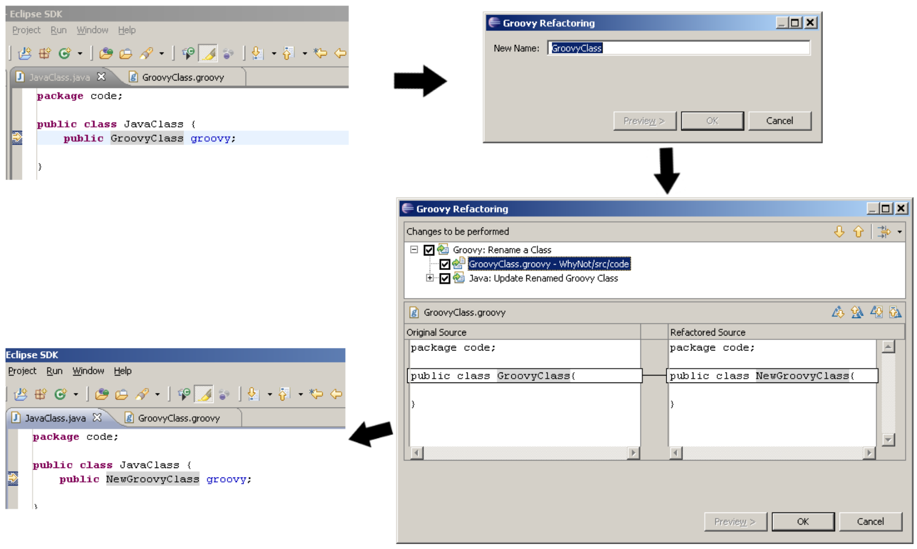
```
<extension point="org.eclipse.ui.commands">
  <command
      categoryId="org.codehaus.groovy.eclipse.refactoring.commands.refactoring"
      id="org.codehaus.groovy.eclipse.refactoring.command.renameGroovy"
      name="Rename Groovy...">
  </command>
</extension>

<extension point="org.eclipse.ui.bindings">
  <key
      commandId="org.codehaus.groovy.eclipse.refactoring.command.renameGroovy"
      contextId="org.eclipse.jdt.ui.javaEditorScope"
      schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"
      sequence="M3+M2+G">
  </key>
</extension>
```

7.4 Restrictions

The former language-isolated Refactorings, both have the same key binding *(CTRL, Alt, R)*. Against some answers in the Eclipse newsgroups, it is possible to overwrite the existing JDT-given key bindings. But it overwrites the key binding all the time. If the *Groovy refactoring plug-in* is loaded, a local Java refactoring (on a Java element) is not possible any more. So we decided to define a different one, for the *Crosslanguage Refactoring*. And because the key “`G`” is almost next to the “`R`” key, and implicate the Groovy language, we took *ALT+SHIFT+G*.

7.5 In Action

When the action is executed, the action listener gets notified, and starts the same refactoring as in scenario 6: `Local Groovy Refactoring`, at page 51 for the selected element.



```
package code;

public class JavaClass {
    public GroovyClass groovy;

}
```

New Name: GroovyClass

Groovy: Rename a Class
GroovyClass.groovy - WhyNot/src/code
Java: Update Renamed Groovy Class

```
package code;

public class GroovyClass{

}
```

```
package code;

public class NewGroovyClass{

}
```

```
package code;

public class JavaClass {
    public NewGroovyClass groovy;

}
```

Figure 7.5: Example of a remote Groovy refactoring

7.6 Further Ideas

As written above, the refactoring for this scenario just calls the same refactoring as the scenario *Local Groovy Refactoring* in chapter 6. So there are also the same further ideas, as in the other scenario. Additionally to them we found out that sometimes, the `selectionChanged` listener doesn't update, when it should. These happens just, if some text is selected, and the user clicks with the right mouse button on a different identifier. This tends to the result, the menu entry is disabled, even the element is derived from Groovy.

8 Automated Testing

Automated testing over the whole deployment time is an important element of a software project and an essential part of quality management. In our extension, we use a lot of already existing and tested code. For the most of these code, its not usefull to test it again. Additionally to this, testing between the two languages does not realy work on the unit-test level. So we build a test infrastructure, which is based on the *Groovy Eclipse Plug-in* testcase. This allowed us to test the whole refactoring in one piece with different testcases.

8.1 Testing Infrastructure

If not working properly, refactoring can be dangerous and is likely to create a lot of work for the refactoring operator, if it doesn't work as intended. So the quality of this automation is the most important. In order to guarantee quality, a lot of tests for any special kind of refactoring are needed.

To easily add new tests during development, we built a test framework which is able to read special testfiles, execute the refactoring and compare the output with the expected one in the testfiles. This test framework is based on the Eclipse plug-in tests, which means that the tests start a new instance of Eclipse, with a temporary testproject and simulate all required user actions.

For every test case execution, the Eclipse instance has to create a new project, and add all the files to it. Then, the project gets compiled, to check if the code is valid, before the refactorings can get started. At the end, the file content gets checked against the expected one, and the whole project has to be deleted. Due to this whole bunch of work, every test case takes a few seconds, which accumulate over all the tests to a few minutes. But the tests are that important for the project, this doesnt matter.

To suit our needs, we used the basic features of the already existing framework to add files and packages. In addition to this, we implemented functions for reading the file content, and comparing it with the expected one. Of course, we also implemented the additional features to programmatically start a refactoring.

Our extension of the tests are split in two parts.

- The test suites, which collect all the tests together.
- The test cases, one for each test, containing the testcode.

The following diagramm shows a schematic overwiev of the test framework

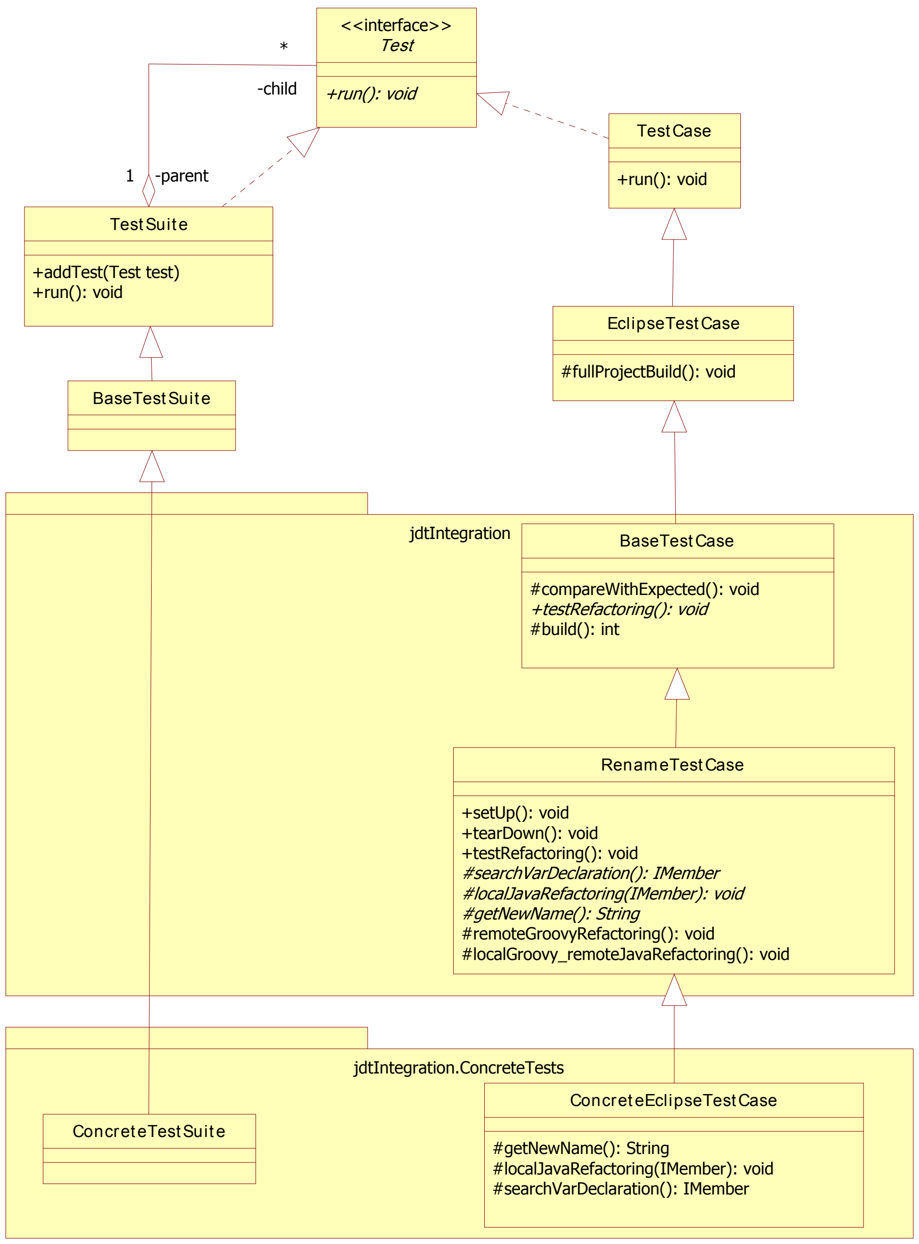


Figure 8.1: Test classes for file-based testing environment

Found failures in Eclipse plug-in test framework

During the work with this framework we encountered a few smaller issues which we fixed directly. Right at the beginning, we discovered that on a Windows plattform, the testproject can't be deleted after each testcase. So from the second test case, there were failures because the project already existed.

After a few tries, we figured out, that somewhere in the project there still were some references to the files, but after manually calling the garbage collector everything gets deleted as it should.

Later we noticed some errors with the `SelectionHelper`. Sometimes the selection was off by a few characters. We identified the testfiles, which worked, and those who had a wrong selection. After compairing these two groups, we could isolate the problem by the `newline`-symbol on different plattforms. When adding a file to an existing project, the test frameworks automatically adds the package name to the top of the files, followed by a blank line. The linebreak character used for this was just a "\n". After replacing this newline symbol with the `System.getProperty("line.separator")` everything worked fine.

Listing 8.1: Function to add a new Java file to the testproject

```java
public IType createJavaType(IPackageFragment pack, String cuName,
      String source) throws JavaModelException {
  String lineSeparator= System.getProperty("line.separator");
  StringBuffer buf = new StringBuffer();
  buf.append("package " + pack.getElementName() + ";");
  buf.append(lineSeparator);
  buf.append(lineSeparator);
  buf.append(source);
  ICompilationUnit cu = pack.createCompilationUnit(cuName,
      buf.toString(), false, null);
  return cu.getTypes()[0];
}
```

8.2 Test Suite

The main task of a test suite is to collect tests together. Additionally, in JUnit, test suites implement the composite pattern [GoF95]. This allows to easily add new test suites to the existing test suites, as a new node to the tree. We built test suites for each kind of element we wanted to rename. (Classes, Fields, Methods) Our implementations of the test suites take the different test files, matching them with a naming pattern, and generate the specific test case object for each one.

8.3 Test Case

A specific test case starts its work with reading properties and contents out of the test files and adding the content files to the test project. This works the same way for all the tests, so it is based on the `BaseTestCase` class. To implement a new kind of test based on this framework, it's only needed to extend `RenameTestCase` and add the abstract function `testRefactoring()`.

In Eclipse, the existing Java-refactorings are started from the JDT-own stucture based on the `IElement` interface. But the Groovy-refactoring are started with a selection in the specified file. To ease up the test framework, we handled this difference in the testfiles.

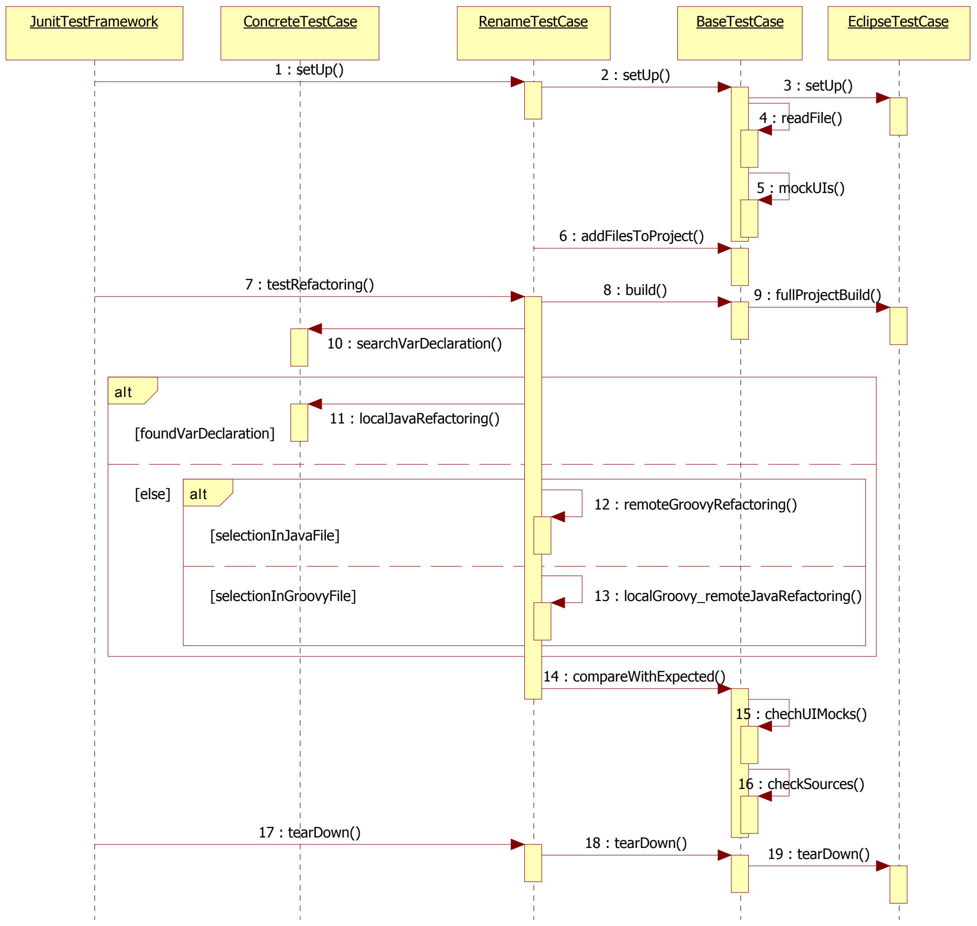


Figure 8.2: Sequence diagram over the whole test framework

The `BaseTestCase` class holds all additional functions and, as you can see in the diagramm above, reads the test files and compares the result of the refactoring with the expected source. All concrete functions are located in the `ConcreteTestCase` class. Because a Groovy-refactoring is called with a text selection, the programmatical start of this refactoring is found in the `RenameTestCase` class.

8.4 Test Files

The testfiles are simple textfiles, with all the information for a specific test. It is divided to three different sections, which are explained below. Every Section starts with a `###[token name]` token, and stops when the next section starts, or the `###end` token.

Property Section

The first section of the testfiles is the property section, defined by the `###prop` token. It is used to define some overall properties, like where the selection will be for the refactoring, or the new name of the renamed element. In the test case, these properties are stored in a map, and can be accessed via the name. ([name]=[value])

```
###prop
selectionInFile=JavaClass.java
startLine=6
startColumn=5
endLine=6
endColumn=8
newFieldName=newNAME
```

Source Section

The property section is closed by the beginning of the source section, defined by the `###src` token. This section contain at least one Java and one Groovy file, which are added to the test project, before the test starts. These files are split by the `#NEXT` token. Also every file in this section has two different sections. At the beginning the file property section, where is the filename and in which package the file belongs. Here are even the Ambiguous candidates for the UI-mock listed. Split by a line with just three colons the file properties section is closed, and the source section with Java or Groovy sourcecode beginns.

```
###src
Package=code
Filename=JavaClass.java
:::
public class JavaClass {
  public void method() {
    Field  field = new Field();
    field.myString = "Ciao";     // selection from the properties
                                 // section refer on this line
  }
}
#NEXT                            // a new file begins
Package=code
Filename=GroovyClass.groovy
:::
public class Field{
  String myString = "Hello"
}
```

```
public class GroovyClass {
  def m(){
    Field field = new Field()
    field.myString = "Hola"
  }
}
```

Expected Section

The expected section has the same layout as the source section, but here, files are stored the way they are expected after running the tests. This is the last section of the testfiles, and closes with the `###end` token.

```
###exp
Package=code
Filename=JavaClass.java
:::
public class JavaClass {
  public void method() {
    Field field = new Field();
    t.newNAME = "Ciao";          // the refactoring should rename here

  }
}
#NEXT
Package=code
Filename=GroovyClass.groovy
:::
public class Field{
  String newNAME = "Hello"
}

public class GroovyClass {
  def m(){
    Field field = new Field()
    field.newNAME = "Hola"       // and here
  }
}
###end
```

8.4.1 Used properties in Test files

In our tests we used the following properties:

Where	What	Function	Parameter Name	Value
test properties	all tests	selection	selectionInFile	filename with ending
			startLine	line of selection start
			startColumn	column of selection start
			endLine	line of selection end
			endColumn	column of selection end
	local Java		className	name of the selected class
	method test	set new name	new method name	new name
	field test		new field name	new name
	class test		new class name	new name
file properties	all tests	package	package	path of the package
		filename	filename	filename of the file
		accept ambiguous element	AcceptLine	line number

Table 8.1: Parameter for Test Files

8.5 Starting a Java refactoring programmatically

The JDT refactorings can even get started programmatically. For this, its just needed to get the `refactoring contributor` of the refactoring you want to start. Out of the contributor it is possible to create a `refactoring descriptor` which you have to parametrize with the `IJavaElement` you want to refactor and a boolean, if you want to update the references too or not. If you want to start a rename refactoring, its also needed to set the new name in the descriptor. With all this information added to the refactoring descriptor, you can create a refactoring and start to check the initial and the final conditions. After everything is ok, you can create a change object and apply the changes.

Listing 8.2: Example of a programmatical refactoring start

```
RefactoringContribution contribution = RefactoringCore
    .getRefactoringContribution(IJavaRefactorings.RENAME_FIELD);

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

// set the Java element you want to rename
descriptor.setJavaElement(element);

// set the new name for the Java element
descriptor.setNewName(newName);

// refactor the references too
descriptor.setUpdateReferences(true);

RefactoringStatus state = descriptor.validateDescriptor();

try {
  Refactoring refactoring = descriptor.createRefactoring(state);
  state.merge(refactoring.checkInitialConditions(pm));
  state.merge(refactoring.checkFinalConditions(pm));
  Change change = refactoring.createChange(pm);
  change.perform(pm);
} catch (CoreExceptioin e) {
  e.printStackTrace();
}
```

8.6 Ambiguous selection mock

To test every kind of the refactorings, even some tests with ambiguous candidates are needed. Usually, if a user starts the refactoring, he can select which of them should be renamed, and which not. When running automatic tests, this doesn't work. (The test machine doesn't even has a graphical user interface.)

There were two different ways to solve this problem. To simulate the user input, or to mock the selection. Simulating the user input smelled of much work and of being error-prone, so we decided to mock the selection. To add the usability of changing the way of the selection, we added a static method `setRefactoring` to the class `AmbiguousSelectionAction` which instantiates the usual refactoring. This method takes the `class object` of a class, extended from `GroovyRefactoring`.

At the instantiation of our mock class, it checks all ambiguous candidates. The ones which are selected in the test file get moved to the definitive list.

Listing 8.3: Add a refactoring mock

```
public static void setRefactoring(Class<? extends GroovyRefactoring> g) {}
```

8.7 Eclipse Unit Tests

Testing all the refactorings at once is realy nice, but searching an error in a huge amount of code (and this is exactly, what a programmatical refactoring is) takes a long time. So during deployment, it's good to have some unit-tests too.

For this, we added a new `TestSuite` to the other suites, especially for the unit tests. The tests themselves are split on what kind of element (class, method or field) will be tested. All these classes extend from the class `ProgrammaticalRenameTest`, which is an extension of `junit.framework.TestCase`.

8.8 Buildserver

As written in the intro of this chapter, automated tests are important for a software project, but just executing the tests manually on the working machine of a developer, doesn't ensure all the tests realy run and pass. The build of the plug-in has to work automaticaly. This improves the quality of the product and safes a lot of time for the developer.

Building an eclipse plug-in isn't easy to automate, because there are a lot of depencies between different plug-ins. For this reason, Markus Barchfeld created his Eclipse plug-in called "Pluginbuilder"[PluginBuilder]. With this plug-in, it is much easier to collect all the necessary parts from the SVN repository and generate an ANT buildfile for the buildserver. Building the *Groovy Eclipse Plug-in* adds a few additional problems. For further information about these problems, please refer to the report of our predecessors [KKK08].

Luckily for us, we had already a running build server from our predecessors and a helping hand from Michael Klenk [1].

8.8.1 Problems

At the beginning of the project, we struggled for three weeks with a non-working build on the server. The simple solution of the problem was, that a package form the groovy core was not exported. This export had to be added to the manifest file of the plug-in, which we didn't knew before.

[1]Michael Klenk is one of the developers of the *Groovy Refactoring plug-in* [KKK08]

8.8.2 Documentation Build

The build server does not just the project builds. Even the generation of this project documentation works on the build server. The setup for this build was simple and did not cause any problems.

1. CruiseControl calls the `cc-build.xml` file.
2. The `cc-build.xml` file pulls the newest LaTeX sources out of the SVN repository.
3. and generates the pdf file.

9 Summary

9.1 Results

Developing for a huge framework such as the Eclipse platform was a big challange for both of us. Before our bachelor thesis, we were simply Eclipse users with a fundamental knowledge of Groovy. In the beginning of our project, we had to quickly familiarize our self's with the back-end of Eclipse, the internals of the Groovy language and with the work of our predecessors. This was a hard time in the beginning, but continuously began to make more and more fun with the progress of the project.

For this reason, we're very happy that we reached all our defined goals and could provide additional values to the *Groovy Eclipse Plug-in*. Surely we learned a lot trough these months. We've experienced Eclipse, that we both use since a long time, from a deeper view and as very mature IDE designed for extensibility. The *Plug-in Development Environment* offers a lot of aid you don't necessarily expect, but surely never want to miss once you know them. We also could further improve our knowledge about Groovy, altough we had expected to have more touching points with the language itself. The biggest challenge was to deal with it's dynamic nature. The testing also required a lot of attention, because simple unit testing couldn't fulfill our requirements. As we depended on the eclipse runtime to be present, we had to build a sophisticated testing architecture to guarantee the quality of our code.

The result of our thesis is a reliably working plug-in that is ready for *Crosslanguage Refactorings*. All directions in which a refactoring between Groovy and Java can be launched, are supported. Special cases, that need additional user input, are covered as well. With our solution, refactoring support for Groovy in Eclipse reaches a satisfying integration level and offers improved support for developers.

9.2 Outlook

With the end of the project, our extension reached a level that is ready for production use. We look forward to integrate our results into the official *Groovy Eclipse Plug-in* repository. This step had to be postponed after the thesis, because we wanted to submit the final stage of the software in one step. We're eager for feed-backs from the community and expansion ideas we didn't thought off yet. Surely we will follow the further development of the *Groovy Eclipse Plug-in* and the Groovy projects itself. As the whole plug-in is still under heavy development, there are always some itches and glitches to fix. Who knows, maybe we keep continuing to help improving the plug-in where possible in our free time?

9.3 Known Issues

Except of the known problems, which were already described in the “Unsolved Problems” chapters in some of the scenarios, there are two remaining issues we couldn’t solve:

- **Uncertain compilation faults:** Sometimes, the compilation of Groovy files results in corrupted class files after saving changes, even if the code itself has no faults. When this happens, the AST cannot be retrieved anymore, thus preventing the refactorings to work. If this happens, the edited files have to be changed and saved again, by adding one whitespace character for example. This bug is a problem with the Groovy compiler from Eclipse.
- **Sub-keyword name changed:** When a field for example consist of a name that is part of a keyword used on the declaration line, the Groovy refactoring tries to rename the keyword instead of the name. This is a bug in the refactoring plug-in that has been opened on the JIRA bug tracking system for Groovy.

9.4 Possible Extensions

This list offers some ideas for further extensions of the *Groovy Refactoring Plug-in*:

- **Groovy Language Model:** The current plug-in lacks a model that represents the language and indexes elements like the Java model. This would be really helpful for the whole plug-ins and also for the refactorings.
- **Starting refactorings from outline selection:** Currently refactorings are launched only from text selections. With a language model, the starting points could be extended to the outline or the type hierarchy.
- **Inline rename:** Replace the name input wizard page by inline renaming the selected element, when a refactoring is launched. The Java editor supports this behaviour and it would be pleasant for the Groovy editor. (Also see chapter 4.10).
- **Switch to ProcessBasedRefactoring** See chapter 6.4 : Groovy Refactoring Participants.
- **Import organisation:** A feature frequently used in the Java editor is the “Organize imports” function. This doesn’t exist for the Groovy editor yet and would be helpful.
- **Improved “Hyperlink to Type”**: When the `Ctrl` key is pressed and the mouse pointer hovers over a type, a link should appear leading the user to the type definition.
- **Search function:** As known from Java, a search function for types or a lookup for references, even if a bit tricky to realize in Groovy, could improve user assistance. This function could be realized more effective, if a language model (See first point) is available.

9.5 Personal Reflections

We wrote our personal reflections in german, because it's our mother tongue and we could express ourselves better.

Stefan Sidler

Da dies für mich die erste Mitarbeit in einem solch grossen Software Projekt ist, hatte ich am Anfang Mühe, die Lage zu überblicken. Ich wusste gar nicht recht, wo ich mit dem einarbeiten beginnen sollte. Durch das Studium der Arbeit unserer Vorgänger lichtete sich einiges, doch es gab immer noch viele unklare Stellen, weshalb der Start des Projektes etwas schleppend verlief. Als dann aber der Stein einmal ins rollen gekommen war, lief es zügig und auch die Motivation stieg wieder an.

Ursprünglich haben wir uns vorgenommen jeden Freitag einen Dokumentiertag einzulegen. Da wir am Anfang aber lange brauchten, bis wir einen Überblick erreichten, schoben wir die Dokumentation vor uns her, was sich leider bis kurz vor Schluss nicht mehr änderte.

Als nach Ostern Stefan Reinhard mit viel neuem Elan zurück kam, zog er das Projekt weiter an. Davon beinahe etwas überrannt, bezog ich in Diskussionen um Entscheidungen oftmals eine zu passive Rolle, was gegen Ende des Projektes zu Diskussionen führte. Dadurch gestärkt gingen wir dann jedoch in den Endspurt über.

Da wir die Dokumentation über lange Zeit vor uns hergeschoben haben, kamen wir gegen Ende des Projektes ziemlich unter Zeitdruck. Ich hätte nie gedacht, dass eine Dokumentation auf englisch so viel mehr Zeit in Anspruch nimmt. Aber das programmieren machte einfach mehr Spass.

Ich habe in diesem Projekt sehr viel gelernt. Nicht nur was das Programmieren angeht, auch wie man ein grosses, unbekanntes Framework anpacken, und sich einlesen muss.

Abschliessend möchte ich mich bei Prof. Peter Sommerlad und Michael Klenk für die gute Unterstützung bedanken. Beide hatten immer gute und konstruktive Ideen, wenn wir bei einem Problem nicht mehr weiter wussten. Auch wenn es gegen den Schluss etwas stressig wurde, hat die Arbeit mit ihnen sehr viel Spass gemacht.

Stefan Reinhard

Mein erster Kontakt mit Groovy kam durch das Grails Framework zu stande, und seither bin ich ein begeisterter Anhänger der Sprache. Daher entschied ich mich schon früh, meine Bachelorarbeit nach Möglichkeit an einem Projekt im Groovy Umfeld zu schreiben. Nachdem eine meiner Ideen jedoch von einem anderen Dozenten verworfen wurde, war ich sehr glücklich, mit dieser Arbeit dennoch etwas spannendes und sinnvolles im Zusammenhang mit Groovy gefunden zu haben.

Eclipse ist im Vergleich mit anderen IDE's (v.a. IntelJ IDEA und Netbeans) leider nicht gerade die erste Wahl für Groovy. Da ich während meiner Studienzeit aber hauptsächlich mit Eclipse zu tun hatte, war ich um so motivierter, mich für eine Verbesserung der Groovy-Unterstützung einzusetzen.

Die ersten Schritte waren etwas schwerfällig, da ich noch nie an einem Eclipse Plug-in gearbeitet habe, aber die Architektur doch zahlreiche Schnittstellen und Technologien bereit hält. Als diese einmal verstanden waren und ich auch die Arbeit unserer Vorgänger gründlich analysiert hatte, ging es plötzlich sehr produktiv voran. Leider war zu diesem Zeitpunkt schon ein gewisser Rückstand im Projektplan festzustellen.

Ich empfand es interessant zu sehen, dass ich zwar nicht gerade sehr viele Zeilen Code pro Tag schrieb, die Funktionalitäten unseres Plug-ins jedoch immer umfangreicher wurden. Dies lag daran, dass ich die selben Module immerzu wieder verbessern und erweitern musste, um das gewünschte Endresultat bei jedem Use Case zu erreichen. Schlussendlich kann ich sagen, dass mir die Entwicklung in einem so umfangreichen Projekt sehr viel Spass gemacht hat und ich bin davon überzogen, dass wir eine brauchbare Erweiterung des Groovy Plug-ins für Eclipse realisieren konnten.

Da dieser Erfahrungsbericht aber auch eine Reflektion der geleisteten Arbeit sein soll, möchte ich auch noch einige selbstkritische Punkte ansprechen. Es hat sich im Verlaufe des Projekts so ergeben, dass ich mich hauptsächlich um die Implementierung des Plug-ins kümmerte, während mein Mitstreiter vor allem das schreiben der Tests übernahm. Diese waren ziemlich umfangreich und komplex und ich gebe offen zu, dass ich froh war, dass er diese Sache übernahm. Dennoch hielt ich es für eine ungeschickte Aufteilung der Aufgaben, da so beinahe keine Ideen mehr gemeinsam entwickelt wurden. Softwareentwicklung, wie ich sie bisher erlebt habe, lebt jedoch vom gegenseitigen Gedankenaustausch und dem gemeinsamen Konsens.

Ausserdem haben wir die Zeiteinteilung grob unterschätzt. Die Dokumentation wurde viel zu spät angegangen und musste ausserdem in Englisch geschrieben werden. Dieser Umstand verzügerte die Arbeit zusätzlich, da ich mich zwar des Englischen mächtig fühle, jedoch nur sehr langsam im schreiben vorankomme. Ich hoffe jedoch, dass wir die wichtigsten Punkte der Entwicklung dennoch einigermassen lesenswert zu Papier bringen konnten.

Mit dem Abschluss dieser Bachelorarbeit bin ich insgesamt sehr zufrieden und möchte neben meinem Kollegen Stefan Sidler vorallem unserem Betreuer, Prof. Sommerlad für die umfangreiche Unterstützung und wertvollen Ratschläge danken.

10 Appendix

10.1 Environment

Development

For developing the *Crosslanguage Refactoring Plug-in*, we both used *Eclipse PDE*, with additional plug-ins for SVN (subclipse) and the Groovy-Eclipse plug-in itself. The build was on a *CruiseControl* server, and we used the *CCTray* utility to observe the builds. Our repository and buildserver was hosted on a virtual debian machine at the *Institute for Software* with apache and svn installed.

Documentation

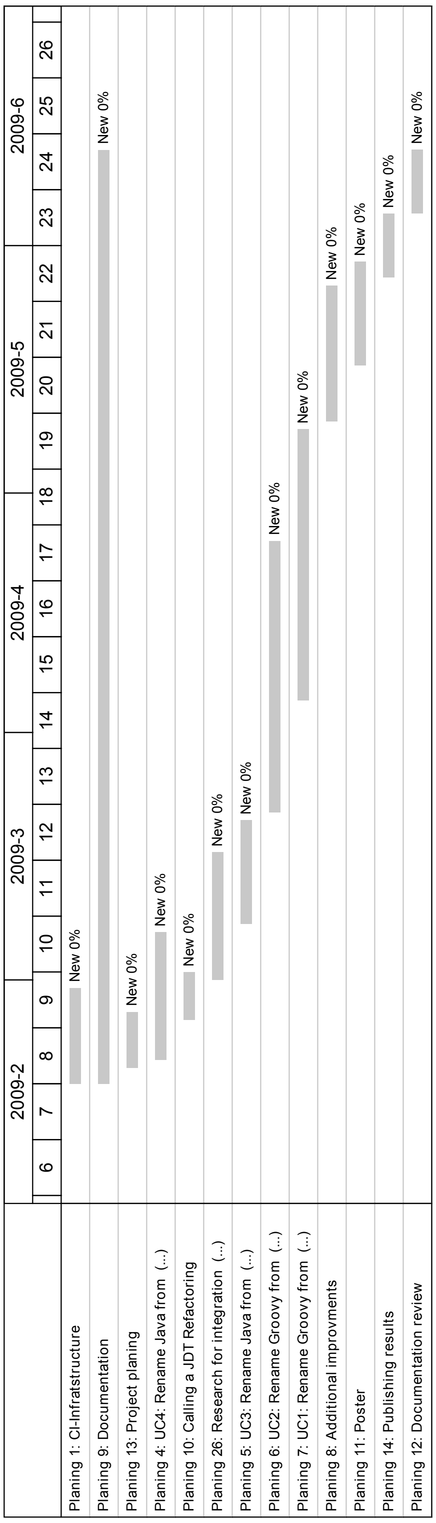
The whole documentation is written in $\LaTeX$. We both wrote in individual editors:*TexnicCenter* and *Texlipse*. All UML diagrams were drawn with *StarUML* and for nearly all other diagrams we used *Inkscape* and exported them to PDFs.

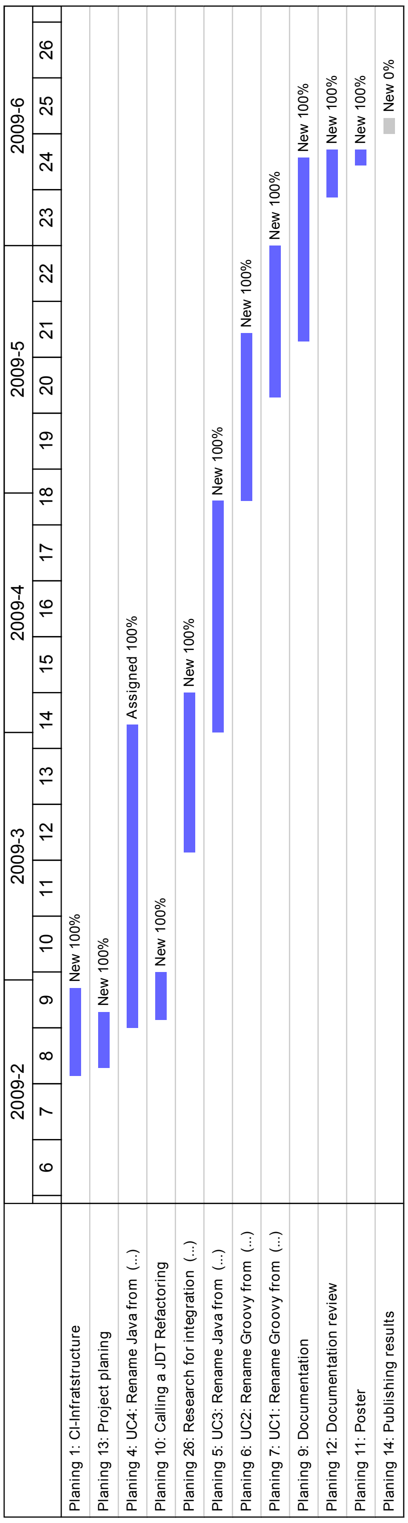
Project Organisation

All project managment issues were covered by *Redmine*, a *Ruby on Rails* software comparable to *trac*. We used *Redmine* for project planing including gantt-charts creation, issue tracking and the wiki for meeting invitations. It's also features a very nice source browser with diff view and syntax highlighting.

10.2 Project Planning

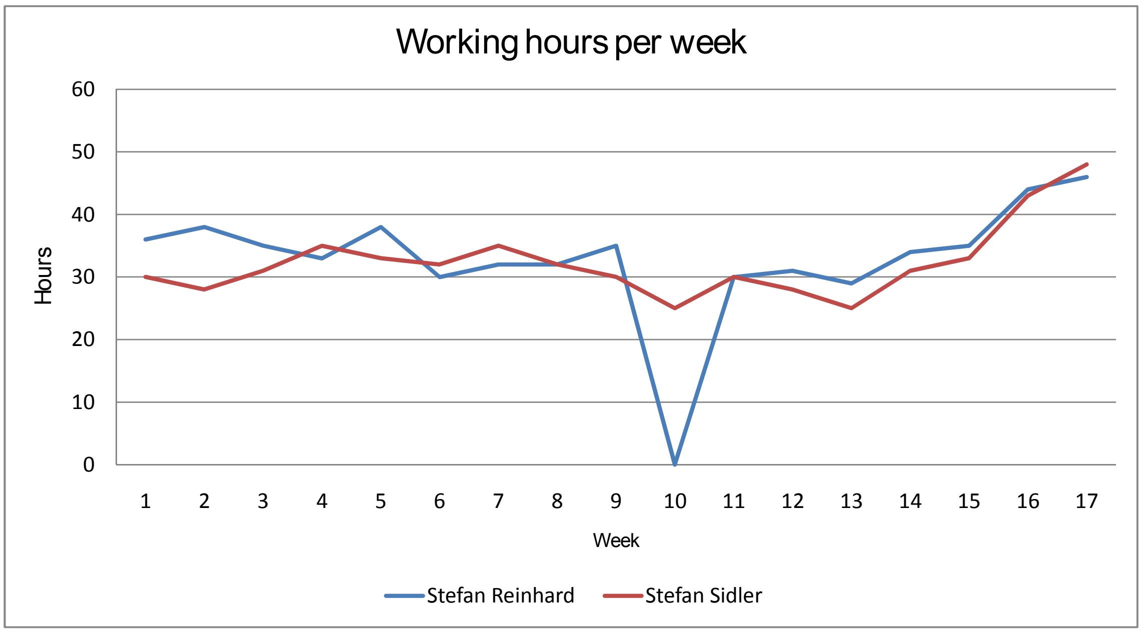
10.2.1 Time Schedule





10.2.2 Working Hours

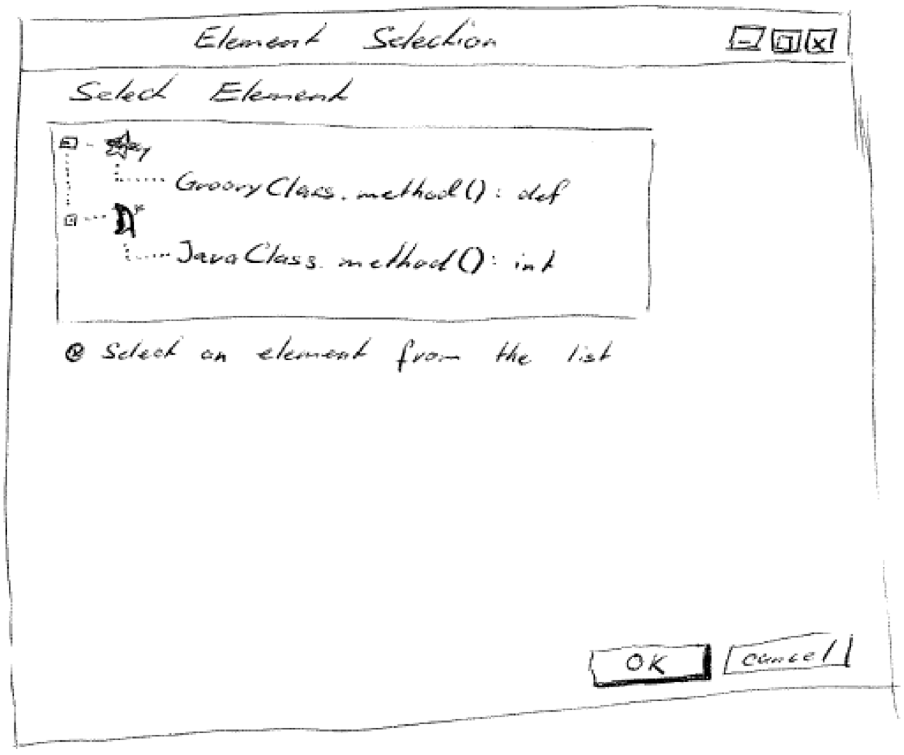
This Bachelor thesis was scheduled over a 17 week time period.



Cumulated working hours	
Stefan Reinhard	558
Stefan Sidler	549

10.3 External Design

We just needed to add one window for our extension. All others already existed from the Groovy refactorings.



Listings

List of Figures

Bibliography

[GoF95] Design Patterns. Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph E. Johnson, Addison-Wesley Longman, 1995

[EPi08] Eclipse Plug-ins 3rd edition, E. Clayberg, D. Rubel, Addison-Wesley, 2008

[GB04] Contributing to Eclipse, Principles, Patterns and Plug-Ins

Erich Gamma, Kent Beck, Addison Wesley, 2004

[Fow99] Refactoring: Improving the Design of Existing Code,

M. Fowler, Addison-Wesley, 1999

[PluginBuilder] Pluginbuilder (Eclipse Plugin),
Official Website: http://www.pluginbuilder.org/, 2007

[Groovy] The Groovy language,

Official Website: http://groovy.codehaus.org, 200

[GinA] Groovy in Action,

D. König, A. Glover, P. King, G. Laforge, Manning, 2007

[Ven08] Programming Groovy,

Venkat Subramaniam, Pragmatic Programmers, 2008

[JNS08] Beginning Groovy and Grails,

Chrisopher M. Judd, Jospeh Faisal Nusairat and James Shingler, Apress, 2008

[JSE] Using the Java search engine (JDT Developer Guide),

http://tinyurl.com/qp4yn9, 2008

[Wid07] Eclipse Corner Article: Unleashing the Power of Refactoring

Tobias Widmer, 2007

http://tinyurl.com/63qh7c

[Cla04] Pragmatic Project Automation,

Mike Clark, Pragmatic Programmers, 2004

[KKK08] Bachelor Thesis: Refactoring Support for the Groovy-Eclipse Plug-in, Martin Kempf, Reto Kleeb, Michael Klenk

http://groovy.ifs.hsr.ch, 2008

[JSR241] The original Java Specification Request for the Groovy Programming Language, James Strachan, Richard Monson-Haefel

http://jcp.org/en/jsr/detail?id=241, 2004

[TI] TIOBE Programming Community Index

http://www.tiobe.com/index.php/tiobe_index

[AST06] Eclipse Corner Article: Abstract Syntax Trees,

Thomas Kuhn, Olivier Thomann, 2006

http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html