



HSR – University of Applied Sciences Rapperswil

Institute for Software

Refactoring Support for the Eclipse Ruby Development Tools

Term Project

Thomas Corbat
tcorbat@hsr.ch

Lukas Felber
lfelber@hsr.ch

Mirko Stocker
me@misto.ch

<http://morki.ch/rubyrefactoring>

supervised by Prof. Peter Sommerlad

July 2006

Abstract

Refactoring is a very useful technique for software engineers to ensure the healthiness of their code over time. Modern IDEs support the developer by automating common refactoring tasks and thus making it even more useful. Although there are several different IDEs for Ruby, none of them support automated refactorings. Nevertheless, in this term project we implemented some refactorings and code generators for Ruby.

To achieve our goal, we extended the functionality of existing projects, mainly JRuby and RDT. JRuby is a Ruby interpreter written in Java. It can generate an abstract syntax tree from Ruby code and thereby allows us to get a lot of information about the source code. Unfortunately, comments were just read over and not represented in the AST, so we had to add this feature. Since modifying the AST without getting back source code does not make much sense, a rewriter has been implemented too.

RDT, the Ruby Development Tools, are a set of Eclipse plug-ins to develop Ruby programs. We added our refactoring plug-in to RDT and extended its functionality with refactoring support. We now have several code generation utilities and refactorings:

- Rename Local Variable
- Push Down Method
- Generate Accessors
- Generate Constructor using Fields
- Override Method

Management Summary

Motivation

During software development, the engineer often has to modify the existing code to make it more robust and less error-prone. While this does not change the functionality of the product, it certainly improves the maintainability, understandability and testability. This process is called refactoring. Doing refactoring by hand is often quite tedious and generally engineers are afraid to change working code. That is why most integrated development environments (IDE) automate common and often used refactorings. Working with automated refactorings can also save a lot of time because the changes an automated refactoring applies to a project in no time, would need a lot of a programmers time.

Ruby is a programming language, that was not very well-known to the bigger part of the world because it was developed in Japan and because there was no big company behind it to push it, like Sun does with Java and Microsoft with C#. A few years ago, Ruby broke trough to the english speaking programming community with the book Programming Ruby written by Dave Thomas. For Ruby, there is no existing programming environment we know of that supports automated refactorings, thus our motivation is to bring refactoring support to Ruby.

There are a few existing Ruby IDEs. The most interesting one for us are the Ruby Development Tools. The RDT are plug-ins written for the popular Eclipse framework, which was originally developed by IBM and is now open source. Our job was to write a further plug-in that extends the functionality RDT already provides for Ruby developers.

Goal

Our goal is to extend the existing functionality of RDT and its components to support automated refactorings. We needed to improve the existing components to satisfy our requirements, afterwards we could advance and implement as many refactorings as possible. Along with the improvements in the source code, we documented our experiences and the various approaches we have tried so any subsequent project can reuse the knowledge we gained.

Results

Our assignment was to write an extension to the RDT Eclipse plug-in. Our first task was to get in touch with the RDT to learn what we needed to implement our own plug-in.

We knew that the existing basic Ruby model needs to be extended to be able to handle comments in the Ruby source code. This extension was established during our work. We first planned to do this in the beginning part of our assignment, but we realized that this job took a lot more time than we expected. One of our team members worked the whole fourteen weeks to solve this problem.

During our analysis on the Ruby basic model, we realized that there were a few bugs. The fixing of those bugs took a lot of time we had not expected to spend.

The third part we took care of in our work was the implementation of the refactoring plugin and five refactorings. This work went well except that we planned to more of this work. Because of the additional work described above we had to care for we were not able to implement as much refactorings as we planned.

To sum up, we were able to rework the base components to fit our purposes. Although there are still some quirks, our work can be used well to refactor code. We had much more work to do on the underlying components than we expected, but most issues were fixed and the patches are sent, or are being sent to the respective people and some useful refactorings were implemented.

Outlook

This work will be continued as a diploma thesis with the current team. After the work on the basis, we can now concentrate on the implementation of other, more powerful refactorings. We also hope to get responses from the community about other refactorings that are wished and to hear what people think about our work.

Contents

1	Introduction	1
1.1	General Situation	1
1.2	Ruby IDEs	2
1.2.1	ActiveState Komodo	2
1.2.2	Arachno Ruby	2
1.2.3	FreeRIDE	2
1.2.4	Mondrian Ruby IDE	2
1.2.5	Ruby in Steel	2
1.2.6	Ruby Development Tools	3
1.3	JRuby	3
1.4	RadRails	3
1.5	Team	3
1.5.1	Mirko Stocker	3
1.5.2	Lukas Felber	3
1.5.3	Thomas Corbat	4
2	Refactorings	5
2.1	Implemented Refactorings	5
2.1.1	Code Generators	5
	Generate Accessors	5
	Generate Constructor Using Fields	7
	Override Method	8
2.1.2	Refactorings	9
	Rename Local Variable	9
	Push Down Method	10
2.2	Other Refactorings	11
2.2.1	Pull Up Method	11
2.2.2	Merge Ruby Class Parts	11
2.2.3	Convert Local Variable to Field	12
2.2.4	Encapsulate Field	13
2.2.5	Extract Class	14
2.2.6	Extract Method	14
2.2.7	Inline Class	15
2.2.8	Inline Method	16
2.2.9	Move Field	17

2.2.10	Move Method	17
2.2.11	Rename	17
2.2.12	Replace Temporary Variable with Query	19
2.2.13	Split Temporary Variable	20
2.3	Comment Handling	21
3	JRuby	22
3.1	Lexer	22
3.1.1	General	22
3.1.2	Classes	23
3.1.3	States	25
3.2	Parser	25
3.2.1	General	26
3.2.2	Classes	27
3.2.3	Call Sequence	29
3.2.4	Generation	29
3.3	Abstract Syntax Tree	30
3.4	Positioning Errors	30
3.5	Errors in the Parser	32
4	Ruby Development Tools	35
4.1	JRuby	35
5	Comment Handling	36
5.1	General	36
5.2	Placing the Comments	36
5.2.1	Decorating Nodes	36
5.2.2	Separate Comment Nodes	37
5.2.3	Comments in the Class Node	37
5.3	Lexer Modifications	38
5.4	Comment Handling in the Parser	38
5.5	Comment Handling in the Lexer	40
5.5.1	Context Buffering	41
5.6	Node Creation	41
5.7	Association Rules	41
5.8	Known Issues	42
5.8.1	Broken Parser Rule	42
5.8.2	Lost Comments	42
5.9	Change Overview	42
5.9.1	Node	42
5.9.2	SourcePosition	43
5.9.3	CommentNode	43
5.9.4	RubyYaccLexer	43
5.9.5	DefaultRubyParser	43

5.9.6	ParserSupport	44
6	AST Rewriter	45
6.1	General	45
6.2	Difficulties and Pitfalls	48
6.2.1	Local Variables	48
6.2.2	Parentheses	48
6.2.3	Strings	50
6.2.4	Operator Precedence	50
6.2.5	Here Documents	51
6.2.6	Blocks	52
6.2.7	Comments	52
6.3	Formatting	52
7	Refactoring Plug-In	54
7.1	Used Extension Points	54
7.1.1	RDT Popup Menu Extension Point	54
7.1.2	Eclipse Menu Bar Extension Point	56
7.2	Fundamental Design of the Plugin	57
7.2.1	Refactoring Model	57
7.2.2	Node Provider	58
7.2.3	Node Decorators	58
7.2.4	Classnode Providers	58
7.2.5	Edit Providers	60
7.2.6	Signature Providers	60
7.2.7	Offset Providers	60
7.2.8	HSRFormatter	60
7.3	Implemented Code Generators	62
7.3.1	Generate Accessors	62
Demonstration		62
Procedure		65
Class Diagram		65
7.3.2	Generate Constructor Using Fields	67
Demonstration		67
Procedure		68
Class Diagram		68
7.3.3	Override Method	70
Demonstration		70
Procedure		71
Class Diagram		72
Extensions and Ideas		72
7.4	Implemented Refactorings	73
7.4.1	Rename Local Variable	73
Demonstration		73

Procedure	74
Class Diagram	75
Extensions and Ideas	75
7.4.2 Push Down Method	76
Demonstration	76
Procedure	77
Class Diagram	78
Extensions and Ideas	78
7.5 Eclipse	79
7.5.1 The Plug-In System	79
7.5.2 Refactoring Support	79
Basic Example	80
Class Diagram	81
7.5.3 Eclipse Rules	82
Lazy Loading Rule	82
Monkey See / Monkey Do Rule	82
Safe Platform Rule	82
7.5.4 Resource Management	82
Resources Class Diagram	83
Code Samples Diagram	83
8 Testing (Automated)	85
8.1 Refactorings	85
8.1.1 Edit Provider Tests	85
8.1.2 Tree Content Provider Tests	86
8.2 AST Rewriter	86
8.2.1 Testing of JRuby adaptations	87
9 Summary	88
9.1 Results	88
9.2 Further Work	88
9.3 Known Issues	89
9.4 Outlook	89
9.5 Personal Comment	89
Appendix	89
Field Reports	90
Environment	91
Project Schedule	94
List of Figures	96
Index	97
Nomenclature	97

1 Introduction

Refactoring is a very useful concept for every software engineer. Martin Fowler, the author of the prominent book Refactoring[?] says:

"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."

During software development, you often have to change your code to adapt it to the ever changing environment or just to make it more readable or testable. Although refactoring does not rely on software tools, it is quite useful if you do not need to perform the necessary steps on your own and you do not have to worry that you missed something or messed it up. Examples of common used refactorings are:

- Renaming of classes, methods and variables.
- Moving around code, for example methods or variables.
- Extracting parts of code into other methods or even classes.

The classic example of an automated refactoring is the Smalltalk Refactoring Browser [?]. A more recent and also very popular automated refactoring implementation can be seen in the JDT of Eclipse.

In Ruby the main difficulty is also one of its greatest language features: dynamic typing. Dynamic typed languages offer a lot of freedom to the programmer, however, it is hard and sometimes even impossible for an IDE to figure out the type of an object. Thus refactorings with a large scope, like the renaming of public methods, are a real challenge.

The goal of this term project is to implement refactoring support for the Ruby Development Tools. To achieve this, we first have to create a basis for the refactorings: we need to introduce comments in the lexing and parsing process and a rewriter for the AST. Depending on the time we have left, we try to implement as many refactorings as possible. We are going to continue our work as a diploma thesis.

1.1 General Situation

Ruby¹ received a lot of attention in the last few years and is becoming more popular every day, especially with the booming web application framework Ruby on Rails. The

¹<http://www.ruby-lang.org/en/>

language has been developed by Yukihiro "Matz" Matsumoto, and was released over ten years ago. Ruby has some very interesting features, like a very strong object orientation and dynamic typing, although some people seem to be afraid of that. But discussing this does not lie in the scope of this document. We believe that Ruby has a bright future and we hope we can contribute to it.

1.2 Ruby IDEs

There are already a lot of IDEs for Ruby available, both commercial and open source, and none of them has support for automated refactorings. We extracted the following information from the respective websites without further investigating it.

1.2.1 ActiveState Komodo

Komodo² from ActiveState is a commercial IDE for Ruby (among others like Python, Perl, etc.) but it does not contain refactoring support. It runs on Linux, OSX and Windows.

1.2.2 Arachno Ruby

Arachno Ruby from Scriptolutions³ is a commercial IDE and does not support refactorings either.

1.2.3 FreeRIDE

FreeRIDE⁴ is an open source project and is written entirely in Ruby. They mention refactoring support on their website, but it is in a very early state and as far as we know not included in the current version.

1.2.4 Mondrian Ruby IDE

Mondrian⁵ is a freely available IDE for Ruby, but the development has stalled recently. It does not support automated refactorings either.

1.2.5 Ruby in Steel

Ruby in Steel⁶ is the Ruby plug-in for Microsoft's Visual Studio and is freely available. As far as we know, it does not contain support for automated refactorings.

²<http://www.activestate.com/Products/Komodo/>

³http://www.ruby-ide.com/ruby/ruby_ide_and_ruby_editor.php

⁴<http://freeride.rubyforge.org/wiki/wiki.pl>

⁵<http://www.mondrian-ide.com/>

⁶<http://www.sapphiresteel.com/>

1.2.6 Ruby Development Tools

The Ruby Development Tools⁷ are a set of plug-ins for Eclipse. Bringing refactoring support to it is the aim of this project.

1.3 JRuby

JRuby⁸ is a Ruby interpreter written in Java. It is fully compatible with Ruby 1.8.2 and provides most of the built-in classes of Ruby. You can even interact with Classes written in Java and let Ruby classes implement Java interfaces. However, there are also some limitations, for example you cannot use Ruby extensions written in C.

1.4 RadRails

RadRails⁹ is an IDE for Ruby on Rails based on Eclipse and the RDT. It contains everything you need to develop, manage, test and deploy your Rails application. We hope they are going to pick up our plug-ins to make RadRails even more powerful.

1.5 Team

The team consists of three computer science students from the University of Applied Sciences in Rapperswil, Switzerland. We are working under the supervision of Prof. Peter Sommerlad, who is well-known as the co-author of the book Pattern-Oriented Software Architecture: A System of Patterns[?].

1.5.1 Mirko Stocker

Mirko Stocker is a 23 years old software engineer from Uerikon, Switzerland who is in his last semester bevor his diploma thesis. Currently, his main interest is in Ruby and of course RDT and JRuby. He is a big fan of the Pragmatic Programmers and is currently working through their book on the Ruby quizzes. Ship-It and Data Crunching are ready in the shelf. Besides the software stuff, he likes to read fantasy literature, going to the cinema and spending time with his girlfriend. You can find his website at <http://misto.ch>.

1.5.2 Lukas Felber

Lukas Felber lives in ZÃ¼rich, Switzerland and is like Mirko 23 years old and in his second last regular semester at the university. He read a book about Ruby, but had no more knowledge than that with Ruby before starting this project. But he likes the part he saw of it till now. In his free time he likes to spend time with friends, go out

⁷<http://rubyclipse.sourceforge.net/>

⁸<http://jruby.sourceforge.net/index.shtml>

⁹<http://www.radrails.org/>

and play floorball. He not only plays floorball but is also president his floorball team in ZÃ¼rich. Besides studying at the university in Rapperswil, he works at the Swiss Federal Institute of Technology ZÃ¼rich, Integrated System Laboratory as part time webapplication developer. When there is time between all the other things he does, he likes to read fantasy literature and go to the cinema as well.

1.5.3 Thomas Corbat

Thomas Corbat is with his 22 year by far the youngest member of the team. He is also studying computer science at the HSR, the university of applied science in Rapperswil. Before this project he has not had any experience with Ruby, but aquired some knowledge about compiler engineering during his studies. In the meantime he learnt almost the whole Ruby syntax just by pouring over the production rules of its parser. Beside the school he has a part time job at SwissLife, a large life insurance company, in the internet security departement. In his free time he likes to read books, go swimming and play pen and paper roleplaying games.

2 Refactorings

In this chapter, we are going to describe the refactorings we have implemented and those we plan to implement in a further project. There is a really huge amount of refactorings, so the list of descriptions following in this chapter will only contain refactorings which make sense to be implemented for Ruby.

It is possible that there are refactorings in the list that are impossible to implement. A reason for this is that the dynamic typing of Ruby prevents the step of automating the process and thus the implementation of a refactoring. So there is no guarantee that all the refactorings can be implemented.

Note that the following descriptions only tell what the refactorings do and not how they are implemented. Implementation details can be found in the chapter Refactoring plugin 7.

If you like to have more detailed descriptions of the following refactorings, we would like to recommend you the book from Martin Fowler[?] or his website¹.

2.1 Implemented Refactorings

In order to get used to the idea of Eclipse's refactoring support and our working environment, we started our work with the implementation of the following code generators. Code generators are not directly refactorings. A refactoring is used to alter code but not to change its behavior. A code generator adds functionality and thus is not a common refactoring but uses similar mechanisms as an automated refactoring. It still is a very useful tool anyway.

In the next section, the implemented code generators are described. The section following that one describes the implemented refactorings.

2.1.1 Code Generators

Generate Accessors

In Ruby, read and write operations on object attributes are performed via so-called accessors. An accessor is a code segment which grants access to an objects attribute from outside the object. When an object has a lot of attributes, it can become really annoying to write all the accessors by hand. This might also be the sign of a bad design. In this case consider to refactor your code.

In Ruby there are two different types of accessors. First, there are simple accessors that allow reading, writing or both operations on an attribute. The second type are accessors

¹<http://www.refactoring.com/catalog/index.html>

implemented as methods. These accessors allow the object to intercept the value that will be assigned to the attribute. So you can check on types, if the value is in a certain range or you can monitor the access on the attribute, for example to notify observers. Seen from the outside where the object attributes are used, there is no difference between the two accessor types.

The generate accessors code generator provides the user with a selection of all the attributes contained in a class. He can choose if he wants to generate a reader, a writer or both and which accessor type he would like to have. The code generator now inserts the requested accessors into the class code.

Example The following code shows how an accessor of an attribute is used.

```
account = Account.new           #Creating an instance
account.balance = 5             #Setting the attributes
my_balance = account.balance    #Reading the attribute
```

This code shows a class providing simple accessors on different attributes.

```
class Account
  attr_reader :unique_id        #Allow reading from unique_id
  attr_write  :warning          #Allow writing to warning
  attr_accessor :balance       #Allow reading and writing
end
```

The following code example shows the usage of method accessors.

```
class Account
  #Allowing to set the balance
  def balance(balance)
    @balance = balance
    log "balance changed to #{balance}"
  end
  #Allowing to read the balance
  def balance
    log "balance was read: #{@balance}"
    @balance
  end
  #Logging the access (just printing to console)
  def log text
    puts text
  end
end
```

Generate Constructor Using Fields

The code generator Generate Constructor Using Fields creates a constructor. The constructor will have a variable number of arguments, which might be selected from a list of the existing fields in the class. In the constructors body, the class fields will be initialized with the values of the constructor parameters.

Examples Next you see the class BlogPost two times. Left without a constructor and on the right with a generated constructor initializing two of the tree fields.

```
class BlogPost

  def to_s
    puts @title
    puts @body
    puts @category
  end
end
```

```
class BlogPost
  def initialize title, body
    @title = title
    @body = body
  end
  def to_s
    puts @title
    puts @body
    puts @category
  end
end
```

Override Method

The Override Method code generator creates method bodies to a class. To do this, we look at the methods of the super class and choose some of them. Now the methods will be appended to the class. The added methods will have the same signature as the one from the super class. Like this the super class method will be overridden.

In Ruby, the constructor is called "initialize" and not named after the classname as in Java or C++. Because of this the code generator Override Method contains implicitly the code generator known as Add Constructor from Superclass. If a constructor is overridden its method body will contain a "super" call with the appropriate arguments to the super class constructor.

Examples In the first code sample you see the super class Dog and the empty class AggressiveDog that extends Dog. In second code segment you see AggressiveDog overriding the methods of Dog.

The original:

```
class Dog
  def initialize sound
    @sound = sound
  end
  def make_sound
    @sound
  end
end
class AggressiveDog < Dog

end
```

After refactoring AggressiveDog looks like this:

```
class Dog
  def initialize sound
    @sound = sound
  end
  def make_sound
    @sound
  end
end
class AggressiveDog < Dog
  def initialize sound
    super sound
  end
  def make_sound
    super
  end
end
```

2.1.2 Refactorings

Rename Local Variable

The Rename Local Variable refactoring does what its name promises. It renames local variables. Variables and parameters in methods are considered to be local, variables in blocks are not supported at the moment. This refactoring is more important than it seems. During the process of software development, you often change your mind about the name of variables. And variables should be named after what they represent. With Rename Local Variable this is done in no time and there will never be any nasty Copy&Paste mistakes.

Examples Here a before-after example demonstrating the Rename Local Variable refactoring.

In the following method the name "string" is not very expressive. So we rename it to "file_name", which we consider much better.

```
class MyFile
  def get_full_path dir_name,
                    string
    "#{dir_name}/#{string}"
  end
end
```

```
class MyFile
  def get_full_path dir_name,
                    file_name
    "#{dir_name}/#{file_name}"
  end
end
```

Push Down Method

The Push Down Method refactoring removes a method from the super class and pastes it into all its child classes. This makes sense if you realize that every child class needs the method to behave in its own way and it is not possible to generalize the behavior in the super class.

If you think that a method in a super class is logically better placed as in its deriving classes, then you also need the Push Down Method refactoring.

This refactoring might be extended with the functionality that it is also possible to push down other class members (e.g. fields, accessors). This would mean that the refactoring will be renamed from Push Down Method to Push Down Member.

Based on the the Eclipse JDT (Java Development Tools) our refactoring does not give the user the choice if the method will be pushed down into all or only a subselection of the childclasses. Here we see another possible extension for the refactoring.

Examples The following two code samples demonstrate the use of the Push Down Method refactoring, before and after refactoring.

<pre>class Shape def get_volume end end class Square < Shape end class Circle < Shape end end</pre>	<pre>class Shape end class Square < Shape def get_volume end end class Circle < Shape def get_volume end end end</pre>
---	---

2.2 Other Refactorings

In the following sections the not yet implemented refactorings are described.

2.2.1 Pull Up Method

The Pull Up Method Refactoring pulls a method out of one, or maybe several, subclass up into the super class. This prevents possible duplicated code if a super class has multiple child classes.

This refactoring might be extended with the functionality to pull up other class members like fields or accessors. This would mean that the refactoring will be renamed from Pull Up Method to Pull Up Member.

Examples The following two code samples demonstrate the use of the Pull Up Method refactoring.

<pre>class Animal end class Frog < Animal def make_sound puts SOUND end end class Sheep < Animal def make_sound puts SOUND end end</pre>	<pre>class Animal def make_sound puts SOUND end end class Frog < Animal end class Sheep < Animal end</pre>
---	--

2.2.2 Merge Ruby Class Parts

In Ruby it is allowed to declare a class in several places in the same or in other files. Class declarations following after a first one will logically be included into the first class declaration.

So an instance of a Ruby class will provide the combined functionality of all the class declaration parts even when they are spread over several files.

The Merge Ruby Class Parts refactoring pulls all those class declarations together and merges them into one single document. This might be useful to get a better overview in a larger Ruby project.

Examples The following class definitions might be in one or several different Ruby files. After the refactoring, the classes could also be placed in files named "animal.rb" and "food.rb".

<pre>class Animal def make_sound print @sound end end # Animal gets another method: class Animal def eat food puts "eating #{food}" end end class Food end</pre>	<pre>class Animal def make_sound print @sound end def eat food puts "eating #{food}" end end class Food end</pre>
--	---

2.2.3 Convert Local Variable to Field

This refactoring converts a local variable into a field. This will grant access to a variable that was only accessible inside a method or block to the whole class. If the local variable is a parameter, we also need to assign it to a corresponding field.

Examples The following class contains two methods, where we intend to use the local variable from the first method inside the second method, so we convert the local variable to a field and can now use it in both methods.

<pre>class Logger def log message @network.send message end def last_message #oops, I need the last # logged message here end end</pre>	<pre>class Logger def log message @message = message @network.send @message end def last_log @message end end</pre>
--	--

2.2.4 Encapsulate Field

The Encapsulate Field refactoring means that you take a public field, make it private and provide getter and setter methods to access it.

Seen through the Ruby glasses, a public field is one with any kind of accessor. Making it private means to remove those accessors. Adding setters and getters means to recreate these accessors. So the only scenario where this refactoring makes sense is when a field with simple accessors needs to get method accessors.

Examples The following two code sections demonstrate the Encapsulate Field refactoring.

```
class Person
  attr_accessor :surname
end
```

```
class Person
  def surname= surname
    # we could now apply
    # some checks here
    @surname = surname
  endwhere
  def surname
    @surname
  end
end
```

2.2.5 Extract Class

The Extract Class refactoring splits one class into two separate classes. This is necessary if the functionality provided by one class would have a better logical structure when it is split into two. This will improve the cohesion and thereby improve the structure and readability of your code. This refactoring uses the refactorings Move Field and Move Method.

Examples Here a demonstration of the Extract Class refactoring.

<pre>class Car def drive destination ... end attr_reader :driver_name def initialize driver_name @driverName = driver_name end end</pre>	<pre>class Car def drive destination ... end end class Driver attr_reader :driver_name def initialize driver_name @driverName = driver_name end end</pre>
--	---

2.2.6 Extract Method

Extract Method removes a block of instructions out of an existing method and creates a new one. The new method will be called from the existing method where the instructions have been removed. The local variables from the existing method that are used in the affected code block are passed to the new method as parameters. If one of those local variables is set inside the selected block it will be returned from the new method as its return value.

This might get very difficult in some cases where e.g. several variables are set.

Examples In the following code samples you can see how the Extract Method refactoring works.

```
def calc_cylinder_volume(radius, height)
  height * Math::PI * radius ** 2
end
```

```
def calc_cylinder_volume(radius, height)
  height * calc_circular_area radius
end
def calc_circular_area radius
  Math::PI * radius ** 2
end
```

2.2.7 Inline Class

The Inline Class refactoring moves the code from a usually small class that does not much into the class that uses the smaller class.

Examples Here you can see a short demonstration of the Inline Class refactoring.

```
class Contact
  def initialize name, number
    @name = name
    @phone = Phone.new number
  end
  def get_phone_number
    @phone.number
  end
end
class Phone
  attr_accessor :number
  def initialize number
    @number = number
  end
end
```

```
class Contact
  attr_accessor :number
  def initialize name, number
    @name = name
    @number = number
  end
  def get_phone_number
    @number
  end
end
```

2.2.8 Inline Method

The Inline Method refactoring removes a method and replaces the call with its content. This might make sense if you have quite empty methods without much logic, perhaps after applying other refactorings. Generally having multiple methods with clear names is usually better than one big chunk of code.

Examples In the following example, you can see how the size of the code is reduced after the use of the Inline Method refactoring.

```
def say_hello
  puts_hello
end
def puts_hello
  puts "Hello!"
end
```

```
def say_hello
  puts "Hello!"
end
```

2.2.9 Move Field

This refactoring moves, as its name says, a field from one class into another class. This is useful if you realize that the responsibility for a field is not in the owning class, but in another one. In this case, the field needs to be moved to that other class.

Examples Here an example showing the result of a moved field.

```
class House
  attr_accessor :spyhole
end
class Door
end
```

```
class House
end
class Door
  attr_accessor :spyhole
end
```

2.2.10 Move Method

The Move Method refactoring behaves almost like the Move Field refactoring, except that it moves methods and not fields. This is also useful if you came across a method in a class that is not really responsible for the functionality provided by this method.

An interesting idea might be to combine the refactoring Move Field and Move Method into one new refactoring called Move Member.

Examples Before the refactoring:

After the refactoring:

```
class Dog
  def get_color
    ...
  end
  ...
end
class Fur
  ...
end
```

```
class Dog
end
class Fur
  ...
  def get_color
    ...
  end
end
```

2.2.11 Rename

When you work on a larger project, sometimes the functionality and responsibility provided by classes, variables and methods might change over time. They need to be renamed so their name stays understandable. To prevent stupid copy and paste mistakes and save some time, the Rename refactoring will give you a hand.

Examples

Before the refactoring:

```
class OldClass
  def old_method old_variable
    ...
  end
  ...
end
```

After the refactoring:

```
class NewClass
  def new_method new_variable
    ...
  end
  ...
end
```

2.2.12 Replace Temporary Variable with Query

Sometimes it is wise to replace a temporary variable with a query or method call. Temporary variables encourage programmers to write longer and less confusing methods, because the temporary variable is only visible in the enclosing method. When you replace the temporary variable with a query, which is visible to the whole class you easily separate the method into smaller ones, using the Extract Method refactoring.

Examples Before the refactoring:

```
class Order
  def get_cost
    base_cost = @quantity * @item_cost
    shipping_cost = 20
    shipping_cost = 0 if base_cost >= 100
    base_cost + shipping_cost
  end
end
```

After the refactoring:

```
class Order
  def get_cost
    shipping_cost = 20
    shipping_cost = 0 if get_base_cost >= 100
    get_base_cost + shipping_cost
  end
  def get_base_cost
    @quantity * @item_cost
  end
end
```

2.2.13 Split Temporary Variable

Programmers often tend to assign a temporary variable several times. This is not recommendable because this means, that the temporary variable takes various responsibilities and thus can not be named properly. So the Split Temporary Variables refactoring helps avoiding this by creating a new temporary variable for each responsibility.

Examples Before the refactoring:

```
temp = get_price
temp.add 50
puts temp
temp = get_date
temp.set_format "DD.MM.YYYY"
puts temp
```

After the refactoring:

```
price = get_price
price.add
puts price
date = get_date
date.set_format "DD.MM.YYYY"
puts date
```

2.3 Comment Handling

We have seen that refactoring is very powerful technique if used sensible. To make a developer use them they have to work correct. An often forgotten or underestimated part of the source code are comments. You might say: Well when a comment is not moved with its method the functionality is not affected. That is truly correct but why should someone pull up a field if afterwards he has to cut and paste the comment seperately anyway?

So comment handling is an important topic that must not be omitted. Unfortunately it is also a bit tricky, especially if it is not planned to handle them from the beginning on. Additionally there is the problem that it can be quite difficult for an application to decide whether a comment belongs to a certain part of the source code or not. Finally it is up to the refactoring developer to create rules about the connection of source code and comment if that is not predefined through the AST. Even in popular and sophisticated products, such as the Eclipse JDT, comments are not always treated correctly. Some refactorings even swallow whole comment lines.

Nevertheless we have the ambition to retain the user comments.

3 JRuby

JRuby is a Ruby implementation written in Java, which is compatible to version 1.8.2 of Ruby. We use it to get an AST from the source files we need to refactor. In this chapter, we will give an overview over those themes and have a closer look at the problems we encountered while working on it. The basics of the JRuby lexer and parser find their roots in the C Ruby implementation. During our work we came across several leftovers of this genesis.

3.1 Lexer

As already mentioned we need to get an AST created from the source code. The creation of the AST is a quite difficult process which can generally be divided in two steps. The first step, which is handled by the lexer, is to break down the source code into so called tokens. These tokens are units of logically belonging together source code parts. The resulting token stream is passed to the parser, that creates the AST according to a set of production rules. In this part we will have a look at the creation of the lexer tokens.

3.1.1 General

The JRuby lexer is a handwritten program that is closely connected to the parser. It is designed to tokenize a file that contains ruby code. The application on source files of other languages is unlikely. All classes of the JRuby lexer are concentrated in the `org.jruby.lexer.yacc` package.

3.1.2 Classes

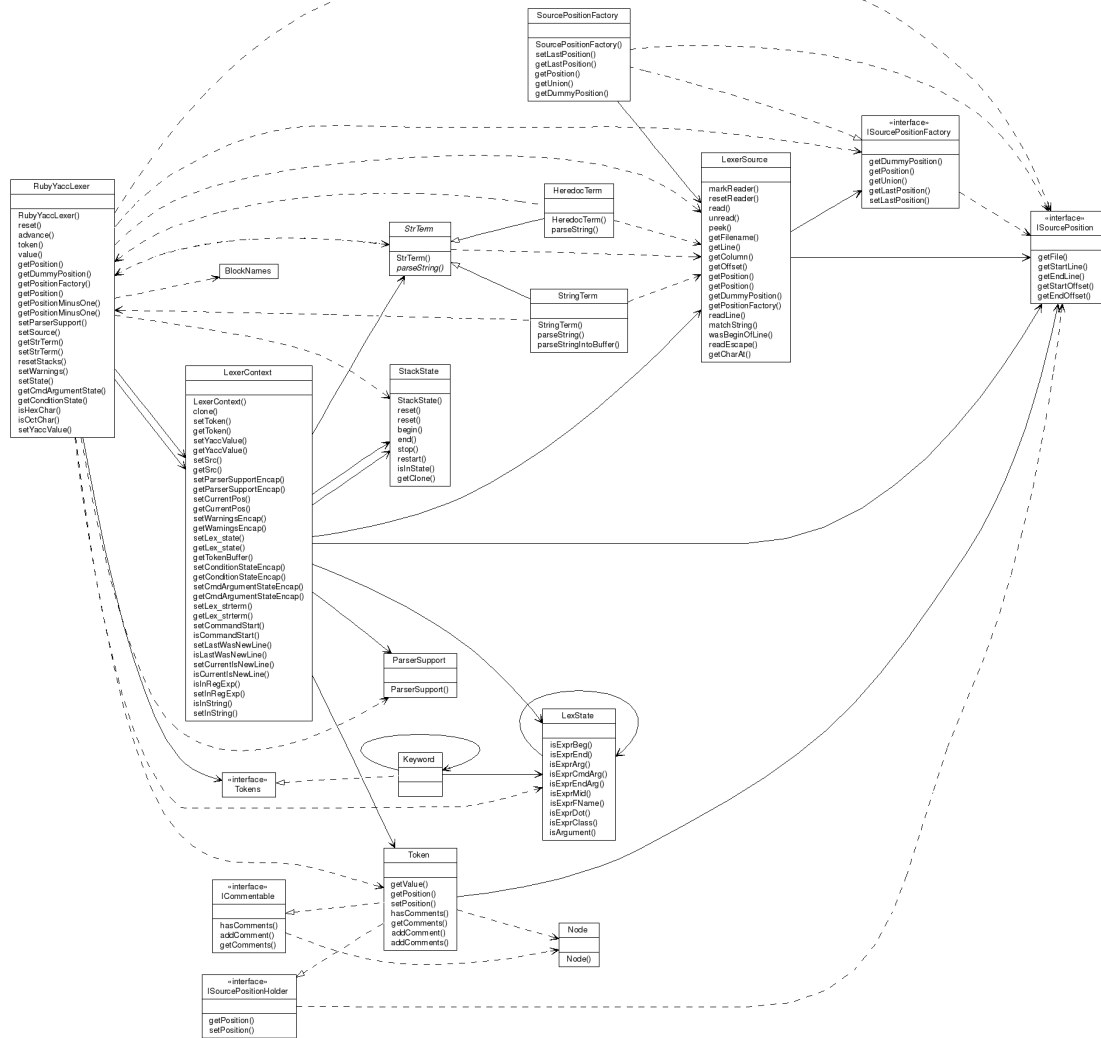


Figure 3.1: Lexer Class Diagram

HeredocTerm Is derived from StrTerm. As the name suggests it handles and parses the here-document parts in the source code. HeredocTerm is created and called by the RubyYaccLexer.

Keyword Contains the reserved keywords of ruby and the according destination states which the lexer switches into when seen. Interesting about this class is the hand-implemented hash table that makes it quite tricky to introduce new keywords. Fortunately we did not need to.

LexerSource Stores and manages the resources of the lexer. Reader and SourcePositionFactory are the nameable parts. Most important is the functionality to unread characters which is provided by the LexerSource.

LexState Contains all the possible states the lexer can be in. These states are made accessible through public constants in this class.

RubyYaccLexer Here we go. In the RubyYaccLexer class the core functionality of the lexer is implemented. Beside several fields, that store the whole context of the lexer, there is the yyLex method, which returns the tokens one by one. This method is not accessed directly but through the advance method. The already existing wrapping of yyLex made the implementation of a look-ahead functionality easier. The 1200 lines switch statement in yyLex seems to be worth some thoughts about Extract Method.

SourcePosition In SourcePosition the position of a token or any other item in the source code file can be stored. Such position information contains the source file path, the first and last character number and the start as well as the end line. The methods provided by SourcePosition are defined in the ISourcePosition interface.

SourcePositionFactory Is used to generate the SourcePositions for the tokens and the nodes. The resulting positions are highly depending on the states of the LexerSource and the SourcePositionFactory itself. It is relevant which position had been generated or returned lastly thus the unobtrusive method getPosition has side effects which might result in unexpected behavior.

StackState Can memorize boolean values and pass them back to its user. It is used in the lexer to keep information about condition and command argument states.

StringTerm Provides almost the same functionality as the HeredocTerm class but for strings instead of heredoc parts. It also extends the StrTerm class.

StrTerm Is an abstract class which implements nothing at all. It just specifies that the deriving classes (HeredocTerm and StringTerm) have to implement the parseString() method. In our opinion this class should clearly become an interface if there is no intention to implement any functionality. This clearly seems to be an artifact of the C Ruby origins and a good target for refactoring.

SyntaxException Thrown if the syntax rules get broken. Additionally to the common exception information it contains an ISourcePosition object to show where the syntax exception occurred.

Token The end-product of the lexer. Although the token generally is represented by an integer value it is usually necessary to provide additional information about the current token. Beside the position any object value might be relevant and is stored in an instance of this class. Via the `RubyYaccLexer` the token can be accessed.

3.1.3 States

To get a better insight into the lexer we have tried to outline its state machine. Especially the behavior concerning the hiding of newline tokens was in our point of interest. Unfortunately the state machine revealed to be an almost fully meshed structure of states and transitions. That means that it is possible to get into every state from nearly any other state. The drawing is complex enough itself so we have not been able to insert all the transition properties (lexer input, lexer output, preconditions). To have a detailed list with the possible state changes we have created a spread sheet showing them grouped by the destination state.

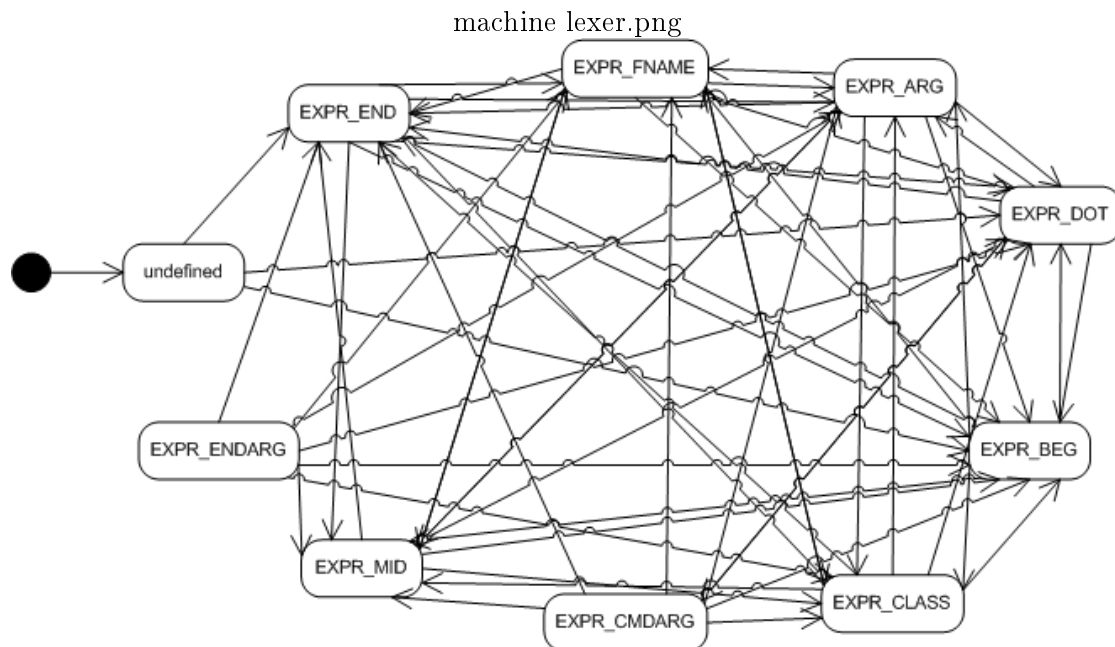


Figure 3.2: State Machine Lexer

3.2 Parser

After having the source code tokenized by the lexer this token stream has to be analyzed and processed. A parser is built for a set of so called production rules. By obeying these rules one or more tokens can be reduced to a new token or node. The newly created tokens can be part of rules themselves. There is a goal token specified to which the whole

token stream should be reduced. If this is possible the syntax is correct. The reduction using the rules can have a side-effect namely executable code can be part of a rule which is executed when the rule fires. In JRuby this capability is used to plant and sprout the abstract syntax tree.

3.2.1 General

In contrast to the lexer the JRuby parser is not handwritten. As mentionable from the lexer name the parser is a yacc generated one. Yacc is the abbreviation for Yet Another C Compiler and is widely spread in the C community. Actually it is not yacc itself to generate the java files. In the case of JRuby Jay is used to generate the parser. But the syntax for the production rules is the same as in yacc. As alluded above the parser can create the syntax tree which is done here in fact. Most production rules create or handle one or several nodes of the tree, that finally is passed back to the caller of the parse method.

DefaultRubyParser Is the main class of this parser. It is generated by Jay out of the DefaultRubyParser.y file. The default ruby parser gets the tokens one by one from the lexer. These are processed by a 3000 lines length switch statement. This an the general construction of this file makes it hard to understand how the parser works exactly. The .y file with the clear production rules is much more human readable and editable. It does not make sense to apply changes in this class directly as these will be overwritten the next time the parser is generated.

ParserSupport Here are the methods that contain common functionality of several production rules. As they do not need to be implemented in every production rule and furthermore will not be processed every time the parser is generated they are externalized in this class. Some of the methods are depending on the state of the parser others are not.

Tokens All the defined Tokens, that can be determined by the lexer, are defined here. Every token gets an integer value assigned which is effectively defined in the DefaultRubyParser. Beside this we consider this interface is misplaced in the parser package, it should be moved to the lexer or all accesses to values of Tokens in the lexer should be done through the Keywords class, which implements the Tokens interface.

YyTables A parser is an extremely complex construction of cases, states and transitions. As it is generated by another program anyway this information is held in huge arrays. Of course this file is generated automatically with the parser.

3.2.3 Call Sequence

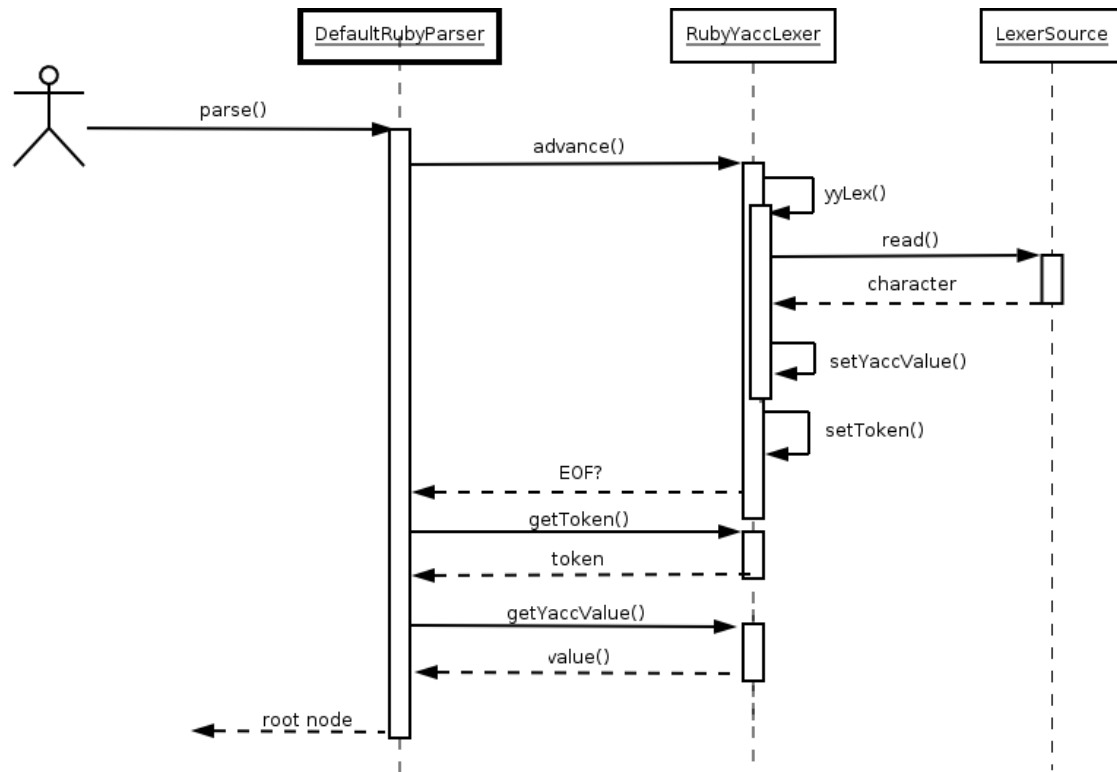


Figure 3.4: Parser Sequence Diagram.

3.2.4 Generation

The parser is not a hand-written class. It is generated out of a rules file. Additionally to the parser the transition tables are generated to the yy-tables file. This generation has to be done every time a rule or anything else in the parser is changed. When editing the production rules this can happen very often. To make this task a bit easier we have written a small script to perform the generation and copy the new files to the correct path:

```
#!/bin/sh

JAY=./jay-1.0.1/jay/jay
PARSER=../repository/rubyrefactoring/\
src/org.jruby/src/org/jruby/parser

cp $PARSER/DefaultRubyParser.y ./DefaultRubyParser.y

$JAY DefaultRubyParser.y < skeleton.java | grep -v "^//t" \
>DefaultRubyParser.out

ruby patch.rb DefaultRubyParser.out > DefaultRubyParser.java

cp DefaultRubyParser.java $PARSER
cp DefaultRubyParser.y $PARSER
cp YyTables.java $PARSER
```

3.3 Abstract Syntax Tree

The Abstract Syntax Tree is a chain of branching nodes. Starting with a node, the root node, a tree representing a whole source code file can be built. In the following we have a simple example of such an AST.

```
class Frog
  SOUND = "Quak"
  def make_sound
    puts SOUND
  end
end
```

As you can see the tree grows big very quickly with even very small blocks of code. Sometimes its a real challenge not to get lost in a branch.

3.4 Positioning Errors

As we began our work on the project, one of the first things we observed while trying to understand how the AST works was that a lot of positions of the nodes were wrong. Usually they were roughly correct, but seldom exactly. Soon we found out that JRuby does not rely on correct positions, so nobody before us cared about them beeing correct.

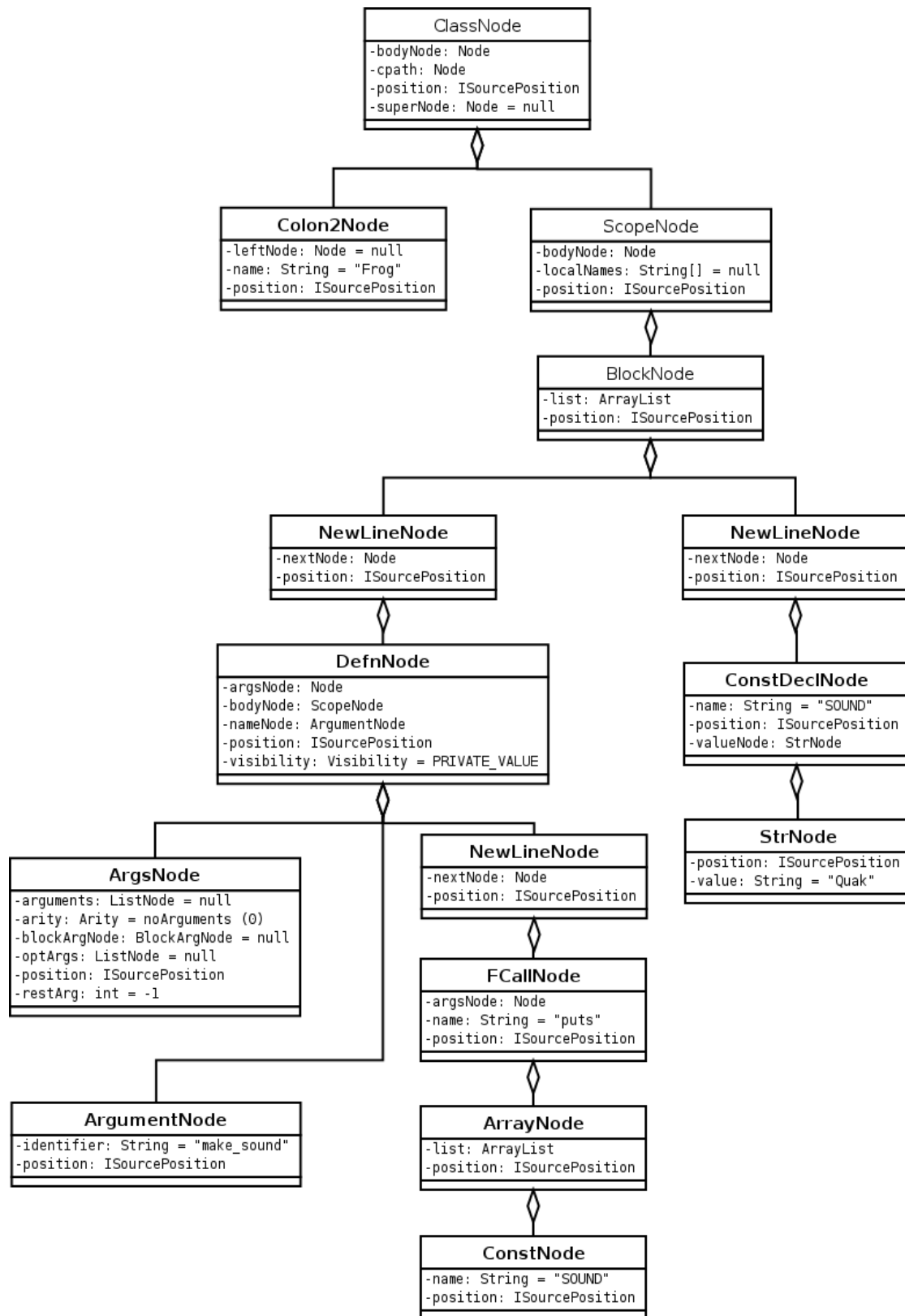


Figure 3.5: AST example.

Usually they were quite easy to fix, it was often enough to merge together multiple nodes whereas often just the position of the first node in a statement was considered. So, for example, we found this code in the nodes, where an argument is attached to another one:

```
public ListNode add(Node node) {
    if (list == null) {
        list = new ArrayList();
    }
    list.add(node);
    return this;
}
```

As you can guess, the sourceposition of \$\$ has to be updated as well, so the code now looks like this:

```
public ListNode add(Node node) {
    if (list == null) {
        list = new ArrayList();
    }
    list.add(node);
    setPosition(
        ParserSupport.union(getPosition(), node.getPosition()));
    return this;
}
```

As we said, most errors resulted of careless assignments and combinations of nodes. Of course we do not blame the JRuby people, since they had no use of the positions. We think that we fixed all positions needed for us to work. There may still remain some wrong positions.

3.5 Errors in the Parser

While working with the parser we had to test and analyze most of the production rules. In some cases we have been able to improve the behavior. Below we have an example of a corrected error in the productions rules. This one was an easy to recognize. As you can read in the comment there were other people doubting this code fragment. By just looking at the code it might be bit awkward to see the mistake. We will give a short explanation to clarify the falsity.

As the commentary says the parser is meant to deal with the second rule item, the \$2. Looking at the rule we can see that this parameter would be the token tFLOAT, which indicates the occurrence of a float value. The tPOW token hints the power operator.

Refactoring Support for the Eclipse Ruby Development Tools

Thus here a node with a mathematical term will be created. Obviously our decimal number has to be part of this new node. Nonetheless \$2 is never accessed in the rule body. Having a closer look we can see that the first parameter is casted into a Double instance. Actually its unlikely that a token representing a minus operator like the first parameter really contains a double value.

```
// ENEB0: Seems like this should be $2
arg | UMINUS_NUM tFLOAT tPOW arg {
  $$ = support.getOperatorCallNode(support.getOperatorCallNode(
    new FloatNode(getPosition($<ISourcePositionHolder>1),
      ((Double) $1.getValue()).doubleValue()), "**", $4),
    "-@");
}
```

We have corrected the code and now it looks like this. Later we have introduced the comment handling but to illustrate the correcting we left it out for this example.

```
arg | UMINUS_NUM tFLOAT tPOW arg {
  Double number = (Double)$2.getValue();
  $$ = support.getOperatorCallNode(
    support.getOperatorCallNode(new FloatNode(getPosition($1),
      number.doubleValue()),
      "**", $4),
    "-@");
}
```

If you wonder what ruby code would fire this rule take at the code in the following.

```
-7 ** 8
```

Generally we have not had to correct many errors. The parser seems to be very stable and represents the Ruby grammar very precisely.

Another example for a fixed part in the lexer, which is not in fact an error but rather more a nasty smell, is the type of the field `yaccValue`. Originally this has been of the type `Object`. We have been very unhappy with this since it caused a lot of casts. So we have tried to figure out whether we could replace it with a more concrete class. In fact we thought about `Token`, that was the type of the most objects assigned to `yaccValue` anyway. When searching the code we came across the following code section:

```
yaccValue = tokenBuffer.toString();
if (IdUtil.isLocal((String)yaccValue) &&
    (((LocalNamesElement) parserSupport.getLocalNames()
     .peek()).isInBlock() &&
     ((BlockNamesElement) parserSupport.getBlockNames()
     .peek()).isDefined((String) yaccValue)) ||
    ((LocalNamesElement) parserSupport.getLocalNames()
     .peek()).isLocalRegistered((String) yaccValue))) {

    lex_state = LexState.EXPR_END;
}

yaccValue = new Token(yaccValue, getPositionMinusOne());
return result;
```

We all agreed that this was definitely no reason to make yaccValue an Object! By introducing a temporary local variable we avoided the abuse of this field.

There was only one further section where yaccValue got a non Token value assigned:

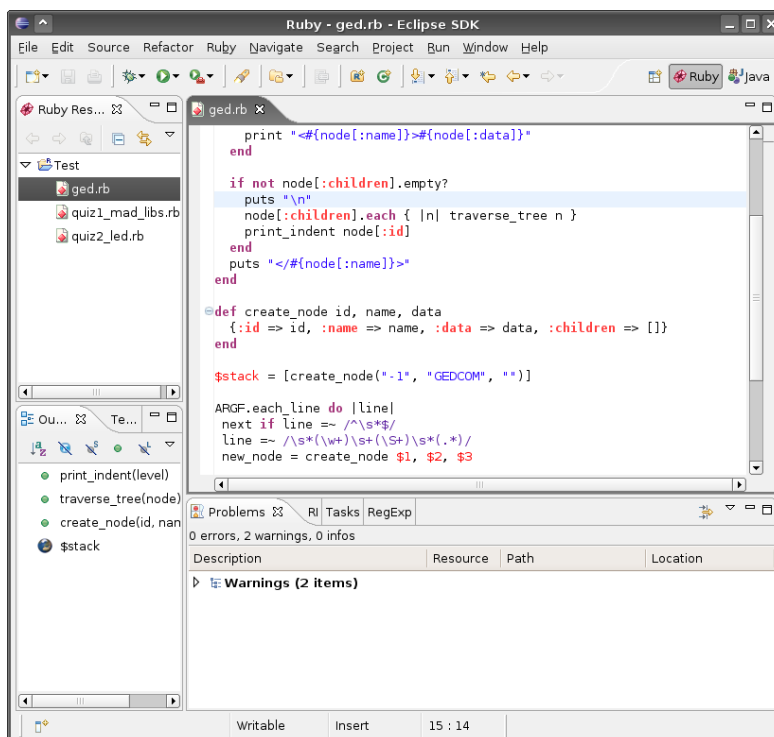
```
case '~': /* $~: match-data */
case '&': /* $&: last match */
case ' ': /* $': string before last match */
case '\': /* $': string after last match */
case '+': /* $+: string matches last paren. */
    yaccValue = new BackRefNode(src.getPosition(), c);
    return Tokens.tBACK_REF;

case '1': case '2': case '3':
case '4': case '5': case '6':
case '7': case '8': case '9':
    tokenBuffer.append('$');
    do {
        tokenBuffer.append(c);
        c = src.read();
    } while (Character.isDigit(c));
    src.unread(c);
    yaccValue = new NthRefNode(src.getPosition(),
        Integer.parseInt(tokenBuffer.substring(1)));
    return Tokens.tNTH_REF;
```

By wrapping these Nodes in Tokens we eliminated the last non-Token assignment. So we could change the type of yaccValue. In the parser we had to adapt the rules for tBACK_REF and tNTH_REF to let it get the nodes from the Token instance.

4 Ruby Development Tools

The Ruby Development Tools are Eclipse plug-ins which provide a fully featured Ruby IDE, including debugging, unit testing, a regular expression tester, an outline view and all other features one would expect from an IDE. It is based on Eclipse and thus can inherit a lot of features, like team support, and good integration into the development cycle.



4.1 JRuby

Since the RDT developers do not always upgrade to the newest JRuby version, we had to change some minor things in RDT to make it work with the JRuby from the repository. There were some annoyances when the selections in the outline did not mark the correct code in the file anymore, which was because we corrected some wrong positions. Fortunately, we only had to change some offsets in one place to get it working again. As soon as our changes to JRuby are accepted, we can start updating the JRuby that RDT uses and then commit our refactorings to the RDT.

5 Comment Handling

At the beginning of our term project we had to decide which AST implementation will be used or if we develop our own. As we selected JRuby it has been clear that the handling of the comments becomes our task.

5.1 General

Here we are. We have a complete parser, that generates the complete AST but unfortunately without comments. When we started with our project we have not had any idea about the yacc syntax at all. At least it has not been that hard to connect the theoretical knowledge with the pages full of rules lying in front of us. With the things we could not figure out ourselves we always found a helping hand asking our supervisor, who is quite familiar with that topic.

After we understood the production rules we still had no clue where we should start to be effectively. To get a hint where we could start placing our comments we contacted those people who obviously knew this best, the JRuby developers. In the very frequently used mailing-list we received useful tips very quickly. There we also came to know that there is a general interest in having comments represented in the AST.

5.2 Placing the Comments

Before we started to implement the comment handling we had to decide how the comments will be inserted into the AST. At that time we have not been aware of all the consequences the different comment handling strategies would come along with. Below three possibilities are listed:

5.2.1 Decorating Nodes

In this model a new class is created that is derived from Node. An instance of this class can contain several comments and the decorated Node object. Thus the decorated Node gets the comments assigned.

This implementation features the following advantages:

- The comments can easily be handled with the existing visitor.
- Comment handling has to be implemented in just one place.
- As nearly everything in the AST is represented by a Node deriving object almost anything can be commented.

and disadvantages:

- Comments can occur in the body of a node, for example a class node could have a comment before the end keyword. This has to be treated with care when rewriting the comment.
- Many nodes can contain other nodes themselves. In several cases these subnodes have a determined type derived from Node. A decorating node cannot derive all these classes too. It would be possible if for every special node an interface was introduced.
- In many cases if a node is accessed there are instanceof queries or directly casts which would throw exceptions or change the behavior when a decorator node is seen.

5.2.2 Separate Comment Nodes

Alternatively there could be a node that represents just the comment. This means that the comment is not associated to another node in the AST. It is quasi standalone.

Advantages:

- The rewriting or generally the visitation of such a node would be very easy.
- Any user of the AST could decide himself what node a comment should belong to.

Disadvantages:

- The advantage that the AST users have to decide themselves what to do with comment is also a duty that might be disturbing.
- Comment nodes could only be placed in nodes that have whole lists of nodes. Else they might occupy a node field that should be free for other nodes.
- There is the same problem concerning casting and throwing exceptions as with the decorator node.

5.2.3 Comments in the Class Node

The third option would result in placing the comments in the Node class. There might be a list with comments and methods to access them.

Advantages:

- Any comment could be placed where it effectively occurred in the source code.
- No new exception causing nodes would have to be created.
- The users of JRuby could take their time to regard the comments without causing unexpected behavior.

Disadvantages:

- The Node class has to be touched.
- Handling of the comments using the visitor might have to be implemented for several classes.
- A comment which cannot be assigned to a node directly might get lost. This issue could be solved by introducing a null object.

As mentioned above at the beginning we have not been aware of all the pros and contras of each design. We especially focused on not changing the Node class. So we decided to introduce the CommentDecoratorNode. After implementing this solution completely we could see quickly the problems of this design. Because we ran out of time we decided to switch to the "Comments in the Class Node" what effectively was easier to realize than solving the problems. We also took this decision as we can be sure that there are no undiscovered side-effects having the comments in the Node object itself.

5.3 Lexer Modifications

The more difficult part of handling comments than storing in the AST is how getting them out of the source code and passing them forward. In this concern the lexer has been the first target for changes. Up to the beginning the lexer just swallowed all comments. So the first step was to introduce new tokens that represent the occurrence of comments. Fortunately they were handled completely in one of the switch cases. Now the lexer can recognize two new tokens:

tTAILCOMMENT This token represents a comment following source code on the same line.

tSOLOCOMMENT Represents a comment being lonely on an own line.

5.4 Comment Handling in the Parser

Having a token stream provided by the lexer we now have to process this information. We hoped to be able to catch all the comments where a newline token could occur. Unfortunately in JRuby the newlines are often swallowed by the lexer and thus the handling of the comments could not be done so easily.

We have not seen any alternative to create and adapt the parsers production rules. This was also the procedure recommended by the people of the mailing list. They recommended us to expand the rule for the primary item with our comment tokens. There we started to create new rules to deal with the new tokens. Soon we recognized that we were not able to find the ultimate production rule to handle all comments in one place. Thus we started to adapt whole production rule blocks.

Refactoring Support for the Eclipse Ruby Development Tools

The main difficulty was to evade shift reduce conflicts. Most of them occurred by trying to handle optional newline nodes. In several cases newline tokens are not returned by the lexer distinctively as in one half of the states they are just jumped in the other half they are returned. We managed to create sets of working rules for some productions, for example the argument handling and the commenting of whole statements, but with every block we wanted to adapt it became harder to avoid the conflicts. Furthermore the number of productions doubled in nearly every set of rules.

We became aware of the fact that it would take too long to implement all the comment handling in this manner, moreover the production rule set increased in size rapidly. As we could not handle every occurrence of a comment token there have been lots of syntax exceptions due to missing rules despite having correct ruby code.

In the following an example of the edited production rules for the statements:

```
stmts
: none
| stmt {
  $$ = support.newline_node($1, getPosition(
    $<ISourcePositionHolder>1, true));
}
| commentedStmt {
  $$ = support.newline_node($1, $1.getPosition());
}
| stmts terms stmt {
  $$ = support.appendToBlock($1, support.newline_node(
    $3, getPosition($<ISourcePositionHolder>1, true)));
}
| stmts terms commentedStmt {
  if($3 instanceof CommentNode) {
    $$ = support.appendCommentToLastStmt($1, $<CommentNode>3);
  } else {
    $$ = support.appendToBlock($1, support.newline_node(
      $3, getPosition($<ISourcePositionHolder>1, true)));
  }
}
| error stmt {
  $$ = $2;
}
\end{lstlinsing}

\begin{lstlisting}
commentedStmt
: soloComment stmt comment {
  CommentDecoratorNode decoratorNode =
    support.commentDecorator_node($2.getPosition(), $2);
  decoratorNode.addCommentBeforeNode($1);
  decoratorNode.addCommentInNode($3);
  $$ = decoratorNode;
}
```

```
}
| stmt comment {
  CommentDecoratorNode decoratorNode =
    support.commentDecorator_node($1.getPosition(), $1);
  decoratorNode.addCommentInNode($2);
  $$ = decoratorNode;
}
| soloComment stmt {
  CommentDecoratorNode decoratorNode =
    support.commentDecorator_node($2.getPosition(), $2);
  decoratorNode.addCommentBeforeNode($1);
  $$ = decoratorNode;
}
| soloComment {
  $$ = $1;
}
```

5.5 Comment Handling in the Lexer

After we got aware of the hitch in the comment handling with production rules we have tried to find an alternative solution. Our supervisor came up with the idea to attach the comment to the tokens. As we could not discover any flaws we revoked the changes in the parser. The output of much work was discarded but we have learned a lot. So we modified the lexer to make it attach the comments to the tokens.

To do that we had to find or in other words create a possibility to look ahead in the lexer. This is necessary because a comment can belong logically to token that comes before the comment itself. That would result in a comment that cannot be assigned to its token as this is possibly already processed by the parser. To avoid this the lexer has to peek if the next token is a comment. If it is in fact one a comment node is created that gets assigned to this token.

There is one big disadvantage about this look-ahead solution. Namely this look-ahead costs calculating time. If the following token is not a comment it has to be discarded. Unfortunately it cannot be stored if it is not one as the parser itself likes to chance to state of the lexer from outside. Thus the following token is predictable for sure. At least such an interference has no effect on the comments but it prevents the buffering of the next element.

We have tried to design this solution to make it easy to implement a reset functionality if the lexer state gets resetted from outside. Unfortunately we ran out of time when we wanted to realize it ourself.

5.5.1 Context Buffering

The lexer contained a lot of fields that represented its context. Most of these variables could change during the lexing process from one token to another. While looking-ahead many of these might get changed and the lexer source jumped over the source code for this token. To make it easier to handle this context we have extracted all the fields into its own class `LexerContext`. This eventually results in many calls in the lexer but is extremely helpful when dealing with several lexer states over time.

The most challenging part was the buffering of the characters read. Despite working with the reader interface, which provides the `mark` and `reset` methods, any `Reader` implementing class does not have to provide the functionality. It is possible that certain `Reader` implementations just throw an exception when trying to `mark` or `reset` the stream. We finally decided to just wrap any reader which does not support the marking with a `BufferedReader`.

The whole story of looking-ahead, storing the context and assigning comments to tokens is handled in the `advance` method of the lexer. This is the method called by the parser to let the lexer read the next token. Up to now its only functionality was to call `yyLex`, that returns the next token, and check whether the token equals `EOF`. To us it seemed to be the right place for our extensions.

5.6 Node Creation

Now the parser received tokens containing `CommentNodes`. To put these comments into the nodes every production rule that handles tokens or creates new nodes has to handle the comments. For easy realization we created a new method in the parser support, that is available to any rule, called `introduceComment`. With a `Node` and an array of `Objects` it returns a `Node` dealing with the comments. The adaption of nearly every rule took a lot of time.

As mentioned before at the beginning we decided to introduce the comments in the AST with decorating nodes. When we recognized the flaws and switched to the comments in the nodes itself solution, due to our design, we had to change only the `introduceComment` method to adapt the behavior totally fitting our requirements.

5.7 Association Rules

A challenge that cannot be solved without analyzing the content of a comment is the association to a part of code. For the comment handling in JRuby we defined the following simple rules:

- A comment standing alone on a line, represented by a `tSOLOCOMMENT` token, belongs to the next statement on the following lines.
- If there is nothing except white-spaces after a stand alone comment it will be associated to the block above.

- Tail comments, represented by a `tTAILCOMMENT` token, are assigned to the item directly before the comment.

Yet not all comments can be stored in a node as not in every production rule, where a comment can occur, a valid node exists. So some comments might get lost, but we are already pouring over solutions for that issue.

5.8 Known Issues

5.8.1 Broken Parser Rule

Now only the production rule of the `when_args` are throwing some unexpected exceptions, when having special arguments. The syntax exception says that there is an empty `when_arg` body even if this is not true. We could not figure out why this exception occurs, but we expect it has its roots in the RDT, as JRuby itself does not complain about empty bodies. And the same code running in JRuby itself runs correctly. But this will need to be fixed. Any other parser rule should work correctly.

5.8.2 Lost Comments

Due to the lack of time we have not been able to test our production rules as thoroughly as we wanted to. So it might be possible that some comments get lost unexpectedly. At least no syntax exception should occur as we have not had to change the reduction parameter sequences to handle the comments.

Yet we have the issue that we cannot comment null nodes. In some cases when a token with a comment occurs in a production rule, where it has to be handled, there is no node where it can be assigned as they are all null. In such a case the comment is lost. A solution for this problem might be the introduction of a null object, a `NullNode`. Because of the lack of time we have not been able to implement this. But it is planned to.

5.9 Change Overview

A lot of changes have been undertaken. In this section you will find a short overview about the changes concerning the comments in JRuby.

5.9.1 Node

The class `Node` has been extended with the functionality to store comments.

- New field: `private ArrayList comments`
- New method: `public void addComment(CommentNode comment)`
- New method: `public void addComments(Collection comments)`
- New method: `public Collection getComments()`

- New method: `public boolean hasComments()`
- New method: `public ISourcePosition getPositionIncludingComments()`

5.9.2 SourcePosition

With the comments having in the nodes, their positions could expand while adding comments. A method for combining positions of nodes was required. We are aware about the union method of the ParserSupport, but its functionality does fit our needs. The SourcePosition seemed to be the right place to implement it.

- New method: `public static ISourcePosition combinePosition(ISourcePosition firstPosition, ISourcePosition secondPosition)`

5.9.3 CommentNode

The CommentNode class has been added to store the comments.

- Field: `private ArrayList comments`
- Constructor: `public CommentNode(ISourcePosition position, String commentValue)`
- Method: `public String getCommentValue()`
- Method: `public void add(CommentNode comment)`
- Method: `private void expandPosition(CommentNode comment)`

5.9.4 RubyYaccLexer

Here we introduced the comment recognition and the conversion to a token. There is now a look-ahead functionality to peek for comments. The type of yaccValue has been changed to Token. The whole context has been externalized into the class LexerContext.

- Changed type of field: `private Token yaccValue`
- New Field: `private boolean lastWasNewLine`
- New Field: `private boolean currentIsNewLine`
- Changed method: `public boolean advance()`
- Changed method: `private int yylex()`

5.9.5 DefaultRubyParser

Now the parser handles commented tokens and passes the comments to the nodes it creates. Most positions have been fixed. The production rule that handles the occurrence of negative decimal values in a power should work correctly now. New Tokens have been introduced. The detailed changes were too extensive to list all the of them here.

5.9.6 ParserSupport

A new method has been defined to combine comments with a node.

- New method: `public Node introduceComment(Node node, Object[] yaccValues)`

6 AST Rewriter

Since we have the comment tokens in the AST and can move nodes around and change its structure, we need a way to recreate the source code from it. That is where the `ASTRewriter` comes into play. It basically takes a node, visits all children and outputs source code for each node. It does not matter whether the root node is a `ClassNode` or a simple `DefnNode`. So we do not have to mess around with searching and replacing text in the refactorings but can leave that to the rewriter.

Additionally we gain the functionality of a complete source re-formatter, who rewrites the whole source in a uniform way. This is not implemented yet, but can be implemented easily.

6.1 General

The `ASTRewriter` is basically just a visitor of the AST. It is implemented like the classical Visitor Pattern described in Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [?]. It basically consists of two elements: Nodes and the `NodeVisitor`.

The Visitor has a visit method for each possible node type, which knows what to do with the specific node, and a general method that calls the accept method on the node with the visitor as argument. The Node contains an accept method which takes the visitor and calls the visitor's visit-method. This may sound confusing, let us have a look at an example:

```
class RewriteVisitor extends Visitor {
  public void visitNode(Node n) {
    n.accept(this);
  }
  public void visitClassNode(ClassNode n) {
    print('class ' + n.getName());
    visitNode(n.getBodyNode());
  }
}

class ClassNode extends Node {
  public void accept(Visitor v) {
    return v.visitClassNode(this);
  }
}
```

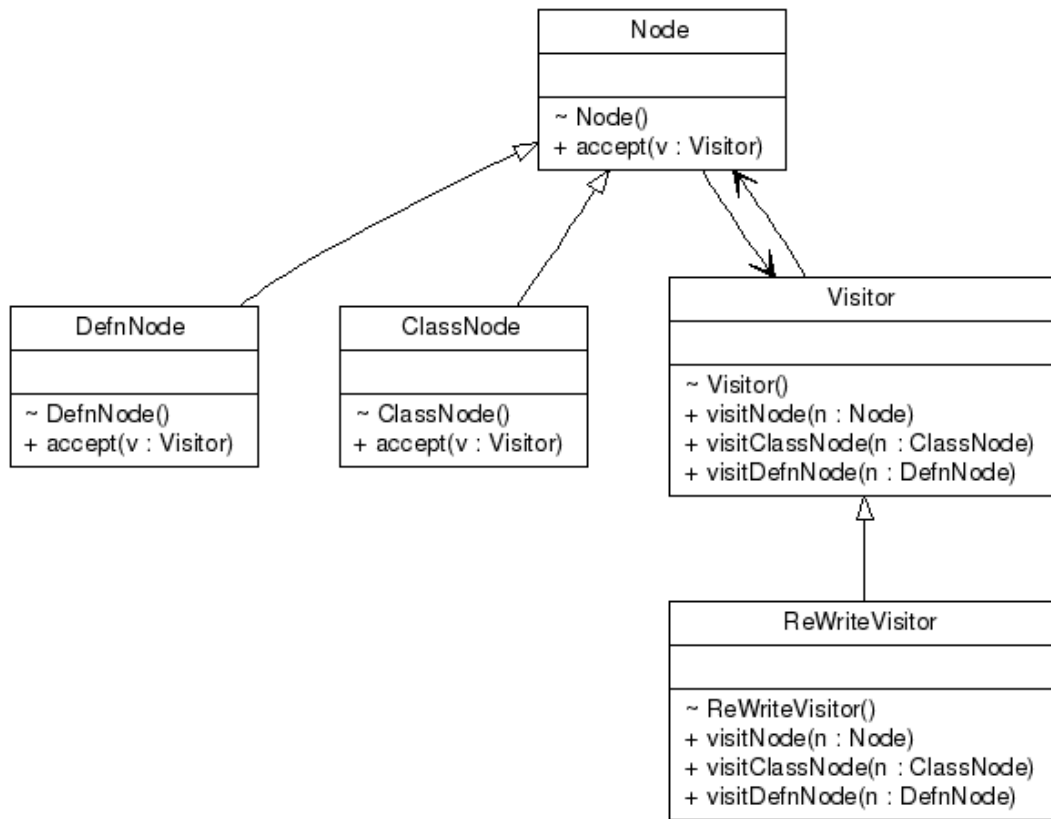


Figure 6.1: Class Diagram of the Visitor Pattern.

The classdiagram can be seen in figure 6.1, a sequence diagram of a typical flow is visualized in figure 6.2.

The probably most important feature of the visitor is the separation of the program flow from the handling of the nodes. In our example, the visitClassNode method does not care of what type its body-node is, it just has to call the visitNode method and can let the targeted node decide which method in the visitor is getting called.

Using the Visitor Pattern, the ReWriter can now traverse the whole abstract syntax tree and write specific sourcecode for each node.

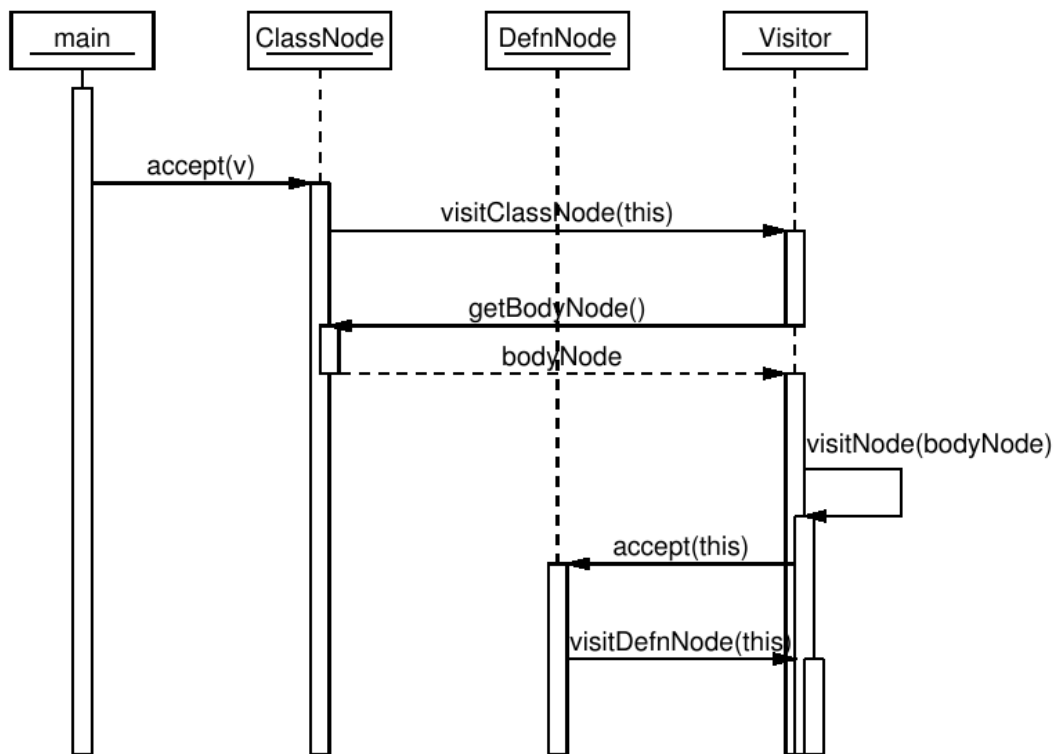


Figure 6.2: Sequence Diagram of the Visitor Pattern.

6.2 Difficulties and Pitfalls

While implementing the ReWriter, we came across a lot of places where it was not so easy to generate the code exactly as the original, most times because there exist multiple ways to express a specific statement or expression in Ruby, for example if-statements, and often there was just not enough information in the AST to determine the correct writing, because he just contains a generic representation. In this chapter, we are going to show the problems we found and how we solved or circumvented them.

6.2.1 Local Variables

Local variables are handled differently in the parser than the other types of variables, because they are just valid in the scope. Their names are held in a list in the ScopeNode they belong to, and if they are used in the scope, the node contains only the index and not the name. So we have to manage this local-names list in the ReWriter and add an entry every time a local variable assignment is visited or a new scope appears.

6.2.2 Parentheses

Parentheses can be omitted in many cases, but it is quite tricky to determine all places where they are needed. For example, parentheses are needed in nested function calls to determine to which call the arguments belong. Take a look at the following code:

```
def method *arg
end
```

and we call it:

```
method 1, method 2, 3
```

Where does the '3' belong to? In this case, we need to clarify and write parentheses:

```
method 1, method(2), 3
# or:
method 1, method(2, 3)
```

Refactoring Support for the Eclipse Ruby Development Tools

Another example shows a 'chained' call:

```
"abcd".split(/c/).first
```

And of course, this does not work without parentheses:

```
"abcd".split /c/ .first
```

In the next sample, we want to pass a hash to a method:

```
do_something {:key => 'value'}
```

As you might expect, this does not work, the correct way would be:

```
do_something({:key => 'value'})
```

We could also write the code like this, but that does not really simplify the problem.

```
do_something :key => 'value'
```

In all these cases, the parser warns us that we are doing something wrong. But that is not always the case, consider the following code:

```
class SuperClass
  def initialize arg = 1
    @arg = arg
  end
  def print_arg; puts @arg; end
end

class Child1 < SuperClass
  def initialize arg
    super()
  end
end

class Child2 < SuperClass
  def initialize arg
    super
  end
end

Child1.new(5).print_arg
> 1
Child2.new(5).print_arg
> 5
```

The only difference are the parentheses, but they are really important, because the super-call in Child2 does automatically pass the initialize-parameters to the super class. So how do we handle this? To determine nested calls, we simply count how deep in the calls we are and print parentheses if needed. In the other cases, we have to check in the particular methods when we write the code.

6.2.3 Strings

Strings in Ruby can be specified in various ways, for us, the important difference is, whether the string should be printed in single ' or double " quotes. We do this by looking at the original source code to find out which separator was used originally.

6.2.4 Operator Precedence

In Ruby, you can write or-operations with the classical || or with `or`. Although they are represented as the same node in the AST, they do not always express the same thing. As described on page 324 in Programming Ruby [?], `or` has lower precedence than ||. If you look at the following irb-session, you can see where the difference matters:

```
irb(main):001:0> b = false or true
=> true
irb(main):002:0> b
=> false
irb(main):003:0> b = false || true
=> true
irb(main):004:0> b
=> true
irb(main):005:0>
```

The same problem applies also to `and` / `&&` and `not` / `!`. In these last two cases, we can define the actual operator by looking at the length of the node, unfortunately with `||` / `or`, that does not work, so we have to look at the original source code.

6.2.5 Here Documents

Here documents are represented as strings in the AST, so if we want to rewrite them as here documents, we have to identify them by looking at the source code and then printing them with the relevant separators. Additionally, we have to change the escaping of the strings, because newlines in here documents don't need to be escaped. Handling here documents gets even more complicated if it does not begin at the end of a line:

```
puts <<EOS, "end"
  start
EOS
```

When we are visiting the here-document string node, we cannot just print it out, because it actually starts on the next line. In the example above, the here-document just contains the string "start". We solved this by storing the string in a variable and letting the newline node handle it.

6.2.6 Blocks

Blocks passed to a method cannot be handled like normal variables; the name of the variable is prefixed with a `&`. If we can not call this method or assign it, the ampersand has to be removed, but if the block-variable is passed to another method, the ampersand has to be retained. Consider the following code:

```
def method &block
  @block = block
  block.call
  another_method &block
end
```

We do this by removing and adding the ampersand depending on the current node: While parsing the arguments of the `DefnNode` and the `ArgsNode`, the ampersand is added and removed at the end of the visit-method.

6.2.7 Comments

Getting comments in the AST does not solve all problems we have with them, we still need to write them back. Since you have a lot of freedom in placing comments nearly everywhere within the source code, we have to mind all those possibilities. All comments that are placed on their own line should be rewritten correctly, but handling comments at the end of the line is a lot trickier. So it might still occur that some comments are lost during refactoring. We are going to improve this in a later version.

6.3 Formatting

Writing syntactically correct code is just one aspect of the `ReWriter`, the generator should also honor the user's formatting as much as possible. The following code,

```
a = check_prerequisites ? "Successful" : "Failed"
```

written as it is represented in the AST, would result in this output:

```
a = if check_prerequisites
  "Successful"
else
  "Failed"
end
```

That is not really acceptable, therefore the rewriter tries to find out what the original format looked like and generates it as close to the original source as possible. Most times, it is sufficient to work with the positions of the elements. In the example above with the if-statement, we can just check if every node is on the same line and thereby assume the short form of the statement. As you can see, we do rely on correct positions from the AST.

What we plan to support in a later version of the AstReWriter is the detection of spaces and parantheses. That should bring the generated code more in line with the users style. We could also implement a user-configurable source formatter for the RDT with fine-grained settings, with for example parentheses, spaces and newlines. Perhaps we could even create a wizard like the new Clean Up wizard Eclipse 3.2 has.

7 Refactoring Plug-In

This chapter gives a detailed description of the implemented refactoring plug-in. This contains a description of the used extension points, the fundamental design for the implemented refactorings and the description of the code generators and finally the refactorings themselves.

This chapter contains only the description for the created refactorings and code generators. The automated tests for those code components are described in the next chapter.

7.1 Used Extension Points

When an Eclipse plug-in is developed you can use extension points provided by other plug-ins to integrate your plug-in into the Eclipse environment. The refactoring plug-in uses an extension point to add its functionality to the popup menu of the RDT editor and another one to add two menu entries to the Eclipse menu bar. To use extension points of other plug-ins, you add entries to the plugin.xml file of your own plug-in. An example for this will follow in the two next sections where the used extension points are described.

7.1.1 RDT Popup Menu Extension Point

The Ruby Development Tools provide several plug-ins. The one used to extend the popup menu of the Ruby editor is the Ruby Development Tools UI plug-in. It provides a really simple but powerful access point, the editorPopupExtender. In the plugin.xml file, you define a class that extends org.eclipse.ui.action.ActionGroup. Here an example:

```
<plugin>
  <extension point="org.rubypeople.rdt.ui.editorPopupExtender">
    <rubyEditorPopupMenuExtension
      class="mypackage.MyActionGroup"/>
    </extension>
  </plugin>
```

Refactoring Support for the Eclipse Ruby Development Tools

The defined class overrides the ActionGroup method fillContextMenu. The method takes an IMenuManager as argument. In the fillContextMenu method you can add menu entries and submenus to the menu. A demonstration follows here:

```
public class RefactoringActionGroup extends ActionGroup {
    public void fillContextMenu(IMenuManager menu) {
        MenuManager submenu = new MenuManager("MySubMenu");
        menu.add(submenu);
        submenu.add(getAction("My First Menu Entry"));
        submenu.add(getAction("My Second Menu Entry"));
    }
    private Action getAction(String entryName) {
        Action action = new Action(){
            public void run() {
                System.out.println(", menu entry was clicked");
            }
        };
        action.setText(entryName)
        return action;
    }
}
```

The good thing about the popupMenuExtender is, that you only need to configure one use of the extension point in plugin.xml and add dynamically as many menu entries as you like.

7.1.2 Eclipse Menu Bar Extension Point

The refactorings and code generators should not only be accessible through the popup menu of the Ruby editor, but also via the Eclipse menu bar. Our refactoring plug-in adds two new menus to the menu bar. These are the refactor menu and the source menu, where the source menu contains the code generators. In the plugin.xml you need to add the extension point ActionSet. This ActionSet contains an entry for each menu, menu separator and menu entry you create.

```
<plugin>
  <extension point="org.eclipse.ui.actionSets">
    <actionSet id="setId" label="MyActionSet" visible="true">
      <menu id="menuId" label="My Menu" path="edit">
        <separator name="menuSeparator0"/>
      </menu>
      <action
        class="mypackage.MyActionClass0"
        id="entryId0" label="First Menu Entry"
        menubarPath="menuId/menuSeparator0"/>
      <action
        class="mypackage.MyActionClass1"
        id="entryId1" label="Second Menu Entry"
        menubarPath="menuId/menuSeparator0"/>
    </actionSet>
  </extension>
</plugin>
```

7.2 Fundamental Design of the Plugin

To implement a refactoring for Ruby, there are some components that are needed in several or all the refactorings. Those components are described in the following sections.

7.2.1 Refactoring Model

In the following diagram you see how our model is set up and which components use or require others and how they are related. Some of the more important components are described in the following sections.

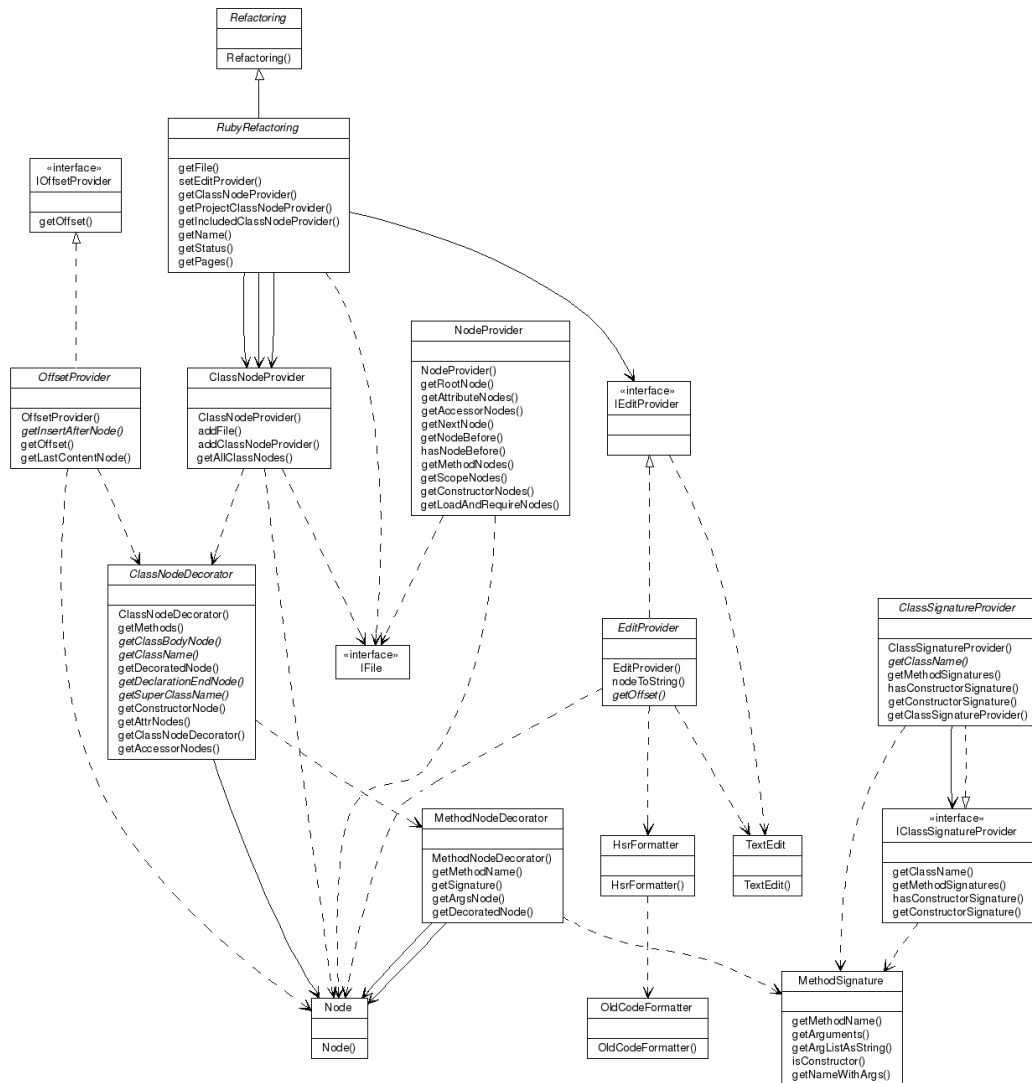


Figure 7.1: Fundamental Design

7.2.2 Node Provider

The Ruby refactorings are based on JRuby. JRuby provides us with an abstract syntax tree that represents a Ruby source file and can be used to get the information we need. The AST consists of different nodes. To get information, for example required AST nodes out of Ruby source documents, we created a helper class called NodeProvider. This class helps to filter subnodes out of nodes and other things like that.

7.2.3 Node Decorators

Some of the nodes are important for our refactorings. This are for example the ClassNode and the DefnNode (method node). The nodes do not really provide the functionality we needed. Because of this, we created decorator classes for those nodes. From our view this is the best solution because we do not have to mess up with the JRuby code by adding a lot of functionality to it, but we still get components that let us do our refactoring work. In this diagram you can see the created node decorators.

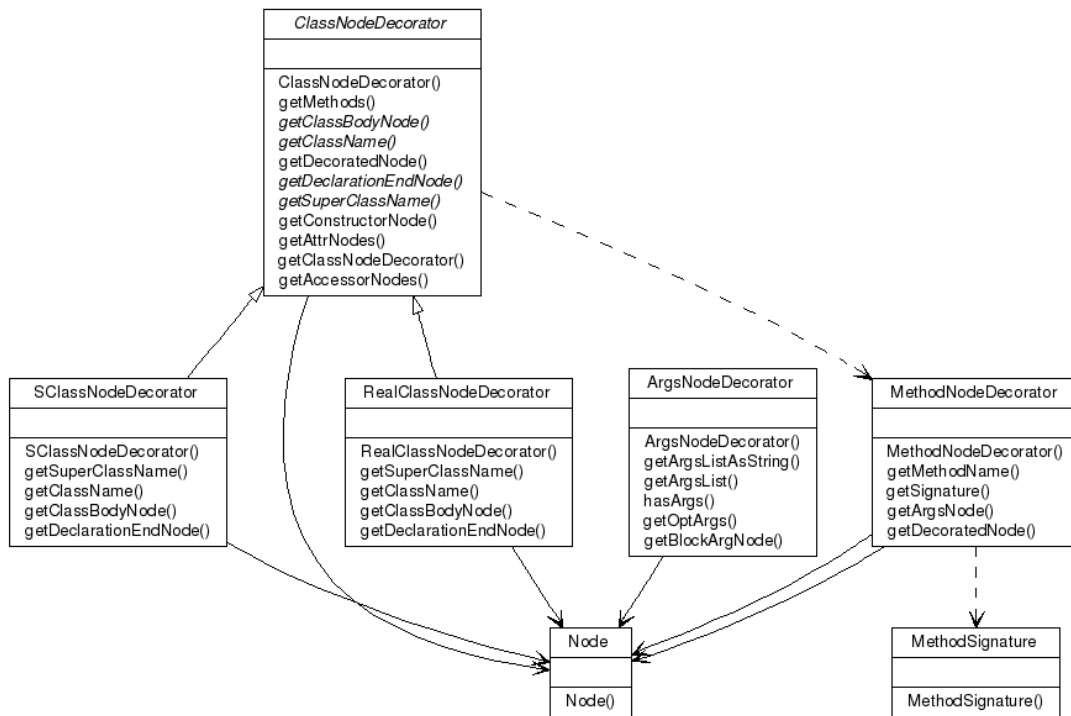


Figure 7.2: Node Decorators Diagram

7.2.4 Classnode Providers

Some of the refactorings are based on the object oriented programming model. This means they work a lot with classes. In Ruby several classes might exist in one file or one

class might be distributed over several files. The class node providers are designed to provide an easy way to find a specific class. So they get fed with Ruby source files and provide functionality to retrieve the class nodes those files contain.

There is a basic and two extended class node providers.

- **ClassNodeProvider**
The basic provider is the `ClassNodeProvider`. It takes single files, parses them and adds the contained class nodes to its content.
- **IncludedClassNodeProvider**
The `IncludedClassNodeProvider` also provides access to the class nodes that are included into the Ruby file with "require" or "load" statements.
- **ProjectClassNodeProvider**
The `ProjectClassNodeProvider` provides access to all the class nodes in the active Ruby project.

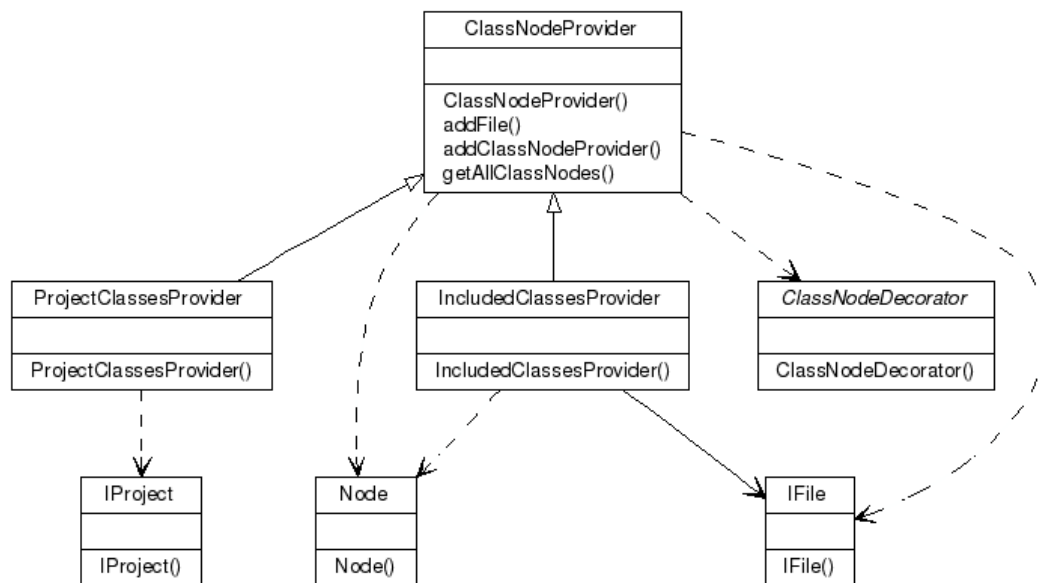


Figure 7.3: Class Node Providers Diagram

7.2.5 Edit Providers

Each refactoring contains an edit provider. The edit provider is used to retrieve the text- or multitext-edit that applies the result of a refactoring to a document. So the real logic of a Ruby refactoring always hides itself behind an edit provider. You can see this in the following diagram:

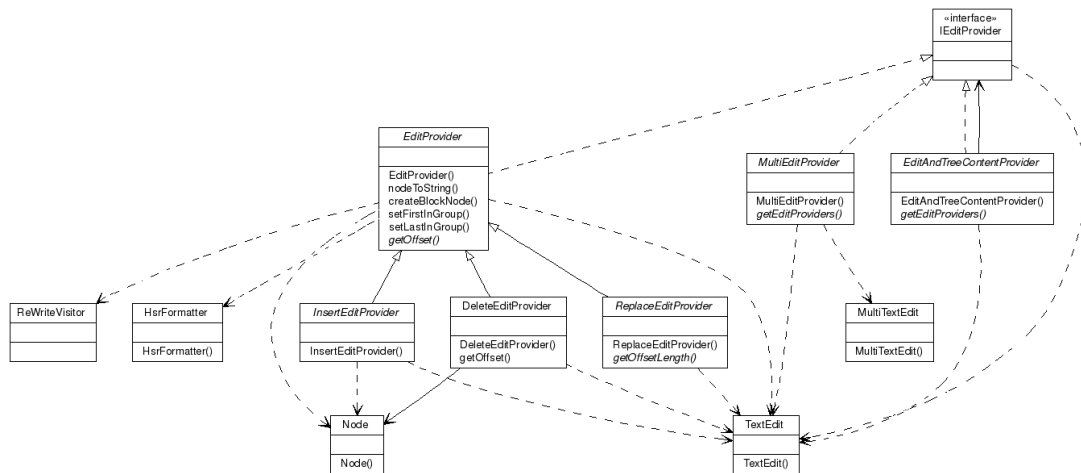


Figure 7.4: Edit Providers Diagram

7.2.6 Signature Providers

A signature provider is a component that provides a refactoring with the signature of ruby methods. The source of a signature provider can on the one side be a class node, and on the other side be a RubyClass. These RubyClasses are needed to provide method signatures of built-in Ruby classes. Built-in classes are not implemented in Ruby, but in c. So they cannot be parsed like normal ruby files.

7.2.7 Offset Providers

Offset providers are designed to decide, on which position in the document the generated text from a Ruby refactoring should be applied.

7.2.8 HSRFormatter

RDT contains a class called OldCodeFormatter that has the job to format unformatted Ruby code. You can give it a string containing Ruby code and the OldCodeFormatter formats it. The formatter is not in all scenarios as smart as we would like it to be. The problem occurs when only a part of a file needs to be formatted. In the case of a refactoring this is the text that will be inserted. Here I give you an example that shows the problem. Unformatted Ruby code:

```
1 class my_class
2   def method0
3     @field = 17
4   end
5 end
```

We want only the code on line 2, 3 and 4 to be formatted. The result of the code that will be formatted should look like this:

```
  def method0
    @field=17
  end
```

Now look at the result the OldCodeFormatter gives us. Pay attention on the indentation. That's where the mistake happens:

```
def method0
  @field=17
end
```

We can see that the indentation of the code is not what we would like to have. The OldCodeFormatter does not know or does not care what is around the code he formats.

This is the reason why we wrote an extension called HSRFormatter that is able to format only a part of a document, but does this under the consideration of the enclosing documents indentation.

7.3 Implemented Code Generators

7.3.1 Generate Accessors

The generate accessors code generator gives the user a tree to check the accessors he would like to create. On the first level of the tree he are the classes contained in the source file, on the second level the attributes and on the third level the accessors he is able to generate. Besides that he can decide if he wants to generate simple accessors or method accessors.

Demonstration

This image shows the tree with the list of accessors and the choice for an accessor type.

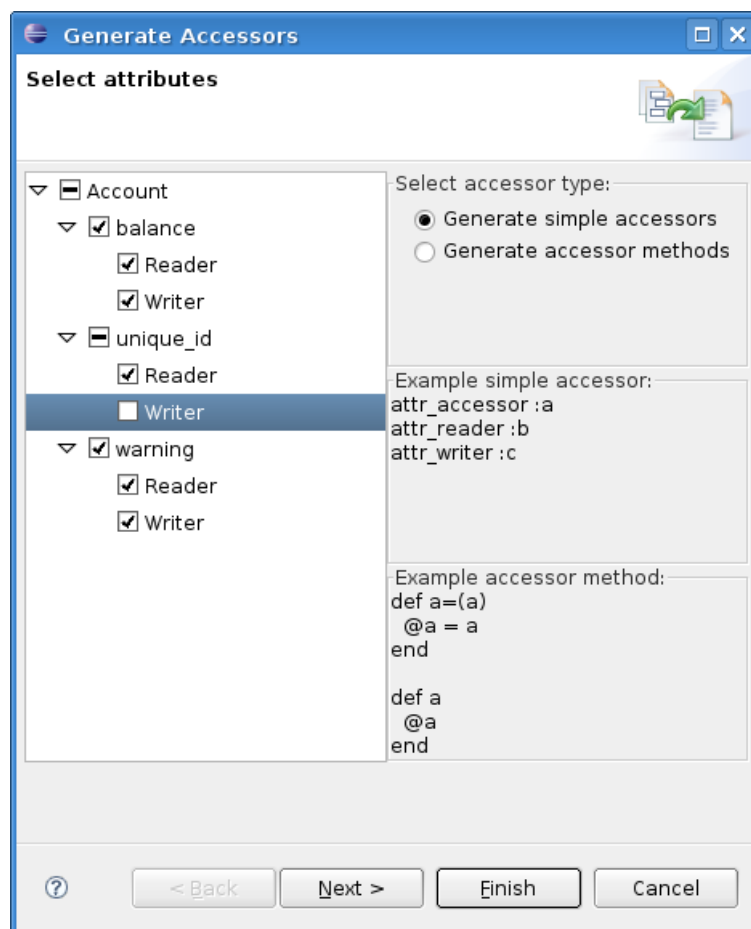


Figure 7.5: Generate Accessors Refactoring Wizard

Refactoring Support for the Eclipse Ruby Development Tools

Here you can see the result of the refactoring for the simple accessor type.

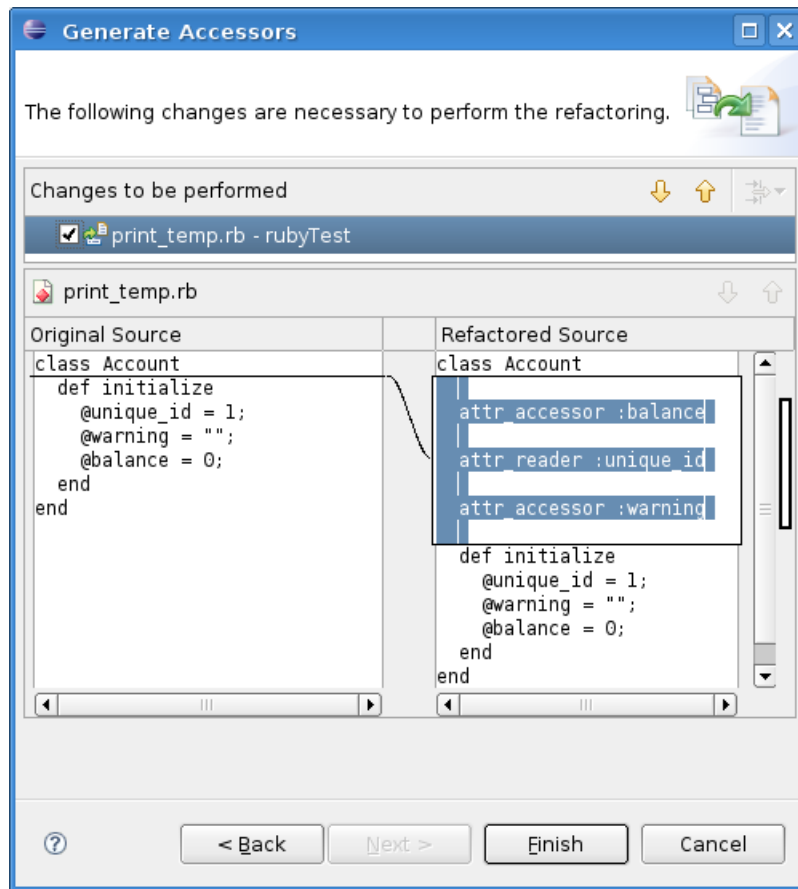


Figure 7.6: Generate Accessors Refactoring Wizard Simple Accessors

Refactoring Support for the Eclipse Ruby Development Tools

Here you can see the result of the refactoring for the method accessor type.

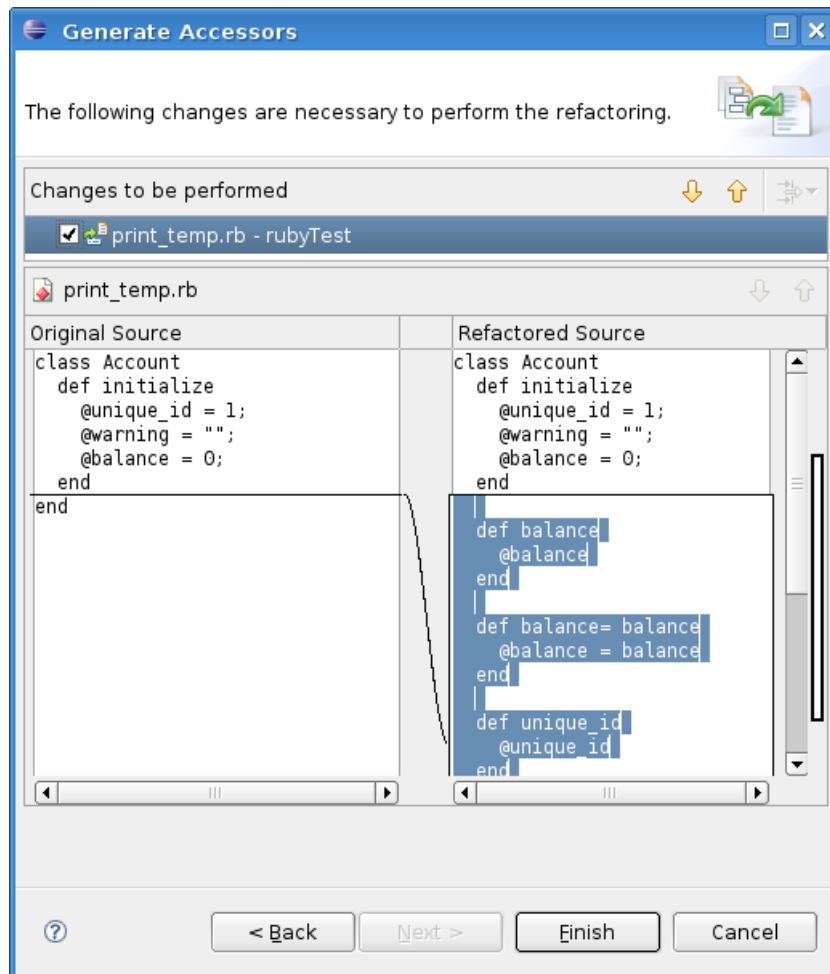


Figure 7.7: Generate Accessors Refactoring Wizard Method Accessors

Procedure

First we create a `ClassNodeProvider` and add our source file to its content. For each class in it, a `TreeClass` object is created that knows the `ClassNodeDecorator` it represents. This `TreeClass` creates `TreeAttributes` for each contained attribute in the `ClassNodeDecorator`. Each `TreeAttribute` contains a reader and writer child. If a tree entry (`TreeClass` or `TreeAttribute`) contains no children, those entries will not be shown in the tree. This might happen if either a Ruby class has no attributes, or all the accessors that could be generated for an attribute already exists in the source.

When the user clicks on the button "next", for each checked attribute a `GeneratedAccessor` object is created. `GeneratedAccessors` are edit providers. The text edit these providers provide will then be used to get shown and after that applied to the code in the document.

Class Diagram

To get an overview over the class model of the generate accessor code generator take a look at the diagram [7.8](#).

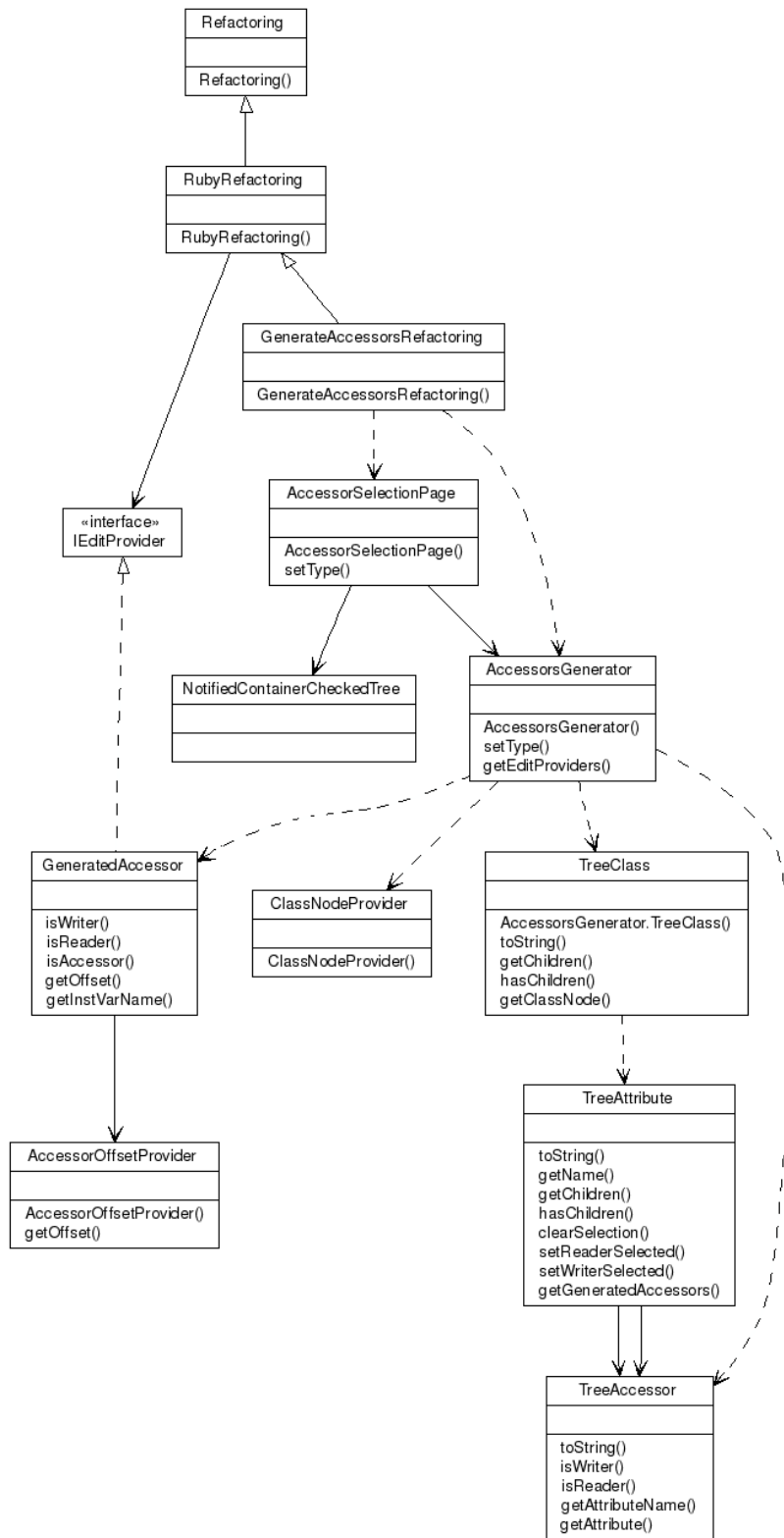


Figure 7.8: Generate Accessors Diagram

7.3.2 Generate Constructor Using Fields

This code generator provides the user with a tree that contains all the classes and below them the attributes of those classes. He might check a class and several of the class's attributes. With this selection a constructor will be generated.

Demonstration

Here you can see the user interface.

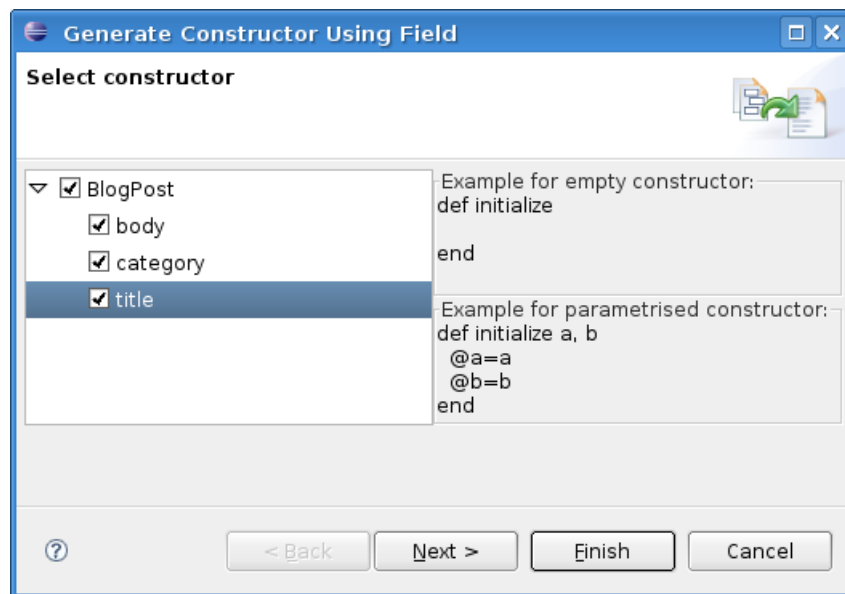


Figure 7.9: Generate Constructor Refactoring Wizard

Refactoring Support for the Eclipse Ruby Development Tools

Here you can see the results of the refactoring that will be applied to the document.

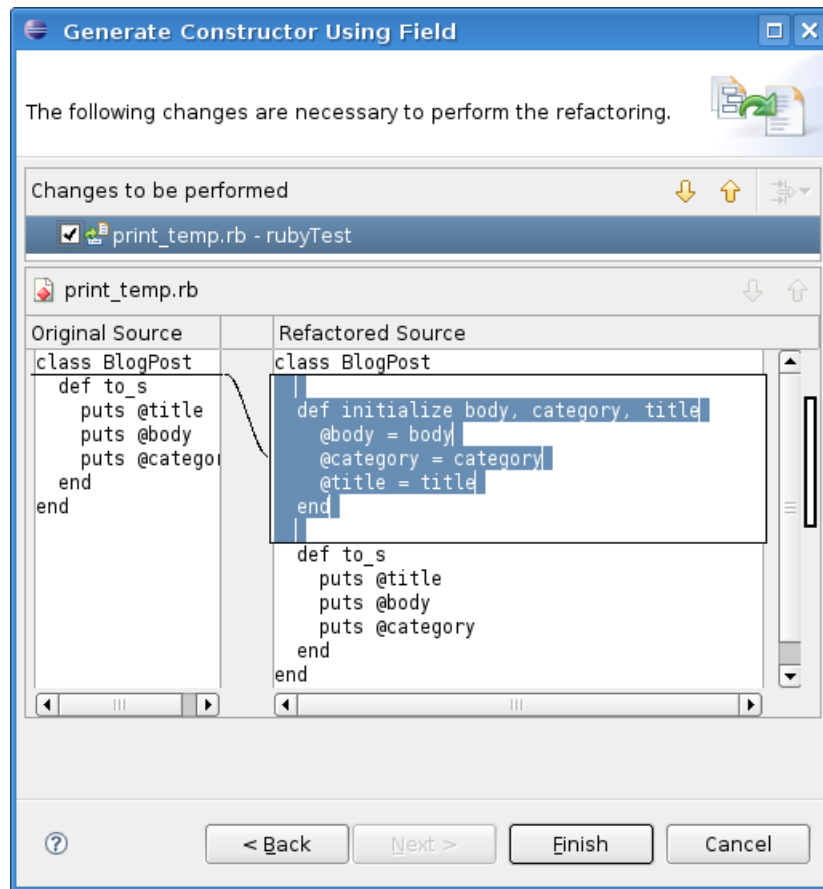


Figure 7.10: Generate Constructor Refactoring Wizard Result

Procedure

With the source file given by the Ruby editor, a `ClassNodeProvider` is created. The `ClassNodeDecorators` it provides are added to the tree. The `ClassNodeDecorator` provides all its attributes which are shown in the tree under its class. After the user made his selection, for each class that got checked, a `GeneratedConstructor` object is instantiated. This again is an edit provider that provides a text edit which can be applied to the document.

Class Diagram

To get an overview over the class model of the Create Constructor Using Fields code generator take a look at the diagram 7.11.

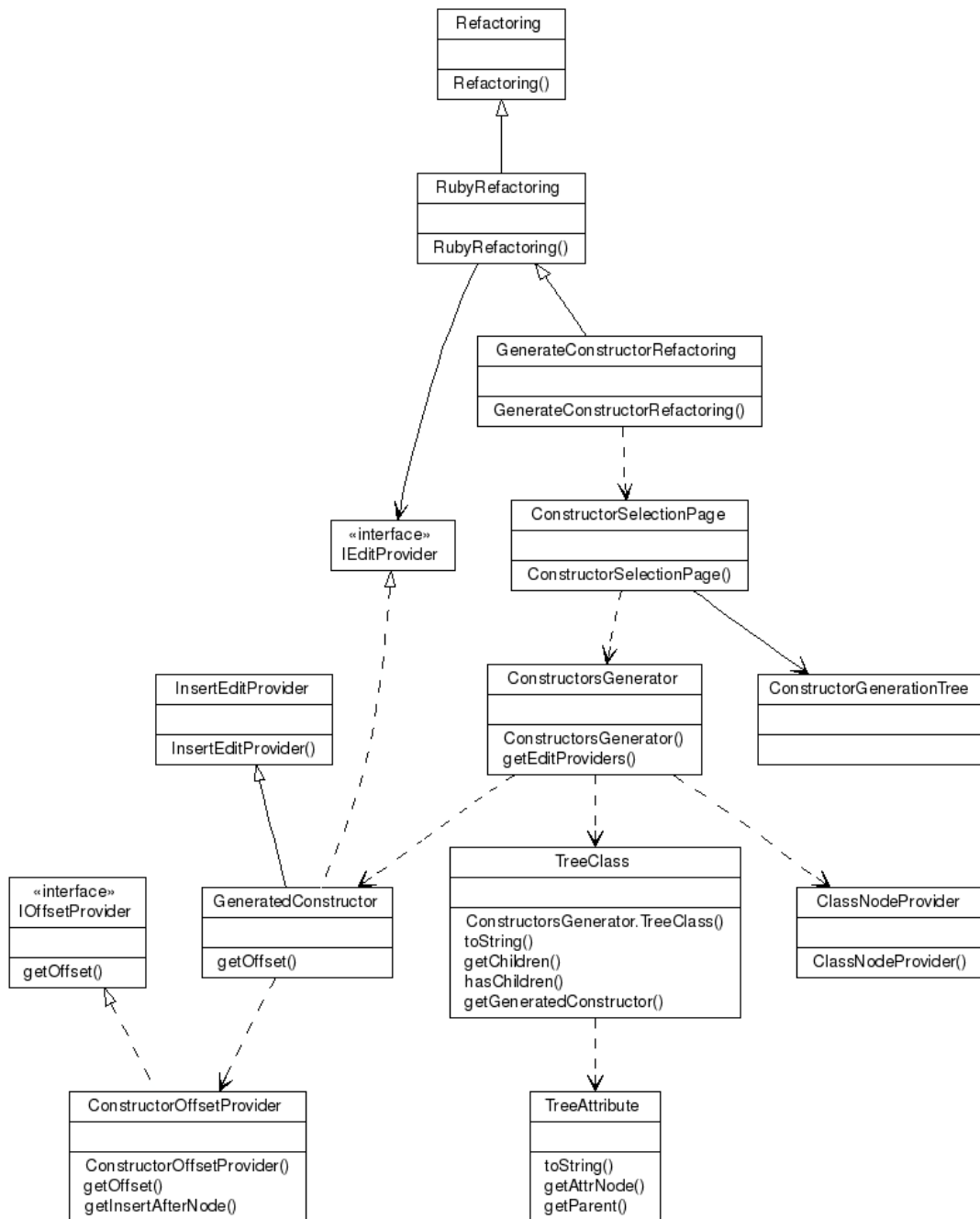


Figure 7.11: Generate Constructor Diagram

7.3.3 Override Method

The Override Method code generator allows the user to override methods from its super class. To do this, a tree with all the classes that have a super class, which are not Ruby built-in classes, are displayed. On the second level of the tree the methods provided by the super class are displayed. The user can now select which of the super class's methods he would like to override.

Demonstration

Here you can see the user interface that will be shown in the first step.

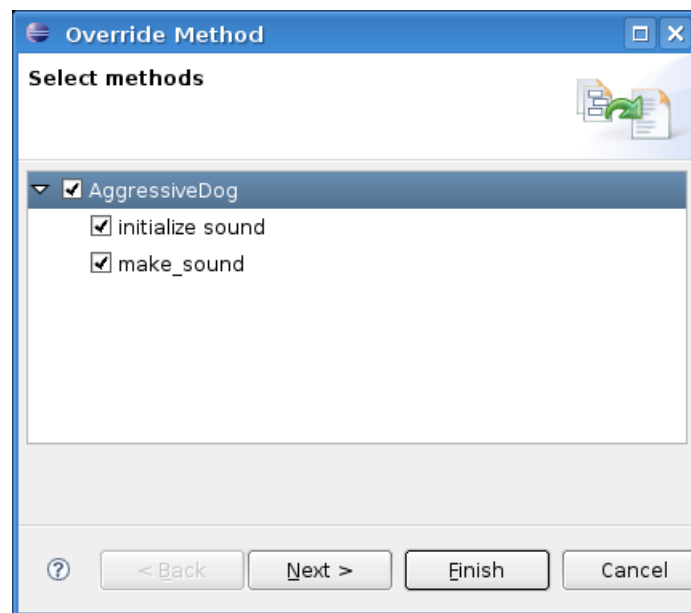


Figure 7.12: Override Method Refactoring Wizard

Here you can see the results of the refactoring that will be applied to the document.

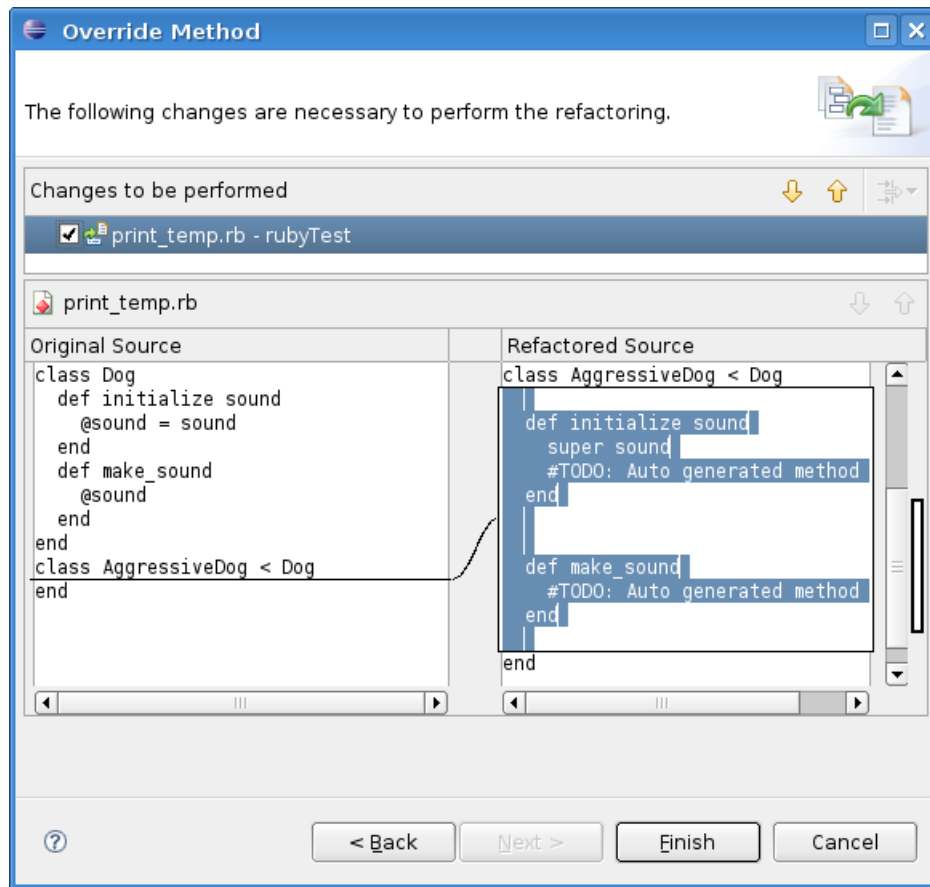


Figure 7.13: Override Method Refactoring Wizard Result

Procedure

To fill the tree with its content, two `ClassNodeProviders` are necessary. One provides all the classes that are in the source file of the editor, the other one is a `IncludedClassNodeProvider`. It also provides all the classes that are imported into that file with "require" or "load" instructions. To show the tree's content, we need to access the super class's methods. Because the super class is not necessarily implemented in the source file itself, this second `ClassNodeProvider` is required.

Now, the classes which have a super class that has methods to override, are displayed to the user. He selects the methods to be overridden. For each selected method an `OverridenMethod` is created, which is an edit provider that provides a text edit to perform the edit on the document.

Class Diagram

To get an overview over the class model of the override method code generator take a look at the diagram 7.14.

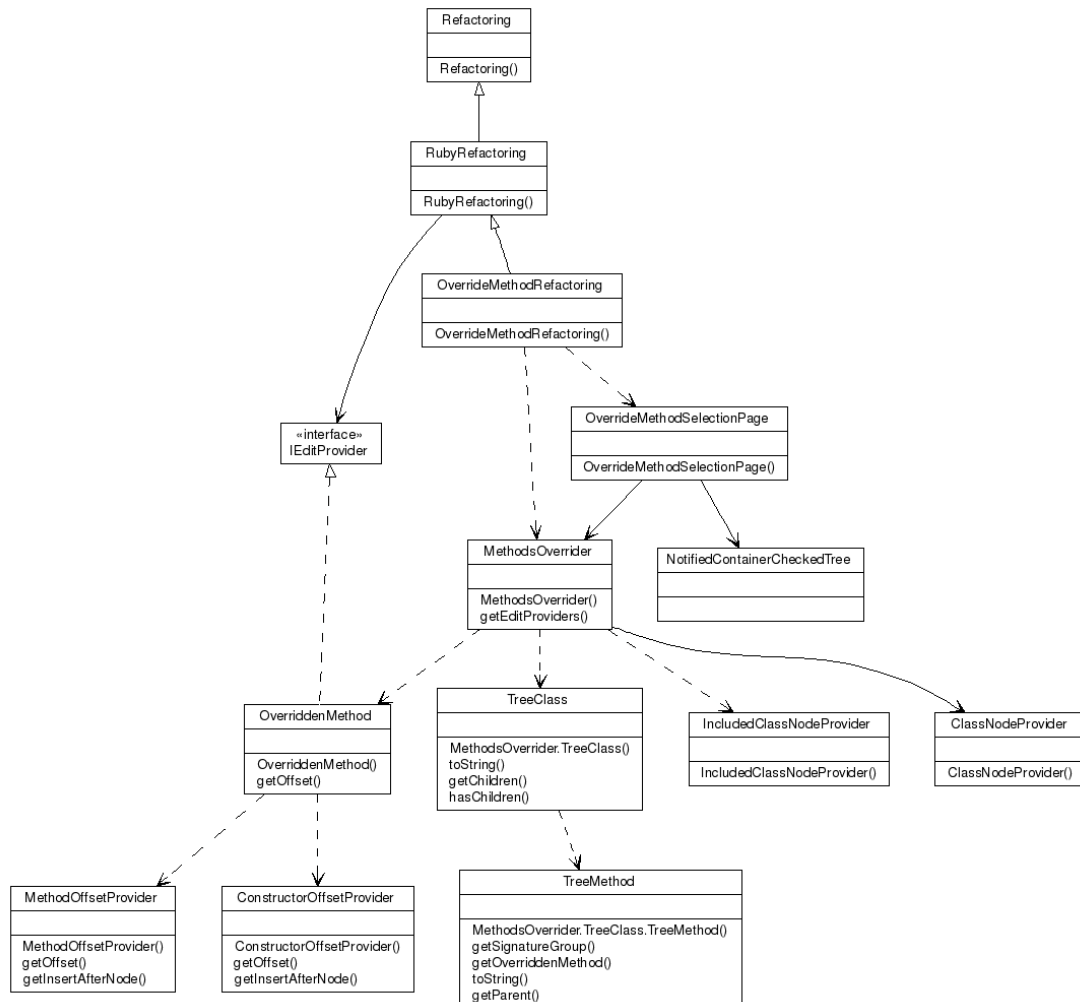


Figure 7.14: Override Method Diagram

Extensions and Ideas

The Override Method refactoring supports only the possibility to override methods of non built-in Ruby classes. This is because the arity of the methods of built-in classes can not be evaluated properly, it works only on constructors with a little hack. If a way could be found to evaluate the arity, the overriding of built-in classes could be allowed, but we do not think this to be a big problem, because you rarely want to inherit from built-in classes.

7.4 Implemented Refactorings

7.4.1 Rename Local Variable

This refactoring provides the user with a list of the local variables that exists in the code block that contains the carret. A code block is normally a method body. He can select one of those variables and type a new name below. At the moment, only variable int the method's scope are supported, variables in iterators will be implemented later.

Demonstration

Here you can see the user interface that will be shown to the user.

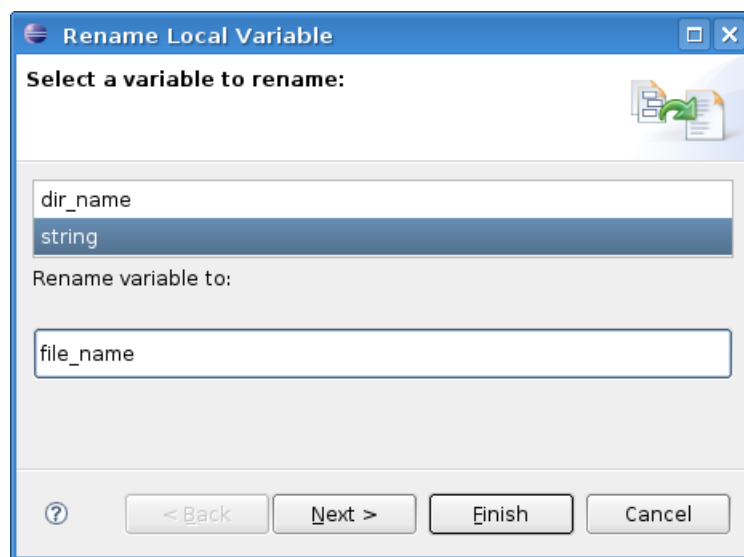


Figure 7.15: Rename Local Variable Refactoring Wizard

Here you can see the results of the refactoring that will be applied to the document.

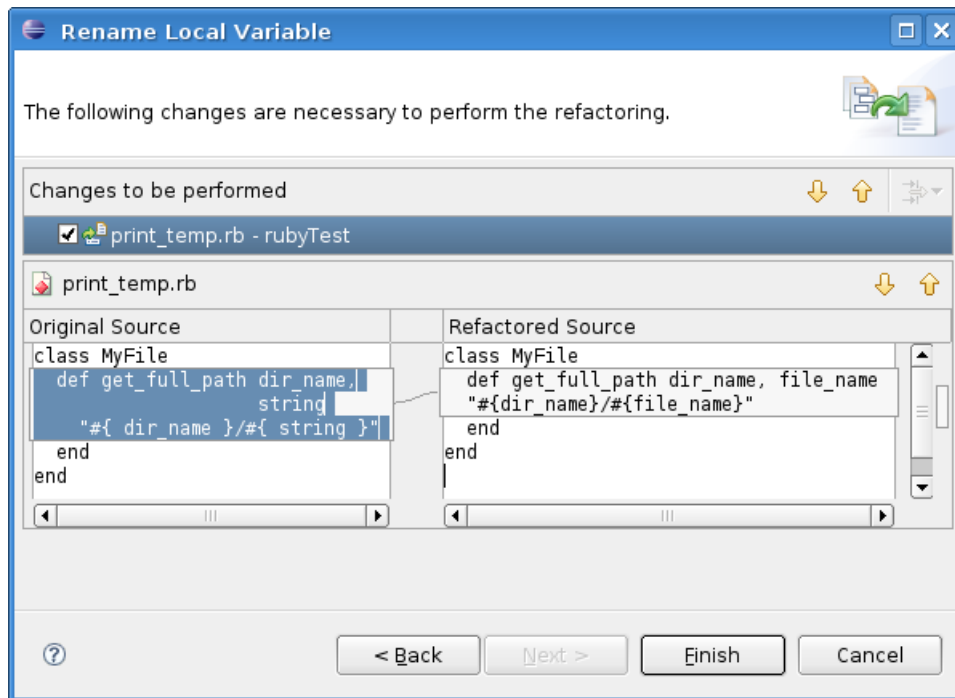


Figure 7.16: Rename Local Variable Refactoring Wizard Result

Procedure

The current document and the caret position are retrieved. The block in which the caret is positioned is evaluated. Now all the local variables in that block are shown in the list of the refactoring. After the user selected a variable out of the list, he can set a new name and apply the refactoring to the document by finishing the wizard.

Class Diagram

To get an overview over the class model of the rename local variable refactoring take a look at the diagram [7.17](#).

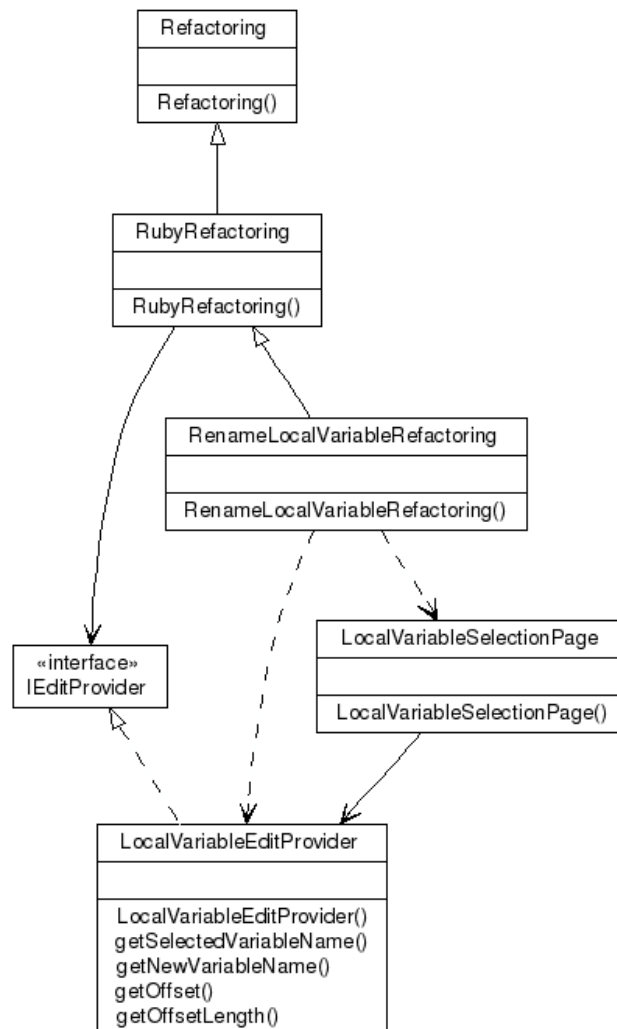


Figure 7.17: Rename Local Variable Diagram

Extensions and Ideas

The rename local variable refactoring might be extended to a rename refactoring that can rename not only local variables but also fields, methods, classes and so on. There is no guaranty that this is possible in all scenarios because Ruby allows you to put parts of the same class, that will become combined depending on how they are included, anywhere in a project's files.

7.4.2 Push Down Method

The push down method refactoring shows a tree of all the classes in the source document, that are super classes to other classes somewhere in the active Ruby project. The user can check methods that are implemented inside those super classes and have the refactoring push them down to the classes that extends the super class.

Demonstration

Here you can see the user interface that will be shown to the user.

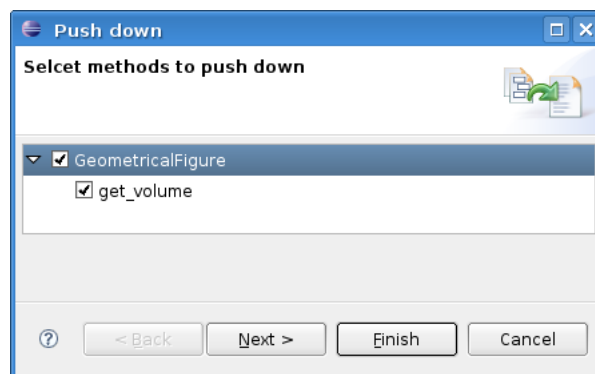


Figure 7.18: Push Down Method Refactoring Wizard

Refactoring Support for the Eclipse Ruby Development Tools

Here you can see the results of the refactoring that will be applied to the document.

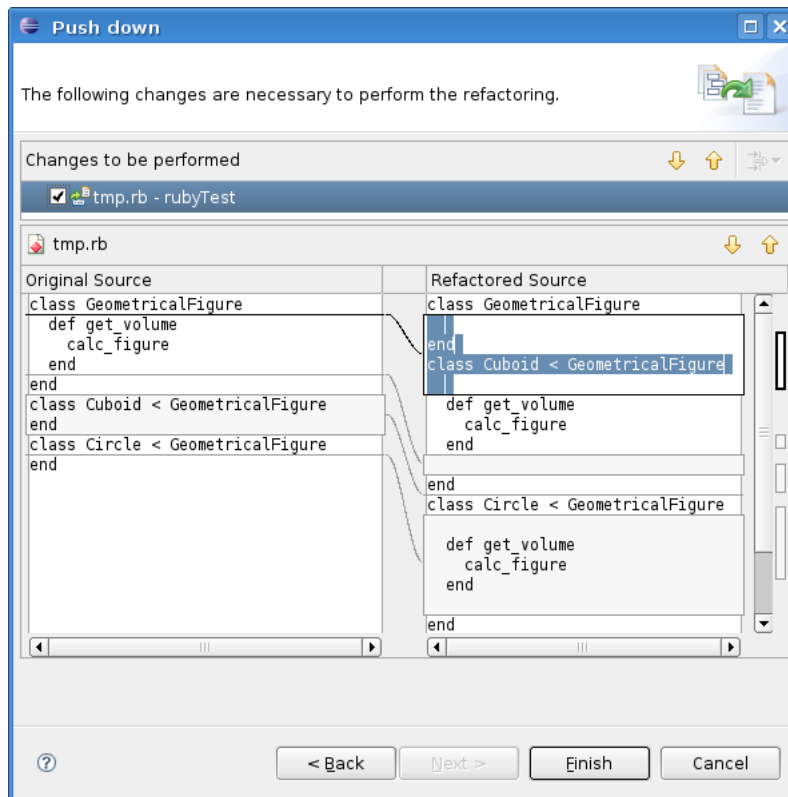


Figure 7.19: Push Down Method Refactoring Wizard Result

Procedure

The super classes are provided over a normal `ClassNodeProvider` containing the source document. To find all the classes that extend one of those super classes, a `Project-ClassNodeProvider` is created. The user can check methods that are implemented in the super classes (shown in the tree below the super class). When he applies the refactoring to the document, text edits are. If a Ruby class that extends a super class does not exist in the refactored document, the class containing the overridden methods gets created. Like this, the overridden method gets applied to that class, even if the main class implementation is somewhere else in the project.

Class Diagram

To get an overview over the class model of the push down method refactoring take a look at the diagram 7.20.

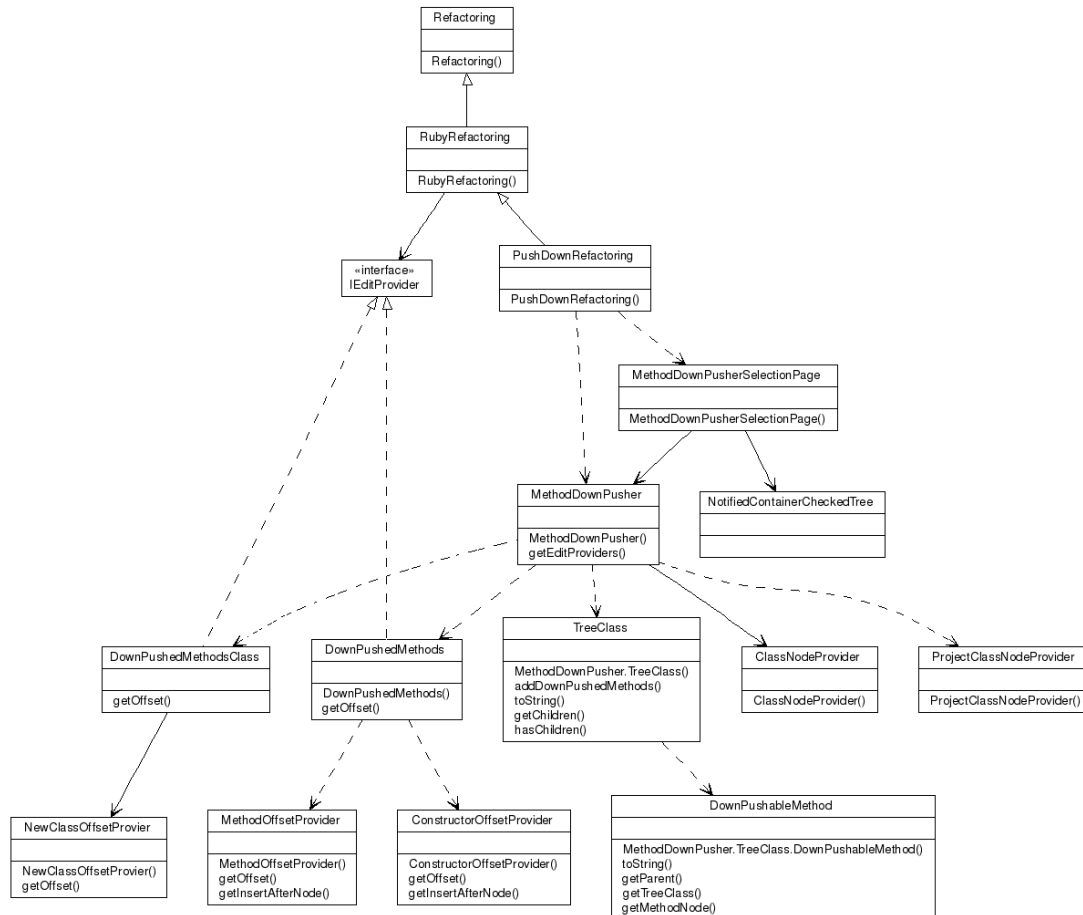


Figure 7.20: Push Down Method Diagram

Extensions and Ideas

This refactoring might be extended that it allows to push down not only methods, but also fields, accessors and so on. In the momentary state, the refactoring pushes the selected method into all its child classes. Another extension might be to let the user select the child classes into which the method will be pushed down.

7.5 Eclipse

The following sections describe how Eclipse works and how plug-ins are contributed to it. If you like to have more detailed information, we recommend the book *Contributing to Eclipse* [?].

7.5.1 The Plug-In System

The whole Eclipse system is based on plug-ins. Everything you see is a plug-in. Always there are the core plug-ins. Those provide some basic functionality in the form of extension points. Other plug-ins can use those extension points, provide other functionality themselves and make this functionality accessible with their own extension points.

The plug-in system is one of the reasons for the success of Eclipse. It provides a dynamic environment to implement whatever functionality a developer can imagine. This system does not limit a developer. You can make Eclipse whatever you want. A word processing software, a development environment for whatever programming language you know, an e-mail client.

The more plug-ins you have, the larger Eclipse grows. But it is not getting slower when you start Eclipse. Even if your Eclipse contains hundreds of plug-ins, Eclipse will only load the plug-ins that are required right now. If another plug-in gets required, Eclipse loads it at that point and not before. This behavior is described with an Eclipse rule, the Lazy Loading Rule. See the section Lazy Loading Rule [7.5.3](#) for more details.

7.5.2 Refactoring Support

Eclipse already provides a basic infrastructure to apply refactorings to documents. This is mainly a model that provides a user interface wizard. This wizard is made up of several wizard pages. First, several refactoring specific pages which are added by the developer who uses the Eclipse refactoring support. The last page gives an overview over the changes that will be made when applied to the document. This page will show you the affected documents before and after the refactoring.

Basic Example

Here you see the code that creates and runs a refactoring.

```
Refactoring refactoring = new XYRefactoring();
RubyRefactoringWizard wizard =
    new RefactoringWizard(refactoring,
                          WIZARD_BASED_USER_INTERFACE);
UserInputWizardPage page = new XYUserInputWizardPage();
wizard.addPage(page);
RefactoringWizardOpenOperation op =
    new RefactoringWizardOpenOperation(wizard);
op.run(shell, "XY Refactoring Dialog Title");
```

Here a simple example for the page which is added to the wizard above.

```
class XYUserInputWizardPage extends UserInputWizardPage {
    public void createControl(Composite parent) {
        //Here you can add your refactoring specific SWT widgets
        //to the composite parent (method argument) that takes
        //input from the user to configure your refactoring.
    }
}
```

Following a simple example-implementation of XYRefactoring

```
public class XYRefactoring extends Refactoring {
    public RefactoringStatus
        checkFinalConditions(IProgressMonitor pm) {
        return new RefactoringStatus();
    }
    public RefactoringStatus
        checkInitialConditions(IProgressMonitor pm) {
        return new RefactoringStatus();
    }
    public Change createChange(IProgressMonitor pm) {
        //Create a new Change object based on the
        //input the user made on the input page
    }
    public String getName() {
        return "XY Refactoring";
    }
}
```

Refactoring Support for the Eclipse Ruby Development Tools

The method `createChange` gets your refactoring change and shows the concerned documents on the last wizard page. When the user confirms, the change will be applied to the concerned documents.

The methods `checkInitialConditions` und `checkFinalConditions` provides a possibility to check if the refactoring is able to perform right now.

The following paragraph shows you how a simple change is created. In this example you see the the insertion of a text segment. Instead of creating an `InsertEdit` there could be a `ReplaceEdit`, a `DeleteEdit` or even a `MultiTextEdit` that can contain multiple edits.

```
IFile file = getFile(); //Retrieves the concerned document
TextChange change =
    new TextFileChange("Name of the Change", file);
String insertText = "Text to insert";
int insertPosition = 17;
TextEdit edit = new InsertEdit(insertPosition, insertText);
change.setEdit(edit);
```

Class Diagram

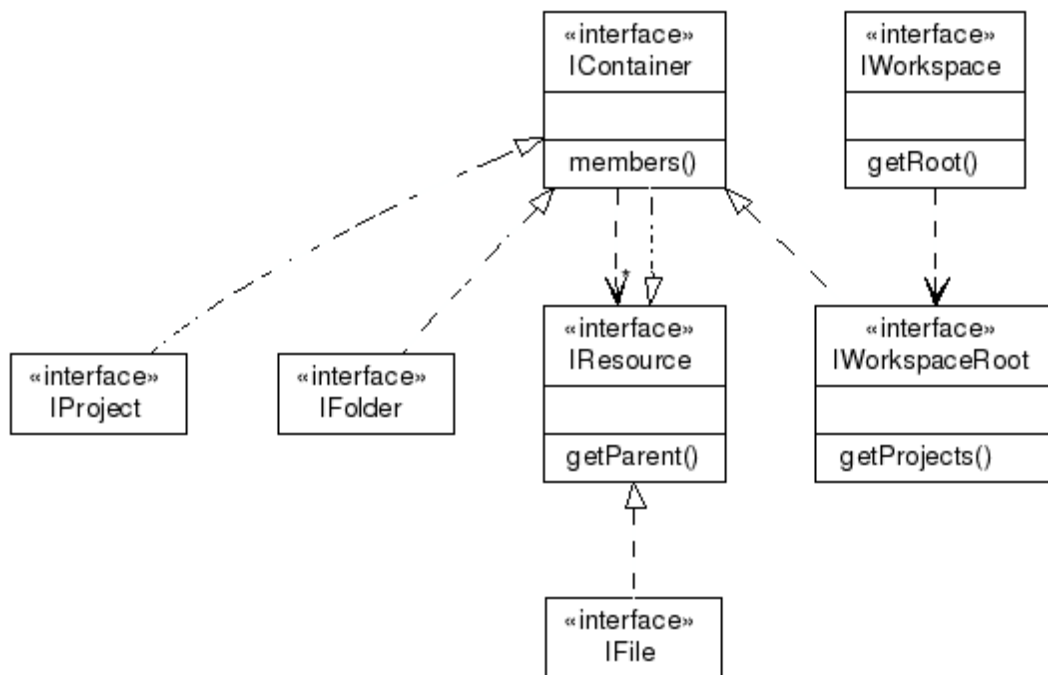


Figure 7.21: Resource Management Diagram

7.5.3 Eclipse Rules

When a developer writes an Eclipse plug-in there are a few rules that give the developer hints how he should behave himself while developing his plug-in. The next sections gives a overview over a small selection of those rules.

Lazy Loading Rule

The lazy loading rule says that you should create "big" objects only at the time when they are really used. The Eclipse plug-in system itself follows this rule. A plug-in is only loaded at the time it gets required and not before.

The Ruby refactoring plug-in follows this rule as well. When the popup menu of the Ruby editor gets shown, only the small action objects are created. These object only know the name that is shown in the menu and the Java class of the refactoring. The instance of this class is created via reflection when the popup menu entry is clicked.

If you want to know more about patterns for resource management, please read Pattern Oriented Software Architecture, Vol.3 [?].

Monkey See / Monkey Do Rule

This rule gives you an advise how you should go on when you do not know how to implement your plug-in. You can access the the source off all existing Eclipse plug-ins. The rule tells you to use this source of information and look at those plug-ins to get an idea how to go on. So you (the monkey) looks at existing things, copies them and adapt them to your needs.

Safe Platform Rule

A principle of Eclipse is, that errors (or Exceptions) are always handled properly and never shown to the user of the plug-in. This gets really important when software of other developers gets involved with your plug-in. This is the case when your plug-in provides extension points that gets extended by foreign plug-ins.

7.5.4 Resource Management

In a plug-in system like Eclipse, there are a lot of different components that might modify (edit, create, delete) resources. Many plug-ins require to track the state of resources. To solve the resource tracking problem, Eclipse adds, what could it else be, another layer of abstraction. Eclipse provides resource handles, IResources, which provide access to the concrete resource itself. These handles are a combination of the known patterns Proxy and Bridge [?]. Like this you can provide functionality for resources that might even have been deleted (Proxy) and you may have different implementation of resources behind one abstract handle.

Resources Class Diagram

Here you can get an overview over the Eclipse resources model. By the way, it is a composite pattern described in Design Patterns [?].

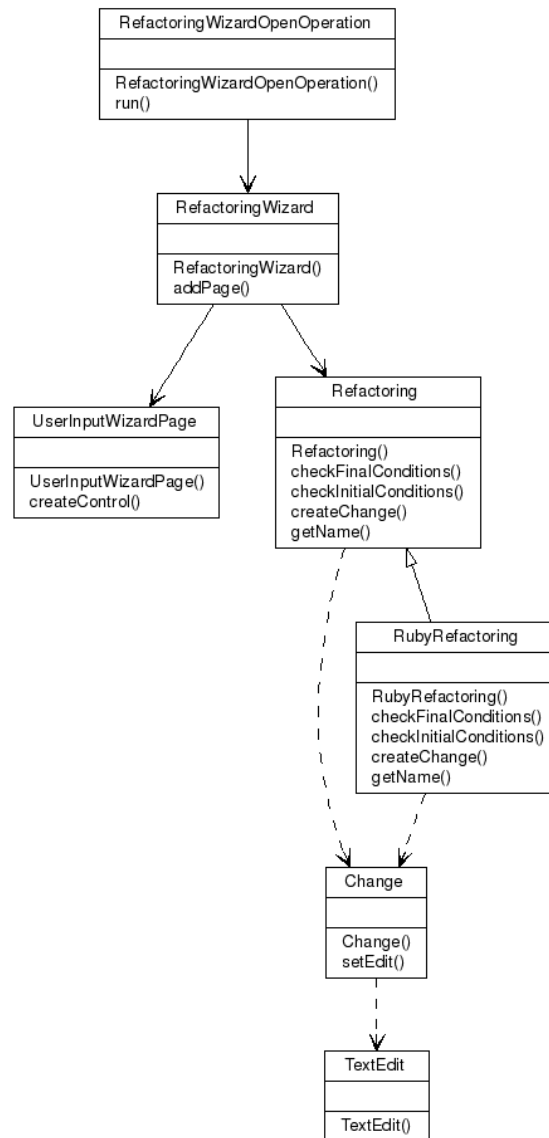


Figure 7.22: Eclipse Refactoring Support Diagram

Code Samples Diagram

To get access to resources under Eclipse, there are two ways. To access resources of the active editor you can use the following:

Refactoring Support for the Eclipse Ruby Development Tools

```
IEditorPart editor = PlatformUI.getWorkbench()
    .getActiveWorkbenchWindow().getActivePage().getActiveEditor();
```

The other more official way to access resources is the `ResourcesPlugin`. It gives access to all the resources from a specific Eclipse project. The following code shows you how you can get to the workspace root.

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
IWorkbenchRoot root = workspace.getRoot();
```

Now you either can go through its child resources by using the `members` function:

```
for(IResource aktResource : root.members()) {
    switch (aktResource.getType()) {
        case IResource.FILE:
            System.out.println("It's a file.");
            break;
        case IResource.FOLDER:
            System.out.println("It's a folder.");
            break;
        case IResource.PROJECT:
            System.out.println("It's a project root.");
            break;
        case IResource.ROOT:
            System.out.println("It's the workspace root.");
            break;
    }
}
```

Or you can use `IPaths` to navigate with file and directory names through the resources:

```
IPath path = (IPath) root.getFullPath().clone();
path.append("project_name/folder_name/file_name.xy");
IFile file = root.getFile(path);
```

8 Testing (Automated)

8.1 Refactorings

The refactorings are tightly integrated with the user interface. This always causes problems when creating tests, because there is, at least with SWT user interfaces, no smart way to test the components with faked user input. So the tests for our refactorings are set up just a little below of the user interface components.

The direct output of any refactoring is always one or multiple text edits. This means that every refactoring implements the IEditProvider interface. And that is where the main tests for the refactoring take place.

Some of the implemented refactoring give the user so-called trees, where the user can check some items. The content of those trees is fed into the tree through an ITreeContentProvider. When a refactoring provides a tree, there is a test that checks if the tree content provider provides the expected content.

8.1.1 Edit Provider Tests

The tests of the refactoring are implemented mainly after the following model. The IEditProvider is created. This is the class that contains the main logic of a refactoring. It is given one or several ClassNodeProviders that are needed to create the resultig text edits. Now we tell the IEditProvider, or we might say fool it, with some "user input". Then we run the test itself. The test is given the IEditProvider, the document on which the text edits needs to be applied, and the expected result of the document after the refactoring was performed on it. The test now takes the IEditProvider, gets all the text edits that it provides, applies them to the document and compares it to the expected document.

Here an example how such a test could look like:

```
public void testXY() {
    addSelection("X", "aThing");
    addSelection("X", "bThing");
    ClassNodeProvider classNodeProvider =
        new ClassNodeProvider(testSourceFile);
    IEditProvider editProvider =
        new XYEditProvider(classNodeProvider);
    validate(editProvider, testSourceFile, expectedResultFile);
}
```

The test above simulates the selection of the content of a tree (first level: the entry "X", below it: "aThing" and "bThing"). Then it creates an IEditProvider and runs the test.

8.1.2 Tree Content Provider Tests

The ITreeContentProvider usually takes one or several ClassNodeProviders to evaluate the tree content to show. So an example for such a test looks like this:

```
public void testXY() {
    addContent(new String[]{"X", "printSomething"});
    addContent(new String[]{"X", "doSomething"});
    ClassNodeProvider classNodeProvider =
        new ClassNodeProvider(testSourceFile);
    ITreeContentProvider treeContentProvider =
        new XYTreeContentProvider(classNodeProvider);
    validate(treeContentProvider);
}
```

The example above tells the test through the addContent method that the expected tree contains on the first level an item "X" and on the second level below the "X" two entries called "printSomething" and "doSomething".

Like this, every tree content provider test first "builds up" the expected tree using the addContent method. Then the ITreeContentProvider that will be tested is created and given the required ClassNodeProvider. After that the ITreeContentProvider is tested using the validate method. This method compares each tree entry the ITreeContentProvider provides with the expected tree content defined before.

8.2 AST Rewriter

Our first unit tests for the rewriter were implemented as the classical JUnit tests where each method had its corresponding test-method and we compared the input with the expected output, using Java Strings to represent the source. But this became very annoying to write, since we had to escape every line and write correct indentation. On our supervisor's advice, we used a much more flexible approach. The tests are now written into a file as normal Ruby code, separated into different sections, to give a better overview. An extract from the tests looks like this:

```
##!Match3Node
date = "12/25/01"
date =~ /(\d+)(\| |:)(\d+)(\| |:)(\d+)/
```

On the first line, you can see the separator for the tests. This is also the displayed name, if the test fails and should give you a hint what you actually tested. The lines written before the next separator are fed into the rewriter and the result is compared to the input. The test fails if they are not fully equal.

If you want to test code where the output will differ from the input, you can use the following syntax:

```
##!Match2Node
next if (/CVS$/ =~ File.dirname(f))
##=
if (/CVS$/ =~ File.dirname(f))
  next
end
```

In this case, the part after the `##=` is used as the expected result for the comparison. We believe that this method has several benefits: For one, you do not have to escape special characters in Java Strings and the test is much better readable and easier to extend. Another benefit of this approach is, that we can use the C-Ruby interpreter to validate our testing source, since it does not make much sense to test incorrect code.

The test file contains about one thousand lines of Ruby source, so we think we have quite a good test coverage. Since the rewriter does heavily rely on the parser and lexer and because we were running out of time, we did not specify any special tests for the parser and lexer but consider them as indirectly tested through the rewriter.

8.2.1 Testing of JRuby adaptations

As the AST Rewriter runs with our adapted JRuby version, we have been able to use it for testing the parser and lexer changes. Beside this we had to create a set of code to fire the adapted production rules to see whether the comments are handled correctly.

9 Summary

In this chapter, we would like to sum up our report and give you an outlook into the future of our refactoring works.

9.1 Results

We think that we achieved a lot in those fourteen weeks of the project: We had to restructure the parser, lexer and modify a lot of the existing code. There still are a few little problems to solve in this area, but we think we finally can fully integrate all comments in the AST. On the other side, we have a basic rewriter, that already honors some of the user's coding style preferences. But this part needs more work, too. It might still have some bugs and comments might be lost during the process. On the RDT side, we have an infrastructure where we can build more refactorings for the future. We implemented three code generators and two refactorings. They work quite well. There are points, where they might be improved and extended. We think that they now need to be applied in the field and we hope we get some reports back from users who tried them out.

9.2 Further Work

As we said before, some existing parts need improvement. Besides that, we are going to implement new refactorings or other features that are useful for the RDT users. We also want to integrate closer with RDT, for example to allow the renaming of types through the outline view. Perhaps we can even implement Drag&Drop of types in the outline.

During the Google Summer of Code project, Jason Morrison¹ is implementing type inference for RDT. Type inference is just a more sophisticated way to express the guessing of the type a certain variable has. Since we do also rely on information about variables, we can perhaps work together, or build upon Jason's work, to create refactorings like the renaming of public methods. We will certainly track the progress of his work.

We also plan to meet with the RadRails developers to discuss some Rails-specific refactorings they might need.

¹<http://soc.jayunit.net/>

9.3 Known Issues

As far as we know, there are no grave issues that would prohibit the use of our refactorings in productive environments. The worst known case is that comments get lost when appended to a null node, Anyway we encourage users to build good unit tests and let them run after refactoring. We would also really appreciate the report of bugs in our Trac².

9.4 Outlook

We will continue this work as a diploma thesis in the fall this year. Since we now have a working basis and a lot of knowledge about various sections of RDT and JRuby, we are then going to implement more refactorings. In the mean time, we are awaiting the reactions of the community and we might even write an article for the german issue of the Eclipse magazine.

9.5 Personal Comment

Working on RDT and JRuby was great fun most of the time. We really appreciated the help of the communities and their encouraging comments and suggestions. All three members of the team were happy to work on something that does really provide a benefit to others, at least we hope it does. We are full of anticipation on continuing our work after the semester break.

²<http://morki.ch/rubyrefactoring/newticket>

Appendix

Field Reports

Lukas Felber

The experiences I made during the work on this project were mostly positive ones. I think my two friends and I made a really good team and that we can be proud of the results we acquired during our fourteen weeks of work. The communication inside the team was very good. Whenever you needed some help you always got a helpful input from a team member.

I also liked the meetings with our supervisor Prof. Peter Sommerlad. He always had a lot of helpful input for the problems we brought to him. He told us directly what he expected, what he liked and what did not like. The communication between him and our team was always relaxed and he was always up for a joke or two.

It was a good experience to work at this project, even if it took a huge amount of time. It is good to see the active community around RDT and JRuby. Their interest in what we did encouraged me to go on with my work.

I learned that you will never be able to tell before how much time planned activities in a project will take. An existing project that provides you with basics increases that problem because you can never know how much of the functionality you need is really implemented and if the code is buggy or not. You always need to plan iteratively and be able to adapt the requirements to the active situation.

I though liked to see how over work got integrated into JRuby and RDT. To me this project was by all means a very good experience.

Mirko Stocker

I remember that I was really happy when I heard that our team got accepted for this term project, since it was the most interesting in the whole assortment. In my opinion, of course. And I was right, the project was very interesting and I think I have learned a lot about Ruby, parsers and Eclipse during the last fourteen weeks. Naturally, there were also things that did not work that well, for example the automated build and ant, which can be quite nasty beasts. But we finally got it more or less working.

One of the highlights during my work was the first time when the rewriter was able to rewrite syntactically correct code for rmagick, rake and freeride and of course when I was

added as an RDT developer at sourceforge. I really enjoyed this project and I am eager for the next part after the semester break.

I also want to thank the community and Prof. Peter Sommerlad for supporting and encouraging us.

Thomas Corbat

I've always expected the last semester to be easiest of all, having only a few lessons and much time for working at the term project. Eventually I was proved to be completely wrong. In fact I now think it was probably the hardest semester I have had at the HSR. Beside inscribing in six advance modules, that take a lot of time to visit, three close relatives died what dragged me down personally for a long time. So I haven't been able to commit to the term project much as I wanted all the time.

Nevertheless I'm very excited about our project, JRuby and the RDT. It's been a very challenging assignment to analyze and implement the various models of introduction comments in JRuby. There have been some throwbacks, that we've tried to compensate with more efforts, especially when time drew close. The only thing I'm not quite satisfied of is that yet we lose some comments in the case of a null node, and that there has been no time to eliminate this behavior. This project imparted the impression that the results are important for someone, this resulted in a special and motivating feeling.

Furthermore I'm very happy in this very skilled and ambitious team and with our supervisor, who always supported us with useful reviews and clues. Now I'm looking forward to the diploma thesis.

Environment

In this section, we are going to give you an overview to our working environment, what tools we used, how our automated buildsystem works and what tools we used to communicate in the team.

Eclipse

Using Eclipse as our IDE was the obvious choice, mainly because it is a great tool to develop Java and developing Eclipse plug-ins without Eclipse would not make much sense. We used the version 3.1 for the most time, in the end of the projet we had to switch to a newer 3.2 release candidate because the latest RDT used features of the new version, so we could keep up with the latest developments.

Cruisecontrol

To automatically build our code we installed Cruisecontrol³ on our server and made it build the whole plug-in after each check-in. Setting up Cruisecontrol is not an easy task, but with the help of Guido Zraggen, the Article on Build and Test Automation for

³<http://cruisecontrol.sourceforge.net/>

plug-ins and features [?] and the already existing buildfiles from the RDT people, we were able to set it up and get it running. We think we will never do a project without automated builds again, the benefits are clearly outweighing the time we spent for the setup. Cruisecontrol was configured to send all notifications to our mailinglist, so we always knew whether our latest commits broke the build or not.

Wiki

Wikis are a fantastic way to communicate and coordinate the work in a team, even more if it integrates the repository and the automated buildsystem and creates an RSS feed for the project, so we have one point where one can access all information. That is why we used Trac⁴. With Trac, we even got a ticket system, which we did not use that much until now, but could definitely become very useful if the project is getting more attention and other people want to report bugs or announce feature wishes. The wiki is located at our website <http://morki.ch/rubyrefactoring>.

Repository

One of the first things at the start of the project was to initialize our Subversion repository. As mentioned above, it is integrated into Trac, so we had a beautiful online source browser available. We made the repository readable for everyone, so people could take a look at our code. The location of the repository is <http://morki.ch/svn/rubyrefactoring>, but that might change in the future when we are moving to an official repository of our school.

Bug Reporting

Trac has a built-in ticket system that can be used to report bugs or wishes for features. We would be very happy if users would use it to give us some feedback.

Mailinglist

We also have a project mailinglist, r@misto.ch. It is not intended for public use, rather to allow the team members to communicate in an uncomplicated manner. Nevertheless, the archives are available at <http://lists.misto.ch/pipermail/r/>.

Time Reporting

Even though time recording was not demanded from our supervisor, we agreed to record the working time using a tool a fellow student developed: <http://reporter.rwf.ch/>. The analysis can be seen in figure 9.1.

⁴<http://projects.edgewall.com/trac/>

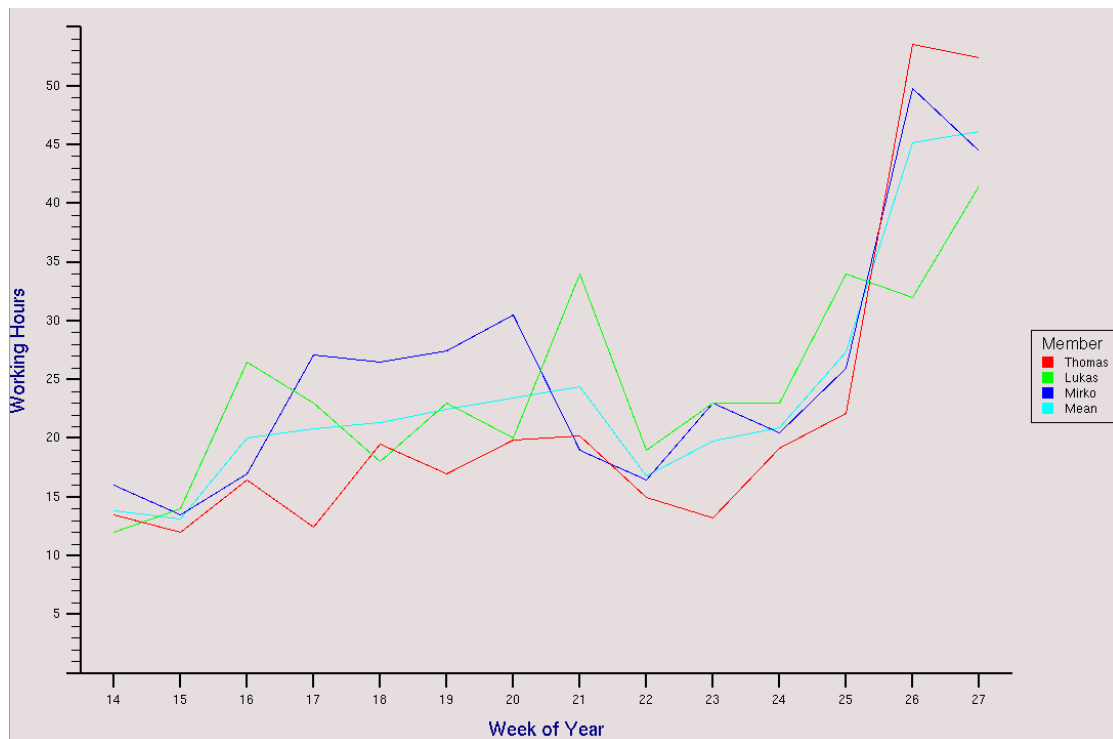


Figure 9.1: Statistics of our working hours.

Tools

The plug-ins were developed in Eclipse, running on GNU/Linux. This documentation was written in $\text{\LaTeX}2_{\epsilon}$ using Kile⁵, pdf \LaTeX and all other tools. The diagrams were generated with UMLGraph⁶.

Project Schedule

The project schedule needed some changes over time, since we got more problems than expected. It can be seen in figure 9.5.

⁵<http://kile.sourceforge.net/>

⁶<http://www.spinellis.gr/sw/umlgraph/>

Refactoring Support for the Eclipse Ruby Development Tools

	W 1	W 2	W 3	W 4	W 5	W 6	W 7	W 8	W 9	W 10	W 11	W 12	W 13	W 14
Environment Setup														
Ant / CruiseControl														
environment setup														
Analysis														
planningrefactorings														
RDT analysis														
JRuby analysis														
create simple Eclipse plug-in														
create simple refactoring														
refactoring interface to RDT														
fix JRuby node positions														
introduce JRuby comment nodes														
AST rewriter														
Refactorings														
Generate Getters and Setters														
Generate Constructor using Fields														
Add Constructor from Superclass														
Override / Implement Methods														
Rename Local Variable														
Push Down Method														
Documentation														

List of Figures

3.1	Lexer Class Diagram	23
3.2	State Machine Lexer	25
3.3	Parser Class Diagram	27
3.4	Parser Sequence Diagram.	29
3.5	AST example.	31
6.1	Class Diagram of the Visitor Pattern.	46
6.2	Sequence Diagram of the Visitor Pattern.	47
7.1	Fundamental Design	57
7.2	Node Decorators Diagram	58
7.3	Class Node Providers Diagram	59
7.4	Edit Providers Diagram	60
7.5	Generate Accessors Refactoring Wizard	62
7.6	Generate Accessors Refactoring Wizard Simple Accessors	63
7.7	Generate Accessors Refactoring Wizard Method Accessors	64
7.8	Generate Accessors Diagram	66
7.9	Generate Constructor Refactoring Wizard	67
7.10	Generate Constructor Refactoring Wizard Result	68
7.11	Generate Constructor Diagram	69
7.12	Override Method Refactoring Wizard	70
7.13	Override Method Refactoring Wizard Result	71
7.14	Override Method Diagram	72
7.15	Rename Local Variable Refactoring Wizard	73
7.16	Rename Local Variable Refactoring Wizard Result	74
7.17	Rename Local Variable Diagram	75
7.18	Push Down Method Refactoring Wizard	76
7.19	Push Down Method Refactoring Wizard Result	77
7.20	Push Down Method Diagram	78
7.21	Resource Management Diagram	81
7.22	Eclipse Refactoring Support Diagram	83
9.1	Statistics of our working hours.	93

Nomenclature

AST	Abstract Syntax Tree, describes the syntax of a program in a tree.
Eclipse	Eclipse is a framework for various types of applications, probably the most known plugin is JDT (Java Development Tools). Eclipse builds on SWT and Java and runs on most operating systems.
IDE	Integrated Development Environment, a software which supports the programmer in every necessary task, like code writing, compiling, version control, etc.
irb	Irb is the interactive ruby shell, which is quite useful if you just want to write some code without the need to create a sourcefile.
JDT	Java Development Tools, one of the most popular Java IDEs.
RDT	Ruby Development Tools, a set of plugins for Eclipse to write programs in Ruby.