HSR – University of Applied Sciences Rapperswil

Institute for Software

Refactoring Support for the Eclipse Ruby Development Tools

Diploma Thesis

Thomas Corbat
tcorbat@hsr.ch

Lukas Felber
lfelber@hsr.ch

Mirko Stocker
me@misto.ch

http://r2.ifsoftware.ch

supervised by Prof. Peter Sommerlad

December 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 supports automated refactorings. Nevertheless, in this term project and diploma thesis 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:

- Convert Local Variable to Field
- Encapsulate Field
- Extract Method
- Generate Accessors
- Generate Constructor using Fields
- Inline Class
- Inline Method
- Inline Temp
- Merge Class Parts
- Move Field
- Move Method
- Override Method
- Push Down Method
- Rename Class
- Rename Field
- Rename Local Variable
- Rename Method
- Split Temporary Variable

Management Summary

This document started as a term project and was continued by the same team as a diploma thesis.

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 that 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 widely known to the bigger part of the world, although it was around for about ten years. In our opinion, this is mainly because there was no big company behind it to push, like Sun does with Java or 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 have 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 of the term project 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 plug-in and five refactorings. This work went well except that we planed to more of this work. Because of the additional work described above we had to care for we were not able to implement as many refactorings as 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 have been implemented.

During our diploma thesis we continued expanding our plug-in. We advanced in our schedule as intended and achieved the realization of several additional refactorings. There we encountered challenges at other domains. The tasks got more user interface specific and built up on the basis we accomplished during the term project.

Outlook

The next step is the integration of the plug-in into the Ruby Development Tools. Also the cooperation with the RadRails team is intended.
With additional refactorings the plug-in could get improved. Furthermore the comment handling has to get revised or redesigned as the current implementation has some performance issues.

Contents

1 Introduction

Refactoring is a very useful concept for every software engineer. Martin Fowler, the author of the prominent book Refactoring[Fow99] 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 code, for example methods or variables.
- Extracting parts of code into other methods or classes.

The classic example of an automated refactoring is the Smalltalk Refactoring Browser [RB]. 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 project is to implement refactoring support for the Ruby Development Tools. To achieve this, we first had to create a basis for the refactorings: We needed to introduce comments in the lexing and parsing process and a rewriter for the AST. Fixing positions or other small bugs in the JRuby has been an important task too. Limited by the time we had we tried to implement as many sensible refactorings as possible.

1.1 General Situation

Ruby[1] 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

Komodo[2] 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 NetBeans

NetBeans[3] from Sun Microsystems is one of the competitors to Eclipse. It provides plug-ins for several languages and supports Ruby too, although it has not been published yet. The creator, Tor Norbye, has a weblog at http://blogs.sun.com/tor/ where writes about his progress. Among very sophisticated syntax highlighting, he implemented a refactoring to rename local variables. Unfortunately, the source code has not been released as of today, but it could be interesting for us since he uses the JRuby AST too.

1.2.3 Arachno Ruby

Arachno Ruby from Scriptolutions[4] is a commercial IDE and does not support refactorings either.

1.2.4 FreeRIDE

FreeRIDE[5] 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.

[2]http://www.activestate.com/Products/Komodo/
[3]http://www.netbeans.org/
[4]http://www.ruby-ide.com/ruby/ruby_ide_and_ruby_editor.php
[5]http://freeride.rubyforge.org/wiki/wiki.pl

1.2.5 Mondrian Ruby IDE

Mondrian[6] is a freely available IDE for Ruby, but the development has stalled recently. It does not support automated refactorings either.

1.2.6 Ruby in Steel

Ruby in Steel[7] 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.

1.2.7 Ruby Development Tools

The Ruby Development Tools[8] are a set of plug-ins for Eclipse. Bringing refactoring support to it is the aim of this project.

1.3 JRuby

JRuby[9] is a Ruby interpreter written in Java. It is fully compatible with Ruby 1.8 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 obviously cannot use Ruby extensions written in C.

1.4 RadRails

RadRails[10] 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 applications. We are in contact with them and they are eager to pick up our plug-ins.

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[BMR+96].

1.5.1 Mirko Stocker

Mirko Stocker is a 23 years old software engineer from Uerikon, Switzerland. Currently, his main interest is in Ruby and of course RDT and JRuby. He is a big fan of the

[6] `http://www.mondrian-ide.com/`
[7] `http://www.sapphiresteel.com/`
[8] `http://rubyeclipse.sourceforge.net/`
[9] `http://jruby.sourceforge.net/index.shtml`
[10] `http://www.radrails.org/`

Pragmatic Programmers and he tries to read all of their books. Besides the software stuff, he likes to read fantasy literature, painting miniatures, 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 Zürich, 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 and play floorball. He not only plays floorball but is also president his floorball team in Zürich. Besides studying at the university in Rapperswil, he works at the Swiss Federal Institute of Technology Zürich, 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 born in 1983 is 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 acquired some knowledge about compiler engineering during his studies. In the meantime he learned almost the whole Ruby syntax just by pouring over the production rules of its parser. Beside school he had a part time job at Swiss Life, a large life insurance company, in the internet security department. 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 the future. 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.
We foremost want to support the user as good as possible and disburden him from repetitious tasks as good as we can.
Note that the following descriptions only tell what the refactorings do and not how they are implemented. Mote details about our implementation can be found in chapter 7.
If you like to have more detailed descriptions of the refactorings, we would like to recommend you the book from Martin Fowler[Fow99] or his website[1].

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 in fact not really a refactoring but uses similar mechanisms as an automated refactoring. They are very useful tools 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 ways to define accessors: First, there are method calls

[1] http://www.refactoring.com/catalog/index.html

for creating accessors to allow reading, writing or both operations on an attribute. The second type are accessors 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 presents to the user 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.

```ruby
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.

```ruby
class BlogPost




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

```ruby
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 for a class. To do this, we look at the methods of the super class and choose some of them. Now the methods will be inserted into 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 class name as 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 Dog and the empty class AggressiveDog that extends Dog. In the 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
```

AggressiveDog after refactoring:

```
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

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.

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.

```
class Logger
  def log logText
    message = logText
    @network.send message
  end

  def last_message
  #oops, I need the last
  #logged message here
  end
end
```

```
class Logger
  def log logText
    @message = logText
    @network.send @message
  end

  def last_log
    @message

  end
end
```

Encapsulate Field

The Encapsulate Field refactoring means that you take a field and hide access to it behind getter and setter methods. This can be done for external access as well as from inside the class.

Seen trough 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 Encapsulate Field refactoring in fact converts accessors to accessor methods.

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

```ruby
class Person
 attr_accessor :surname







end
```

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

Extract Method

Extract Method removes a block of instructions out of an existing method and creates a new one, that contains this functionality. The new method will be called from the existing one 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 for example several variables are set.

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

```ruby
def cylinder_volume(r, h)
  h * Math::PI * r ** 2
end
```

```ruby
def cylinder_volume(r, h)
  h * circular_area r
end
def circular_area(r)
  Math::PI * r ** 2
end
```

Inline Class

The Inline Class refactoring integrates the code from a usually small class which had not much functionality into another class. Usually the target class uses the inlined one.

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

```
class Contact
  def initialize name, phone
    @name = name
    @phone = phone
  end
  def get_phone_number
    @phone.number
  end
end

class Phone
  def number
    @number
  end
  def number= number
    @number = number
  end
end
```

```
class Contact
  def initialize name, phone
    @name = name
    @phone = phone
  end
  def get_phone_number
    @phone.number
  end
  def number
    @number
  end
  def number= number
    @number = number
  end
end
```

Inline Method

The Inline Method refactoring removes a method and replaces the call with its content. This might make sense if you have almost 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
```

Inline Temp

The Inline Temp refactoring replaces all occurrences of a local variable with the value it got assigned once. Furthermore the assignment is removed as it will not be used anymore. Usually this refactoring is used to simplify other refactorings or if it does not make sense to keep the variable as it makes the code just more complex.
Of course method arguments cannot be inlined as they have no fixed assignment and thus a possibly different value with every call.

Examples Here a before-after example demonstrating the Inline Temp refactoring.

In the following method the variable `one` is senseless.

```
class Counter
  def increment
    one = 1
    @count = @count + one
  end
end
```

So we replace any occurrence of `one` with its value.

```
class Counter
  def increment

    @count = @count + 1
  end
end
```

Merge Ruby Class Parts

In Ruby it is allowed to declare a class at several places in the same or in different files. Class declarations following after a first one will logically be added to the first class declaration. This language feature is called open classes.
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 part. 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.

```ruby
class Animal
  def make_sound
    ...
  end
end

# Animal gets another method:
class Animal
  def eat food
    ...
  end
end

class Food
...
end
```

```ruby
class Animal
  def make_sound
    ...
  end



  def eat food
    ...
  end
end

class Food
...
end
```

Move Field

This refactoring moves, as its name says, a field from one class into another one. 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 should better be moved to that 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
```

Move Method

The Move Method refactoring behaves almost like the Move Field refactoring, except that it moves methods instead of fields. This is useful if you come 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 Members.

Examples Before the refactoring:

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




end
```

After the refactoring:

```
class Dog




end
class Fur
  ...
  def get_color
    ...
  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 whether 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.

```
class Shape
  def get_volume
  end
end
class Square < Shape


end
class Circle < Shape


end
```

```
class Shape


end
class Square < Shape
  def get_volume
  end
end
class Circle < Shape
  def get_volume
  end
end
```

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, as are block or iterator variables. 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 and 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.

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

So we rename it to “file_name”, which we consider much better.

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

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 search and replace mistakes and save some time, the Rename refactorings will give you a hand.

Examples Before the refactoring:

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

After the refactoring:

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

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 more confusing methods. When you replace the temporary variable with a query, which is visible in the whole class, you can 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
```

Split Temporary Variable

Programmers, especially inexperienced ones, sometimes tend to assign a temporary variable several times. The fact that Ruby does not have typed references might misleadingly encourage this. 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
```

2.2 Not Implemented Refactorings

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

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 was 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.

```
class Car
  def drive destination
    ...
  end
  attr_reader :driver_name
  def initialize driver_name
    @driverName = driver_name
  end
end
```

```
class Car
  def drive destination
    ...
  end




end
class Driver
  attr_reader :driver_name
  def initialize driver_name
    @driverName = driver_name
  end
end
```

2.2.1 Pull Up Method

The Pull Up Method Refactoring pulls a method out of one or maybe several childclasses 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. 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.

```
class Animal



end
class Frog < Animal
  def make_sound
    puts SOUND
  end
end
class Sheep < Animal
  def make_sound
    puts SOUND
  end
end
```

```
class Animal
  def make_sound
    puts SOUND
  end
end
class Frog < Animal




end
class Sheep < Animal




end
```

2.3 Comment Handling

We have seen that refactoring is a very powerful technique if used sensible. To make a developer use them they have to work correct. An often fortgotten or underestimated part of the source code are comments. You might say: Well, when a comment is not moved with its method, then the functionality is not affected. That is truly correct but why should someone want to pull up a field if he afterwards has to cut and paste the comments separately 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. 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. You can read more about our comment handling strategies in chapter 5.

3 JRuby

JRuby is a Ruby implementation written in Java and is compatible to version 1.8 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 topics 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 a 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

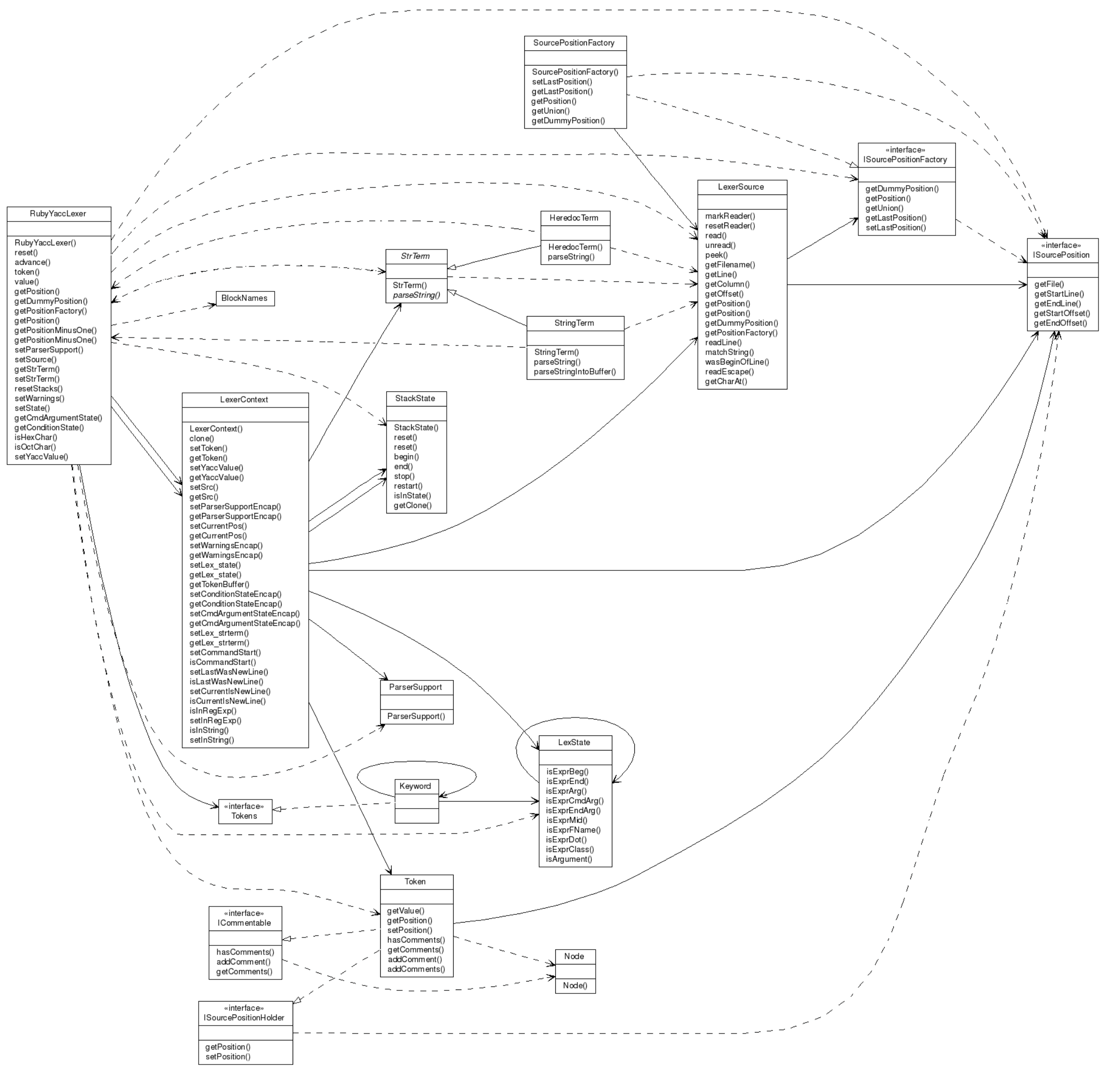


Figure 3.1: Lexer Class Diagram

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 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 inconspicuous 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. Altough 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.

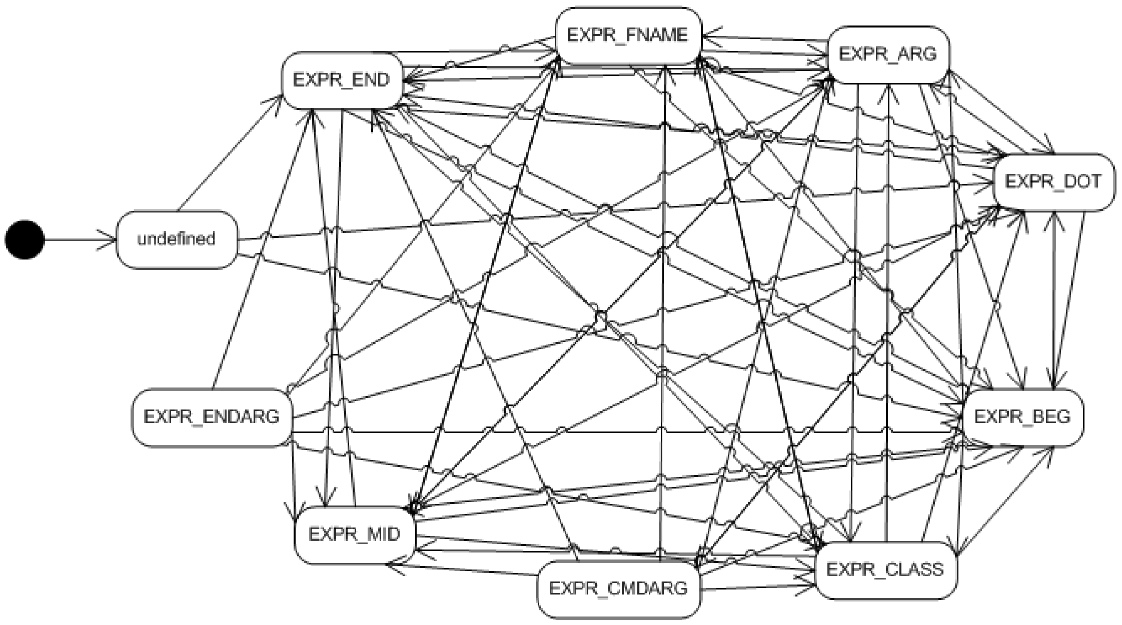


Figure 3.2: State Machine Lexer

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 themself. 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 Compiler 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 allude 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.

3.2.2 Classes

The classes of the JRuby parser are collected in the org.jruby.parser package. In this section not all the files of this package are described. We focused on the important ones.

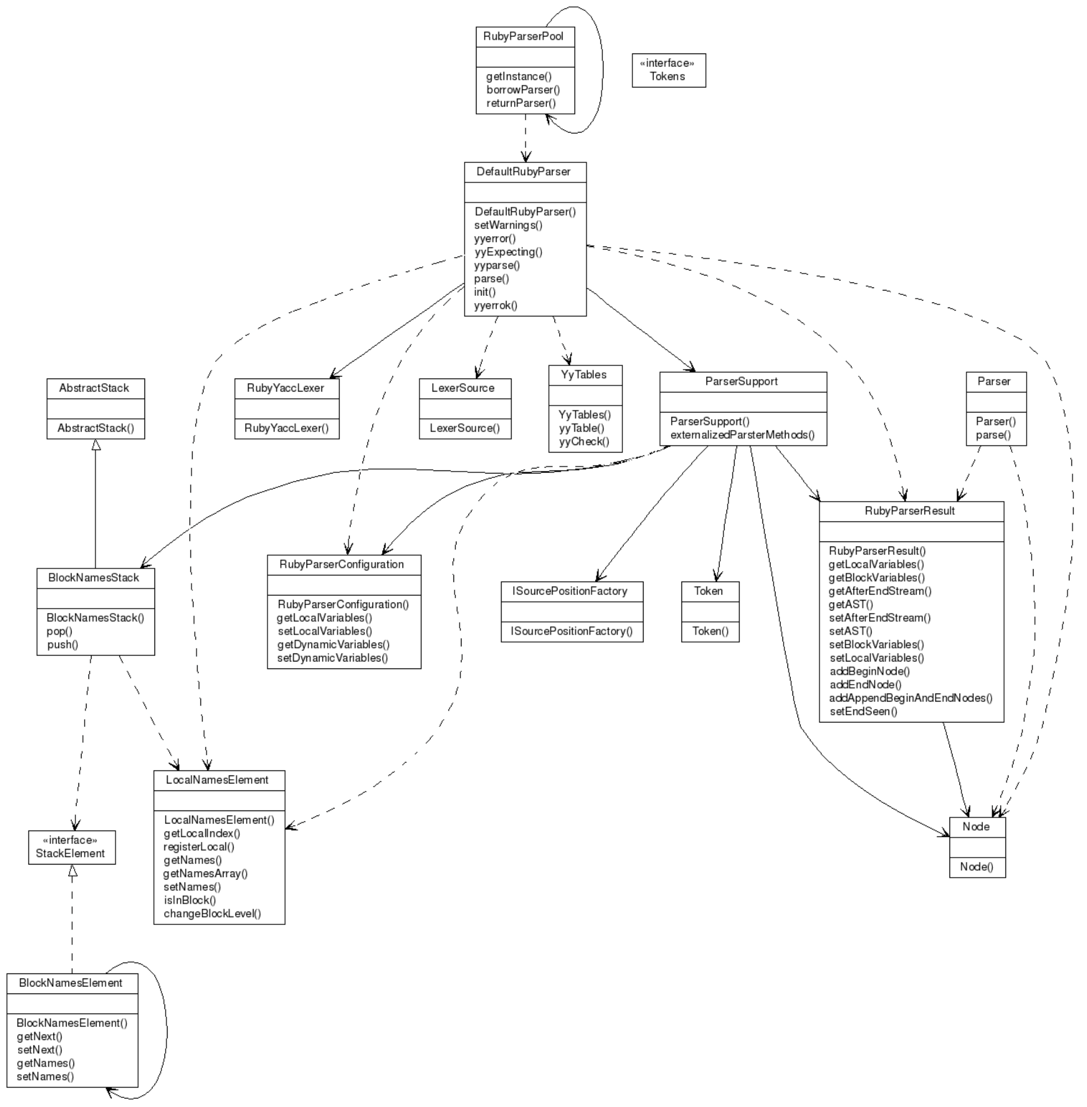


Figure 3.3: Parser Class Diagram

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 contruction 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 Default-RubyParser. 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 contruction of cases, states and transitions. As it is generated by another program anyway and not intended to be read by humans, this information is held in huge arrays. Of course this file is generated automatically with the parser.

3.2.3 Call Sequence

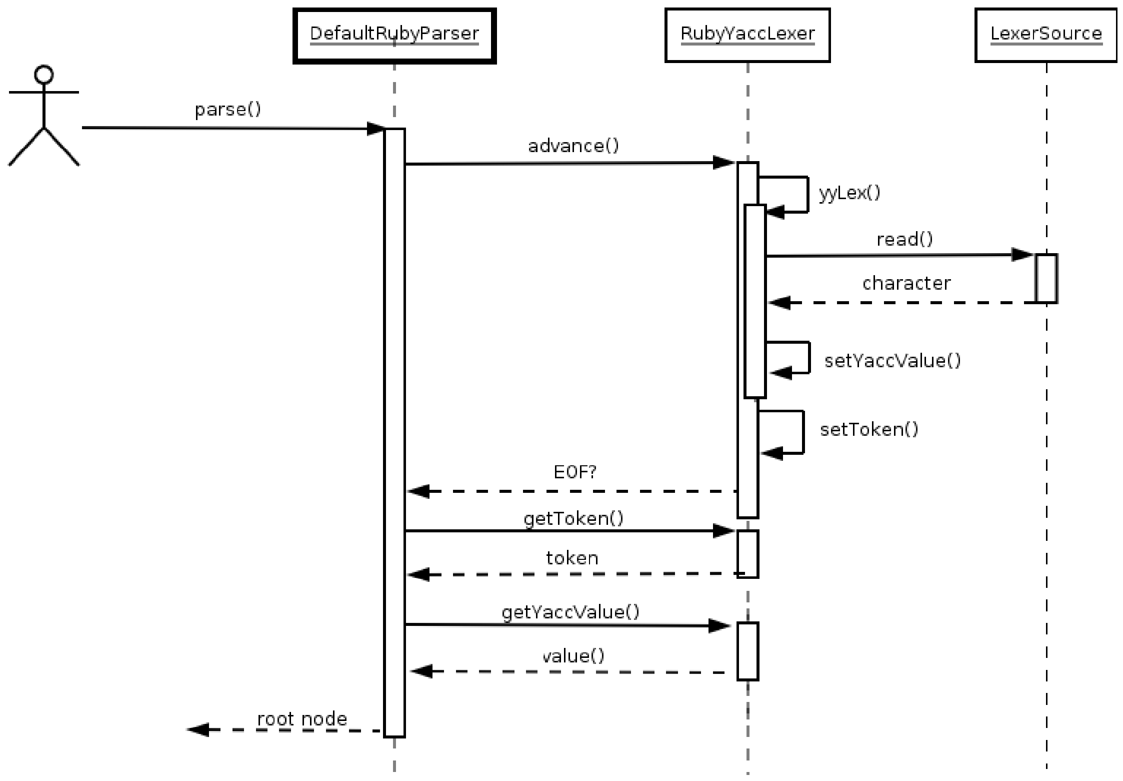


Figure 3.4: Parser Sequence Diagram.

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 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 in figure 3.5, the tree grows big very quickly with even very small blocks of code. Sometimes it is 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 being correct.

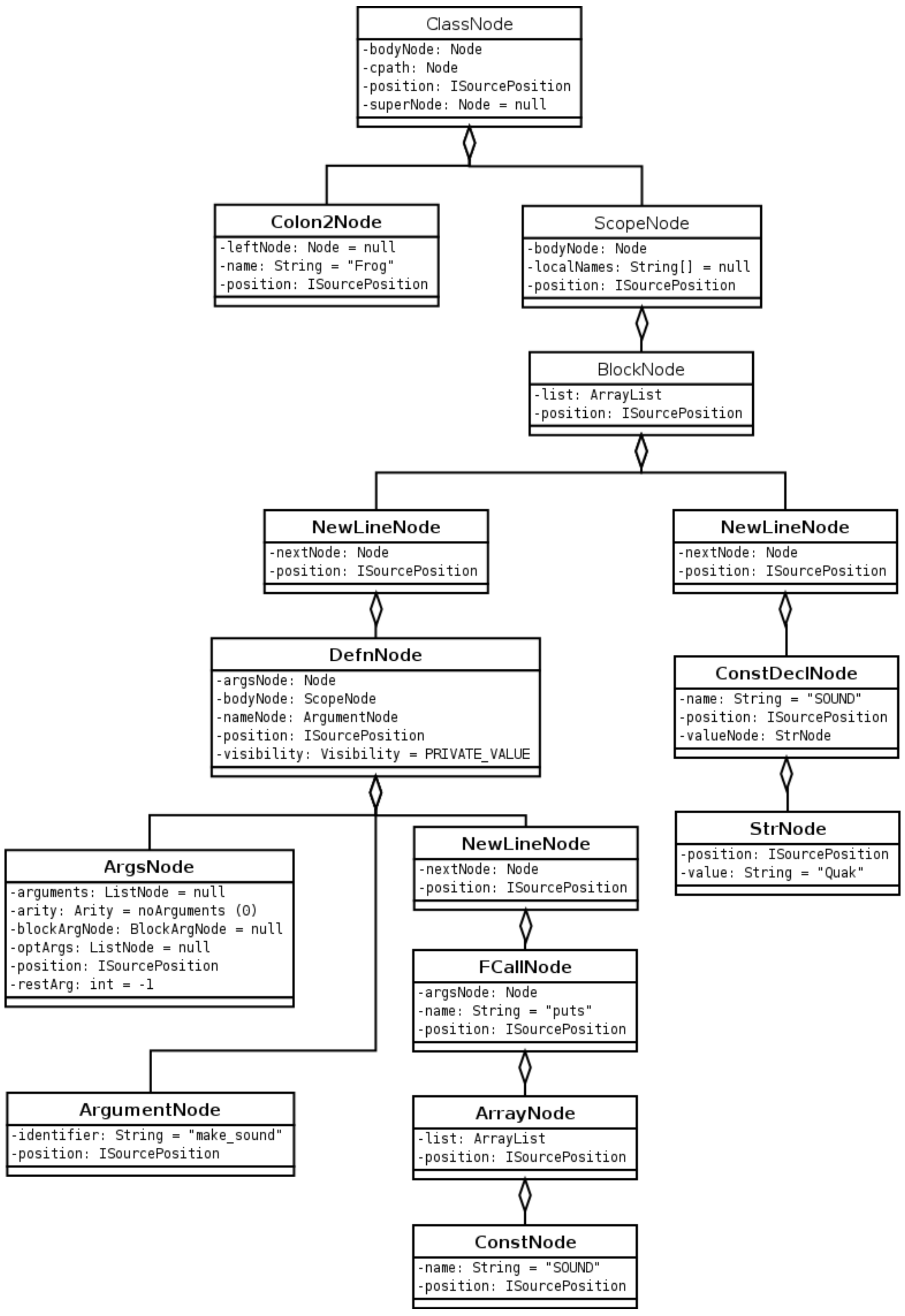

ClassNode
-bodyNode: Node
-cpath: Node
-position: ISourcePosition
-superNode: Node = null
Colon2Node
-leftNode: Node = null
-name: String = "Frog"
-position: ISourcePosition
ScopeNode
-bodyNode: Node
-localNames: String[] = null
-position: ISourcePosition
BlockNode
-list: ArrayList
-position: ISourcePosition
NewLineNode
-nextNode: Node
-position: ISourcePosition
NewLineNode
-nextNode: Node
-position: ISourcePosition
DefnNode
-argsNode: Node
-bodyNode: ScopeNode
-nameNode: ArgumentNode
-position: ISourcePosition
-visibility: Visibility = PRIVATE_VALUE
ConstDeclNode
-name: String = "SOUND"
-position: ISourcePosition
-valueNode: StrNode
StrNode
-position: ISourcePosition
-value: String = "Quak"
ArgsNode
-arguments: ListNode = null
-arity: Arity = noArguments (0)
-blockArgNode: BlockArgNode = null
-optArgs: ListNode = null
-position: ISourcePosition
-restArg: int = -1
NewLineNode
-nextNode: Node
-position: ISourcePosition
FCallNode
-argsNode: Node
-name: String = "puts"
-position: ISourcePosition
ArgumentNode
-identifier: String = "make_sound"
-position: ISourcePosition
ArrayNode
-list: ArrayList
-position: ISourcePosition
ConstNode
-name: String = "SOUND"
-position: ISourcePosition

Figure 3.5: An AST example.

Usually they were quite easy to fix, it was often enough to merge together the positions of 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:

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

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

```java
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.

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.

```
 // ENEBO: 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 then take a look at the following code.

```
-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;
```

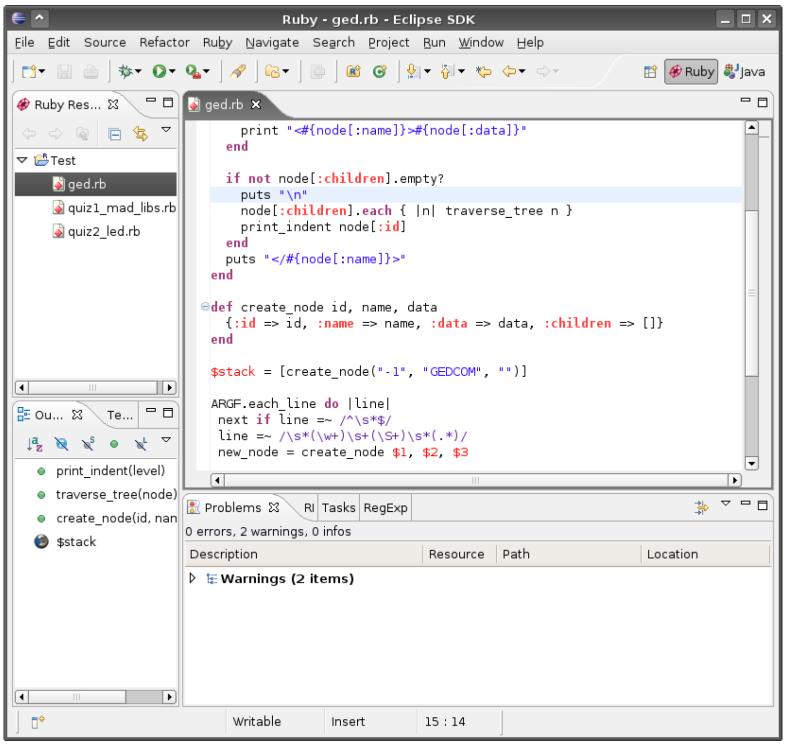
We all agreed that this was definitely no reason to make yaccValue an Object! By introducing a temporary local variable we avoided the abuse of this field.
There was only one further section where yaccValue got a non Token value assigned:

```
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 a mature IDE. It is based on Eclipse and thus can inherit a lot of features, like team support, and good integration into the development cycle.



```
    print "<#{node[:name]}>#{node[:data]}"
  end

  if not node[:children].empty?
    puts "\n"
    node[:children].each { |n| traverse_tree n }
    print_indent node[:id]
  end
  puts "</#{node[:name]}>"
end

def create_node id, name, data
  {:id => id, :name => name, :data => data, :children => []}
end

$stack = [create_node("-1", "GEDCOM", "")]

ARGF.each_line do |line|
 next if line =~ /^\s*$/
 line =~ /\s*(\w+)\s+(\S+)\s*(.*)/
 new_node = create_node $1, $2, $3
```

```
0 errors, 2 warnings, 0 infos
```

Description	Resource	Path	Location
Warnings (2 items)			

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 we used. 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 we are going to use 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 ourself 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 themself. 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 instances of 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 themself what to do with comment is also duty that might be disturbing.
- Comment nodes could only be placed in nodes that have whole lists of node. 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 Node Class" 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 foreward. 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 on 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.

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 listing 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;
 }
```

```
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 gets 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 first-Position, 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)

5.9.7 Outlook

The developed conditioning of the comments in the Ruby code had been proposed to the JRuby development team. Functionality and usefulness got assessed as good. Unfortunately there is an issue in this solution when considering the runtimes, which we did not recognize during our work. The modifications that expand nearly every production rule are too costly. The performance in large projects is unbearable slow. As JRuby is dealing with performance problems anyway our work got few chance. Mirko managed to improve the runtime behavior seriously by removing much indirection through methods, but this still was not enough to get approved to commit our changes to JRuby.
The JRuby developers intended to implement the comment handling themself. They underestimated this task and hat to recognize that is quite tricky and cannot be achieved by just modifying some parser rules. But they devised a concept how the comments can be handled anyway. In the lexer the comment tokens are put into a separate collection. Afterwards they could be interlaced into the AST by using the positions, which are mostly correct now. This concept has not been realized yet.

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. Such a plug-in is not implemented yet, but could easily be done.

6.1 General

The ASTReWriter is simply 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 [GHJV97]. 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);
  }
}
```

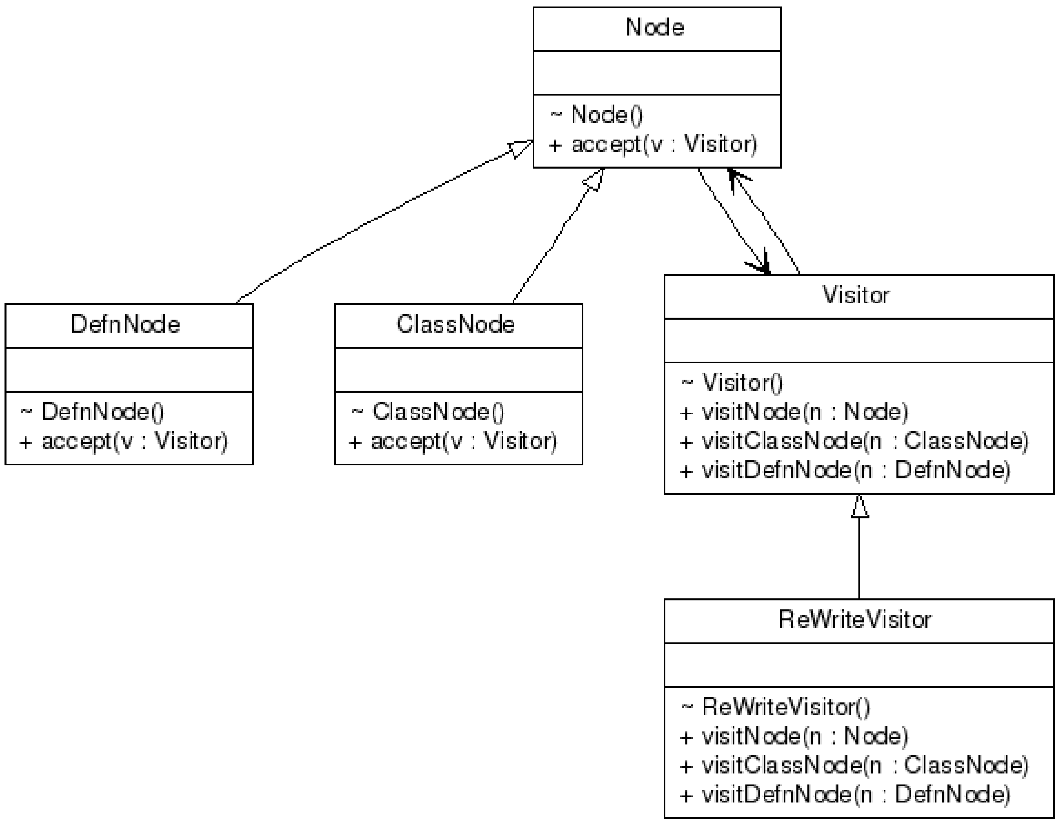


Figure 6.1: Class Diagram of the Visitor Pattern.

The class diagram 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 to be called.
Using the visitor pattern, the ReWriter can now traverse the whole abstract syntax tree and write specific source code for each node.

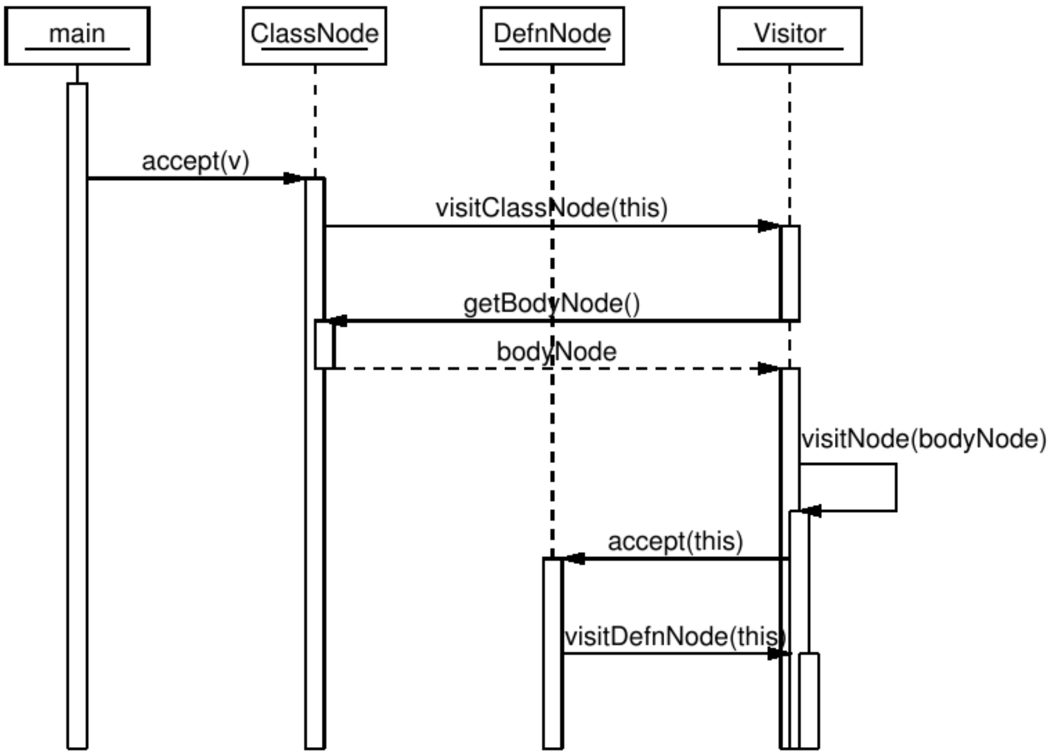


Figure 6.2: Sequence Diagram of the Visitor Pattern.

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 contains a simplified 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 a certain 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)
```

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 because this syntax is used to pass blocks. 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 automatically passes 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 [TFH05], `or` has lower precedence than ||. If you look at the following irb-session print, 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 do 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 an &. If we call this block 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 generater 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 parentheses. 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 pop-up 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 Pop-up Menu Extension Point

The Ruby Development Tools provide several plug-ins. The one used to extend the pop-up 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>
```

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 sub-menus to the menu. A demonstration follows here:

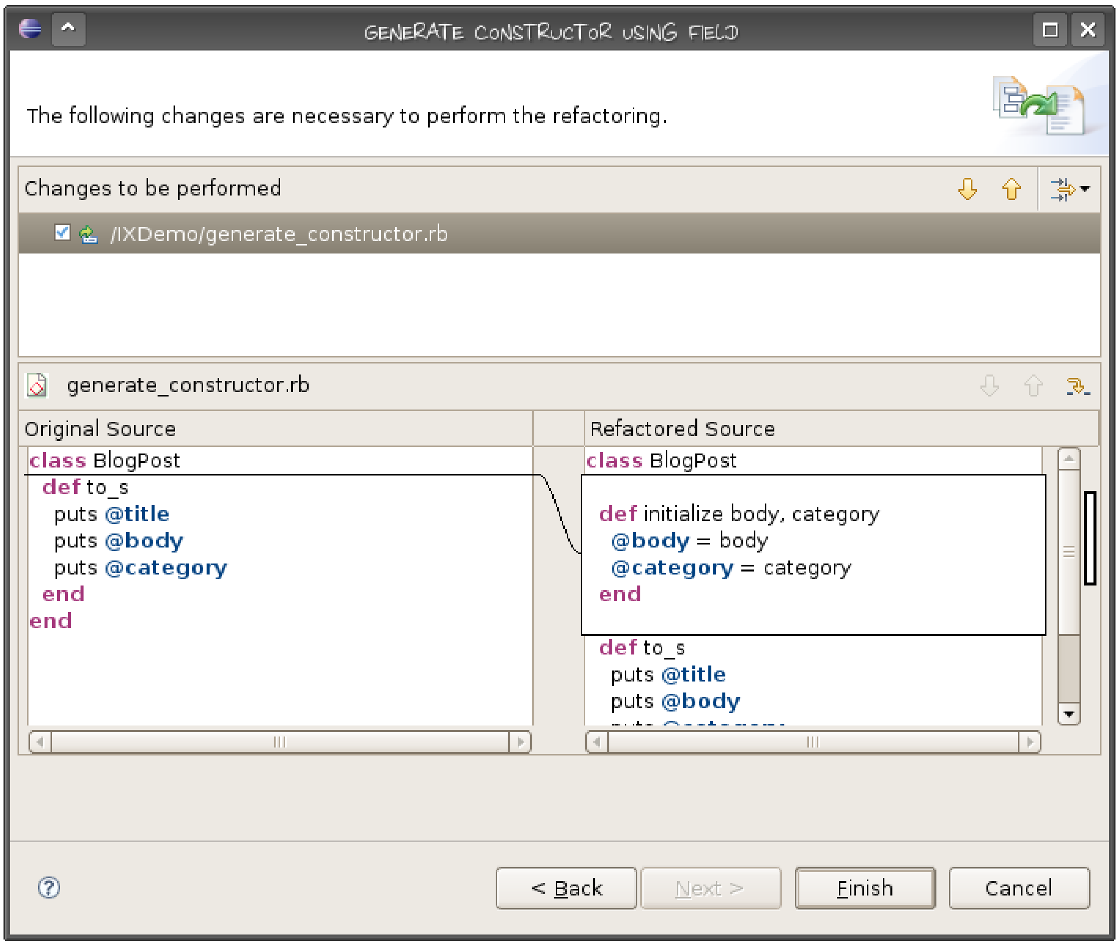
```
public class RefactoringActionGroup extends ActionGroup {
  public void fillContextMenu(IMenuManager menu) {
    MenuManager submenu = new MenuManager("MySubMenu");
    menu.add(submenu);
    submenu.add(getAction("My First Menu Entry"));
    submenu.add(getAction("My Second Menu Entry"));
  }
  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 pop-up 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>
```


GENERATE CONSTRUCTOR USING FIELD
The following changes are necessary to perform the refactoring.
Changes to be performed
/IXDemo/generate_constructor.rb
generate_constructor.rb
Original Source
Refactored Source
< Back
Next >
Finish
Cancel

Original Source:

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

Refactored Source:

```
class BlogPost

 def initialize body, category
  @body = body
  @category = category
 end

 def to_s
  puts @title
  puts @body
```

Figure 7.1: The colored merge view.

7.1.3 Colored Merge View

The merge view is used to show the changes the current refactoring is about to apply. The standard view is independant of the underlying plug-in and therefore does not apply any specific syntax coloring. To get this working, we extended the `changePreviewViewers` extension point from `org.eclipse.ltk.ui.refactoring` and added our own implementation of a `changePreviewViewer`. Most of the code was taken from Emanuel Graf and Leo Büttikers CDT Refactoring Project and we just adapted it to our environment using the relevant parts from RDT instead of CDT. You can see a screenshot in figure 7.1.

7.1.4 Outline Context Menu Integration

The refactorings were traditionally invoked either through the context menu or the "Source" and "Refactor" menus from the menubar. To integrate it better into the workflow of the user, we also inserted the refactorings into the context menu of the outline

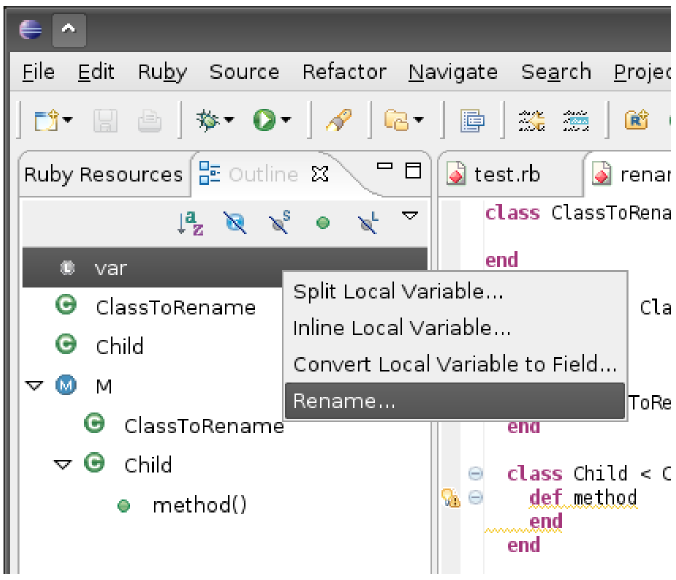


Figure 7.2: Invoking refactorings from the outline.

view where they were applicable. For example, the local variable in the outline now has four items in the contextmenu to invoke either “Split Local Variable”, “Inline Local Variable”, “Convert Local Variable to Field” and “Rename”, as you can see in figure 7.2. To associate our actions with the items in the outline, we just had to create an `objectContribution` for the `org.eclipse.ui.popupMenus` extension point.
This snippet adds the four menu points as seen in the screenshot in figure 7.2:

```
<objectContribution
        id="org.rubypeople.rdt.refactoring.refactoringContext"
        objectClass="org.rubypeople.rdt.internal.core.RubyLocalVar">
<action
        class="org.rubypeople.rdt.refactoring.action.RenameLocalVariableAction"
        id="org.rubypeople.rdt.refactoring.action.RenameLocalVariableAction"
        label="%rubyRefactoring.RenameLabel"
        menubarPath="additions" />
<action
        class="org.rubypeople.rdt.refactoring.action.ConvertTempToFieldAction"
        id="org.rubypeople.rdt.refactoring.action.ConvertTempToFieldAction"
        label="%rubyRefactoring.ConvertTempToFieldLabel"
        menubarPath="additions" />
<action
        class="org.rubypeople.rdt.refactoring.action.InlineTempAction"
        id="org.rubypeople.rdt.refactoring.action.InlineTempAction"
        label="%rubyRefactoring.InlineTempLabel"
        menubarPath="additions" />
<action
        class="org.rubypeople.rdt.refactoring.action.SplitTempAction"
        id="org.rubypeople.rdt.refactoring.action.SplitTempAction"
        label="%rubyRefactoring.SplitTempLabel"
        menubarPath="additions" />
</objectContribution>
```

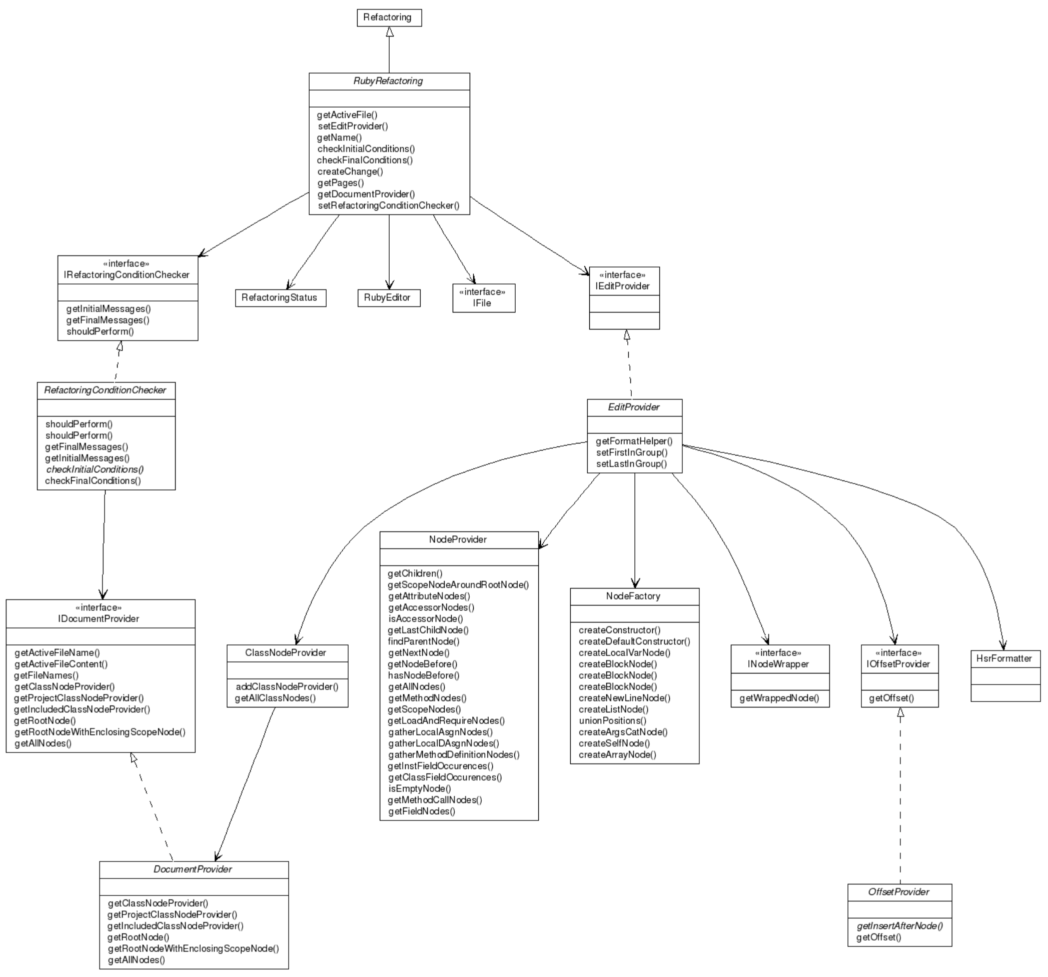


Figure 7.3: Fundamental Design

7.2 Fundamental Design of the Plugin

To implement a refactoring for Ruby, there are some components that are needed in several or all of 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. In the diagram 7.3 you can get a overview over the plugin.

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 `class` 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 Factory

Almost every refactoring generates new little ASTs that are converted to source code using the AST Rewriter (see chapter 6) and then inserted into existing Ruby documents. To keep the creation of those ASTs in one place and make them reusable for all the Refactorings, the factory class NodeFactory was created.

7.2.4 Node Wrappers

Some of the nodes are important for our refactorings. These are for example the ClassNode and the DefnNode (method node). The nodes do not really provide the functionality we needed. That is why we created wrapper classes for those nodes. From our view this is the best solution because we do not have to mess 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 diagram 7.4 you can see the created node wrappers.

7.2.5 Classnode Providers

Some of the refactorings 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.

In the figure 7.5 you can see the classnode provider diagram.

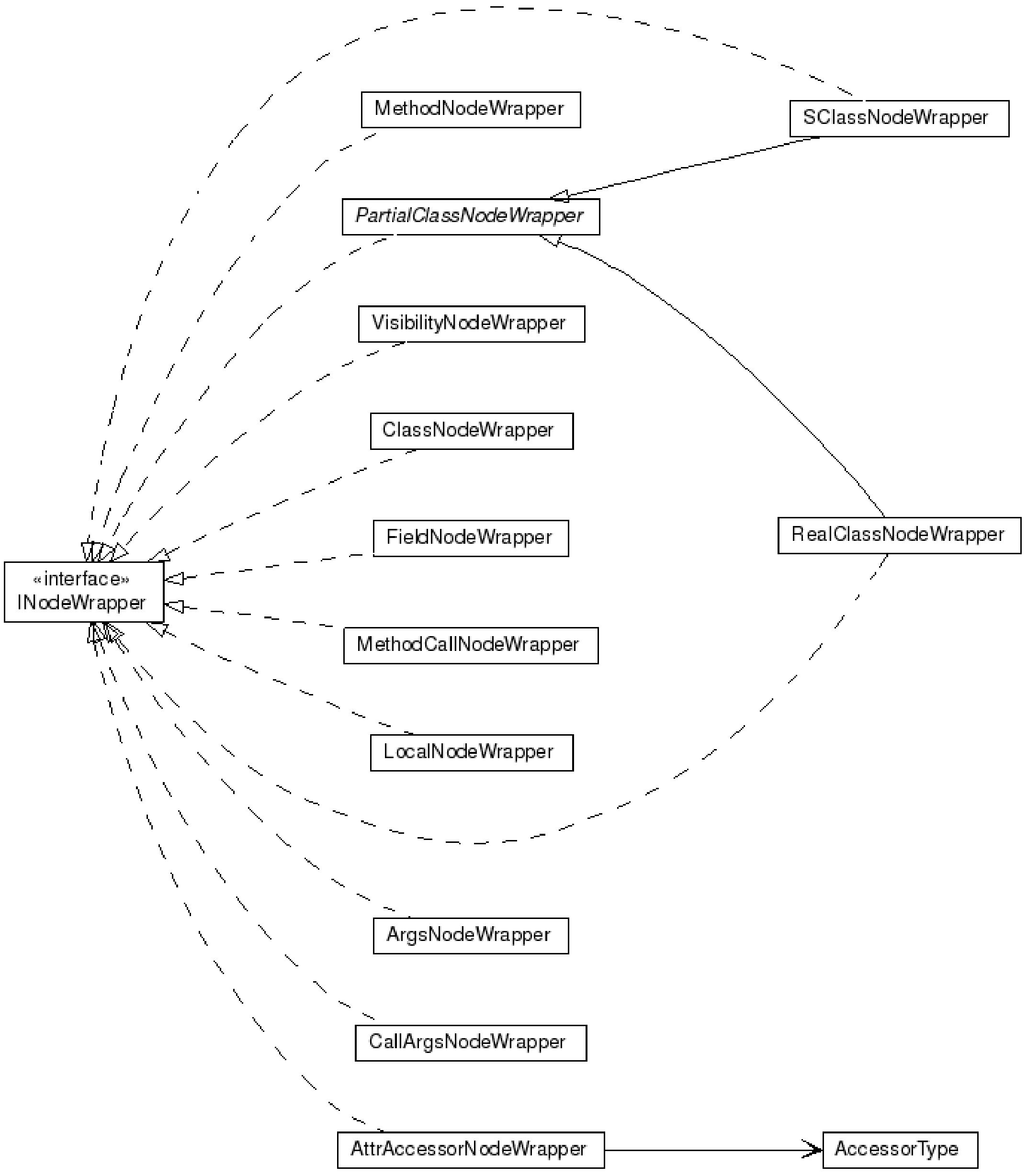


Figure 7.4: Node Decorators Diagram

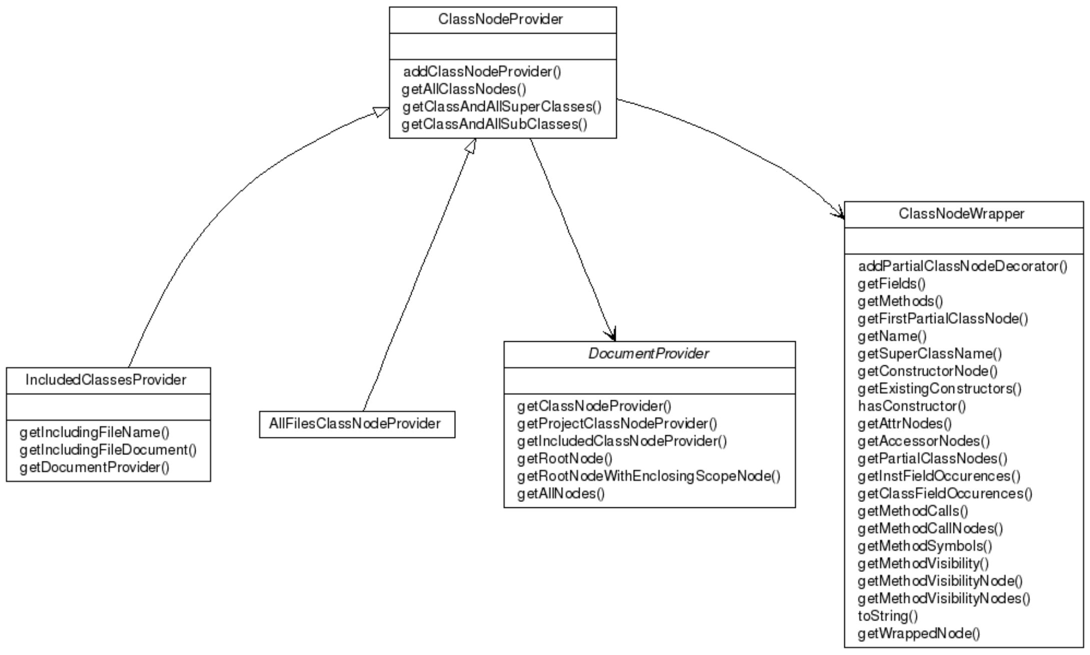


Figure 7.5: Class Node Providers Diagram

7.2.6 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 diagram 7.6.

7.2.7 Offset Providers

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

7.2.8 Document Providers

The document provider components were introduced because of a problem with our automated tests (see 8). Since all the refactorings use the active editor content and the files of the active Ruby project in Eclipse, all the refactorings depend on the Eclipse Resources Plugin. Our automated JUnit tests could not use any Eclipse plugins because they were not available when running normal JUnit tests.
To solve this problem, we introduced the document provider interface. A document provider gives the possibility to get the name and content of the active document. Further it provides you with all the filenames in the active project and with functionality that allows to get the content of those documents. All the refactorings were rewritten to re-

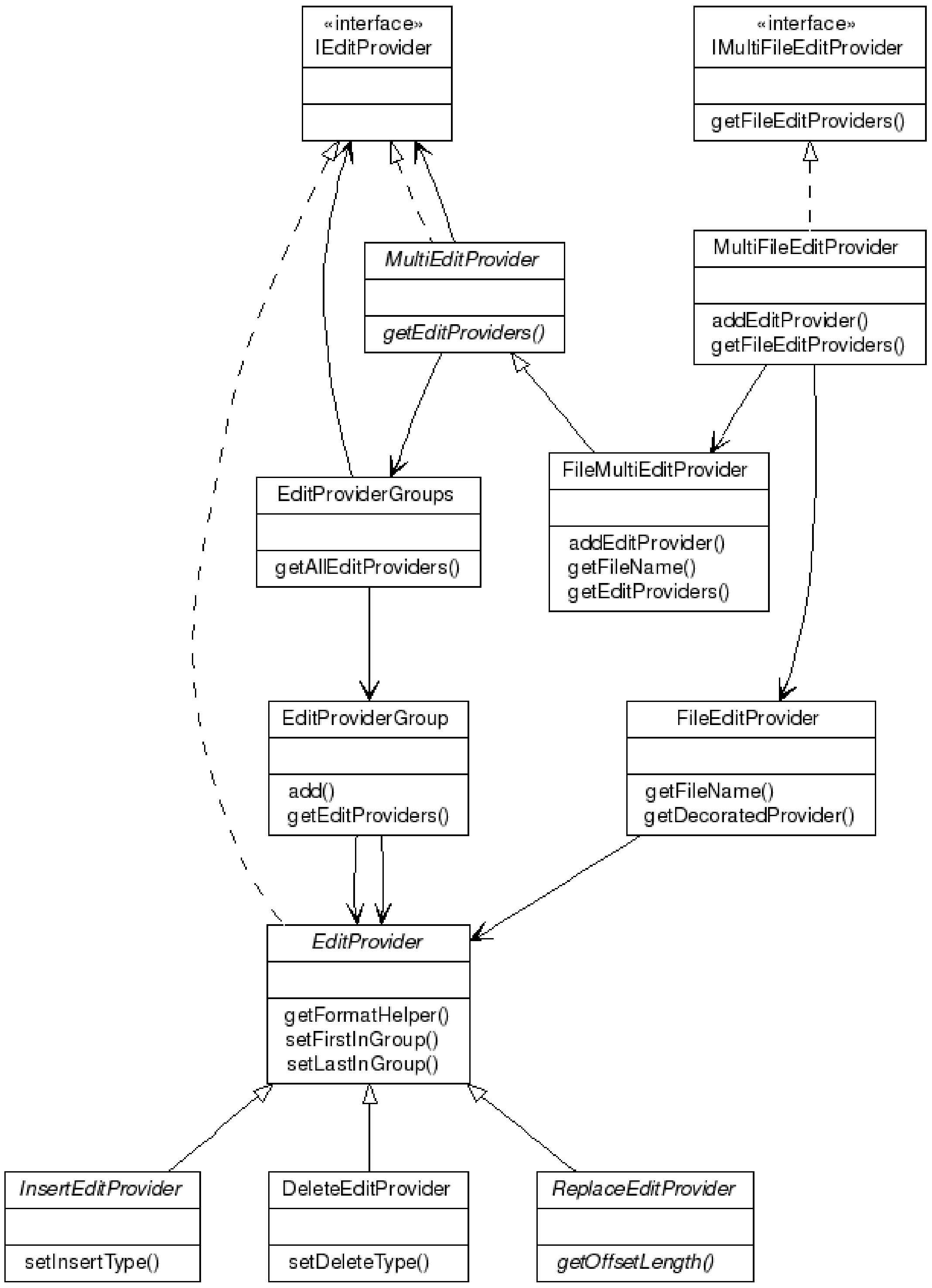


Figure 7.6: Edit Providers Diagram

move the dependency to the Eclipse Resources Plugin. Instead, they now use a document provider that grants them access to the needed files. The default document provider, the WorkspaceDocumentProvider, delegates all the access to files to the ResourcesPlugin. While testing, the refactorings use another implementation of IDocumentProvider, that gives the refactoring access to the test source files. Those source files are read directly from the OS file system and have nothing to do with the Eclipse Resources Plugin.
Like this, the refactoring does not care which kind of document provider it gets, it just takes the one it becomes and uses it. The refactorings do not know it they run in the real Eclipse environment or in the faked test environment. You can see the document providers in diagram 7.7.

7.2.9 Refactoring Condition Checker

In a refactoring, there are two method to be overwritten that are called from the Eclipse LTK to ensure that a refactoring will run properly. Those two methods check refactoring specific conditions. The first check is made after the user invoked the refactoring. This check is called initial condition check. It is meant to analyze the source code in the active editor and if needed in other files in the active project. The second check, called the final condition check is run after the user made its input on the refactoring pages. It checks if the selections the user made are valid and allows the performing of the refactoring.
Both those checks return a component called a RefactoringStatus. A refactoring status contains error, warning or information messages that will be shown to to user. If the status contains fatal errors, the execution of the refactoring will be prevented.
Each of our refactorings contains a RefactoringConditionChecker that is resresponsible run the two checks and create the necessary messages. If the condition checker says that a refactoring is not able to perform after the initial check, the refactorings edit provider and user input pages are not created. To remove the dependency between our refactorings and the Eclipse LTK plugins, the RefactoringConditionCheckers only generate collections of error and warning strings. Like this, the output of the RefactoringConditionCheckers can be tested in our automated tests.

7.2.10 HSRFormatter

RDT contains a class called OldCodeFormatter that has the job of formatting Ruby code. You can give it a string containing 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:

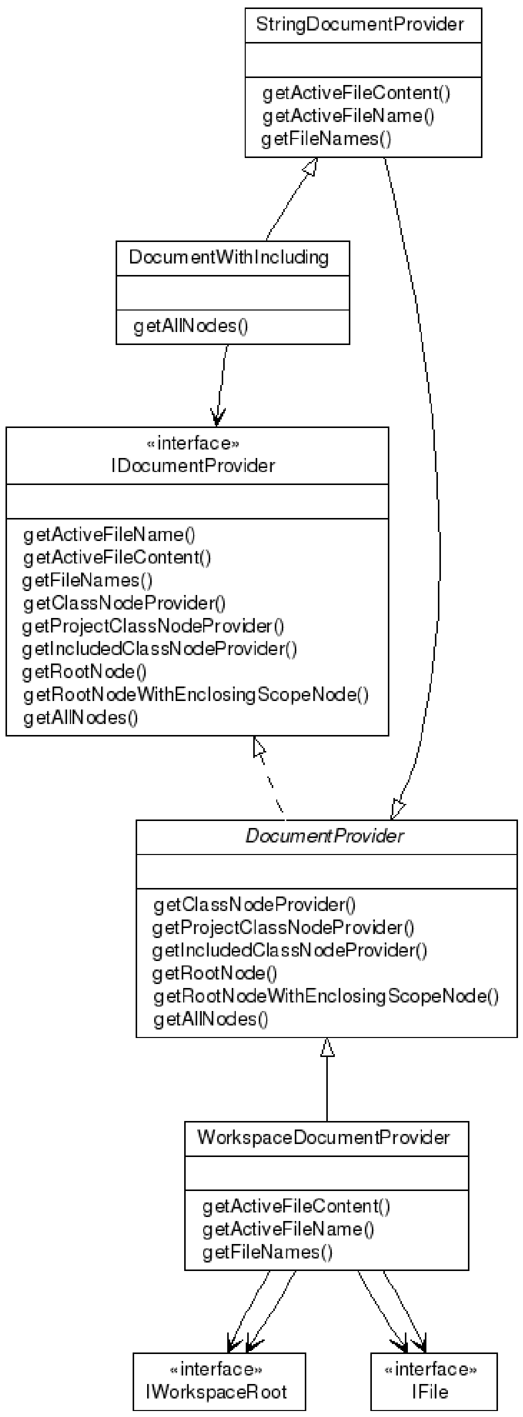


Figure 7.7: Document Providers Diagram

```
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 formated. The result of the code that will be formated should look like this:

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

Now look at the result the OldCodeFormatter gives us. Pay attention to the indentation. That is 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.

7.2.11 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 be a class node or 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.

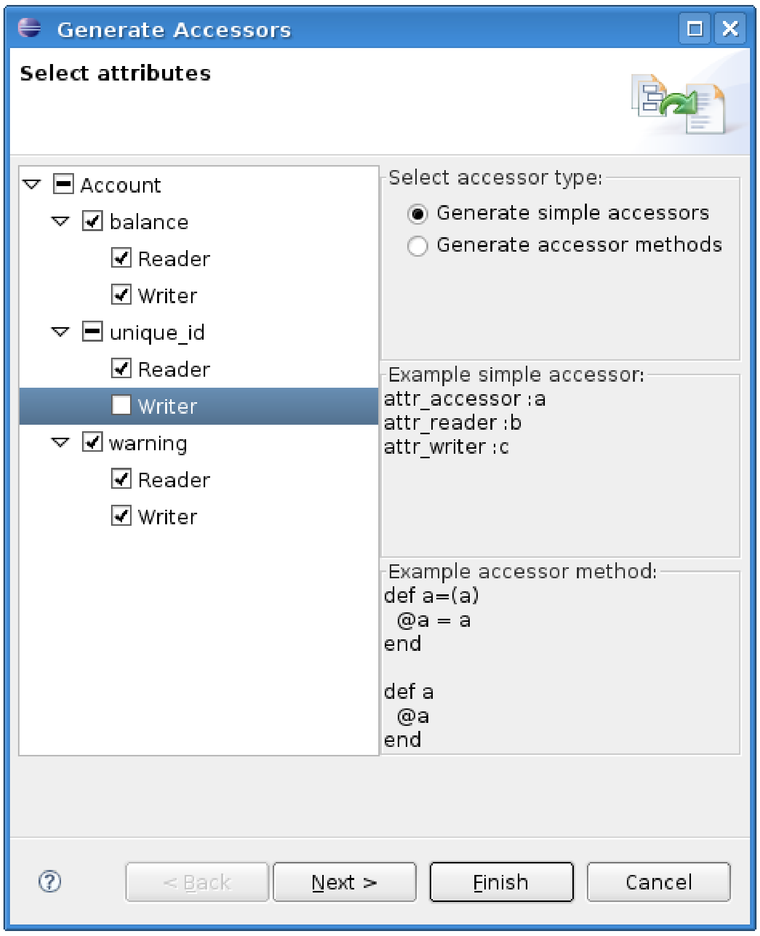
7.3 Implemented Code Generators

7.3.1 Generate Accessors

The generate accessors code generator presents the user a tree to check the accessors he would like to create. On the first level of the tree 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 of accessor type.



- Account
  - [✓] balance
    - [✓] Reader
    - [✓] Writer
  - unique_id
    - [✓] Reader
    - [ ] Writer
  - [✓] warning
    - [✓] Reader
    - [✓] Writer

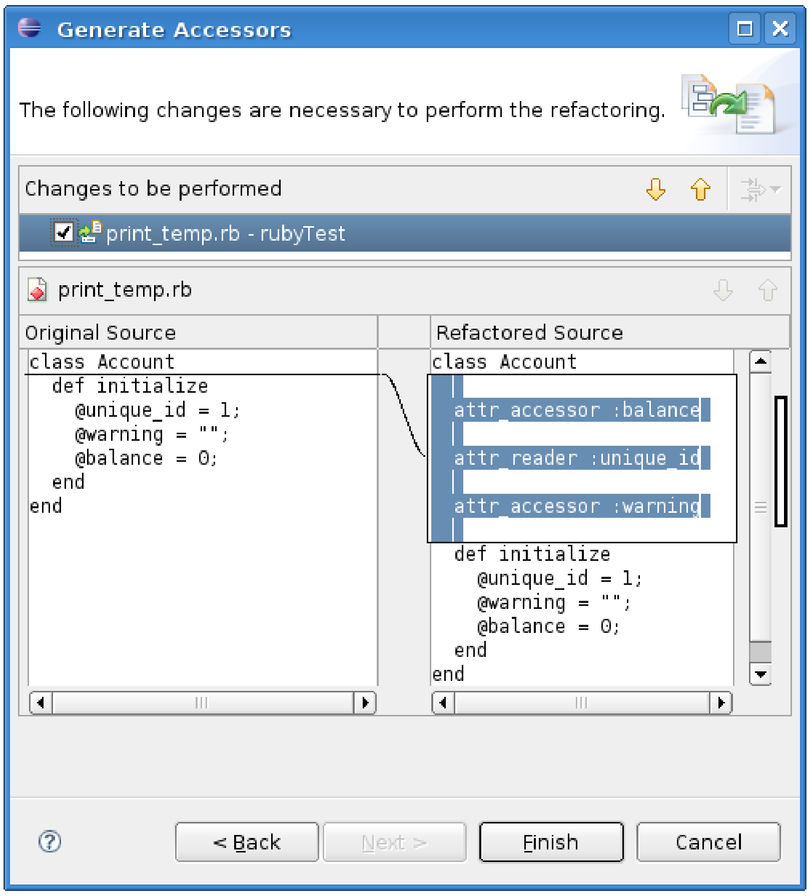
Select accessor type:
- ● Generate simple accessors
- ○ Generate accessor methods

Example simple accessor:
```
attr_accessor :a
attr_reader :b
attr_writer :c
```

Example accessor method:
```
def a=(a)
  @a = a
end

def a
  @a
end
```

Figure 7.8: Generate Accessors Refactoring Wizard

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



Original Source:

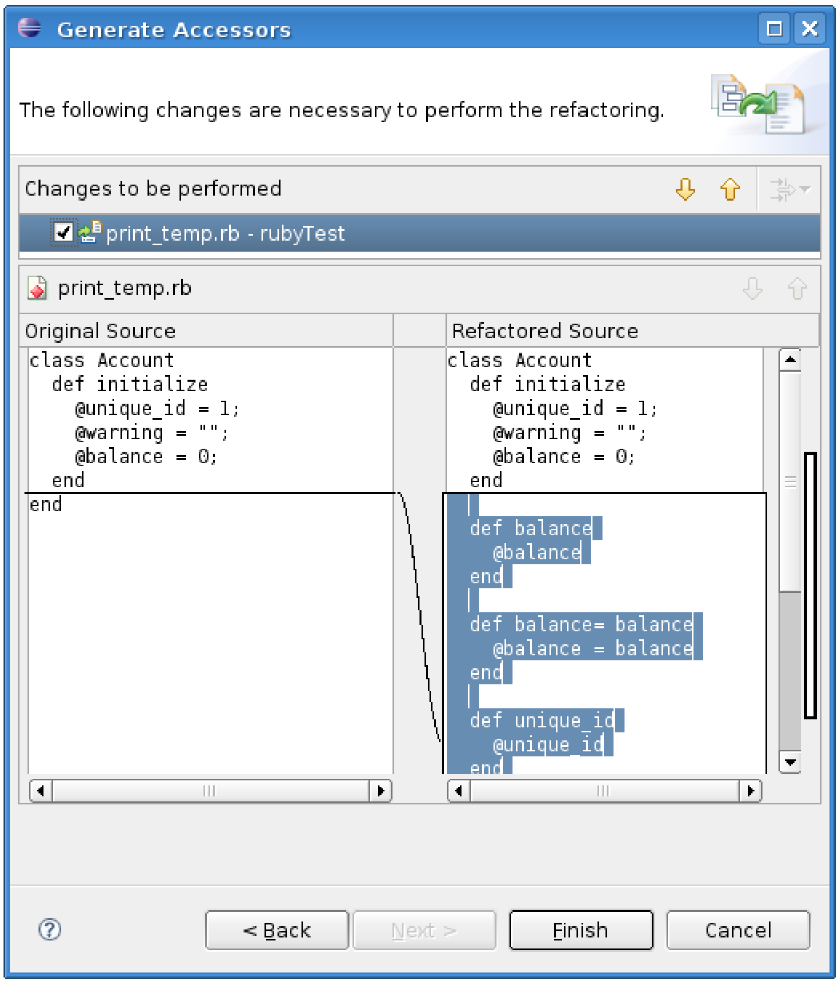
```
class Account
  def initialize
    @unique_id = 1;
    @warning = "";
    @balance = 0;
  end
end
```

Refactored Source:

```
class Account

  attr_accessor :balance

  attr_reader :unique_id

  attr_accessor :warning

  def initialize
    @unique_id = 1;
    @warning = "";
    @balance = 0;
  end
end
```

Figure 7.9: Generate Accessors Refactoring Wizard Simple Accessors

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



The following changes are necessary to perform the refactoring.

Original Source:

```
class Account
  def initialize
    @unique_id = 1;
    @warning = "";
    @balance = 0;
  end
end
```

Refactored Source:

```
class Account
  def initialize
    @unique_id = 1;
    @warning = "";
    @balance = 0;
  end

  def balance
    @balance
  end

  def balance= balance
    @balance = balance
  end

  def unique_id
    @unique_id
  end
```

Figure 7.10: 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 whose output will be 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 7.11.

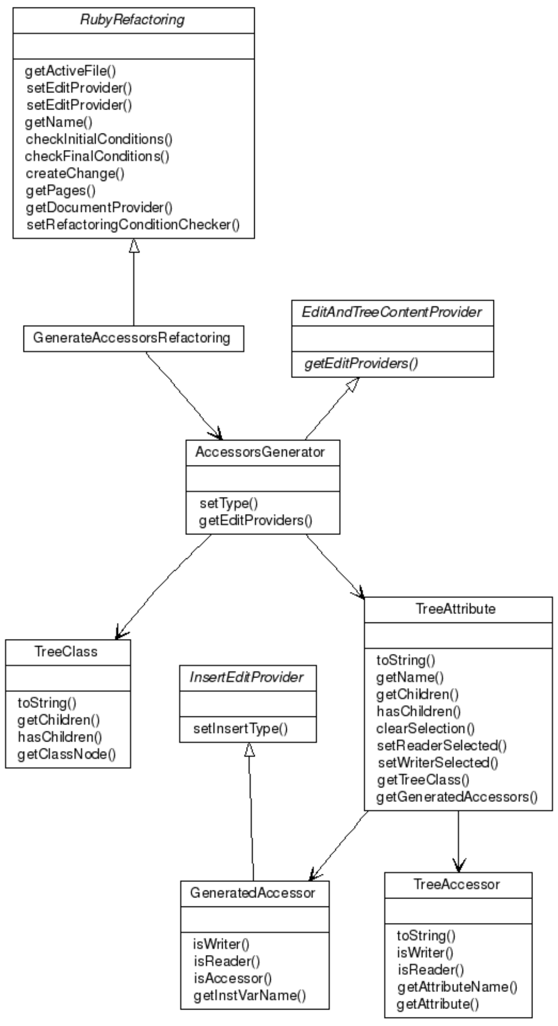


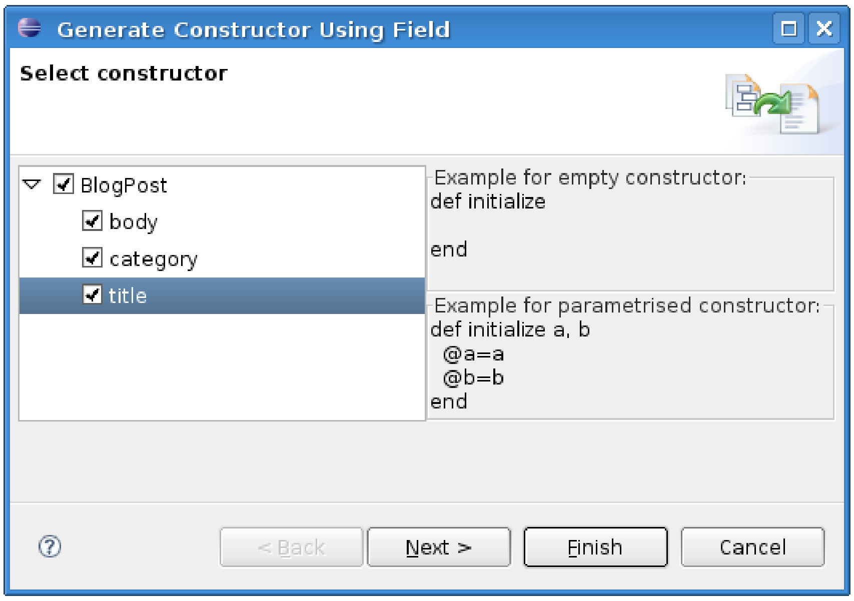
Figure 7.11: Generate Accessors Diagram

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 can check a class and several of the class's attributes. With this selection a constructor be generated.

Demonstration

Here you can see the user interface.


Generate Constructor Using Field

Select constructor

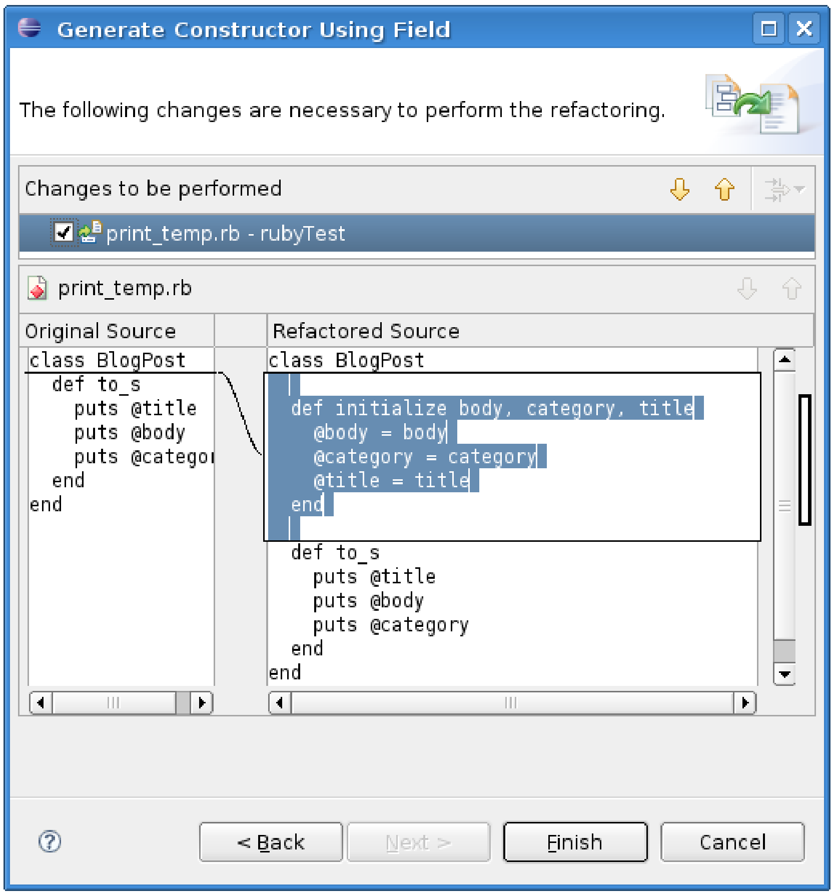
- [✓] BlogPost
  - [✓] body
  - [✓] category
  - [✓] title

Example for empty constructor:
```
def initialize

end
```

Example for parametrised constructor:
```
def initialize a, b
  @a=a
  @b=b
end
```

Figure 7.12: Generate Constructor Refactoring Wizard

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



```
class BlogPost
  def to_s
    puts @title
    puts @body
    puts @categor
  end
end
```

```
class BlogPost

  def initialize body, category, title
    @body = body
    @category = category
    @title = title
  end

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

Figure 7.13: 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 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 7.14.

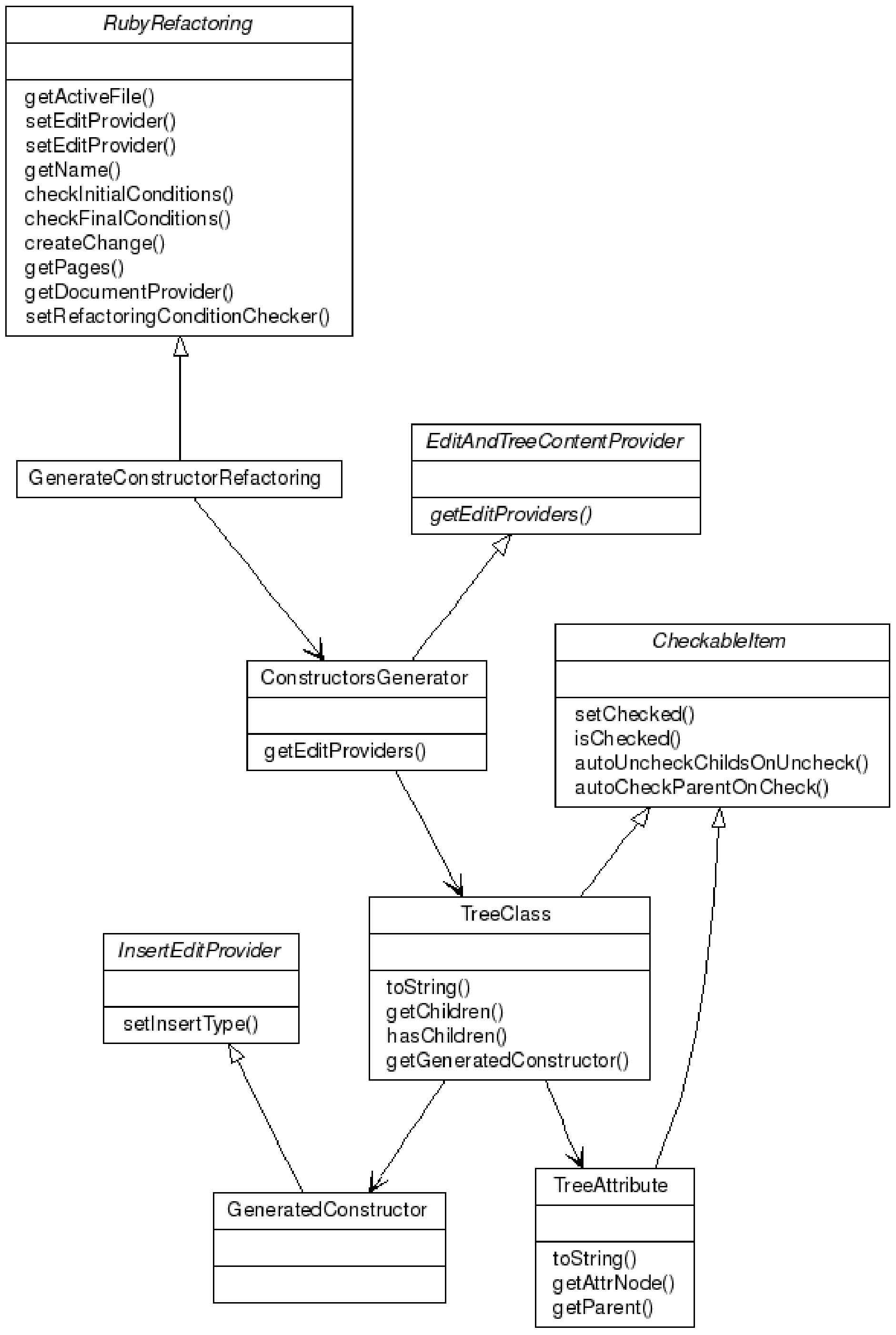


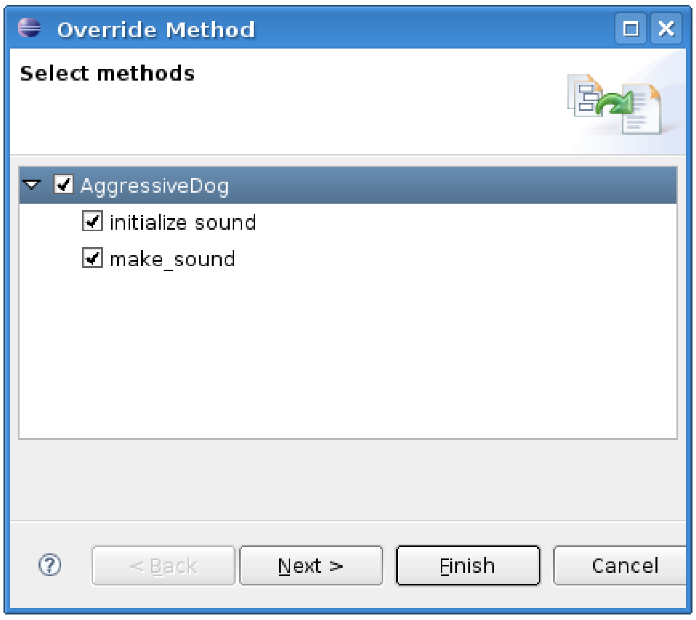
Figure 7.14: Generate Constructor Diagram

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.

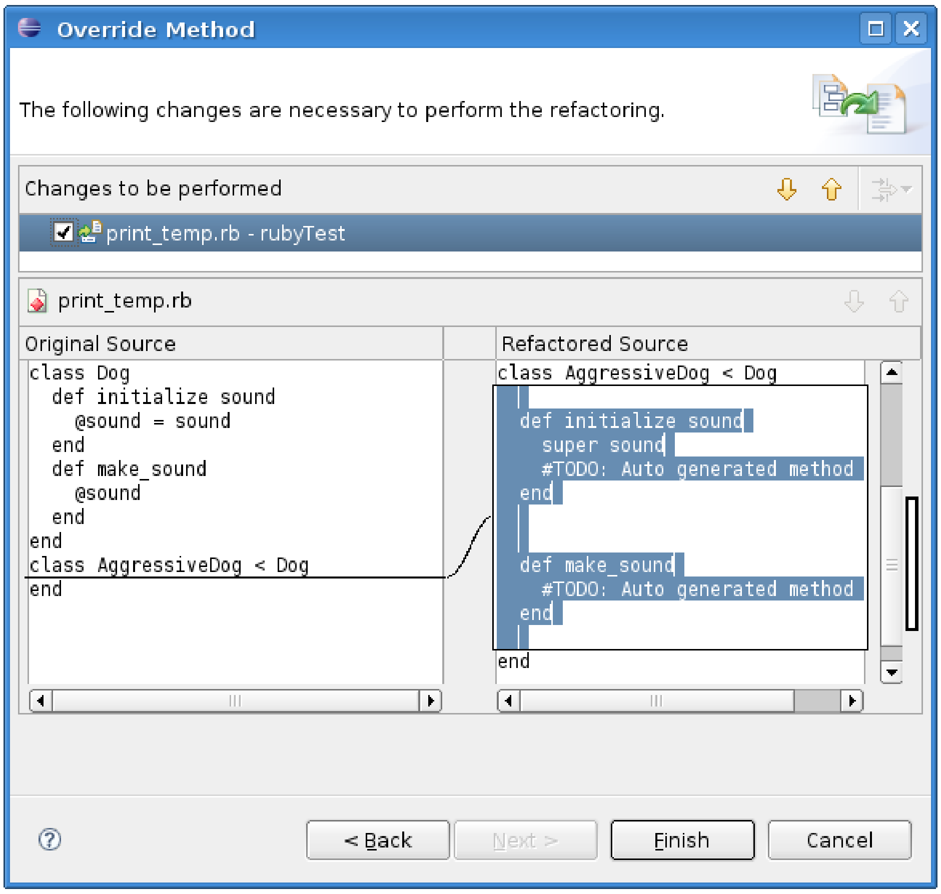

Override Method

Select methods

[✓] AggressiveDog
  [✓] initialize sound
  [✓] make_sound

Figure 7.15: Override Method Refactoring Wizard

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



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

```
class AggressiveDog < Dog

  def initialize sound
    super sound
    #TODO: Auto generated method
  end


  def make_sound
    #TODO: Auto generated method
  end

end
```

Figure 7.16: 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 IncludedClassNode-Provider. 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 each selected method an Overridden-Method 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 7.17.

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 Convert Local Variable to Field

With this refactoring a local variable in a method can be converted into an instance or class field. This is very useful when recognized that a value is needed beyond its current scope. Before invoking this refactoring the caret has to be placed in or beside a local variable. With a right-click Convert Local Variable to Field can be selected from the appearing context menu, in the refactoring section. Consequently the wizard is opened. It shows a text field where the name of the variable can be changed, which is especially necessary if a field with corresponding name already exists. It is possible to move the initialization of the field to the constructor of the class. With the Declare as Class Field flag the variable can also be converted into a class field.

Demonstration

The user interface can be seen in figure 7.18 and 7.19 shows the difference between the original source and the refactored source.

Procedure

The refactoring retrieves the caret position and searches for surrounding nodes which match the node types of local variables. If such a node can be found the enclosing method, scope and class are determined. The user can enter a new name for the field or leave it as it was, as long as its unique. Additionally the initialization location can be chosen, as it might make sense to have the field initialized in the constructor. After confirming the settings the refactoring checks the code for conflicts. If found, the user gets a report. Otherwise the changes are marked in a preview window where they can be accepted or not. When accepted the changes are applied to the source code.

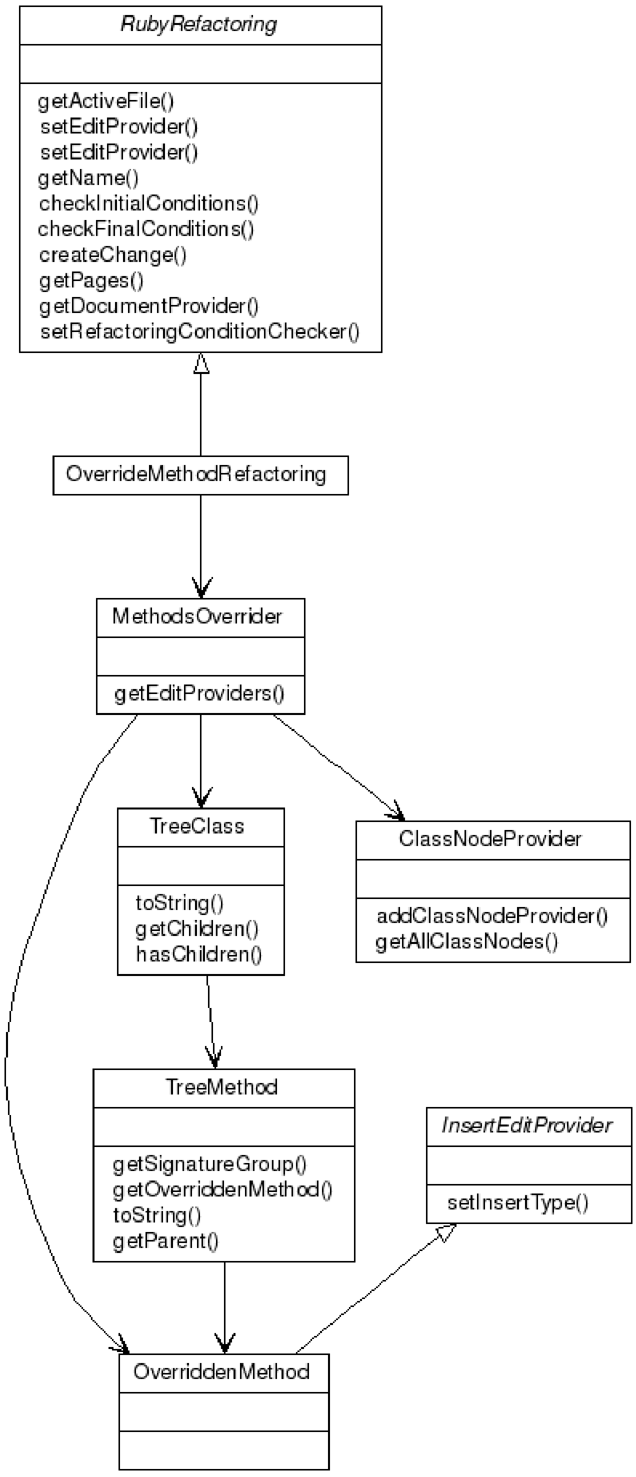
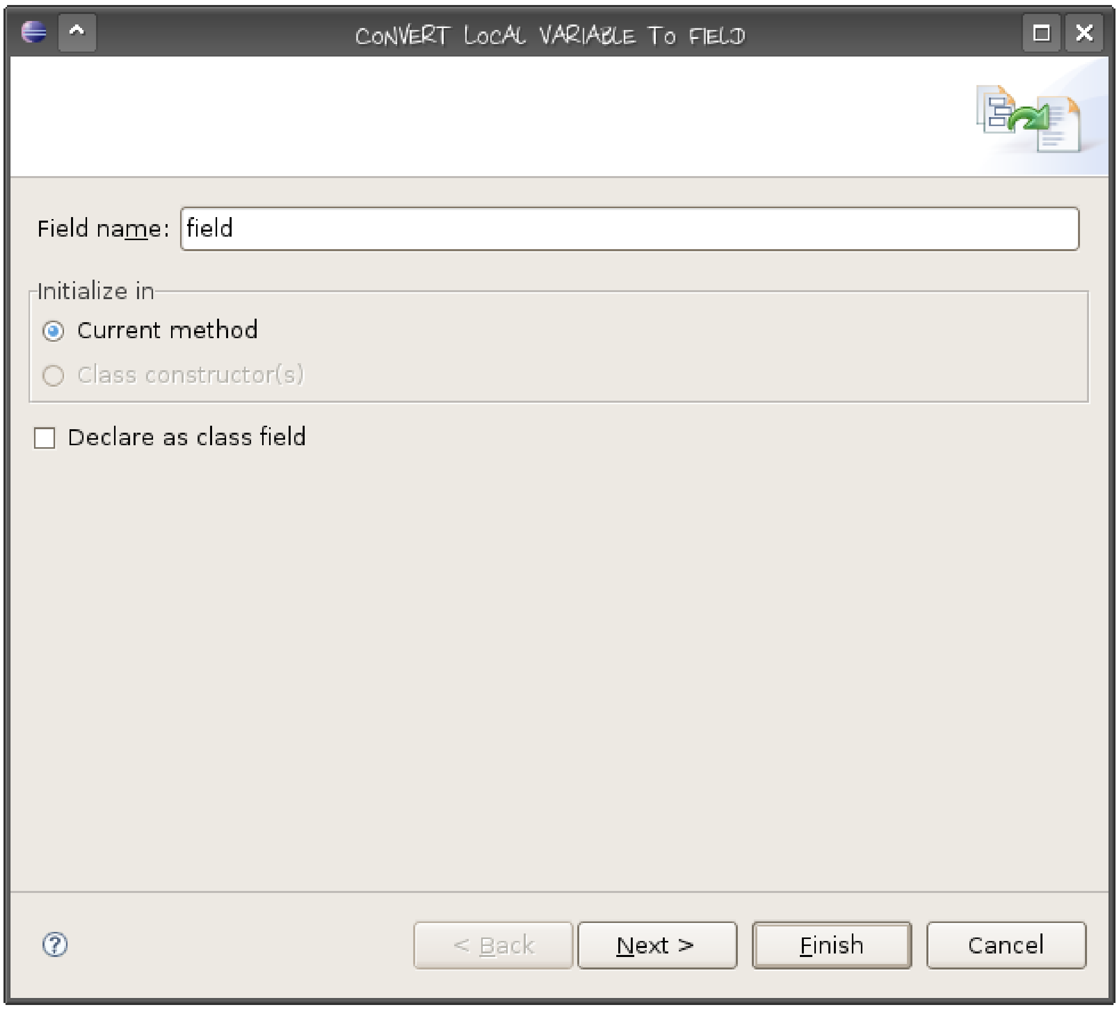


Figure 7.17: Override Method Diagram

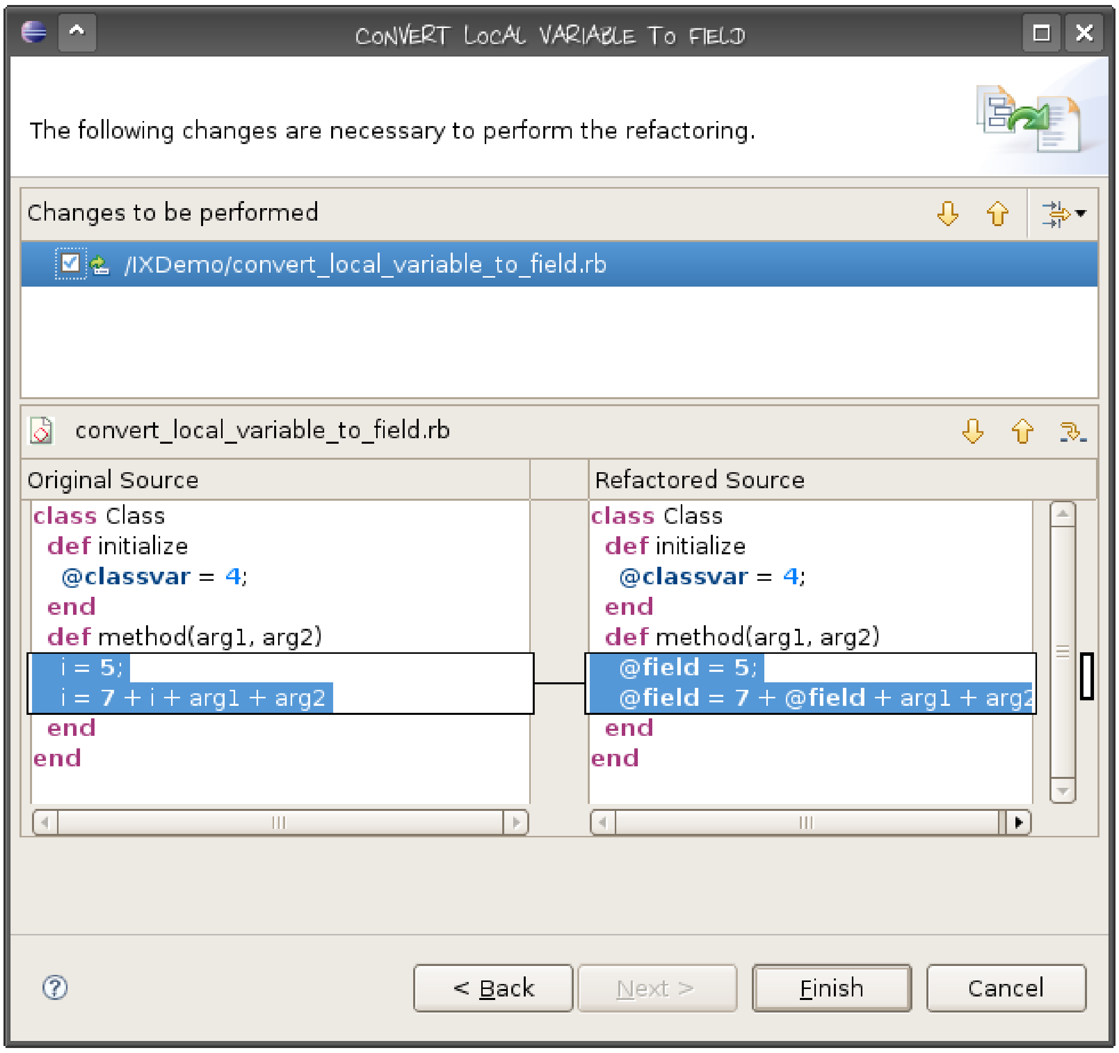


Field name: field

Initialize in
- [✓ Current method]
- Class constructor(s)

☐ Declare as class field

Figure 7.18: Convert Local Variable to Field Wizard



The following changes are necessary to perform the refactoring.

/IXDemo/convert_local_variable_to_field.rb

convert_local_variable_to_field.rb

Original Source

```
class Class
  def initialize
    @classvar = 4;
  end
  def method(arg1, arg2)
    i = 5;
    i = 7 + i + arg1 + arg2
  end
end
```

Refactored Source

```
class Class
  def initialize
    @classvar = 4;
  end
  def method(arg1, arg2)
    @field = 5;
    @field = 7 + @field + arg1 + arg2
  end
end
```

Figure 7.19: Convert Local Variable to Filed Refactoring Wizard Result

Class Diagram

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

Extensions and Ideas

The refactoring cannot yet handle arguments of methods as local variables which might be possible to handle too.
Classes which are split among several files are not checked for duplicates yet, which means that there still could occur an unrecognized conflict between the converted local variable and an existing class field in another files.

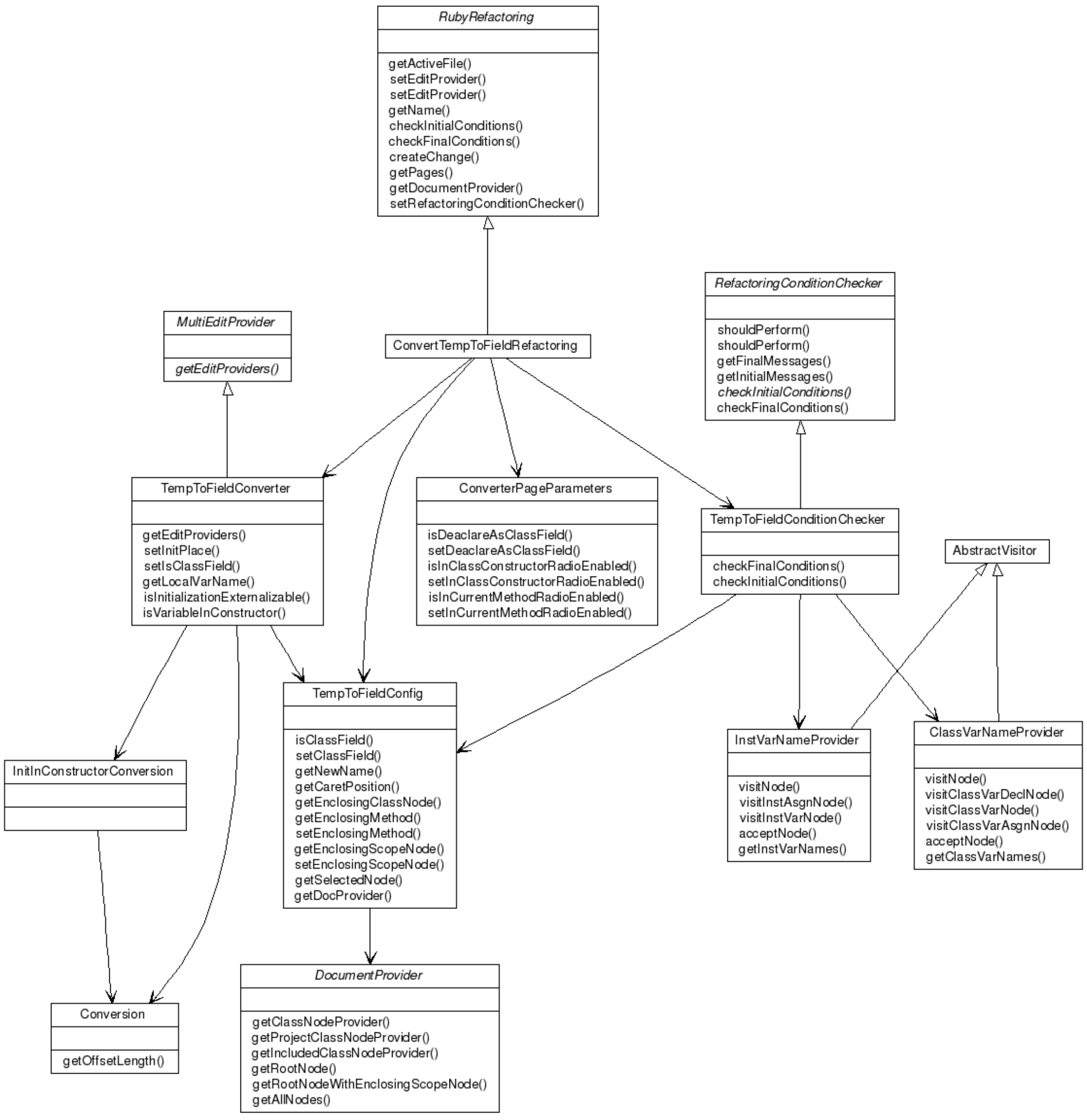


Figure 7.20: Convert Local Variable to Field Diagram

7.4.2 Encapsulate Field

This refactoring adds an encapsulation to a field. You can also allow a public access only to read or write the field. For this case the are the statements `attr_reader :field_name` for reading access or `attr_writer :field_name` for writing access. Depending on what type of access to the field is granted, the according accessor methods are generated.

Examples:

```ruby
class Integer


  attr_accessor :int_value



end
```

```ruby
class Integer
  def int_value
    @int_value
  end
  def int_value= int_value
    @int_value = int_value
  end
end
```

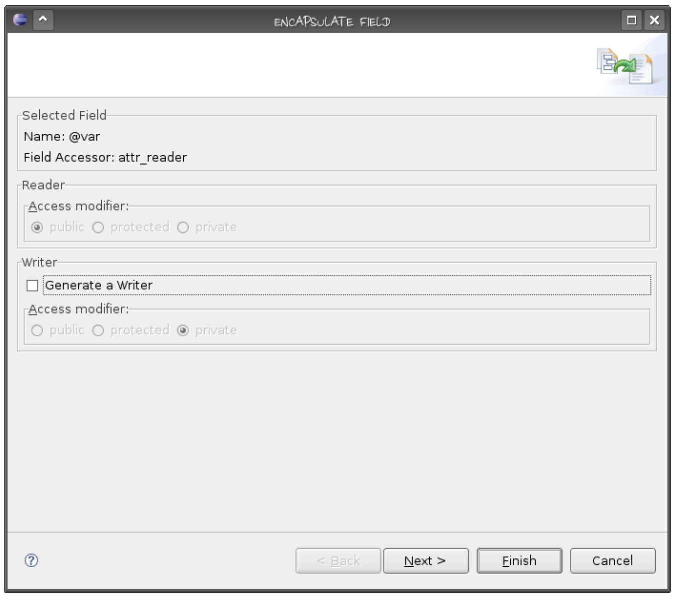
```ruby
class Integer

  attr_reader :int_value

end
```

```ruby
class Integer
  def int_value
    @int_value
  end
end
```

```ruby
class Integer

  attr_writer :int_value

end
```

```ruby
class Integer
  def int_value= int_value
    @int_value = int_value
  end
end
```

The user is also allowed to generate encapsulations for a field that has no public access. In this case he can choose which type of access he wants to grant (read or write) and how the visibility of the generated accessor methods will be.



Name: @var
Field Accessor: attr_reader

Access modifier: [✓ public] protected private

Generate a Writer

Access modifier: public protected [✓ private]

Figure 7.21: Encapsulate Field Wizard

Demonstration

Here the interface that will be shown to the user can be seen.

Procedure

The first thing to do is to evaluate the selected field at the caret position. The user can select the usage or an assignment of the field itself or select the symbol after an `attr_xxx` statement. Then the enclosing class is evaluated and it is searched for existing simple accessors of the selected field. The user then gets a dialog displayed where he can see information about the generated code. If the selected field has only public reader or writer access, he can decide to add an accessor method for the access type that was not defined public yet. If this is done, the visibility of this method can be set too.
If there exists a method with the same name as an accessor method, which will be generated, a warning is displayed. After confirming, the user can check the changes on the preview page and let them be applied to the document.

Restrictions

To invoke the refactoring successfully there are the following requirements:

- The caret needs to be at a field.
- The selected field needs to be inside a class.
- The name of the accessor methods, that will be generated, should not exist in the active class.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.22.

Testing

The tests for the EncapsulateField refactoring are file driven tests (see 8.1.1). Here follows a description of the properties that can be set for a test.

- cursorPosition: Integer - Position of the caret in the active document.
- enableReaderGeneration: Boolean / Optional - Enables generation of a reader accessor method.
- readerVisibility: private, protected, public / Optional - Visibility of the generated reader.
- enableWriterGeneration: Boolean / Optional - Enables generation of a writer accessor method.
- writerVisibility: private, protected, public / Optional - Visibility of the generated writer.

Challenge

This refactoring has not been a big challenge since we allready got experience in generating accessor methods and removing other parts from files.

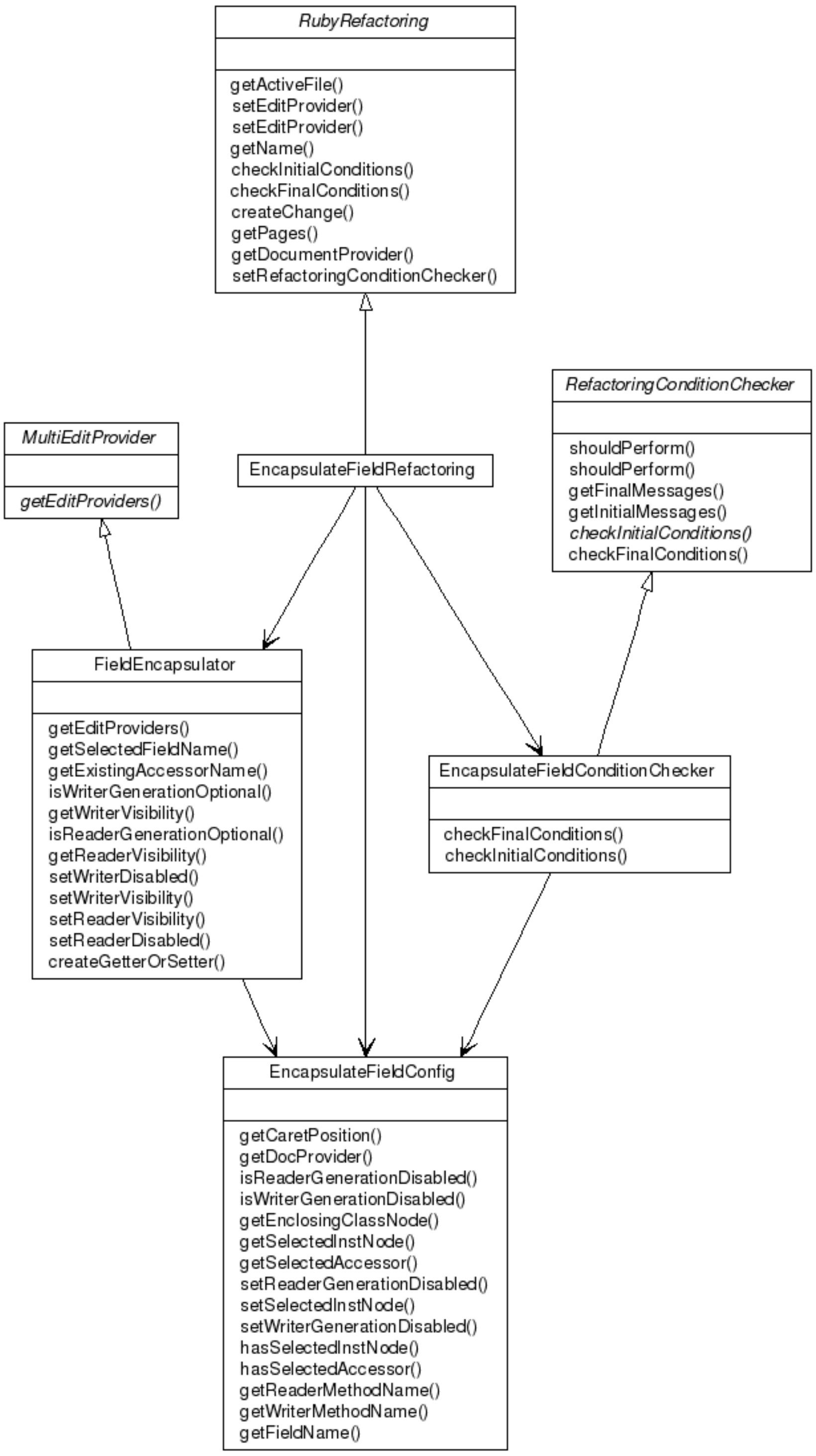


Figure 7.22: Encapsulate Field Diagram

7.4.3 Extract Method

The extract method refactoring is one of our favorites. It is very handy and often applicable if the code is getting unreadable. To extract a method, the user has to select a part of the source code. The selection is automatically widened to capture complete statements, for example, when an else statements is marked, the selection is expanded to contain the whole if-statement. The user also has to provide the name of the new method, and can rename and rearrange the parameters of the new method. If the new method creates local variables that are used after the extracted code, these variables are returned from the created method. In Ruby, such multiple value return statements are possible. Like this the Ruby Extract Method is even more useful than an Extract Method refactoring in another languages.

Demonstration

The main user interface can be seen in figure 7.23.

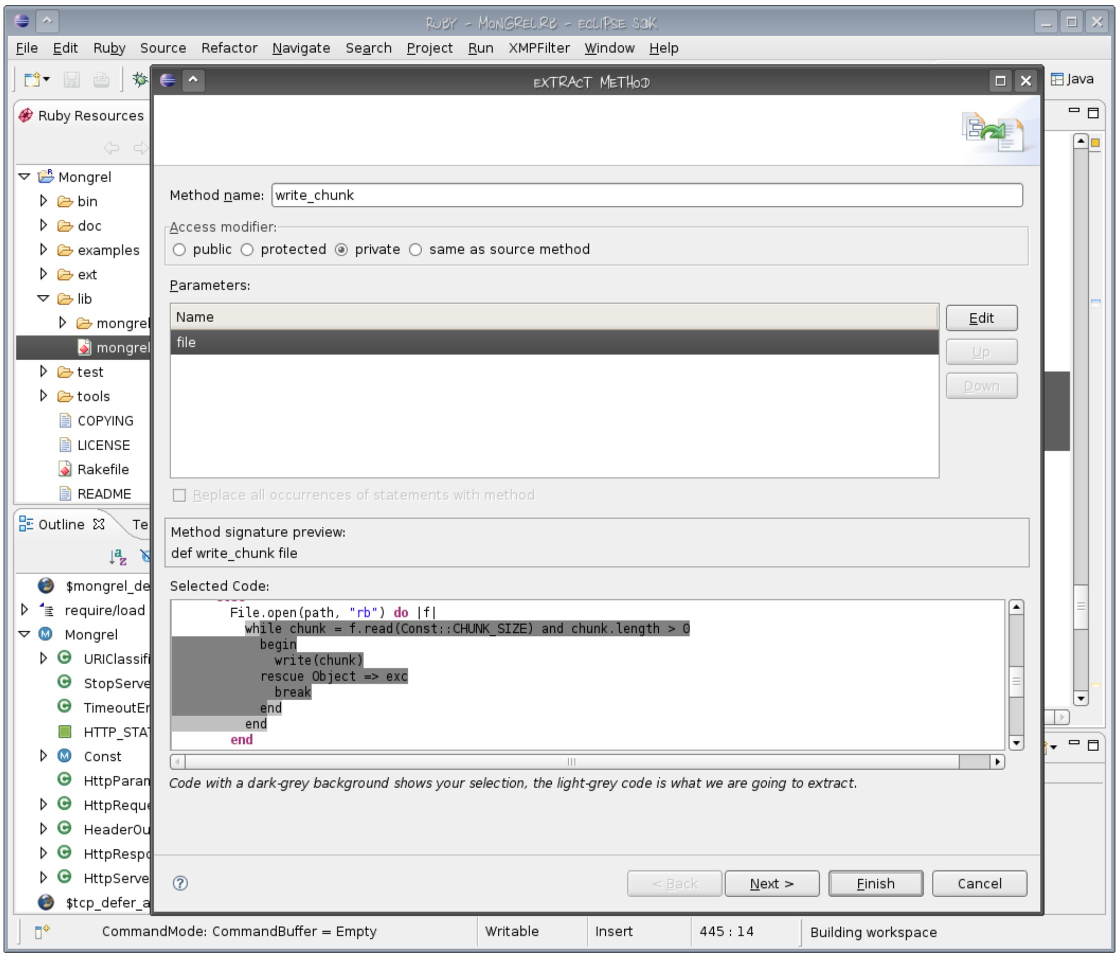
Procedure

After the user selected the code which he would like to get extracted and invoked the Extract Method refactoring, the selected statements are evaluated. To make the selection of extractable code easy we created a component, which expands the selection until it contains a whole node or multiple nodes on the same level.

Now all the local fields in the selected code are analyzed one after the other. The Extract Method refactoring builds two collections. One that contains all the local variables that need to be passed into the extracted method as arguments, and the other one that contains those responsible for the return value of the extracted method.
The argument collection is filled as follows. When a local variable is encountered the first time and this occurrence is not a local assignment, this local variable is needed as an argument
The local variable is added to the return values collection if the local field is assigned in the selected code and if it is used after the selected code.

After both collections have been filled, the user is shown the Extract Method page, where he can set the name of the extracted method and rearrange and rename its arguments.
When the user has finished, two text edits are prepared. The first is a replace text edit that replaces the selected code with a call to the extracted method. The second is an insert text edit which inserts the new method into the document.

At last the user can see the changes on the preview page and apply them to the document by finishing the refactoring.


Ruby - MongrelRB - Eclipse SDK
File Edit Ruby Source Refactor Navigate Search Project Run XMPFilter Window Help
Extract Method
Ruby Resources
Outline
Edit
Up
Down
&lt; Back
Next &gt;
Finish
Cancel

Method name: write_chunk

Access modifier: public protected [✓ private] same as source method

Parameters:

Name
file

Replace all occurrences of statements with method

Method signature preview:
def write_chunk file

Selected Code:

```
    File.open(path, "rb") do |f|
      while chunk = f.read(Const::CHUNK_SIZE) and chunk.length > 0
        begin
          write(chunk)
        rescue Object => exc
          break
        end
      end
    end
```

Code with a dark-grey background shows your selection, the light-grey code is what we are going to extract.

CommandMode: CommandBuffer = Empty   Writable   Insert   445 : 14   Building workspace

Figure 7.23: Extract Method Wizard

Restrictions

To invoke the refactoring successfully there are the following requirements:

- The user needs to select at least one statement.
- No calls to super are made in the selection.
- If the selected code is inside a class, it also needs to be in a method.
- No yield statements are selected.
- If the extracted code is inside of a method, this method must not contain nested methods.
- The selected code must not contain methods or classes.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.24.

Testing

Refactoring Evolution

Since we are often using the extract method from JDT, one thing that often annoyed us was that the selection has to be very precise. We wanted to change that for our implementation and are accepting a more fuzzy selection and just enlarge it. Because of this automagically enlargement, we needed a way to show the user what we are actually going to extract before it is shown in the compare view. That is why we use a read-only editor view in the dialog and highlight the marked and expanded areas of the source code.

Challenge

Extensions and Ideas

We already have an editor view in the refactoring dialog. It would be very useful if it allowed to change the selection without restarting the whole refactoring. We think that this would be really useful for the user.
A quite interesting topic is the finding and extracting of similar methods. It would be great if we could realize that.

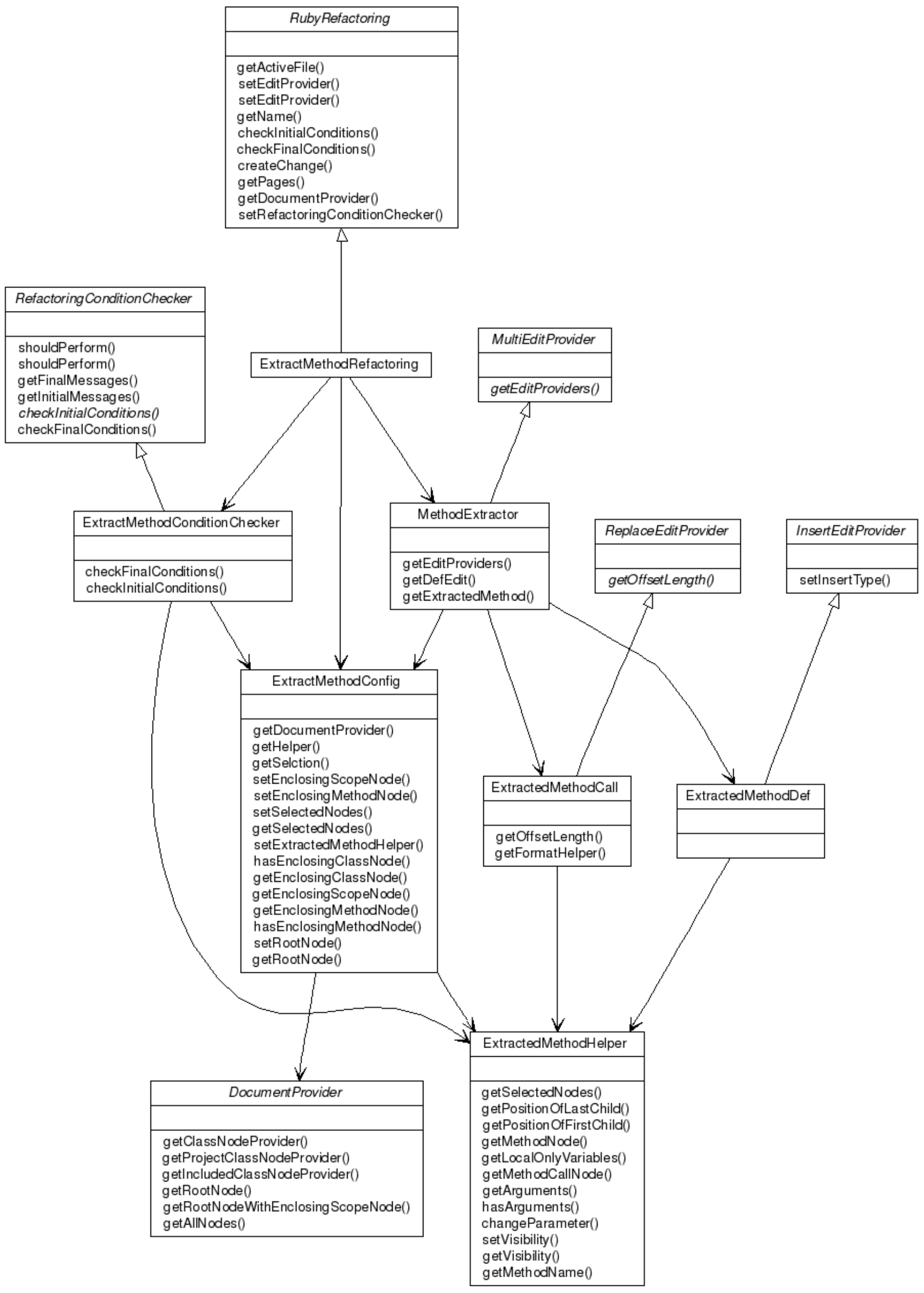
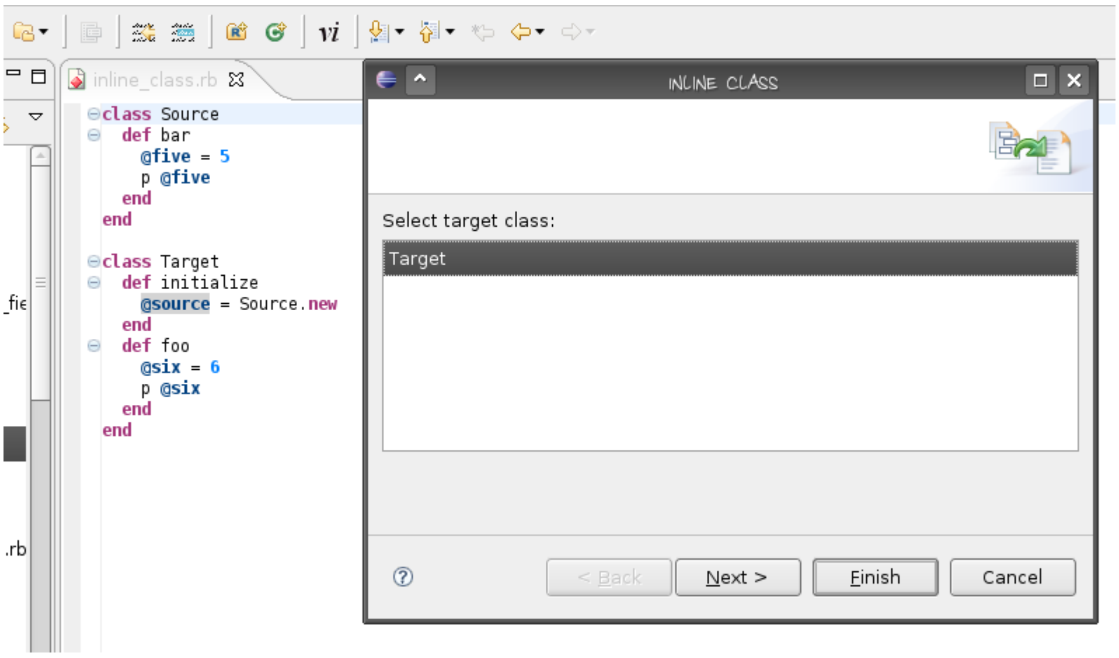


Figure 7.24: Extract Method Diagram



```
class Source
  def bar
    @five = 5
    p @five
  end
end

class Target
  def initialize
    @source = Source.new
  end
  def foo
    @six = 6
    p @six
  end
end
```

Select target class:

Target

Figure 7.25: Inline Class Wizard

7.4.4 Inline Class

The Inline Class refactoring integrates a class into another one which is using the class. Usually the inlined class has not much functionality. That sometimes occurs during the process of refactoring. The class normally is integrated into another class that is using it. This is not yet assured by the refactoring. Thus a class can be inlined into any other class.
The refactoring merges the classes together. It checks if there would occure a name-conflict between fields or methods, which are automatically solved. The constructor of the inlined class is renamed in <classname>_initialize. This new method will be called instad of the <classname>.new calls in the constructor of the target class.
The inlined class must not be split apart. To inline a splitted class it has to be merged first.

Demonstration

In figure 7.25 the interface that will be shown to the user can be seen.

Procedure

First a class definition is sought at the caret position. If found it is checked whether this class is multiparted, as classes that are split cannot be inlined. Then all names of fields and methods are collected. The user gets a selection page displayed in the wizard, where he can select the target class and if the constructor shall be moved. Only classes that use

the inlined class, or namely classes that create an instance of the inlined class in their constructor and assign it to a field. We expect this behavior to be right, as there are few other cases where this refactoring would be right to be applied.
After the selection the refactoring checks for name conflicts. For every conflicting item a new name is generated. By using Rename Method and Rename Field the conflicting names are refactored in a temporary internal document. The field used to keep the instance of the inlined class gets <self> assigned as all calls targeting the inlined class are now available locally. At last edits are created for the target class to insert the new methods and fields.

Restrictions

To invoke the refactoring successfully there are the following requirements:

- A class has to be selected.
- The selected class must not be multiparted.
- The selected class must not be subclasses
- The selected class must be used at least in one other class in the project.
- The Target class must create an instance of the inlined class in its constructor an assigns it to a field.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.26.

Testing

- caretPosition: Integer - The position of the caret in the active file. This position is used to find the inlined class.
- targetClassFile: String - The target file that contains the class or class part which the source class is inlined into.
- targetClassPos: Integer - A position in the targetClassFile-file to determin the target class (part).

Refactoring Evolution

The refactoring was intended to be implemented in the middle of the project. Due to dependencies to the rename refactorings we decided to postpone it. For Inline Class we did not have a JDT refactoring as template. So we had no hints how the refactoring should be presented to the user. First we just implemented the stupid basics of the rafactoring which included the merging of the bodies and the check for name-conflicts. In the last days we extended the refactoring for the checks and replacements of the constructor calls of the inlined class, which seem quite usefull and necessary to us.

Challenge

The inline class refactoring relies on the Rename Field and Method refactorings. It is important to provide all information needed by the subrefactorings to support their functionality. The toughest part was the adaption which requires to find a field that is using the inlined part and to find and replace all occurrences. Furthermore there is the inclusion of the constructor in the target class.

Extensions and Ideas

Perhaps it would be nice to enable to inline the class when it is not instantiated in the constructor but in any other method. This is not realized yet. A very challenging feature would be the inlining of a subclasses structure into another one. To be able to do this there has to be a target structure which is congruent to the inlined structure. We are not sure if this is really realizable.

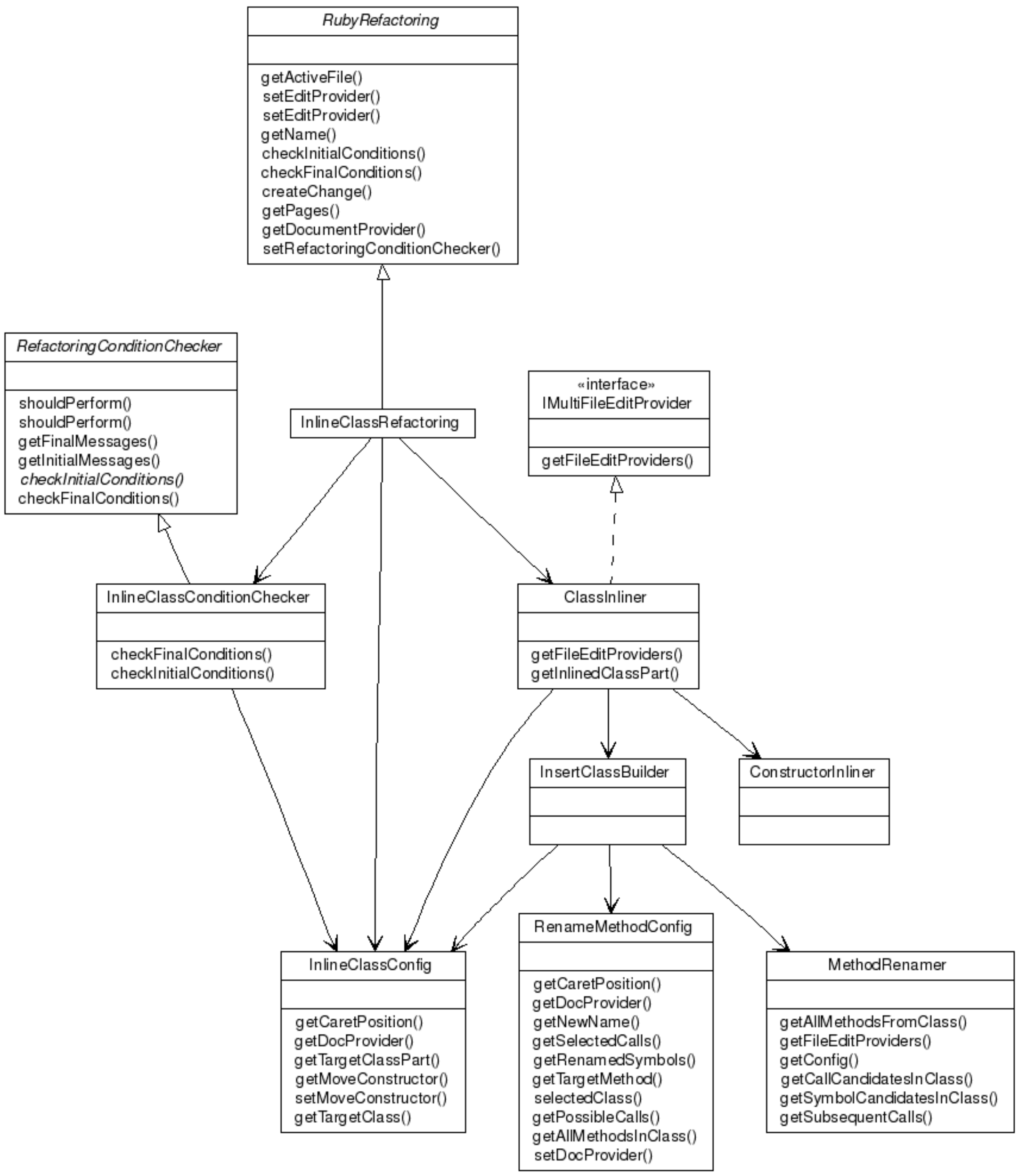
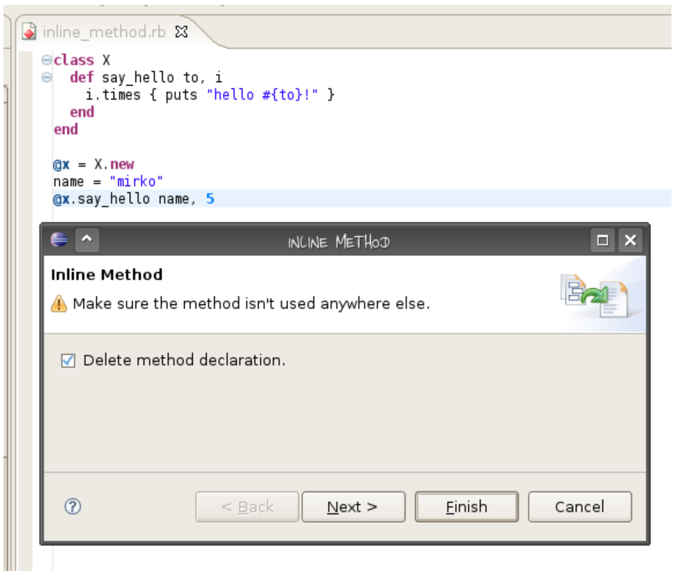


Figure 7.26: Inline Class Diagram



inline_method.rb

```
class X
  def say_hello to, i
    i.times { puts "hello #{to}!" }
  end
end

@x = X.new
name = "mirko"
@x.say_hello name, 5
```

Inline Method

Make sure the method isn't used anywhere else.

☑ Delete method declaration.

Figure 7.27: Inline Method Wizard

7.4.5 Inline Method

Inline method might not seem the most useful refactoring on the first glance, since why should you desire to clump your code together and maybe even duplicate it (if you do not remove the definition)? Inline method can be useful if you did other refactorings first and have methods that do not really do anything anymore and are just delegating calls. Then it might be useful to inline the method.

Demonstration

Here the interface that will be shown to the user can be seen.

Procedure

The inline method refactoring starts from a selected method call. The first thing to do is to find out the receivers type. If we do not have a receiver, the case is quite simple and we search for the method definition in the surrounding class. But if we have a receiver, we need to guess the type of that reference. We do that by travelling backwards in the code and looking for assignments to the selected variable which are `new` calls on a class constant. You can see an illustration of these steps in figure 7.28. If this kind of poor type inference does not work, we just ask the user what class the method belongs to.
Once we have the class, finding the definition of the method the user selected is not very hard anymore. To prevent us from modifying the original definition, we cut out the definition and work on a copy of the source code. The next step is to match the arguments of the call to their corresponding parameters in the definition. If the user passes only variables to the call, we just need to rename them in the method definition.

In the case that the user calls the method with fixed arguments, for example an integer or a string, we first have to assign this value to a local variable:

```ruby
def log string
  ...
end


var.log "finished"
```

```ruby
def log string
  ...
end

string = "finished"
var.log string
```

Now we need to prepare the body of the method definition: all calls to self need to be replaced with the reference of the object.

```ruby
def log string
  self.log_to_file string
end

var.log "finished"
```

```ruby
def log string
  var.log_to_file string
end

var.log "finished"
```

We also eliminate the return call on the last line and assign it to the variable from the caller.

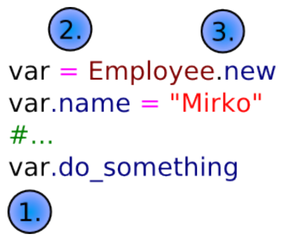
```ruby
def log two_plus_three
  return 2 + 3
end

result = var.two_plus_three
```

```ruby
result = 2 + 3
```

One problem that might occur is when the same variable names are used in the body of the caller and the method definition, then we just rename the names of the inlined methods, like JDT does, by adding or incrementing a digit at the tail of the name.

Restrictions

To invoke the refactoring successfully there are the following requirements:



```
var = Employee.new
var.name = "Mirko"
#...
var.do_something
```

Figure 7.28: Steps to find the type of a variable.

- A method call must be selected.
- We must either be able to guess the type of the variable on our own or get one from the user.
- The method has at the most one return statement at the end.

One thing the refactoring does not cover yet is the visibility of other methods that might get called from the inlined method. In this case, the user has to modify the inlined code himself.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.29.

Testing

Testing is done as usual, with corresponding source and result files.

- pos: Integer - The position in the source code where the caret should be, favorably on a method call.
- remove: Boolean - A flag to indicate whether the definition of the method should be removed or not.

Refactoring Evolution

Challenge

Inline method consist of several small parts, which can be addressed on their own and are not too hard to solve, like the renaming of parameters. In contrary to other refactorings and because of all these small changes, we could not work with change objects all the time and usually applied them to a temporary document from time to time.

Extensions and Ideas

To refine the refactoring, we could change the visibility of methods or generate accessors for fields which are accessed and not accessible anymore after the method has been inlined.

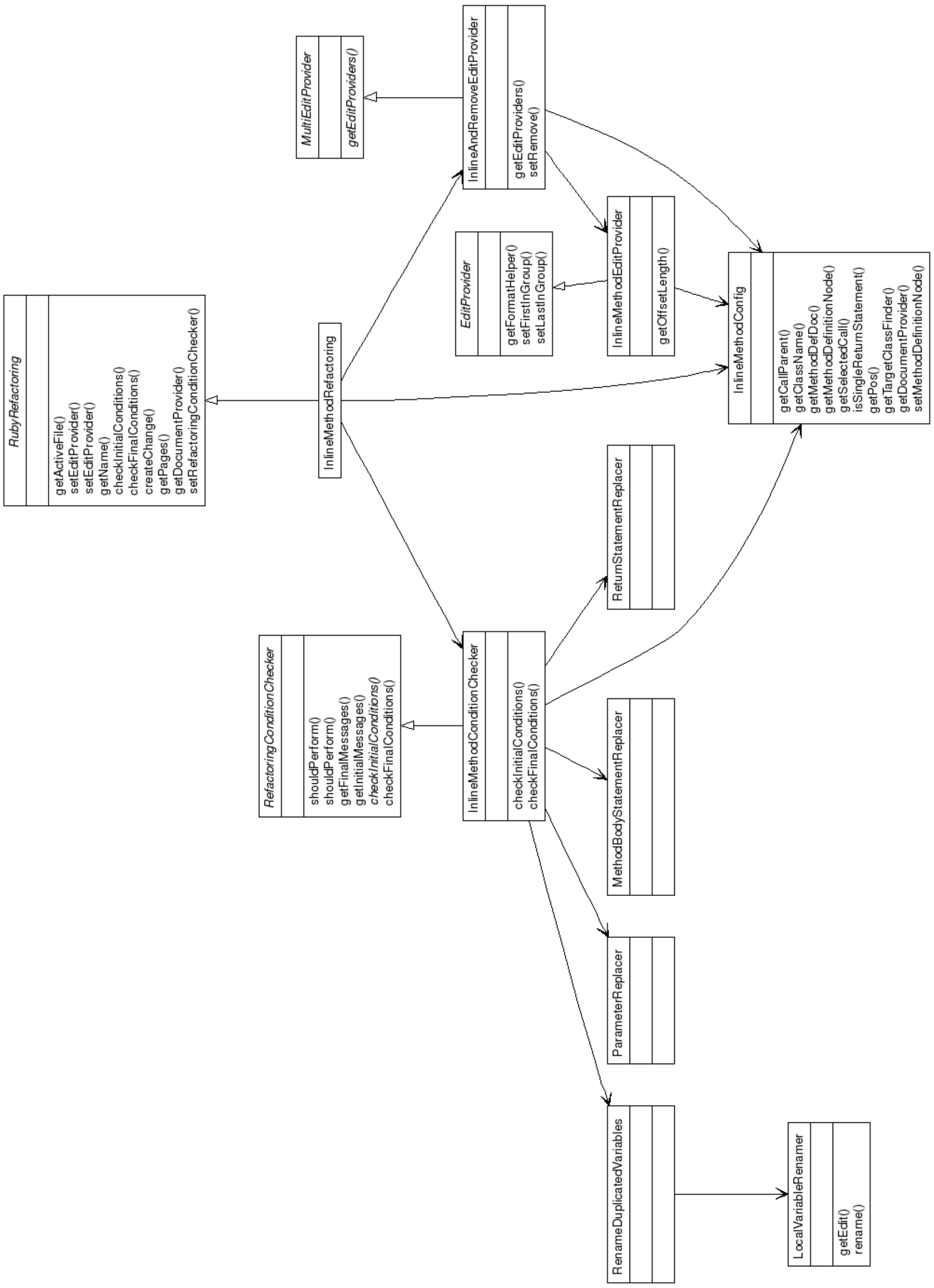


Figure 7.29: Inline Method Diagram

7.4.6 Inline Temp

The Inline Temp refactoring generally replaces all occurrences of a local variable with the value it got assigned. Usually this value is a simple one, for example a fixed number or a single method call. Actually the replacement of temps that have very complex mathematical operations and very long chains of invocation is possible too. But there is the question if this makes sense. For those times when it is really needed to have complex assignments inserted, there is the possibility to extract the whole value of the local variable into its own method. This results in functionality that belongs to Replace Temp with Query.
Inline Temp is especially helpful if one has to get rid of local variables to simplify other refactorings. A tool which has to be handled with care, as it might be possible to alter the functionality when the replaced value has side-effects. An issue that is yet up to the user to evade. Even in the Eclipse JDT this is not checked.
To invoke the refactoring one occurrence of the variable has to be selected with the caret. Through the context or drop down menu the Inline Temp refactoring can be started. A dialog is displayed that announces how many occurrences of the the variable have been found. With a check box the extraction of the assignment into its own method can be selected. By selecting the check box the text field is enabled, where the new name of the extracted method can be entered.

Demonstration

Here the interface that will be shown to the user can be seen.

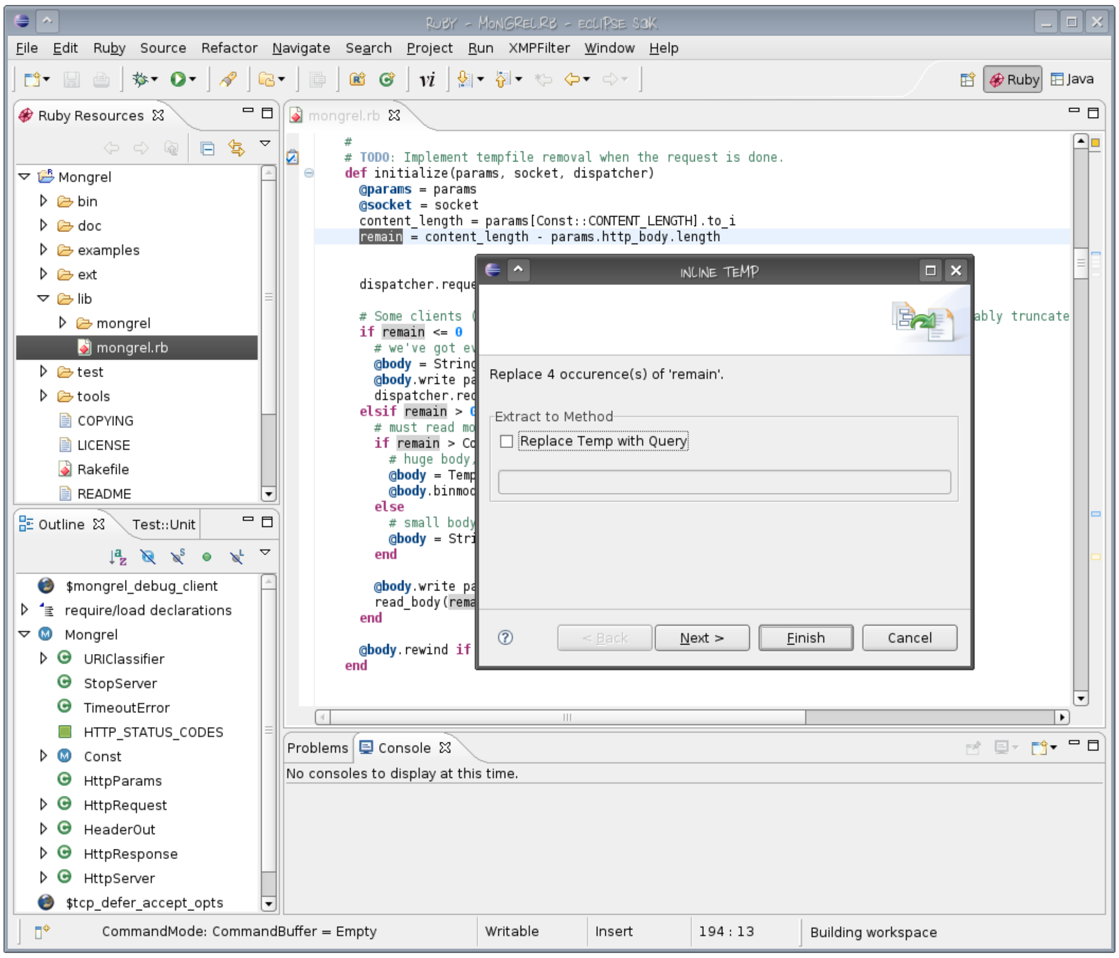
Procedure

When the refactoring is selected the caret position is checked. At the current position an item that represents a local variable (assignment or usage) is sought. When the whole local context is determined. Then any occurrence of the local variable is checked. There might be only exactly one assignment to the temp to proceed with the refactoring. The definition is determined and the value that is assigned retrieved. If the extraction of the value into its own method is selected, this method is created. All usages of the variable are subsequently replaced with either the value of the temp or the method call, depending on the user selection. Finally the definition of the local variable is removed.
Of course the chosen method name has to be a correct Ruby method name and must not already exist. If the value is a mathematical expression and is not extraced, the replaced occurrences have to be enclosed in brackets for not altering the functionality.

Restrictions

To invoke the refactoring successfully there are the following requirements:

- The caret has to be set in or beside a local variable.
- The local variable must not be a method parameter.



```
    #
    # TODO: Implement tempfile removal when the request is done.
    def initialize(params, socket, dispatcher)
      @params = params
      @socket = socket
      content_length = params[Const::CONTENT_LENGTH].to_i
      remain = content_length - params.http_body.length


      dispatcher.reque

      # Some clients (                                           ably truncate
      if remain <= 0
        # we've got ev
        @body = String
        @body.write pa
        dispatcher.rec
      elsif remain > 0
        # must read mo
        if remain > Co
          # huge body,
          @body = Temp
          @body.binmoc
        else
          # small body
          @body = Stri
        end

        @body.write pa
        read_body(rema
      end

      @body.rewind if
    end
```

Replace 4 occurence(s) of 'remain'.

Extract to Method

[ ] Replace Temp with Query

No consoles to display at this time.

CommandMode: CommandBuffer = Empty | Writable | Insert | 194 : 13 | Building workspace

Figure 7.30: Inline Temp Wizard

- There has to be exactly one assignment.
- Multiple assignments with temporary variables like “a, b = 5, 6” cannot be inlined yet.

The refactoring cannot recognize side effects which might be duplicated and alter the functionality somehow. This is up to the user. Such unwanted behavior is not expected to occur often, as in this cases there exist other design flaws in the code, that should not be or the refactoring is abused. In both cases the user is assumed to be fully aware of his deeds. To prevent the creation of unexpected changes there is always a preview for. When the result is not contempting it can be discarded or made undone completely.

Class Diagram

To get an overview over the class model of the Inline Temp refactoring take a look at the diagram 7.31.

Testing

The following parameters can be configured:

- caretPosition: Integer - The position in the source code where the caret should be. It is used in the refactoring to determine the local variable to inline.
- replaceWithQuery: Boolean - A flag to indicate whether the value of the variable should be extracted to a method.
- newMethodName: String - The caption for the method containing the extracted variable value. This property is only used when the replaceWithQuery parameter is set to 'true'.

Refactoring Evolution

First when we went through our list of refactorings for planning the diploma thesis there has not been an Inline Temp refactoring. We had Replace Temp with Query in our list. During the meeting with our professor we looked at our description of the refactoring which was completely weired. None of us recognized the intention by the explanation of this refactoring. Eventually we decided to replace this whateveritis-refactoring with an inline refactoring for temps. By re-specifying the refactoring we came across the Inline Temp refactoring description, which perfectly matched our intentions.
By the time we started to plan and design this refactoring for our plug-in we tried to find out what the real intention of Replace Temp with Query had been. As we just finished the implementation of Extract Method we recognized the easy extension of Inline Temp to provide the functionality of Replace Temp with Query. So this refactoring became the first that uses the functionality of another one.
Before implementing the Inline Temp we decided to relocate some core functionalities of the refactorings. We intended to uncouple the refactoring process from its invocation. In other words we assigned clear tasks to the RubyRefactoring-deriving, the page-displaying and the edit-providing classes. This refactoring became the first one implemented matching these definitions.

Challenge

Mathematical Terms As refactorings usually do not change the behavior of the application we intend not to do this either in our refactorings. While exploring the possible conflicts in using this refactoring, we came across mathematical terms. We encountered the following issue:

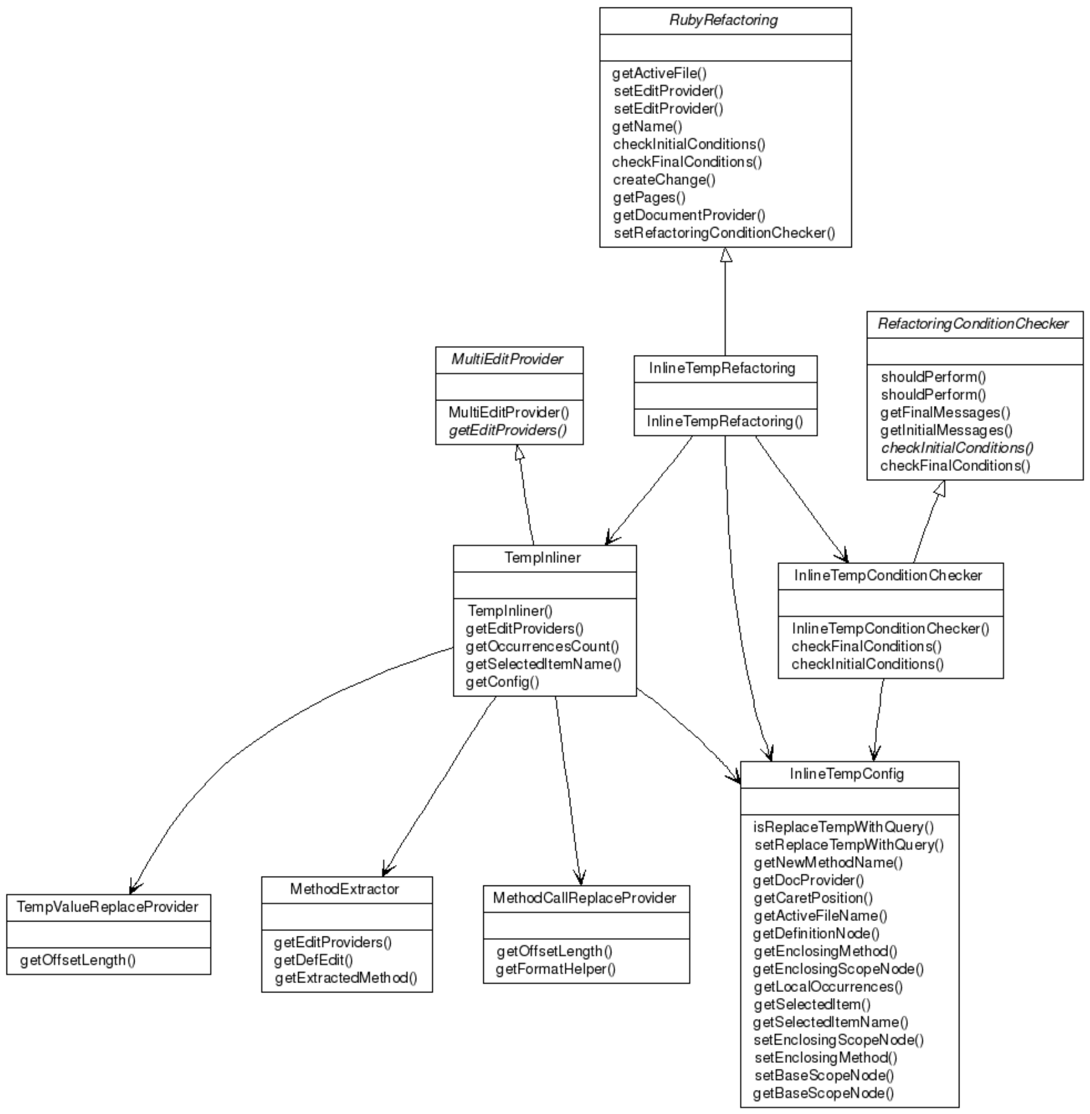


Figure 7.31: Inline Temp Diagram

Example Before inlining cSquare:

```
class Triangle
  def Triangle.hypotenuse a, b
    cSquare =  a ** 2 + b ** 2
    cSquare ** 0.5
  end
end

c = Triangle.hypotenuse 3, 4
p c

#output: 5.0
```

After stupid replacement:

```
class Triangle
  def Triangle.hypotenuse a, b

    a ** 2 + b ** 2 ** 0.5
  end
end

c = Triangle.hypotenuse 3, 4
p c

#output: 16.10...
```

Obviously this is not what the refactoring is supposed to do. The functionality is broken due to precedence rules of operators. To solve this issue we put brackets around the term. Subsequently the refactored code looks as shown below.
After replacement considering mathematical expressions:

```
class Triangle
  def Triangle.hypotenuse a, b

    (a ** 2 + b ** 2) ** 0.5
  end
end

c = Triangle.hypotenuse 3, 4
p c

#output: 5.0
```

This is now correct again. As it is not always right to insert brackets, they are inserted only when a mathematical operator exists in the variable value.

Method Call Brackets When extracting the value into a method or when having a method call as the variables value itself, then sometimes another problem occurred. In some cases the call argument assignment is ambiguous or generates syntax errors.

Example Before inlining squareRadius:

```
class Circle
  def printArea r
    squareRadius = r ** r
    p Math.PI * squareRadius
  end






end
```

After extraction and replacement:

```
class Circle
  def printArea r

    p Math.PI * squareRadius r
  end

  def squareRadius r
    r ** r
  end
  private :squareRadius

end
```

Here the refactoring has generated a syntax error. The expression 'p Math.PI * squareRadius r' is not valid as it is not clear where the 'r' belongs to. Hence we decided to introduce the brackets in all method calls in the ReWriter. So the refactoring now creates the following valid code:

```
class Circle
  def printArea r
    p Math.PI * squareRadius(r)
  end

  def squareRadius r
    r ** r
  end
  private :squareRadius

end
```

Extensions and Ideas

As an extremely complex an challenging extension an Inline Temp version that recognizes several parts of the local variable where it has a certain value. It might be solved by using the functionality of Split Temp. Eventually in every part the current value is inserted. Another interesting feature would be the recognition of side-effects when executing the value part of the temp several times. So that the user is shown a warning when the value part changes its result with every call, but we do not yet have a clue how to implement it. Furthermore the handling of multiple assignments should be possible.

7.4.7 Merge Class Parts

Merge Class Parts is a Ruby specific refactoring. As in Ruby a class may have parts in several different files, we thought it would be a comfortable feature if one can merge those parts together. There are two possibilities how classes can be split up. Either in one file that has several 'class'-parts or in different files by including the other parts in the extending file. For the least load or require calls are needed.
Merging class parts is used when the overview about a class gets lost as it is spread over too many parts. Or when an extension for a class is written which becomes more important for this class and should be integrated into the main part. To have a clear differentiation between the two refactorings, we split them up. Now there are two refactoring calls: Merge Class Parts in File and Merge with External Class Parts.
To run one of those refactorings there has to be something to merge in the current file. If that is the case the refactorings can be started from the context or pull down menu. Then the user can select the parts he wants to get merged. In the Merge Class Parts in File refactoring the user has to select the destination part, which the other parts are merged into.

Demonstration

In figure 7.32, the interface that will be shown to the user can be seen.

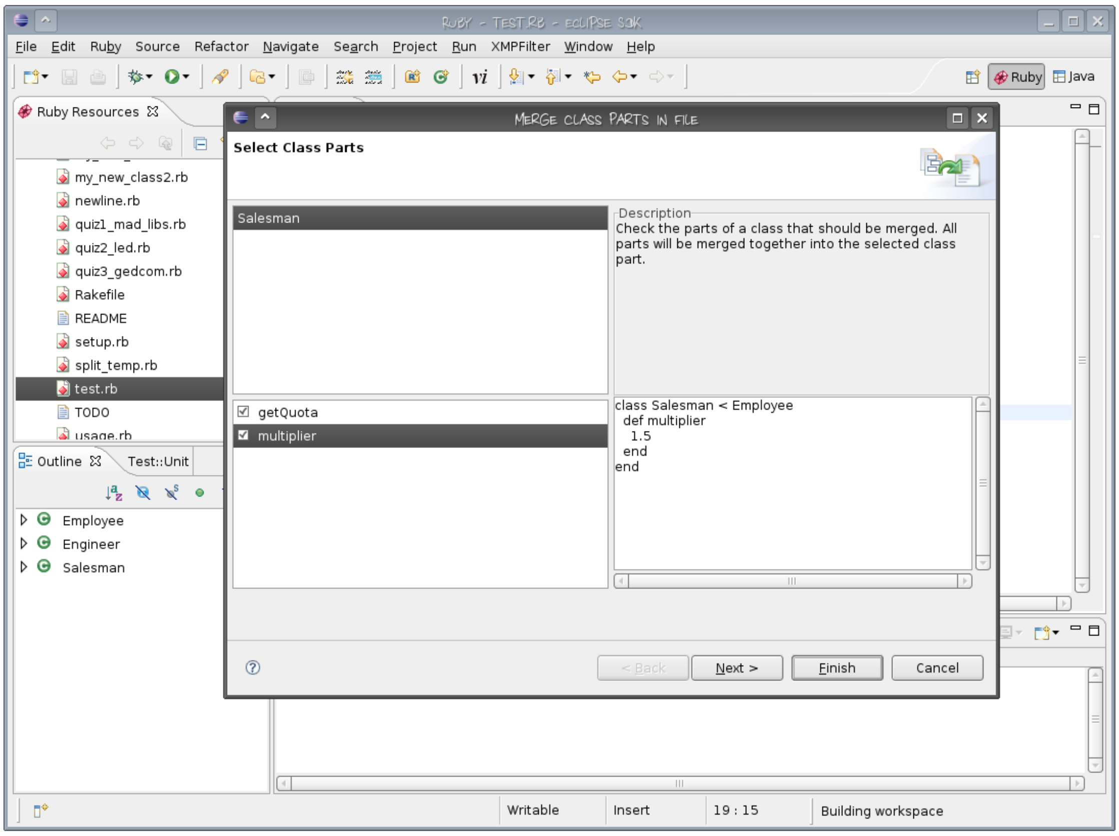
Examples

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

Procedure

In File To start this refactoring at least one class with two or more parts has to be defined in the current file. All class parts in the current file are scanned. A list with possible classes is generated. From this list one item can be selected. Subsequently all parts of the selected class are presented to the user. All parts that should be merged have to be checked. The target part is determined by the selection. After confirming the bodies of the checked class parts are inserted into the selected part. The remaining class definitions, which are not used anymore, are deleted.

With External Parts Invoking this refactoring needs inclusion of other files with the load or require command. The presentation is a bit different to the Merge Class Parts in File refactoring. Here all class parts are presented in a tree with check boxes. Every class part from the current file has subitems that represent other parts from the same class in different files. As the merged class part might be wanted to exist in more than just one included file, it is possible to place the merged part in more than one included file. This might lead to duplicated code and has to be used with care. After the confirmation the selected part bodies are cut out and pasted at the bottom of the target parts. The remaining class definition, which is not used anymore, is deleted.

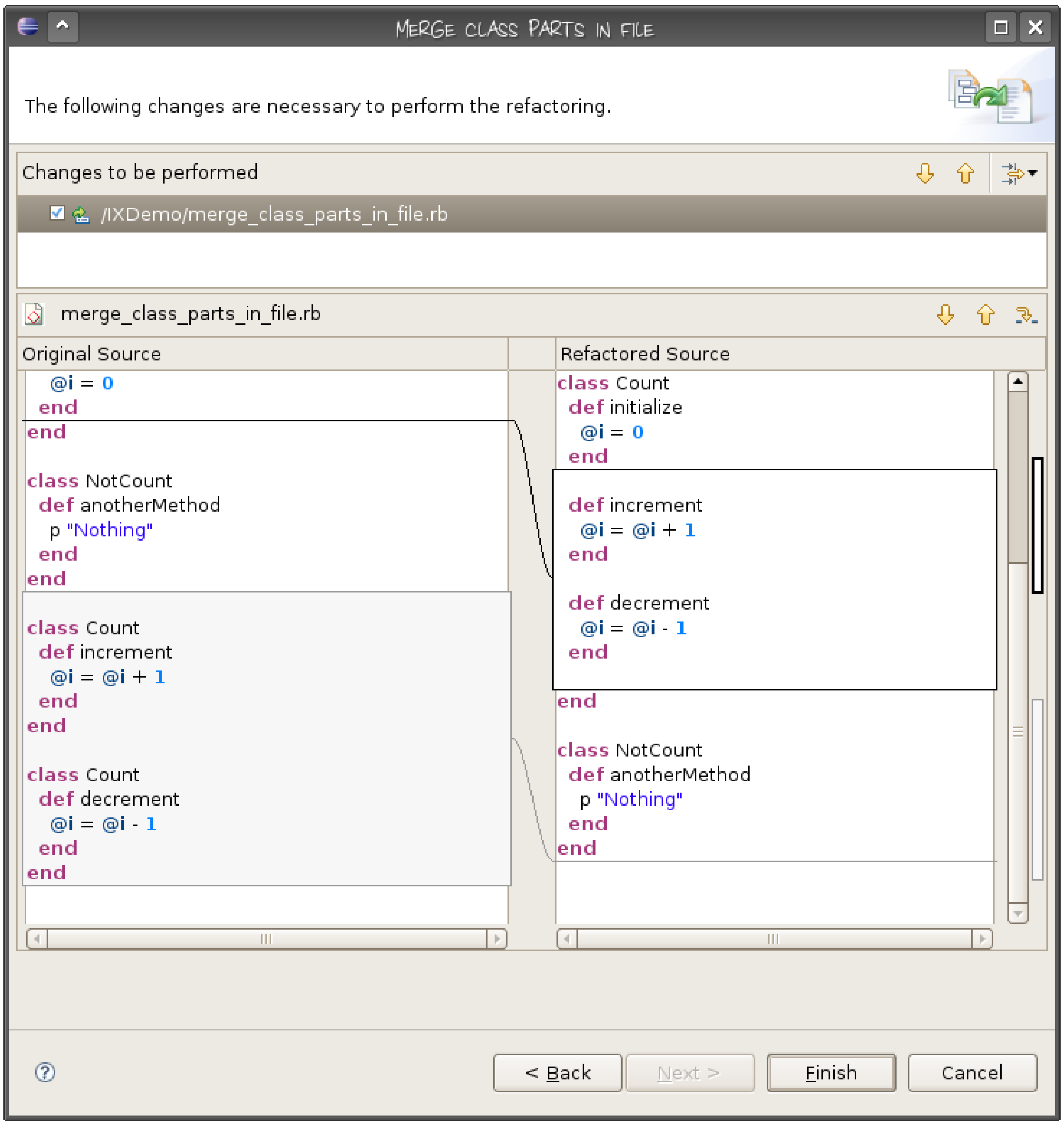


Check the parts of a class that should be merged. All parts will be merged together into the selected class part.

☐ getQuota
☑ multiplier

```
class Salesman < Employee
  def multiplier
    1.5
  end
end
```

Figure 7.32: Merge Class Parts in File Wizard



The following changes are necessary to perform the refactoring.

/IXDemo/merge_class_parts_in_file.rb

merge_class_parts_in_file.rb

Original Source

```
    @i = 0
  end
end

class NotCount
  def anotherMethod
    p "Nothing"
  end
end

class Count
  def increment
    @i = @i + 1
  end
end

class Count
  def decrement
    @i = @i - 1
  end
end
```

Refactored Source

```
class Count
  def initialize
    @i = 0
  end

  def increment
    @i = @i + 1
  end

  def decrement
    @i = @i - 1
  end
end

class NotCount
  def anotherMethod
    p "Nothing"
  end
end
```

Figure 7.33: Merge Class Parts in File Wizard Result

Restrictions

It might be possible that the inserted part of a class overwrites another one, defined before. If exactly this overwritten functionality is used between the two merged parts, then this refactoring alters the behavior. But it was never intended to prevent this. When using this refactoring we expect the user to write code which is not that dirty to profit from this ruby feature in that way.

In File To invoke the refactoring successfully there are the following requirements:

- At least one class has to be defined.
- There have to be at least two independent parts of this class.
- Both parts should have at least one method in their body.

With External Parts To invoke the refactoring successfully there are the following requirements:

- At least one require or load command is existing in the file.
- One class, that is accessible in the current file, has to be spread over two or more files.

Testing - In File

- targetClass: String - The name of the target class that should be merged in the refactoring.
- selectedPart: Integer - Since there could be several parts of the same class in a file, the part that should be merged has to be selected.
- checkedParts: Integer - The selected part can be merged with several other parts. These have to be chosen with the part numbers. This property is also comma-separated.

Testing - With External Parts

- targetClass: String - The name of the target class that should be merged in the refactoring.
- sourceClassPartNumber: Integer - Since there could be several parts of the same class in a file, the part that should be merged has to be selected.
- destinationPartNumbers: Integers - For every selected destination file in the destinationFiles property a target class part should be entered. That is done by entering the part number. This property is also comma-separated.

Class Diagram

To get an overview over the class model of the merge class parts refactorings take a look at the diagrams 7.34 and 7.35.

Refactoring Evolution

When creating our list of possible refactorings, we mainly oriented us at the JDT refactoring possibilities and the book 'Refactoring' by Fowler [Fow99]. But we also wanted to have something Ruby specific. The language provides lots of nice features. One of those seemed to be worth to implement a refactoring for. The possibility to dynamically add functionality to a class. We expect in large projects, that have classes spread over several files, it is quite hard to keep the overview. To support the developer in keeping his classes together we decided to implement a refactoring that lets class parts be merged.
By pouring over the design we realized that there are two different cases how to merge parts. Either there are class parts in different files or there are all parts in the same file. We decided to design one refactoring for each case to adapt the user interface specifically for each case.

Challenge

Merging class parts in different files was the first task in a refactoring where we had to apply changes to several files. This demanded completely new handling of the edits and the changes. In both refactorings the task is not quite easy to display to the user. So we had to think about several possibilities to show the class parts, the files and even the code to support the user in selecting parts. Thus we also decided to create two different refactorings as the interfaces look quite different.

Extensions and Ideas

It is Ruby specific to be able to have classes split over several files. There is the idea to do the exact opposite of this refactoring. A big class could be split up an spread over different files. This would be more an own refactoring than an extension. The Merge with External Class Parts refactoring now just works for the file that includes the other class part files. Probably it might be useful to be able to invoke the refactoring from the extended class, for pulling all parts in.

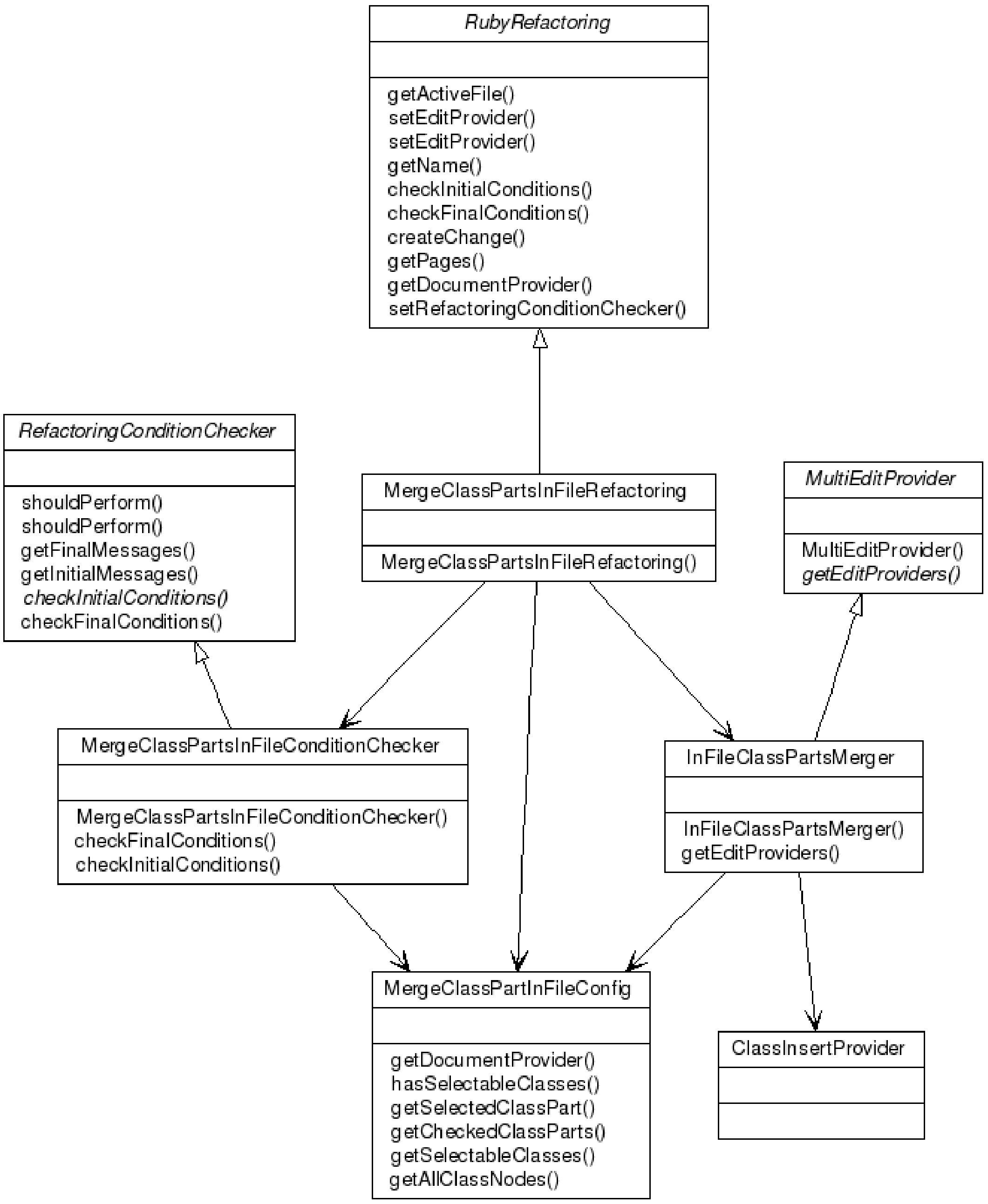


Figure 7.34: Merge Class Parts Diagram

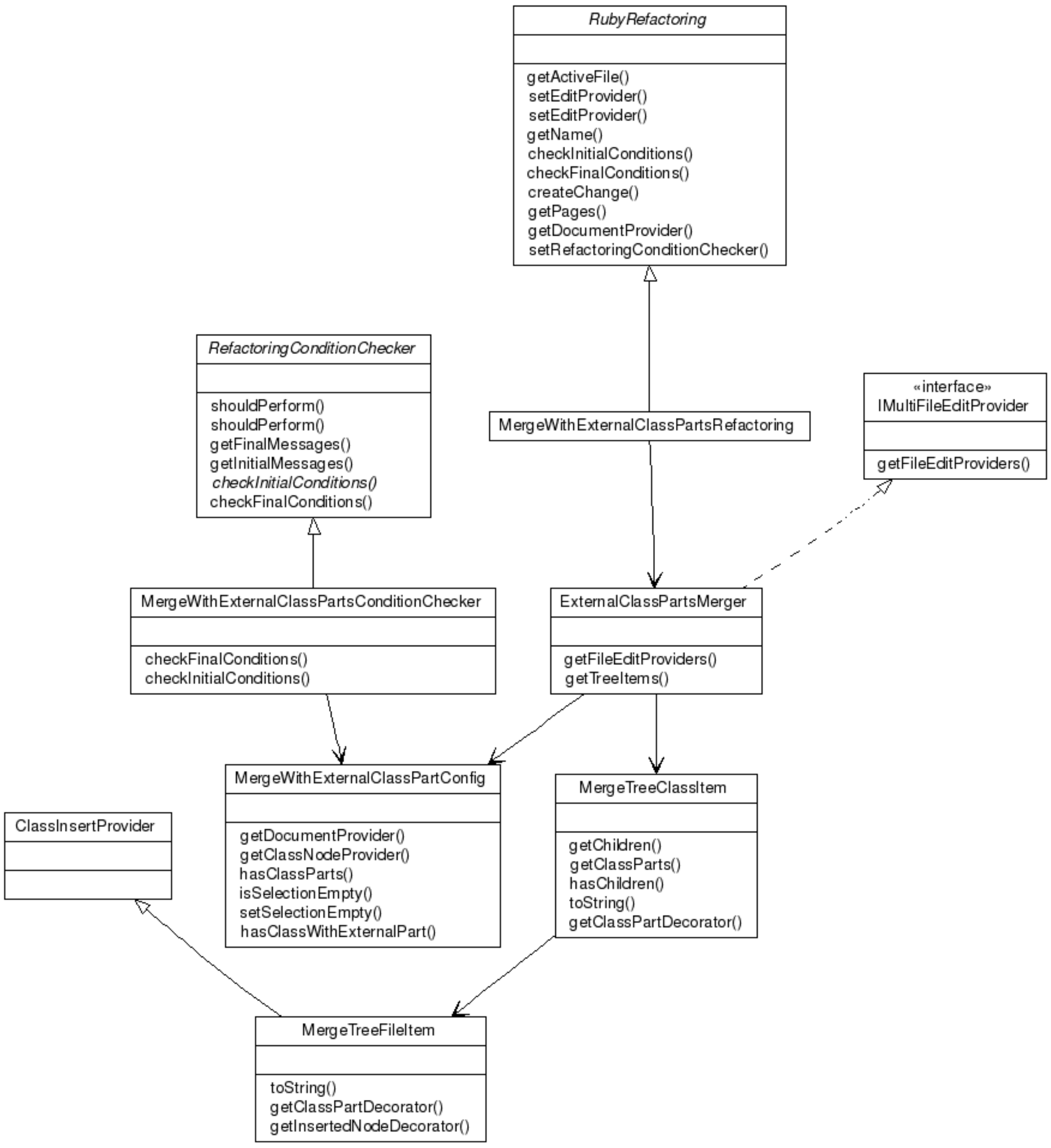


Figure 7.35: Merge Class Parts Diagram

7.4.8 Move Field

The Move Field refactoring is used to move a field from one class to another. The public interface to that field is preserved, access is just delegated to the target class. The refactoring heavily depends on the refactorings Encapsulate Field, Generate Accessors and Rename Field. With Encapsulate Field, the public interface is maintained, Generate Accessors creates a public accessible field in the target class while the Rename Field refactoring delegates the calls to the target.
We also need two inputs from the user: the target class and a variable that references an object of the target class which we can use to delegate the calls.

Procedure

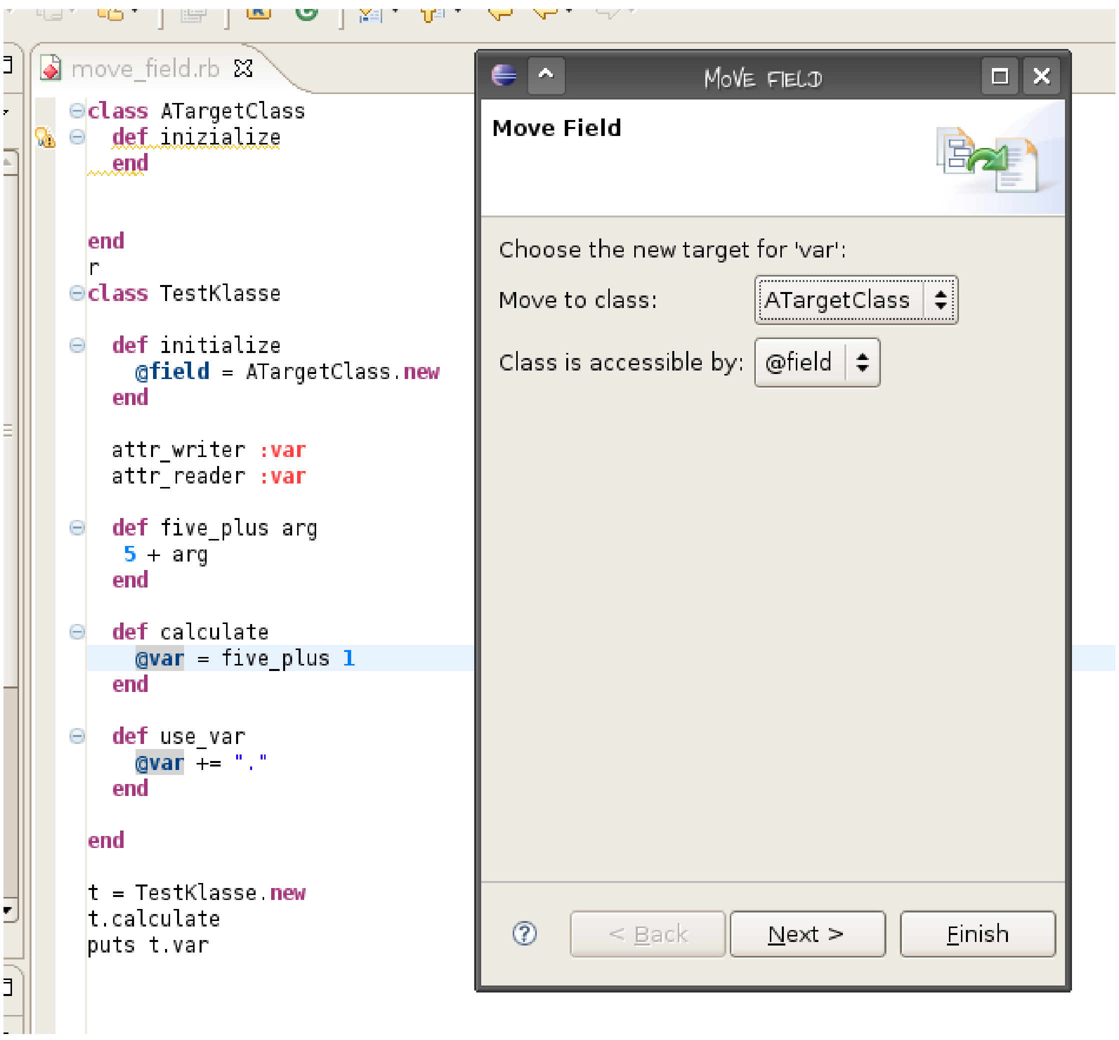
The selected field is used to evaluate the source class and the available fields we can use to delegate to. All from the current source location visible classes are found and presented to the user to choose from. After that, three things have to be done:

- Locate the target class and use the Generate Accessors refactoring to create a new field with the same name as the field we want to move.
- Use the Encapsulate Field refactoring to encapsulate the field we are moving. This way, all defined accessors are converted into their corresponding methods.
- The Rename Field refactoring is abused to create delegates to the target class via the field the user selected. Listing 7.38 illustrates this step further.

Restrictions

Moving fields has a few restrictions and needs some user input:

- To call the refactoring, a field has to be selected.
- The field needs to be inside a class.
- A reference to the target class must exist and the user needs to choose it.



```
class ATargetClass
  def inizialize
  end


end
r
class TestKlasse

  def initialize
    @field = ATargetClass.new
  end

  attr_writer :var
  attr_reader :var

  def five_plus arg
    5 + arg
  end

  def calculate
    @var = five_plus 1
  end

  def use_var
    @var += "."
  end

end

t = TestKlasse.new
t.calculate
puts t.var
```

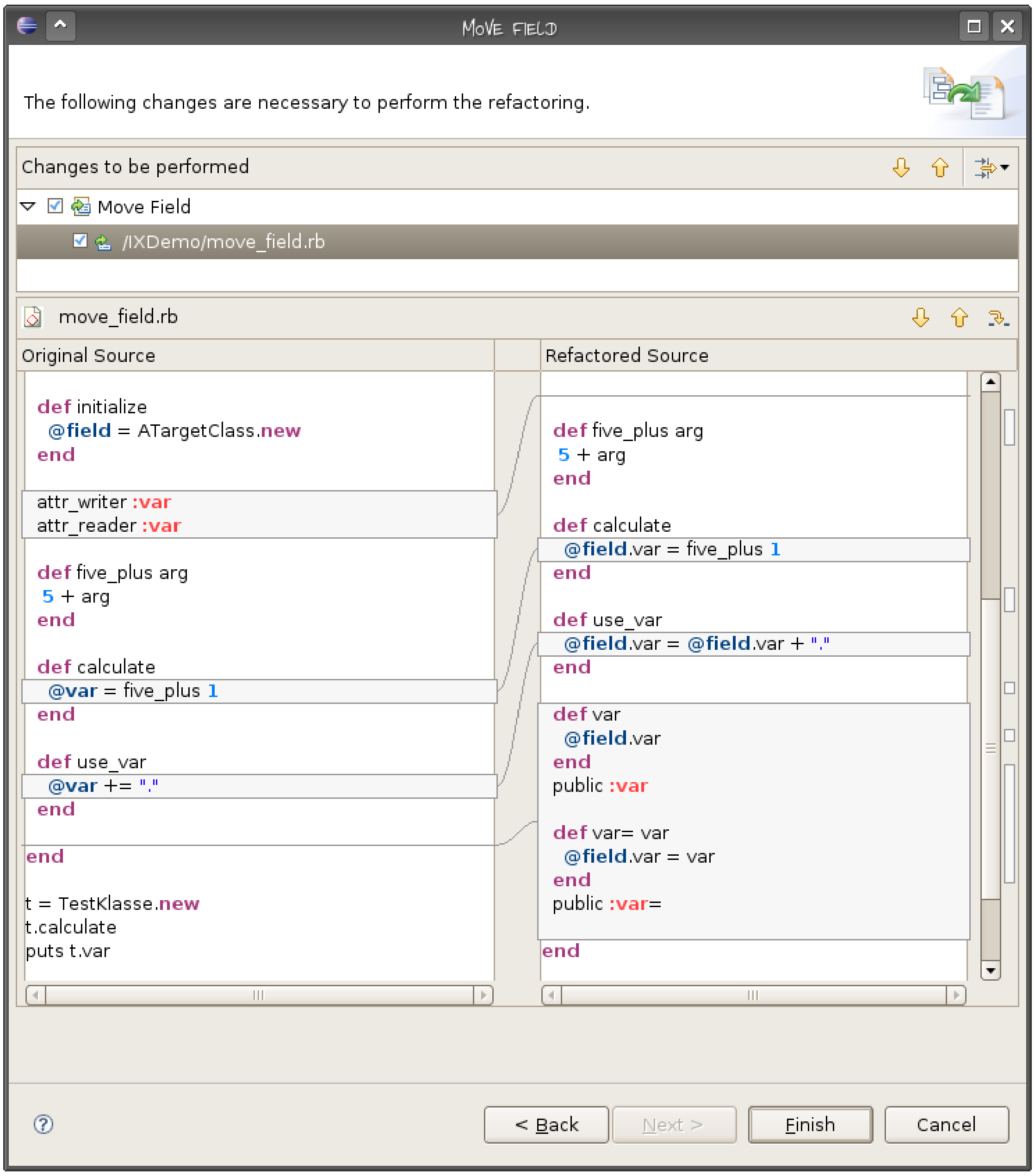
Move Field

Choose the new target for 'var':

Move to class: ATargetClass

Class is accessible by: @field

Figure 7.36: Move Field: The user needs to select the target class and a referencing variable.



Original Source:

```
 def initialize
  @field = ATargetClass.new
 end

 attr_writer :var
 attr_reader :var

 def five_plus arg
  5 + arg
 end

 def calculate
  @var = five_plus 1
 end

 def use_var
  @var += "."
 end

end

t = TestKlasse.new
t.calculate
puts t.var
```

Refactored Source:

```
 def five_plus arg
  5 + arg
 end

 def calculate
  @field.var = five_plus 1
 end

 def use_var
  @field.var = @field.var + "."
 end

 def var
  @field.var
 end
 public :var

 def var= var
  @field.var = var
 end
 public :var=

end
```

Figure 7.37: Move Field: The result of the refactoring.

```
class Source
  def initialize
    #we use this as delegate:
    @ref_to_target = Dest.new
    @var = 5
  end
  def var
    @var
  end
end
```

```
class Source
  def initialize
    #we use this as delegate:
    @ref_to_target = Dest.new
    @ref_to_target.var = 5
  end
  def var
    @ref_to_target.var
  end
end
```

Figure 7.38: Applying the Rename Field in the Move Field refactoring.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.39.

Testing

Like all refactorings that affect multiple files, the test properties file has the two mandatory fields and basically the same inputs as the user interface asks for:

- activeFile: String - The file that would be selected in the editor.
- destinationFiles: List of Strings - All included and otherwise known files.
- caretPosition: Integer - The position of the caret.
- selectedClass: String - The target class of the operation.
- selectedReference: String - The field we use to delegate.

Refactoring Evolution

The refactoring was done as the last on our list because of the dependencies to the other ones, so there was no time for big evolutions to take place.

Challenge

The refactorings was not very challenging, we just had to plug the other refactorings together. But it showed us how important it is to separate the core of the refactoring from the user interface, and how well we achieved it.

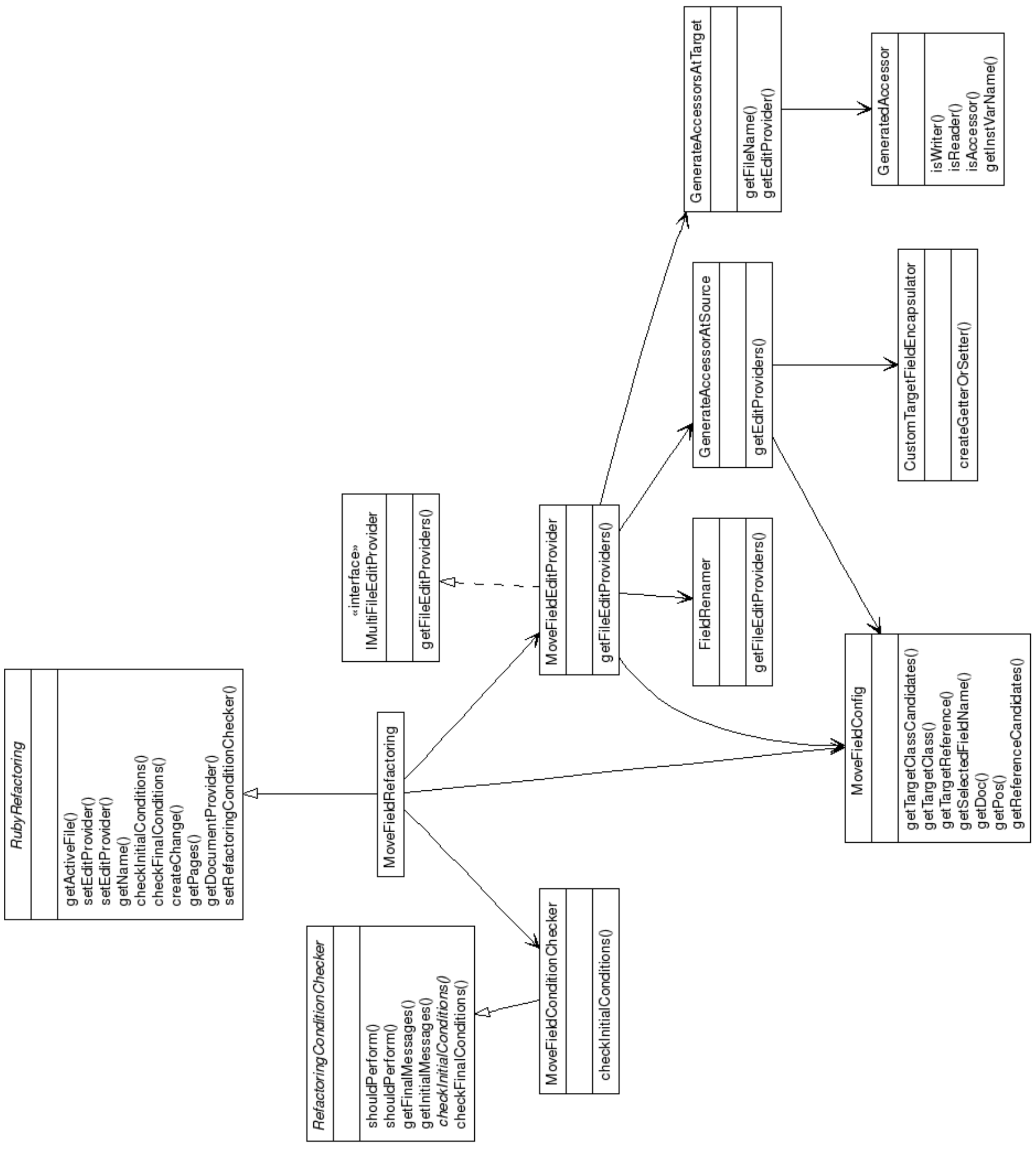


Figure 7.39: Move Field Diagram

Extensions and Ideas

Making the refactoring more intelligent would be desirable, so the user need not provide us with a field anymore.

7.4.9 Move Method

At first view, moving a method seems quite simple. Just remove it in a source class and place it in a target class. After looking closer there are lots of details, that make the Move Method refactoring change a lot of code.
First of all there is the problem, if the method you like to move has a public visibility and is accessible from outside of the class. Since Ruby is dynamically typed it is nearly impossible to find all calls to the affected method.

In the following paragraphs, the difficulties of the Move Method refactoring are described.

Leave a delegate method The Move Method refactoring provides the possibility to leave the method in place with a delegate to the new method in the target class. This will guarantee that all your code is still fit to work.
If a delegate method is left in place the Move method refactoring becomes really simple because there is no need to change calls to the original method.

Singleton method If the method that is moved is a singleton method, moving becomes simpler, because the moving method might not contain references to the instance fields of the class. And method calls out of the method are allways calls to other singleton methods.

Field of type target class If the method that will be moved is called in the source class, the source class needs an instance field of the type of the target class, so all the method calls from the source class to the moving method can be called via that instance field.
When such method calls exist in the source class, the visibility of the method that is moved needs to be set to public to assure that it is visible from the source class.
Since Ruby is dynamically typed, the type of its instance field cannot just be evaluated. Because of this, the user needs to decide which of the instance fields is the right one.

Method calls to the moving method If a delegate method is left in the source class, nothing has to be done with the statements that call the method that is moved. If that is not the case, all those calls need to be changed, so they call the new method in the target class. This is done using the instance field described above.
If the visibility of the moving method was private or protected in the source class, it will be changed to public so the method is visible to all the calls.

Method calls to the source class in the moving method The method that is moved may contain calls to other methods from the originating class. To maintain those calls, an additional method argument, named after the source class, is added to the method. All the calls will be changed so they use all their old arguments and as the last (additional) argument they use “self”. Now the moved method in the target class has access to the source class and can call all the source class’ methods through the added argument. To assure that there will be no errors, all the source class methods that are called in the moving method will be made publicly visible.

Access to source class fields in the moving method If the moving method uses instance fields of the source class they are handled using the same additional method arguments as described in the previous paragraph. To assure that the access to the instance fields will be granted, the needed accessors (`attr_reader`, `attr_writer` or `attr_accessor`) will be generated.

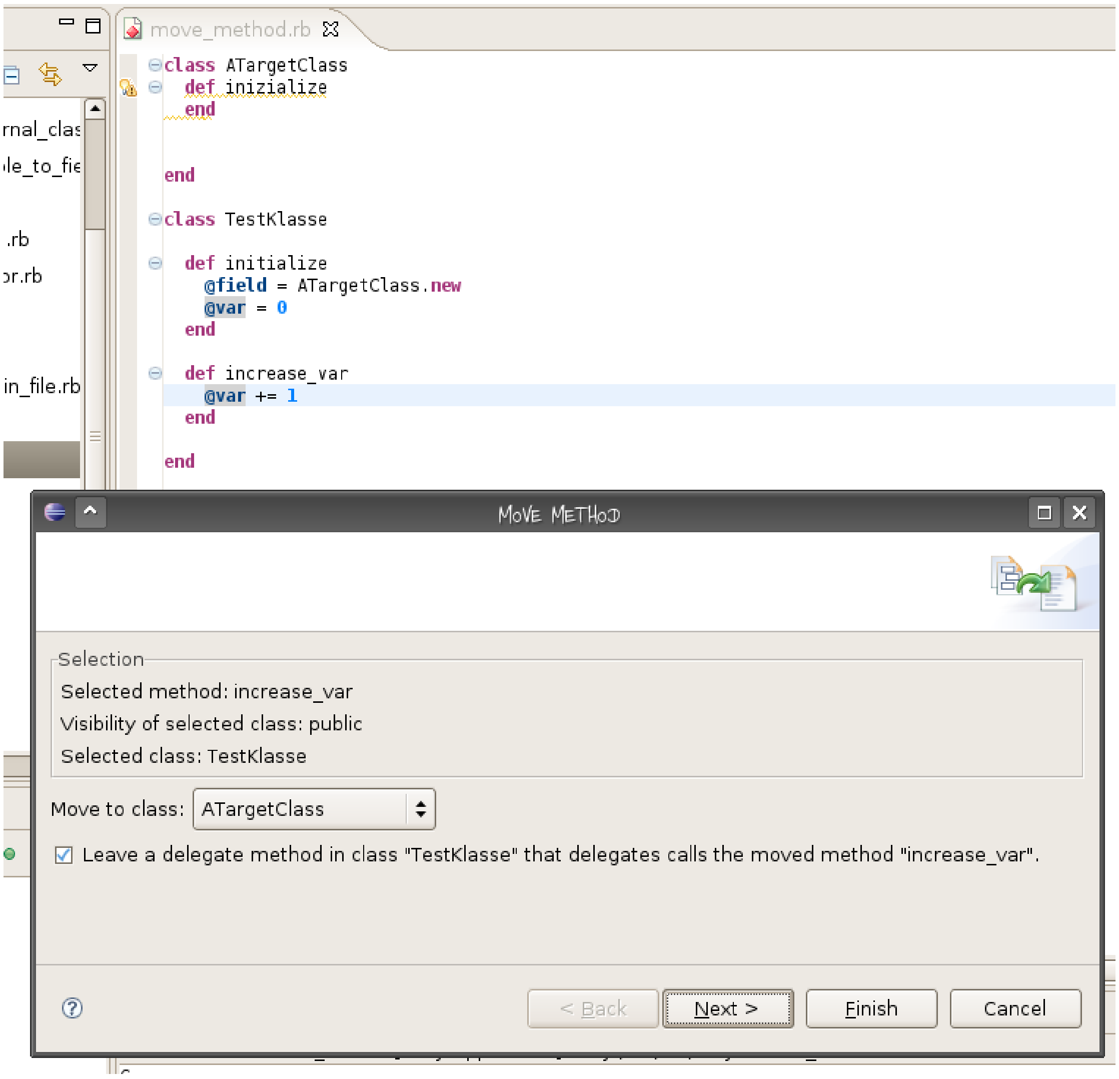
Method and Argument Names If a method is moved into another class, the possibility exists that the target class already contains a method with the same name. In this case, the moving method will be given a new name to prevent name conflicts. The same applies for the name of the additional argument that might be added to the moving method.

Demonstration

Here the interface that will be shown to the user can be seen.

Procedure

The user selects the method that he would like to move with the caret and invokes the Move Method refactoring. Now the selected method and its surrounding class are evaluated. A collection containing the calls from the source class to the moving method is created. Another one containing references (fields or method calls) from the moving method to the source class is built.
After the initial condition tests were passed successfully (see the next section), the user is asked to select the target class out of a list of class names. If necessary, the user needs to decide which of the existing instance fields in the source class have the type of the target class. Now all the changes in the source and destination document, which might be the same, are prepared.
In the destination document, this includes to insert the moving method in the target class. This step also covers the adding of the additional self-referencing argument and the replacement of all the references to the source class (instance fields and method calls) with calls trough this reference. Depending on the visibility the inserted method requires, a statement to set the visibility might be generated if required. Whether this is done or not depends on the visibility that is met at the insertion point of the moving method.
In the source method, the moving method is either removed or replaced with a method that delegates calls to the target class. If no delegate method is generated, all the calls to the moving method in the source class are replace with calls to the new method in the target class over the instance field selected by the user. The instance fields and methods that are accessed out of the moved method are made public so that this access is possible out of the target class.



```
class ATargetClass
  def inizialize
  end


end

class TestKlasse

  def initialize
    @field = ATargetClass.new
    @var = 0
  end

  def increase_var
    @var += 1
  end

end
```

Selection
Selected method: increase_var
Visibility of selected class: public
Selected class: TestKlasse

Move to class: ATargetClass

[✓] Leave a delegate method in class "TestKlasse" that delegates calls the moved method "increase_var".

Figure 7.40: Move Method Wizard

After the user saw the changes that will be done on the preview page, he can apply the changes to the documents and finish the refactoring.

Restrictions

To invoke the refactoring successfully there are the following requirements:

- The caret needs to be inside a method.
- The selected method needs to be inside a class.
- At least one other class as possible target needs to exist in the active project.
- If the source class contains calls to the selected method at least one instance field needs to exist in the source class as a possible candidate for the field with the type of the destination class. This restriction only applies, if the selected method is not a static method.
- The selected class should not contain any usages of class fields. Moving such a method might affect the functionality of the source class.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.41.

Testing

The tests for the Move Method refactoring are file driven tests (see 8.1.1). Here follows a description of the properties that can be set for a test.

- caretPosition: Integer - Position of the caret in the active document.
- selectedClass: String - Name of the target class.
- leaveDelegateMethod: Boolean - Leave a delegate method or not.
- selectedField: String - Optional - Name of the instance field referencing an object of the target class.

Challenge

This refactoring was a big challenge because in the worst case it affects half of the code of a class and several source code files too. There is a huge amount of possibilities to consider. Is the moving method a static method or not? Does the user want to leave a delegate method in the source class? Are there references in the moving method to the source class? Are there visibilities to change or accessors to generate? Are there calls to the moving method? Are these calls static method calls or not? To consider all those possibilities we spent a lot of time thinking about the design and implementation of this refactoring.

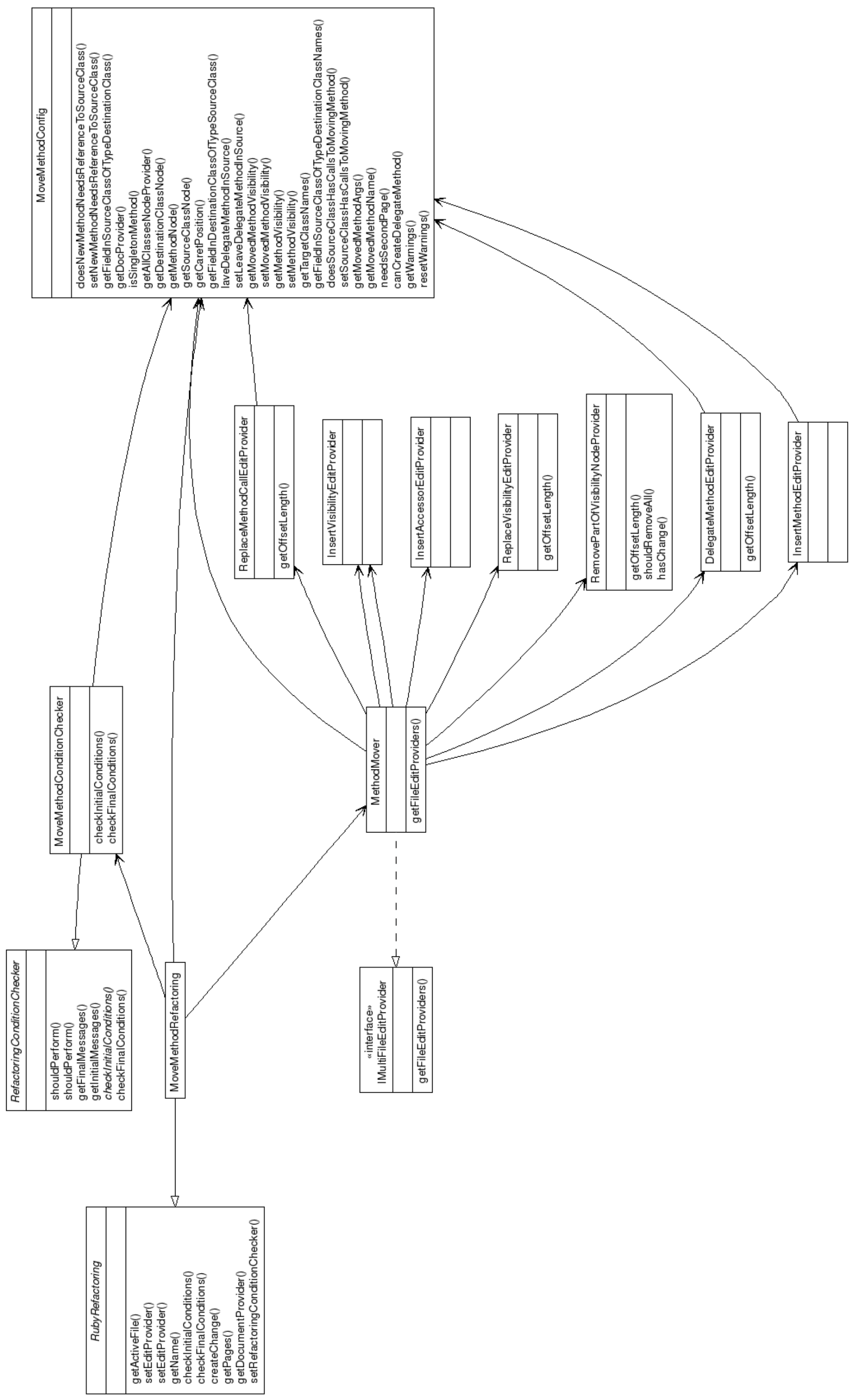


Figure 7.41: Move Method Diagram

7.4.10 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.

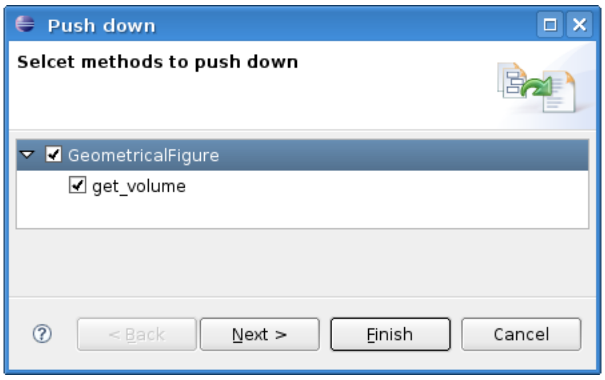
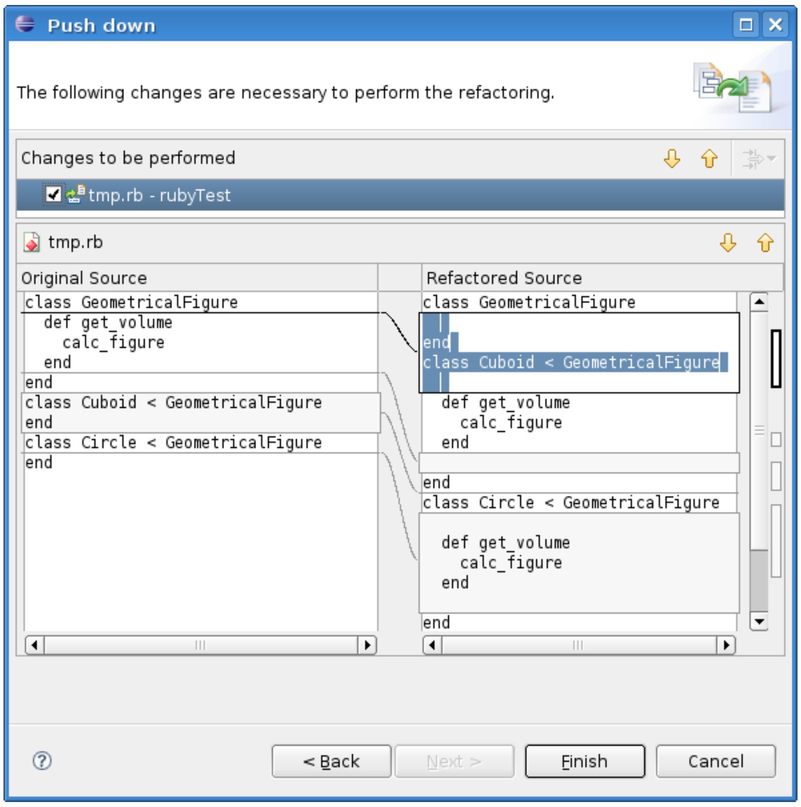


Figure 7.42: Push Down Method Refactoring Wizard

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



The following changes are necessary to perform the refactoring.

Original Source:

```
class GeometricalFigure
  def get_volume
    calc_figure
  end
end
class Cuboid < GeometricalFigure
end
class Circle < GeometricalFigure
end
```

Refactored Source:

```
class GeometricalFigure

end
class Cuboid < GeometricalFigure

  def get_volume
    calc_figure
  end

end
class Circle < GeometricalFigure

  def get_volume
    calc_figure
  end

end
```

Figure 7.43: Push Down Method Refactoring Wizard Result

Procedure

The super classes are provided a normal ClassNodeProvider containing the source document. To find all the classes that extend one of those super classes, a ProjectClassNodeProvider is created. The user can check methods that are implemented in the super classes (visualized as child nodes of the super class in the tree). 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 7.44.

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 current 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.

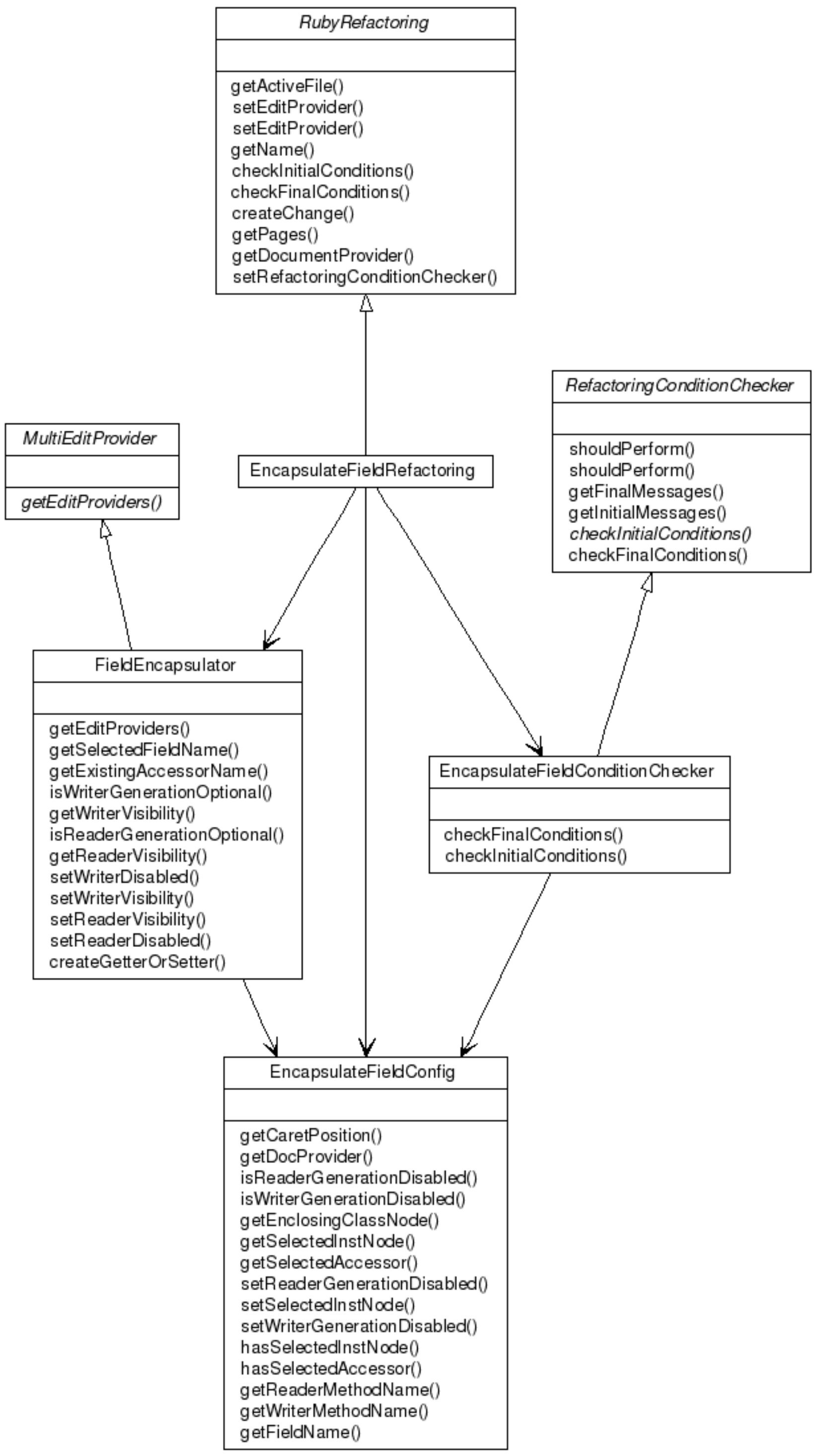
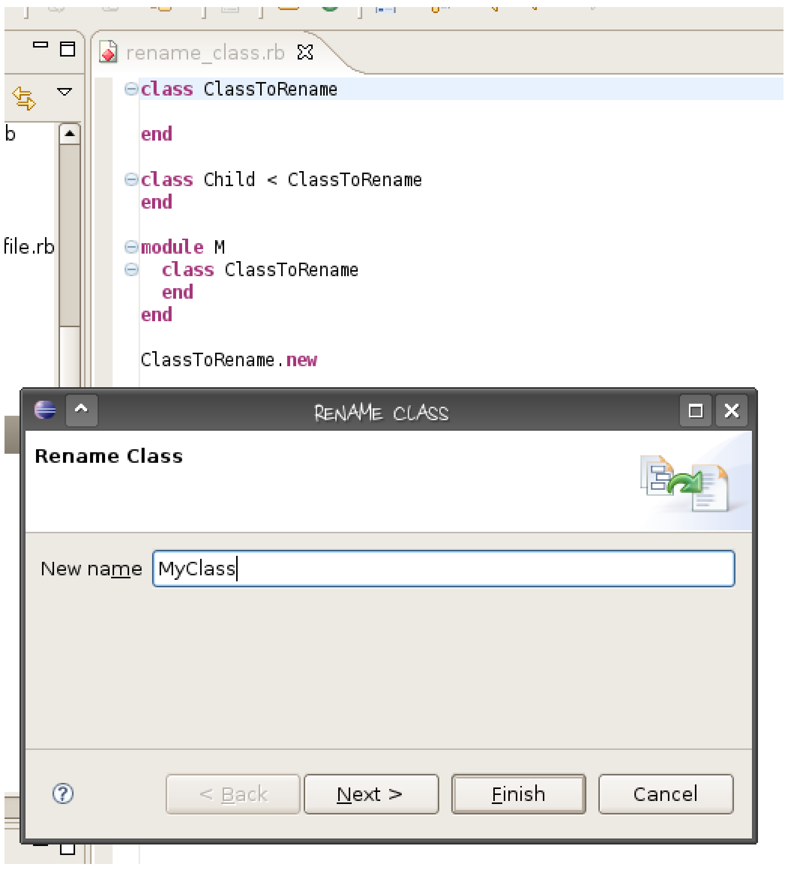


Figure 7.44: Push Down Method Diagram



```
class ClassToRename

end

class Child < ClassToRename
end

module M
  class ClassToRename
  end
end

ClassToRename.new
```

Figure 7.45: Sample of a class we want to rename.

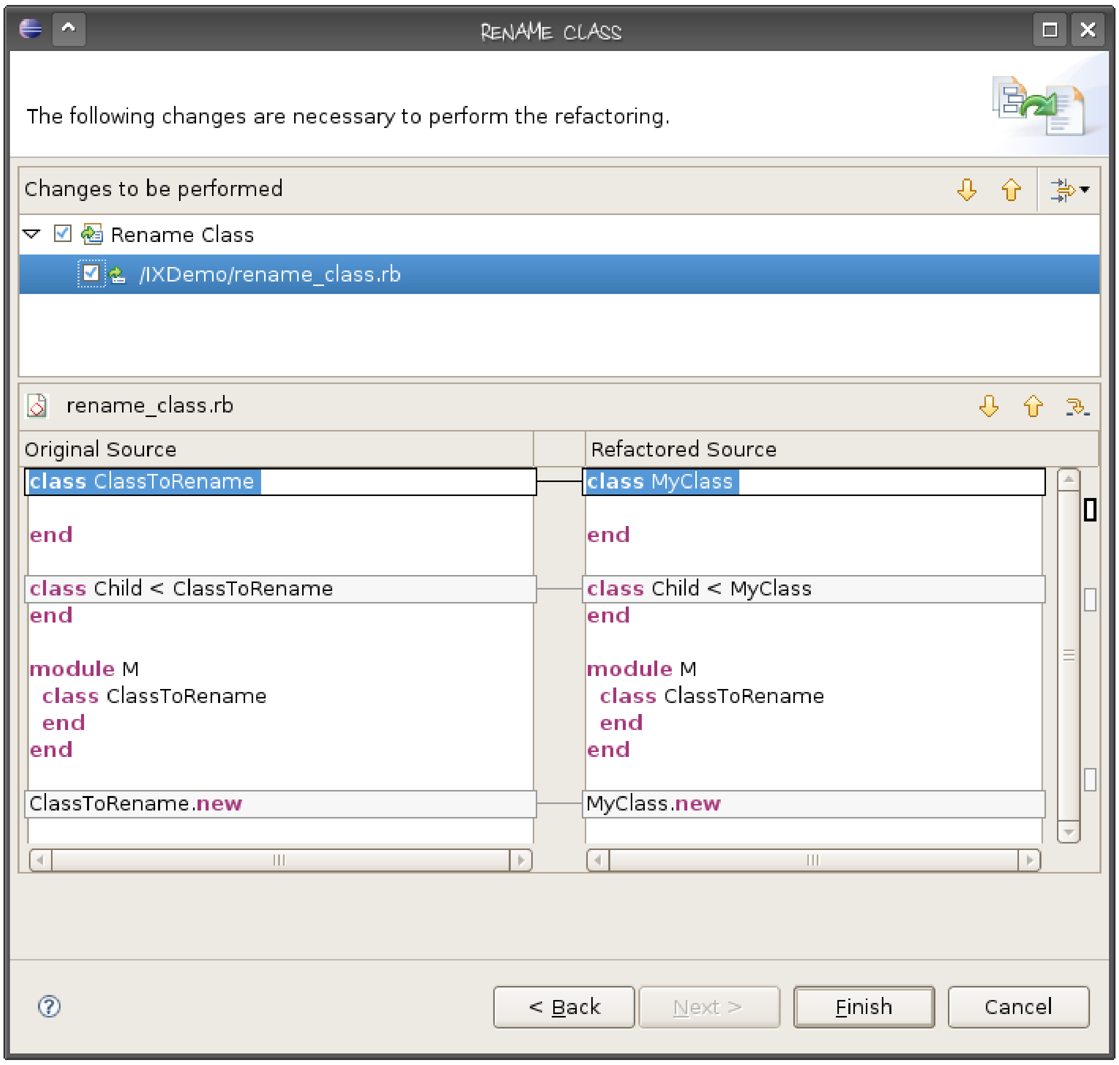
7.4.11 Rename Class

The Rename Class refactoring can be used to rename a class at all places where it is defined, reopened, subclassed and instantiated. A sample can be seen in figures 7.45 and 7.46.

Procedure

If the user selected the name in the definition of a class, we extract it and find the surrounding modules, if there are any. Then we have three different kind of occurrences that we are interested in:

- Other parts of the class, if it was reopened.
- Classes that derive from our class.
- Instantiations of the class.



Original Source:

```
class ClassToRename

end

class Child < ClassToRename
end

module M
  class ClassToRename
  end
end

ClassToRename.new
```

Refactored Source:

```
class MyClass

end

class Child < MyClass
end

module M
  class ClassToRename
  end
end

MyClass.new
```

Figure 7.46: Difference before applying the changes.

We traverse through all classes and calls that are visible from our location and create edits for the three above mentioned cases. To determine if we need to rename the class, we just compare the name and possible module prefixes.

Restrictions

The only restriction we impose is the selection of a class to start from.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.47.

Testing

The refactoring might affect multiple files, so in addition to the usual activeFile and destinationFiles, we have the following properties:

- pos: Integer - The cursor position which should be inside a class.
- name: String - The new name of the class.

Challenge

This refactoring was quite easy to implement, the most challenging part was to find all files that depend on the file the class is defined. Apart from that, the most work has already been done in the different node providers.

Extensions and Ideas

Right now, we can only rename classes. Modules could be renamed in a similar fashion, with the addition that they can be included into other classes as mixins. A refactorings that encapsulates a certain class into a module would also be nice.

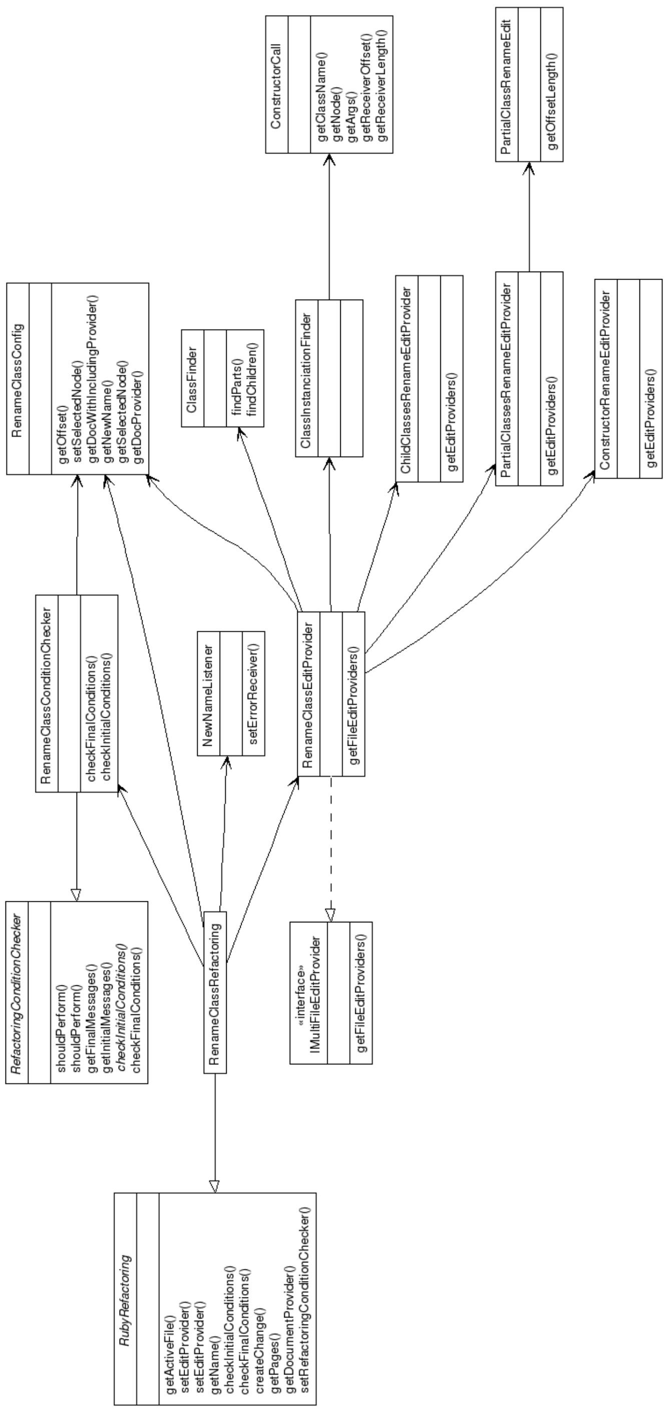


Figure 7.47: Rename Class Diagram

7.4.12 Rename Field

Rename Field replaces all occurrences of a field name with a new one. In Ruby a field can be recognized by the "@" symbol at the beginning. Class fields have two leading "@" symbols. Furthermore an instance field is accessible through the attribute definitions: `attr`, `attr_accessor`, `attr_reader` and `attr_writer`. When renaming, all these field declarations have to be adapted too.
Usually Rename Field is applied when a misleading name of a field has been assigned or if there has been a spelling mistake. The Rename refactorings are certainly used very often when developing software, even if you are not wearing the refactoring hat.
The refactoring can be invoked by setting the caret at the corresponding position of any occurrence of a field and selecting the Rename Field refactoring from the refactoring menu. Then the user is shown a list of occurrences, if the field has got accessors. Outside the class it is quite difficult or almost impossible to determine if a method call accesses a certain field. Thus the user has to select those calls himself to get them refactored.

Demonstration

Here the interface that will be shown to the user can be seen.

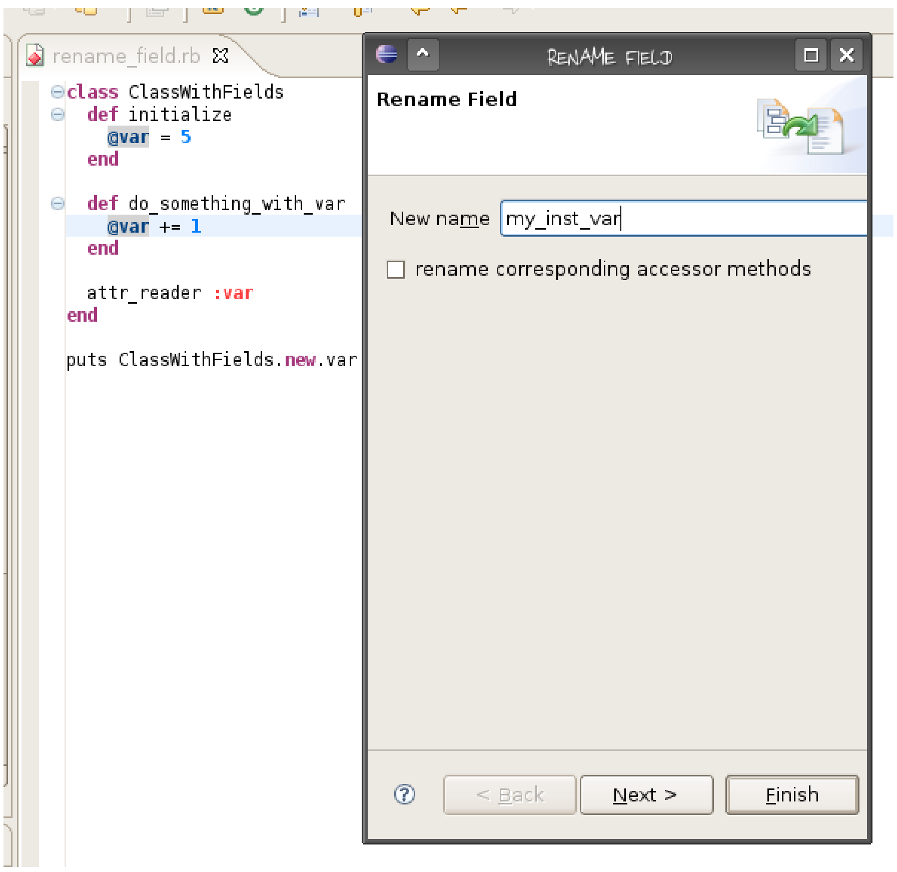
Procedure

When the refactoring is started the plug-in seeks a field at the caret position. It is determined whether there is a class or instance field. The whole class is scanned for other field occurrences to determine which names are not available anymore. Through the refactoring wizard a new, unique name can be entered.
While the attr-accessors are renamed in any case, the method accessors have to be selected separately. That is because the attr-accessors have no functionality besides returning or setting a value, whereas the method accessors can be doing whatever the user want them to do. This is especially interesting if the new field should be accessible through the old name.
The user receives a list of occurrences of the selected field. There he can choose which occurrences should be renamed. All occurrences within the class are selected by default. Those beyond the class, if there are any accessors, are deselected as it cannot be assured that the types match.
After the selection is confirmed the selected field and field-call occurrences are renamed.

Restrictions

To invoke the refactoring successfully there are the following requirements:

- The caret has to be set at or in a field name.
- A new name, that does not already occur in the current class has to be entered.
- Renaming a field requires a surrounding class-definition.



```
class ClassWithFields
  def initialize
    @var = 5
  end

  def do_something_with_var
    @var += 1
  end

  attr_reader :var
end

puts ClassWithFields.new.var
```

New name: my_inst_var

☐ rename corresponding accessor methods

Figure 7.48: Rename Field Wizard

Because of the dynamic typing in Ruby the access to a field from outside the class is very hard to determine. Candidates are of course method calls with the same name. But its not always sure what class an object belongs to. In some cases it is possible that a call is correct for several different types of objects. So the plug-in just provides the possibility to the user to select the occurrences outside a class himself, but does not make a proposal what calls to rename.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.49.

Testing

The testing relies on our file based testing environment. The properties file name has the common structure: `rename_field_<NR>.test_properties`
The following options can or have to be set in the properties file.

- caretPosition: Integer - Represents the position with where the caret is to be expected in the editor. This is to determine which field name occurrence is selected.
- newName: String - Stands for the new field name that should be set.
- replaceMethodAccessors: Boolean - A flag to determine whether the method accessors should be renamed.

Refactoring Evolution

During the planning of the diploma project we intended to implement RenameField at the end, as it seemed to be a quite difficult refactoring. What we had not realized then, that we have dependencies on RenameField from other refactorings. Or in other words that we could profit from an existing RenameField so we would not have to re-implement similar functionality several times. So we decided to bring it forward in our schedule for some weeks. It is now used by the InlineClass and MoveField refactorings.
Nevertheless we expect this to be a refactoring that will be used quite often as most of the other rename refactorings.

Challenge

The most challenging part of this refactoring was the matching of the several different types of accessing and defining a field. Usually the fields are represented by an identifier having a leading “@” symbol. Furthermore any kind of accessor can represent a field too, but the field there is identified by a symbol with a similar name. But solving this problem was quite easy compared to the challenges of other refactorings.

Extensions and Ideas

RenameField lacks especially in the point that we cannot determine the type of an object for sure. There now is quite simple expectation about which calls are to the selected field and what calls are not. If there was a certain possibility to determine this a more useful selection of the renamed field access could be provided to the user.

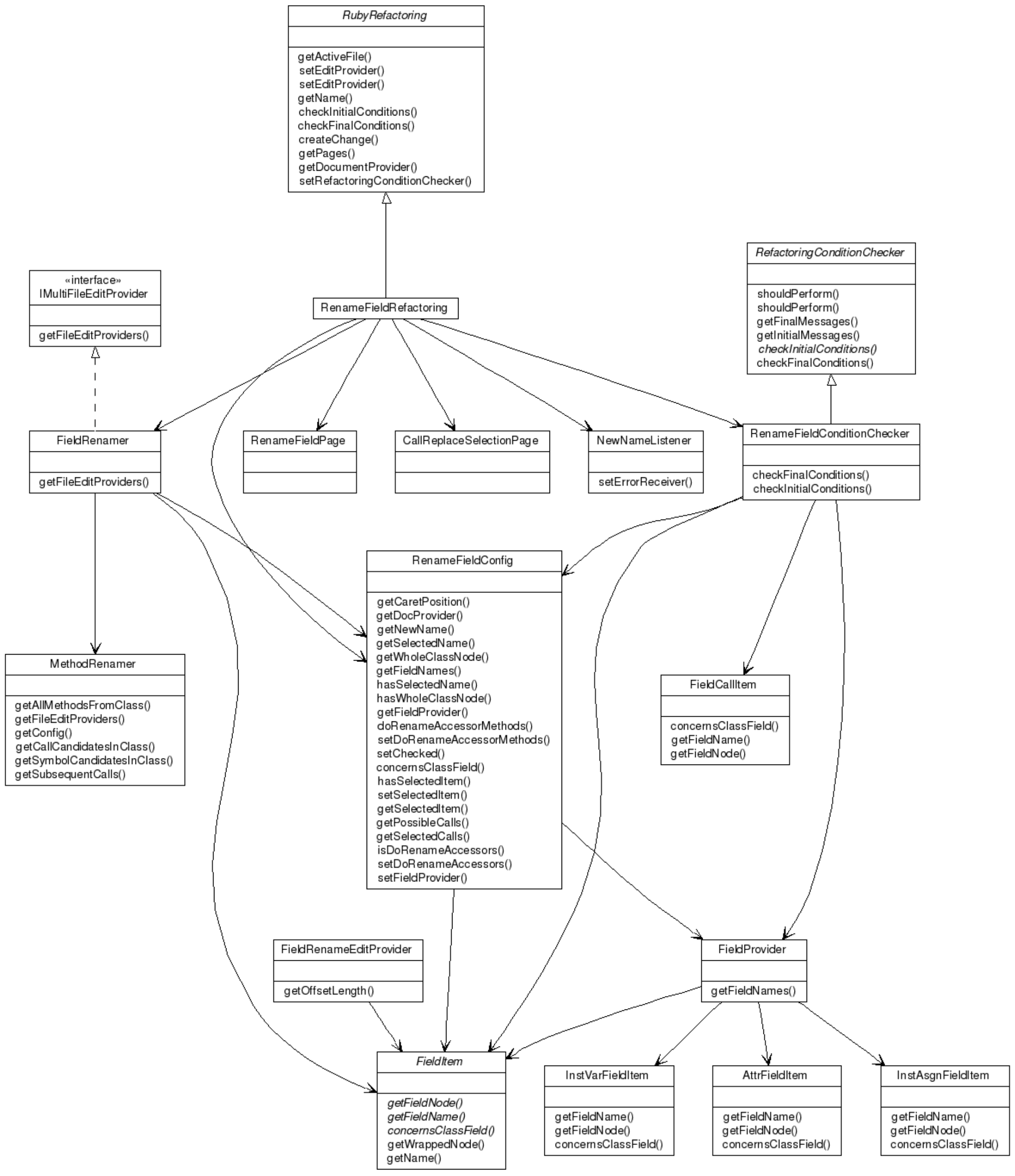


Figure 7.49: Rename Field Diagram

7.4.13 Rename Local Variable

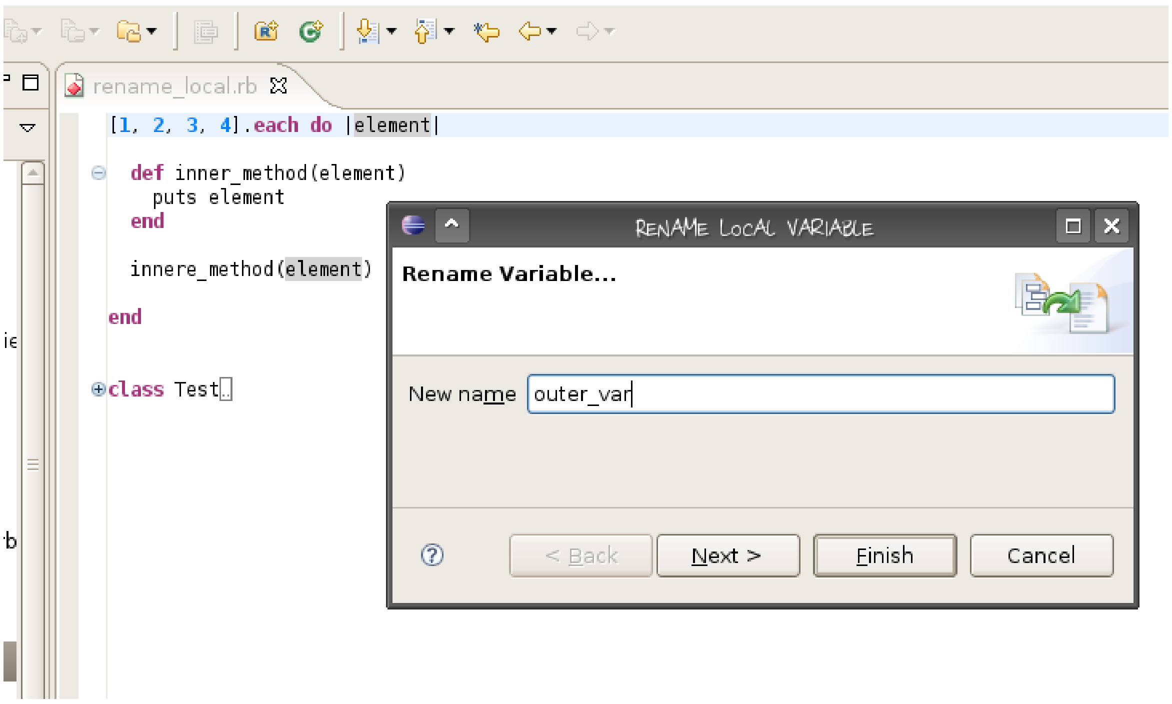
Rename Local Variable can be used to rename locally created variables, parameters from method definitions and variables introduced in blocks (see listing 7.50). The user can assign a new name and the variable is renamed in its scope.

```
#all typed of method arguments can be renamed:
def method argument, default_value = 5, *rest, &block
  #normal local variables:
  var = argument
  #and block, or iterator variables:
  rest.each {|r| p r}
end
```

Figure 7.50: Support variable types in Rename Local Variable

Demonstration

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

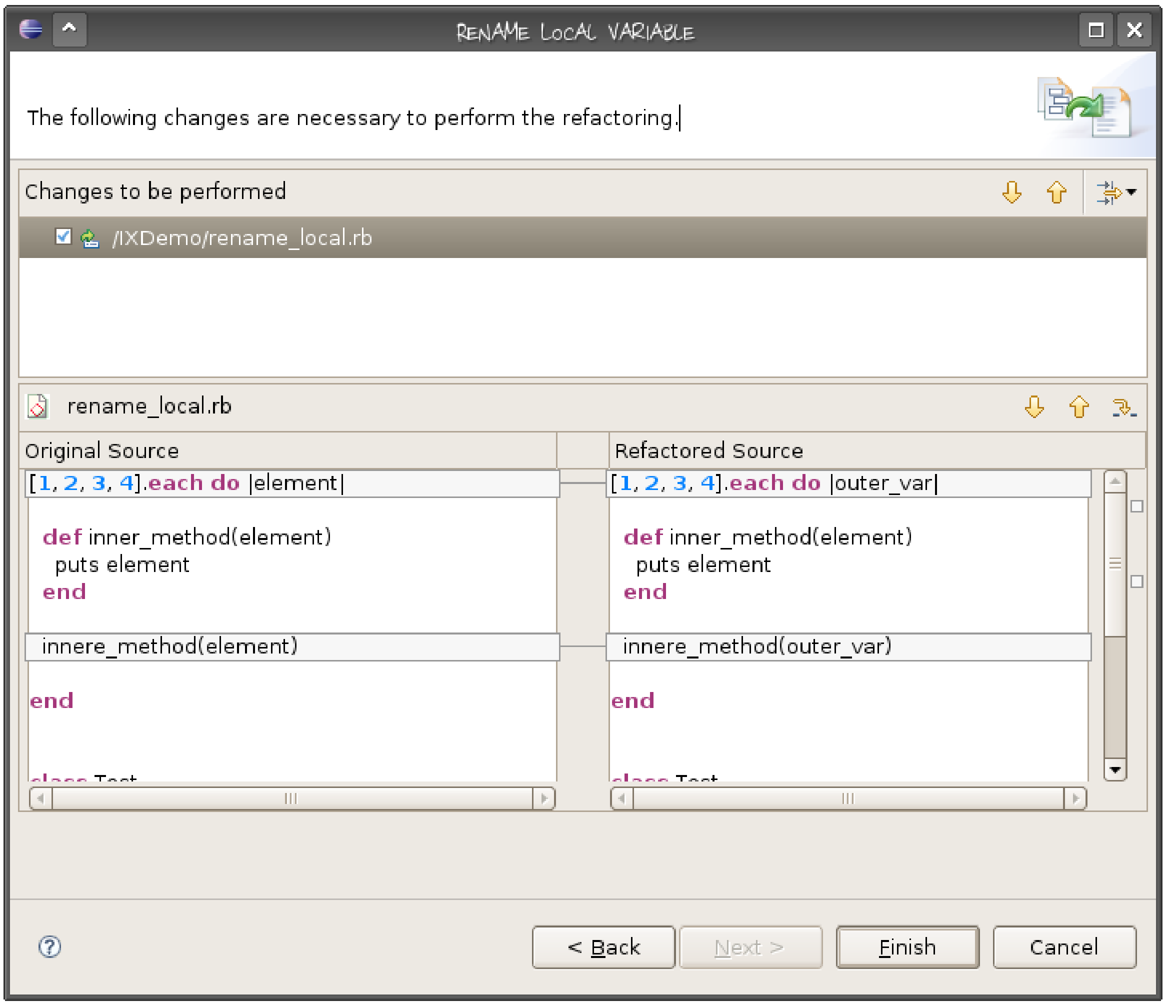


```
[1, 2, 3, 4].each do |element|

  def inner_method(element)
    puts element
  end

  innere_method(element)

end

class Test
```

New name: outer_var

Figure 7.51: Rename Local Variable Refactoring Wizard

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



Original Source:

```
[1, 2, 3, 4].each do |element|

  def inner_method(element)
    puts element
  end

  innere_method(element)

end


class Test
```

Refactored Source:

```
[1, 2, 3, 4].each do |outer_var|

  def inner_method(element)
    puts element
  end

  innere_method(outer_var)

end


class Test
```

Figure 7.52: Rename Local Variable Refactoring Wizard Result

Procedure

We start by finding the selected variables surrounding ScopeNode or create one if we cannot find it. This can happen if the variable is not defined inside a method or class. After the user has given us a new unused name, we simply rename all occurrences in the scope and in subscopes, such as blocks. The same procedure applies to block variables. We have to rename until we encounter another definition in a subscope, as shown in example 7.53.

```
def method
  #rename local_var to something more adequate
  local_var = 42
  def nested_method local_var
    # this variable should not be renamed
    puts local_var
  end
end
```

Figure 7.53: Renaming must stop if we encounter another definition of the name.

Testing

The properties file for the rename tests is quite simple, all you need to provide is the caret position, at which a local variable should be written, and the new name of the variable.

Refactoring Evolution

The first implementation of the rename local variable refactoring has been done in the term project. At the beginning of the diploma thesis, we reworked the whole refactoring and refined it. The first version could not rename block variables and always rewrote the whole body of the method. In contrary, the newer version should be able to handle all kinds of variables and replaces only the changed parts.

Challenge

Local variables exist in a lot of different forms, the usual local variable node in the AST for example does not know its name, it has only an id which refers to a string array in the surrounding scope and corresponds with an assignment, which has both the name and this id. Block variables are always defined between two pipes ($|i|$) and are visible in all sub-blocks, unless the name is redefined. The common method parameters work like local variables, but block arguments have their own node type. All this makes sense if you use the AST to interpret the program, but it adds a lot of work when creating refactorings.
Another nasty thing was the renaming of variables that assign to themself, like `i += 1`. This statement is represented in the AST as `i = i + 1`. If we rename the variables, we encounter the dreaded "overlapping text edits" error message, since the two `i`s share their position. That is one of the shortcomings of our simplified and optimized AST, but we can of course work around its limitations, it is just more work.

Class Diagram

To get an overview over the class model of the rename local variable refactoring take a look at the diagram 7.54.

Extensions and Ideas

The rename method functionality should be quite complete, but it would be nice if we could implement the renaming inline, like the JDT does, rather then with an obtrusive dialog.

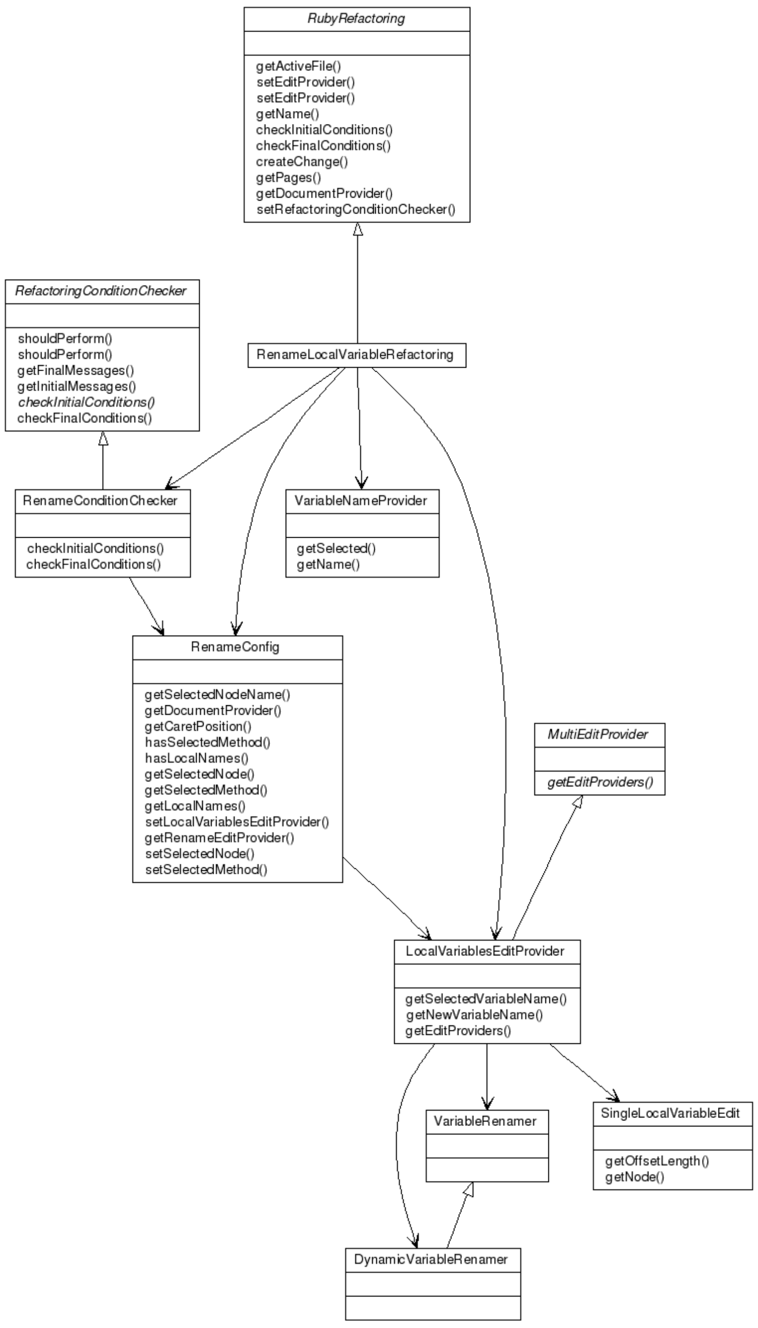
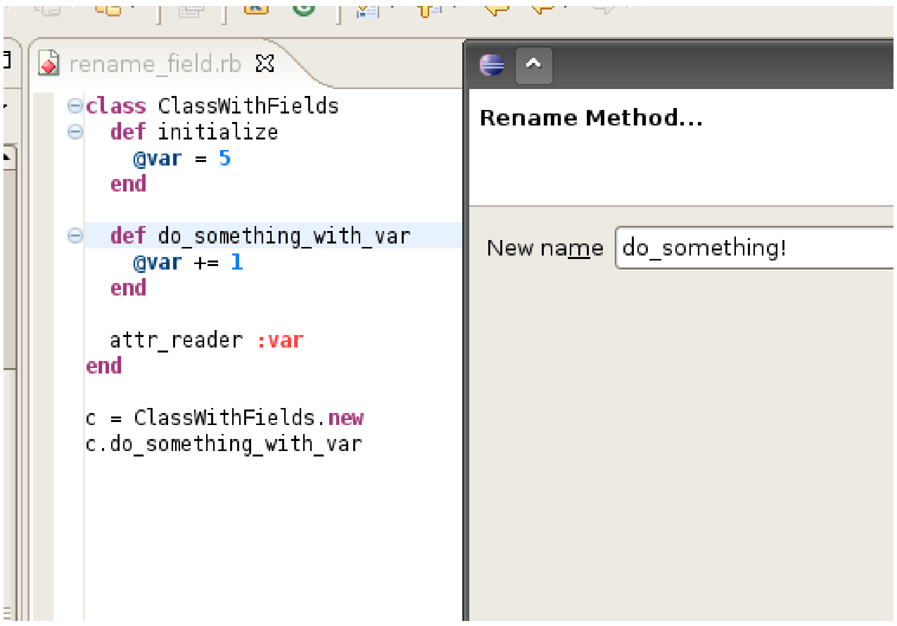


Figure 7.54: Rename Local Variable Diagram



```
class ClassWithFields
  def initialize
    @var = 5
  end

  def do_something_with_var
    @var += 1
  end

  attr_reader :var
end

c = ClassWithFields.new
c.do_something_with_var
```

New name: do_something!

Figure 7.55: Rename Method Wizard

7.4.14 Rename Method

The Rename Method refactoring, as its name lets assume, renames a chosen method. This of course includes the change of the calls to that method. As in Ruby actually everything is an object, not only methods in class definitions can be renamed. Even if a file contains just the method definition the refactoring is possible.
Due to the dynamic typing not all the method calls can be determined and not to every call the correct definition can be found. So this refactoring can only be used by selecting the method definition. In Ruby methods cannot be overloaded. A new definition of a method with the same name overwrites the original method. This makes it a bit easier as just the selected definition and the calls have to be considered.
Rename Method is usually used when a method name is not meaningful or as a subrefactoring, for example when a method has to be moved into another class. The refactoring is started by pointing at a method definition and then selecting Rename Method from the refactoring menu. In the appearing refactoring wizard the new name has to be entered and the method calls that have to be replaced can be selected. Calls inside the class are preselected by default. Method calls outside the class are available to select, but have to be checked by the user himself.
If the method overwrites or is overridden, the methods in sub and super classes are renamed too.

Demonstration

Figure 7.55 shows the interface that will be shown to the user.

Procedure

The refactoring searches the AST for a method definition at the caret position. Every method call is checked if it concerns a method with the same name as the selected definition node. For the definition node and each super and sub class, a MethodRenameEditProvider is constructed that creates the same definition node with the entered new name. As for the definition node for every call node found a MethodRenameEditProvider is created. The user can select the calls he wants to be renamed in the wizard.
No further options can be set. After the configuration is confirmed, all edits are created and applied to the documents.

Restrictions

To invoke the refactoring successfully there are the following requirements:

- A method can only be renamed when the caret is inside its definition.
- Calls outside the class cannot be determined certainly because of the dynamic typing. So they have to be chosen manually.
- The new method name has to be unique in the class.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.56.

Testing

The testing relies on our file based testing environment. The properties file name has the common structure: "rename_method_<NR>.test_properties"
The following options can or have to be set in the properties file.

- caretPosition: Integer - Represents the position with where the caret is to be expected in the editor. This is to determine which method definition is selected.
- newName: String - Stands for the new method name that should be set.
- renameCalls: Boolean - A flag to determine whether the method probably matching calls.
- renameAllCalls: Boolean - If renameCalls is true, renameAllCalls can be set to true for enable the renaming of all possible calls.

Refactoring Evolution

Rename Method was planned to be implemented at the end of the diploma project. Like the Rename Field refactoring Rename Method was brought foreward due to dependencies of other refactorings. Rename Method is used in Inline Class when having name conflicts and in Rename Field for renaming field accessors. Effectively it was the first refactoring used in another one when Rename Method was used in Rename Field. Generally it has a very similar design to the Rename Field refactoring and has been more or less the same to implement.

Challenge

There has been no especially challenging part when implementing this refactoring. As the methods can only be defined once and the lack of knowing the type of a variable made it impossible to create a hundred percent sure determination of the occurrences of call names. So they could just be assumed.

Extensions and Ideas

For Rename Method type inferencing would be very useful when determining what calls are affected by the change. Which is also a similarity with Rename Field.

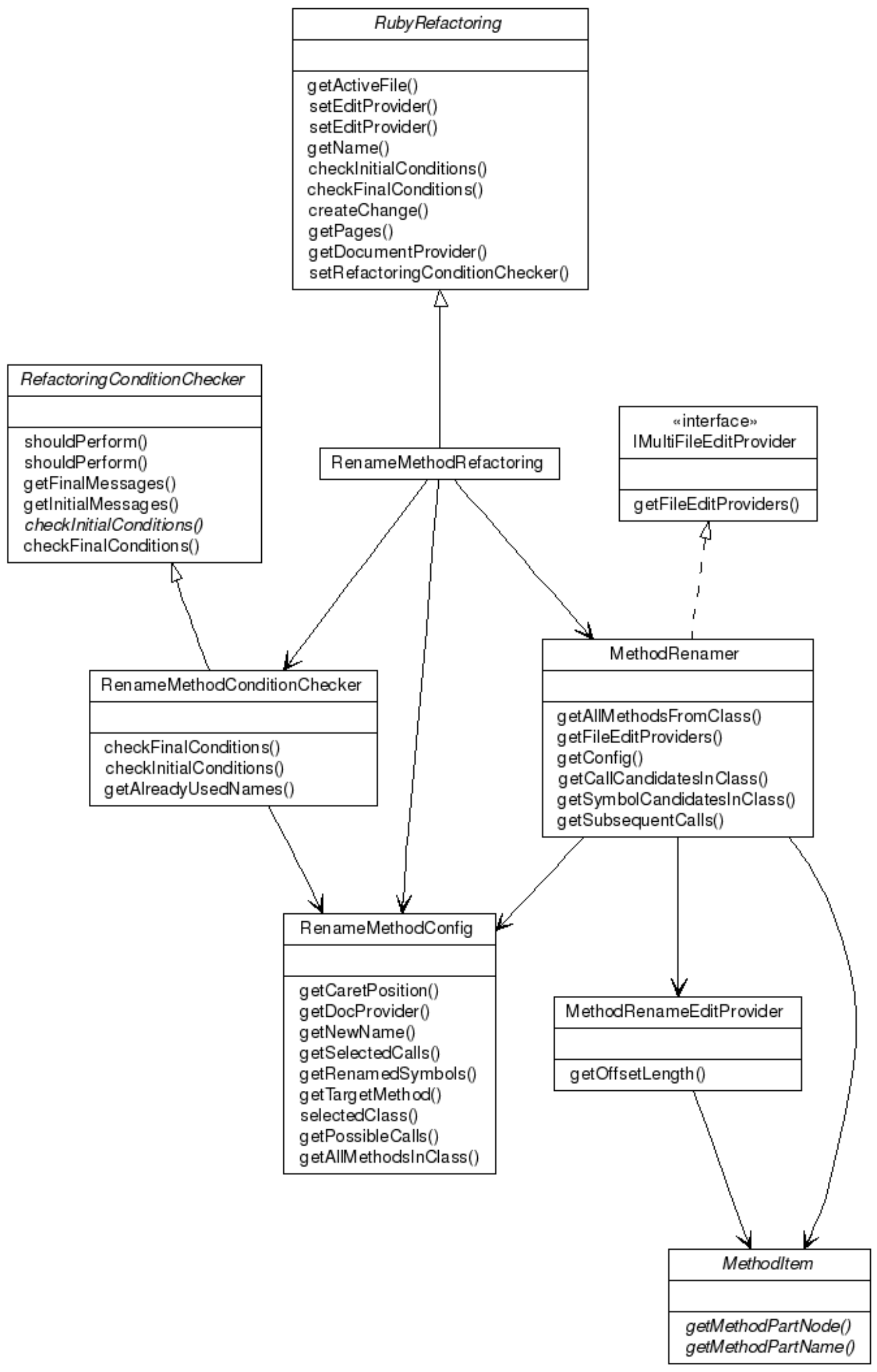
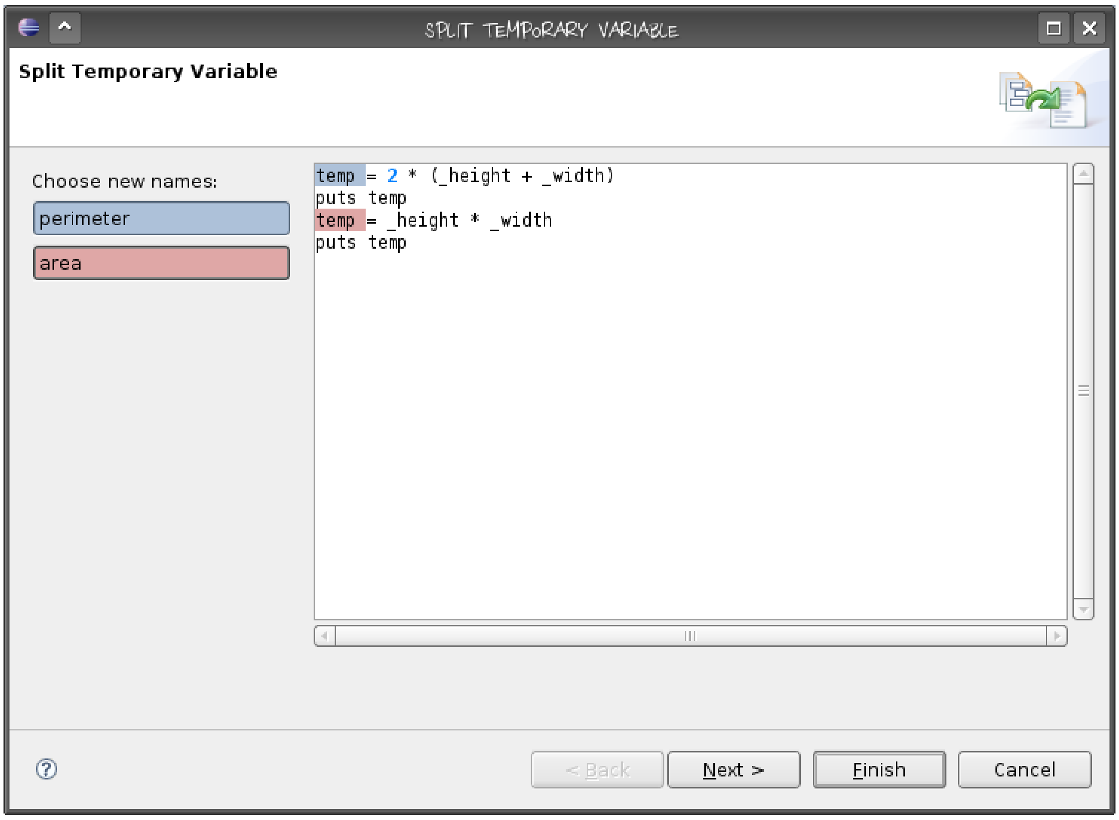


Figure 7.56: Rename Method Diagram



```
temp = 2 * (_height + _width)
puts temp
temp = _height * _width
puts temp
```

Figure 7.57: Split Temporary Variable: Assigning new names.

7.4.15 Split Temporary Variable

The Split Temporary Variable refactoring hopefully is not needed very often, but if you encounter code that uses a variable for different concerns, refactoring is really necessary. The refactoring can be started from an assignment to a local variable. The user then can assign a new name for each usage of the variable. It basically works like the Rename Local Variable refactoring, but on a limited scope – from an assignment to the next one or until the end of the scope.
Note that the refactoring is named "Split Local Variable" in menus and dialogs to present a consistent user interface.

Demonstration

The first step after the selection of the variable is to choose the new names, as shown in picture 7.57. As you can see, the assignments and their corresponding field share the same background color to make them distinguishable from each other.

Procedure

We first need to find all local assignments that share the same name as the selected assignment. These are all candidates to split. While gathering these assignments, we have to skip variables that assign to themselves, like loop variables:

```ruby
i = 0
while i < 500
  i += 1
  puts i
end
```

Here the increment on `i` should not be renameable on its own. Once we have all assignments, we determine their scope, where the scope of each assignment reaches just before the next one or to the end of the scope. After we got the new names from the user, we use the Rename Local Variable refactoring with our own abortion definition to assign the new names.

Restrictions

The only requirement we have is the selection of a local variable. If there is no other assignment, the refactoring works like Rename Local Variable.

Class Diagram

To get an overview over the class model of the refactoring take a look at the diagram 7.58.

Testing

The tests for the Split Temporary Variable Refactoring uses the following properties:

- pos: Integer - The caret position of the assignment.
- names: List of Strings - The new names that should be applied to the different usages.

Challenge

During the implementation of the refactoring, it became clear that we could reuse the Rename Local Variable refactoring. To implement our renaming based on smaller scopes, we changed the refactoring to allow more control over the renaming process by introducing an interface that is used to determine the abortion of the rename. This happens when we leave the previously determined area of the current variable.

Extensions and Ideas

An assistant that automatically finds local variables that are used for different purposes would perhaps be appreciated by the user.

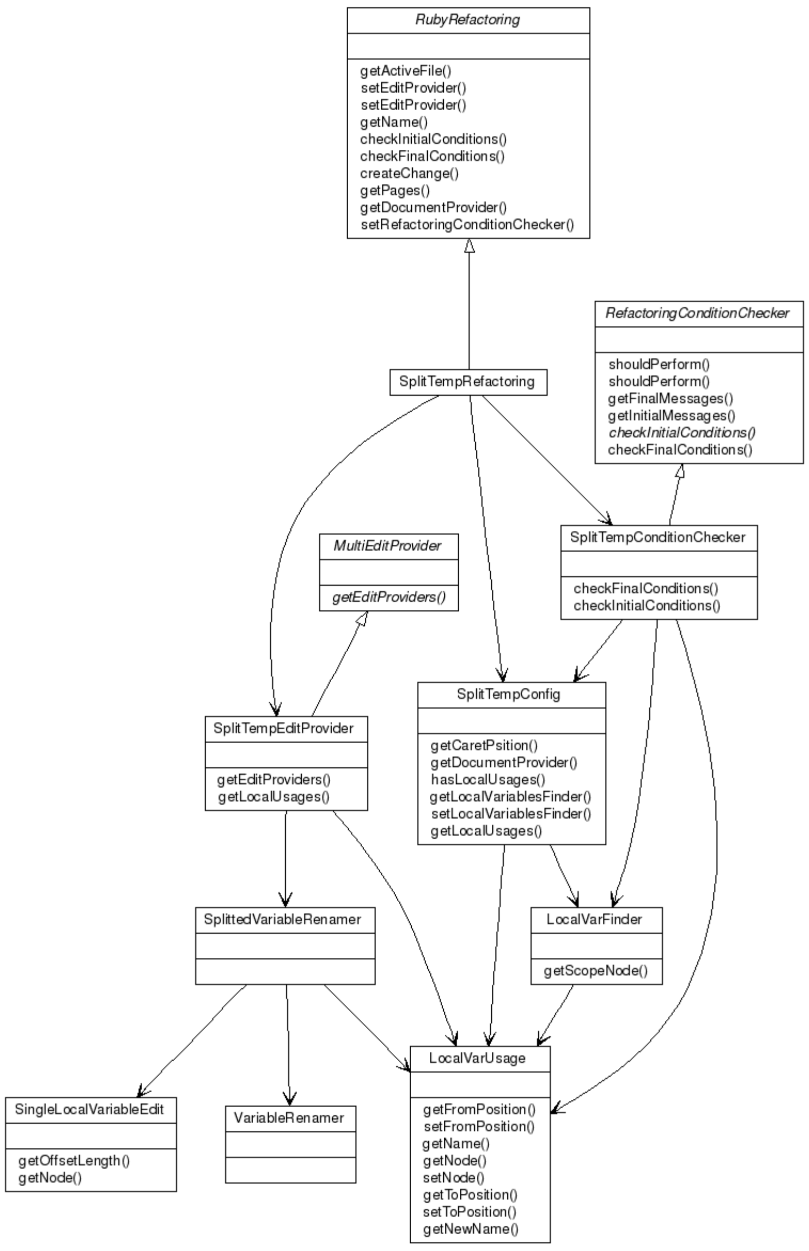


Figure 7.58: Split Temporary Variable Diagram

7.5 Other Plug-Ins

During our diploma thesis, we developed several Eclipse plug-ins that support us in certain tasks we often had to do. This section should give an overview over those tools.

7.5.1 ASTViewer

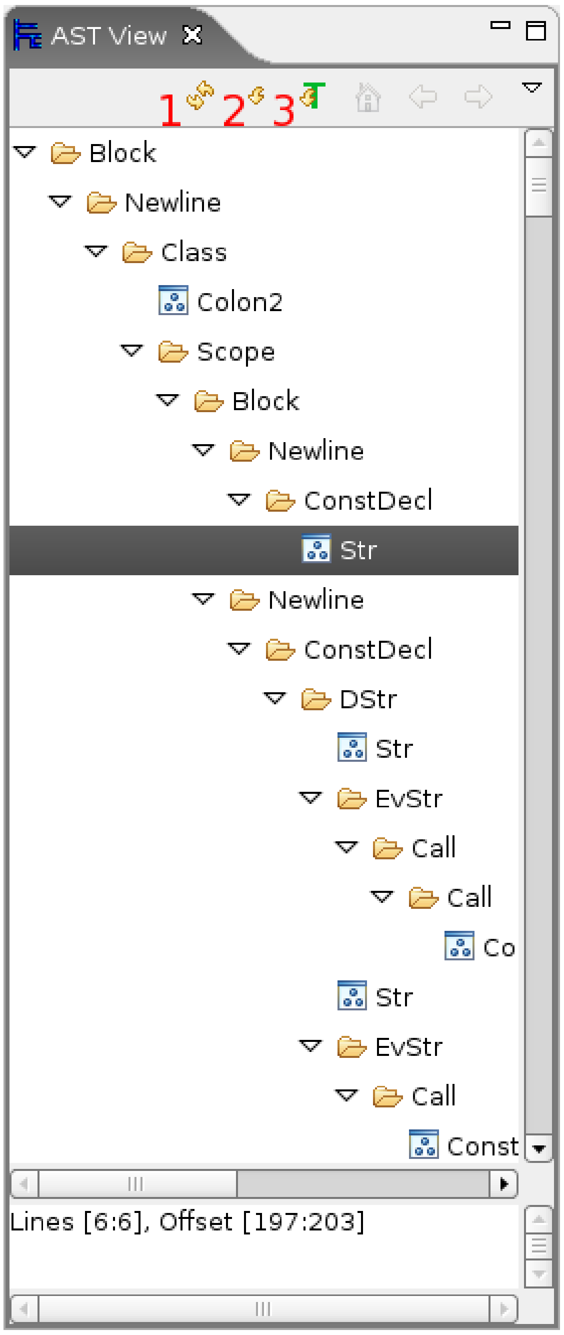
The ASTViewer plug-in started from a small Eclipse experiment and became a very useful tool in our daily work. Basically, it just displays the abstract syntax tree from the active Ruby file. Over time, more and more functionality has been added:

- Double clicking on a node selects the node position in the editor. Very useful to find wrong position information.
- Information about the selected node is shown in a small split window at the bottom of the view, this includes the line positions and character offsets.
- Refreshing is not done automatically, but can be invoked with the first icon.
- The second icon prints the selected nodes type and all childnodes to the console.
- The third icon generates a test case for the JRuby position tests and prints it to the console. This is very useful to create tests before fixing positions, just write the code, generate the test, change the wrong positions to the expected ones and copy the source to JRuby's test/testPosition.rb file.

A screenshot can be see in figure 7.59.

7.5.2 Showoffset

The Showoffset plug-in prints out the current caret position in number of characters from the start of the document. This is often very useful when writing tests where we have to specify the position in characters from start and not with line and column offsets that Eclipse provides. The plug-in can also select a position or range, mostly used to verify a position. The integration in the user interface is done via two toolbar buttons, shown in figure 7.60.



Lines [6:6], Offset [197:203]

Figure 7.59: The ASTView plug-in.

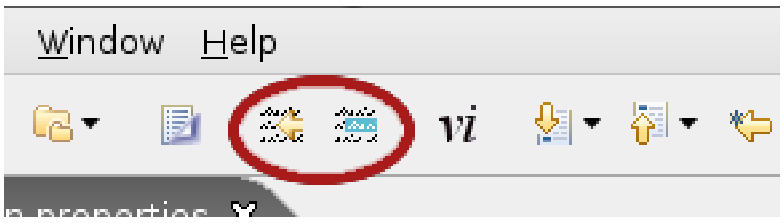


Figure 7.60: The Showoffset plug-in.

7.6 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 [GB04].

7.6.1 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 your Eclipse installation 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 7.6.3 for more details.

7.6.2 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 RubyRefactoringWizard(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";
  }
}
```

The method `createChange` gets your refactoring's changes 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 and 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", file);
String insertText = "Text to insert";
int insertPosition = 17;
TextEdit edit = new InsertEdit(insertPosition, insertText);
change.setEdit(edit);
```

Class Diagram

7.6.3 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 give an 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 pop-up 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 pop-up menu entry is clicked.

If you want to know more about patterns for resource management, please read Pattern Oriented Software Architecture, Vol.3 [KJ04].

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 source of most existing Eclipse plug-ins. The rule tells you to use this source of information and look at those plug-ins to get an

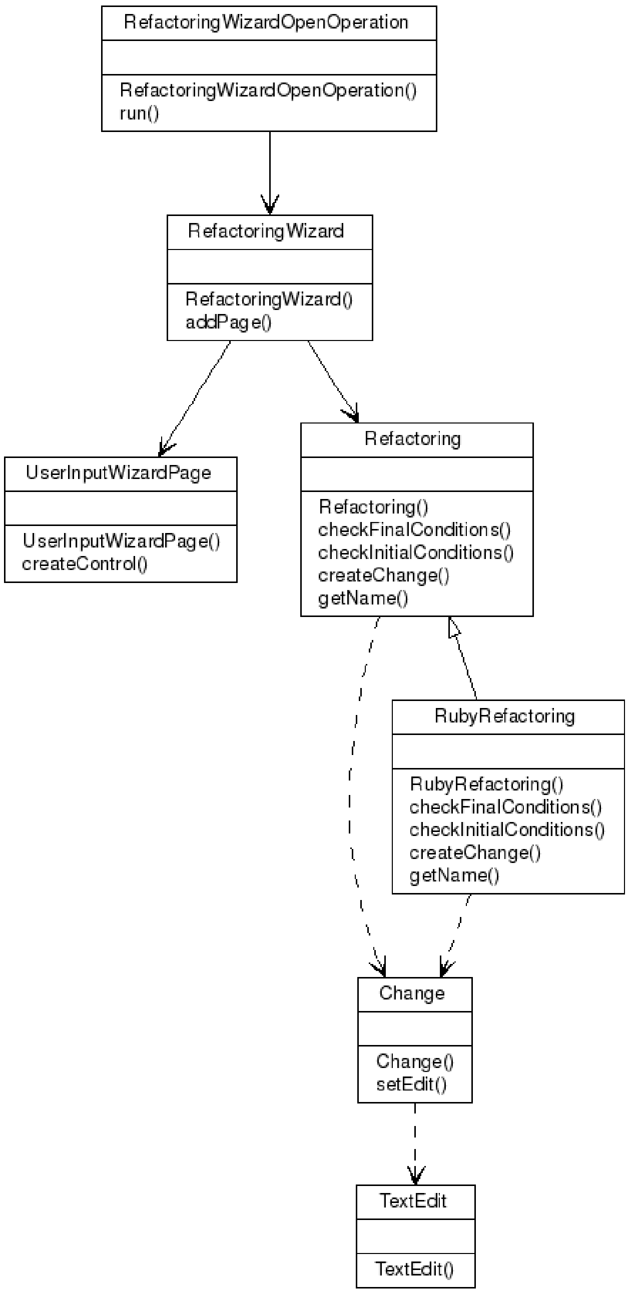


Figure 7.61: Eclipse Refactoring Support Diagram

idea how to go on. So you (the monkey) look at existing things, copy 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.

7.6.4 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 [GHJV97]. 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 [GHJV97].

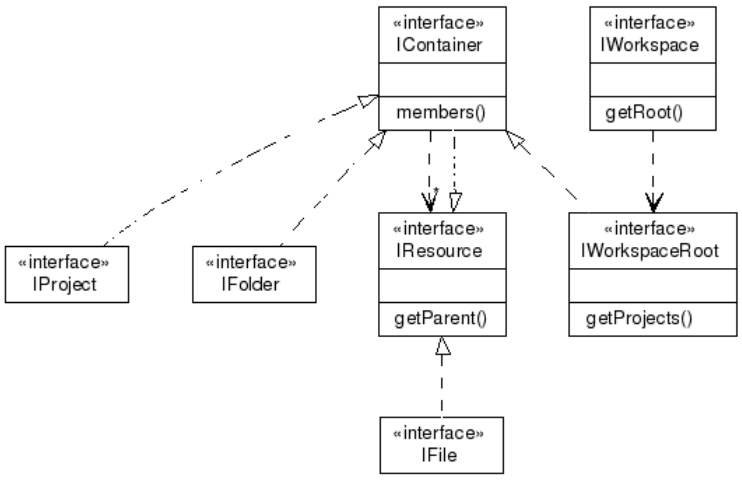


Figure 7.62: Resource Management Diagram

Code Samples

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

```
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);
```

7.7 Refactoring Implementation Guide

This section gives an introduction to the several steps needed to create a new refactoring. We take the Rename Class refactoring as example.

7.7.1 RefactoringClass

- Create a new package `renameclass` in `org.rubypeople.rdt.refactoring.core` with the name of the refactoring.
- Inside the new package, create the class RenameClassRefactoring that is derived from RubyRefactoring.
- Create a class RenameClassConfig. This class will hold the data for the refactoring over the different steps.
- Create a class RenameClassEditProvider that derives from one of the different EditProviders.

- In the `org.rubypeople.rdt.refactoring.ui.pages` package, create a class that will hold the user interface elements of the refactoring. The class must derive from UserInputWizardPage.
- Finally, create the RenameClassConditionChecker which extends RefactoringConditionChecker.

The constructor of the RenameClassRefactoring should have a similar structure:

```
RenameClassConfig renameClassConfig = new RenameClassConfig(/*...*/);
RenameClassConditionChecker conditionChecker = new RenameClassConditionChecker(/*...*/);
setRefactoringConditionChecker(conditionChecker);

if(conditionChecker.shouldPerform()) {
    RenameClassEditProvider editProvider = new RenameClassEditProvider(/*...*/);
    setEditProvider(editProvider);
    pages.add(new RenamePage(/*...*/);
}
```

The initialization of the refactoring specific data should happen in the condition checker's `init` method, which only gets called if no errors happened.

7.7.2 plugin.xml

To integrate the plug-in into the menubar and the context menu, several steps are necessary:

- First you create a new action in the `org.rubypeople.rdt.refactoring.action` package that is derived from the WorkbenchWindowActionDelegate.
- In the plugin.xml, add the following code:

```
<action class="org.rubypeople.rdt.refactoring.action.RenameClassAction"
    definitionId="org.rubypeople.rdt.refactoring.command.RenameClass"
    id="org.rubypeople.rdt.refactoring.renameClassAction"
    label="%rubyRefactoring.RenameClassLabel"
    menubarPath="org.rubypeople.rdt.refactoring.refactoringMenu/
        org.rubypeople.rdt.refactoring.refactoringGroup"
    style="push"
    tooltip="Renames a class."/>
...
<command
    categoryId="org.rubypeople.rdt.refactoring.commands.refactoring"
    id="org.rubypeople.rdt.refactoring.command.RenameClass"
    name="%rubyRefactoring.RenameClassLabel"/>
```

- To add the action into the context menu, add the following line into the RefactoringActionGroup class:

```
submenu.add(new RefactoringAction(RenameClassRefactoring.class,
        RenameClassRefactoring.NAME));
```

7.7.3 GUI

Your wizard page needs to override the `createControl` method in which the SWT GUI can be created. It is recommended to create a new composite with the composite you get from the caller as parent. You should do everything with your own composite from now on. You maybe need to disable the forward button, for example if an invalid option is set, this can be achieved by setting `setPageComplete` to `false`. Errors and warnings in the page can be set using the `setErrorMessage` or the more generic `setMessage` methods.

7.7.4 EditProvider

The concept of the EditProvider is described in section 7.2.6. Generally all edits in the code are generated in EditProviders, which make the realization of a refactoring much easyier. As almost all of the common methods are implemented in the baseclasses only the essential parts of the edit-creation have to be realized by the user himself.

7.7.5 DocumentProvider

DocumentProviders are responsible to handle all the files an RDT-Project or a TestSuite has to deal with. There is an implementation specifically for the Eclipse environment to handle the its files. We abstracted this interface to be able to work with more fundamental classes like Strings and Streams. Furthermore we have implemented DocumentProviders that can give ClassNodeProviders which simplify the access to the classes in a project. They can easily accessed even when they are spread over several files an class parts.
Beside the DocumentProvider there are several other classes that support in getting specific nodes from the AST. Extremly helpful are the SelectionNodeProvider, when working with known positions in a file, the NodeProvider when searching scopes for nodes or subnodes and the NodeFactory for creating new nodes when inserting.

7.7.6 Testing

We introduced our own small testing framework which can be easily extended with new tests just by creating new source and result files. More about this topic is described in the section 8.

7.8 Known Issues

7.8.1 Require and Load Statements

There are two methods in Ruby to include other existing files in the current file. These are statements load and require. The second one includes the files dynamically. In the current implementation of our refactoring plug-in we expect these commands always to be at the beginning of a document. This actually is not always the case. It would be a nice feature if these statements were treated correctly.

One idea to achieve this would be to create one large document including all parts of the project code. This would simplify some refactorings. The most challenging opponent in this game would probably be the resource management.

7.8.2 Type Inferencing

In many refactoring it was a huge advantage if one could determine what type is used where. The refactorings are actually working without this feature but when this would be realized it could be an extremely helpful thing when trying to select refactoring targets. But all refactorings would have to be adapted for the new functionality.

7.8.3 Going Back in the Wizards

As in most refactorings the data in our configs is modified on the pages of the wizard and we did not figure out how to reset the refactoring when using the back button, in some refactorings this leads to errors. So yet it is better restarting a refactoring than going back in the wizards an changing the parameters.

7.8.4 Treatment of Modules

The refactorings have been designed to handle the code in classes an in most cases outside classes. They usually lack the testing for compatibility with modules. In the normal case the rafactorings should work as intended but this is not assured in every case when dealing with modules.

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 has a component that implements the IEditProvider interface. And that is where the main tests for the refactoring take place.
Some of the implemented refactorings 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 File Driven Testing

Refactorings basically just change one code representation into another one, where the input and output are plain text. They are configured by setting some properties taken from the user, but they should not depend on that. We thought that the easiest way to implement tests for such refactorings should be based on the same principle: We take an input file, read some properties from another file, run the refactoring and compare the result to a third file. The overall workflow of the refactoring is always the same, it does not depend on the input or the properties nor should it care if it is run via the GUI or not. New tests should be created simply by adding more files, no messing with Java code or compilation. This is what we call File Driven Testing.

Of course we need to write some code initially, you need to instantiate a DocumentProvider, which works on test files and not Eclipse Resources, the configuration, edit provider and you need to read and apply the settings from the properties file. The comparison of the result and the expected content is then done by the framework.

After you adapted the refactoring to the testing environment, the last step to create a test suite that contains a pattern to match the file name of your test files – this is how the tests are found. Take a look at the

New tests can now be created without much effort and could even be done by someone who does not know about the internals of a refactoring.

Structure of Test Files

The tests are usually split into three files: source, destination and property file. This works fine as long as the refactoring affects just one file. If multiple source and result files need to be compared, the MultiFileTestData document provider comes into play. It works with the test's property file which needs to have two additional properties to describe the dependant files:

- activeFile: String - This property represents the name of the file that should be represented as active in the editor. For this property there have to be two files with the file names following the pattern:
  `refactoring_<NR>.<activeFilePropertyValue>.source` and
  `refactoring_<NR>.<activeFilePropertyValue>.result` – these files represent the original file content before and the resulting file after the refactoring.
- requiredFiles: List of Strings - This comma-separated value contains a list of all files besides the active file, that are used in the current test and need to be found through the document provider. For every file specified, two files have to be created: `refactoring_<NR>.<requiredFileProperty>.source` as source before the refactoring and `refactoring_<NR>.<requiredFileProperty>.result` as the outcoming of the refactoring.

The `<NR>` stands for an arbitrary number that distinguishes the tests and `refactoring` should be replaced with the actual name. Note that the exact naming can vary in the different refactorings, that is no problem as long as the test suite contains the right regular expression to match the files. For example, the package of the Move Field refactoring has the following content:

```
move_field_test_1.activeFile.rb.result.rb
move_field_test_1.activeFile.rb.source.rb
move_field_test_1.includedFile.rb.result.rb
move_field_test_1.includedFile.rb.source.rb
move_field_test_1.test_properties
move_field_test_2.activeFile.rb.result.rb
move_field_test_2.activeFile.rb.source.rb
move_field_test_2.includedFile.rb.result.rb
move_field_test_2.includedFile.rb.source.rb
move_field_test_2.test_properties
move_field_test_3.activeFile.rb.result.rb
move_field_test_3.activeFile.rb.source.rb
move_field_test_3.includedFile.rb.result.rb
move_field_test_3.includedFile.rb.source.rb
move_field_test_3.test_properties
move_field_test_4.activeFile.rb.result.rb
move_field_test_4.activeFile.rb.source.rb
move_field_test_4.test_properties
MoveFieldTester.java
TS_MoveField.java
```

As you can see, there are 4 different tests where each but the last one has an included file. The content of the first test's property file looks like this:

```
activeFile=activeFile.rb
destinationFiles=includedFile.rb
...
```

Inside the refactoring, the file is then known as `includedFile.rb`.

8.1.2 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 resulting 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.

8.1.3 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 ITreeContenProvider is tested using the validate method. This method compares each tree entry the ITreeContent-Provider 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 have 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 adaptions

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 of the Term Project

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 are 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 quiet 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 Results of the Diploma Thesis

During the diploma thesis, we had much more time to concentrate on creating refactorings contrary to the term project. We could estimate our plan quite good since we had a much better understanding of the different components we needed to use. Additionally, we did not need to spend any time to become acquainted with the code.

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 Trac[1].

[1] http://r2.ifs.hsr.ch

9.4 Outlook And Further Work

The next steps include the rewriting of the comment handling part for JRuby and the committing of our refactorings into the official RDT repository. We also need to check if our refactorings still work with the latest version of JRuby, as far as we know some changes to the representation of scopes have been made which we must adapt.

Of course there is still potential for extensions in our refactorings. As always with software, they certainly are not error free and could be improved in functionality too. A larger extension would be to adapt the refactorings for RadRails, for example to allow the renaming of views, controllers or even the models in the database.

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.

Appendix

Field Reports for the Term Project

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 planed 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 allways 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 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 foreward to the diploma thesis.

Field Reports for the Diploma Thesis

Lukas Felber

The first thing when I look back on our diploma thesis is, that it was an overall success. The teamwork in our group was really good. We had a lot of fun together and helped each other out when encountering problems. We rarely needed the help of our superviser anymore, since we allready knew our environment and encountered seldomly larger problems we could not solve ourself. But even if we did, he always gave us good advice if this was needed.

Again I learned that you can never tell the time needed to implement a certain part of the project. You can create a timetable as accurately as you like, it will always stay a rough guideline and nothing more. Who could say that an Extract Method was less effort than a Move Field.

This project was a big challenge for me and my team mates. I thank both of them for their help and the good job they did working with me on this project.

Mirko Stocker

For me, the work did not start on the official first week but a week earlier when we got our own server to run Trac and CruiseControl! Thanks again for that, he was much faster and we did not have to rely on my cablecom connection.

Looking back to the start of the project, I remember that I seriously doubted our, in my opinion, way to optimistic plan. But when I look at our list of open tickets now, there is just one refactoring left. So I am really proud of our achievements. Of course the project is not yet finished, we still have to get it back into the official RDT. I hope we get time for this or we will certainly do it in our spare time.

I also want to thank my fellow co-workers for their engagement and enthusiasm. While the work sometimes was hard and we thought we were overwhelmed by Ruby's subtleties, together we were able to manage it.

Thomas Corbat

This is probably the last text part I'm writing for this project as a student. I'm looking back at eight very interesting, challenging and tough weeks. This diploma project provided lots of new impressions to me. It has been the first time working full time at a software development project. I experienced this work as quite diversified because there are so many different tasks to engage. In contrast to the term project I didn't work at the complex base of JRuby anymore. Like my teammates I concentrated on developing refactorings, which was unknown territory to me. Thus I first had to get an introduction into using our existing environment.
By the time we realized that not all the design decisions we have made in the past could be applied to the new requirements the planned refactorings he. A lot of time we invested in keeping the already finished refactorings up-to-date. But I would never call it wasted time, as all these changes made the whole plug-in more consistent, easier to adapt and more robust.
One other thing that defied us several times were the special features of Ruby which let the code behave in ways never expected when one's used to Java or other static typed languages.
I'm happy to be allowed to work in this team. I think our cooperation was great. Every time one of us got stuck we engaged the difficulty together and usually overcame the obstacle. In the very tricky cases our supervisor always gave us a hand, which forutnately we didn't need that often during this project.
I'm proud to have contributed to this project and to the world of Ruby.

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 of the term project. The diploma thesis was done with Eclipse 3.2 and even 3.3M4 because of the colored merge view.

Cruisecontrol

To automatically build our code we installed Cruisecontrol[2] 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 Zgraggen, the Article on Build and Test Automation for plug-ins and features [Bar05] 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 outweighting the time we spent for the setup. Cruisecontrol was configured to send all notifications via e-mail so we always knew whether our latest commits broke the build or not. Of course the documentation was built on every commit and stored with the build artifacts too.

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 Trac[3]. With Trac, we even got a ticket system, which we did not use that much until know, 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://r2.ifs.hsr.ch`.

Coverage Reports

During the diploma thesis, Mirko read in a book about coverage reports created by Cobertura[4]. He then chose to try them out with our project and they are now integrated into the build process[5]. They are very useful to determine code that needs more tests.

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 on-line source

[2] `http://cruisecontrol.sourceforge.net/`
[3] `http://projects.edgewall.com/trac/`
[4] `http://cobertura.sourceforge.net/`
[5] `http://r2.ifs.hsr.ch/coveragereport/`

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 of the term project can be seen in figure 9.1, the one from the diploma thesis in 9.2.

Tools

The plug-ins were developed in Eclipse, running on GNU/Linux. This documentation was written in LaTeX 2_εusing Kile[6], pdflatex and all other tools. The diagrams were generated with Diomidis Spinellis' UMLGraph[7].

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.

[6] http://kile.sourceforge.net/
[7] http://www.spinellis.gr/sw/umlgraph/

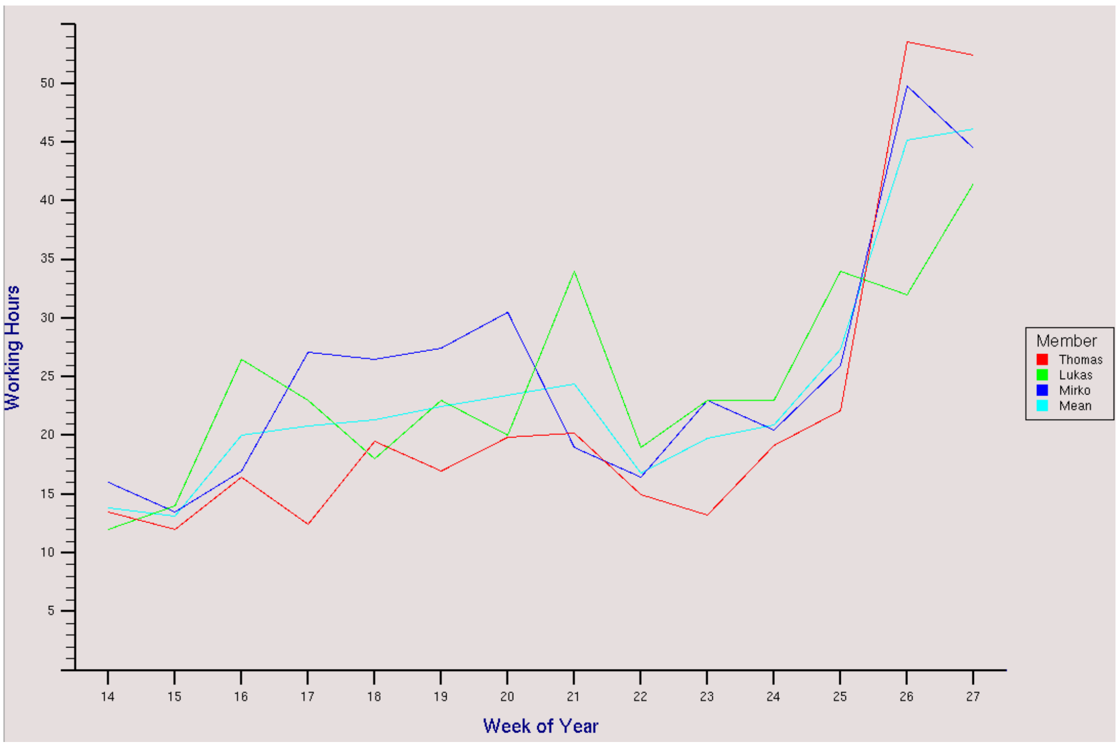


Figure 9.1: Statistics of our working hours for the term project.

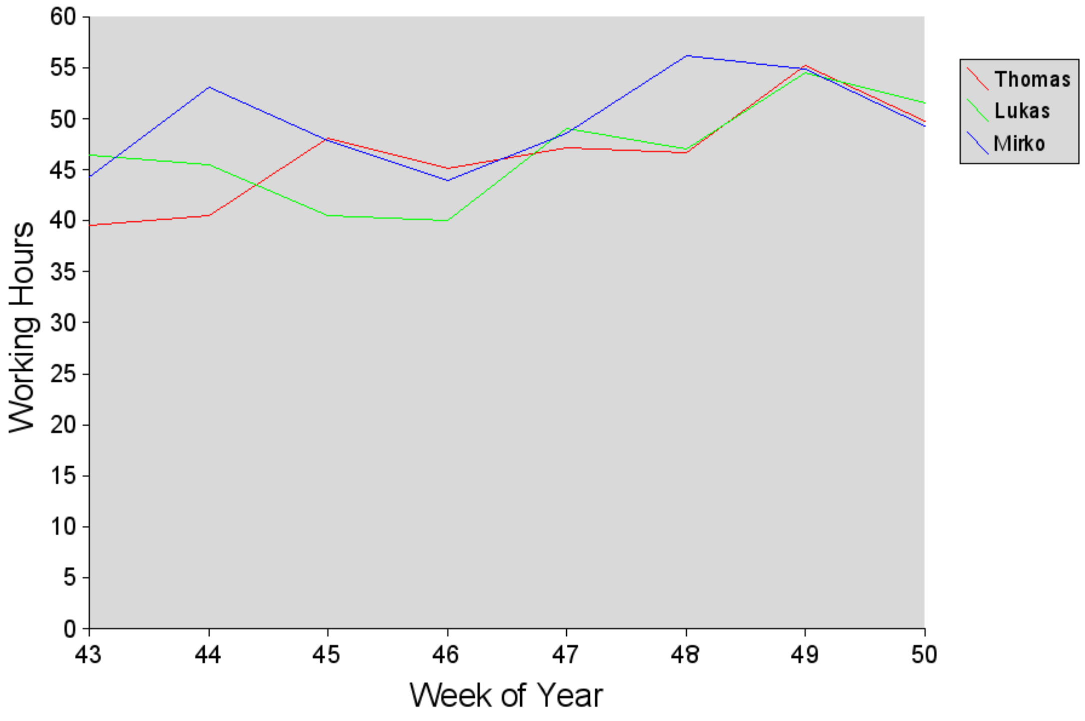


Figure 9.2: Statistics of our working hours for the diploma thesis.

Changes Since The Term Project

This section shows the parts of the document which experienced changes from the term project to the diploma thesis. Other parts of the documents were changed too, but most times not fundamentally and thus nod listed here.

Section 5.9.7 The comment issue with JRuby

Section 7.1.3 New

Section 7.1.4 New

Section 7.2 Descriptions of new components

Section 7.4 Descriptions of all new refactorings, almost completely new

Section 7.5 New

Section 7.7 New

Section 8.1.1 New

Section 9.2 New

Chapter 9.5 New field reports and statistics

	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 commment 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

Bibliography

[Bar05] BARCHFELD, Markus: *Build and Test Automation for plug-ins and features.* http://www.eclipse.org/articles/Article-PDE-Automation/automation.html, 2005

[BMR+96] BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons Ltd, 1996

[Fow99] FOWLER, Martin: *Refactoring.* Addison-Wesley Professional, 1999

[GB04] GAMMA, Erich ; BECK, Kent: *Contributing to Eclipse.* Addison Wesley, 2004

[GHJV97] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns.* Addison-Wesley Professional, 1997

[KJ04] KIRCHER, Michael ; JAIN, Prashant: *Pattern-Oriented Software Architecture, Vol.3 : Patterns for Resource Management.* John Wiley and Sons Ltd, 2004

[RB] ROBERTS, Don ; BRANT, John: *Refactoring Browser.* http://st-www.cs.uiuc.edu/users/brant/Refactory/,

[TFH05] THOMAS, Dave ; FOWLER, Chad ; HUNT, Andy: *Programming Ruby.* Pragmatic Bookshelf, 2005

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.