

UNIVERSITY OF APPLIED SCIENCES RAPPERSWIL
TERM PROJECT

Fall Semester 2009

Refactoring for Scala

scala.ifs.hsr.ch

Author
Mirko STOCKER

Supervisor
Prof. Peter SOMMERLAD

Abstract

This report documents the elaboration and the development of a library that provides automated refactoring support for the Scala programming language. Refactoring is a widely adopted practice among software engineers, but doing it by hand is tedious and therefore usually provided by integrated development environments (IDE). Several IDEs support Scala, but none of them provides a comprehensive suite of refactorings. The presented refactoring library is not tied to a specific IDE, but can be integrated into any one of them; support for Eclipse is already provided by this project.

Another goal is to make the development of new refactorings as simple as possible. This is achieved by extending the existing Scala compiler infrastructure with the necessary facilities to develop refactorings. As a proof-of-concept, the Extract Method refactoring has been implemented.

Management Summary

This documentation of a one-semester term project describes the development of a new refactoring library for the Scala programming language, performed at the University of Applied Sciences Rapperswil, Switzerland.

Motivation

Refactoring is a cornerstone of agile development and is applied by many professional software developers. Refactoring means to alter software in such a way that it does not change its semantics but improves its internal structure to make the code more maintainable. An example of a refactoring is the renaming of program entities. Refactoring can be done by hand, but is usually provided by integrated development environments (IDEs).

The Scala programming language, developed by Martin Odersky and his team at EPFL, runs on the Java virtual machine (or on Microsoft .NET alternatively) and excels with its unique combination of object-oriented and functional programming concepts. An other advantage of Scala is its interoperability with Java code, which makes it possible to develop mixed Java and Scala projects.

Scala's main goal, and also where its name comes from, is to be a language that scales, a scalable language.

Goals

The goal of this project is to create a library providing such automated refactorings, that can then be integrated into any Scala IDE. Current Scala IDEs have only limited refactoring functionality, if at all.

A further aim is to develop the library in such a way that the development of new refactorings is as simple as possible, to speed up the creation of new refactorings. To proof the developed concept, an implementation of a refactoring shall be aspired; including an integration into the Eclipse based IDE for Scala.

Results

During this term project, the foundations of the refactoring library have been laid. The various components that are needed to do automated refactorings are in place, but not

yet completed. They still sufficed to implement a usable refactoring, namely Extract Method, on top of them. Extract Method was chosen because it is a rather complex refactoring and thus a good showcase for the library.

The results also include an integration into the Eclipse based IDE for Scala, but has not yet been committed back into the official distribution of the Eclipse Scala IDE.

Outlook

A subsequent master's thesis will continue the project. While implementing more refactorings, the underlying library will evolve along with the refactorings, making the library more stable and easier to use; eventually enabling other developers without knowledge of the library's inner workings to implement new refactorings.

Other advancements can be made at the user interface level, making the refactorings easier to use. Yet another approach would be to implement so-called cross-language refactorings, from which projects that combine Java and Scala would greatly benefit.

In the near future, the library should also be integrated with as many IDEs as possible, to gain real world experience with it.

Contents

| | |
|--|-----------|
| 1. Introduction | 1 |
| 1.1. Problem Outline | 1 |
| 1.1.1. Objectives | 2 |
| 1.1.2. Vision | 2 |
| 1.1.3. Desired Results | 2 |
| 1.2. Contents of This Report | 3 |
| 2. Scala Internals | 4 |
| 2.1. Compiler Organization | 4 |
| 2.2. Scala AST | 4 |
| 2.2.1. Trees Hierarchy | 7 |
| 2.2.2. Working With Trees | 7 |
| 2.2.3. Trees Example | 8 |
| 2.2.4. Symbols | 11 |
| 2.2.5. Obtaining an AST | 11 |
| 2.2.6. AST Oddities and Pitfalls | 11 |
| 2.3. Modifications to the AST | 12 |
| 2.3.1. Representation of Modifiers | 13 |
| 2.3.2. Retaining Imports | 13 |
| 3. Scala Refactoring Implementation | 14 |
| 3.1. Architecture | 14 |
| 3.2. Source Analysis | 16 |
| 3.3. Source Transformation | 17 |
| 3.4. Source Regeneration | 19 |
| 3.4.1. Comparison to Other Approaches | 19 |
| 3.4.2. The AST Dilemma | 20 |
| 3.4.3. A New Approach | 21 |
| 3.4.4. Merging | 25 |
| 3.5. IDE Integration | 29 |
| 3.6. Conclusion | 29 |
| 4. Extract Method Implementation | 31 |
| 4.1. Extract Method Revisited | 31 |
| 4.2. Implementation Details | 35 |

| | |
|--------------------------------|-----------|
| 4.3. Examples | 35 |
| 5. Outlook | 39 |
| 5.1. Results | 39 |
| 5.2. Next Steps | 39 |
| 5.3. Acknowledgments | 40 |
| A. Project Environment | 41 |
| A.1. Project Plan | 41 |
| A.2. Time Report | 41 |
| A.3. Tools | 41 |
| Bibliography | 43 |

1. Introduction

Refactoring is a cornerstone of agile development and a technique that has been adopted by software engineers all over the world. According to Martin Fowler [Fow99], 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.

Or in other words, “improving the design after it has been written”. Doing refactoring does not require a tool; it can be done by hand, but this is often quite tedious and should only be approached with a comprehensive suite of automated tests to back up any changes.

Many programmers today use integrated development environments (IDE), especially for statically typed languages like Java, where all popular IDEs provide reliable support for automated refactoring. All major IDEs that support Java also have support for Scala; these are the Scala IDE for Eclipse [Sab09b], the Scala Plugin for IntelliJ IDEA [ZP09], and the Scala Plugins for NetBeans [Net09].

Although Scala is supported on all three platforms, not to the extent of the respective Java support. One area where the gap is evidently is in supporting automated refactoring, and this is where this project comes into play. Except for IntelliJ – which supports renaming and the introduction of local variables – support for automated refactoring is non-existent. Even though a recent study by Emerson Murphy-Hill and colleagues [MHPB09] shows that refactoring tools are generally underused, refactorings like Rename, Move and Extract Method are used by many developers. A separate study by Gail C. Murphy and associates [MKF06] confirms these results.

1.1. Problem Outline

If Scala wants to be adopted by a broad audience, it needs strong IDE support, and this includes automated refactorings. I have worked on various projects implementing (Ruby [CFS07], C++ [GZS07]) and supervising (Groovy [KKKS08b], JavaScript) refactoring tools. Taking advantage of this experience, I believe that a sound refactoring tool can be built for Scala in due time.

1.1.1. Objectives

The refactoring tool should be IDE independent. It can then be used by all IDEs and other tools to refactor Scala code – fostering collaboration to get truly great refactoring capabilities instead of several half-baked implementations from competing IDEs (as the situation with other languages – e.g. Ruby – is today). The only requirement is that the IDE is able to run Scala code and that it integrates the Scala compiler, which will be used heavily by the refactorings.

The basis for a successful refactoring tool is the source code manipulation strategy – that is, how the program is modified. Just as important is how the changed source code is obtained (regarding its layout, comments, indentation, etc.). At the Institute for Software, we experimented with different strategies but have yet to come up with a silver bullet. Therefore, the first focus of this project will be to contemplate existing ways and to come up with a solution for this project.

Another goal is to make the development of new refactorings as simple as possible. Implementing a new refactoring should be as easy as writing a transformation that manipulates the program’s tree representation. Scala’s functional aspects (especially pattern matching) should lend itself ideally to the processing of such data structures.

The generation of the source code from the transformed tree should be a separate activity and work independently of the performed transformation. For the transformations, several granularities of abstractions will be needed. Basic building blocks to modify the source trees (e.g. “move x to y ”, “rename x to y ”, “create a new X at y ”) can then be assembled to complete refactorings (“cursor position is x , rename all references to y ”) and exposed together with the means to analyze the source code. The result of such a refactoring operation will be a set of changes the IDE can then present to the user.

1.1.2. Vision

Automated refactoring implementations are usually heavily tied to a specific IDE, and not only to the programming language. The only ones writing new refactorings are the developers of the IDEs. If this project succeeds, Scala will have a simple refactoring library to manipulate its source code on which refactorings and other code manipulation tools (e.g. code generators) can be built by Scala programmers and not only IDE adepts.

1.1.3. Desired Results

The desired results of this project are an API to refactor code along with an implementation of a concrete refactoring. This project shall also provide the integration into the Scala Eclipse plug-in. Eclipse was chosen because its Scala plug-in is also offered by EPFL – giving it an air of officiality – and because of my already available knowledge with Eclipse’s refactoring framework from previous projects.

1.2. Contents of This Report

The remainder of this report is organized as follows. Chapter 2 on the following page will introduce some of the Scala compiler's internals that are needed when doing refactoring; primarily this is the structure of the abstract syntax tree. After this foundation has been laid, Chapter 3 on page 14 introduces the basic architecture and the various phases of a refactoring, providing an understanding of the grounds on which the forthcoming refactorings will be implemented. This leads us to Chapter 4 on page 31, where the renowned Extract Method refactoring is introduced in Scala's ambiance. The chapter also features how this particular Extract Method implementation looks like, along with several demonstrations of refactored code. Chapter 5 on page 39 concludes with a review of the achievements and an outlook onto the upcoming follow-up thesis. Finally, Appendix A on page 41 describes the project's environment.

2. Scala Internals

This chapter introduces some of the Scala compiler's internals that are useful when doing refactoring, which is primarily the abstract syntax tree (AST) and symbols. The AST consists of different kinds of trees and can be traversed and transformed.

2.1. Compiler Organization

The Scala compiler operates in several phases, starting with the parser and ends after the target code has been generated. These phases can also be contributed by plug-ins, thus new language features can be introduced and used quite easily (for example, continuations are handled by a plug-in [RMO09]). In the first phase, the parser creates an initial – naked – AST which is subsequently enriched in later phases. For our refactorings, the important phases are *namer* and *typer*, which run right after parsing (all available phases and their order can be seen by running `scalac -Xshow-phases`, a description can be found in the manual pages of `scalac` [OO07]). After the typer has been run, all the type information we need is available.

To see how the code changes with later phases, `scalac -Ybrowse:<phase> <scala-file>` opens a graphical explorer for the code. `-Xprint` simply prints the code after the specified phase has been run to the console. With later phases, the code will look more like Java, with Scala language constructs like pattern matching replaced by nested conditional expressions. Figure 2.1 on the next page shows an example of how Scala code is transformed over several compiler phases.

The current Scala compiler is actually the new Scala compiler – `nsc` – and located in the `scala.tools.nsc` package. All further references to the location of Scala compiler packages and their contents in this chapter are meant relative to `scala.tools.nsc`.

2.2. Scala AST

The Scala AST consists of `Trees`, which derive from the abstract `Tree` class in the `Trees` trait (all AST specific parts of the compiler are located in the `ast` package). `Trees` are dependent types of the compiler to make sure that trees from different compiler instances cannot be mixed. The *structure* of the trees is immutable – there are no operations to insert new children or replace subtrees, this can only be achieved by creating a new tree. Other information like the position or an associated symbol or type can still be updated in later compiler phases.

We will start with a minimal valid Scala source file, which is a class, plus one member called name of type String.

```
class Person(val name: String)
```

After the namer phase (scalac -Xprint:namer) the implicit package and super classes are visible and the parameter has been transformed into a paramaccessor field (paramaccessor is a modifier flag and defined – with many others such as param for method parameters – in the nsc.syntab.Flags object).

```
package <empty> {  
  class Person extends scala.ScalaObject {  
    <paramaccessor> val name: String = _;  
    def <init>(name: String) = {  
      super.<init>();  
      ()  
    }  
  }  
}
```

The next phase – typer – turns the code into the following, adding explicit types.

```
package <empty> {  
  class Person extends java.lang.Object with ScalaObject {  
    <paramaccessor> private[this] val name: String = _;  
    <stable> <accessor> <paramaccessor> def name: String = Person.this.name;  
    def this(name: String): Person = {  
      Person.super.this();  
      ()  
    }  
  }  
}
```

And after an even later stage – cleanup, where most of the phases have been run and just before the code generation starts – even more things are made explicit.

```
package <empty> {  
  class Person extends java.lang.Object with ScalaObject {  
    @remote def $tag(): Int = scala.ScalaObject$class.$tag(Person.this);  
    <paramaccessor> private[this] val name: java.lang.String = _;  
    <stable> <accessor> <paramaccessor> def name(): java.lang.String = Person.this.name;  
    def this(name: java.lang.String): Person = {  
      Person.this.name = name;  
      Person.super.this();  
      ()  
    }  
  }  
}
```

Figure 2.1.: An example of how the syntax tree changes over time with different compiler phases applied.

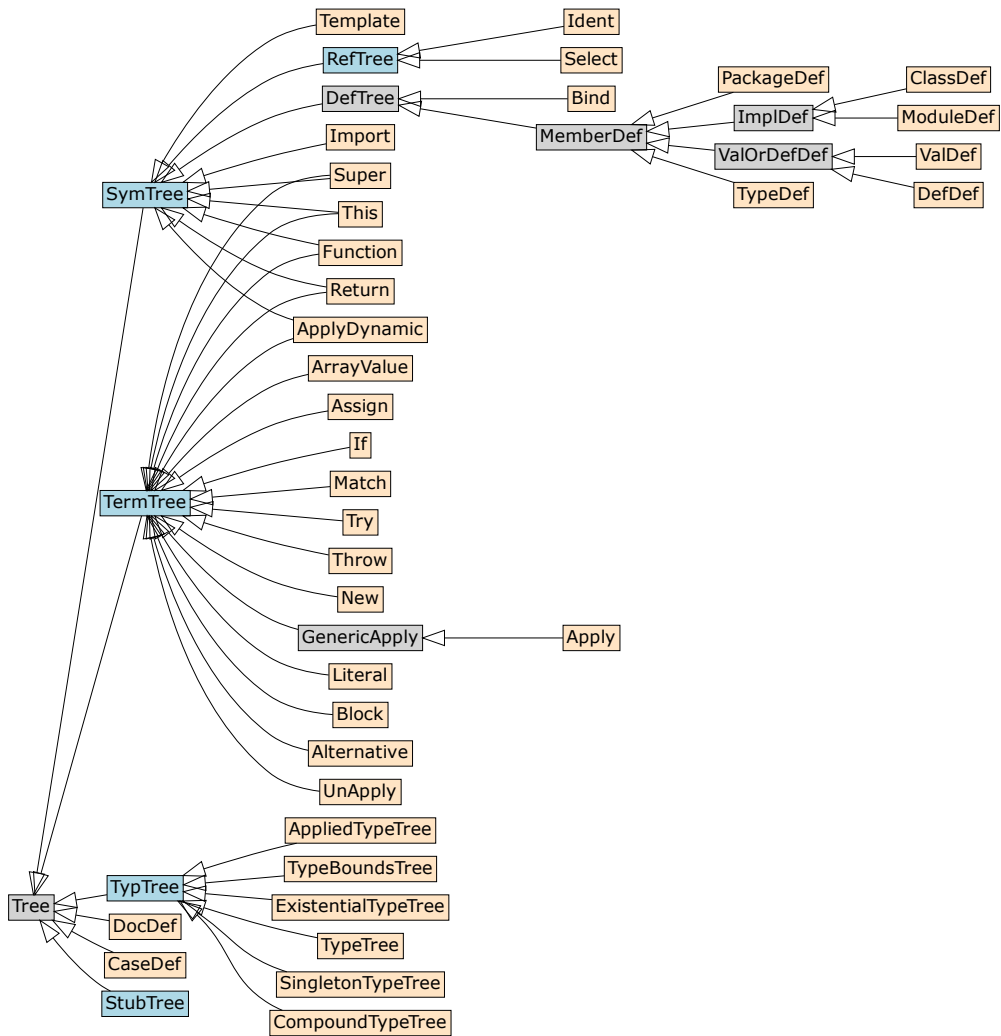


Figure 2.2.: The Tree class with some of its subclasses (some have been omitted for the sake of readability). The gray colored classes are abstract, blue ones are traits and the beige colored leaves of the tree are case classes that can be used in pattern matching.

```

case PackageDef(pid, stats) =>

case ClassDef(mods, name, tparams, impl) =>

case ModuleDef(mods, name, impl) =>

case ValDef(mods, name, tpt, rhs) =>

case DefDef(mods, name, tparams, vparamss, tpt, rhs) =>

```

Figure 2.3.: An excerpt of a standard pattern match on the Scala AST. Pattern matching makes the deconstruction of tree structures a breeze.

2.2.1. Trees Hierarchy

Figure 2.2 on the preceding page shows the subtype relationships of all tree classes. Notice that the leafs of the tree – the bisque colored nodes – are all case classes and thus can be used in pattern matching. Pattern matching eliminates the need for the Visitor design pattern [GHJV94] and makes working with the AST a far more pleasant endeavor than it is in other languages. An extract from a *standard pattern match* as described in `Trees.scala` is shown in Figure 2.3.

2.2.2. Working With Trees

The `Trees` trait also contains the `TreeCopier`, `Transformer`, and `Traverser` family of classes. The `Transformer` uses a `TreeCopier` to create the new tree, and the `Traverser` implements a full pattern match on all case classes to visit the complete AST.

Let us take a look at an example of a traverser that prints all classes with a name that starts with a lower-case letter. Note that `global` is a compiler instance and the dependee of the trees.

```

object checkClassNames extends global.Traverser {
  override def traverse(t: global.Tree): Unit = t match {
    case global.ClassDef(_, name, _, _) if name.toString.head.isLower =>
      println(name)
      super.traverse(t)
    case _ =>
      super.traverse(t)
  }
}

```

```
checkClassNames.traverse(someTree)
```

For the common tasks that traversing and doing a pattern match on a tree are, the `Trees` trait contains helper classes like `ForeachTreeTraverser` and `FilterTreeTraverser` that take a function from `Tree` to `Unit` or `Bool` respectively.

The `Transformer` will be used extensively in the refactoring transformations described in Section 3.3 on page 17.

Trees do not only have subtrees, but some of them also contain names and most relate to a `Symbol`, which are described later. All trees have a position as well, although they do not necessarily point to a position in a source file, as we shall see in a moment. Positions are defined in `util.Position.scala`. In addition to a position, all trees have a type as well, or at least the possibility to have a type, because not all trees represent something that has a type. An example of this is a package, which has the type `NoType`.

2.2.3. Trees Example

Figure 2.4 on page 10 shows the tree structure of a class with a single data member, as seen in Figure 2.1 on page 5. The tree has been generated from the syntax tree after the typer phase. The blue nodes denote trees that do not have a position (the tree's position is the `NoPosition` object). The bisque colored nodes are those that have a range position, that is, they cover a clearly identifiable part of the source code. The rest of the nodes are colored gray, meaning that they have a position but not a range. These are the trees that are generated by the compiler and usually have an offset position to mark where they approximately belong to. For refactoring, we are mostly interested just in those trees that have a range position.

We shall now take a closer look at each node in the tree.

PackageDef is the top level tree of each compilation unit, even if there is no explicit package specified in the source code (as it is the case in our example). A `PackageDef` has a list of statements as its sub trees.

Ident contains the identifier of the package. As can be seen from the color in Figure 2.4 on page 10, this code is not explicitly placed in a package.

ClassDef, Template define a class (or an object, in this case, we would have a `ModuleDef` instead of the `ClassDef`). The `ClassDef` contains the modifiers (visibility, case, abstract, etc.) and type parameters; the parameter list, parents, and the implementation body are both specified in the `Template`.

TypeTree, TypeTree specify the parents of the class. In this case, the first `TypeTree` instance points to `java.lang.Object`, which is implicit in this example, and `ScalaObject`, a trait that the compiler attaches to mark classes as coming from Scala.

Trees\$emptyValDef\$, TypeTree would be used to specify an explicit self-type.

ValDef, TypeTree contains the only explicit member of this class – the `val` name, with a `TypeTree` that holds the `String` type. We see that class

parameters are not treated any different in the AST than a member value or variable specified in the body of the class would be. The difference is solely in the modifier paramaccessor. ValDefs can also have an optional right-hand side tree that holds the assigned value or a block that computes it. A VarDef tree does not exist – variables are also represented by ValDefs with a modifier.

DefDef is the public read accessor for the above ValDef (remember that class parameters with a val instruct the compiler to generate a public getter).

TypeTree specifies the type of the *return value* of the function, not the function's type in the form $A \rightarrow B$.

Select is this DefDef's right-hand side and contains the tree on the right of the equals sign. A Select consists of a name – incidentally this is name in our example – and a qualifier to that name, which is represented by a child tree:

This is the AST representation of the self-reference this.

DefDef corresponds to the <init> method and is the constructor of the class. In contrast to the first method, this one has a parameter, a

ValDef, TypeTree . Actually, in the DefDef class, parameters are represented by a list of lists. This is necessary to support the groups of parameters in currying.

TypeTree specifies the result type of the method. The body of this method is contained in a

Block tree. In the previous method, the right-hand side is just a single statement which does not need to be further wrapped in an extra tree. A block consists of a list of statements, and an expression. The expression – the statement that gets executed last – is special because its result is also the return value of the method. This block contains an

Apply tree. Apply trees represent function calls, they *apply* an optional argument to a receiver, which in this case is

Select, Super , a call to the parent constructor.

Literal represents (), the instance of Unit and the return value of the constructor.

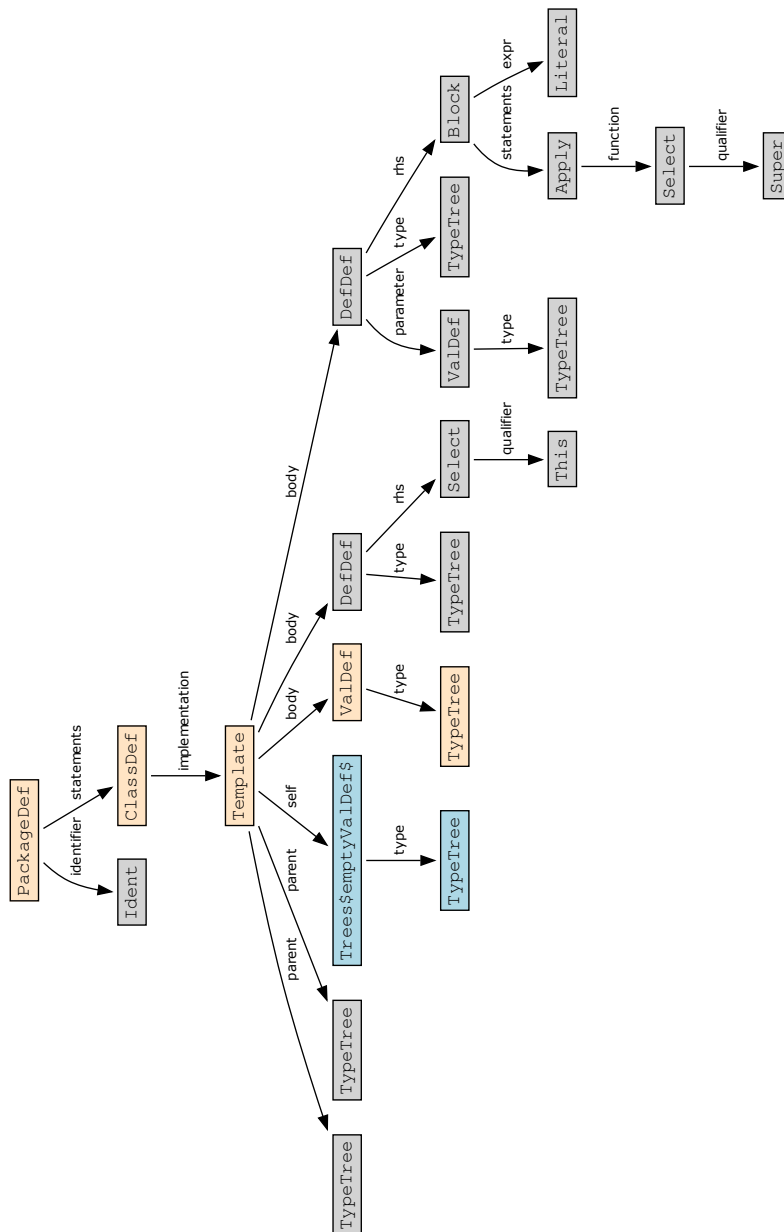


Figure 2.4.: An example of an abstract syntax tree, generated from the class `Person` (val name: `String`). The colors denote the different kind of positions the members of the AST have: bisque colored trees have a range position, and are the most interesting for us. The gray colored trees have a position, but it is not a range, so we can clearly see that these trees have been generated by the compiler and do not come from the source code. Finally, the blue trees have a no position information at all.

2.2.4. Symbols

Symbols are another important part of the program representation. As we can see in Figure 2.2 on page 6, the family of definition trees inherits from `SymTree` and introduce a `Symbol` together with a name, like for example a class or method. Complementary, `RefTree` is also a `SymTree` and references a certain symbol.

Symbols also form a hierarchy of their own; each `Symbol` has an owner, usually corresponding to a symbol in an enclosing scope. Symbols also provide a lot of information, there are almost one hundred is-methods defined on `Symbol` that can be used to, for example, find out whether a `Symbol` is a class, or whether a `Symbol` is an auxiliary constructor.

The Scala website contains three recordings of compiler internal walk-throughs [Ode09] with Martin Odersky who mostly talks about how symbols and types in the compiler work.

2.2.5. Obtaining an AST

Now that we know all the things we can do with an AST, how do we obtain one? The interactive `Global` class represents a compiler that can be instantiated and used to compile `SourceFiles` via the `typedTree` method. A `BatchSourceFile`, which inherits from `SourceFile` can be created from two strings: one for the filename and one for the content.

The following listing shows a minimal example of how the compiler can be instantiated and used.

```
val settings = new Settings
val global = new Global(settings, new ConsoleReporter(settings) /*error reporting*/)
new global.Run

val tree = global.typedTree(new BatchSourceFile("MyFile", "class A"), true /*force reload*/)
```

It is important to let the compiler execute once before using it – via `new global.Run` – otherwise some operations are going to fail because the compiler does not recognize its built in definitions (the symptomatic error is a failed assertion somewhere in the symbol table code).

Note that to use the compiler outside of an OSGi environment, it needs some additional setup to specify the compiler's classpath. An example of how this can be done is shown in the `scala.tools.refactoring.test.utils.CompilerInstance` object, the instructions are also provided in [Dic09].

2.2.6. AST Oddities and Pitfalls

As is probably the case with every language implementation, the Scala AST contains oddities, at least on the first glance.

For Comprehensions

When doing refactoring, it is crucial to know as much about the underlying syntax structure as possible. Then it might come as a surprise that the following two expressions:

```
val v1 = List(1,2,3) map (i => i * 2)
val v2 = for(i <- List(1,2,3)) yield i * 2
```

are transformed to these identical expressions (as described in Section 23.4 of Programming in Scala [OSV08]) by the parser.

```
val v1 = List(1, 2, 3).map(((i) => i.$times(2)));
val v2 = List(1, 2, 3).map(((i) => i.$times(2)))
```

This of course makes sense from the compiler writer's perspective, after all, it is an abstract syntax tree; but it can be problematic for other users of the AST.

Range Position Equality

When working with positions of Trees, one should be aware that the equality of RangePositions does not compare whether the start and end are equal, but just their points (only RangePositions contain a start and an end, others like OffsetPositions only point to an offset inside the file). This can lead to surprising effects – for example, consider the following listing.

```
if ...
else {
  try {
    ...
  }
}
```

Comparing the positions of the condition's else tree and else's body – the try term – for equality using == yields true. To compare the full positions, sameRange should be used.

2.3. Modifications to the AST

This section describes the problems and hurdles encountered when starting the project. Experience shows that the internal structures of programming language implementations are often not suitable for creating refactorings without foregoing modifications. Problematic areas are the correctness of positions assigned to nodes in the AST or simplifications and abstractions done in an early stage, i.e. while parsing. The situation is

not different in Scala. In the remainder of this section, problems and their undertaken solutions are shown.

2.3.1. Representation of Modifiers

According to the Scala Language Specification 5.8 [OO09], “member definitions may be preceded by modifiers which affect the accessibility and usage of the identifiers bound by them”. For example, in the following listing, the keywords `sealed`, `abstract` and `final` are all modifiers of the successive class (or object). This also includes the case modifier and the `val` and `var` keywords preceding variable definitions and declarations.

```
final class A
sealed abstract class B
private abstract case class C {
  val any: Any
}
```

A problem arises because modifiers are not contained in the position – highlighted in blue – of their respective syntax trees. If the modifier is the first element of a source file, then its position is not contained in any syntax tree.

Moreover, after parsing, modifiers are only represented as a flag in the concerned class and their position in the source code is not saved. As soon as comments are involved, handling modifiers becomes even more complicated.

This has been fixed by introducing a mapping from flags to positions in the `Modifier` class and has since shown to effectively solve the problem.

2.3.2. Retaining Imports

Import statements were originally not retained until the typer phase, were they would still be needed when doing refactorings (or other transformations, like organizing imports) that affect imports. A patch from Miles Sabin [Sab09a] corrected this and added import position and name information to the AST as well.

This concludes our introduction to the Scala AST, the following chapter constitutes the central part of this report and introduces the implementation of the Scala refactoring library.

3. Scala Refactoring Implementation

In this chapter, I will introduce the concepts and ideas behind the implementation of the Scala refactoring library. But before we start, I would like to spend a few words on a primary design goal.

Just as with every other software project, when implementing refactorings, the developer has to manage essential and accidental complexity. The essentials in refactoring are transforming a program representation such that the result preserves the semantics of the original. To be able to do that, one needs to thoroughly understand the programming language. The transformation includes various parts – new program trees need to be generated, dependencies of the code have to be tracked so that references can be modified accordingly, and all of this as generic as possible to be able to refactor all kinds of programs.

After the transformation, the trees need to be converted back to source code, and only those parts affected by the refactoring should lead to changes in the source file. This is a consequence of programs being stored as plain text files, and not necessarily essential to a refactoring.

Ideally, when working on a refactoring implementation, one should not be bothered with any of the source generating concerns – the accidental part of the refactoring. Based on former experience (see the comparison to other approaches in Section 3.4.1), I can say that wherever possible, this should be separated from the refactoring implementation and be provided by the underlying refactoring library.

To sum up, my goal was that developing a new refactoring should be no more complicated than writing the transformation of the tree.

In the remainder of this chapter, we will first take a big picture look at the architecture of the implementation and shall then elaborate each refactoring phase in its own section in more detail.

3.1. Architecture

The refactoring process can be separated into several stages, as shown in Figure 3.1 on the next page. Parsing of the source code is the responsibility of the IDE, or the compiler, and should – for the sake of efficiency – not be redone by the refactoring. This is also the reason why the Scala compiler's AST was used and not a new implementation. Other benefits include the already available infrastructure (tree visitors and transformations) and the familiarity for people who already know the Scala AST.

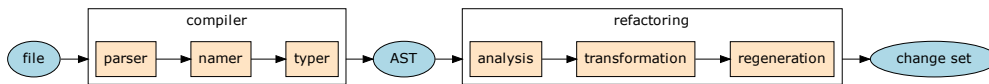


Figure 3.1.: The work-flow of the refactoring. The IDE uses the compiler to parse the source file and passes the resulting syntax tree to the refactoring tool. The result of a refactoring is a set of changes, viz. a patch, that the IDE has to apply to the source files.

The input to the refactoring is a set of ASTs, which are then processed and result in a set of changes to instruct the IDE in what way which file is to be changed. Typically, the IDE shows the changes the refactorings wants to perform and asks the user for confirmation.

The refactoring itself can roughly be separated into three stages, these are:

Analysis where for example an index of symbols is built and the code is analyzed to know whether a refactoring is possible,

Transformation takes an existing AST and modifies it according to the specific refactoring, and

Regeneration where the modified AST is transformed back into plain source code.

These three stages are also represented in the organization of the source code with the three analysis, transformation, and regeneration packages in `scala.tools.refactoring`. Each package provides a trait with the same name that offers the package's functionality to consumers. The `scala.tools.refactoring.Refactoring` class mixes in the three traits (see Figure 3.2) from these packages and combines them to form the basic refactoring functionality. This allows us to form several coherent traits that can be combined as needed.

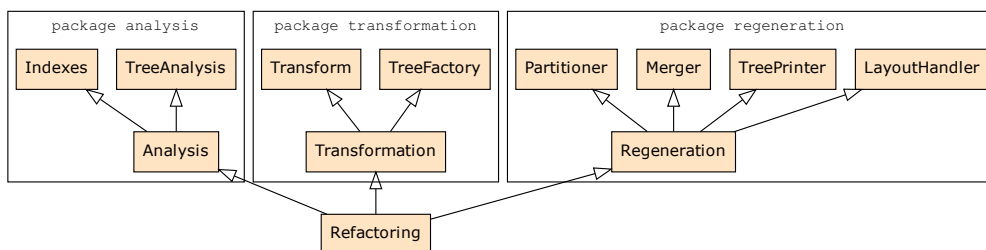


Figure 3.2.: An overview over the various traits that take part in the refactoring. The Refactoring trait is composed of the analysis, transformation, and regeneration traits from the respective package; those traits in turn are internally composed of other traits that have a specific functionality.

We shall now take a closer look at the individual refactoring stages, using the implemented Extract Method refactoring as our canonical example.

3.2. Source Analysis

Analysis of the source code is primarily the parser's responsibility. In Scala, after the typer phase has run, all symbols are in place and the type inferencer has done its work (see Section 2.2.4 on page 11 for more information on symbols).

To recap, the Extract Method refactoring takes a selection of consecutive statements and extracts them into a new method in the same class (variations would be possible, like creating local functions instead of methods). A call to this new method is inserted at the originating site. All used variables and functions that are not already accessible at the destination are passed as parameters; definitions that are used past the selection are returned from the new method and assigned to local variables.

In order to implement the Extract Method refactoring, given a selection of trees, the following information needs to be known about the source code:

Inbound dependencies are all usages of local symbols (references) in the selection.

From the selection's parent – usually the method we extract from – we take all of its children (the definitions inside the method) and subtract those defined inside the selection.

Outbound dependencies encompass all symbols defined in the selection which are referenced outside – i.e. escape – the selection.

As one might guess, inbound dependencies need to be passed into the extracted method and outbound symbols need to be returned from it. Of course, we need all type information of these symbols so we can form the appropriate method signature, but this information is already available from the symbols.

Not readily available is the relationship between a definition of a symbol – definitions extend from DefTree – and the references – which extend from RefTree – to it. But the symbol members of a definition-reference pair are equal by comparison with `==`, so by traversing the whole AST, all references to a certain symbol can be found and vice-versa for the definition.

This information would ideally be saved in an index by the IDE, updated when the sources change, and be passed to the refactoring (the refactoring implementation itself does not share any state between consecutive invocations). Right now, the index is built from scratch on every refactoring action; a future version might require the IDE to provide this index if the performance is too heavily impacted because of all the index building.

The index not only holds the definition and reference information, but also all children of a given symbol (the parent information is available on a symbol).

With these information, we are ready to go to the next phase, the transformation of an AST.

3.3. Source Transformation

The transformation phase is the heart of the whole refactoring process. The basis is the Scala compiler's Transformer class. In its default configuration, the transformer simply walks the complete AST and lazily copies (trees that did not change are not copied) each tree in it. By overriding the transform method, one can intervene and inject ones modifications.

As an example, let us implement a transformation that simply reverses the order of all class members.

```
val reverser = new Transformer {
  override def transform(tree: Tree): Tree = {
    super.transform(tree) match {
      case Template(parents, self, body) => Template(parents, self, body.reverse)
      case t => t
    }
  }
}

val result = reverser.transform(tree)
```

Transformations will be done very often, so it makes sense to simplify this process. Making use of Scala's partial functions and multiple argument lists for methods, we can define a function transform with the signature $Tree \rightarrow (Tree \rightsquigarrow Tree) \rightarrow Tree$. The first *Tree* denotes the root of the current transformation, which was the tree on the last line in the listing. Then follows a partial function that transforms one *Tree* into another one. The function finally returns a tree, the result in the listing. Using this function, the above example shrinks down to the following. (Which is also a nice example of how one can build custom control structures in Scala.)

```
val result = transform(tree) {
  case Template(parents, self, body) => Template(parents, self, body.reverse)
}
```

Alternatively, using Scala 2.8 named parameters and the case class' copy method, it could as well be written as follows.

```
val result = transform(tree) {
  case tpl: Template => tpl copy (body = tpl.body.reverse)
}
```

When multiple transformations are needed, the partial function can simply be extended with additional case clauses. Now, let us take a look at a more complex example: the transformation for the Extract Method refactoring.

The definitions of the blue colored values are not shown in the example. Except for `selectedTrees` – which is a `List[Tree]` – they are all of type `Tree`.

The method `replace` is a generic method that replaces sub-lists of a list with other lists (for example in `body`, the `selectedMethod` is replaced with `selectedMethod :: newDef`; consequently inserting the new method after the selected one). Finally, `mkBlock` simply turns a list of `Trees` into a `Block` tree instance.

Now that we know the role of each element in the example, let us take a look at the code and examine how it works.

```
transform(tree) {
  case d: DefDef if d == selectedMethod => {
    if(selectedTrees.size > 1) {
      transform(d) {
        case block: Block => {
          mkBlock(replace(block, selectedTrees, callNewMethod :: Nil))
        }
      }
    } else {
      transform(d) {
        case t: Tree if t == selectedTrees.head => callNewMethod
      }
    }
  }
  case tpl @ Template(_, _, body) if body exists (_ == selectedMethod) => {
    tpl.copy(body = replace(body, selectedMethod :: Nil, selectedMethod :: newMethod :: Nil))
  }
}
```

The first case clause takes care of replacing the selected trees with a call to the new method: once the selected method has been found, depending on the number of selected trees, a sub-transformation is started from the definition. The sub-transformation then continues transforming the tree and, in the case where multiple trees have been selected, replaces them with a call to the new method. If only one tree is selected, we simply replace the tree with the call.

Why do we need to enter a sub-transformation? There are two reasons for this: First, we do not know where the selected trees are located inside the selected method, so we need to travel further down the tree. Second, if we put the sub-transformations out into the first transformation, we would also replace the trees inside the extracted method. By running the transformation only in a sub-tree, we guard the rest of the tree from accidental modification.

The second case clause is much simpler, it looks for a template (`Template` represents the body of a class or object) whose body contains the selected method and then creates a copy of it with the new method inserted after the selected one.

3.4. Source Regeneration

After the AST has been transformed, the inevitable step of regenerating the concrete source code from the abstract tree marks the last step in the refactoring process. The source regeneration phase plays a crucial part in the applicability of a refactoring – the user will refrain from using a refactoring that wreaks havoc in his source files. The aims of the source regeneration phase are to:

minimize changes to a bare minimum that are necessary to accomplish the refactoring and nothing more;

retain formatting of code that is merely moved around in the file and not actually changed; and to

handle comments as sensible as possible – that is, not to lose a comment under any circumstance and to move comments along with its presumed target.

Before I start to explain how the source regeneration is done in this project, we shall take a look at how the AST is transformed back to source code in other projects.

3.4.1. Comparison to Other Approaches

This subsection describes two existing approaches to source regeneration that have been employed in previous projects I was involved, namely in creating refactoring support for the Eclipse Ruby Development Tools [CFS07] and the work on refactoring for the C++ Development Tools (CDT) for Eclipse [GZS07].

The Ruby Way

The Ruby Refactoring project used a rather simple approach to source regeneration; in retrospect, it probably was too simplistic. The main problem was that the refactoring implementation had to specify exactly which parts of the code need to be changed and how the result should look like.

Having the two activities – refactoring and source generation – interleaved with each other made the process more complex than necessary. In practice though, a separation is not easily achieved. Ruby is dynamically typed, which makes a refactoring tool's work immensely more complicated. A pragmatic approach is to make assumptions and ask the user for confirmation. This also means that all changes by the refactoring need to be clearly labeled and distinguished, because the user might not want to accept certain changes on purpose. For example, the refactoring might be too eager in renaming, proposing an incorrect modification.

The advantages of this process of manually specifying each change simplified the generation of the code. An elaborate pretty printer that could be configured to the users needs was used to generate source code, and no code that does not need to be

changed can accidentally be changed. On the other hand, handling of concerns that are common to all refactorings – for example the handling of comments – needs to a certain degree to be re-done in each refactoring, making the life of the refactoring developer even harder.

Another problem was that for certain refactorings, re-generating the complete changed source code was not acceptable – for example when doing an extract method, instead of re-writing the extracted code, it was cut-and-pasted instead. But this had to be done by the refactoring and made the implementation much more complex.

Ideally, introducing a new refactoring should be as simple as specifying the transformation, the rest should be handled by a general purpose component of the system.

The CDT Way

In contrast to the AST used by the Ruby Refactoring project, CDT's AST is not used by a compiler but was written solely for the IDE. Unfortunately, the ability to easily transform the AST has not been one of its design goals. Therefore changes to the AST are stored separately by so-called ASTRewrites, which allow for nodes to be inserted, removed, or replaced; transformations that affect children of already modified parent AST nodes need to be done on sub-rewrites.

Using these recorded modifications and the original AST, a custom visitor can then pretend to be visiting the transformed AST.

The first instantiation of the CDT refactorings [GZS07] used a simple pretty-printer to create source code for changed AST. The current version uses the afore mentioned visitor in combination with the pretty-printer to create the new source code. An optimization minimizes accidental changes to the layout of the source code: the new and original source code is compared – ignoring all whitespace – and only these differences are then used to create changes.

For more information how comments are handled in these two approaches, see [SZCF08].

3.4.2. The AST Dilemma

An *abstract* syntax tree by itself is by definition unsuitable for layout preserving source regeneration; if all information from the source code were contained in the AST, it would not deserve the label *abstract* anymore.

The Scala AST printed in Figure 3.3 on the following page is an example that demonstrates the inadequateness of an abstract syntax tree to create source code. The graph shows a class with the sealed and case modifiers and two constructor parameters forename and surname.

We can see that only the essential parts of the source code are represented in the tree, this excludes all white space, comments but also certain keywords like class and extends that are not modifiers.

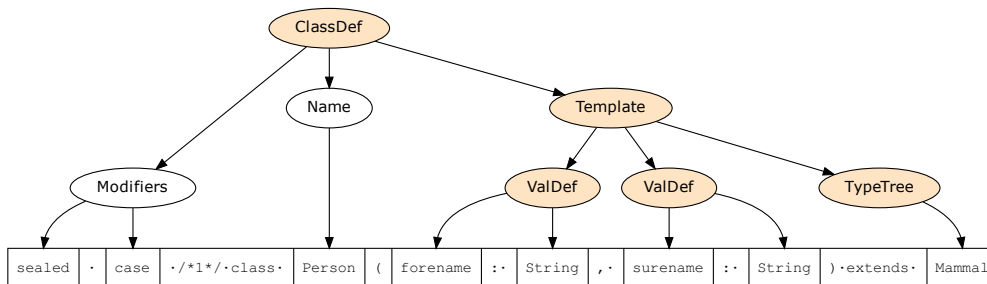


Figure 3.3.: A simple example of a Scala AST, we can see the hierarchy of trees (only the colored nodes are represented as Trees, name and modifiers are simply attributes of their parent Tree). The corresponding source code is shown at the bottom.

For example, when modifying a list of parameters, the comma separating the parameters has to be considered, but the AST has no means to represent that.

3.4.3. A New Approach

The solution developed in this project uses the AST of a compilation unit together with the underlying source file to build a new representation that will be used to regenerate source code; automatically preserving the full layout and all comments. This agrees with the design decision that the user of the refactoring should be bothered as little as possible with the source regeneration concerns but should only be concerned with modifying the AST, without having to keep track of changed trees or using special facilities to make transformations as it is the case in the CDT refactorings.

Source Partitioning

The basic idea is simple: a source file is in essence a linear sequence of characters, which is tokenized by the parser and eventually arranged into an AST. We have seen that the AST is too abstract to be useful for exact code regeneration, so we forsake that step and go back to a simple stream of tokens. To avoid confusion with the parser's tokens, we shall call our code representation a stream of *source fragments*, or just *fragments* in short.

This stream of fragments can easily be constructed from an AST and its source file. The source file is only needed to preserve layout, in the case where a source file is not available – i.e. when the AST consists of new trees constructed during the refactoring process – the absence of the file is no problem, a default layout can be used for new code or derived from similar existing code fragments. Figure 3.4 on the next page shows a simple example of a partition into fragments.

```
case class Person(name: String)
```



Figure 3.4.: An example of how source code is split into fragments. Essential fragments are colored bisque and layout fragments are white.

Fragments can be categorized into two distinct groups: so-called *essential* fragments and *layout* fragments. Essential fragments are the names, types, modifiers and literals of the source, whereas layout fragments contain all white space, comments and the various braces that are used to group the program's expressions. In principle, it would be possible to recreate a semantically equivalent program from just the information of the essential parts. In Figure 3.4, essential fragments are colored bisque and layout fragments are white.

The generation of the fragments can be done in two steps.

1. An existing AST is traversed and the essential parts of the tree are converted to fragments. Essential fragments know the AST they represent, the source file and their exact position in it.
2. From the existing fragments and their position information, we can also calculate all the layout fragments – the blanks between the positions of the fragments – of a source file and interleave these with the essential fragments.

The fragments now contain the full source code and re-creating the source code is as simple as concatenating all fragments. If no changes are made to the AST, then the result will be identical to the original. This trivial case is not of much practical use to us, after all, refactoring should have an impact on the code.

Now, how exactly does this help us with re-creating the source code after a refactoring? By repeating the first step above with the refactored AST, we get a second set of fragments (the second step would not make much sense, after the transformation, the trees are now in a completely different order). The basic idea is that we can now traverse the fragments from the refactored AST, look up the layout fragments from the original AST and thus regenerate the source code with all layout information from the original source.

Unfortunately, there exists one additional complexity that forces the stream of fragments into a *tree of fragments*. Programs form trees; in the source code, a new level of a tree is often formed by explicit braces or the association rules of the language. The

```

class Person {
  def name = {
    "Mirko"
  }
}

```

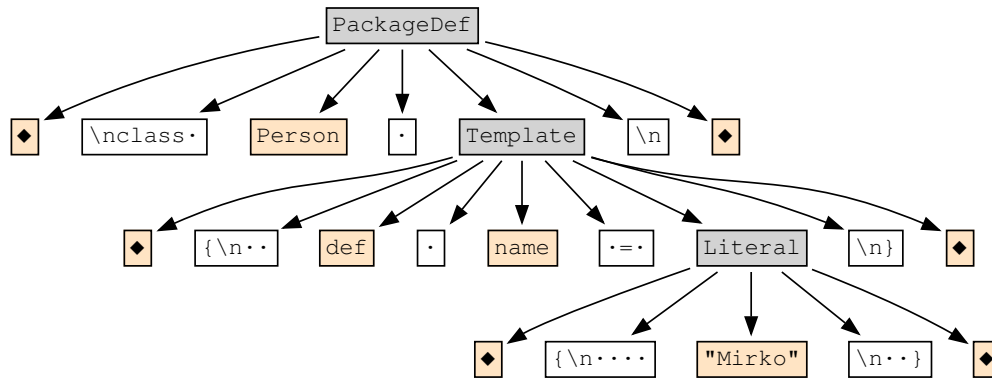


Figure 3.5.: Representing the above source code as a tree of fragments. The gray colored nodes in the tree are those introducing a scope, which is usually where braces are placed when writing code.

complexity for us comes from all the various *braces* that can be used in the program. If we take a look at one-dimensional stream of such fragments, we have no information to which original fragment a closing parenthesis belongs. This is why we need to introduce a notion of *scoping* into our fragments representation.

With scopes, the stream becomes a simple tree. Each scope has an artificial begin- and end fragment that marks the area of the scope. An example can be seen in Figure 3.5. This *normalized* AST (the terminology is due to [Par09]) is much simpler than the original *irregular* AST: there's no restriction on the kinds of children a scope can have, and there are only a handful of different classes of fragments: scopes, essential, and layout fragments.

Layout Handling

We can now generate a tree of fragments from the AST before and after the refactoring. All that remains is to fill in the blanks between the refactored essential fragments with some layout. The problem is that two fragments that were originally adjacent can now be at completely different positions, so the process of filling in the blanks becomes much more complex.

Basically, an essential fragment has leading and trailing layout, the corollary is that each layout fragment has to be split between two essential fragments. This process I

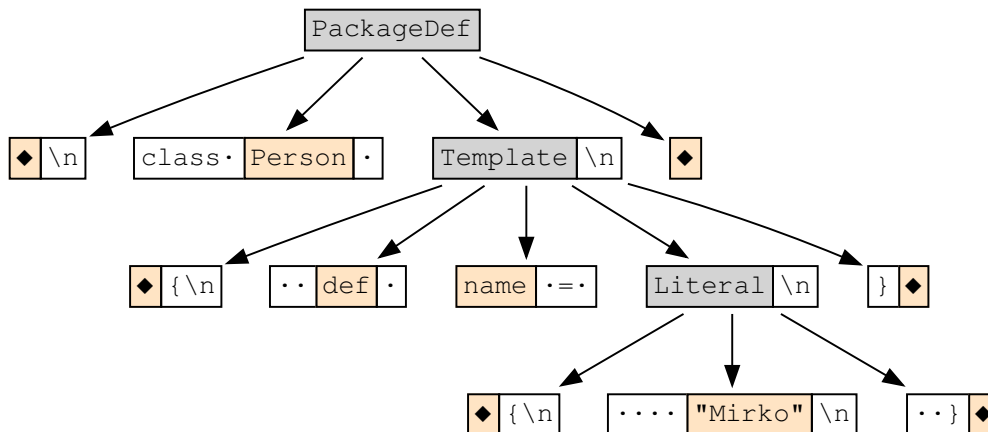


Figure 3.6.: The same source code from Figure 3.5 on the preceding page, but with the layout code associated to either one of its neighbor fragments.

call *layout splitting*. Figure 3.6 shows how the layout from Figure 3.5 on the preceding page is split between the essential fragments.

Unfortunately, there can also be parts of the layout that can not be unambiguously associated with either the left or the right fragment – for example, a comma in a list of parameters. Depending on the refactoring, such a comma has to be removed (e.g. if there’s now only one parameter) or introduced (if we suddenly have more than one parameter) to form a valid program. In our case, the comma is not associated with any fragment but simply discarded.

To fix the process, an essential fragment can specify required layout that has to be present before or after itself. For example, while traversing a list of parameters, we simply specify that there has to be a comma separating the parameters.

Layout splitting is also concerned with the comments, which are in the process associated with a certain essential fragment. The quality of the source generation aspects of a refactoring thus largely depends on the layout splitting algorithm. A few examples of rules to split layout are:

- If the layout contains an opening parenthesis, everything to the left including the parenthesis belongs to the left fragment, the rest to the right.
- A comma splits into left and right, but the comma itself belongs to neither.

All that is left now is to merge all these fragments into the resulting source code.

3.4.4. Merging

To generate the resulting source code after a refactoring, we need the the fragments tree of the original AST *Original* as well as the essential fragments of the changed AST *Changed*.

The creation of the final stream is done as follows. The process starts with the top-level scope in *Changed* and iterates over all child fragments in pairs: (*current*, *next*). (The actual process contains an optimization, if two nodes are still in the same neighborhood relation, then we don't have to split the whitespace at all.)

1. If *current* is a scope, then recurse the whole process with *current* as scope. Otherwise, render *current* as a string and add it to the output stream.
2. The layout code between *current* and *next* is calculated as follows.
 - a) Lookup *current* in *Original* and select all layout parts up to the next essential fragment. Split the layout code and retain its left part, that is, the half that is associated with *current*.
 - b) Lookup *next* in *Original* and select all layout parts in front of it, including the previous essential fragment. Split the layout code between the previous essential fragment and *next* and retain the second half that is associated with *next*.
 - c) These two layout fragments form all the existing layout that separates *current* from *next*.
 - d) Make sure *current* and *next*'s specific pre- and post-required layout is satisfied. If necessary, insert the required layout code into the existing layout.
 - e) Fix the indentation of the layout code.
3. Add the layout code between *current* and *next* to the stream.

Concatenating all elements of the output stream yields the final source code. As a precaution, the implemented algorithm also keeps track whether a scope has changed at all, and if not, the complete scope of the original source code is reused. This makes the whole process more stable for the time it is still under development. Before showing an example of the algorithm, we shall take a look at how changes to the indentation of the code are handled.

Indentation Fixing

To correct the indentation of the code, each scope knows (from looking it up in the source code or assuming a default indentation) how much it is indented, relative to its enclosing scope. Using this information, we can determine a relative indentation for each fragment in the scope to its enclosing scope. Using this information, when an fragment changes its scope, we can use the calculated relative indentation of the

fragment in combination with the new scope's indentation to get the new indentation. Using this technique, we can retain all indentation, even irregular ones like the following.

```
def calculate {
  val one = 1
  val thousand = 4
}
```

The definition of calculate has an indentation of two spaces and it opens a new scope, containing its body. The value one has an absolute indentation of nine spaces; or seven spaces relative to its enclosing scope. The second value has a relative indentation of two spaces. When moving these two values to a scope with a different indentation, we can easily indent them using their relative indentation together with the new scope's indentation to keep the same appearance.

Merge Example

Let us now take a look at a simple example of the merge algorithm. We start from the code: `class LR(left: Int, right: String);` or in the fragments notation:

◆ `\nclass·` `LR` `(` `left` `::` `Int` `,` `right` `::` `String` `)\n` ◆. The transformation we're going to apply is to reverse the list of arguments in the class. To split the layout, we are going to use the following 3 rules:

Left Parenthesis If there is an opening parenthesis, the parenthesis and everything left of it are associated with the left fragment, the remainder with the right fragment.

Right Parenthesis If there is a closing parenthesis, everything left of it is associated with the left fragment, the parenthesis and the remainder with the right fragment.

Comma If there exists a comma, the comma and all trailing white space splits the layout into the left and right parts. The comma and whitespace are removed.

Remember that we have two trees of fragments: one built from the original AST containing all layout fragments, and another one built from the refactored AST, which only contains essential fragments (our example does for the sake of simplicity not contain scopes). The algorithm then proceeds as follows.

1. After the transformation, the essential parts can be obtained.

◆ `LR` `right` `String` `left` `Int` ◆

2. We will now iterate over all refactored fragments in pairs, the first pair is

◆ `LR` ◆.

3. There are no sub-scopes in this example, so we will never recurse. We see that these two fragments are still in the same order as in the original tree, so we can just take their enclosed layout code and continue with the next pair, LR and right.

◆ \nclass· LR right

4. This time, there is no shortcut, because the order of the trees has changed. So we go to the original tree fragments and find for both fragments of the pair the respective neighbors and the layout fragments they enclose, these are:

LR (left Int ,· right .

5. The first layout (is split according to rule **Left Parenthesis**; The second layout is handled by the **Comma** rule.

LR (left Int right

If there would have been blanks or a comment to the right of the parenthesis, this would have gone to the right. In this case, there's no further layout left to assign. So the resulting layout is a single opening parenthesis.

6. The opening parenthesis can be added to the result, as well as right. Our result so far is

◆ \nclass· LR (right .

7. The next element that forms a pair with right is String, and because they are still in their original order relative to each other, we can add all their intermediate layout code as well (we would still need to correct the indentation though if the layout contained multiple lines).

◆ \nclass· LR (right :: String

8. The next essential fragment that needs to be processed together with String is left.

◆ \nclass· LR (right :: String left

9. Again, we retrieve the respective neighbors from the original tree:

String)\n ◆ LR (left

10. Splitting the layout code according to the rules **Right Parenthesis** and **Left Parenthesis** yields the following associations:

String)\n ◆ LR (left

11. We are interested in the layouts of `String` and `left`, but as we can see, there is no layout left. But because we are processing the elements of a comma separated argument list, the `String` fragment requires that we print a comma and space (the default formatting according to the Scala style guide [Spi09]). This results in the following new layout code:

```
◆ \nclass· LR ( right :· String ,· left
```

12. Just as with the first argument to the class, we now have an unchanged sequence of fragments which we can just take as they are.

```
◆ \nclass· LR ( right :· String ,· left :· Int
```

13. We now come to the last fragment and the end of the scope.

```
◆ \nclass· LR ( right :· String ,· left :· Int ◆
```

14. Just as in the previous steps, we lookup the original neighbors and split the layout according to the same rules: **Comma** and **Right Parenthesis**.

```
Int right String )\n ◆
```

15. Adding these final fragments yields the final result.

```
◆ \nclass· LR ( right :· String ,· left :· Int )\n ◆
```

This example has shown how the presented algorithm can be used to re-create source code after a refactoring, including all layout. Perhaps the example was too simple – for example, would it also work if we have add a comment somewhere? Let us find out and change some of the layout to include a comment:

```
LR ( //first parameter left .
```

These two essential fragments are handled in steps 5 and 10 through the rule **Left Parenthesis**. If we apply the same rule on our comment, the layout is split as follows:

```
LR ( //first parameter left .
```

This would not have affected step 6, but in step 11, the result would now be

```
◆ \nclass· LR ( right :· String ,· //first parameter left .
```

We have now seen that the algorithm works just as well if there are comments present. Of course, these comments need to be handled somewhere and assigned to a certain fragment, which might not always be what the author of a comment intended. But at least using this scheme no comment can ever get lost in translation.

Now that we have transformed our refactored tree back into source code, all that is left is to integrate this into the IDE.

3.5. IDE Integration

The interaction with an IDE consists of two steps: first the refactoring needs to be *prepared*, and then it can be *performed*. The preparation phase is used to determine whether the refactoring is feasible before actually doing it.

The Refactoring class contains the following abstract members that need to be provided by a concrete refactoring inheriting from it.

type PreparationResult

type RefactoringParameters

def prepare(f: AbstractFile, from: Int, to: Int):
Either[PreparationError, PreparationResult]

def perform(prepared: PreparationResult, params: RefactoringParameters):
Either[RefactoringError, ChangeSet]

The arguments to the prepare method are a file and a selection inside it. The refactoring implementation then analyzes the AST of the given file (a compiler instance has already been passed to the Refactoring constructor) and returns either an instance of PreparationError or PreparationResult. PreparationError simply contains a string that explains why the refactoring cannot be performed. The PreparationResult holds all the information the refactoring assembled, for example the enclosing method, so that the IDE can show this to the user. In Eclipse terms, the prepare method corresponds to the checkInitialConditions.

To actually perform the refactoring, the IDE passes the PreparationResult together with the RefactoringParameters to the perform method. The refactoring parameters are completely refactoring dependent, in Extract Method, they contain the name of the extracted method. The result of this operation is either another error message or a ChangeSet that holds the refactored source.

3.6. Conclusion

This chapter has introduced each phase of the refactoring process. The *analysis* step takes care that the following *transformation* step has all the information it needs, before it hands over a modified AST to the source *regeneration* phase.

The focus, and this is also where most of the time in this term project has been spent, was clearly on the source regeneration phase. None of the phases are by any means done, in fact, the only supported refactoring for now is Extract Method (which will be explained in more detail in the next chapter). The following is an overview of the work that still needs to be done in the different phases.

Analysis: The analysis phase needs to be extended with refactoring specific information – for example, only single ASTs are processed at the moment, for more global refactorings (e.g. rename or inline) multiple ASTs and their information need to be combined.

Transformation: The phase is largely refactoring specific, but there exists a lot of potential for providing helpers to the user – for example to straightforwardly create new trees, or modify existing kinds of trees. These helpers are not strictly needed and will likely be implemented along with a refactoring and then moved to the transformation package to be reused by new refactorings.

Regeneration: Is mostly refactoring independent; additional work needs to be put into making sure that the fragments are built correctly and that scopes are introduced where needed. Regarding the layout splitting, only a handful of rules are implemented, some more will likely be needed – for example to adjust the assignment of comments.

IDE: The IDE integration is still a fast-moving target and is subject to change once more refactorings and IDEs are going to be integrated with each other.

By implementing more refactorings and thus completing the three stages, the library as a whole will mature and eventually be usable by contributors that do not know the inner workings, which is the ultimate goal of the project.

The next chapter will introduce the Extract Method refactoring in Scala, showing possible extensions of the classical refactoring description by Fowler and the status of the current implementation.

4. Extract Method Implementation

In this chapter, we shall revisit the well known Extract Method refactoring and see how it can be realized in the context of Scala, as well as introduce the actual implementation of it.

Why did I choose to start with Extract Method? Martin Fowler wrote [Fow01] that

if you can do Extract Method, it probably means you can go on more refactorings [because it] requires some serious work. You have to analyze the method, find any temporary variables, then figure out what to do with them.

On the other hand, Extract Method is also convenient to start with because it operates only in a local scope, thereby avoiding operations on multiple files, which – depending on the environment – can also be challenging.

4.1. Extract Method Revisited

To quote Martin Fowler again [Fow99], the Extract Method refactoring should be applied whenever

you have a code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method.

Several variations of the refactoring are possible, depending on how local variables are used inside the extracted method. Note that for brevity's sake, whenever we are talking about variables – which is the common denominator with Java – in the remainder of this chapter, we also mean to include values and functions. When we talk about functions, this includes methods as well.

No local variables is the trivial case. All that needs to be done is to move the selected code into a new method and insert a call to it at the originating side.

Local variables are used inside the extracted method, but new ones are either not defined or used just in the extracted part. These inbound dependencies need to be passed into the newly created method.

Local variables are defined inside the extracted method and are used at the originating site; these need to be returned from the new method and assigned to local variables at the call site. In Java, this scheme works as long as just one variables needs to be returned. In C++, passing a reference is the solution. Scala supports

```
def parse(source: String): (Int, String) = {  
  // read an integer from the source, return it along with the rest of the source:  
  (intResult, restSource)  
}  
  
val (parsedInt, restSource) = parse("5$")
```

Figure 4.1.: Scala tuples in action: The function `parseInt` of type $String \rightarrow (Int, String)$ returns a tuple of `Int` and `String`. Line 3 demonstrates how a tuple is returned from such a function; the last line in the example shows how the returned tuple is deconstructed into the two variables `parsedInt` and `restSource`.

tuples and syntactic sugar to decompose them into independent variables (an example is shown in Figure 4.1). This allows us to perform the refactoring in Scala where it would not be possible with similar code in Java.

Besides the support for tuples that allows us to return multiple values from a method, Scala has more language features that allow variations of the classical Extract Method refactoring: currying and functions as first class values.

Extract Block

In contrast to many other languages that support some notion of currying, Scala curried functions are simply functions with multiple argument lists (it is also possible to turn a regular function into a curried one using `Function.curried`). This feature could be supported by an automated refactoring implementation, but has not been implemented yet.

Instead of generating multiple argument lists, the developer might want to extract some code to a method, but keeping the extracted code at the call site, passing a function to the extracted method that contains the code. This might be useful in preparation for a larger refactoring; Figure 4.2 on the following page shows an example.

Extract With Higher Order Functions

Much more potential for a useful variation of Extract Method lies in Scala's support for higher order functions in combination with its object-oriented features. Often when extracting some code, this code calls methods on other objects. In Scala, a method can also be used where a function with the same signature is expected. Therefore, instead of passing objects on which then methods are called, we pass the method directly! This makes the extracted method much more reusable – instead of depending on an object's interface, we now only depend on a function signature. This should increase

```
val msg = "Hello World"
```

```
println("=====")
println(msg)
println("=====")
```



```
def printWithBanner(msg: String)(body: String => Unit) = {
  println("=====")
  body(msg)
  println("=====")
}
```

```
val msg = "Hello World"
```

```
printWithBanner(msg) { msg =>
  println(msg)
}
```

Figure 4.2.: Consider the last three lines of the first listing; the code might have more such occurrences where a message is printed and enclosed with some strings. Using Scala's multiple argument lists, parts of the extracted code can remain at the call site. Passing the String argument is of course optional, because the passed function can also be a closure. The biggest challenge in implementing such a refactoring would likely be to find a good user interface.

```
val sb = new StringBuilder
val name = new BufferedReader(new InputStreamReader(System.in)).readLine
sb.append("Your name is: ")
sb.append(name)
```



```
def askName(out: String => Any) {
  val name = new BufferedReader(new InputStreamReader(System.in)).readLine
  out("Your name is: ")
  out(name)
}

val sb = new StringBuilder
askName(sb.append)
```

Figure 4.3.: A variation of the Extract Method refactoring that is possible in Scala using higher order functions. From the first listing, we want to extract all but the first line into a new method `askName`. Instead of requiring a reference to a `StringBuilder` object, the new method takes as its argument a function of type `String → Any` called `out`. Scala allows us to pass an object's method instead of a function, as long as the signatures match. This is the case for `StringBuilder`'s `append` method and can be seen on the last line. The extracted method can now be used with any other function or method that satisfies the signature, thus increasing its reusability.

the chances to reuse the extracted method in other places. An example of this variation is shown in Figure 4.3.

Method Placement

In Java, an extracted method is usually made a private member of the same class. Scala allows us more freedom in the placement – instead of making it a member of the class, the extracted method can be put directly into the originating method. This is obviously a trade off between reusability and keeping the namespace of the class clean. When the method is nested, one could do away with passing arguments; then a move refactoring would come handy once there is a chance for re-use and the method has to be moved to the enclosing class.

4.2. Implementation Details

Coming back to the actually implemented version of Extract Method, let us take a look at what features it supports. Maybe a word of caution is in order; the refactoring has not been extensively tested with real world code and is little more than a proof of concept for the underlying infrastructure.

We have already seen most there is of the refactoring's transformation in Section 3.3 on page 17 and the needed analysis in Section 3.2 on page 16. The Analysis, Transformation and Regeneration traits are all accessible via the Refactoring class (as introduced in Section 3.1 on page 14), which needs to be provided with an instance of interactive.Global – the compiler in an IDE setting.

Once the tree has been generated, the changed source code can be obtained with a call to the refactor method which has the signature $(Tree, Tree) \rightarrow String$. The first parameter denotes the original tree (alternatively, a view from AbstractFile to Tree exists as well), the second is the refactored one. Note that in the current implementation, the complete source code is returned, not just the individual changes. This will likely change in the future to return several changes, including a description of the change to be displayed to the user.

Now we shall finally take a look at a few examples of refactorings the current implementation performed.

4.3. Examples

Let us start with a simple example, extracting a method that has a parameter and returns a value. The blue highlight indicates the user's selection.

```
class Magic {  
  def calculateTwo() {  
    val one = 1  
    val two = one + 1  
    two  
  }  
}
```

Extracting the highlight statement and providing an appropriate name yields the following listing.

```
class Magic {  
  def calculateTwo() {  
    val one = 1  
    val two = increment(one)  
    two  
  }  
}
```

```

def increment(one: Int): Int = {
  val two = one + 1
  two
}

```

We can already see an improvement in the extracted method: the superfluous value `two` could be inlined and the method could return `one + 1` immediately. We then would not even need the block anymore and could extract directly to `def increment(one: Int): Int = one + 1`. We also see that the return type is explicitly specified, which is also unnecessary in most cases. Such improvements will be considered for a future version.

Increasing the level of fidelity, we force the refactoring to create a higher order function:

```

object Higher {
  def yieldsTrue(): Boolean = {
    val invert: Boolean => Boolean = ! _
    val a = false
    val b = invert(a)
    b
  }
}

```

After extracting the highlighted statement, it is turned into the following two methods.

```

object Higher {
  def yieldsTrue(): Boolean = {
    val invert: Boolean => Boolean = ! _
    val a = false
    val b = offshore(invert, a)
    b
  }
  def offshore(invert: (Boolean) => Boolean, a: Boolean): Boolean = {
    val b = invert(a)
    b
  }
}

```

Again, there is room for improvement, besides the already noted enhancements, there is a superfluous set of parenthesis around the signature of the formal `invert` parameter.

Finally, let us do a refactoring that is not possible with the Extract Method from any Java refactoring tool.

```

class MyMath {
  def calculate {
    val values = 1 to 10 toList
    val sum = values reduceLeft (_+_ )
    val product = values reduceLeft (_*_ )
    println("The sum from 1 to 10 is "+ sum +"; the product is "+ product)
  }
}

```

The problem is that the extracted method needs to return both the sum and product values. This is no problem in Scala, as the following listing shows.

```

class MyMath {
  def calculate {
    val values = 1 to 10 toList
    val (sum, product) = sumAndProduct(values)
    println("The sum from 1 to 10 is "+ sum +"; the product is "+ product)
  }
  def sumAndProduct(values: List[Int]): (Int, Int) = {
    val sum = values reduceLeft (_+_ )
    val product = values reduceLeft (_*_ )
    (sum, product)
  }
}

```

You might have noticed that in the last three examples, we always extracted one or more statements from within a method. The last example shows that we can also extract expressions; let us try the condition from an if expression, followed by a comment.

```

trait Check {
  def whatIsIt(check: Boolean) {
    if (check == false /*check if false*/)
      println("It's false")
    else
      println("It's true")
  }
}

```

Assuming the comment is meant to annotate the comparison operation (because it is inside the parenthesis), it gets extracted along with the highlighted text.

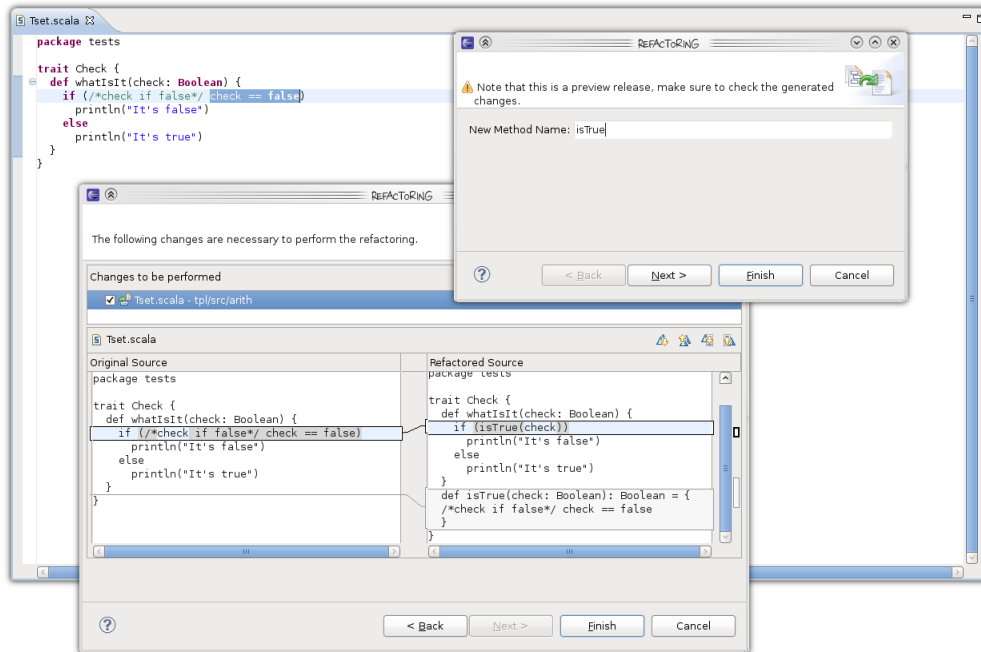


Figure 4.4.: An demonstration of Extract Method inside the Scala Eclipse plug-in. A default warning is enabled while the refactoring is still under development.

```

trait Check {
  def whatIsIt(check: Boolean) {
    if (isFalse(check))
      println("It's false")
    else
      println("It's true")
  }
  def isFalse(check: Boolean): Boolean = {
    check == false /*check if false*/
  }
}

```

This last example is also shown as screen shots in Figure 4.4. This chapter has demonstrated that the implemented refactoring is already able to perform various extractions, but there is room for improvement. Besides the already mentioned enhancements, the refactoring also needs to offer more choices to the user (e.g. where the method should be placed), including some of the afore mentioned variations of Extract Method. The next chapter will outline the plans for the future.

5. Outlook

In this chapter, I am going to sum up this report by recapitulating shortly on what has been achieved and in which direction the project could evolve.

5.1. Results

Reflecting on the problem outline in the introduction (Section 1.1 on page 1), the goals of the project were to create a library to refactor Scala code with an implementation of a refactoring, including its integration into Eclipse.

All these goals have been reached, at least to a certain degree. The refactoring library is by no means complete, rather it is a skeleton that has proven to be suitable for the first refactoring implementation, or in other words: the Rubicon has been crossed! All that is left to do now is to put some flesh on the skeleton, what this implies for the library was discussed in the conclusion of Chapter 3 on page 29.

5.2. Next Steps

One way way to evolve the project is to tackle another refactoring, for example a Rename refactoring. In contrast to Extract Method, Rename has to analyze a complete project, not just a single compilation unit. This change from a local to a global scope would lead to some necessary changes in the refactoring interfaces, while at the same time not challenging the source regeneration phase. A combination would be a Move refactoring, which is concerned with more than one compilation unit and also modifies all kind of existing code.

Adding Extract Local Variable, a refactoring quite similar to Extract Method, these four refactorings would – according to Murphy-Hill [MHPB09] – already cover over ninety percent of all refactoring usages.

At the same time, the existing Extract Method could be enhanced, for example with an intelligent enhancement of the user's selection, as was done in the Extract Method refactoring for Ruby [CFS07]. The afore mentioned study also discovered that ninety percent "of configuration defaults of refactoring tools remain unchanged when programmers use the tools". Based on these results, existing interfaces to refactorings should probably be simplified, making a refactoring easier and more light-weight to use (a trend in the Eclipse Java Development Tools (JDT) is to avoid popping-up

dialogs for refactorings, embedding the refactoring action into the editor instead of making it a distinct activity).

As soon as refactorings that affect a global scope are available and because Scala code is often combined with Java code, the issue of cross-language refactorings comes up, as described by my colleague Mike Klenk [KKKS08a]. Naturally, such work would then be largely IDE specific – for example in the Scala plug-in for Eclipse, where most of the integration with JDT is done using aspect oriented programming, hooking into a JDT refactoring should not be a problem.

A different approach to continue the project would be to study refactorings that are common in functional programming (for example from Haskell [Has10]) and trying to integrate them with Scala; or to work on refactorings that make code more idiomatic Scala (e.g. something like Replace Conditional With Pattern Matching).

Whichever course will be taken for the follow-up masters thesis remains yet to be decided.

5.3. Acknowledgments

I would like to thank my advisor Peter Sommerlad for his continuous support during this thesis and for giving me a free hand to realize my desired project. Many thanks also to Miles Sabin who helped me getting up to speed with the Scala Eclipse plug-in and patiently answered my questions about the Scala compiler, just as the other members of the Scala mailing lists. The Scala listings style definition I gratefully received from Miguel Garcia, who also showed a great interest in my work and maintains a fantastic collection of information about the Scala internals in his Scala Compiler Corner [Mig10] – I wish the information had already been there when I started with the project.

A. Project Environment

This project was performed as a one-semester term project at the University of Applied Sciences Rapperswil, Switzerland. The project's accompanying server that hosts the Trac issue tracker, the repository, and the Hudson build server can all be found at <http://scala.ifs.hsr.ch>.

A.1. Project Plan

The project plan shown in Figure A.1 on the next page was created at the beginning of the project and then continuously updated to reflect the actual spent time per component. Initially, the project was planned to take 14 weeks, but due to several interruptions, another week had to be added at the end.

A.2. Time Report

The specified duration of such a term project is 360 hours of work (12 ECTS). My time reporting in Figure A.2 on the following page shows that I have worked for over 400 hours, but this also includes the time spent on two conference papers.

A.3. Tools

The following is an accumulation of tools I have used that helped me produce this report as well as writing over 4000 lines of code (2500 of it are tests).

L^AT_EX 2_ε generated this document, using the KDE's *Kile* editor in combination with the *Okular* PDF viewer. *Graphviz* was used to create (most of them by hand, using *Vim*) all the figures in this report (except for the time reporting, that was done using *worked*¹). *Inkscape* helped me post-process some of the generated figures, namely the fragments merging example. *Eclipse* was the obvious choice for development, in combination with a custom build of the *Scala IDE*'s latest trunk. *Git* connected me to EPFL's Subversion repository and allowed me to push all the code to my *Hudson* build server.

¹<http://worked.rubyforge.org>

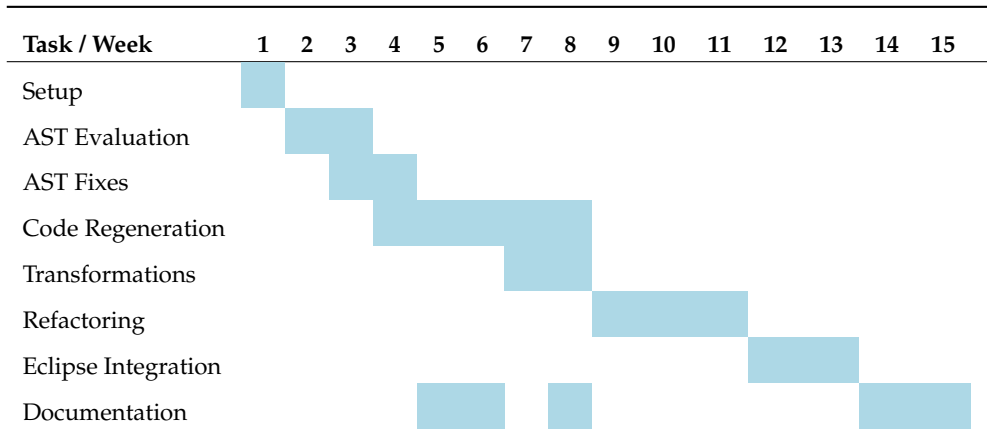


Figure A.1.: The actual project plan, showing how much time was spent on which component.

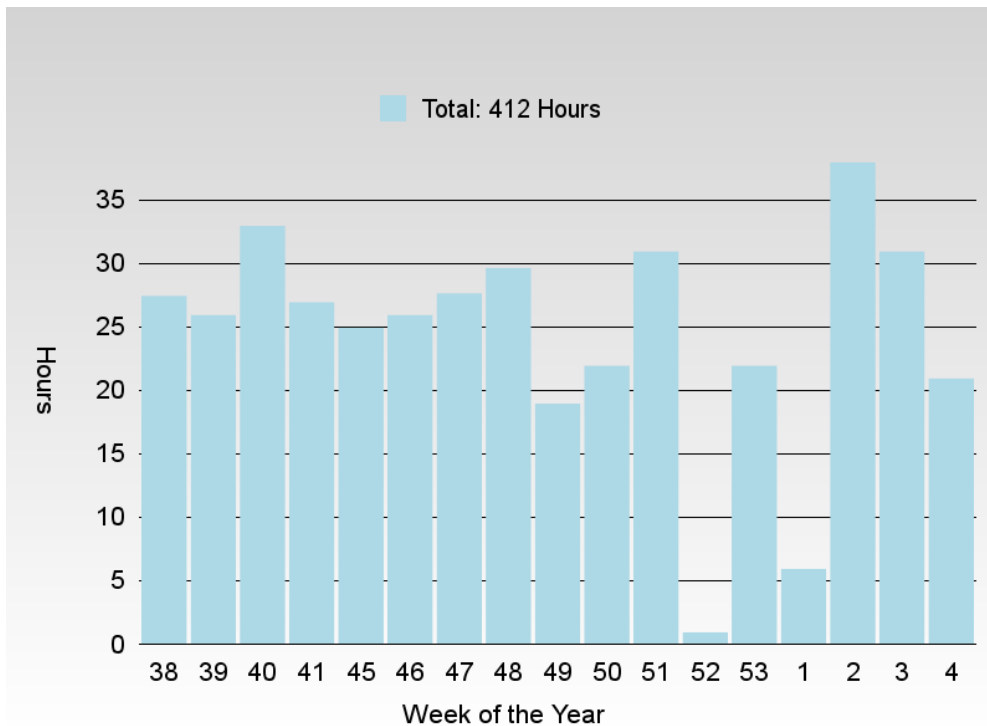


Figure A.2.: Hours of work per week of the year. A total amount of 412 hours averages to 27.5 hours per week.

Bibliography

- [CFS07] Thomas Corbat, Lukas Felber, and Mirko Stocker. Refactoring support for the eclipse ruby development tools. Technical report, Institute for Software, HSR – University of Applied Sciences Rapperswil, 2007.
- [Dic09] Vassil Dichev. Embedded scala interpreter. <http://www.speaking-my-language.blogspot.com/2009/11/embedded-scala-interpreter.html>, Archived at <http://www.webcitation.org/5meZZsfbe>, 2009.
- [Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [Fow01] Martin Fowler. Crossing refactoring’s rubicon. <http://martinfowler.com/articles/refactoringRubicon.html>, Archived at <http://www.webcitation.org/5mjVmnOci>, 2001.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.
- [GZS07] Emanuel Graf, Guido Zraggen, and Peter Sommerlad. Refactoring support for the c++ development tooling. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 781–782, New York, NY, USA, 2007. ACM.
- [Has10] Refactorings for haskell. <http://www.cs.kent.ac.uk/projects/refactor-fp/catalogue/>, Archived at <http://www.webcitation.org/5mvP6ayDc>, January 2010.
- [KKKS08a] Martin Kempf, Reto Kleeb, Michael Klenk, and Peter Sommerlad. Cross language refactoring for eclipse plug-ins. In *WRT '08: Proceedings of the 2nd Workshop on Refactoring Tools*, pages 1–4, New York, NY, USA, 2008. ACM.
- [KKKS08b] Michael Klenk, Reto Kleeb, Martin Kempf, and Peter Sommerlad. Refactoring support for the groovy-eclipse plug-in. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 727–728, New York, NY, USA, 2008. ACM.

- [MHPB09] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.
- [Mig10] Miguel Garcia and the Scala Community. Scala compiler corner. <http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/>, Archived at <http://www.webcitation.org/5mg1YaesY>, 2010.
- [MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23:76–83, 2006.
- [Net09] Scala Plugins for NetBeans. http://wiki.netbeans.org/Scala#Scala_Plugins_for_NetBeans, Archived at <http://www.webcitation.org/5mstestVuy>, 2009.
- [Ode09] Martin Odersky. Scala compiler internals. <http://www.scala-lang.org/node/598>, Archived at <http://www.webcitation.org/5meXOPQBx>, 2008-2009.
- [OO07] Martin Odersky and Others. Scalac man page. <http://www.scala-lang.org/docu/files/tools/scalac.html>, Archived at <http://www.webcitation.org/5meZXpNPF>, 2007.
- [OO09] Martin Odersky and Others. The Scala Language Specification. 2009.
- [OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 2008.
- [Par09] Terence Parr. *Language Implementation Patterns*. Pragmatic Bookshelf, 2009.
- [RMO09] Tiark Ropmf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 317–328, New York, NY, USA, 2009. ACM.
- [Sab09a] Miles Sabin. Patch to retain import nodes. <http://lampsvn.epfl.ch/trac/scala/changeset/18962>, 2009.
- [Sab09b] Miles Sabin. Scala IDE for Eclipse. <http://www.scala-lang.org/node/94>, Archived at <http://www.webcitation.org/5mser1sle>, 2009.
- [Spi09] Daniel Spiewak. Scala style guide. <http://www.codecommit.com/scala-style-guide.pdf>, Archived at <http://www.webcitation.org/5mgH6f2m7>, 2009.
- [SZCF08] Peter Sommerlad, Guido Zraggen, Thomas Corbat, and Lukas Felber. Retaining comments when refactoring code. In *OOPSLA Companion '08*:

Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pages 653–662, New York, NY, USA, 2008. ACM.

- [ZP09] Sergey Zhukov and Alexander Podkhalyuzin. Scala Plugin for IntelliJ IDEA. <http://www.jetbrains.net/confluence/display/SCA>, Archived at <http://www.webcitation.org/5msesJVeV>, 2009.