Unobtrusive Refactoring Tools for Code Extraction in Scala

Author
Lukas Wegmann

Supervisor
Prof. Peter Sommerlad

Technical Adviser
Mirko Stocker

University of Applied Science Rapperswil

Term Project, January 27, 2014

Abstract

While the support for automated refactoring for the Scala programming language increased steadily over the last few years, it is still not as elaborated as the tools available for other statically typed languages like Java or C#. The few refactoring tools provided by major Scala IDEs are heavily influenced by their Java equivalents and are not particularly tailored to the lightweight and expressive nature of Scala.

This project provides a set of new refactoring tools for code extraction that offer lightweight invocation methods and require minimal user interactions while still remaining flexible. The source transformation logic of the new tools has been implemented as a part of the Scala Refactoring library and is also integrated in the Scala IDE for Eclipse.

The new tools support the refactoring techniques *Extract Method*, *Extract Value*, *Extract Parameter* and a new refactoring called *Extract Extractor* that allows to create abstractions of patterns via Scala's extractor syntax.

Management Summary

This report describes a new set of tools integrated into Scala IDE for Eclipse for automated extraction refactorings and proposes a lightweight but powerful concept for further refactoring tools.

Status Quo

Refactoring is the process of improving the internal design of a program without changing its external behavior. It is a technique that is not restricted to a particular programming language or a development environment and can be performed manually. Because applying refactorings manually is error prone and requires sometimes global, labor-intensive changes to the program code, many IDEs offer automated tools to support developers performing some common refactoring techniques.

For Scala, a statically typed but lightweight programming language, there are currently several IDEs that offer a more or less elaborated suite of automated refactoring tools. One of these is the Eclipse based Scala IDE which uses the Scala Refactoring library developed by Mirko Stocker during his master's thesis. Scala Refactoring is an IDE independent library providing implementations of common refactoring tools and additional utilities for code analysis, generation and transformation.

Conceptually, the refactoring tools offered by Scala IDE are quite similar to the according tools in Eclipse JDT, a development environment for the Java programming language. These tools mostly use a common work flow:

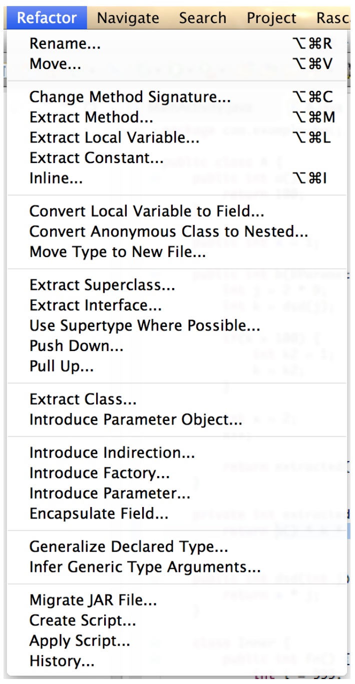
1. Select a code snippet to refactor
2. Invoke the tool that can apply the desired refactoring
    - The tool checks if the selected code fulfills the preconditions and offers some configuration options
3. Configure the refactoring according to your needs
    - The tool validates the configurations and calculates the changes to the code
4. Review the changes in a preview window (optional for some refactorings)
    - If the user confirms, the changes are applied automatically to the code in the editor window

Additionally to these refactoring tools there are shortcuts for some refactorings that offer a simpler invocation method without any configuration options. In this case, the refactoring result is applied directly in the editor window. But this simplified invocation methods tend to trade simplicity with flexibility and power.

Goals

The initial goal of this project was to implement additional refactoring tools for the Scala Refactoring library and provide integrations in Scala IDE. During the analysis of the objectives it emerged that there is a class of refactoring tools that perform basically the same kind of transformation on different kinds of code and can be grouped under the term *Extraction Refactorings*. These refactorings can be applied when a complicated code snippet should be abstracted such that the code becomes more readable and self-explanatory. Depending on the properties of the concerned code the new abstraction is for example a method or a value definition but the initial intention remains the same. This observation led us to the idea to create a new refactoring tool that combines and unifies these extraction refactorings. A user should no longer have to remember which of the available extraction tools suites his needs but has only to tell the IDE that he wants to extract the selected code snippet.

Figure 0.1.: The Eclipse JDK refactoring menu. Probably only a few developers know what kind of transformation is exactly performed by each of these refactoring tools.



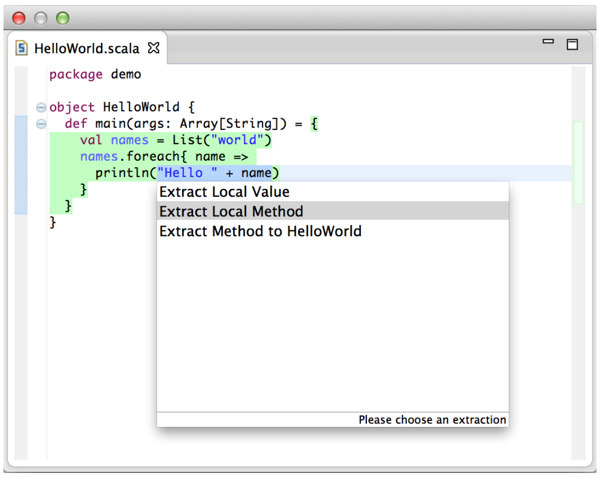
Result

We have designed and implemented a set of refactoring tools for code extraction that use a work flow closer to a conversation between the developer and the IDE:

1. Select a code snippet to refactor
2. Invoke the refactoring tool that matches your intention
   - The tool proposes several concrete refactorings that are applicable on the selected code
3. Select one of the concrete refactoring
   - The tool applies the changes automatically to the code in the editor window
4. Review the applied changes in the editor window
   - If the code does not matches your expectations undo the refactoring
   - If the code requires further refinement, modify it accordingly or use a supplementary refactoring tool to improve the result

The refactoring starts with the intention of the user like "I want to extract this piece of code" and the tool proposes concrete applicable transformations. There are no more modal dialogs that require you to make many configuration decisions to tell you in the end that this particular refactoring can not be applied automatically. Furthermore, the refactoring tool performs only the error prone and labor intensive parts of the refactoring without distracting you by asking for minor details that can easily be applied manually to the transformed code (e.g. the visibility of an extracted method).

Figure 0.2.: The new extraction assistant with three concrete extractions that can be applied to the selected code.



```
package demo

object HelloWorld {
  def main(args: Array[String]) = {
    val names = List("world")
    names.foreach{ name =>
      println("Hello " + name)
    }
  }
}
```

The following refactoring techniques are currently supported by the new extraction tools:

Extract Value
: Introduces a new value for a sequence of expressions. Similar - but more limited - refactorings in other IDEs are *Extract Constant* and *Extract Local Variable.*

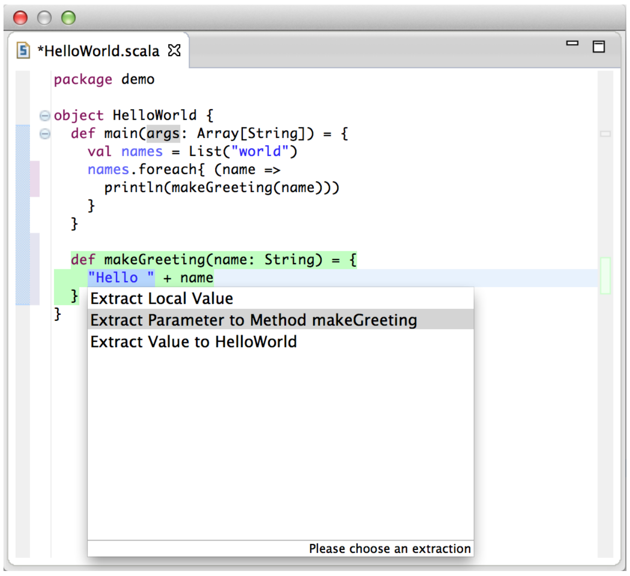
Extract Method
: Creates a new method definition based on the selected code.

Extract Parameter
: Adds a new parameter to an enclosing method whose default value becomes the original expression.

Extract Extractor
: Creates a new extractor object based on a selected pattern in a `case` statement.

While the *Extract Value* and *Extract Method* tools were already available in Scala Refactoring the new implementations are more flexible because they also allow to choose the target scope where the new value or method should be inserted. Furthermore, the new versions of these tools are more consistent in how they handle certain edge cases.

Figure 0.3.: After the extraction illustrated in Figure 0.2. the extraction tool is invoked once again on a string literal for further refinements. In this example the assistant proposes also the extraction into a new parameter.



```
package demo

object HelloWorld {
  def main(args: Array[String]) = {
    val names = List("world")
    names.foreach{ (name =>
      println(makeGreeting(name)))
    }
  }

  def makeGreeting(name: String) = {
    "Hello " + name
  }
}
```

Extract Local Value
Extract Parameter to Method makeGreeting
Extract Value to HelloWorld
Please choose an extraction

Contents

1. Introduction

The inital objective of this term project was to add further refactoring tools to the Scala Refactoring library [Sto10] and to integrate them in Scala IDE for Eclipse [Sca]. The implementation of the following refactorings was targeted:

Extract Closure
: Extracts an expression and creates a new closure method.

Extract Superclass
: Extracts methods and attributes from a class or object and inserts these in a new superclass or supertrait.

Pull Up and Push Down Member
: Moves methods and attributes of classes up or down in the class hierarchy.

Extract Extractor
: Creates a new extractor based on a selected pattern in a case statement and replaces the pattern by a call to the extractor.

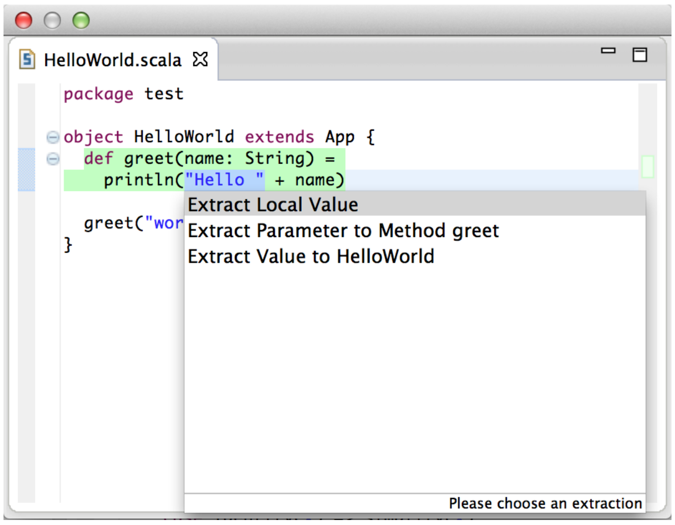
Toggle Inferred Types
: Shows or hides optional inferred type annotations.

During the analysis of the current state of the Scala Refactoring library and the *Extract Closure* refactoring it emerged that this refactoring shares many aspects with the already implemented *Extract Method* and *Extract Local*. Therefore, a self-contained implementation of *Extract Closure* was no longer a reasonable option because it would have led to redundant and poorly maintainable code.

Further studies of several refactorings as those described in [FB99] and [Ker04] revealed a category of refactorings that share a common work flow and can be termed *Extraction Refactorings*. All extraction refactorings introduce a new abstraction for a specific code fragment and replace the fragment by a call to this abstraction.

Based on these findings we formed the concept of a refactoring tool that proposes extractions depending on the selected code in the source code editor. In contrast to current implementations of extraction refactorings, this tool provides a unified and minimal workflow. By clicking the according menu item or invoking the hot key command the user opens a code extraction assistant menu as shown in Figure 1.1. This menu proposes applicable extractions and highlights the scope in which the selected extraction takes place. After the user selects an extraction it is applied to the source code. Finally, the user can rename the newly introduced abstraction right in the editor window.

Figure 1.1.: Code extraction assistant



```
package test

object HelloWorld extends App {
  def greet(name: String) =
    println("Hello " + name)

  greet("wor
}
```

HelloWorld.scala
Extract Local Value
Extract Parameter to Method greet
Extract Value to HelloWorld
Please choose an extraction

In chapter 2 we first describe the not yet documented refactorings techniques *Extract Closure* and *Extract Extractor* and define the term *Extraction Refactoring* as it is used in this report. The subsequent chapter 3 covers the core concepts that are used to implement the new refactoring tools based on recent studies on how automated refactoring tools are actually utilized. In chapter 4 we analyze the individual extraction tools, what kind of transformations they perform and what edge cases must be considered. chapter 5 explains the architectural concepts that we use to implement the individual but composable refactoring tools as it is necessary to offer a uniform invocation method. The supportive components for code analysis and transformation that were required for the implementation are described in chapter 6 and finally we give an outlook on how the concepts used for the extraction tools can be applied to other refactorings in chapter 8.

2. New Refactoring Techniques

The initial objective of this project included, amongst others, the implementation of two refactorings called *Extract Closure* and *Extract Extractor*. Because there exists no specification for both refactorings so far, they are described in a more detailed level in this chapter. Additionally, we describe the category of extraction refactorings and how the term is used in this report.

2.1. Extract Closure

Because people referring to the term "Closure" do not always have the same concept in mind, we first define how this term is used in this report. A closure is a function that captures references to variables from an enclosing local scope in its execution environment. The following example demonstrates the concept:

```scala
def mkCounter() = {
  var c = 0
  def inc(n: Int) = {
    c += n
    c
  }
  inc _
}

val inc1 = mkCounter()
val inc2 = mkCounter()

inc1(7) // returns 7
inc2(5) // returns 5 not 12
```

As we see, the variable `c` in the local scope of `mkCounter` is bound in both functions `inc1` and `inc2`. But because `c` is enclosed in separate execution environments calls to `inc1` and `inc2` do not increase the same state. Scala also has a shorter notation for anonymous functions such that the `mkCounter` method can also be defined as following:

```scala
def mkCounter() = {
  var c = 0
  (n: Int) => {
    c += n
    c
  }
}
```

While in the above examples the closure function is returned by the enclosing method, it is also possible to define closures as local functions or methods that are only used in the enclosing method and do not live beyond its context.

The intention to perform an *Extract Closure* refactoring is basically the same as for *Extract Method* or *Extract Local* (also known as *Introduce Explaining Variable*). According to [FB99] the former is applicable if "you have a code fragment that can be grouped together" and the later if "you have a complicated expression". Hence, both address long statements that decrease code readability. In languages that support closure objects *Extract Closure* is an additional solution to such issues.

The situation in which one should perform an *Extract Closure* refactoring could be described as **"you have a complicated expression that is used with varying parameters at different places"**. And the proposed solution is to **put the expression in a closure function with a meaningful name**. While this seems similar to *Extract Method*, the difference is that we could define closure functions as values in a local scope. Furthermore, closure functions could access other local variables of the enclosing scopes. Altogether, closure functions are powerful abstractions that could be defined in a very limited scope and are therefore accessible exactly where they are used.

2.1.1. Motivating Example

The following listing demonstrates when *Extract Closure* is applicable:

```
def getOsInfo(os: String) = {
  if(os.toLowerCase.indexOf("linux") != -1)
    "Penguins"
  else if(os.toLowerCase.indexOf("android") != -1)
    "Tiny Little Robots"
}
```

Because this method contains some duplications and the intention behind `os.toLowerCase...` is not very clear, applying *Extract Closure* helps to improve the code:

```
def getOsInfo(os: String) = {
  def osIs(token: String) =
    os.toLowerCase.indexOf(token) != -1

  if(osIs("linux"))
    "Penguins"
  else if(osIs("android"))
    "Tiny Little Robots"
}
```

If we perform an *Extract Method* refactoring on the example code we would achieve a similar result, except that we have to introduce a new method in the enclosing class.

In addition to the unnecessary wide visibility of this functionality that is probably just used in exactly this method, we would also have to pass the `os` variable as an additional argument to the new method. While this may be not a big issue in the example it could be cumbersome to pass a long list of local variables to the extracted method.

The other possibility according to Fowler would be to “introduce an explaining variable”. But in this example we would have to introduce two new variables like `osIsLinux` and `osIsAndroid`. Hence, in this case it is not a suitable refactoring especially when additional checks for other operation systems will be added later.

2.1.2. Mechanics

To perform the *Extract Closure* refactoring manually the following steps are necessary:

1. Create a new function in a scope that is visible from the original expression and name it after the intention of the code
2. Copy the extracted code to the new function
3. Determine what dependencies should become parameters to the function and add them to the function signature
4. Replace every usage of the dependencies in the new function by the according parameter names
5. Replace the extracted code by a call to the new function

Because the syntax for class member methods and local methods is effectively identical in Scala, one could also describe *Extract Closure* as *Extract Method* whereat the new method is created in a local scope. This is why we do not continue to distinguish between *Extract Closure* and *Extract Method* in this report and refer to both when writing *Extract Method*.

2.2. Extract Extractor

Pattern matching is a language mechanism to deconstruct data objects and extract specific values [EOW07]. In Scala, pattern matching is implemented by `match` expressions, which are sometimes described as “very powerful `switch` statements”. Every `match` expression consists of at least one `case` statement containing a pattern and a right-hand-side which is evaluated if the pattern matches.

Typically, pattern matching is used over instances of case classes. A common use case and often seen example of case classes and pattern matching is transforming tree structures respresenting arithmetic expressions as shown in the following listing:

```
trait Expr
case class Mult(l: Expr, r: Expr) extends Expr
case class Num(i: Int) extends Expr
```

```scala
def eval(e: Expr): Int = e match {
  case Mult(x, y) => eval(x) * eval(y)
  case Num(i) => i
}

eval(Mult(Num(7), Num(6))) // returns 42
```

Additionally, patterns can be defined without case classes in extractors. Extractors are yielded by methods called `unapply` that return an `Option` type or a `Boolean`. An extractor called `Adult` that matches only instances of `Person` older than 18 could be implemented as following:

Listing 2.1: Extractors in scala

```scala
object Adult{
  def unapply(p: Person): Option[Person] =
    if(p.age >= 18) Some(p) else None
}

val greeting = p match {
  case Adult(a) => "Dear Sir or Madam"
  case p => s"Hi ${p.name}"
}
```

In addition to extractors that extract a fixed number of values from an object, it is also possible to create extractors that yield a variable number of values by using `unapplySeq`. A complete overview of pattern matching and extractors in Scala is given in [Rü09].

Beside the ability to create patterns for which no case classes exists, extractors are also usefull to abstract complicated patterns and reuse them in arbitrary match expressions. This leads us to the *Extract Extractor* refactoring that is applicable if **you have a complicated pattern used in several places** and proposes to **move the pattern into a new extractor object with a meaningful name**.

2.2.1. Motivating Example

The transformation of arithmetic expressions is a problem often used to explain pattern matching. In the following example we continue with this tradition and show how *Extract Extractor* can be applied to abstract over patterns:

```scala
trait Expr
case class Mult(l: Expr, r: Expr) extends Expr
case class Num(i: Int) extends Expr

def simplify(e: Expr): Expr = e match {
  case Mult(Num(1), x) => simplify(x)
  case Mult(x, Num(1)) => simplify(x)
  case e => e
}
```

The `simplify` method in the above example takes an arithmetic expression and recursively simplifies it by using predefined rules. E.g. `1 * (x + 2)` becomes `x + 2`. Because we cannot use the same binding `x` in multiple alternatives of a pattern, we have to write for each rule a separate `case` statement. Hence, we have to repeat the right-hand-side of the `case` statement also for identical cases. Furthermore, the pattern logic for matching expressions that evaluate to the identity of one subexpression can not be reused. Both issues can be addressed by introducing an `Identity` extractor:

```
// trait Expr ...

object Identity {
  def unapply(e: Expr): Option[Expr] = e match {
    case Mult(Num(1), x) => Some(x)
    case Mult(x, Num(1)) => Some(x)
    case _ => None
  }
}

def simplify(e: Expr): Expr = e match {
  case Identity(x) => simplify(x)
  case e => e
}
```

2.2.2. Mechanics

To perform the *Extract Extractor* refactoring manually the following steps are required:

1. Create a new object named after the properties that are matched by the extracted pattern
2. Add an `unapply` method which takes an instance of the type to match
3. The `unapply` method returns:
   - a `Boolean` if there are no bindings in the extracted pattern.
   - an `Option` if there is only one binding.
   - an `Option` whose type parameter is a n-tuple if there are n bindings and $n > 1$. E.g. the according `unapply` of the pattern `Mult(Num(x), e)` returns an `Option[(Int, Expr)]`
4. Implement the semantics of the extracted pattern in the unapply method body.
   - Return an instance of `Some` or `true` if the pattern matches.
   - Return `None` or `false` otherwise.
5. Replace the original pattern by a call to the extractor.

2.3. Extraction Refactorings

The description of refactorings in [FB99] and in this report reveals a common pattern in some refactorings that could be grouped under the term “Extraction Refactorings”. Each extraction refactoring has all of the following properties:

1. Part of the input to the refactoring is a code snippet that should be extracted.
2. It creates a new abstraction that somehow contains the extracted code, for instance a new method or a new variable.
3. The new abstraction is created in a scope that is visible from the original code snippet, for instance in the surrounding class or right before the snippet in a method.
4. It replaces the extracted code by a reference to the new abstraction.

Testing the refactorings described in [FB99] for those properties, reveals at least the following list of extraction refactorings:

- Extract Method
  - Input is a consecutive code fragment
  - Create a new method in the enclosing class
  - Replace code fragment by a method call
- Introduce Explaining Variable (also known as *Extract Local*)
  - Input is a complicated expression
  - Create a new local variable before the original expression
  - Replace code fragment by variable access
- Replace Method with Method Object
  - Input is a long method
  - Create a new class that does the same calculation
  - Replace content of the method by an instantiation of a method object and the call to its start method
- Introduce Foreign Method
  - Input is an expression that extends the behavior of a class that cannot be modified
  - Create a new method with an instance of the class as its first argument
  - Replace the expression by a call to the method

- Introduce Local Extension
  - Input is similar as in *Introduce Foreign Method*
  - Create a new wrapper class with an instance of the not modifiable class as its single constructor argument
  - Replace the expression by an instantiation of the wrapper object and a call to the according method
- Replace Magic Number with Symbolic Constant
  - Input is a literal used in an expression
  - Create a constant with a meaningful name
  - Replace literal by access to the constant
- Decompose Conditional (identical to *Extract Method*)

Because there are some refactorings described by Fowler, that also follow the extraction pattern in some special cases, the list stated above is not complete. Additionally there are more known refactorings as some described in [Ker04] that could be added to the list. This includes *Extract Parameter* where the input is an assignment to a local field or variable that is moved to the method's parameter list. After that, the right hand side of the assignment could be replaced by the access to the new parameter.

Finally we could also add the two refactorings *Extract Closure* and *Extract Extractor* described in this chapter to the list of extraction refactorings. This gives us at least ten refactorings that follows a common pattern.

Note that while many of the listed refactorings are named like "Extract ...", "Replace ..." or "Introduce ...", not every refactoring with such a name falls exactly into this pattern. For example the *Extract Class* refactoring does not satisfy the fourth property of the enumeration above. Instead of replacing the extracted class members by something, it proposes to replace every use of the extracted members by the call to the new delegation class. Additionally there are refactorings like *Extract Superclass* or *Pull Up Method* that just move the affected code.

3. More Permissive and Universal Refactoring Tools

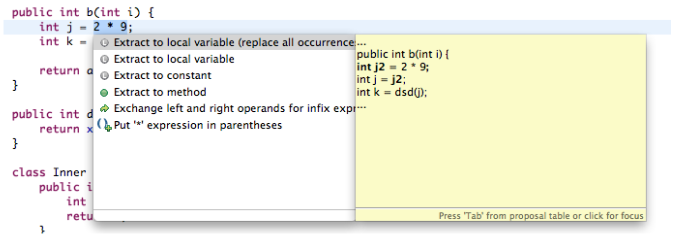
This chapter describes the core concepts behind the extraction tools implemented during this project. These fundamental concepts form the base of any further design decision.

3.1. Unobtrusive and Simple User Interfaces

How We Refactor and How We Know It is an exhaustive study about how automated refactoring tools are used in Eclipse [MHPB09]. One of the observations made in this study is that “programmers often don’t configure refactoring tools”. The study examined 11 configuration options of the 5 most commonly used refactoring tools and observed a change frequency of these configurations between 0% and 24%, whereas only three configurations had change frequencies higher than 10%. Hence, many default values of configurations are in fact almost never changed by the user.

Another observation was that “refactoring tools are underused” and many refactorings that are supported by automated tools are often performed manually. Although the study gives no explanation why these tools are not used more frequently. [VCN+12] examines this question by both analyzing usage statistics of Eclipse and interviewing participants. This study also compares the different methods Eclipse offers to invoke a refactoring. Depending on the refactoring Eclipse has basically two different methods to invoke refactorings. Most refactorings are accessible in the “Refactoring” menu and open a modal dialog which offers some configuration options. Additionally, some refactorings can be invoked using the *Quick Assist* menu (Ctrl + 1). The *Quick Assist* menu proposes refactorings based on the currently selected code. In contrast to the refactorings invoked over the “Refactoring” menu *Quick Assist* refactorings do not offer configurations in a modal dialog and perform the refactoring instantaneous after the user selected it.

Figure 3.1.: Quick Assist menu in Eclipse JDT


Extract to local variable (replace all occurrence…
Extract to local variable
Extract to constant
Extract to method
Exchange left and right operands for infix expr…
Put '*' expression in parentheses
Press 'Tab' from proposal table or click for focus

```
public int b(int i) {
    int j = 2 * 9;
    int k =

    return a
}

public int d
    return x
}

class Inner
    public i
        int
        retu
    }
```

```
public int b(int i) {
int j2 = 2 * 9;
int j = j2;
int k = dsd(j);
```

[VCN$^+$12] observed that “programmers prefer lightweight methods of invoking refactorings like Quick Assist” over the more obtrusive counterparts in the refactoring menu. Furthermore, it indicates that programmer tend to quickly apply refactorings and fix the outcome afterwards in order to fit their needs. This supports the finding of the former study that configurations in refactorings are seldom changed.

Given these findings it seems reasonable to focus the development of further refactoring tools on lightweight invocation methods with a minimum of configuration options. In general, the idea is to reduce the cognitive overhead of such a tool and to try not to interrupt the user work flow. This means that we should prefer inline editing and components like the *Quick Assist* menu over modal dialogs. Such a less obtrusive style of IDE tools can also be found in other IDEs like IntelliJ IDEA [Jet] or CodeRush [Dev], a plugin for VisualStudio.

3.2. Do One Thing and Do It Well

What is commonly called the Unix philosophy [MPT78] can also be applied to refactoring tools: “do one thing and do it well”. In [FB99] Fowler states “Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug.” Furthermore, he notes that “[. . .] the cumulative effects of these small changes can radically improve the design.” The reason behind this policy of performing small iterative improvements to the code rather than heavy lifts at once is the assumptions that programmers make mistakes while refactoring. Going from one syntactically correct version of the source file to another in minimal steps reduces the risk of introducing erroneous code and makes it easier to reproduce what went wrong in case of an error.

This policy should be continued in the new extraction refactoring tools. Keeping an automated refactoring as simple as possible comes with several advantages. First, it is easier for the user to comprehend what changes have been applied by a refactoring tool. Secondly, simple refactorings are easier to implement and less error prone. Third, small refactorings are easily composable by the user to perform big design changes. And finally, simpler refactorings require less user interactions and are therefore a perfect fit to our policy of unobtrusive user interfaces.

4. The Extraction Tools

As described in chapter 3, we want to create refactoring tools that perform just small transformations on the code and are easy to invoke. Additionally, we have identified a list of refactorings following a common pattern called *Extraction Refactorings*. Because those refactorings behave very similar and mostly just differ in the kind of the extracted code and the scope where the new abstraction is going to be inserted, it is possible to unify the according refactoring tools in a common invocation method called *Extract Code*. This chapter first describes the distinct code extraction tools tailored to the Scala language and finally how they could be composed in *Extract Code*.

Because this and the following chapters use many listings to demonstrate the behavior of the extraction tools, we use the convention that code enclosed in `/*(*/` and `/*)*/` comments mark the selection initially made by the user before invoking the refactoring tool.

4.1. Extraction Targets

Before performing an extraction a programmer has to know where he plans to insert the new abstraction he is going to create. Usually there are many possible extraction targets in different scopes for a certain extraction. A new abstraction can be inserted as a member of the enclosing class, right before the original code, at the beginning of the enclosing method body, and so on.

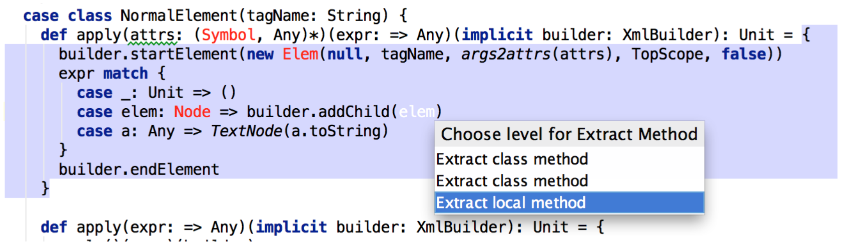
There are different strategies implemented in current IDEs that allow to select the desired extraction target in a more or less fine grained way:

- In Eclipse JDT the extraction target is bound to the invoked extraction tool. *Extract Local* inserts the new variable definition right before the extracted expression. *Extract Constant* on the other hand creates a new static member at the beginning of the enclosing class. If *Extract Constant* is invoked inside of a nested class it always adds the constant definition to the innermost class. A slightly different behavior is implemented in *Extract Method* as it inserts the new method right after the method that contains the extracted code in the enclosing class. If a method is extracted from a nested class the user can choose which of the enclosing classes should be used as the extraction target.

- The current version of the Scala IDE for Eclipse offers with *Extract Method* and *Extract Local* two extraction refactorings with a similar implementation as in Eclipse JDT but without the possibility to choose from nested classes in *Extract Method*.

- IntelliJ IDEA for Java acts similar to Eclipse JDT but is more consistent as it also allows to choose nested classes in *Extract Constant*.

Figure 4.1.: Extraction Target Selection in IntelliJ IDEA for Scala



```
case class NormalElement(tagName: String) {
  def apply(attrs: (Symbol, Any)*)(expr: => Any)(implicit builder: XmlBuilder): Unit = {
    builder.startElement(new Elem(null, tagName, args2attrs(attrs), TopScope, false))
    expr match {
      case _: Unit => ()
      case elem: Node => builder.addChild(elem)
      case a: Any => TextNode(a.toString)
    }
    builder.endElement
  }

  def apply(expr: => Any)(implicit builder: XmlBuilder): Unit = {
```

- IntelliJ IDEA for Scala allows a more sophisticated selection of the extraction target when performing *Extract Method*. After invoking the refactoring, it shows a selection of possible extraction targets and highlights the according scope as shown in Figure 4.1. Furthermore, it allows to select both method bodies as well as class scopes as extraction targets. If the former is selected the refactoring creates a closure method in the local scope. *Extract Variable* acts analog to *Extract Local* in IntelliJ IDEA for Java and *Extract Field* is similar to *Extract Constant*.

- CodeRush for VisualStudio allows the selection of the extraction target for *Extract Method* by moving a target picker up and down in class declaration with the arrow keys. The target picker is illustrated as an arrow that points to the position where the new method is inserted between two present methods.

It is striking to note, that the differences between the refactoring tools for Java and Scala are mainly rooted in the differences between the two programming languages. While Java only allows methods on the class level and does not (yet) support anonymous functions Scala is more flexible in its syntax. Because Scala allows method and class definitions in method bodies and heavily relies on anonymous functions it is not uncommon to see code with nested scopes several levels deep.

Another observation is the sometimes inconsistent implementation of the extraction target selection. E.g. Eclipse JDT allows to select a nested class for *Extract Method* but not for *Extract Constant* and IntelliJ IDEA for Scala offers a convenient scope selection drop down for *Extract Method* but not for *Extract Variable*.

Because we want to create simple to use but powerful refactoring tools, we decided to follow a similar approach as used in the *Extract Method* tool in IntelliJ IDEA for Scala. After the user invokes an extraction the IDE proposes several possible target scopes right in the editor window in a drop down menu similar to the *Quick Assist* menu. By pressing the up and down arrow keys he can switch between the scopes. As a visual help the currently selected scope is always accordingly highlighted in the editor. Finally, the user chooses one of the scopes by hitting the enter key or a double click on the item in the drop down menu.

To tie extraction targets to scopes seems to be a good compromise between flexibility and convenience. It allows to fine tune the visibility of the extracted abstraction by moving it to more or less widely accessible scopes but does not require any new and unfamiliar UI components to navigate through the options.

The following listing demonstrates how extraction targets should be handled in the new code extraction tools:

```scala
trait T {
  object O {
    def fn(is: List[Int]) = {
      // Insert in Local Scope 3
      if(is.length > 10)
        is.foreach{ i =>
          // Insert in Local Scope 2
          if(i > 0) {
            val j = i * 2
            // Insert in Local Scope 1
            println(/*(*/j + 1/*)*/)
          }
        }
    }
    // Insert in Member Scope O
  }
  // Insert in Member Scope T
}
```

All available extraction targets when extracting the marked expression `j + 1` are annotated in the listing. We distinguish between member scopes and local scopes. Member scopes are formed by definitions of traits, classes and objects. If the user selects a member scope the new abstraction is always inserted after the member that contains the original code. Local scopes are formed by methods, anonymous functions, case statements, for expressions, variable definitions and blocks or - in other words - by every kind of statement that can introduce new term names. If a local scope is selected as extraction target the new abstraction is inserted as the last statement in the according scope that comes before the extracted code.

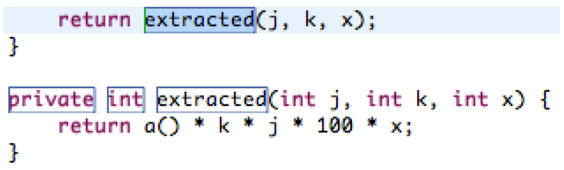
4.2. Inline Renaming

Part of every extraction refactoring is to choose a name for the newly introduced abstraction. Thus, automated extraction tools have to offer a way to name the extracted value, method or other kinds of abstraction.

Basically there are two distinct input methods for names in common refactoring tools. One is to enter the abstraction name in a text field that is part of the modal dialog window along with other configuration options. Usually, the tool also checks for collisions in the targeted namespace and produces a warning or error message if the name is already in use.

The other method is to apply the refactoring with a generated placeholder name and let the user rename it inline right in the editor window. Because, in most cases, a refactoring inserts the abstraction name in the definition as well as in the reference to the abstraction that replaces the original code, a change of the name at one place must also be applied to the other occurrences. This is usually done by using the “linked mode” facility of the editor. Figure 4.2 demonstrates the linked mode in Eclipse. The boxes mark the text snippets that are editable. Boxes containing the same text form a linked group and edits are synchronized between them. Additionally, the user can switch between groups by pressing `Tab`.

Figure 4.2.: Renaming of an Extracted Method in Linked Mode in Eclipse JDT



```
        return extracted(j, k, x);
}

private int extracted(int j, int k, int x) {
        return a() * k * j * 100 * x;
}
```

The first method is mainly useful when the new abstraction will be inserted into another source file and the name is therefore used in two or more files. In this case the tool can help to detect name collisions that may occur in one or another file. Because the extraction tools described in this report perform transformations local to one file we solely use the inline renaming method.

4.3. Inbound and Outbound Dependencies

Except for trivial cases, expressions do usually interact in some way with the surrounding code. When extracting such expressions it is crucial to consider these interactions in order to not create invalid code and preserve its behavior. This is why we also include inbound and outbound dependencies of selections into the discussion of the new refactoring tools.

Every symbol that is defined outside of a selection and referenced inside of it is an inbound dependency. This includes symbols for both values and variables as well as types.

```
object Demo {
  case class Person(title: String, name: String)

  def becomeADoctor(p: Person) = {
    /*(*/println(s"Congratulations ${p.name}")
    val dr = Person("Dr.", p.name)
    println(s"Uhm...${dr.title} ${dr.name}")/*)*/
    dr
  }
}
```

In the above example, the marked code has several inbound dependencies: The parameter value `p`, the method `println` and the class `Person`. Properties of inbound dependencies are not treated as such because they are already covered by their owner.

Outbound dependencies on the other hand are symbols defined in a selection and used outside of it. In the above example, the value `dr` is an outbound dependency of the selected code.

Symbols used in a selection that refer to values, methods or classes are either inbound or outbound dependencies. Symbols referring to variables may become both inbound and outbound dependencies if the variable is reassigned in the selection.

4.4. Extract Value Tool

The *Extract Value* tool is applicable on all statements that evaluate to a value or introduce new local variables. It creates a new `val` definition that includes the extracted code and replaces the original expressions by a reference to the new value. Amongst others, *Extract Value* automatically performs the *Introduce Explaining Variable* refactoring according to [FB99] (often called *Extract Local*). Because variables defined using the `val` keyword are by definition immutable constants and *Extract Value* allows to extract into new class members, it can also be used to perform *Replace Magic Number with Symbolic Constant* (sometimes referred to as *Extract Constant*).

The following class definition shows how *Extract Value* transforms source code by introducing new variables:

```
trait Bodies {
  class Cylinder(r: Double, h: Double){
    val surface =
      // Extract Local Value
      2 * PI * r * r + /*(*/2 * PI * r * h/*)*/

    // Extract Value to Cylinder
  }
}
```

In this case *Extract Value* offers the two possible extractions as they are annotated by comments in the listing. The first extraction “Extract Local Value” creates a new block that becomes the right hand side of the variable definition of `surface`:

```
    val surface = {
      val sideArea = 2 * PI * r * h
      2 * PI * r * r + sideArea
    }
```

The second extraction “Extract Value to Cylinder” adds a new member to the `Cylinder` class:

```
   val surface = 2 * PI * r * r + sideArea

   val sideArea = 2 * PI * r * h
```

Because the referenced variables `r` and `h` are not visible from trait `Bodies`, *Extract Value* does not offer an extraction that adds a new member to the `Bodies` trait.

4.4.1. Extraction of Value Definitions

Scala offers also a native notation to return multiple values from one expression. Therefore, it is also possible to apply *Extract Value* on a sequence of statements that contains one or more term name definitions.

```
/*(*/val a = 1; val b = 2; val c = a + b/*)*/
println(b * c)
```

In this case, *Extract Value* creates a `val` whose right hand side returns a tuple containing every value that is used outside of the extracted code. Additionally, the original code is replaced by a multi assignment that reassigns the required values:

```
val extracted = {
  val a = 1; val b = 2; val c = a + b
  (b, c)
}
val (b, c) = extracted
println(b * c)
```

Note that `a` is not returned by `extracted` because there is no reference to `a` outside of the extracted code.

4.4.2. Extraction of Anonymous Functions

Because functions are also objects in Scala, it is possible to apply *Extract Value* on definitions of anonymous functions:

```
List(1, 2, 3).map(/*(*/_ * 3/*)*/)
```

The extraction of anonymous function requires special treatment because in most cases the compiler can't infer the type of the parameters of the extracted function when the explicit type annotation is not available. This is why *Extract Value* must print in this case the type annotation `Int => Int`:

```
val extracted: Int => Int = _ * 3

List(1, 2, 3).map(extracted)
```

4.4.3. Mutable Variables

If one symbol that is defined inside the extracted code and used outside of it (an outbound dependency) is declared with the `var` keyword, *Extract Value* must ensure that the mutability property remains in the resulting code:

```
/*(*/var i = 1; val j = 2/*)*/
i += j

// becomes
val extracted = {
  var i = 1; val j = 2
  (i, j)
}
var (i, j) = extracted // j is now also mutable
i += j
```

In this case, the assignment to the result of `extracted` uses the `var` keyword instead of `val` because `i` is reassigned in a later instruction. This leads to the probably unwanted effect that the other outbound symbol `j` becomes also mutable due to the refactoring.

4.4.4. Side Effects

When applying *Extract Value* on a selection that triggers side effects (e.g. some output to the console), *Extract Value* can't guarantee that the behavior of the program is preserved.

```
object O {
  def main(args: Array[String]) = {
    print(1)
    /*(*/print(2); 100/*)*/
  }
}
```

In the above example, executing the application results in the output `12`. But after applying *Extract Value* on the marked code and selecting the enclosing object `O` as the extraction target, the program prints `21`. This is because the extraction moves the expression `print(2)` to the initialization block of the extracted class member and is therefore evaluated during the initialization of the class and before the body of `main`.

4.5. Extract Method Tool

Like *Extract Value*, *Extract Method* is applicable to every sequence of statements that does either evaluate to a value or introduces one or more variables. It creates a new method definition based on the selected code and replaces it by the according method invocation. Because a method also allows parameters, extraction targets of *Extract*

Method are not limited to scopes where all inbound dependencies of the extracted code are accessible. Furthermore, *Extract Method* preserves the behavior of the program also when the extracted code invokes side effects. Finally, *Extract Method* behaves analogous to *Extract Value* when applied on anonymous functions and definitions of mutable and immutable values.

4.5.1. Method Parameters

Our implementation of *Extract Method* creates a method with the minimal set of parameters that is required to satisfy all inbound dependencies in the selected target scope. Furthermore, it does not offer any options to change the order of the parameters during the refactoring as in IntelliJ IDEA and Eclipse JDT. This decision is based on the principle outlined in section 3.2 to prefer small and easily invokable refactoring tools. If there is a need for modifying the parameter list of the extracted method, Scala Refactoring offers separate refactorings to alter method signatures. Furthermore, the *Extract to Parameter* refactoring implemented during this project allows to use arbitrary expressions in the new method as additional parameters.

Unfortunately, not every inbound dependency can be used as a parameter to an extracted method. *Extract Method* is only applicable if every inbound dependency is either accessible in the target scope, it is a value declared by a `val` statement or it is a class member accessor. Types, constructors and objects can not become parameters. This is usually not an issue, because it is often not feasible to create methods that operate on inaccessible types.

4.5.2. Reassigned Variables

One limitation of *Extract Method* is the handling of inbound dependencies that are used as parameters and reassigned in the extracted code. The arising problem is demonstrated in the following listing:

```
class A {
  def fn = {
    var a = 1
    /*(*/a += 1/*)*/
    a
  }
}
```

Applying the *Extract Method* refactoring on the marked code with class `A` as the extraction target would result in code like:

```
class A {
  def fn = {
    var a = 1
    a = extracted(a)
    a
  }

  def extracted(temp: Int) = {
    var a = temp
    a += 1
    a
  }
}
```

In this example, ‘a’ is both an inbound as well as an outbound dependency. It has to be passed to the extracted method and must be reassigned to the result of the method call. Additionally, the temporary value `temp` is required due to the immutability of parameters in Scala.

Such extractions that require reassignments of local variables are currently not supported by *Extract Method*. First, they result in some additional edge cases, that are hard to handle like methods that reassign a variable and return a value. Second, the resulting code is not idiomatic for Scala as the language encourages a style that does not uses mutability.

4.6. Extract Parameter Tool

Extract Parameter is mainly a complement to *Extract Method* and adds a new parameter to a method based on a arbitrary expression that evaluates to a value. The refactoring uses the extracted expression as the default value of the new parameter and does not modify any calls to the affected method. Thus, *Extract Parameter* does just apply a small and local transformation that is easily verifiable by the user. To propagate the default value to the method invocations, he uses another (not yet available) refactoring that globally searches for references to the method name and updates all according calls.

The extraction of anonymous functions is handled analog as in *Extract Value*. Because *Extract Parameter* does not support the extraction of expressions with outbound dependencies, mutable variables are not an issue.

Because nested methods are not uncommon in Scala, *Extract Parameter* allows parameter lists of all methods that enclose the current selection as an extraction target. Although, the extraction is only applicable if every inbound dependency is accessible in the according parameter list.

The following example shows how *Extract Parameter* can be used to refactor a specific function into a more general one:

```
def osIsLinux(os: String) =
  os.toLowerCase.indexOf(/*(*/"linux"/*)*/) != -1

println(osIsLinux(system.osName))
```

Applying *Extract Parameter* on the marked code results in the following program:

```
def osIsLinux(os: String, token: String = "linux") =
  os.toLowerCase.indexOf(token) != -1

println(osIsLinux(system.osName))
```

4.7. Extract Extractor Tool

The *Extract Extractor* tool enables automatic extraction of case patterns into new extractor objects with identical semantics and replaces the original pattern by an according call to the extractor. Because extractors are sometimes counterintuitive for programmers not used to the concept, *Extract Extractor*'s goal is to support them by proposing an initial version of an extractor object with the correct signature of the `unapply` method.

4.7.1. Patterns in Scala

Patterns form something like an anti language to Scala. Like normal expressions are compositions of a limited number of atomic components (`if` expressions, literals, method calls etc.), patterns are as well compositions of some atomic elements. But what would look like an argument when used in an expression becomes sort of a return value when used in a pattern (therefore the "anti").

Because there is currently no comprehensive description of all patterns, syntactic sugar for patterns and how they are represented in the AST, we collected the information in this section from [EOW07], the helpful overview of the AST classes in the appendix of [Sto10] and by inspecting the ASTs generated by the Scala compiler.

A pattern is a composition of the following elements:

- **Constant patterns** - Are either literals like `1` or `"hello"` or references to immutable variables as defined by `val` or `object` statements. Matches only itself and are represented by the corresponding `Literal` or `RefTree` trees in the AST.
- **Wildcard patterns** - Have the form _ and match any value. Wildcard patterns are represented by `Ident` trees with name _.
- **Variable binding patterns** - Have the form `x@p` where `x` is a name and `p` is a pattern. Matches the same values as matched by `p` and binds the variable `x` to the matched value. Binding patterns are represented by `Bind` trees.
- **Type patterns** - Matches objects of the according type. Examples are `i: Int` or `_: String`. The former example is actually a shortcut for `i @ (_: Int)` and therefore a composite of a binding pattern and a type pattern. Type patterns are represented by `Typed` trees.
- **Constructor patterns** - Are of the form C(p_1, ..., p_n) where C is a case class and p_1, ..., p_n are patterns matching the constructor arguments that were used

to build the matched value. Constructor patterns are represented by `Apply` trees where `fun` is a reference to the case class constructor C and `args` are the patterns p_1 to p_n.

- **Alternative patterns** - Are of the form $p_1 \mid \ldots \mid p_n$ where p_1 to p_n are patterns. Matches values that match one of the alternative patterns. Are represented by `Alternative` trees.

- **Extractor patterns** - Have the identical form as constructor patterns $E(p_1, \ldots, p_n)$ but E is a reference to an object with an `unapply` method. Such objects are also called extractors. Matches values that have the same type as the argument of the extractor method and where its return values matches p_1 to p_n. Extractor patterns are represented by `UnApply` trees.

- **Variadic extractor patterns** - Have the identical form as extractor patterns $E(p_1, \ldots, p_n)$ but E is a reference to an object with an `unapplySeq` method. Variadic extractor patterns are also represented by `UnApply` trees.

- **Star patterns** - Are of the form `_*` and match remaining arguments of argument lists with variable length. For example the pattern `List(head, rest @ _*)` matches every list with a head element and binds the first element to `head` and a sequence of the remaining elements to `rest`. Star patterns are represented by `Star` trees and are only meaningful as a part of a constructor or variadic extractor patterns.

- **Tuple patterns** - Syntactic sugar for the extraction of tuple values. Are of the form $(p_1, \ldots, p_n)$ and represented internally by a `TupleN` constructor pattern.

Listing 4.1 shows how the described patterns can be used and combined in match expressions. The first three cases demonstrate how the compiler distinguishes between references to constant values and bindings to a new variable. If a name starts with a lower-case letter, it is interpreted as a variable binding pattern. Names starting with an upper-case letter are treated as constant patterns. To use a value whose name starts with a lower-case letter as a constant pattern, the name must be enclosed in ticks as shown in the third example.

Listing 4.1: Examples of patterns and composed patterns

```
obj match {
  // A constant pattern matching any object that equals to the value of ‘None‘
  case None => ???
  // Matches any value and binds it to the variable ‘none‘
  case none => ???
  // A constant pattern matching any object that equals to the value of ‘none‘
  case ‘none‘ => ???
  // A type pattern
  case _: Int => ???
  // A type pattern with a binding to the symbol ‘i‘ of type ‘Int‘
  case i: Int => ???
  // Desugared form of the above
  case i @ (_: Int) => ???
  // A constructor pattern referring to the case class constructor ‘Some‘
      containing a constant pattern
  case Some(123) => ???
  // An alternative between two type patterns
  case _: Int | _: Long => ???
  // A variadic extractor pattern referring to ‘List.unapplySeq‘ with two
      constant patterns and a star pattern; matches any list whose first two
      values are 1 and 2
  case List(1, 2, _*) => ???
  // Does additionally bind the remaining elements to a symbol ‘rest‘ of type
      ‘Seq[Int]‘
  case List(1, 2, rest@ _*) => ???
  // Matches a list by using the cons constructor
  case 1 :: Nil => ???
  // Desugared form of the above
  case ::(1, Nil)
  // Syntactic sugar for matching tuples
  case ("first", "second") => ???
  // Desugared form of the above
  case Tuple2("first", "second") => ???
  // A binding pattern with a guard
  case i: Int if i > 0 => ???
  // A wildcard pattern matching all remaining cases
  case _ => ???
}
```

4.7.2. Abstracting over Patterns with Extractors

Because patterns are composable and can therefore become arbitrary complicated, extractors are sometimes used to abstract over patterns. Thus, a complicated pattern can be replaced by a call to an extractor with a meaningful name. Furthermore, an extractor is a reusable abstraction and can also be unit tested.

Every pattern can be abstracted by creating an extractor that contains a match expression whose first case uses the original pattern. This is best demonstrated by an example:

```
obj match {
  case is @ List(1, _*) => ???
}
```

This pattern matches any list whose first value is a `1` and binds it to the variable `is`. Abstracting the pattern into the extractor `StartsWith1` leads to the following program:

```
object StartsWith1 {
  def unapply(x: Any): Option[List[Any]] = x match {
    case is @ List(1, _*) => Some(is)
    case _ => None
  }
}

obj match {
  case StartsWith1(is) => ???
}
```

The introduced extractor reuses the original pattern in its first case statement and returns an instance of `Some` containing the matched list if the parameter `x` matches. The second case statement catches all remaining values and returns `None` to indicate that the extractor does not match. In this example, `unapply` takes an argument of type `Any` which is the base type of any class in Scala. Another option is to accept only instances of the `List[Any]` type. In this case, the Scala compiler would only call the extractor if `obj` is actually a list.

By using the following template, almost any pattern can be transformed into an identical extractor:

```
object <ExtractorName> {
  def unapply(x: <MatchedType>) = x match {
    case <OriginalPattern> => Some(<BindingsDeclaredInPattern>)
    case _ => None
  }
}
```

Placeholders are enclosed by $<$ and $>$. $<$`ExtractorName`$>$ is a name describing the intent of the extractor. In order to preserve the behavior of the pattern in any case, $<$`MatchedType`$>$ must equals to the type of the expression at the left hand side of the original match expression (`obj` in the above example). $<$`OriginalPattern`$>$ has to be replaced by the original pattern. Finally, every variable bound by the pattern must be referenced in the `Some` constructor as indicated by the $<$`BindingsDeclaredInPattern`$>$ placeholder.

The only pattern that is not abstractable with this template is the star pattern because it is only allowed in the argument list of constructor or variadic extractor patterns.

4.7.3. Automated Extraction of Extractors

The *Extract Extractor* tool uses the template described above to extract arbitrary patterns. Furthermore, it allows the extraction of the complete left hand side of a case statement including the boolean expression in the guard clause.

Like *Extract Value*, *Extract Extractor* allows every scope in which all inbound dependencies of the extracted pattern are accessible as an extraction target. Concerning outbound dependencies, it behaves somewhat different to *Extract Value*, as it returns every bound variable from the `unapply` method and not just those used outside of the pattern. This solution has been chosen because the number of bound variables in a pattern is usually not greater than three or four. Furthermore, there is no need to create a binding in a pattern if there is no intention to use it afterwards because the variable cannot be referenced in the pattern itself. Thus, patterns like `(a, a)` that reuses the bound variable `a` to match only pairs with two identical values are not allowed. The only exception to this assumption is, when a bound value is only used in the if clause of the case statement. E.g. as in the statement `case (a, b) if a == b => ???` that actually matches tuples with two identical values.

The following listing demonstrates how *Extract Extractor* can be used to extract from a pattern:

```
trait Math {
  def simplify(e: Expr): Expr = e match {
    case /*(*/Mult(1, e)/*)*/ => simplify(e)
    case e => e
  }
}
```

The extraction proposes in this case two extraction targets. One in the local scope of the `simplify` method and another in the member scope of `Math`. Choosing the member scope results in the following program:

```
trait Math {
  def simplify(e: Expr): Expr = e match {
    case Identity(e) => simplify(e)
    case e => e
  }

  object Identity{
    def unapply(x: Expr) = x match {
      case Mult(1, e) => Some(e)
      case _ => None
    }
  }
}
```

As mentioned above, *Extract Extractor* can also extract parts of a pattern (which must be a pattern itself) and the complete left hand side of a case statement including the guard clause:

```
trait Math {
  def simplify(e: Expr): Expr = e match {
    case /*(*/Add(Num(l), Num(r)) if l == -r/*)*/ => Num(0)
    case e => e
  }
}
```

Table 4.1.: Refactoring tools used by Extract Code

Selection	Refactoring Tool
Expressions evaluating to a value	Extract Value
	Extract Method
	Extract Parameter
Expressions with side effects	Extract Method
A Pattern	Extract Extractor
A Pattern with a guard	Extract Extractor

In this case the template for extractors described in the last section is also applicable but the if clause must be copied as well:

```
trait Math {
  def simplify(e: Expr): Expr = e match {
    case Zero(l, r) => Num(0)
    case e => e
  }

  object Zero {
    def unapply(x: Expr) = x match {
      case Add(Num(l), Num(r)) if l == -r => Some(l, r)
      case _ => None
    }
  }
}
```

This is one of the cases where the bound variables ‘l’ and ‘r’ are probably not intended to be reused in the right hand side of the case statement but still returned by the unapply call `Zero(l, r)`. But this can be fixed with a relatively small effort by removing the superfluous variable from the expression `Some(l, r)` returned by `unapply` and the according call `Zero(l, r)`.

4.8. Extract Code Tool

The intention of the *Extract Code* tool is to offer a unified invocation method for extraction refactorings and to help the user to explore the available extractions. It is invoked by selecting an arbitrary code snippet in the editor window and pressing the assigned keyboard shortcut or selecting the tool in the “Refactoring” menu. Afterwards, *Extract Code* expands the selection to the next code snippet for which at least one extraction is available and lists these extractions in a drop down menu. After the user chooses one of the extractions it is applied to the program code. Finally, the user can rename the newly introduced abstraction right inside the editor window.

Depending on the selected code, *Extract Code* delegates to at least one of the tools as listed in Table 4.1. Note, that one selection can match several of the conditions. For example the literal `"abc"` in the pattern `List("abc")` is an expression evaluating to a value as well as a pattern because it matches an according string. In this case *Extract Code* offers value extractions as well as extractor extractions.

Because several extraction tools are used to process a selection and one extraction tool offers concrete extractions for each valid extraction target, there are usually many possible extractions. This is illustrated in the following example:

```
object System{
  def printOsInfo(os: String) = {
    // Extract Value or Method to Local Scope
    if (/*(*/os.toLowerCase().indexOf("linux") != -1/*)*/)
      println("Penguins!")
  }
  // Extract Method to Member Scope
}
```

In this example there are three possibilities to perform an extraction. *Extract Code* could either create a value or a method in the local scope of `printOsInfo` or a new method in the member scope of `System`. Creating a member value in `System` is not possible because the inbound dependency `os` is not accessible outside of `printOsInfo`. Because extracting a method from an expression that triggers no side effects into a scope with all inbound dependencies available is in most cases not the desired refactoring result, *Extract Code* does prefer the value extraction if there is a method extraction and a value extraction with the same extraction target. Thus, in the above example *Extract Code* does not display the possible method extraction in the local scope of `printOsInfo`.

4.8.1. Detection of Side Effects

As mentioned above, the absence or presence of side effects indicates if it is more appropriate to extract a value or a parameterless method from a given expression.

This is illustrated by the following listing where one wants to extract the marked expressions into a new method `dropRequest` of the class `Service`:

```
class Service{
  var activeRequests = 0

  def handle(request: String) = {
    if(activeRequests > 100){
      /*(*/println("Request droped")
      None/*)*/
    } else {
      Some(expensiveCalculation(request))
    }
  }
}
```

The selection has no inbound dependencies that are not accessible from the `Service` class and returns the value `None`. When ignoring the side effect triggered by the call to `println`, it seems natural to extract the selection into a new value member. But this changes the behavior of the program because `println` would only be called once during the initialization of the `Service` instance. In order to keep the service printing the message every time a request is dropped, the selection must be extracted into a method.

In order to not propose extractions that may change the behavior of a program, *Extract Code* searches the expressions to extract for potential side effects and proposes either a value extraction or a method extraction if both is possible.

The heuristic used by *Extract Code* to detect side effects is based on the assumption that an expression does either perform some calculation and evaluates to a value or it triggers some side effects and returns nothing. Because in Scala every function and expression must return a value, such an expression does actually return an object of type `Unit`. Hence, the occurrence of the type `Unit` is a strong indicator for side effects. Naturally, this heuristic is not perfectly accurate and does in some cases miss side effects hidden in called method.

4.8.2. Collecting Applicable Extractions

When the user invokes *Extract Code* on a selection, the tool collects proposed refactorings from every supported extraction tool. Because it is also possible that the user selected code on which no extraction is applicable, *Extract Code* tries in this case to expand the selection to the enclosing statement or expression. The algorithm used by *Extract Code* to at one hand find a selection that can be extracted and at the other hand find the according extractions by delegating to the available extraction tools is listed in Listing 4.2

Listing 4.2: Collecting applicable extractions for a given selection

```
Input: A selection of code, a list of supported extractionTools
Output: A list of applicableExtractions

applicableExtractions = empty list
while applicableExtractions is empty {
  for each tool in extractionTools {
    if tool is applicable on selection {
      applicableExtractions += extractions collected by tool for selection
    }
  }
  if selection is expandable {
    selection = selection expanded to the next enclosing AST tree
  } else {
    return empty list
  }
}

return applicableExtractions ordered by target scope
```

5. A Modular Refactoring Architecture

This chapter discusses the architecture used for the implementation of the refactoring tools described in the previous chapter.

The analysis of the extraction tools indicates several requirements to the architecture. First, functionality has to be highly reusable between the distinct implementations. The extraction tools share many aspects of how specific edge cases should be processed. In order to reduce the effort for testing these edge cases and for a consistent behavior of the tool, one needs reusable abstractions of these aspects.

Second, as mentioned in the discussion of *Extract Code* it must be possible to create composed refactoring tools that coordinate and delegate to the according implementations.

Third, a concrete refactoring proposed by a tool must not fail during its application. This implies that all preconditions have to be checked during the preparation phase of the refactoring. Because these checks are highly coupled to the actual transformation, they have to be grouped together with the according implementation of the refactoring logic. Furthermore, it should be possible to share information collected during the precondition checks with the subsequent processing steps.

5.1. Status Quo

The Scala Refactoring library offers several implementations of various refactoring tools. Most of these implementations extends the abstract class `MultiStageRefactoring`. This class offers a common interface for refactoring tools and gives simple access to the source generation functionalities of Scala Refactoring to convert the changes in the AST to actual changes in the source code.

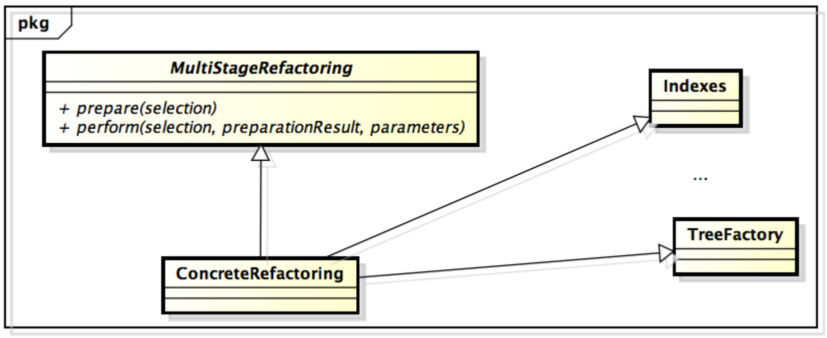
Furthermore, additional functionality is mixed in from reusable components. These components offer convenient tools for code analysis, tree transformations and tree construction. One example of such a component is the `Indexes` trait which offers access to several useful functions for looking up definitions and uses of names in one or more source files.

Figure 5.1 shows a simplified class diagram of the current implementations in Scala Refactoring.

This design meets our architecture requirements when it comes to reusability, and precondition checks and the execution of the transformation are grouped together in a coherent class.

Though, this design is just partially suited for the composition of refactoring tools. Because each sub class of `MultiStageRefactoring` must implement `prepare()` and

Figure 5.1.: Current design of refactoring tools in Scala Refactoring

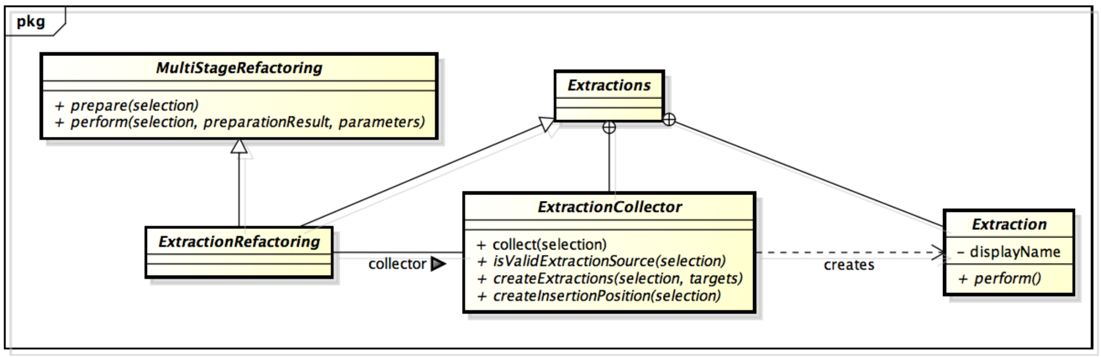


`perform()` in order to be an independently executable refactoring, a composed refactoring tool would have to heavily use `super` calls to specific implementations. Furthermore, this approach would strongly couple the composed refactoring to its components because inheriting from another refactoring is only possible if it uses the identical types for preparation results and refactoring parameters.

5.2. Refactoring Modules

Based on the architecture requirements and the current design of the refactoring library we finally came up with the design illustrated in Figure 5.2 which is used for all refactoring tools implemented during this project.

Figure 5.2.: Architecture used for the implementation of the new extraction tools



The core concepts of this design are `ExtractionCollector` and `Extraction`. An `Extraction` is a concrete transformation of the source code like "extract the expression `2 * a` at offset position 127 into a new value called `aTwice` that lives in the local scope of the enclosing method". Thus, an `Extraction` contains all informations required to perform a concrete transformation including all required and optional parameters. Because of this, instances of `Extraction` have to conform to a very simple interface consisting of the field `displayName` which returns a string describing the according transformation and the method `perform()` which calculates and returns a list of all required transformations to the AST. To sum up, classes inheriting from `Extraction` represent the "perform" phase of `MultiStageRefactoring`.

Instances of `ExtractionCollector` on the other hand implement the "preparation" phase of `MultiStageRefactoring`. An `ExtractionCollector` collects a list of applicable, concrete refactorings for a given selection of code. E.g. the extraction collector for the *Extract Value* tool analyzes the selection for inbound dependencies and returns a list of concrete value extractions for every supported extraction target where all inbound dependencies are accessible. Because all collectors share the same interface and transform a given selection into a list of instances of `Extraction`, a collector may also be implemented as a composition of several different collectors as required for *Extract Code*.

The `Extractions` trait (note the trailing s) is the base trait of every module that offers an extraction functionality and groups extraction collectors with the according `Extraction` classes. `Extractions` do usually offer at least one instance of an `ExtractionCollector` and contain at least one `Extraction` class that represents the concrete extractions returned by the collector.

Finally, the `ExtractionRefactoring` trait is used to adapt a specific collector to the interface forced by `MultiStageRefactoring`. This allows to integrate the new extraction tools by using the same pattern as required by the other implementations offered by Scala Refactoring.

This design is better suited for the requirements stated at the beginning of this chapter: Concrete subtraits of `Extractions` can reuse common functionality by mixing in the according modules and group the extraction collectors with the corresponding `Extraction` classes. Furthermore, it is possible to combine both collectors and instances of `Extraction`.

Such a composition of extraction collectors can be found in the `AutoExtractions` trait which implements the collector used for the unified code extraction tool *Extract Code* as described in section 4.8. The following listing shows how the `AutoExtraction` collector delegates to other collectors:

```scala
trait AutoExtractions extends MethodExtractions with ValueExtractions with
    ExtractorExtractions with ParameterExtractions {
  object AutoExtraction extends ExtractionCollector[Extraction] {
    val availableCollectors =
      ExtractorExtraction ::
        ValueOrMethodExtraction ::
        ParameterExtraction ::
        Nil

    override def collect(s: Selection) = {
      var applicableCollectors: List[ExtractionCollector[_ <: Extraction]] = Nil
      val sourceOpt = s.expand.expandTo { source: Selection =>
        applicableCollectors =
            availableCollectors.filter(_.isValidExtractionSource(source))
        !applicableCollectors.isEmpty
      }

      val extractions = applicableCollectors.flatMap { collector =>
        collector.collect(sourceOpt.get).right.getOrElse(Nil)
      }

      if (extractions.isEmpty)
        Left("No applicable extraction found.")
      else
        Right(extractions.sortBy(-_.extractionTarget.enclosing.pos.startOrPoint))
    }
    // ...
  }
  // ...
}
```

The `AutoExtractions` component mixes in all implemented extraction refactorings. `availableCollectors` holds a list of the supported collectors. Note that `ValueOrMethod Extraction` itself is also a composed collector that proposes value extractions for target scopes with all inbound dependencies accessible and method extractions for the remaining scopes or only method extractions if the selection contains (obvious) side effects.

The overridden `collect()` method implements the algorithm described in subsection 4.8.2 and coordinates the collecting by the supported collectors.

To provide an implementation of `MultiStageRefactoring` we finally have to mix in the `AutoExtractions` trait into `ExtractionRefactoring` and register `AutoExtraction` as the collector used for this refactoring:

```
abstract class ExtractCode extends ExtractionRefactoring with AutoExtractions {
  val collector = AutoExtraction
}
```

This gives us the `ExtractCode` refactoring that conforms to the `prepare()` and `perform()` schema as used by almost any implementation in Scala Refactoring.

6. Additional Refactoring Components

This chapter describes the modifications and enhancements that were made on the Scala Refactoring library during this project.

6.1. Scope analysis

Many of the new refactoring tools require detailed knowledge of where a given symbol is accessible. E.g. *Extract Value* can only insert the new value definition in scopes that see every inbound dependency of the extracted code. In general, the extraction tools perform queries of the form "is symbol `x` accessible in the scope formed by tree `y`?" or less specific "is symbol `x` accessible at position `p`?" whereas `p` is the insertion position of the new abstraction.

Such queries are typically performed for every inbound dependency and for every possible target scope. In order to reduce the overhead of repeated queries for the same symbol we decided to implement a data structure supporting such tasks.

6.1.1. Scope Trees

The data structure used for scope lookups in `ScopeAnalysis` is inspired by the *Symbol Table for Nested Scopes* pattern described in [Par09]. This pattern proposes to build a scope tree with a node for every scope. Each node references its outer scope such that every symbol visible in the inner scope is also visible in the outer scope.

Because we are only interested in the symbols used in a given selection, we do not build the complete scope tree but only the branch that represents all outer scopes of the selection. Figure 6.1 shows the scope tree used by `ScopeAnalysis` for inspecting the scopes based on the selection in the following listing:

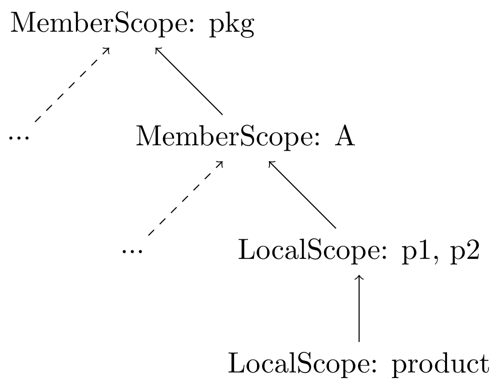
```
package pkg

class A {
  def fn(p1: Int, p2: Int) = {
    val product = p1 * p2
    /*(*/println(product)/*)*/
  }

  def fm(q: Int) = ???
}

class B {
```

Figure 6.1.: A scope tree as used by ScopeAnalysis



```
  // ...
}
```

The dashed lines indicate the missing branches that would have been added to a complete scope tree as proposed in [Par09]. We distinguish between two kind of scopes:

Local Scopes

Each definition in a block introduces a new local scope. This also includes parameters, which are basically value definitions only visible in the methods body. Because it is not possible to reference other parameters of the same parameter list in a parameter definition, all parameters of the same parameter list form a shared local scope (demonstrated by `p1` and `p2` in the above example).

Member Scopes

Packages, classes, traits and objects form member scopes. Opposed to local scopes, symbols defined in member scopes (members) are visible everywhere in the according scope, independent of their declaration order.

In order to query whether a symbol used in the selection is accessible at a given position, one has to traverse the scope tree from the innermost scope to the scope that contains the query position. If one of the traversed scopes defines the symbol, it is not accessible anymore in the outer scopes.

Note that this approach is only feasible when the symbol is effectively accessible in the innermost scope. This can be assumed for all symbols referenced in the selection that was used to build the scope tree. E.g. the `println` symbol referenced in the selection of the above example is not defined by any scope of the tree. Nevertheless, we know that `println` must therefore be globally accessible because otherwise the example would not compile.

6.1.2. Limitations

Because the `ScopeTree` data structure was specifically designed to fulfill the requirements of the extraction tools, there are several limitations that may limit its use for other refactorings:

- The data structure allows only queries for symbols referenced in the selection that was used to built the scope tree.
- The scope tree is limited to the compilation unit containing the selection.
- The scope tree consists only of the branch that leads to the scope of the selection.
- Symbols defined in base classes are currently not associated to the according member scope.

While the former three limitations do not affect the functionality of the extraction tools, the last bullet point can lead in some cases to invalid refactoring results. This issue is illustrated in the following listing:

```scala
trait Base{
  val i = 1
}

trait Outer{
  object Inner extends Base{
    def fn = /*(*/i/*)*/
  }
}
```

Invoking *Extract Value* on the marked code, the tool spuriously proposes the two extraction targets `Inner` and `Outer` because it assumes that the referenced value `i` is globally accessible. But effectively applicable is only the extraction to `Inner` and `i` is not accessible in scopes enclosing `Inner`.

6.2. Import analysis

When one copies an expression from one scope to another, it is possible, that one or more symbols are not imported anymore and could not be resolved by the compiler. This is especially an issue in Scala, because the language allows imports in blocks and class templates. Listing 6.1 shows how the extraction of a value generates broken code when imports are ignored.

Listing 6.1: Issue with imports when extracting expressions

```scala
object O{
  def fn = {
    import scala.math.Pi
    Pi // <- extracted expression
  }

  // Pi is not bound here
  val extracted = Pi
}
```

We identified three strategies to cope with missing import statements in extracted code. The first and seen in most refactoring implementations is to simply ignore missing imports and shift the responsibility to fix the code to the user. While this approach is flexible and obviously simple to implement, it results sometimes in code that requires some labor to be fixed. Especially when the user used wildcard imports in the original code.

The second approach is to print the whole qualifier of references that are not bound in the new scope. In the example above, the extraction would result in a definition `val extracted = scala.math.Pi`.

Finally, one could copy missing import statements at the beginning of the new abstractions body. This approach has the advantage, that it results in valid code that closely resembles the extracted code. Especially when it contains many symbols bound by a wildcard import.

In order to increase correct refactoring results, we decided to not ignore unbound symbols and follow either the second or third approach. Independent of the chosen approach, this requires a tool that helps to determine where in the code a symbol is bound by an import statement and where not. This functionality is provided by the `ImportAnalysis` component.

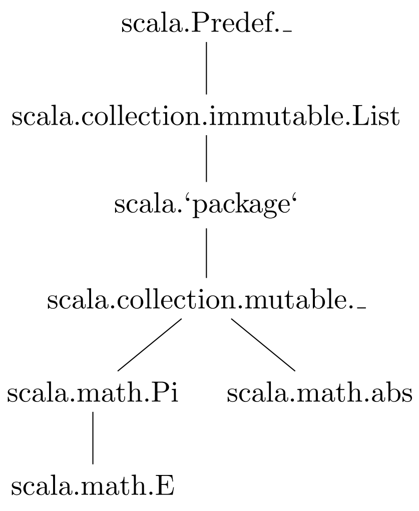
To analyze imports used in a compilation unit or an arbitrary sub tree, we use a data structure called `ImportTree`. Calling `buildImportTree(rootNode)` with `rootNode` as the AST representing the following code builds an import tree as shown in Figure 6.2.

```
// Scala's default imports:
// import scala.Predef._
// import scala.collection.immutable.List
// import scala.`package`
import scala.collection.mutable._

object O{
  import scala.math.{Pi, E}
  // ...
}

object P{
  import scala.math.abs
  // ...
}
```

Figure 6.2.: An import tree as used by TreeAnalysis



Every import tree contains three nodes that represent Scala's default imports. A node represents exactly one selector of an import statement and is either a `WildcardImport` or an `ExplicitImport`. The parent of a node is always the next visible import statement. E.g. `scala.math.abs` sees `scala.collection.mutable._` and is therefore one of its child nodes.

An instance of the import tree structure allows queries of the form "is symbol x imported at position y?". Because the Scala compiler resolves import bindings while building the AST, there is no simple way to determine if a referenced symbol has been imported or the programmer wrote the full qualifier. Furthermore, import selectors in the AST contain only names which do not resemble the full qualifier name of a symbol.

The heuristic used to determine if a symbol could have been imported by a specific import selector, tries to rebuild an import statement from the symbols name and its owners and compares it to the selector's import statement. E.g. there are two imports `scala.collection.mutable` and `scala.math.Pi` in a compilation unit and we want to check if the symbol `Pi` is imported by one of these imports. Comparing the name in the selector of each import with the symbol's name we get the candidate `scala.math.Pi`. Next, we reproduce an import qualifier from the symbols owner chain and get `scala.math` and we can conclude that `Pi` has in fact been imported by `scala.math.Pi`.

While this heuristic works well for imports of package members, it fails on statements importing object members. Beside packages, Scala also allows any immutable value as a qualifier in an import statement [Ode13]. Many uses of this language features are found in the scala-refactoring library in modules that require access to a compiler instance:

```scala
trait ExtractCode extends /*...*/ with CompilerAccess{
  // global is a member of CompilerAccess
  import global._
  // ...
}
```

Here, `CompilerAccess` is an adapter trait that provides a compiler instance in the member `global`. Because a refactoring uses many references to members of `global`, the noise in the code is heavily reduced by importing all these members. But in this case, the import qualifier is `ExtractCode.this.global` and the owner chain of `global` members `scala.tools.refactoring.common.CompilerAccess.global` and the heuristic used in `ImportAnalysis` fails.

6.2.1. Print Missing Qualifiers

Initially, we tried to handle not imported symbols by explicitly printing its whole qualifier in the refactored source code. Because the AST already contains the according qualifiers, it is basically just a matter of forcing the source generator to print these if they are missing. But it turned out, that it is hard to find a feasible heuristic that determines if a qualifier needs to be printed out or not.

Part of the problem is, that the heuristic used to determine if a symbol is imported at a new position returns in some cases false negatives, which leads to unnecessarily printed qualifiers in the extracted code. Because users of the refactoring would have to remove those additional qualifiers manually we want to minimize such cases. Another difficulty is that the Scala compiler desugars many expressions and it is therefore preferred to reuse the original representation in the code rather than to rebuild it from the ground up [Sto10].

The following listing demonstrates the problem with rebuilding extracted expressions:

```
def fn = {
  import scala.collection.mutable.LinkedList
  LinkedList(1)
}
```

When extracting the expression `LinkedList(1)` into a scope where `LinkedList` is not imported, we would have to rebuild the expression with the full qualifier `scala.collection.mutable`. Because the extracted expression is internally represented as `LinkedList.apply(1)`, the refactoring results in the following code:

```
val extracted = scala.collection.mutable.LinkedList.apply(1)

def fn = {
  import scala.collection.mutable.LinkedList
  extracted
}
```

Because it is not likely that this result matches the users expectation, we need an approach that allows to fully reuse the original code.

6.2.2. Copy Missing Import Statements

The second approach to handle missing imports was to copy the according statements to the beginning of the extracted abstraction's body. It turned out, that this approach is more robust and tends to lead to "nicer" refactoring results without unnecessarily bloated expressions.

Applied to the example in subsection 6.2.1, the extraction leads to the following result:

```
val extracted = {
  import scala.collection.mutable.LinkedList
  LinkedList(1)
}

def fn = {
  import scala.collection.mutable.LinkedList
  extracted
}
```

6.3. Enhanced Selections

Almost every refactoring offered by Scala Refactoring takes a selection of code as its initial input and performs further processing steps based on this information. A selection in this sense can be described as a subsequent list of statements and/or expressions in a given context. In Scala Refactoring selections are represented by the `Selection` trait in the `Selections` module and are defined by a range with start and end offset in a source file.

Because especially extraction refactorings depend heavily on the properties of the used selection, this section describes the most important enhancements to the `Selection` class made during this project.

6.3.1. Expansions

One of the first steps performed by automated refactoring tools is usually to check whether the passed selection fulfills some conditions required to successfully apply the desired transformations. E.g. *Extract Value* requires that the selection represents something that evaluates to a value and does not work on type references or class definitions. If the selection does not match the requirements, the tool must either stop the refactoring and report an error or try to recover by finding an appropriate selection. Because we want to offer permissive refactoring tools, we decided to follow the later approach.

The recovery strategy used for the new refactorings is to expand the selection such that it represents an extractable code fragment. This strategy covers mainly accidentally incomplete selections as in the following example:

```
if(/*(*/cond) a else b/*)*/
```

The above selection does select the three expressions `cond`, `a` and `b`. If we would apply *Extract Value* on these expressions, the resulting value definition would be rather meaningless:

```scala
val extracted = { cond; a; b }
```

More likely is, that the user intends to extract the `if` expression as a whole. Thus, the extraction tools must expand such selections to get a valid input.

In order to support expansions of selections, several methods have been added to the `Selection` trait whereas `expand` is probably the most important. `expand` "normalizes" selections such that:

- A selection always encloses at least one AST tree
- Empty selections are expanded to the nearest tree (according to its position in the source file)
- No tree is partially selected

6.3.2. Inbound and Outbound Dependencies

Scala Refactoring already offers with the `TreeAnalysis` trait a facility to analyze selections for inbound and outbound dependencies. These implementations use the index to lookup whether a symbol used inside the selection is declared outside of it (inbound) or whether a symbol declared in the selections is referenced afterwards (outbound). Because building the index is a rather expensive operation and not necessary for every refactoring, we replaced the index dependent implementations by new ones that does not require any index lookups.

The index independent implementations are based on the following observations:

Inbound Dependencies
: As mentioned before, inbound dependencies are symbols defined outside of the selection and used inside of it. This implies, that every symbol used and not defined in the selection must be an inbound dependency. Rather than to check if the definition of the symbol is somewhere else (via the index) it is therefore sufficient to check if the definition is not part of the selection. This can easily be accomplished by collecting all definitions in the selected trees.

Outbound Dependencies
: To detect outbound dependencies, one has to distinguish between local and global accessible symbols. Global symbols are symbols representing packages, classes, traits, objects and members. All other symbols are only accessible in the local scope (function parameters, definitions in blocks and bindings in case statements). If we assume that global symbols are always outbound dependencies, only local symbols remain. Because the scope of local symbols is limited to the block in which they are declared, they can only be referenced after the definition in the

same block or in one of its subtree. Therefore, in order to determine if a symbol defined in a selection is a local outbound dependency, one has only to search the children of the enclosing block after the selection for references to the symbols. This principle is demonstrated in Listing 6.2.

Listing 6.2: Accessibility of local symbols in Scala

```
{
  // println(a) // forward reference to ‘a‘ not allowed
  /*(*/val a = 1/*)*/
  // possible references to ‘a‘ from here
  println(a)
  {
    println(a)
  }
} // no references to ‘a‘ possible
```

6.4. Testing

Scala Refactoring already comes with a rich test suite that covers both the implemented refactoring tools as well as the individual components for analysis, tree construction and code printing and has been steadily expanded during this project. This way, the particular functionalities of the supporting components are well tested on the unit level and their integration and interaction is also covered in the particular tests of the implementations.

This is especially convenient during debugging because a bug can easily be reproduced in a new integration test which in turn is used to track it down to the erroneous component. After the bug is fixed and unit tested, one can verify the fix and affirm that the changes did not accidentally result in further bugs. The only disadvantage is that small changes in the specification of a component may lead to a large amount of failing integration tests. While this is not an issue for components whose behavior can accurately be described (e.g. finding occurrences of a symbol via indexes), it becomes very labor intensive for components with weaker specifications.

This applies mainly to the source generation components. When transforming a modified AST back to a representation in Scala source code, there are almost uncountable possibilities of how this transformation can be done and none of them is clearly right or wrong (as long as it results in syntactically correct code). Especially the handling of layout (whitespace and comments) is rather unpredictable. This is demonstrated by the following test from the `ExtractLocalTest` class (*Extract Local* is a simpler implementation of the *Extract Value* tool described in this project):

```
@Test
def extractValRhs = new FileSet {
  """
    object Demo {
      def update(platform: String) {
        val s = /*(*/platform/*)*/
      }
    }
  """ becomes
  """
    object Demo {
      def update(platform: String) {
        val plt = /*(*/platform
        val s = plt/*)*/
      }
    }
  """
} applyRefactoring(extract("plt"))
```

The mini DSL used in the integration tests allows to specify a set of source files associated with the expected results via the `becomes` method (see [Sto10]). The selection made by a hypothetical user - which becomes the initial input of the refactoring - is annotated by `/*(*/` and `/*)*/` comments. These comments would not exists in the according "real life" code and it is unlikely that comments arranged in this way are often used in genuine code. Hence, it is fairly irrelevant where those comments exactly end up in the refactoring result. In fact, predicting what exactly happens to these annotations is rather hard because this heavily depends on how the Scala compiler calculates the position of an AST node in the source file and may change with future releases of the compiler. Also internals of the implementation can subtly influence the exact position of those comments. For example if a new AST gets an associated position of an existing tree, it may happen that a comment that was adjacent to the existing tree is printed next to the new one.

Because correcting these arbitrary changes of the selection annotations or other minor differences to the layout all over again after unrelated changes of internal components is cumbersome and time consuming, we decided to mainly focus on the semantical modifications to the source files rather than to compare them character by character in the integration tests.

This is done by slightly enhance the integration test DSL with two additional methods `assertEqualTree` and `assertEqualSource`. The following test case from the `ExtractValueTest` class demonstrates how the former of these methods can be used:

```
@Test
def extractFunction = new FileSet{
  """
  object Demo {
    List(1, 2, 3).map(i /*(*/=> i + /*)*/1)
  }
  """ becomes """
  object Demo {
    List(1, 2, 3).map(extracted)
    val extracted: Int => Int = i => i + 1
  }
  """
}.performRefactoring(extract("extracted", 0)).assertEqualTree
```

`assertEqualTree` applies the refactoring to the source file, reparses the result and calls the `toString` method on the reparsed AST to get the uniform string representation offered by the Scala compiler. The expected source is also parsed and serialized by using `toString`. Finally, the two strings are compared. This way, we ensure that the refactoring produces correct code (otherwise the parser would report an error) and check if the semantics of the result match our expectations. But the tests are more robust against changes to the Scala compiler and minor modifications of implementation internals.

Technically it would be sufficient to compare the reparsed AST of the refactored source to the AST of the expectation. But comparing two AST by traversing all children and check if their properties are equal is not as trivial as one would intuitively think because AST nodes are not implemented as pure value objects and usually contain contextual information that should not be asserted (e.g. positions). Furthermore, producing a readable report of the not matched trees in case of an assertion error is another issue. This is why we use `toString` on the reparsed AST. `toString` produces a Scala like, desugared representation of the AST including all inferred types and implicitly applied modifiers and constructors. The following listing shows the output of `toString` applied to the AST of the expectation of the above test:

```
package <empty> {
  object Demo extends scala.AnyRef {
    def <init>(): Demo.type = {
      Demo.super.<init>();
      ()
    };
    immutable.this.List.apply[Int](1, 2, 3).map[Int,
        List[Int]](Demo.this.extracted)(immutable.this.List.canBuildFrom[Int]);
    private[this] val extracted: Int => Int = ((i: Int) => i.+(1));
    <stable> <accessor> def extracted: Int => Int = Demo.this.extracted
  }
}
```

Differences between the actual and the expected AST are now conveniently reported in the same way as JUnit compares other strings and even highlighted when using the JUnit Eclipse plugin.

If the exact outcome of the refactoring is crucial for the user experience one has still the options to use `assertEqualSource` which compares the refactoring outcome character by character to the expected source. This may come in handy when one wants to test if the refactoring did not accidentally print out the desugared form of an expression (e.g. in for-comprehensions).

7. Integration in Scala IDE

This chapter describes how the extraction refactoring tools are integrated into Scala IDE for Eclipse. The integration is done according to the already implemented refactoring tools described in [Sto10].

7.1. Extraction Actions

Because all extraction refactorings use exactly the same work flow, the UI logic is entirely implemented in the common base trait of all extraction actions `ExtractAction`. This trait coordinates the following tasks:

- Initialize the refactoring implementation and provide it with the Scala compiler instance and the current selection in the editor.
- Generate a placeholder name for the extracted abstraction
- Expand the selection in the editor according to the selection that is actually processed by the refactoring
- Revert the selection if the refactoring is aborted
- Display the extraction selection assistant
- Apply the refactoring result to the editor window
- Enter linked mode for in line renaming of the extracted abstraction and eventually its parameters
- Display an error message if no applicable extraction has been found

Opposed to most of the currently available refactoring tools, the extraction tools do not display any wizard pages. Furthermore, all of the currently implemented extractions perform only changes local to one compilation unit. This is why the Eclipse Language Toolkit (LTK) is only used for displaying error messages.

7.2. Extraction Selection Assistant

To display the extractions proposed by an extraction tool as described in chapter 3, we implemented the custom UI component `CodeSelectionAssistant`. This assistant offers the following functionality:

- Displays a drop down similar to the Eclipse quick fix menu with several items
- Each item is associated with a snippet of code in the current editor window
- The code that is associated with the currently selected item is highlighted in the editor

In order to implement this functionality we extended the `ContentAssistant` component of the JFace library. Because `ContentAssistant` offers no callbacks to listen on changes of the currently selected item, we had to implement a custom content assists processor that yields instances of `ICompletionProposal`. This interface defines a method `getAdditionalProposalInfo()` that gets called every time a user selects the item in the drop down and is meant to supply additional information displayed in a window right next to the content assistant. By using this method to invoke selection callbacks it became possible to highlight the according code snippet on selection changes.

While this implementation misuses the content assists in some aspects, it is the only way to provide the required functionality without reimplementing the whole component.

To highlight the selected code snippet we use a marker type called `codeSelection` which is configured in Scala IDEs `plugin.xml` and may be customized by the user via the preferences menu.

7.3. Testing

Developing IDE plugins is one of the rare occasions that allows to use the product while creating it. Therefore, we continuously deployed a custom build of Scala IDE including the new extraction tools and installed the plugin in our development environment.

Using the extraction tools while developing them offered several benefits. First, it allows to test the tools on a real life code base instead of only the toy examples used for the integration tests. Because we had to work both on the Scala Refactoring and the Scala IDE projects, we had also the chance to see how the tools handle fairly different programming styles. While Scala Refactoring relies more on a functional style with none or mere local side effects, Scala IDE is tightly bound to the Eclipse framework and uses therefore many imperative aspects of Scala like loops, reassigned variables and stateful classes.

Additionally, one gets immediate feedback concerning the user experience and performance. For example the initial implementation of the *Extract Method* tool displayed a dialog for modifying the parameters of the new method. While using the tool in production it emerged that the dialog is mostly disturbing and that choosing configuration options in a dialog that hides the concerned code is rather hard. This led to the idea of the more lightweight concept used for the further implementation.

Because the integration layer in Scala IDE is rather small and almost the whole code covers UI interaction, we decided to not use additional automated tests for this layer.

8. Conclusion

This final chapter covers the results of this project and describes how the concepts used for the extraction refactoring tools may be applied to other refactoring tools in Scala IDE to create a powerful tool suite.

8.1. Accomplishments

With this project we show that powerful automated refactoring tools must not necessarily be complex. We've created tools for a specific class of refactoring techniques (extractions) that require minimal user interaction but are more flexible than many comparable tool suites in other IDEs. The tools are based on the following core concepts:

Minimal Configuration
The only configuration options of all extraction tools is the selection of the target scope and the in-line renaming of the extracted abstraction.

Minimal User Interfaces
All user interactions are embedded in the editor window and have therefore a rather small cognitive footprint.

Minimal Refactoring Steps
Every refactoring performs a small transformation that is easily comprehensible by the user.

The tools implemented during this project are *Extract Value*, *Extract Method*, *Extract Parameter*, *Extract Extractor* and a tool composed of the former four called *Extract Code* that offers a uniform invocation method for all extraction tools. With *Extract Extractor* we've created a novel tool that supports the automated abstraction of patterns and gives users new to the pattern matching constructs an easy start to the sometimes counterintuitive syntax of extractors.

Finally, we made some contributions to the Scala Refactoring library that may become an important part of further refactoring tools:

Mini Framework for Modular Refactorings
The described architecture for the extraction tools is technically also applicable to other refactoring tools if their work flow fit to the concept of proposing a small number of applicable transformations.

Enhanced Selections
The additions made to the `Selections` component simplifies the inspection of user selections.

Scope Analysis
The scope analysis component may also be convenient for other tools that move code from one scope to another (e.g. *Pull Up*, *Push Down*, *Extract Class* etc.).

Import Analysis
Import analysis is interesting in similar situations as scope analysis. This component may also be used to improve the pretty printer which does currently print the full qualifier of type references even if the according type name is already bound by an import statement. Furthermore, *Organize Imports* could use the import analysis in order to also support import statements in classes and local scopes. Currently, this refactoring considers only imports listed at the beginning of a compilation unit.

8.2. Limitations and Possible Enhancements

While the new extraction tools are ready to use at the current state, there is still room for improvement:

- As mentioned in section 6.1 the scope analysis component does currently not cover symbols inherited from base classes. Though, this has only impact on few extractions it may lead to incorrect refactoring results.
- The heuristic used for detecting side effects to decide whether a value or a method extraction is more appropriate is rather primitive and does not cover concepts like `Future` and `Try`.
- When the body of an anonymous function that is a single expression becomes a block due to an extraction, the additionally required curly braces are arranged in the style of `() => {...}` while `{ () => ... }` is commonly considered as a nicer style.
- The source code generation component is not yet perfect. During this project we fixed many issues concerning invalid printed code but there are probably still some bugs left. Especially Scalas syntactic sugar for for-comprehensions, right associative methods, infix operators, tuples etc. makes it relatively hard to always generate correct code while reusing the untouched expressions.

Extract Code currently allows to abstract over expressions and patterns but not over types. Thus, there are still some extractions left that would nicely fit into the extraction tools:

Extract Type Alias
Creates based on a selected type a new type member or local type alias. This may be useful for introducing explaining names for composed types like `Either[String, List[Int]]`

Extract Extension
Creates a new implicit class with an extension method based on a selected expression or adds the method to an existing implicit class. Fowler describes a similar refactoring *Introduce Local Extension* (see [FB99] and section 2.3).

8.3. Further Tools

The concepts used to implement the extraction tools can also be transfered to other refactoring tools that commonly make use of heavy modal dialogs. Such tools would also consist of several minimal tools unified in a single invocation method that bundles coherent refactorings.

8.3.1. Change Method Signature

Refactoring tools that support the automated modification of method signatures do successfully reduce the labor required to apply such changes to globally accessible methods. They usually propagate the changes to all calls of the concerned method in the same project or workspace. Additionally, they can be used subsequently to the *Extract Method* refactoring to customize the generated method signature.

Scala Refactoring offers currently implementations for *Change Parameter Order*, *Merge Parameter Lists* and *Split Parameter List* and all three are also integrated in Scala IDE. Unfortunately, the invocation of these refactorings is currently only possible via the "Refactoring" menu and it can take up to several seconds until the required index calculations are performed and the dialog is finally displayed.

A *Change Method Signature* tool that uses a selected parameter as its input and proposes a list of changes applicable to this parameter ("Move p after q", "Split parameter list after p" etc.) would probably be a handy complement to the *Extract Code* tools.

Another useful parameter list refactoring would be *Inline Default Parameter Value* that propagates default parameter values to the actual method calls and may be used subsequent to *Extract Parameter*.

8.3.2. Move Member

Another class of refactorings that may be grouped in a uniform invocation method are tools that move class members in the class hierarchy. This includes for example *Pull Up* and *Push Down Member*, *Extract Superclass*, *Extract Supertrait* and *Move to Companion*. All of these refactorings have in common, that they have to track the dependencies between the members to move and the remaining members and let the user choose how these dependencies should be resolved (either by moving them accordingly or by introducing abstract members).

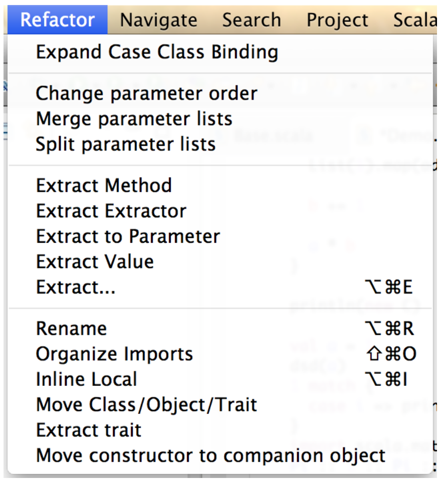
8.4. Acknowledgments

I would like to thank Prof. Peter Sommerlad and Mirko Stocker for their continuous feedback during the weekly meetings and their valuable advices.

A. User Guide for the Extraction Refactorings

The extraction refactorings offer automated extraction of almost arbitrary code fragments into new methods, values or even extractor objects. All you have to remember is that when you want to extract something you have to select it, hit `Cmd` + `Alt` + `E` and the assistant guides you through the available extractions. Naturally, each of the refactorings is also invokable over the refactoring menu.

Figure A.1.: The extraction tools in the Refactoring menu of Scala IDE



An extraction creates a new abstraction (value, method etc.) with the identical behavior as the selected code and replaces the selection by an according call to this abstraction. Extractions are typically useful to make the code more readable and to allow the reuse of a code fragment in other places.

"Extract..." proposes based on the selected code one or more of the following extractions:

Extract Value
: Creates a new `val` definition from the selected code and replaces the selection by a reference to the new value

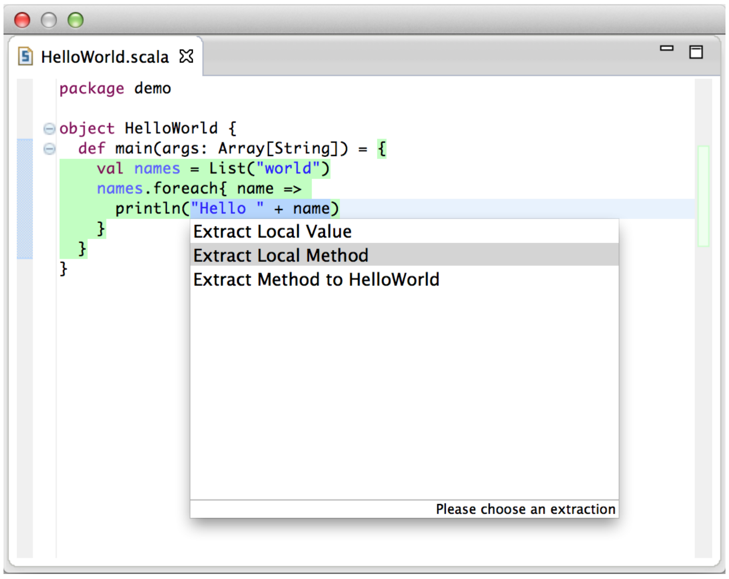
Extract Method
: Creates a new method (`def`) from the selected code and replaces the selection by the according call

Extract Parameter
: Adds a new parameter to an enclosing method whose default value is the selected code and replaces the selection by a reference to the new parameter

Extract Extractor
: Creates a new extractor object based on a selected pattern in a case statement and replaces the pattern by an according call to the new extractor

After invoking "Extract..." a drop-down opens that contains a list of available extractions. When you select one of the extractions with the `Up` and `Down` keys the scope in which the extraction creates the new abstraction is highlighted green. Additionally, if you select only a part of an expression that can be processed by "Extract..." the selection is automatically expanded such that you see what exactly will be extracted.

Figure A.2.: The extraction selection assistant. The code marked green represents the target scope of the selected extraction.



```
package demo

object HelloWorld {
  def main(args: Array[String]) = {
    val names = List("world")
    names.foreach{ name =>
      println("Hello " + name)
    }
  }
}
```

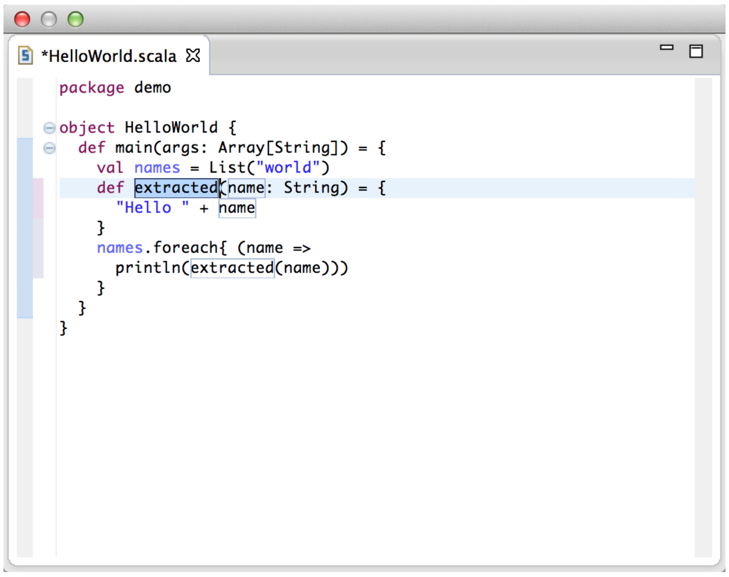
Extract Local Value
Extract Local Method
Extract Method to HelloWorld
Please choose an extraction

When you found the extraction that you want to become applied you can choose it by hitting `Enter` and the extraction tool performs the according transformations to the source code. Finally, Eclipse enters the linked mode that allows you to rename the new abstraction right in the editor window. If you extracted a method with parameters you can also jump to the parameters by hitting `Tab` and rename them as well.

The extraction refactorings offer many useful features:

- Only one shortcut for all kinds of extraction refactorings
- Lets you precisely choose the scope in which you want to create the new abstraction

Figure A.3.: Inline renaming of an extracted method and its parameter



```
package demo

object HelloWorld {
  def main(args: Array[String]) = {
    val names = List("world")
    def extracted(name: String) = {
      "Hello " + name
    }
    names.foreach{ (name =>
      println(extracted(name)))
    }
  }
}
```

- "Extract…" helps you to choose the right extraction by analyzing the selected code
  - If the code triggers no side effects…
    - it proposes a value extraction for every scope in which all variables referenced in the selection are accessible
    - and a method extraction for every scope in which at least one of these variables is not accessible anymore
    - and a parameter extraction for every method that encloses the selection
  - If the code triggers side effects…
    - it proposes only method extractions
  - If the code is a pattern in a case statement…
    - it proposes extractor extractions
- All tools support idiomatic features of the Scala language
  - Extraction of higher order functions
  - The resulting code uses type inference when possible
  - Extraction into local closure methods
  - Multiple return values with tuples

B. Bibliography

[Dev] DevExpress. Coderush. [Online; https://www.devexpress.com/Products/CodeRush/].

[EOW07] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007 – Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[FB99] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, 1999.

[Jet] JetBrains. Intellij idea. [Online; http://www.jetbrains.com/idea/].

[Ker04] Joshua Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004.

[MHPB09] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[MPT78] M. D. McIlroy, E. N. Pinson, and B. A. Tague. Unix Time-Sharing System Forward. Technical report, Bell Laboratories, March 1978.

[Ode13] Martin Odersky. The scala language specification version 2.8. Technical report, 2013.

[Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Pragmatic Bookshelf, 1st edition, 2009.

[Rü09] Michael Rüegg. Pattern matching in scala. Technical report, 2009.

[Sca] ScalaIDE. Scala ide for eclipse. [Online; http://scala-ide.org/].

[Sto10] Mirko Stocker. Scala refactoring. Technical report, IFS, 2010.

[VCN$^+$12] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 233–243, Piscataway, NJ, USA, 2012. IEEE Press.