Refactoring for Dynamic Languages

Rafael Reia, rafael.reia@tecnico.ulisboa.pt

Instituto Superior Técnico, Universidade de Lisboa

Abstract. Dynamic languages are becoming popular, specially among inexperienced programmers. However, refactoring tools for dynamic languages have less functionality when compared with refactoring tools for static languages, because they have less information at compile time, particularly type information. The lack of refactoring tools for a language forces the programmer to do tedious and error prone work to improve the quality of the software. This is even worse for inexperienced users, because it requires specific knowledge about the language making it even more error prone. Due to the increasing popularity of dynamic languages and because of their utility for teaching introductory courses, the need for refactoring tools for dynamic languages is now evident. This paper proposes the creation of a refactoring tool adequate for inexperienced users.

Keywords: Refactoring, Dynamic Languages, Pedagogy

1   Introduction

Over time, software tends to change, for instance when existing requirements are changed or new requirements are adopted. Even after development, bugs are found or some critical new features are added. These changes make the software drift apart from its original design. Typically these changes increase the complexity of the software, making it less readable and harder to change [1]. Consequently, it lowers the quality and increases maintenance costs. Continuous changes create the need for continuously improving the software structure.

Refactoring is a meaning-preserving transformation or a set of meaning-preserving transformations that are meant to improve the program structure and therefore the software quality [2]. Refactoring is a very desirable activity to maintain the software quality, but because it is tedious and error prone, it is preferable to use a tool that provides automated support for it, saving time and preventing the addition of errors to a previously correct program.

There are many refactoring tools available but, unfortunately, they require considerable programing experience, making them unsuitable for inexperienced programmers.

However, these are the programmers that would definitely benefit from these tools. Indeed, when an inexperienced programmer implements a solution to a given problem, it tends to write large fragments of code, lacking the modularity that an experienced programmer would do. For example, see Listing 1.1 which

is an implementation in Python of a algorithm to print the first N Fibonacci numbers. This implementation can be improved if, instead of having one big function, we subdivide it in smaller functions that do the same job. To do that the user needs to extract functions, which is a refactoring operation. By having a refactoring tool that allows the user to extract functions the inexperienced users would improve the quality of the program in a safe way, as is visible in Listing 1.2

Listing 1.1. Algorithm first implementation

```python
def fibonacci(number=1):
        previous = 0
        current = 1
        fib_numbers = []

        for i in range (0, number):
                aux = current + previous
                previous = current
                current = aux
                fib_numbers.append(current)

        for i in range(number):
                print fib_numbers[i]
```

Listing 1.2. Algorithm after using extract function

```python
def fibonacci_seq(number):
        previous = 0
        current = 1
        fib_numbers = []
        for i in range (0, number):
                aux = current + previous
                previous = current
                current = aux
                fib_numbers.append(current)
        return fib_numbers

def print_fibonacci(fib_numbers, number):
        for i in range(number):
                print fib_numbers[i]

def fibonacci(number=1):
        fib_numbers = fibonacci_seq(number)
        print_fibonacci(fib_numbers, number)
```

In the extract function refactoring operation, the user starts by choosing a set of expressions to extract and transform into a function, usually it is a basic

block or a set of basic blocks, where a basic block is a portion of code that only has one entry point and one exit point. In the algorithm that prints the first N Fibonacci numbers example in Listing 1.2, the two functions extracted are the "print_fibonacci" and "'fibonacci_seq".

The use of refactoring tools is fully adopted by the object oriented and statically typed programming languages with their IDEs (integrated development environments) support. For example, languages such as Java have IDEs such as, Eclipse [3], IntelliJ [4] or NetBeans [5], while C# has Visual Studio [6]. Unfortunately that is not the case for the refactoring tools for dynamic languages. The lack of information available during the refactoring is the biggest difference between refactoring tools for static languages and refactoring tools for dynamic languages. This difference is the main difficulty that made the refactoring tools for dynamic languages not evolve as the ones made for static languages, therefore making the refactoring tools for static languages largely used and considered a common tool in contrast with the refactoring tools for dynamic languages.

Nonetheless, the importance of dynamic languages is growing. Dynamic languages are becoming popular among the scientific community. In addition, dynamic languages are often used in introductory courses around the world, for example, Scheme, Racket and Python.

A refactoring tool for a dynamic language adequate for inexperienced users would be an important step in filling the lack of refactoring tools for dynamic languages and it would also help the inexperienced users to have the first contact with refactoring tools and improve their code quality. Creating such refactoring tool is the problem that we want to solve.

Section 2 addresses the objectives for this thesis work. Section 3 describes some definitions related to refactoring tools. Section 4 describes how the users refactor, it compares refactoring tools for static languages, it describes refactoring tools for dynamic languages, and language-independent refactoring tools. Section 5 describes the architecture of the proposed solution. Section 6 explains how the tool will be evaluated and finally section 7 sums up the work.

2   Objectives

The main goal of this thesis is to implement refactoring operations adequate for inexperienced users. Because it is targeted for inexperienced users it is ideally designed for a language that is used to teach programming in introductory courses, for example Racket or Python. It is also important to have a pedagogic environment instead of a complex IDE such as Eclipse. Having a refactoring tool for inexperienced users is important because those users tend to introduce errors in the programs they attempt to modify. Using a refactoring tool to do the refactoring operation will do a quicker and better job because it is safer than doing it manually.

Those operations must be:

- Correct
- Simple to use

– Useful

These characteristics are fundamental to create a refactoring tool targeted for inexperienced users.

3   Definitions

This section presents some definitions regarding refactoring activities.

3.1   Classification of the refactoring

There are several levels of refactoring, from a high level refactoring, like design refactoring, to a low level refactoring such as the extract method refactoring operation. In between, exists combinational refactoring that is a combination of several low level refactoring operations. Refactoring operations can also be classified by the effect they have on software quality attributes. In order to classify the effect based on software quality attributes it is necessary to map the changes in the internal software quality metrics, e.g. lines of code, cohesion or coupling, to the external software quality attributes, e.g. adaptability, reusability or testability. [7] However, this is a complex task that is outside the scope of this thesis and therefore it will not be further detailed.

3.2   Refactoring as a Process

Refactoring can be done in two ways. One is doing the refactoring in between programming and constantly performing it. The other one is to do the refactoring as a separate activity and done in bulk. Regardless when is done, the refactoring can always be decomposed in different activities: [8]

1. Identify where to change the software
2. Determine the adequate refactoring operation
3. Have a way to protect the planned changes (e.g, automated tests)
4. Make the planned changes
5. Access the refactoring benefits
6. Maintain consistency between refactored and non-refactored code

3.3   Refactoring Correctness

Refactoring must preserve the program's behavior in order to be correct. However, there is no consensual definition for what behavior preservation is. Some authors say that the behavior of the program is the output and preserving the input-output behavior preserves the program behavior suggested by Opdyke [9]. However, other authors think that preserving the output is insufficient, since other aspects may be relevant as well. For example, for real-time software, the execution time of certain operations is an important aspect of the behavior.

One way to deal with behavior preservation is to have an extensive set of test cases and if all these tests still pass after the refactoring, it is highly probable that the refactoring is correct [10]. A more formal approach is to prove that the refactoring operations preserve the full program semantics. For Prolog, that has simple and formally defined semantics, it is simple to prove that refactoring preserve the program semantics [11]. But for more complex languages such as C++, which the formal semantics is extremely difficult to define, typically it is necessary to put restrictions to the refactoring operations or to the language constructs and the refactoring tool may be limited to a particular version of a particular compiler [12].

3.4   Case Study - Manual Refactoring

One way to learn how the users manually refactor is to do it while taking notes. The case study [13] consists in refactoring an Haskell program with 400 lines, written by a student. The program's goal is to build a semantic tableaux, which is a truth tree used for example to proof procedures for first order logic or solve satisfiability of finite sets.

In order to better understand in what consists a refactoring they applied manual refactoring operations to the program. They started by changing the name of some variables to avoid misunderstandings and to be easier to read. After that they renamed some functions to names that reflect better what the function did. Then they replaced explicit recursion by calls to higher order functions and they rename some variables and functions. In the end they generalized some functions and modified the representation type because it was giving a lot of work keeping the initial representation.

This case study shows that the order of the refactoring operations is somehow arbitrary. The refactoring operations were applied whenever they thought it made more sense. They also conclude that generally, refactoring a program is a good way to find out more about the program. And that the refactoring operations need to have a way of doing undo, redo or revert changes, otherwise it would be more difficult to correct mistakes.

It is crucial to document the refactoring operations applied in detail. This aspect was stressed because having documentation about the version previous to the refactoring, or outdated, is not good for the readability of the program since it can mislead the programmer.

3.5   Classification of refactoring tools

Refactoring tools can be subdivided in manual, semi-automated and fully-automated according to the degree of automation. In the manual case there is no support for detecting refactoring opportunities, but the transformation is applied by the refactoring tool. If the transformation itself is left to the user, the tool can not be considered a refactoring tool. The fully-automated one, automatically identifies refactoring opportunities and automatically applies them. Finally the

semi-automated one identifies refactoring opportunities but waits for the user to decide the application of the refactoring.

3.5.1   Manual Refactoring Tool is a refactoring tool that applies the refactoring operations selected by the user. Is available for all kind of languages and it is the most common level of refactoring. There are several examples of this tools, including Eclipse and IntelliJ that are focused in static languages and for dynamic languages there is the refactoring browser for Smalltalk. This kind of tools will be further detailed in the related work section.

3.5.2   Semi-Automated Refactoring Tool is a refactoring tool that suggests refactoring opportunities to the user and applies the refactoring operations that the user selected. In order to know what refactoring operations to do, the tools can use metrics that can support the decision of where and which refactoring operations to apply. The tool [14] was created as proof of concept and it uses the metrics to identify where the code should be refactored. The tool takes into account the "bad smell" of a code to suggest a refactoring. A "bad smell" is a human intuition in which a specific code should be refactored. An example of a "bad smell" that trigger a Move method refactoring, which moves a method from one location to another, occurs when one method is used more by other class than the class in which the method is defined.

To quickly show to the user the identified "bad smells" a visualization of the methods and attributes is generated and those objects are linked to the corresponding source code, as it can be seen in the Figure 1.

In order to make an automated approach to identify "bad smells" a distance based cohesion metric is used. With the distance based algorithm it is possible to identify violations to the rule "put together what belongs together". There are some refactoring operations that are related to this rule such as, move attribute, extract class, in-line class and move method. Regarding the distances, a method using only locally defined methods or attributes has a high distance to the methods of other classes, whereas methods that use many attributes and/or methods of other classes have a low distance to them. The attributes are compared by the methods that use them. For example if an attribute is only used by methods of other classes, that attribute probably should be moved to a different class.

3.5.3   Automated Refactoring Tool is a refactoring tool that automatically applies the refactoring opportunities that the tool detects. This kind of tool is useful when doing a source to source transformation, eliminating features that are not necessary and translating them into equivalent ones. For example, if there is a profiling tool that only works in the previous versions of Java 1.4 and the user wants to use such profiling tool, an automated refactoring tool can be used in order to refactor the program by translating the new features, such as anonymous classes, into equivalent ones. However, automated refactoring tools can also be used like a normal refactoring tool but has some restrictions because
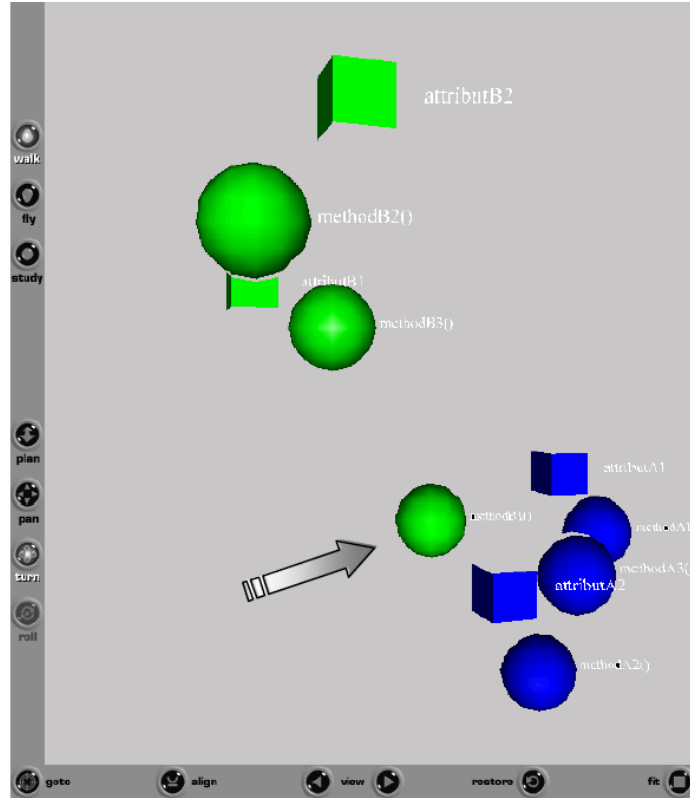
Fig. 1. Motivates the refactoring move method

the user do not know what the refactoring tool is doing. Casais [15] or Moore [16] are good examples of those refactoring tools.

3.5.4   Analysis: Having the automated refactoring tool for inexperienced users is not what is intended. Transforming automatically the program will create a new program that the user might not comprehend, especially if the user is inexperienced.

The Semi-automated tool with the suggestions would be an advantage to inexperienced users that are still learning what refactoring operations exists and that way the users would learn new refactoring operations and have programs with better quality. However, detecting refactoring opportunities is highly dependent on the application domain, witch invalidates this type of tools since they are not meant for one type of application only.

The ideal approach is a Manual Refactoring tool that applies exactly what the user wants to do. This type of tool do not automatically detects refactoring opportunities, but it is faster and safer than doing the refactoring operation without any support.

4   Related Work

This section starts to present the use of refactoring tools and an overview of static refactoring tools. Afterwards it presents the refactoring tools for the dynamic languages such as, Scheme, JavaScript, Smalltalk, Python and Racket. Then it presents the language-independent refactorings. Finally it has a conclusion about the related work.

4.1   Use of static refactoring tools

Understanding how users refactor and use refactoring tools is an important step to better improve the later. The information necessary to reason about how the users refactor was gathered by collecting some data sets [17].

The User data set was collected [18] in 2005. It has records of 41 volunteer programmers using eclipse, from which, 95% of them programmed in Java.

The Everyone data set was collected from the Eclipse Usage Collector. The data used aggregates activity from over 13000 Java developers between April 2008 and January 2009 and it also includes non-Java developers.

The Toolsmiths data set that consists in information about 4 developers who primarily maintain eclipse's refactoring tools from December 2005 to August 2007. However, it is not publicly available and it is not described in other papers. There is only a similar study [19] that uses data from the author and another developer.

Using all the data sets it is possible to see which are the most common refactoring operations used by the users and they are: rename, extract local variable, inline, extract method and move. The sum of the use percentages of this refactoring operations is between 86.4% and 92% of the data sets.

However the refactoring behavior differs among users. The most used refactoring operations is the rename for all the sets, but the used percentage drastically differs between Toolsmiths and the other sets. Toolsmiths usage of the rename refactoring is 29% while the User set and Everyone set is 62% and 75% respectively.

Using the data sets of Users and Toolsmiths it was possible to confirm that refactoring operations are frequent. In the Users data set, 41% of programming sessions contained refactoring activities and the sessions that did not have refactoring activities were the sessions where less edits were made. In the toolsmith data set only 2 weeks of the year 2006 did not have any refactoring operation and, in average, had 30 refactoring operations per week. In 2007 every week had refactoring activities and the average was 47 refactoring operations a week.

Besides refactoring operations being frequent, the refactoring tools are underused. After evaluating the refactoring activities in the data set they were unable to link 73% of the refactoring operations to a tool supported refactoring. All these numbers are computed from the Toolsmiths data set which is in theory the group who knows, and better uses the refactoring tools.

Table 1. Refactoring operations available by default

Refactoring \IDE	Visual Studio	Eclipse	CDT	IntelliJ	NetBeans	Jbuilder
Rename	x	x	x	x	x	x
Move		x	x	x	x	x
Change method signature		x		x		
Extract method	x	x	x	x		x
Extract local variable		x		x		x
Extract constant		x	x	x		
Inline		x		x	x	
To nested		x		x		
Move type to new file		x		x		
Variable to field		x		x		
Extract superclass		x	x	x	x	
Extract interface	x	x		x	x	
Change to supertype		x			x	
Push down		x		x	x	
Pull up		x		x	x	
Extract class		x		x		
Introduce parameter		x		x	x	
Introduce indirection		x				
Introduce factory		x		x	x	
Encapsulate field	x	x		x		
Generalize declared type		x				
Type Migration				x		
Remove Middleman				x		
Wrap Return Value				x		
Safe Delete				x	x	
Replace Method duplicates				x		
Static to instance method				x		
Make Method Static				x		
Change to interface				x		
Inheritance to delegation				x		

Table 2. Refactoring operations definitions:

Refactoring name	Definition
Rename	Renames the selected element and corrects all references.
Move	Moves the selected elements and corrects all references.
Change signature	Change parameter names, types and updates all references.
Extract method	Creates a new method with the statements or expression selected and replaces it with a reference to the new method.
Extract local variable	Creates a new variable assigned to the expression selected and replaces it with a reference to the new variable.
Extract constant	Creates a static final field from the selected expression.
Inline	Inline local variables, methods or constants.
To nested	Converts an anonymous inner class to a member class.
Move type to new file	Creates a new compilation unit and updates all references.
Variable to field	Turns a local variable into a field.
Extract superclass	Creates a new abstract class, changes the current class to extend the new class and moves the selected methods and fields to the new class.
Extract interface	Creates a new interface and makes the class implement it.
Change to Supertype	Replaces, where it is possible, all occurrences of a type with one of its supertypes.
Push down	Moves a set of methods and fields from a class to its subclasses.
Pull up	Moves a field or method to a superclass, if it is a method, declares the method as abstract in the superclass.
Extract class	Replaces a set of fields with new container object.
Introduce parameter	Replaces an expression with a reference to a new method parameter and updates all callers of the method.
Introduce indirection	Creates an indirection method delegating to the selected method.
Introduce factory	Creates a new factory method, which calls a selected constructor and returns the created object.
Encapsulate field	Replaces all references to a field with getter and setter methods.
Generalize type	Allows the user to choose a supertype of the selected reference.
Type Migration	Change a member type and data flow dependent type entries.
Remove Middleman	Replaces all calls to delegating methods with the equivalent calls.
Wrap Return Value	Creates a wrapper class that includes the current return value.
Safe Delete	Finds all the usages or, simply delete if no usages found.
Replace duplicates	Finds all the places in the current file where the selected method code is fully repeated and change to corresponding method calls.
Static to instance	Converts a static method into an instance method with an initial method call argument being a prototype of newly created instance method call qualifier.
Make Method Static	Converts a non-static method into a static one.
Change to Interface	Used after using Extract an Interface it searches for all places where the interface can be used instead of the original class.
Inheritance to delegation	Delegates the execution of specified methods derived from the base class/interface to an instance of the ancestor class or an inner class implementing the same interface.

4.2   Overview of static Refactoring tools

Table. 1 compares refactoring tools with each other and a explanation of the refactoring operations, that were taken from Eclipse[1] and IntelliJ[2] can be found on **table. 2**.

Table. 1 compares the Visual Studio refactoring operations for C# with the CDT refactoring operations for C++ and with Eclipse refactoring operations for Java, because the languages have similarities and the refactoring operations are similar.

The table only lists the refactoring operations that each IDE has by default in order to have a fairer way to compare them with each other. It is easy to see that IntelliJ has almost all the refactoring operations in this table, followed by Eclipse and NetBeans. However, even having significantly less refactoring operations available by default than the other tools, JBuilder[20] has the most used ones as shown above. Visual Studio has only 2 out of the 5 most used refactoring operations available by default, but there are easily installed plugins that cover the more important refactoring operations.

4.3   Haskell

HaRe [21] is a refactoring tool for Haskell that integrates with Emacs and Vim. This tool was made for being used instead of being a proof of concept prototype and it is implemented in Haskell. The system also allows the users to design their own refactoring operations using the HaRe API.

The HaRe system uses an AST (abstract syntax tree) of the program to be refactored in order to reason about the transformations to do. The system has also a token stream in order to preserve the comments and the program layout by keeping information about the source code location and the comments of all tokens.

4.4   Scheme

Griswold [22] proved that meaning-preserving restructuring can substantively reduce the maintenance cost of a system. A prototype was created to prove the concept, by creating restructuring operations for the Scheme programming language implemented in Common Lisp. The prototype was developed for Scheme because of its imperative features, simple syntax and was available an implementation in Common Lisp of a Scheme PDG (program dependence graph). The prototype had simple restructuring operations to prove the concept, such as: moving an expression, renaming a variable, abstracting an expression, extracting a function, scope-wide function replacement, adding a reference indirection and adding looping to list references.

---

[1] help.eclipse.org/luna/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm

[2] https://www.jetbrains.com/idea/features/refactoring.html

4.4.1   Tool aided vs Manual Restructuring Griswold starts comparing automated restructuring with manual restructuring. To do that Griswold creates an experience. It was given an initial program and a description of four transformations goals to six subjects, although it was an experience with a small number of subjects, Griswold took several conclusions on how people manually restructure programs. People use the copy/paste and the cut/paste paradigms to do the transformations. This means that they copy or cut the code and then paste it in the desired location. Although the Cut/Paste paradigm is more dangerous because it is a destructive operation. This means if the user makes any error it is more difficult to correct it because it is a destructive operation.

Griswold also concluded that people make mistakes even with small and simple programs, and the cost of correcting mistakes is higher than the time to do the restructuring itself. It also concludes that manual restructuring is haphazard. Meaning that the transformations were done in almost a random way when compared to the computer-assisted process.

4.4.2   Architecture In order to be able to correctly implement these operations it was used contours and a PDG.

The main program representation is the contours. Contours are an abstraction of the essential semantic properties that the AST represents in an efficient and complete form.

Whereas the PDG explicitly represents the key relationship of dependence between the operations in the program. The PDG is used because simple graph algorithms can extract this information and it has been a popular program representation for aiding program parallelization, optimization and version merging. These features combined with the right semantic support make the PDGs a good foundation for preserving meaning during restructuring. With these structures it is possible to combine them and have a single formalism to reason effectively about flow dependencies and scope structure.

4.5   Smalltalk

The Refactoring Browser [23] is a refactoring tool for Smalltalk programs which goal was to make refactoring known and used by everyone. To quote them *"The goal of our research is to move refactoring into the mainstream of program development. The only way this can occur is to present refactorings to developers in such a way that they cannot help but use them"*.

To do that they implemented the refactoring browser with the concern that the refactoring operations did by the programmer using the refactoring browser needed to be faster than by hand.

It was considered that the user of this tool is intelligent therefore automated refactorings would not suit them. Automated refactorings do not suit the user because they could generate code that would not make sense in the domain. For example, the tool would generate new classes in order to eliminate duplicated code, by creating an abstract class, which might not make sense in the domain.

Instead of doing that, the tool points out possible refactoring operations and lets the user decide whether or not to do those operations.

In order to ensure behavior preservation the tool checks the preconditions of each refactoring operation before execution. However, there are some conditions that are more difficult to determine statically, such as dynamic typing and relationships cardinality between objects. Instead of checking the precondition statically the refactoring browser checks the preconditions dynamically.

The preconditions checks are done using method wrappers to collect runtime information. The Refactoring Browser starts by doing the refactoring operation and then it adds a wrapper method to the original method. While the program is running, the wrapper detects the source code that called the original method and changes it for the new method. For example, in the rename operation, after applying the rename and while the program is running, whenever the old method is called, the browser suspends the execution and changes the code that called the old method, so that it now calls the new method. The problem of this approach is that the dynamically analysis is only as good as the test suit used by the programmer.

However, there is already a tool [24] for Eclipse that receives as input the source code and the refactoring operation to be applied. Then it generates unit tests to the code and at the end it reports if the refactoring is safe to apply or not.

The tool uses a static analysis to identify methods that have exactly the same modifier, return type, qualified name, parameters types and exceptions thrown. After identifying those methods it uses Randoop [25], a tool that generates unit tests for classes within a time limit.

Pairing this tool with the Refactoring Browser would remove the main limitation of the Refactoring Browser. However the tool is created for static languages and it is not that trivial to create one for dynamic languages because the tools have less information in compile time.

4.6   JavaScript

There are few refactoring tools for JavaScript but, there is a framework [26] for refactoring JavaScript programs. One reason for the low number of refactoring tools is the additional difficulty that the refactoring tools have to deal with, when compared with refactoring tools made for static languages. For example, when refactoring JavaScript, the refactoring tools do not have information about the bindings in compile time.

In order to guarantee the correctness of the refactoring operation the framework uses preconditions, expressed as query analyses, that are provided by pointer analysis. The pointer analysis produces over-approximations of sets in a safe way to have correct refactoring operations. For example in the rename, it over-approximations the set of expressions that must be modified when a property is renamed. By using over-approximations it is possible to be sure when a refactoring operation is valid. However, it has the catch of not applying every possible refactoring operation, because the refactoring operations that the

framework cannot guarantee behavior preservation are prevented. To prove the concept, three refactoring operations were implemented, namely rename, encapsulate property and extract module.

It was expectable that this framework would prevent a big percentage of refactoring operations, however, after testing with 50 JavaScript programs, the overall unjustified rejections were 6.2% of all the rejections. The rejections regarding to imprecise preconditions represent 8.2%. Finally, the unjustified rejections due to imprecise pointer analysis were of 5.9% for the rename and 7.0% for the encapsulate property.

4.7   Python

The following section presents two refactoring tools for Python. It starts with Bicycle-Repair-Man[3], a refactoring tool that attempts to create a refactoring browser. Afterwards it presents Rope[4], a refactoring tool that works like a Python library.

4.7.1   Bicycle Repair Man is a Refactoring Tool for Python written in Python. This refactoring tool can be added to IDEs and editors, such as Emacs, Vi, Eclipse, and Sublime Text. However, this tool did not improve since 2004.

Bicycle Repair Man is an attempt to create the refactoring browser functionality for Python and has the following refactoring operations: extract method, extract variable, inline variable, move to module and rename.

The tool has an AST to represent the program and a database that has information about the several program entities and dependencies information. Bicycle Repair Man does its own parsing so it replaces the Python's parser with its own wrapper to be easier to develop the refactoring operations.

4.7.2   Rope is a Python refactoring tool written in Python, which works like a Python library. In order to make it easier to create refactoring operations Rope assumes that a Python program only has assignments and functions calls. The tool can easily get information about the assignments. However for functions calls it is necessary to have other approaches in order to obtain the necessary information.

Rope uses a Static Object Analysis which analyses the modules or scope to get information about functions. Rope only analyses the scopes when they change and it only analyses the modules when asked by the user, because this approach is time consuming.

The other approach is the Dynamic Object Analysis that starts working when a module is running. Then, Rope collects information about parameters passed to and returned from functions in order to get all the information needed. The

---

[3] https://pypi.python.org/pypi/bicyclerepair/0.7.1
[4] https://github.com/python-rope/rope

main problem is that this approach is slow while collecting information, but not while accessing the information.

Rope stores the information collected by the analysis in a database. If Rope needs the information and there is nothing on the database the Static object inference starts trying to infer the object information.

Rope uses an AST in order to store the syntax information about the programs.

4.8 Racket

Racket[5] programming language is a dialect of Lisp and a descendant of Scheme and it supports objects, types and lazy evaluation. For the Racket language the most used IDE is DrRacket. DrRacket is an IDE, that was formerly known as DrScheme. It is a simple IDE that was initially build for Scheme programming language and it is aimed at inexperienced programmers.

Regarding refactoring operations DrRacket only has one, namely the rename. It could be viewed in two ways: the first is they only implemented one refactoring operation and forgot the other ones. Or it could be viewed as they implemented a refactoring operation that is the most used one.

4.9 Language-independent Refactoring

Some refactoring operations make sense in different languages, such as the rename, the move or even the extract function. In order to use that similarity there are some tools that aim to create refactoring operations independently of the language.

4.9.1 Famix [27] is a Meta-model for Language-independent refactoring. The goal of the FAMIX is to check the preconditions of the refactoring operations supported and to analyze which changes need to be done for every supported refactoring at a language independent level.

Language independence is useful because a large part of the refactoring operations are described and analyzed on a language-independent level and similar concepts in different languages are treated in the same way. With that it is possible to reuse the analysis and reduce the language specifics to only the modifications in the source code.

Based on this meta-model it is possible to construct a refactoring engine that can do primitive refactoring operations for a few implementations such as Smalltalk and Java.

However there are some downsides to this approach which leads to an increase of the algorithms complexity. The complexity increases because the model needs to be general in order to deal with several languages.

There are difficulties in mapping the changes to the actual code, because sometimes the concepts that are generalized in the language-independent level

---

[5] http://racket-lang.org/

need to be mapped back to their language-specific. For example the invocation of methods in Java is different for the constructors. It is also difficult to abstract a language when there are apparent standards in all languages but needs language-specific interpretation such as when a name is a valid name for a class in that specific language.

Mapping back the language to the FAMIX is difficult in some languages because FAMIX meta-model does not have the concept of meta-classes or interfaces. However, the most problematic issue was with dynamic languages, because they have less information available at compile-time and that makes the dependency analysis through invocations and accesses more difficult. For example the rename method can only be done if there is no other method with the same signature.

4.9.2   JunGL [28] is a domain-specific language for refactoring. It is a hybrid of a functional language and a logic query language. JunGL's goal is to create a scripting language that allows the user to create their owns refactoring without errors.

The JunGL has a Program Graph as main data structure that is composed by nodes and edges and both nodes and edges can have a type. For example: control flow successor. This graph have all the relevant information about the program, including the ASTs, variable binding and control flow.

It defines the Program Dependence Graph using path queries. This allows the correct application of transformations that reorder statements. The language has lazy edges. The lazy edges initially are only composed by the ASTs of the program, the rest of the information will be added later to the edges as it may be necessary, like lazy evaluation but for the Program Graph. The tool can handle incomplete programs, when the branch information of the graph is non existent, because of the lazy edges used.

4.10   Conclusions

Dynamic languages, like Racket and Python are used in introductory courses across the world. However, in spite of being recognized as good languages to learn to program there are few refactoring tools for dynamic languages. Consequently making the inexperienced programmer contact with refactoring tools latter on and not while learning. Additionally refactoring tools for dynamic languages are still far away from the capabilities offered by the refactoring tools for static languages. In average, they have less refactoring operations than the refactoring tools for static languages.

The biggest difficulty of the refactoring tools for dynamic languages is the lack of information available. This happens because the dynamic languages only know type information in runtime. That makes the refactoring operations more difficult when compared with the refactoring operations for static languages, where the information is always available.

In addition, dynamic languages are very different from each other whereas the static languages are more similar. For example, Java and C# are very similar

and they have similar refactoring operations. This difference makes it a bit more difficult for the refactoring tools for dynamic languages when compared to the static one's.

Another difference between refactoring tools is the IDE support that helps managing the program's information. Usually dynamic refactoring tools do not have that support, making it harder to create refactoring operations when compared to the static refactoring tools. However, this becomes an advantage when considering the interoperability of the refactoring tools, thus making it easier to use in different text-editors or IDEs.

Even having way less refactoring operations available, when compared with static refactoring tools, the refactoring tools for dynamic languages at least have the most used refactoring operation, which is the rename.

To conclude, there is a lack of refactoring tools for dynamic languages and none of the existent ones is adequate for inexperienced users. And there are some dynamic languages such as Scheme, Racket and Python that are used to teach inexperienced programmers how to program. Without a refactoring tool for these languages, the first contact with these tools is postponed and that might explain the low adoption of refactoring tools.

5   Architecture

This section proposes a solution to solve the need of a refactoring tool designed for inexperienced users. Because it is targeted for inexperienced users it is important to have an environment that is not complex, like Eclipse, and to be for a programing language that is used in introductory courses, such as Python and Racket. Racket has an IDE, DrRacket, that is a pedagogic environment [29] and it also supports development and extension of other programming languages [30]. Recently an implementation of Python for Racket was added [31]. This addition makes it possible to have Python programs in DrRacket, and with that the pedagogical environment gains another language that is used in introductory courses around the world. Because of that DrRacket environment was chosen. Racket programming language was chosen because it is easier to create refactoring operations for Racket in the DrRacket environment. This tool will motivate the inexperienced programmers to start using tool assisted refactoring operations while they are learning.

5.1   Refactoring Operations

The refactoring tool consists in a set of refactoring operations. This section describes some of the refactoring operations desired and their use.

5.1.1   Rename is the most used refactoring operation and it is indispensable in any refactoring tool. The DrRacket environment already has a rename operation, however the rename has a bug when renaming imported functions.

This bug, as it can be seen in Figure 3 happens when the user renames an imported function, with the original rename operation of DrRacket, it renames all the imported functions from the same file and it also renames the name of the imported file. This is an incorrect refactoring because it makes the program syntactically incorrect. It should only rename the function with the selected name, not every function from that file. This situation may occur when using functions from other file, whether there is a name conflict or because it is better for the readability to rename that function name.
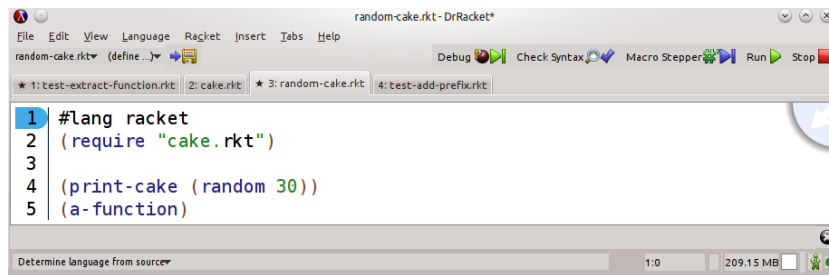
Fig. 2. Before the Rename
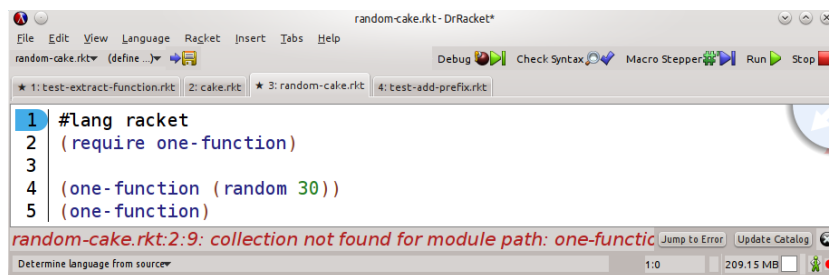
Fig. 3. Rename error on DrRacket

5.1.2 Extract-function is a simple and useful refactoring, as shown previously in Section 1, it is one of the most used refactoring operations.

Extract-function is an important refactoring for inexperienced users since those users tend to create a big function that does all the work.

By having a way to restructure the code, inexperienced programmers will be able to create programs with better structures.

5.1.3   Move expression is also one of the most used refactoring operations. This operation allows the user to move safely the expressions, by correcting all references, to the new location.

5.1.4   Add-prefix is a particular refactoring operation for Racket. When several functions are imported from several different libraries the code starts to become difficult to understand. With so many functions from different libraries it is hard to remember where a specific function came from. To solve this problem, Racket has a prefix that can be added to the imported functions. This prefix helps improving the readability of the code and, therefore, the code's quality. However, doing that manually is a tiresome task because the user has to remember and change one by one all function invocations.

The Add Prefix refactoring does all that for the user. This is also useful when the name of the functions are similar, adding a prefix makes it easier to distinguish between libraries.

5.2   Proposed Architecture

Figure 4 presents the proposed architecture of the tool. The refactoring tool uses, mainly, two important informations gathered by the compiler, the def-use-relations and the AST. A def-use-relation consists in a definition of a variable and all the uses that reach the variable without any intervening definitions. The information gathered in the AST and in the def-use-relations with some preconditions is enough to ensure the correctness of the refactoring operations. The refactoring tool uses the information generated by the compiler and some conditions to ensure that the refactoring is correct. After that it changes the program creating another version. The AST in the Racket programing language composed by a list of s-expressions, which are basically a nested list of basic data, and have the syntax information about the program and other useful information such as in which line and column a term is declared. DrRacket provides the def-use-relations which is an important help to do the refactoring operations. The def-use-relations are visually represented as arrows in DrRacket.

5.3   Validation

In order to validate this architecture some refactoring operations were implemented. This allow us to validate the architecture and have more trust in this architecture. Only the refactoring operations that use exclusively the def-use-relations were validated.

5.3.1   Extract-function The user selects the expressions that wants to extract and chooses a name for the new function as seen in **Figure 5**. Then the user pastes the result of the extraction where he thinks it's the best place for the function. The final result can be seen in **Figure 6**. The extract function refactoring could automatically paste the extracted function, however this way the user can choose the function's location.
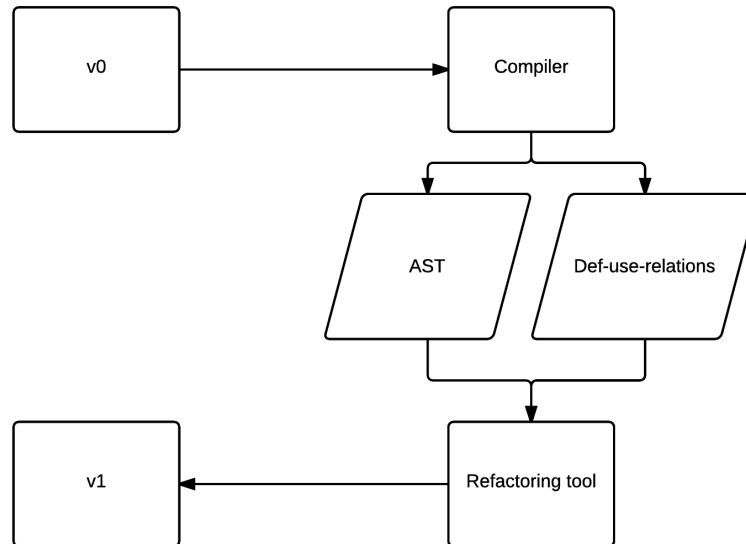
Fig. 4. The arrows determines the information flow; v0 is the initial program; The compiler produces two databases the AST and the Def-use-relations; The Refactoring Tool use the AST and the Def-use-relations produced by the compiler to generate v1, which is the refactored program
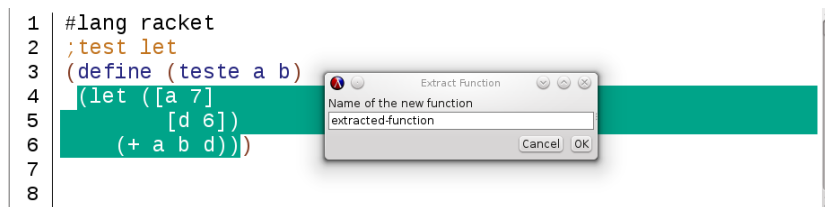

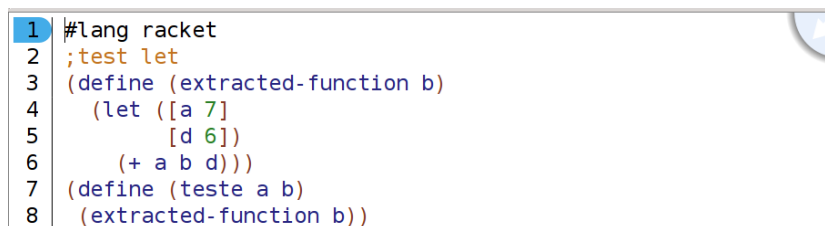**Fig. 5.** Before the Extract function

Fig. 6. After the Extract function

5.3.2   Imported renames As explained in 5.1.1 the only refactoring operation supported by DrRacket, the Rename, has bugs when it has to rename imported functions. To do the rename, the user selects the function that wants to be renamed, as seen in **Figure 2** and in this case the user chose "print-cake". Afterwards the user chooses the name that wants go give and finally it renames all the functions. The end result is shown in **Figure 7**.

Fig. 7. After the Rename

5.3.3   Add-prefix The user selects the library name that wants to add the prefix, in this case the "pict3d". In **Figure 8** shows the the arrows pointing to the functions belong to the library. Then the user selects a name for the prefix that name will be added to each function. The final result can be seen in **Figure 9**.
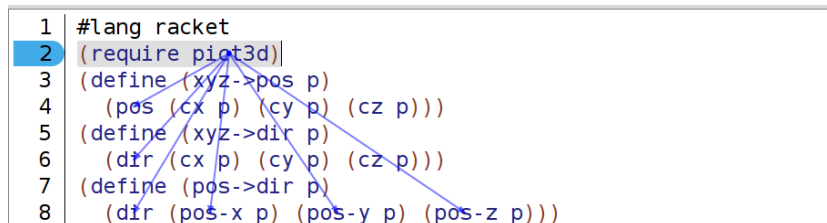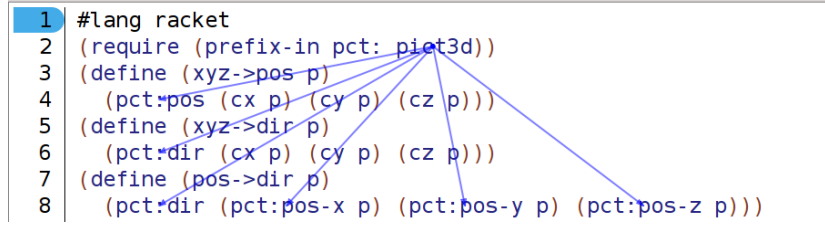

Fig. 8. Before adding the prefix

```
1  #lang racket
2  (require (prefix-in pct: piet3d))
3  (define (xyz->pos p)
4    (pct:pos (cx p) (cy p) (cz p)))
5  (define (xyz->dir p)
6    (pct:dir (cx p) (cy p) (cz p)))
7  (define (pos->dir p)
8    (pct:dir (pct:pos-x p) (pct:pos-y p) (pct:pos-z p)))
```

Fig. 9. After the Add-prefix refactoring

6   Evaluation

In order to evaluate the correctness of the refactoring operations it would be preferable to have a formal proof for each refactoring operation. However, formal proofs are hard to do and take too much time. Moreover, formal proofs that exist are usually done for theoretical languages and not for languages actually used, which might explain why even professional refactoring tools such as IntelliJ, Eclipse or NetBeans have errors. [28]

One solution to that problem is to prove correctness informally. The proof consists in having several correct Racket programs with unit tests, preferably developed with test-driven-development, then apply some refactoring operations and then run the unit tests again. This allows to test if the refactoring operations do not introduce errors in the programs neither change the programs' meaning.

To evaluate the simplicity and usability of the refactoring operations, we will test them by having users following a set of use cases and test the users' response to the tasks.

7   Conclusions

Refactoring operations are important in order to maintain the quality of the software but because they are error prone and time consuming, refactoring tools are needed. Unfortunately, there is a lack of refactoring tools adequate for inexperienced users, mainly because most of them start programming with dynamic languages which also do not have enough refactoring tools. Having a refactoring tool adequate for inexperienced users might help them to start using the refactoring tools and refactoring their programs alongside with programming.

The proposed solution uses the information given by the compiler, namely the AST and the def-use-relation. With this information it is possible to create correct refactoring operations necessary for a refactoring tool for inexperienced users.

To validate the solution some refactoring operations were already implemented, such as extract-function, add-prefix and rename. So far this architecture is working, however none of the refactoring operations implemented uses the AST and that part remains unvalidated.

In the future, more refactoring operations will be added to the work.

References

1. Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Electronic Notes in Theoretical Computer Science*, 82(3):483–499, 2003.
2. Fabrice Bourquin and Rudolf K Keller. High-impact refactoring based on architecture violations. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 149–158. IEEE, 2007.
3. David Carlson. *Eclipse Distilled*. Addison-Wesley Reading, 2005.
4. Heiko Böck. Intellij idea and the netbeans platform. In *The Definitive Guide to NetBeans Platform 7*, pages 431–437. Springer, 2011.
5. Tim Boudreau, Jesse Glick, Simeon Greene, Vaughn Spurlin, and Jack J Woehr. *NetBeans: the definitive guide.* " O'Reilly Media, Inc.", 2002.
6. Sam Guckenheimer and Juan J Perez. *Software Engineering with Microsoft Visual Studio Team System (Microsoft. NET Development Series)*. Addison-Wesley Professional, 2006.
7. Elish. A classification of refactoring methods based on software quality attributes. 2008.
8. Stephan Erb. A survey of software refactoring tools. *Student research paper*, 2010.
9. William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
10. Tom Mens and Tom Tourwé. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2004.
11. Maurizio Proietti and Alberto Pettorossi. Semantics preserving transformation rules for prolog. In *ACM SIGPLAN Notices*, volume 26, pages 274–284. ACM, 1991.
12. Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.
13. Simon Thompson and Claus Reinke. A case study in refactoring functional programs. In *In Brazilian Symposium on Programming Languages*. Citeseer, 2003.
14. Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 30–38. IEEE, 2001.
15. Eduardo Casais. The automatic reorganization of object oriented hierarchies-a case study. 1994.
16. Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *ACM SIGPLAN Notices*, volume 31, pages 235–250. ACM, 1996.
17. Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.
18. Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.
19. Romain Robbes. Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 15. IEEE Computer Society, 2007.
20. Eric Armstrong. *JBuilder 2 bible*. IDG Books Worldwide, Inc., 1998.
21. Simon Thompson. Refactoring functional programs. In *Advanced Functional Programming*, pages 331–357. Springer, 2005.
22. William G Griswold. Program restructuring as an aid to software maintenance. 1991.

23. Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Urbana*, 51:61801, 1997.
24. Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, 2010.
25. Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.
26. Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. *ACM SIGPLAN Notices*, 46(10):119–138, 2011.
27. Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154–164. IEEE, 2000.
28. Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, pages 172–181. ACM, 2006.
29. Flanagan C. Flatt M. Krishnamurthi S. & Felleisen M. Findler, R. B. DrScheme: A pedagogic programming environment for Scheme. *Programming Languages: Implementations, Logics, and Programs*, 12(2):369–388, 1997.
30. Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.
31. Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. In *7 th European Lisp Symposium*, 2014.

A Appendix

A.1 Work Scheduling

	May	Jun.	Jul.	Aug.	Sept.	Oct.	Nov.	Dec.
State of the Art	■	■						
Solution Development	■	■						
Implementation		■	■	■	■			
Tests and Evaluation					■	■		
Writing		■	■	■	■	■	■	■
Review							■	■

Table 3. Work scheduling