Refactoring Dynamic Languages

Rafael José Trindade Reia

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. David Manuel Martins de Matos

May 2016
Acknowledgments

Firstly, I would like to thank to professor António Leitão for his knowledge, support, and sharing of knowledge that were fundamental to make this Thesis possible. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

I would also like to thank to my colleagues from the Architecture and Computing group, that gave important feedback and help along the way.

A special thanks to my parents and my sister for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible.

Last but not least, to all my friends that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.
Abstract

Typically, beginner programmers do not master the style rules of the programming language they are using and, frequently, do not have yet the logical agility to avoid writing redundant code. As a result, although their programs might be correct, they can also be improved and it is important for the programmer to learn about the improvements that, without changing the meaning of the program, simplify it or transform it to follow the style rules of the language. These kinds of transformations are the realm of refactoring tools. However, these tools are typically associated with sophisticated integrated development environments (IDEs) that are excessively complex for beginners. On the other hand, there are several different programming languages being used in introductory courses to teach beginner programmers and, it is not expected that the languages used in these introductory courses will converge into one single language in the near future.

In this thesis, we present a refactoring tool designed for beginner programmers, which we made available in DrRacket, a simple and pedagogical IDE. Our tool provides several refactoring operations for the typical mistakes made by beginners and is intended to be used as part of their learning process. We also present a framework designed to simplify the creation of refactoring tools for dynamic languages, which we evaluate by creating refactoring tools for Python and Racket.

Keywords: Refactoring Tool, Pedagogy, Framework, Racket
Resumo

Tipicamente, os programadores inexperientes não dominam as regras de estilo das linguagens de programação que estão a usar e frequentemente não tem a agilidade lógica para evitar a escrita de código redundante. Consequentemente, os seus programas até podem estar correctos do ponto de vista lógico, mas podem ser melhorados. Por isso, é importante para o programador aprender o que pode fazer para melhorar o programa, sem mudar o significado do programa, transformando-o para seguir as regras de estilo da linguagem de programação usada. Estas transformações são do domínio das ferramentas de refatorização. Contudo, estas são tipicamente associadas a ambientes de desenvolvimento integrados (IDEs) sofisticados que são excessivamente complexos para programadores inexperientes. Por outro lado, como existem várias linguagens de programação a serem usadas em cursos de introdução a programação, não é expectável que conviram para uma só linguagem a ser ensinada nos vários cursos de introdução a programação.

Nesta tese, apresentamos uma ferramenta de refatorização desenhada para programadores inexperientes, que disponibilizamos no DrRacket, um IDE simples e pedagógico. A nossa ferramenta fornece várias operações de refatorização específicas para os erros típicos feitos por programadores inexperientes e é pretendido que seja usado como parte do processo de aprendizagem. Apresentamos ainda, uma framework desenhada para simplificar a criação de ferramentas de refatorização para linguagens dinâmicas, que avaliamos ao criar ferramentas de refatorização para Python, Processing e Racket.

Palavras-Chave: Ferramenta de refatorização, Pedagogia, Framework, Racket
## Contents

### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xi</td>
</tr>
</tbody>
</table>

### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xiii</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Introduction .................................................. 3
1.2 Goals .......................................................... 4
1.3 Outline ......................................................... 5

### 2 Related Work

2.1 Definitions .................................................... 7

#### 2.1.1 Refactoring classification ................................ 7
#### 2.1.2 Refactoring as a Process .................................. 7
#### 2.1.3 Refactoring Correctness .................................. 8
#### 2.1.4 Case Study - Manual Refactoring .......................... 8
#### 2.1.5 Classification of refactoring tools ......................... 8

##### 2.1.5.1 Manual Refactoring Tool .............................. 8
##### 2.1.5.2 Semi-Automated Refactoring Tool ..................... 9
##### 2.1.5.3 Automated Refactoring Tool .......................... 9
##### 2.1.5.4 Analysis .............................................. 9

#### 2.1.6 Code Clones ................................................ 10

#### 2.1.7 Clone Detection .............................................

##### 2.1.7.1 Textual Comparision .................................. 11
##### 2.1.7.2 AST Comparision ...................................... 11
##### 2.1.7.3 Other techniques ...................................... 11

2.2 Use of static refactoring tools ................................. 12

2.3 Overview of static Refactoring tools .......................... 12

2.4 Dynamic Languages ..............................................

#### 2.4.1 Scheme .................................................... 15
#### 2.4.2 Haskell .................................................... 15
#### 2.4.3 Python .................................................... 16
#### 2.4.4 Javascript ................................................ 16
#### 2.4.5 Smalltalk ............................................... 17

2.5 Language-independent Refactoring ............................. 17

#### 2.5.1 Famix .................................................... 17
#### 2.5.2 Parallel Refactoring ...................................... 18

2.6 Analysis ........................................................ 19

#### 2.6.1 Dynamic Refactoring Tools ............................... 19
List of Tables

2.1 Refactoring operations definitions .................................................. 13
2.2 Refactoring operations available by default ...................................... 14
2.3 Data Structures ............................................................................. 19

5.1 Refactoring Operations .................................................................... 45
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Motivates the refactoring move method since the MethodB1 (green sphere near the blue cluster) uses more attributes and methods from the Class A (blue objects) than it uses from its own class, the Class B (green objects)</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Parallel Refactorer architecture</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Main modules and information flow between modules. Arrows represent information flow between modules</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Before the Extract function</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>After the Extract function</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Before the Rename</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>After the Rename</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Before adding the prefix</td>
<td>29</td>
</tr>
<tr>
<td>3.7</td>
<td>After the Add-prefix refactoring</td>
<td>29</td>
</tr>
<tr>
<td>3.8</td>
<td>Nested Ifs to Cond Refactoring</td>
<td>29</td>
</tr>
<tr>
<td>3.9</td>
<td>Nested Ifs to Cond Refactoring outcome</td>
<td>30</td>
</tr>
<tr>
<td>3.10</td>
<td>Before the highlight</td>
<td>30</td>
</tr>
<tr>
<td>3.11</td>
<td>After the highlight</td>
<td>31</td>
</tr>
<tr>
<td>3.12</td>
<td>Previewing the outcome of a refactoring operation</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Framework Architecture</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Refactoring Suggestions in Python</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Refactoring Suggestions in Processing</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Preview in Python</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>Preview in Processing</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Error in Racket Refactoring Tool</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>Error in Python Refactoring Tool</td>
<td>47</td>
</tr>
<tr>
<td>5.7</td>
<td>Error in Processing Refactoring Tool</td>
<td>48</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

In order to become a proficient programmer, one needs not only to master the syntax and semantics of a programming language, but also the style rules adopted in that language and, more important, the logical rules that allow him to write simple and understandable programs. Given that beginner programmers have insufficient knowledge about all these rules, it should not be surprising that their code reveals what more knowledgeable programmers call “poor style,” or “bad smells”. As time goes by, it is usually the case that the beginner programmer learns those rules and starts producing correct code written in an adequate style. However, the learning process might take a considerable amount of time and, as a result, large amounts of poorly-written code might be produced before the end of the process. It is then important to speed up this learning process by showing, from the early learning phases, how a poorly-written fragment of code can be improved.

After learning how to write code in a good style, programmers become critics of their own former code and, whenever they have to work with it again, they are tempted to take advantage of the opportunity to restructure it so that it conforms to the style rules and becomes easier to understand. However, in most cases, these modifications are done without complete knowledge of the requirements and constraints that were considered when the code was originally written and, as result, there is a serious risk that the modifications might introduce bugs. It is thus important to help the programmer in this task, so that he can be confident that the code improvements he anticipates are effectively applicable and will not change the meaning of the program. This has been the main goal of code refactoring.

Code refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Nowadays, any sophisticated IDE includes an assortment of refactoring tools, e.g., renaming variables or methods, for extracting methods, or for moving methods along a class hierarchy. It is important to note, however, that these IDEs were designed for advanced programmers, and that the provided refactorings require a level of code sophistication that is not present in the programs written by beginners. This may make the refactoring tools inaccessible to beginners.

In addition, since the choice of the language to teach in an introductory course often differs from faculty to faculty and has evolved throughout the years, there are several different programming languages being used to teach beginner programmers. Furthermore, even the requirements used to decide the language in the introductory courses have changed. Considering all these factors, it is not expected that the languages used in these introductory courses will converge into one language in the near future.

In order to correctly refactor a program, refactoring tools often require the same kind of information
about the program. Therefore they often have similar architecture. This similarity between refactoring tools creates the possibility of reusing some modules instead of creating every module from scratch, thus making the development of refactoring tools development faster. Grouping all those modules in a refactoring framework would make them more easily accessible. Moreover, it is also possible to share features for the refactoring tools thus implemented, improving the support available for the users of those refactoring tools.

In this thesis, we present a tool that was designed to address the above problems. In particular, our tool (1) is usable from a pedagogical IDE designed for beginners [2, 3], (2) is capable of analyzing the programmer’s code and inform him of the presence of the typical mistakes made by beginners, and finally, (3) can apply refactoring rules that restructure the program without changing its semantics.

We also present a framework designed for creating refactoring tools for dynamic languages. This Framework (4) helps the developer to create a new refactoring tool by providing functions to support the refactoring tool, (5) is capable of sharing the features once created for one of the refactoring tools, and finally (6) provides automatic tests to maintain the refactoring tools created.

To evaluate our proposal, we implemented a refactoring tool in DrRacket, a pedagogical IDE [4, 5] used in schools around the world to teach basic programming concepts and techniques. Currently, DrRacket has only one simple refactoring operation which allows renaming a variable. Our work significantly extends the set of refactoring operations available in DrRacket and promotes their use as part of the learning process. We also implemented a framework for creating refactoring tools in DrRacket which already have implemented several languages besides Racket, such as Python and Processing. Our work significantly simplifies the creation of new refactoring tools and provides useful features to the developer of the refactoring tool.

1.2 Goals

We propose a Refactoring Tool aimed at inexperienced programmers, which fulfills the following goals:

- Designed for Beginners - we implemented a set of refactoring operations useful for beginner programmers that correct typical errors, helping those users to create better programs.

- Simple Feedback - our goal is to provide enough feedback to the user before and during the refactoring operation, therefore we implemented a set of features to aid the beginner programmer such as previewing the refactoring outcome and identifying possible refactoring operations.

- Correctness - since we are building a refactoring tool, we need to ensure that each refactoring operation do not change the meaning of the program.

Since we detect that there are very few refactoring tools for dynamic languages we also propose a Framework for creating refactoring tools for dynamic languages, which fulfills the following goals:

- Simple - we designed the framework to simplify the creation of a refactoring tool, where we provide functions to access core aspects of the refactoring tool, such as the AST (Abstract Syntax Tree), a code-walker, and a printing function.

- Re-usability - we designed our framework to support the reuse of features developed for a refactoring tool into all the remaining refactoring tools created with this framework.

- Maintainability - we designed a tool that automatically applies tests to the refactoring tools available helping the user to maintain the complex program that is a refactoring tool.

1www.processing.org
1.3 Outline

Chapter 2 presents some definitions concerning refactoring, some related refactoring tools for dynamic typed languages, and language independent refactoring tools. Chapter 3 describes the refactoring tool for Racket created, from the architecture, to some of the refactoring operations, the refactoring tool features, and some examples. Chapter 4 describes the framework for creating refactoring tools, describing its architecture, the features, and some examples of the different refactoring tools created. Chapter 5 presents the evaluation of the refactoring tool and then the evaluation of the framework. Finally, chapter 6 presents the main conclusions of our work, describing additional work to be explored in the future.
Chapter 2

Related Work

This chapter presents some definitions concerning refactoring tools, then it presents the use of the refactoring tools and an overview of static refactoring tools. Afterwards, it presents refactoring tools for dynamic languages such as Scheme, Haskell, JavaScript, Python, and Racket. In the end, it presents language-independent refactoring tools. Finally, it has a conclusion about the related work.

2.1 Definitions

This section presents some definitions regarding refactoring activities.

2.1.1 Refactoring classification

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, there is the combinational refactoring which 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 do that, 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, re-usability or testability [6]. However, that is outside the scope of this thesis, it will not be further detailed.

2.1.2 Refactoring as a Process

Refactoring can be done in two ways. It can be done as a part of the program development and be constantly performed. Or it can be done a separate activity and performed in bulk. Regardless of how it is done, refactoring can always be decomposed in different activities [7]:

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

Refactoring must preserve the program’s behavior in order to be correct and preserving the input-output behavior is sufficient to be correct [8]. However, for programs that have other constraints, such as performance, preserving the input-output behavior might be 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 and modifying the execution time modifies the program’s 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 [9]. 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 preserves the program semantics [10]. But for more complex languages, such as C++, with formal semantics that 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 [11].

2.1.4 Case Study - Manual Refactoring

One way to learn how the users manually refactor is to do it while taking notes. The case study [12] by Thompson, Simon, and Claus Reinke consists in the authors 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 prove procedures for first order logic or solve satisfiability of finite sets.

In order to better understand what constitutes 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 better reflect what the functions did. Then, they replaced explicit recursion by calls to higher order functions and they renamed some variables and functions. In the end, they generalized some functions and modified the representation type because it was becoming too complex to keep 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.

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.

2.1.5 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 cannot 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.

2.1.5.1 Manual Refactoring Tool

A refactoring tool that only applies the refactoring operations selected by the user is classified as a manual refactoring tool. Manual refactoring tool is the most common type of refactoring tool. There
are several examples of this tools such as Eclipse\footnote{help.eclipse.org/luna/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm}, IntelliJ\footnote{https://www.jetbrains.com/idea/features/refactoring.html} that are focused on static languages and for dynamic languages there are Bicycle Repair Man\footnote{https://pypi.python.org/pypi/bicyclerepair/0.7.1} and Rope\footnote{13, p. 109}.

2.1.5.2 Semi-Automated Refactoring Tool

A refactoring tool that suggests refactoring opportunities to the user and applies the refactoring operations that the user selected is classified as a semi-automated refactoring tool. 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. A prototype\footnote{14} was created as proof of concept and it uses the metrics to identify where the code that should be refactored. The tool takes into account the bad smell, which is a human intuition in which a specific code should be refactored, of a code to suggest a refactoring. An example of a bad smell that triggers 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\ref{fig:example}.

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 cohesion rule. There are some refactoring operations that are related to this rule, such as, move attribute, extract class, inline 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.

2.1.5.3 Automated Refactoring Tool

A refactoring tool that automatically applies the refactoring opportunities that the tool detects is classified as an automated refactoring tool. 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 a 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 have some restrictions because users do not know what the refactoring tools are doing. Casais\footnote{15} or Moore\footnote{16} are good examples of these refactoring tools.

2.1.5.4 Analysis

Having the Automated Refactoring tool for inexperienced users is not what is intended. Automatically transforming the program will create a new program that the user might not comprehend, especially if the user is inexperienced.

The more common approach is the Manual Refactoring tool that applies exactly what the user wants to do. This type of tool does not automatically detect refactoring opportunities, but it is faster and safer than doing the refactoring operation without any support.
Figure 2.1: Motivates the refactoring move method since the MethodB1 (green sphere near the blue cluster) uses more attributes and methods from the Class A (blue objects) then it uses from its own class, the Class B (green objects)

Semi-automated tools with the suggestions would be an advantage to inexperienced users that are still learning what refactoring operations exists. That way, users would learn new refactoring operations and have programs with better quality, since the detection will alert them of a possible refactoring that the user might not know. However, detecting refactoring opportunities is highly dependent on the application domain, which invalidates this type of tools since they are not meant for one type of application only.

The best suited approach is the semi-automated one, since it only applies the refactoring operations the user decides to but it can be used to detect possible refactoring operations that a less experienced user would not detect.

2.1.6 Code Clones

A code clone is defined as two fragments of code that are similar enough according to a given definition of similarity. Different definitions of similarity allow different types of clones.

In general, a clone can be categorized in three types:
The detection of duplicated code can be used to detect possible refactoring operations, e.g. if a list of expressions should be extracted into a function or not. Detection of duplicated code is also used in the refactoring operation that compares a recently extracted function with the rest of the program to detect if there is duplicated code that could be deleted and add a function call to the recently extracted function.

2.1.7 Clone Detection

Clone detection is an active field of research and there are several clone detection approaches and it can also be used in some refactoring operations such as the wide-scope-replacement which is usually used after an extract function refactoring operation and replaces all the function definitions by the function calls. We focus on Textual and AST comparisons since these are the resources typically used by refactoring tools.

2.1.7.1 Textual Comparision

One of the simplest approaches to clone detection is to compare whole lines to each other textually [17]. In order to increase performance, lines are partitioned using a hash function for strings and only lines that are in the same partition are compared. Consecutive lines can be summarized to larger clone sequences automatically.

It is also possible to compare lines by using token sequences instead of text comparison by comparing the token sequences of lines efficiently through a suffix tree. First, each other token sequence for a whole line is summarized by a functor that abstracts from concrete values of identifiers and literals [18].

2.1.7.2 AST Comparision

An AST that represents the abstract syntactic structure of the program. Therefore, it is possible to detect clones comparing the AST of two programs. It is done by comparing the partition subtrees of the program’s AST, based on a hash function and then compare subtrees in the same partition through tree matching, using dynamic programming to find differences between two versions of the same file [19].

2.1.7.3 Other techniques

There are other techniques to clone detection, like using control and data flow dependencies of a function, which may be represented by a PDG (Program Dependence Graph) and, therefore, clones may be identified as isomorphic subgraphs [20] However since this problem is NP-hard approximated solutions are used.

Another approach consists metric comparison [21] by collecting different metrics for code fragments and then comparing these metric vectors instead of comparing code directly.

Latent semantic indexing, an information retrieval technique, can also be used to identify fragments in which similar names occur [22].

It is also possible to combine syntactic and semantic techniques through a combination of comparison functions [23] that compare various aspects such as, similar call subgraphs, commutative operators,
user-defined equivalences and, transformations into canonical syntactic forms. Each comparison function returns an evidence that is summarized in an evidence-factor model resulting a clone likelihood.

It is even possible to cast the search for similar fragments as a data mining problem [24]. Statement sequences are summarized to item sets, an adapted data mining algorithm searches for frequent item sets.

2.2 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 users refactor was gathered by collecting some data sets [25].

The User data set was collected by Murphy and colleagues [26] 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 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 [27] 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. This variation is explained by the more often used of other refactoring operations rather then the rename.

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 Toolsmiths data set, only 2 weeks of the year 2006 did not have any refactoring operation and, on average, had 30 refactoring operations per week. In 2007, every week had refactoring activities and the average was 47 refactoring operations in a week.

Besides refactoring operations being frequent, the refactoring tools are underused [25].

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, the group are more familiarized with the refactoring tools since they maintain the Eclipse’s refactoring tools.

2.3 Overview of static Refactoring tools

Table 2.1 contains a brief explanation of the refactoring operations, that were taken from Eclipse and IntelliJ.

Table 2.2 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
<table>
<thead>
<tr>
<th>Refactoring name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename</td>
<td>Renames the selected element and corrects all references.</td>
</tr>
<tr>
<td>Move</td>
<td>Moves the selected elements and corrects all references.</td>
</tr>
<tr>
<td>Change signature</td>
<td>Change parameter names, types and updates all references.</td>
</tr>
<tr>
<td>Extract method</td>
<td>Creates a new method with the statements or expression selected and replaces it with a reference to the new method.</td>
</tr>
<tr>
<td>Extract local variable</td>
<td>Creates a new variable assigned to the expression selected and replaces it with a reference to the new variable.</td>
</tr>
<tr>
<td>Extract constant</td>
<td>Creates a static final field from the selected expression.</td>
</tr>
<tr>
<td>Inline</td>
<td>Inline local variables, methods or constants.</td>
</tr>
<tr>
<td>To nested</td>
<td>Converts an anonymous inner class to a member class.</td>
</tr>
<tr>
<td>Move type to new file</td>
<td>Creates a new compilation unit and updates all references.</td>
</tr>
<tr>
<td>Variable to field</td>
<td>Turns a local variable into a field.</td>
</tr>
<tr>
<td>Extract superclass</td>
<td>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.</td>
</tr>
<tr>
<td>Extract interface</td>
<td>Creates a new interface and makes the class implement it.</td>
</tr>
<tr>
<td>Change to Supertype</td>
<td>Replaces, where it is possible, all occurrences of a type with one of its supertypes.</td>
</tr>
<tr>
<td>Push down</td>
<td>Moves a set of methods and fields from a class to its subclasses.</td>
</tr>
<tr>
<td>Pull up</td>
<td>Moves a field or method to a superclass, if it is a method, declares the method as abstract in the superclass.</td>
</tr>
<tr>
<td>Extract class</td>
<td>Replaces a set of fields with new container object.</td>
</tr>
<tr>
<td>Introduce parameter</td>
<td>Replaces an expression with a reference to a new method parameter and updates all callers of the method.</td>
</tr>
<tr>
<td>Introduce indirection</td>
<td>Creates an indirection method delegating to the selected method.</td>
</tr>
<tr>
<td>Introduce factory</td>
<td>Creates a new factory method, which calls a selected constructor and returns the created object.</td>
</tr>
<tr>
<td>Encapsulate field</td>
<td>Replaces all references to a field with getter and setter methods.</td>
</tr>
<tr>
<td>Generalize type</td>
<td>Allows the user to choose a supertype of the selected reference.</td>
</tr>
<tr>
<td>Type Migration</td>
<td>Change a member type and data flow dependent type entries.</td>
</tr>
<tr>
<td>Remove Middleman</td>
<td>Replaces all calls to delegating methods with the equivalent calls.</td>
</tr>
<tr>
<td>Wrap Return Value</td>
<td>Creates a wrapper class that includes the current return value.</td>
</tr>
<tr>
<td>Safe Delete</td>
<td>Finds all the usages or, simply delete if no usages found.</td>
</tr>
<tr>
<td>Replace duplicates</td>
<td>Finds all the places in the current file where the selected method code is fully repeated and change to corresponding method calls.</td>
</tr>
<tr>
<td>Static to instance</td>
<td>Converts a static method into an instance method with an initial method call argument being a prototype of newly created instance method call qualifier.</td>
</tr>
<tr>
<td>Make Method Static</td>
<td>Converts a non-static method into a static one.</td>
</tr>
<tr>
<td>Change to Interface</td>
<td>Used after using Extract an Interface it searches for all places where the interface can be used instead of the original class.</td>
</tr>
<tr>
<td>Inheritance to delegation</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 2.2: Refactoring operations available by default

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Visual Studio</th>
<th>Eclipse</th>
<th>CDT</th>
<th>IntelliJ</th>
<th>NetBeans</th>
<th>JBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Move</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Change method signature</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Extract method</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Extract local variable</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Extract constant</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inline</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>To nested</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move type to new file</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable to field</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extract superclass</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Extract interface</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change to supertype</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push down</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pull up</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extract class</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduce parameter</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduce indirection</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduce factory</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encapsulate field</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generalize declared type</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Migration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove Middleman</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wrap Return Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safe Delete</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Replace Method duplicates</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Static to instance method</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Make Method Static</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Change to interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Inheritance to delegation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
and the refactoring operations have similar purposes. The most used refactoring operations are marked bold.

Table 2.2 only lists the refactoring operations that each IDE has by default, in order to have a simpler 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\[28\] 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 easy-to-install plug-ins that cover the more important refactoring operations.

2.4 Dynamic Languages

The large majority of these tools were designed to deal with large statically-typed programming languages such as Java or C++ and are integrated in the complex IDEs typically used for the development of complex software projects, such as Eclipse or Visual Studio.

On the other hand, it is a common practice to start teaching beginner programmers using dynamically-typed programming languages, such as Scheme, Python, or Ruby, using simple IDEs. As a result, our focus was on the dynamic programming languages which are used in introductory courses and, particularly, those that promote a functional programming paradigm.

In the next sections, we present an overview of the refactoring tools that were developed for the languages used in introductory programming courses.

2.4.1 Scheme

In its now classical work \[29\], Griswold presented a refactoring tool for Scheme that uses two different kinds of information, namely, an AST and a PDG.

The AST represents the abstract syntactic structure of the program, while the PDG explicitly represents the key relationship of dependence between operations in the program. The graph vertices represent program operations and the edges represent the flow of data and control between operations. However, the PDG only has dependency information of the program and relying only in this information to represent the program could create problems. For example, two semantically unrelated statements can be placed arbitrarily with respect to each other. Using the AST as the main representation of the program ensures that statements are not arbitrarily reordered, allowing the PDG to be used to prove that transformations preserve the meaning and as a quick way to retrieve needed dependence information. Additionally, contours are used with the PDG to provide scope information, which is non-existent in the PDG, and to help reason about transformations in the PDG. With these structures, it is possible to have a single formalism to reason effectively about flow dependencies and scope structure.

2.4.2 Haskell

Although Haskell is a statically-typed language, its use in introductory programming justifies a brief discussion. HaRe \[30\] is a refactoring tool for Haskell that integrates with Emacs\[4\] and Vim\[5\]. The HaRe system uses an AST 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. It retrieves scope information from the AST, allowing it to have refactoring operations that require binding information of

\[4\]https://www.gnu.org/software/emacs/
\[5\]http://www.vim.org/
variables. The system also allows the users to design their own refactoring operations using the HaRe API.

2.4.3 Python

Rope[13] 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 function calls. Thus, by limiting the complexity of the language it reduces the complexity of the refactoring tool.

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

Rope also uses Dynamic Object Analysis, requiring running the program in order to work. Dynamic Object Analysis gathers type information and parameters passed to and returned from functions. It 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 it. This approach makes the program run much slower. Thus, it is only active when the user allows it. Rope uses an AST in order to store the syntax information about the programs.

Bicycle Repair Man is another refactoring tool for Python and is written in Python itself. This refactoring tool can be added to IDEs and editors, such as Emacs, Vi, Eclipse, and Sublime Text. It attempts 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 uses an AST to represent the program and a database to store information about several program entities and their dependencies.

Pycharm Educational Edition, or Pycharm Edu, is an IDE for Python created by JetBrains, the creator of IntelliJ. The IDE was specially designed for educational purposes, for programmers with little or no previous coding experience. Pycharm Edu is a simpler version of Pycharm community which is the free Python IDE created by JetBrains. It is very similar to their complete IDEs and it has interesting features such as code completion and integration with version control tools. However, it has a simpler interface than Pycharm Community and other IDEs such as Eclipse or Visual Studio.

Pycharm Edu integrates a Python tutorial and supports teachers that want to create tasks/tutorials for the students. However, the refactoring tool did not received the same care as the IDE itself. The refactoring operations are exactly the same as the Pycharm Community IDE which were made for more advanced users. Therefore, it does not provide specific refactoring operations to beginners. The embedded refactoring tool uses the AST and the dependencies between the definition and the use of variables, known as def-use relations.

2.4.4 Javascript

There are few refactoring tools for JavaScript but there is a framework for refactoring JavaScript programs [31]. In order to guarantee the correctness of the refactoring operation, the framework uses pre-conditions, expressed as query analyses provided by pointer analysis. Queries to the pointer analysis produce over-approximations of sets in a safe way to have correct refactoring operations. For example,

[^6]: https://pypi.python.org/pypi/bicyclerepair/0.7.1
[^7]: https://www.jetbrains.com/pycharm-edu/
[^8]: https://www.jetbrains.com/pycharm/
while doing a rename operation, it over-approximates the set of expressions that must be modified when a property is renamed in a safe manner.

To prove the concept, three refactoring operations were implemented, namely rename, encapsulate property, and extract module. By using over-approximations, it is possible to be sure when a refactoring operation is valid. However, this approach has the disadvantage of not applying every possible refactoring operation, because the refactoring operations for which the framework cannot guarantee behavior preservation are prevented. The wrongly prevented operations accounts for 6.2% of all rejections.

2.4.5 Smalltalk

The Refactoring Browser [32] is a refactoring tool for Smalltalk programs whose goal was to make refactoring known and widely-accepted. 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 done by the programmer using the refactoring browser needed to be done faster than by hand.

The Refactoring Browser is a semi-automated refactoring tool since it 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.

2.5 Language-independent Refactoring

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

2.5.1 Famix

Famix [33] is a Meta-model for language-independent refactoring with support for Java, C++, and ADA. The goal of 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, such as Add Method, Remove Method, Add Attribute, and Remove Attribute.

However, there are some downsides to this approach which leads to an increase of the algorithm’s 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 some of the concepts that are generalized in the language-independent level need to be mapped back to their language-specific level with some restrictions. For example, the invocation of methods in Java is different for the constructors. It is also difficult to abstract a language when there are general rules in all languages that need language-specific interpretation, such as when a name is a valid name for a class in that specific language.

Mapping back the language to Famix is difficult in some languages since the Famix meta-model does not have the concept of meta-classes or interfaces. Therefore the refactoring operations have some additional requirements for the refactoring operations. For example, in the Pull Up Method there is a specific requirement for Java which defines a method as not being a constructor. 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.

2.5.2 Parallel Refactoring

Refactoring has been mainly applied to sequential programs in detriment of parallel ones, and to introduce and help fine tune parallelism parallel refactoring, a language independent parallel refactoring framework for C/C++ and Erlang [34] was proposed. This tool uses refactoring combined with parallel design patterns that will introduce parallelism into the programs to help the users creating better parallel programs.

![Parallel Refactorer architecture](image)

Figure 2.2: Parallel Refactorer architecture

Figure 2.5.2 describes an overview of the architecture of the system. It parses the user’s code, in
Table 2.3: Data Structures

<table>
<thead>
<tr>
<th>Name</th>
<th>AST</th>
<th>PDG</th>
<th>Database</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Griswold</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haskell</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rope</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bicycle</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pycharm Edu</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Javascript</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

C/C++ or Erlang, into an AST which is then translated into a unified intermediate language. This intermediate language is what allows to have a language independent refactoring tool. It is then refactored and then pretty printed into the program’s source language.

2.6 Analysis

In this section, we do an analysis of the refactoring tools presented above and then we compare them with what we did in our solution.

2.6.1 Dynamic Refactoring Tools

Table 2.3 summarizes the data structures of the analyzed refactoring tools. It is clear that the AST of a program is an essential part of the refactoring tool information with every refactoring tool having an AST to represent the program. Regarding the PDG and Database it contains mainly information about the def-use-relation of the program. The PDG has also control flow information of the program.

Some tools, like the one build by Griswold, focus on the correctness of the refactoring operations and therefore need more information about the program, such as the information provided by the PDG. Others, focus on offering refactoring operations for professional or advanced users. However, the goal of our refactoring tool is to provide refactoring operations designed for beginners. Therefore, we are not interested in proving formally correct the refactoring operations or provide refactoring operations only used in advanced and complex use cases.

We intend to have simple, useful, and correct refactoring operations to correct the typical mistakes made by beginners. With this, we exclude from the refactoring tool scope, macros, classes, and other complex language structures not often used by beginners.

2.6.2 Language-independent refactoring tools

Both Famix and the Parallel Language-independent refactoring tool use the same model of a meta-language in order to have a language-independent refactoring. This meta-language model allows the possibility of language-independent refactoring operations, by mapping the original language into a meta-language that can represent all of the implemented languages. However, both have some difficulties to seamlessly map those languages to the meta-language. Therefore, it led to some language specific cases which adds more complexity to the refactoring tool.

Our framework uses the languages that are implemented for DrRacket and those implementations allow the possibility of using a meta-language. Thus, allowing the possibility of language-independent refactoring operations. However, since we intend to have a framework that simplifies the creation of refactoring tools we do not focus on having only language-independent refactoring operations. Since, relying only in a meta-language that for more complex refactoring operations brings more complexity
to the refactoring tool it is not very productive. This led to a model that allows meta-programming as a resource to reduce the implementation of the simpler refactoring operations combined with specific refactoring operations for the less simple refactoring operations.

2.7 Conclusions

Dynamic languages, such as Racket or Python are often used in introductory courses across the world. However, in spite of being recognized as good languages to learn, there are not any refactoring tool targeted to beginner programmers. Consequently, missing a change to speed up the learning process by showing to the beginner programmers, while they are still learning, how a poorly-written fragment of code can be improved.

Even if we considered the existing refactoring tools for dynamic languages as suitable for beginners, they are and probably always be far away from the capabilities offered by the refactoring tools for static languages.

One difference between refactoring tools is in the IDE support that helps managing the program’s information, since dynamic languages only know type information at runtime, making it more difficult for the IDE. In addition, dynamic refactoring tools do not have the same support as the static counterparts, making it harder to develop refactoring operations when compared to the static refactoring tools. However, the lack of support provided from the IDE can make the refactoring tool more independent of the IDE, thus more portable to other text-editors or IDEs.

Even having less and simpler 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.

Creating a refactoring tool is not a trivial job, especially if the support given by the text-editors or IDEs is small or non-existent. In addition, there are several languages used in introductory courses that do not have a refactoring tool designed for beginner programmers. Thus, having a framework for creating refactoring tools for dynamic languages, which helps the developers to create refactoring tools, is an important step to provide beginner programmers better support. Such framework would also simplify the re-usability of features created between tools improving the tools created within the framework. Therefore, simplifying the creation of refactoring tools dynamic languages which are often the languages used by beginner programmers.

To conclude, there is a lack of refactoring tools for dynamic languages and the existent ones are not adequate for beginner programmers. There are some dynamic languages such as Scheme, Racket, and Python that are used to teach beginner programmers how to program. Without a refactoring tool targeted to beginners, the first contact with the refactoring tools tools is postponed slowing down the learning process.
Chapter 3

Refactoring Tool

This chapter presents the refactoring tool developed for beginner programmers to help them safely improve their code.

We start by presenting the architecture of the refactoring tool, then we present some of the refactoring operations implemented, then some features available and we end this chapter with some examples of the refactoring tool.

3.1 Architecture

In this section, we present the architecture of our refactoring tool, developed for the Racket programming language and, more specifically, for the DrRacket IDE.

Racket is a language designed to support meta-programming and, in fact, most of the syntax forms of the language are macro-expanded into combinations of simpler forms. This has the important consequence that programs can be analyzed either in their original form or in their expanded form.

In order to create correct refactoring operations, the refactoring tool uses two sources of information, the def-use relations and the AST of the program. The def-use relations represent the links between definition of an identifier and its usage. In the DrRacket IDE, these relations are visually represented as arrows that point from a definition to its use. The opposite relation, the use-def relation, is also visually represented as an arrow from the use of an identifier to its definition. The AST is the abstract syntax tree of the program which, in the case of the Racket language, is represented by a list of syntax-objects.

Figure 3.1 summarizes the workflow of the refactoring tool, where the Reader produces the non ex-
panded AST of the program while the Expander expands the AST produced by the Reader. In order to produce the def-use relations, it is necessary to use the expanded AST produced by the Expander because it has the correct dependency information. The Transformer uses the Code Walker to parse the ASTs and the information of the def-use relations to correctly perform the refactoring operations. Then it goes to the Writing module to produce the output in DrRacket's definitions pane.

3.1.1 Syntax Expressions

The syntax-object list represents the AST, which provides information about the structure of the program. The syntax-object list is already being produced and used by the Racket language and, in DrRacket, in order to provide error information to the user. DrRacket already provides functions which computes the program's syntax-object list and uses some of those functions in the Background Check Syntax and in the Check Syntax button callback.

3.1.1.1 Syntax Expression tree forms

DrRacket provides functions to compute the syntax-object list in two different formats. One format is the expanded program, which computes the program with all the macros expanded. The other format is the non-expanded program and computes the program with the macros unexpanded.

The expanded program has the macros expanded and the identifier information correctly computed. However, it is harder to extract the relevant information when compared with the non-expanded program.

For example, the following program is represented in the expanded form, and in the non-expanded form.

Listing 3.1: Original Code

(and alpha beta)

Listing 3.2: Expanded program

#<syntax:2:0
 (#%app call-with-values
 (lambda ()
   (if alpha beta (quote #f))
   print-values)>

Listing 3.3: Non-expanded program

#<syntax:2:0 (and alpha beta)>

Note that the expanded program transforms the and, or, when, and unless forms into ifs which makes refactoring operations harder to implement.

Racket adds internal representation information to the expanded program which, for most refactoring operations, is not necessary. In addition, the expanded program has a format that is likely to change in the future. Racket is an evolving language and the expanded form is a low-level and internal form of representation of the program. However, the expanded program has important information regarding the binding information that is not available in the non-expanded form, and this information might be useful, e.g., to detect if two identifiers refer to the same binding. Additionally, we do not consider macro definitions as part of the code that needs to be refactored, since the refactoring tool is targeted at unexperienced programmers and these programmers typically do not define macros.
Taking the previous discussion into consideration, it becomes clear that it is desirable to use the non-expanded form for the refactoring operations whenever possible and use the expanded form only when needed.

3.1.2 Def-use relations

Def-use relations hold important information needed in order to produce correct refactoring operations. They can be used to check whether there will be a duplicated name or to compute the arguments of a function that is going to be extracted.

Def-use-relations are computed by the compiler that runs in the background. However, they are only computed when a program is syntactically correct.

3.1.3 Code-walker

The code-walker is used to parse the syntax tree represented by a syntax element that is a list of syntax-objects in Racket. A syntax-object can contain either a symbol, a syntax-pair, a datum (number, boolean or string), or an empty list. While a syntax-pair is a pair containing a syntax-object as its first element and either a syntax pair, a syntax element or an empty list as the second argument. Each syntax-object has information about the line where they are defined and this information is used to search for the correct elements.

Most of the time, the code-walker is used to search for a specific syntax element and the location information contained in the syntax-object is used to skip the syntax blocks that appear before the syntax element wanted in the first place.

The Code-walker is a core part of the refactoring tool, ensuring that the selected syntax is correctly fed to the refactoring operations.

3.1.4 Pretty-printer

Producing correct output is an important part of the refactoring tool. It is necessary to be careful to produce indented code and we decided to use a pretty-printer that is already available in the Racket language. However, it should be noted that this pretty-printer does not follow some of the Racket style conventions, such as cond clauses surrounded by square brackets. This is not considered a problem because Racket supports both representations. One possible solution is to use a different pretty-printer in order to keep language conventions.

3.1.5 Comments preservation

Preserving the comment information after a refactoring transformation is an important task of the refactoring tool. If the comment in a specific place of the program changes its location, thus affecting a different part of the program, it could confuse the programmer. However, comment preservation is not implemented yet, making it a limitation of this prototype.

One possible solution is to modify the syntax reader and add a comment node to the AST. While the new node will not be used during refactoring transformations it is used during the output part of the refactoring operation, preserving the comment with the correct syntax expression.

3.1.6 Syntax-parse

The syntax-parse function provided by Racket is very useful for the refactoring operations. It provides a wide range of options to help matching the correct syntax, using backtracking to allow several
rules to be matched in the same syntax parser, helping to create more sophisticated rules.

3.2 Refactoring operations

In this section we explain some of the more relevant refactoring operations and some limitations of the refactoring tool. The complete list of refactoring operations is available in Appendix A.

3.2.1 Semantic problems

There are some well-known semantic problems that might occur after doing a refactoring operation. One of them occurs in the refactoring operation that removes redundant ands in numeric comparisons. Although rarely known by beginner programmers, in Racket, numeric comparisons support more than two arguments, as in \((< 0 \ ?x \ 9)\), meaning the same as \((\text{and} \ (< 0 \ ?x) \ (< \ ?x \ 9))\), where, we use the notation \(?x\) to represent an expression. Thus, it is natural to think about a refactoring operation that eliminates the \(\text{and}\). However, when the \(?x\) expression somehow produces side-effects, the refactoring operation will change the meaning of the program.

Despite this problem, we support this refactoring operation because, in the vast majority of the cases, there are no side-effects being done in the middle of numerical comparisons, it is not often to have the same argument repeated in a comparison, and with the short-circuit evaluation there is no guarantee that the side effect will occur twice.

Another example of a semantic problem occurs when refactoring the following \(\text{if}\) expression.

Listing 3.4: Code sample

\[
(\text{if} \ ?x
\begin{align*}
& (\text{begin} \ ?y \ ...)
& & \#f)
\end{align*}
\]

There are two different refactoring transformations possible:

Listing 3.5: Refactoring option 1

\[
(\text{when} \ ?x
\begin{align*}
& ?y \ ...
\end{align*}
\]

Listing 3.6: Refactoring option 2

\[
(\text{and} \ ?x \ (\text{begin} \ ?y \ ...))
\]

Note that the first refactoring option changes the meaning of the program, because if the test expression, in this case \(?x\), is false, the result of the \(\text{when}\) expression is \#<void>. However, the programmer may still want to choose the first refactoring option if the return value when the \(?x\) is false is not important.

3.2.2 Extract Function

Extract function is an important refactoring operation that every refactoring tool should have. In order to extract a function, it is necessary to compute the arguments needed for the correct use of the function. While giving the name to a function seems quite straightforward, it is necessary to check for name duplication in order to produce a correct refactoring as having two identifiers with the same name in the same scope produces an incorrect program. After the previous checks, it is straightforward to compute the function body and replace the original expression with the function call.
However, the refactoring raises the problem of where should the function be extracted to. A function cannot be defined inside an expression, but it can be defined at the top-level or at any other level that is accessible from the current level.

As an example, consider the following program:

```
;; top-level
(define (level-0)
  (define (level-1)
    (define (level-2)
      (+ 1 2)
      (level-2))
    (level-2))
  (level-1))
```

When extracting \((+ 1 2)\) to a function where should it be defined? Top-level, Level-0, level-1, or in the current level, the level-2? The fact is that is extremely difficult to know the answer to this question because it depends on what the user is doing and the user intent. Accordingly, we decided that the best solution is to let the user decide where to define the function.

### 3.2.2.1 Computing the arguments

In order to compute the function call arguments, we have to know in which scope the variables are being defined, in other words, if the variables are defined inside or outside the extracted function. The variables defined outside the function to be extracted are candidates to be the arguments of that function. However, imported variables, whether from the language base or from other libraries, do not have to be passed as arguments. To solve this problem, we considered two possible solutions:

- Def-use relations + Text information
- Def-use relations + AST

The first approach is simpler to implement and more direct than the second one. However, it is less tolerant to future changes and to errors. The second one combines the def-use relations information with the syntax information to check whether it is imported from the language or from other library.

We choose the second approach in order to provide a more stable solution to correctly compute the arguments of the new function.

### 3.2.3 Let to Define Function

A `let` expression is very similar to a function, which may lead the user to mistakenly use one instead of the other. Therefore, we decided to provide a refactoring operation that would make such transition simpler.

There are several `let` forms, but since we want to explore the similarity between `let` and function, we are going to focus in the ones that are more similar to a function, namely `let` and `named let`.

There are some differences between them: `named let` can be directly mapped to a named function, using `define` keyword, whereas `let` can only be directly mapped to an anonymous function, `lambda`. We decided to focus first in the transformation of a `named let` to a function.

However, this refactoring operation which transforms a `named let` into a `define` function could have syntax problems since a `let` form can be used in expressions, but the `define` cannot. In the vast majority of cases, this refactoring is correct. However, when a `named let` is used in an expression it transforms the program into an incorrect one. e.g.
Listing 3.8: Let in an expression

```
(and (let xpto ((a 1)) (< a 2)) (< b c))
```

Modifying this named let into a define would raise a syntax error since a define could not be used in an expression context. Encapsulating define with the local keyword, which is an expression like the named let, can solve the problem. However, the local keyword is not used very often and might confuse the users. Therefore, we decided to keep the refactoring operation without the local keyword that works for most of the cases.

Listing 3.9: Let example

```
(let loop ((x 1))
  (when (< x 10)
    (loop (+ x 1))))
```

Listing 3.10: Let to Define Function example

```
(define (loop x)
  (when (< x 10)
    (loop (+ x 1))))
(loop 1)
```

### 3.2.4 Wide-Scope Replacement

The Wide-Scope replacement refactoring operation searches for the code that is the duplicate of the extracted function and then replaces it for the call of the extracted function and it is divided in two steps:

- Detect duplicated code
- Replace the duplicated code

Replacing the duplicated code is the easy part. However, the tool might need to compute the arguments for the duplicated code again.

Correctly detecting duplicated code is a key part for the correctness of this refactoring. Even the simplest form of duplicated code detection, where it only detects duplicated code when the code is exactly equal, may have some problems regarding the binding information. For example, if the duplicated code is inside a let that changes some bindings, that must be taken into consideration. We decided to use AST comparison to detect clones because it was the representation of the program used by our refactoring tool and it has very high precision, but currently has considerably higher costs in terms of execution time [36]. In order to solve the binding problem we can use functions already provided in Racket. However, that does not work if we use the program in the non-expanded form to do the binding comparisons because there is not enough information for those bindings to work. Therefore, in order to compute the correct bindings, it is necessary to use the expanded form of the program.

The naive solution is to use the expanded program to detect the duplicated code and then use this information to do the replacing of the duplicated code. However, when expanding the program, Racket adds necessary internal information to run the program itself that are not visible to the user. While this does not change the detection of the duplicated code, it adds unnecessary information that would have to be removed. In order to solve this, in a simple way, we can use the expanded code to correctly detect duplicated code and use the non-expanded program to compute which code will be replaced. The duplicated code detection is a quadratic algorithm which might have some performance problems for larger programs, however, for the programs typically written by beginners this is not a problem.
3.3 Features

This section describes some of the features created to improve the usability by providing sufficient feedback to the user, and way to inform the user of the presence of the typical mistakes made by beginners.

3.3.1 User Feedback

It is important to give proper feedback to the user while the user is attempting or preforming a refactoring operation. Previewing the outcome of a refactoring operation is an efficient form to help the users understand the result of a refactoring before even applying the refactoring. It works by applying the refactoring operation in a copy of the AST and displaying those changes to the user.

3.3.2 Automatic Suggestions

Beginner programmers usually do not know which refactoring operations exist or which can be applied. By having an automatic suggestions of the possible refactoring operations available, beginner programmers can have an idea what refactoring operations can be applied.

In order to detect possible refactoring operations, it parses the code from the beginning to the end and tries to check if a refactoring is applicable. To do that, it tries to match every syntax expression. In other words, it uses brute force to check whether an expression can be applied a refactoring operation or not.

To properly display this information, it highlights the source-code indicating that there is a possible refactoring. This feature could be improved by having a set of colors for the different types of refactoring operations. Moreover, the color intensity could be proportional to the level of suggestion, e.g. the recommended level to use extract function refactoring increases with the number of duplicated code found.

3.4 Examples

In this section we present examples of some refactoring operations provided by our refactoring tool and some of the implemented features.

3.4.1 Extract-function

The user selects the expressions to extract and chooses a name for the new function as seen in Figure 3.2. Then, the user pastes the result of the extraction where he thinks it is the best place for the function. The final result can be seen in Figure 3.3. The extract function refactoring could automatically paste the extracted function. However, this way, the user can choose the function’s location.

````racket
  (let ([a 7] [d 6])
    (+ a b d))
````
3.4.2 Imported renames

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 3.4, and in this case the user chose “print-cake”. Afterwards the user chooses the name to give and finally it renames all the functions. The end result is shown in Figure 3.5.

3.4.3 Add-prefix

The user selects the library name to add the prefix, in this case “pict3d”. Figure 3.6 shows the arrows pointing to the functions that belong to the library. Then the user selects a name for the prefix name that will be added to each function. The final result can be seen in Figure 3.7.
3.4.4 If to Cond refactoring

If to Cond refactoring operation targets the nested ifs that beginner programmers often use in their programs. This refactoring operation can drastically reduce the complexity of a piece of code. The user selects the code that will be refactored, in this case, the nested if, (Figure 3.8). The final result can be seen in Figure 3.9. As is possible to see, the result is, smaller and simpler piece of code.
3.4.5 Highlight refactoring operations

The user can use the automatic suggestion to detect refactoring operations available in a program.

```
#lang racket
(define (search-aux? board line column piece)
  (cond
    ((> column 8) #f)
    ((= line 1)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    ((= line 2)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    ((= line 3)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    ((= line 4)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    ((= line 5)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    ((= line 6)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    ((= line 7)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    ((= line 8)
     (#t
      (search-aux? board line (+ 2 column) piece)))
    (else null)))
```

Figure 3.9: Nested Ifs to Cond Refactoring outcome

The refactoring operations found by the refactoring tool are then highlighted to inform the user.
3.4.6 Preview

It is important to provide enough feedback to the user and the preview of the outcome of the selected refactoring operation is an important information. Figure 3.12, it is possible to see the outcome of the refactoring of \((\text{not } (\geq a b))\).

```
1 #lang racket
2  
3 (not (>= a b))
4 (not (< a b))
5 (not (<= a b))
6
7 (define l (list))
8 (define (length l) 0)
9 (cons 1 (list 2 3 4 5 6))
```

Figure 3.11: After the highlight

Figure 3.12: Previewing the outcome of a refactoring operation

3.5 Summary

In this chapter, we explained the architecture of the refactoring tool and how it works. We also described some of the refactoring operations available in the refactoring tool and some of the features, such as refactoring suggestions and the preview. In the end, we show some examples of the refactoring operations and features of the refactoring tool.
Chapter 4

Framework

This chapter presents the framework system which provides support for creating refactoring tools for dynamic languages. The provided support consists of a set of utilities used by the refactoring tools that simplify the development.

We start by presenting the framework itself. Then, we present some features of the framework, and the last section presents the languages that have refactoring tools developed using the framework.

4.1 Framework

Refactoring tools often share a similar architecture since they usually require the same information about the program. They often require program information such as the AST and the def-use relations, in order to correctly reason upon the program. The use of the same program’s information creates an architectural similarity between refactoring tools, which raises the possibility of reusing some modules instead of creating every module from scratch. Thus, by reusing those components that provide the program’s information that refactoring tools need we are speeding up the refactoring tool development process. Therefore, a framework for creating refactoring tools increases the development speed and simplifies the development of a refactoring tool. In addition, our framework is able to reuse the features already available for the developed refactoring tools without extra implementation effort. Such features highly improve the utility of a refactoring tool, for example the highlight of possible refactoring operations, the previewing of the result and support to detect duplicated code.

Ideally, in order to achieve maximum re-usability, our framework should only have one implementation of each refactoring operation. This can be accomplished by having a meta-language that represents all the languages supported by the refactoring tool combined with specialized pretty printers to output in the desirable language. However, such general representation would be too complex or, in order to keep it manageable it would reduce the language expressiveness. Instead of having a general representation that can represent all the supported languages and only have one implementation per refactoring, we have language-specific refactoring operations. In other words, we have language-dependent refactoring operations.

Nevertheless, we also have a meta-language that abstracts all the supported languages in order to have language-independent refactoring operations. However, it is only for a small set of refactoring operations, the simpler ones.

With this combination between a meta-language that has language-independent refactoring operations and language-specific refactoring operations we do not have a purely language-independent framework, but instead, we have a framework for creating refactoring tools for dynamic languages.
We decided to implement the framework in DrRacket since it were already some programming languages implemented which simplified development of the framework.

![Figure 4.1: Framework Architecture](image)

As described in Figure 4.1, we have a module that provides language-independent refactoring operations and a module for each supported language that has specialized refactoring operations. We also have pretty-printing modules for each supported language.

We already have refactoring operations for Racket, Python, and Processing. We selected these languages since they are already implemented in Racket and they are commonly used by beginners.

### 4.1.1 Statement-based languages

This framework was initially build for expression-based languages. However, in statement-based languages, such as Python, the program’s flow dependence is needed to decide whether or not a refactoring can be correctly performed or even to decide where and how many returns a function needs. However, this problem could be resolved with a PDG (program dependence graph) that has the control flow information needed to compute correct refactorings.

### 4.1.2 Language-independent refactoring operations

Creating a fully language-independent refactoring tool is rather complex since programming languages are semantically different from each other, and there are even some operations that are semantically equal for most of the cases, but not for all the cases, making it very difficult to do a general refactoring operation. However, for simple refactoring operations, which do not require much program semantics, it is possible to have language-independent refactoring operations in a meta-language that represents the semantics of the other implemented languages.

### 4.1.3 Language-dependent refactoring operations

It is necessary to have refactoring operations that are language-dependent, since some refactoring operations have particular cases for each supported language. By having a specific refactoring operations,
language dependent ones, it is possible to implement some useful refactoring operations in a simple way.

4.1.4 Analysis

Combining the two approaches allows for the simple creation of refactoring operations for several languages targeted at beginner programmers. Some of these refactoring operations are rather simple, when compared to more advanced ones, and can even be reproduced in a meta-language and, therefore, implemented only once for several programming languages. For the rest of the refactoring operations, they can be added to the dependent language module in which it is possible to have the special cases covered.

With the combination usage of reader + expander + code-walker working for the several implemented languages, most of the work is already done and, when combined with the syntax-parser, it is possible to create refactoring operations using only one line of code, e.g. \((p\text{-}\text{not} (p\text{-}\text{gt} a b)) \#\text{'(p\text{-}\text{le} a b)})\).

Even having some refactoring operations that are more complex it is way quicker and simpler to only have to implement the refactoring operation itself.

4.1.5 Program information structures

A refactoring tool needs program information to reason about the program to correctly provide refactoring operations. There are several types of information of the program, for example the AST, def-use-relations, PDG, and type’s information.

4.1.5.1 AST

In order to provide enough information about the program, it is important to provide access to the AST. In the AST there is enough information to do the vast majority of the refactoring operations in an expression-based language. In addition, it also has information for some refactoring operations in statement-based languages.

4.1.5.2 Def-use-relation

The def-use-relation has information about the variable definition and their uses. This information is useful, for example, in order to check whether a variable was defined inside the set of expressions that we are extracting into a function or not. Our framework provides access to this information to simplify the refactoring operations that require such information.

4.1.5.3 PDG

Since our main focus are expression-based languages, like Racket, our framework does not provide access to the PDG information yet. Nevertheless, it is still possible, with some restrictions, to do some refactoring operations in statement-based languages.

4.1.5.4 Type information

We do not explore type information to do refactoring operations in statically typified programing languages. However, there are fragments/subsets of those languages that may be refactored by our framework. Those fragments are parts of the code that can be reason upon without using the type information.
4.1.6 Framework Usage

Since our goal is to provide a way to create refactoring tools, it is necessary to have some care with the way we expose the framework to the refactoring tool developers. Therefore, we only require a single file that needs to have a function that receives two arguments, one being the AST of the program and the other being the def-use-relations. With this information, it is possible to have several useful refactoring operations. We assume that there is already a pretty printer for that programming language that is used to correctly output the refactoring operations.

In addition, we can also provide simple refactoring operations (language-independent ones) for the languages implemented in DrRacket.

This framework makes it easier to implement refactoring operations for dynamic languages, with the requirement that they have to be implemented for DrRacket. Helping minimizing the problem of the difficulty and lacking of refactoring operations for dynamic languages (for at least every language implemented for DrRacket).

4.2 Features

One important part of this framework is to seamlessly provide the features created for a refactoring tool to the other refactoring tools developed within this framework. This is possible since the refactoring tools use the same information represented in the form of syntax-objects which have several information regarding location and position that can be used.

Having the possibility to share the features within the framework makes all the refactoring tools developed within this framework more powerful and better suited to help the user of those refactoring tools to improve the user’s code.

For example, it is possible to use the detection of the detected refactoring operations in the several languages supported by this refactoring tool.

4.2.1 Interoperability

IDEs are always evolving and DrRacket is no exception. Therefore, developing a refactoring tool inside the editor itself is not a good idea. The solution to cope with all the future changes is to create a plugin. This plugin ensures that for the vast majority of modifications in DrRacket our refactoring tool will be able to work properly without any modification. The plugin itself also makes it easier to port the refactoring tool to another editor. Another advantage of the plugin relies on how DrRacket manages the plugins. If the plugin is deployed to a git repository any commits made to that repository will trigger an update to the plugin installation from DrRacket therefore providing the most updated version of the refactoring tool.

4.2.2 Tool maintenance

A refactoring tool as any piece of software requires maintenance. Automatic testing is used in general by developers to ensure that the software is bug free for those test cases. Automatically testing all the refactoring operations is a good way to test if the changes inserted any bugs in the refactoring tool. One way to automatically test the refactoring operations is to apply every refactoring operations possible to a piece of code. By having such possibility it is simpler for the developer to create specific test cases, simplifying tool maintenance. In order to do this, it is necessary to correctly identify all the possible refactoring operations in one piece of code, which is already implemented in the refactoring tool, and apply all of the possible refactoring operations found. Since every refactoring operation changes the
AST, the position of the lines of code and may also change the possible refactoring operations, thus it may be necessary to recompute the program’s AST before the next iteration of the algorithm.

There is also an automatic tool that runs tests in a unit test way. When the refactoring tool starts it reads the test.in file and expand it and computes the refactoring. After that, it compares the output against the test.out file. If one test did not match the expected result, the refactoring tool raises an error alert to the user and writes the error into a log file. If, for some reason, the test.out file did not exist, the tool creates such file with the output computed by the refactoring of the test.in.

4.3 Languages

In this section, we present the languages, besides Racket, that have refactoring tools developed in our framework. We start by presenting Python, then we present Processing.

4.3.1 Python

Python is being promoted as a good replacement for Scheme and Racket in science introductory courses. It is an high-level, dynamically typed programming language and it supports the functional, imperative and object-oriented paradigms. Using the architecture of this framework and the capabilities offered by Racket, combined with an implementation of Python for Racket, called PyonR [37] [38], it is also possible to provide refactoring operations in Python.

Using Racket’s syntax-objects to represent Python as a meta-language [33], it is possible to use the same structure used for the refactoring operations in Racket to parse and analyze the code in Python.

However, there are some limitations regarding the refactoring operations in Python. Since Python is a statement-based language instead of expression base, it raises some problems regarding what the refactoring operation has to do or even the possibility of some refactoring operations. For example, when extracting a function in a expression-based language it is not necessary to compute the location of the return expression, whereas in statement-based languages like Python it is necessary to compute if it is necessary to have the keyword return and where.

The following examples shows some of the refactoring operations we can perform in Python.

Example of removing an If expression:

```python
1 True if (alpha < beta) else False

1 (alpha < beta)
```

The next one shows an extract function:

```python
1 def mandelbrot(iterations, c):
2     z = 0+0j
3     for i in range(iterations+1):
4         if abs(z) > 2:
5             return i
6     z = z*z + c
7     return i+1
```

```python
1 def computeZ(z, c):
2     return z*z + c
```
def mandelbrot(iterations, c):
    z = 0+0j
    for i in range(iterations+1):
        if abs(z) > 2:
            return i
        z = computeZ(z, c)
    return i+1

4.3.2 Processing

Processing [39] is a programming language based on the Java programming language, but with a simplified syntax and graphics programming model.

Even though Processing is a static programming language similar to Java, using P2R [40], an implementation of Processing for Racket, makes it possible to provide some refactoring operations in Processing.

The refactoring operations for Processing are highly limited since they do not take into account the type information of the language present in the program’s AST, therefore, making them less powerful. It is important to note that this framework was developed for creating refactoring tools for dynamic languages. Thus, the refactoring tool for Processing is only a proof of concept to show the flexibility of the created framework. To fully support static languages, it is necessary to gather and manipulate type information.

The following example shows some of the refactoring operations we can perform in Processing.

```python
boolean example = !(alpha < beta);
```

It removes unnecessary Nots of the expression.

```python
boolean example = (alpha >= beta);
```

4.3.3 Meta-Language Refactorings

The meta-language is what allows the framework to provide language-independent refactoring operations. Since we did not want to add additional conditions to the refactoring operations in the meta-language, we only provide some basic refactoring operations. Nevertheless, in addition to these operations being simple, they can be useful as well.

We take advantage of the similarities between the implementations of Python and Processing to Racket syntax-objects and we abstract them to our meta-language. For example, the following code represents !(a>b) in processing:

```python
(p-not (p-gt a b))
```

While the following code represents not(a>b) in python:

```python
(py-not (py-gt a b))
```

With such similarities in the language implementation to Racket syntax-objects, it is straightforward to abstract those languages to a meta-language.
4.3.3.1 Translator

Using the meta-language capabilities, it is even possible to translate code from one language to another. We present an example where we translate the Fibonacci function from Python to Racket and then from Racket to Processing using the meta-language capabilities provided by the framework.

The definition of the Fibonacci function in Python:

```python
def fib(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return fib(n-1) + fib(n-2)
```

The framework translates from the Python programming language to the meta-language that is then pretty-printed to the selected output, in this case the Racket programming language.

```racket
(define (fib n)
    (cond ((eq? n 0) 0)
          ((eq? n 1) 1)
          (else (+ (fib (- n 1)) (fib (- n 2)))))
)
```

The process to transform from Racket to Processing, is the same as stated above, but it has some features hard-coded like the type information since it is not implemented in the framework yet. However, this proof of concept proves that the meta-language can be an interesting tool in a Framework for developing refactoring tools.

```processing
int fib (int n) {
    if( n == 0) return 0;
    if( n == 1) return 1;
    return fib(n - 1) + fib(n -2);
}
```

4.4 Summary

In this chapter, we described the framework and explain how the framework works. We also described the features, such as interoperability and tool maintenance. In the end, we show the languages that have refactoring tools developed in the framework besides Racket, namely Python, Processing, and the meta-language.
Chapter 5

Evaluation

In this chapter we present the evaluation of the refactoring tool and then the evaluation of the framework.

5.1 Refactoring Tool Evaluation

Our main goal is to provide refactoring operations designed for beginner programmers. Therefore, we used the submissions of the final project of an introductory programming course. In this section, we present some code examples of the final project and their possible improvements using the refactoring operations available in our refactoring tool. The examples show the use of some of the refactoring operations previously presented and here is explained the motivation for their existence. Lastly, we present the results of the refactoring operations applied to the final project submissions.

5.1.1 Beginner’s Examples

The first example is a very typical error made beginner programmers.

```lisp
(if (>= n_plays 35)
  #t
  #f)
```

It is rather a simple refactoring operation, but nevertheless it improves the code.

```lisp
(>= n_plays 35)
```

The next example is related with the conditional expressions, namely the and or or expressions. We decided to choose the and expression to exemplify a rather typical usage of this expression.

```lisp
(and
  (and
    (eq? #t (correct-moviment? player play))
    (eq? #t (player-piece? player play)))
  (and
    (eq? #t (empty-destination? play))
    (eq? #t (empty-start? play))))
```

Transforming the code by removing the redundant and expression makes the code cleaner and simpler to understand.
(and (eq? #t (correct-moviment? player play))
  (eq? #t (player-piece? player play))
  (eq? #t (empty-destination? play))
  (eq? #t (empty-start? play)))

However, this code can still be improved, the (eq? #t ?x) is a redundant way of simple writing ?x.

(and (correct-moviment? player play)
  (player-piece? player play)
  (empty-destination? play)
  (empty-start? play))

While a student is in the initial learning phases, it is common to forget whether or not a sequence of expressions need to be wrapped in a begin form. The when, cond and let expressions have a implicit begin and as a result it is not necessary to add the begin expression. Nevertheless, sometimes students still keep the begin keyword because they often use a trial and error approach in writing code. Our refactoring tool checks for those mistakes and corrects them.

(if (odd? line-value)
  (let ((internal-column (sub1 (/ column 2))))
    (begin
      (if (integer? internal-column)
        internal-column
        #f))
  (let ((internal-column (/ (sub1 column) 2))))
    (begin
      (if (integer? internal-column)
        internal-column
        #f))))

This is a simple refactoring operation and does not have a big impact. However, it makes the code clearer and helps the beginner programmer to learn that a let does not need a begin.

(if (odd? line-value)
  (let ((internal-column (sub1 (/ column 2))))
    (and (integer? internal-column)
      internal-column)
  (let ((internal-column (/ (sub1 column) 2))))
    (and (integer? internal-column)
      internal-column)))

The next example shows a nested if. Nested ifs are difficult to understand and error prone.
(define (search-aux? board line column piece)
  (if (> column 8)
      #f
      (if (= line 1)
          (if (eq? (house-board board 1 column) piece)
              #t
              (search-aux? board line (+ 2 column) piece))
          (if (= line 2)
              (if (eq? (house-board board 2 column) piece)
                  #t
                  (search-aux? board line (+ 2 column) piece))
              (if (= line 3)
                  (if (eq? (house-board board 3 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
              (if (= line 4)
                  (if (eq? (house-board board 4 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
              (if (= line 5)
                  (if (eq? (house-board board 5 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
              (if (= line 6)
                  (if (eq? (house-board board 6 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
              (if (= line 7)
                  (if (eq? (house-board board 7 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
              (if (= line 8)
                  (if (eq? (house-board board 8 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
              null))))))))

It is much simpler to have a cond expression instead of the nested if. In addition, every true branch of this nested if contains if expressions that are or expressions and by refactoring those if expressions to ors it makes the code simpler to understand.
(define (search-aux? board line column piece)
  (cond
   [(> column 8) #f]
   [(= line 1)
    (or (eq? (house-board board 1 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [(= line 2)
    (or (eq? (house-board board 2 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [(= line 3)
    (or (eq? (house-board board 3 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [(= line 4)
    (or (eq? (house-board board 4 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [(= line 5)
    (or (eq? (house-board board 5 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [(= line 6)
    (or (eq? (house-board board 6 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [(= line 7)
    (or (eq? (house-board board 7 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [(= line 8)
    (or (eq? (house-board board 8 column) piece)
     (search-aux? board line (+ 2 column) piece))]
   [else null]))

However, this code could still be further improved by refactoring it into a case.

The examples presented above appear repeatedly in almost every code submission of this final project supports the need to provide a better support to beginner programmers.

5.1.2 Results

Table 5.1 shows he average reduction in lines of code (LOC) is 10.63%, which shows how useful these refactoring operations are. It also shows how many refactoring operations were applied. This tool is not only for beginners: during the development of the tool we already used some of the refactoring operations, namely the extract function, in order to improve the structure of the code.

5.2 Framework Evaluation

In this section, we evaluate the framework by evaluating the simplicity of creating a refactoring tool. The re-usability of a feature specifically developed for a refactoring tool, created using this framework, to be used by any refactoring tool under this framework. Finally the tools provided to help the developer maintain the refactoring tools developed using this framework.
Table 5.1: Refactoring Operations

<table>
<thead>
<tr>
<th>Code #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial LOC</td>
<td>582</td>
<td>424</td>
<td>332</td>
<td>1328</td>
<td>810</td>
<td>569</td>
<td>798</td>
<td>614</td>
<td>4045</td>
</tr>
<tr>
<td>Final LOC</td>
<td>545</td>
<td>373</td>
<td>300</td>
<td>1259</td>
<td>701</td>
<td>527</td>
<td>733</td>
<td>457</td>
<td>3705</td>
</tr>
<tr>
<td>Difference</td>
<td>37</td>
<td>51</td>
<td>32</td>
<td>69</td>
<td>109</td>
<td>42</td>
<td>65</td>
<td>140</td>
<td>340</td>
</tr>
</tbody>
</table>

| Remove Begin | 11  | 4   | 6   | 9   | 5   | 2   | 0   | 7   | 44    |
| If to When   | 4   | 0   | 0   | 0   | 0   | 7   | 0   | 0   | 11    |
| If to And    | 3   | 1   | 0   | 0   | 0   | 2   | 0   | 0   | 6     |
| If to Or     | 6   | 6   | 1   | 13  | 20  | 3   | 2   | 0   | 51    |
| Remove If    | 0   | 3   | 7   | 6   | 3   | 5   | 0   | 2   | 26    |
| Remove And   | 0   | 2   | 0   | 4   | 0   | 0   | 0   | 0   | 6     |
| Remove Eq    | 0   | 4   | 0   | 0   | 0   | 0   | 0   | 0   | 4     |
| Extract function | 0 | 3 | 0 | 0 | 0 | 4 | 0 | 1 | 5 | 13 |
| If to Cond   | 0   | 0   | 0   | 2   | 3   | 0   | 1   | 0   | 6     |

5.2.1 Simplicity

The following piece of code is the function that implements the refactoring operations available for Processing.

```scheme
(define (processing-parser arg)
  (syntax-parse arg
    #:datum-literals (p-not p-and p-or p-lt p-gt p-le p-ge)
    [(p-not (p-gt a b)) #'(p-le a b)]
    [(p-not (p-le a b)) #'(p-gt a b)]
    [(p-not (p-lt a b)) #'(p-ge a b)]
    [(p-not (p-ge a b)) #'(p-lt a b)]
    [_ (void)])
)
```

It is quite simple to create refactoring operations in the framework. With simple rules, like the ones presented above, it is possible to create simple, but useful refactoring operations.

5.2.2 Reusability

We can share features between the implemented refactoring tools in the framework, such as the automatic suggestion of refactoring operations and the preview of the outcome of such refactorings, already created for the Racket refactoring tool. In this section, we show and explain what was needed to improve in order to be able to simple re-use the features initially developed for the Racket refactoring tool, such as the Refactoring Suggestions (section 5.2.2.1) and the Preview (section 5.2.2.2).

5.2.2.1 Refactoring Suggestions

Figure 5.1 shows, the automatic suggesting of the detected refactoring opportunities working for the Python refactoring tool. The main difference for the original is that the highlighting now has to use the pretty printing to compute the size of the region where the refactoring opportunities occurs. This difference, a cosmetic one, happens because the syntax-objects for the Racket refactoring tool have the
same length to be highlighted as the program’s code, whereas the Python’s representation syntax-objects do not. Figure 5.1 highlights the refactoring operation that removes unnecessary if conditions.

```
#lang python
True if (a < b) else False
```

Figure 5.1: Refactoring Suggestions in Python

Figure 5.2 shows the automatic suggesting of the detected refactoring opportunities works for the Processing refactoring tool. Like for the Python refactoring tool, it was necessary to require the pretty printing to correctly highlight the region of the refactoring opportunity. However, since it was previously improved for the Python refactoring tool, the re-usability of the Highlight feature was straightforward for Processing refactoring tool. Figure 5.2 highlights the refactoring operation that removes unnecessary not conditions.

```
#lang processing

boolean example = !(a < b);
```

Figure 5.2: Refactoring Suggestions in Processing

5.2.2.2 Preview

Figure 5.3 shows, the preview of the outcome of the refactoring operation works for the Python refactoring tool. The main difference for the original is that the preview now, like the highlight feature, has to use the pretty printing to compute the size of the region where the refactoring opportunities occurs. Figure 5.3 shows the preview of the outcome of the refactoring operation that removes unnecessary if conditions.

```
#lang python
True if (alpha < beta) else False
```

Figure 5.3: Preview in Python
Figure 5.4 shows, the preview of the outcome of the refactoring operation works for the Processing refactoring tool. Figure 5.4 shows the preview of the outcome of the refactoring operation that removes unnecessary not conditions.

```processing
1 #lang processing
2
3 boolean example = !(a < b);

Figure 5.4: Preview in Processing
```

### 5.2.3 Maintainability

The provided automatic testing tool can help the developer maintaining the developed refactoring tools. The tests run at the startup of the IDE, providing feedback to the developer check the correct running of the refactoring tool and the test case that is failing.

Figure 5.5 shows an error in the Racket Refactoring Tool. The error is from the test case testNot.

![Error in Racket Refactoring Tool](image)

Figure 5.5: Error in Racket Refactoring Tool

Figure 5.6 shows an error in the Python Refactoring Tool. The error is from the test case removeIf.

![Error in Python Refactoring Tool](image)

Figure 5.6: Error in Python Refactoring Tool

Figure 5.7 shows an error in the Processing Refactoring Tool. The error is from the test case RemoveNot.

![Error in Processing Refactoring Tool](image)

Figure 5.7: Error in Processing Refactoring Tool
Since these tests run each time DrRacket starts up they are run frequently, the bugs are detected frequently improving the maintainability of the refactoring tools.

5.3 Conclusion

In this chapter, we presented the evaluation of the refactoring tool and the framework. Regarding the refactoring tool, we presented the impact of the developed refactoring operations in the programs written by students in their final project of a programming introductory course. Their programs become smaller, in average, 10% and the refactoring operations applied removed several typical mistakes made by beginner programmers improving the quality of the programs. Regarding the framework, we presented the evaluation of the initial goals, the simplicity of creating refactoring operations, the re-usability achieved of the features already provided, and the help provided for the maintenance of the refactoring tools created within the framework.
Chapter 6

Conclusion and future work

6.1 Conclusion

A refactoring tool designed for beginner programmers would benefit them by providing a tool to restructure the programs and improve what more knowledgeable programmers call “poor style” or “bad smells”. In order to help those users the refactoring tool must be usable from a pedagogical IDE, to inform the programmer of the presence of the typical mistakes made by beginners, and to correctly apply refactoring operations preserving semantics.

Our solution tries to help those users to improve their programs by using the AST of the program and the def-use-relations to create refactoring operations that do not change the program’s semantics. This structure is then used to analyze the code to detect typical mistakes using automatic suggestions and correct them using the refactoring operations provided.

We proposed some goals which we address here:

• Designed for Beginners - the refactoring tool currently provides a set of refactoring operations designed for beginner programmers, which allow them to correct typical errors. We have implemented refactoring operations that focus on improving their code, such as if-to-cond, simplifying conditions (removing unnecessary ifs, whens, nots, etc), and extract function. This set of refactoring operations could be further expanded, but it already can make the difference in improving the beginner programmer code.

• Simple Feedback - the refactoring tool presently provides two forms of feedback to the user, namely previewing and the suggestion of possible refactoring operations. Previewing the outcome of the refactoring operation currently consists of displaying the result transformation of the selected code. It is not the best approach and it can be further improved by highlighting the changes, but nevertheless it already gives enough feedback to help the user decide whether or not to perform the refactoring operation. The automatic suggestion highlights all the possible refactoring operations found by the refactoring tool. Identifying the possible refactorings and providing feedback to the user it is an important feature to help the user learn what can be further improved and what a better code looks like.

• Correctness - it is crucial for a refactoring tool to preserve the meaning of the program, therefore we made some compromises regarding which refactoring operations to allow, namely, the refactoring operations were tested and corrected several times and there are no known bugs.

A framework for creating refactoring tools for dynamic languages would benefit the refactoring tool developers by simplifying their work and would indirectly benefit its users since the developer would
have more time to spend improving the refactoring tool. In order to be helpful, the framework must be simple to use by the developers, must facilitate the reuse of the already implemented features, and must provide maintainability support to the refactoring tools.

Our framework tries to help developers to create refactoring tools. We achieve that by providing modules to access program information such as the AST or the def-use-relations. Combined with the provided tools to expand and process the AST it drastically simplifies the refactoring tool creation. We also created a prototype architecture to have simple language-independent refactoring operations in the framework. These simple refactoring operations can be helpful for the user for those languages that the user do not has refactoring tools and wants to continue use the same IDE.

We had proposed some goals which we address here:

- **Simple** - our framework was designed to be simple to use so the developer would only focus on creating the refactoring operation itself. We provide access to core aspects of the refactoring tool, such as the AST, the code-walker, and a printing function. As previously shown in section 5.2.1 it is possible to create refactoring operations with only one or two simple lines of code, therefore freeing the developer to develop more complex refactoring operations.

- **Re-usability** - we designed our framework to provide the possibility to re-use the features previously developed for a refactoring tool created within this framework. As previously shown in section 5.2.2 we re-used the Previewing and the Automatic suggesting features for both the Processing and Python refactoring tools. Those features were initially developed for the Racket refactoring tool and then integrated for the newly created refactoring tools. Re-usability can be further improved by providing automatic detection of new refactoring tools. At this moment, it is necessary to explicitly define which is the language and the refactoring tool files associated with that language. With an automatic detection of the refactoring tool files, which works if the developer follows the naming convention for the refactoring tool functions, the re-usability would be fully automatic.

- **Maintainability** - we designed a tool that automatically applies tests to the refactoring available tools helping the user to maintain the complex program that is a refactoring tool. Our automatic testing tool runs every time the refactoring tool starts, comparing the result of the refactoring operation in the input file with the output one, alerting the user only if the comparison failed. This support is crucial to free the user of manually testing the refactoring tool or to build a script to automate the testing itself.

### 6.2 Future Work

There are still some improvements that we consider important for the user of the refactoring tool, regarding feedback information and support optimization. Firstly, the detection of duplicated code is still very naive and improving the clone detection to understand if a variable is the same, in the cases that the names are different or even if the order of some commutative expressions is not the same would make a huge improvement on the automatic suggestion. Thus supporting at least the detection of type 2 clones (section 2.1.6), which are a syntactically identical copy where variables, type or function identifiers may differ.

Then, it is possible to further improve the automatic suggestion of refactoring operations by having different colors for different types of refactoring operations, with a low intensity for low “priority” refactoring operations and a higher intensity for higher “priority”, thus giving the user a better knowledge of what is a better way to solve a problem or what is a strongly recommendation to change the code.
For example, if a piece of code has several clones, it would have a higher color intensity than a piece of code that has only one clone.

The previewing of the outcome of the refactoring operation is still very naive and could be further improved by having a comparison of the code before refactoring with the code after refactoring instead of the current one that only shows how the code will look like.

The def-use-relations are used in very few cases and it should be used to check more preconditions, thus making the refactoring tool more robust than it is.

It would also be interesting to have support for refactoring operations in the read-eval-print-loop (repl), allowing the user after testing their code to refactor it before copying it to the definitions area.

Finally, it would be interesting to finish the refactoring with the tool support [41], thus raising awareness to the user of the existing refactoring operations supported by the refactoring tool.

Regarding the framework, it would be improved with a better program’s information support, improved feedback, and a better decoupling in the framework architecture.

Firstly, having PDG support will allow the framework user to develop more useful and complex refactoring operations for statement-based languages such as Python. In addition, it would also be interesting to add type information support to the framework since Racket already has a typed Racket [35] which would allow a refactoring tool for typed Racket.

A partial parser would support refactoring operations in a program that is not correct but that has pieces of code that are necessarily correct to be applied the refactoring operation. A partial parser would allow the refactoring tool developer to create refactoring operations that can help the user to change from one representation to another without having to finish the implementation of the first representation. e.g. It would help the user that is in the middle of a nested if in Racket to change to a cond clause without having to finish the nested if.

Expanding feedback support beyond the already provided, the preview, it would be very useful especially for statement-based languages. Since those languages have more constraints regarding the refactoring operations, it is necessary to check if it is possible to do the refactoring operation safely or not. Thus, by informing the user about the problems/constraints that do not allow the refactoring operation would be useful in order to help the user change that and then do the refactoring operation safely.

Finally, the addition of new features could be simplified by a better decoupling with the framework itself instead of belonging to a core part of the refactoring tool.

6.3 Publications

Some of the contributions presented in this thesis have also been published in one paper:

- Reia, Rafael and Leitão, António Menezes. “Refactoring Dynamic Languages” in 9th European Lisp Symposium, May 2016

6.4 Availability

The source-code for the refactoring tool and the framework is available on GitHub at: https://github.com/RafaelReia/RefactoringToolDevelop.
Bibliography


Appendix A

Refactoring Operations

- Remove Begin: Removes redundant `begin` expression either in `when`, `let`, `cond`, and `define` expressions.
- Remove If: Removes redundant `if` expression
- Remove And: Removes redundant `and` expression
- Remove Eq: Removes redundant `eq` expression
- Remove Not: Removes unnecessary `not` expression
- Eta Abstraction: Adds a lambda abstraction over a function
- Merge List: Merges `cons` with `list`
- If to When: Transforms an `if` expression into a `when` expression
- If to And: Transforms an `if` expression into a `and` expression
- If to Or: Transforms an `if` expression into a `or` expression
- If to Cond: Transforms an `if` expression into a `cond` expression
- Cond to If: Transforms an `cond` expression into a `if` expression
- Let to Define: Transforms an `let` expression into a `define` expression
- Extract Function: Extracts a function from the selected text
- Add Prefix: Adds a prefix for every expression imported from the selected import.
- Imported Renames: correctly renames imported functions